GNU Emacs Lisp Reference Manual

For Emacs Version 27.1

**by Bil Lewis, Dan LaLiberte, Richard Stallman,
the GNU Manual Group, et al.**

This is the *GNU Emacs Lisp Reference Manual* corresponding to Emacs version 27.1.

Copyright © 1990–1996, 1998–2020 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "GNU General Public License," with the Front-Cover Texts being "A GNU Manual," and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled "GNU Free Documentation License."

(a) The FSF's Back-Cover Text is: "You have the freedom to copy and modify this GNU manual. Buying copies from the FSF supports it in developing GNU and promoting software freedom."

Published by the Free Software Foundation
51 Franklin St, Fifth Floor
Boston, MA 02110-1301
USA
ISBN 1-882114-74-4

Cover art by Etienne Suvasa.

Short Contents

1 イントロダクション 1
2 Lisp のデータ型 8
3 数値 37
4 文字列と文字 51
5 リスト 71
6 シーケンス、配列、ベクター 95
7 レコード 116
8 ハッシュテーブル 118
9 シンボル 124
10 評価 133
11 制御構造 145
12 変数 174
13 関数 209
14 マクロ 242
15 カスタマイゼーション設定 250
16 ロード 269
17 バイトコンパイル 285
18 Lisp プログラムのデバッグ 296
19 Lisp オブジェクトの読み取りとプリント 331
20 ミニバッファー 342
21 コマンドループ 375
22 キーマップ 422
23 メジャーモードとマイナーモード 461
24 ドキュメント 519
25 ファイル 530
26 バックアップと自動保存 578
27 バッファー 590
28 ウィンドウ 608
29 フレーム 696
30 ポジション 755
31 マーカー 768
32 テキスト 777
33 非 ASCII文字 852

34 検索とマッチング 881
35 構文テーブル 906
36 abbrev と abbrev 展開 922
37 スレッド 929
38 プロセス 934
39 Emacs のディスプレー表示 981
40 オペレーティングシステムのインターフェース 1090
41 配布用 Lisp コードの準備 1134
A Emacs 26 のアンチニュース 1140
B GNU Free Documentation License 1143
C GNU General Public License 1151
D ヒントと規約 1162
E GNU Emacs の内部 1175
F 標準的なエラー 1216
G 標準的なキーマップ 1220
H 標準的なフック 1223
Index 1227

Table of Contents

1 イントロダクション 1
1.1 注意事項 1
1.2 Lisp の歴史 1
1.3 表記について 2
1.3.1 用語について 2
1.3.2 `nil`と `t` 2
1.3.3 評価の表記 3
1.3.4 プリントの表記 3
1.3.5 エラーメッセージ 3
1.3.6 バッファーテキストの表記 4
1.3.7 説明のフォーマット 4
1.3.7.1 関数の説明例 4
1.3.7.2 変数の説明例 5
1.4 バージョンの情報 6
1.5 謝辞 7

2 Lisp のデータ型 8
2.1 プリント表現と読み取り構文 8
2.2 特別な読み取り構文 9
2.3 コメント 10
2.4 プログラミングの型 10
2.4.1 整数型 10
2.4.2 浮動小数点数型 11
2.4.3 文字型 11
2.4.3.1 基本的な文字構文 11
2.4.3.2 一般的なエスケープ構文 12
2.4.3.3 コントロール文字構文 13
2.4.3.4 メタ文字構文 13
2.4.3.5 その他の文字修飾ビット 13
2.4.4 シンボル型 14
2.4.5 シーケンス型 15
2.4.6 コンスセルとリスト型 15
2.4.6.1 ボックスダイアグラムによるリストの描写 16
2.4.6.2 ドットペア表記 17
2.4.6.3 連想リスト型 18
2.4.7 配列型 19
2.4.8 文字列型 19
2.4.8.1 文字列の構文 19
2.4.8.2 文字列内の非 ASCII文字 20
2.4.8.3 文字列内の非プリント文字 20
2.4.8.4 文字列内のテキストプロパティ 21
2.4.9 ベクター型 21
2.4.10 文字テーブル型 21

2.4.11 ブールベクター型 22
2.4.12 ハッシュテーブル型 22
2.4.13 関数型 22
2.4.14 マクロ型 23
2.4.15 プリミティブ関数型 23
2.4.16 バイトコード関数型 23
2.4.17 レコード型 24
2.4.18 型記述子 24
2.4.19 autoload 型 24
2.4.20 ファイナライザー型 24
2.5 編集用の型 25
2.5.1 バッファー型 25
2.5.2 マーカー型 25
2.5.3 ウィンドウ型 26
2.5.4 フレーム型 26
2.5.5 端末型 26
2.5.6 ウィンドウ構成型 27
2.5.7 フレーム構成型 27
2.5.8 プロセス型 27
2.5.9 スレッド型 27
2.5.10 ミューテックス型 27
2.5.11 状態変数型 28
2.5.12 ストリーム型 28
2.5.13 キーマップ型 28
2.5.14 オーバーレイ型 28
2.5.15 フォント型 29
2.6 循環オブジェクトの読み取り構文 29
2.7 型のための述語 29
2.8 同等性のための述語 33
2.9 可変性 35

3 数値 37

3.1 整数の基礎 37
3.2 浮動小数点数の基礎 38
3.3 数値のための述語 40
3.4 数値の比較 40
3.5 数値の変換 42
3.6 算術演算 43
3.7 丸め処理 46
3.8 整数にたいするビット演算 46
3.9 標準的な数学関数 49
3.10 乱数 50

4 文字列と文字 51
4.1 文字列と文字の基礎 51
4.2 文字列のための述語 52
4.3 文字列の作成 52
4.4 文字列の変更 55
4.5 文字および文字列の比較 56
4.6 文字および文字列の変換 60
4.7 文字列のフォーマット 61
4.8 カスタムフォーマット文字列 65
4.9 Lisp での大文字小文字変換 67
4.10 case テーブル 69

5 リスト 71
5.1 リストとコンスセル 71
5.2 リストのための述語 72
5.3 リスト要素へのアクセス 73
5.4 コンスセルおよびリストの構築 75
5.5 リスト変数の変更 79
5.6 既存のリスト構造の変更 81
5.6.1 `setcar`によるリスト要素の変更 81
5.6.2 リストの CDR の変更 83
5.6.3 リストを再配置する関数 84
5.7 集合としてのリストの使用 85
5.8 連想リスト 88
5.9 プロパティリスト 92
5.9.1 プロパティリストと連想リスト 93
5.9.2 プロパティリストと外部シンボル 93

6 シーケンス、配列、ベクター 95
6.1 シーケンス 95
6.2 配列 107
6.3 配列を操作する関数 107
6.4 ベクター 109
6.5 ベクターのための関数 109
6.6 文字テーブル 110
6.7 ブールベクター 113
6.8 オブジェクト用固定長リングの管理 114

7 レコード 116
7.1 レコード関数 116
7.2 後方互換 117

8 ハッシュテーブル 118
8.1 ハッシュテーブルの作成 118
8.2 ハッシュテーブルへのアクセス 120
8.3 ハッシュの比較の定義 121
8.4 ハッシュテーブルのためのその他関数 122

9 シンボル 124
9.1 シンボルの構成要素 124
9.2 シンボルの定義 125
9.3 シンボルの作成と intern 126
9.4 シンボルのプロパティ 129
9.4.1 シンボルのプロパティへのアクセス 129
9.4.2 シンボルの標準的なプロパティ 130

10 評価 133
10.1 フォームの種類 134
10.1.1 自己評価を行うフォーム 134
10.1.2 シンボルのフォーム 134
10.1.3 リストフォームの分類 135
10.1.4 シンボル関数インダイレクション 135
10.1.5 関数フォームの評価 136
10.1.6 Lisp マクロの評価 137
10.1.7 スペシャルフォーム 137
10.1.8 自動ロード 138
10.2 クォート 139
10.3 バッククォート 139
10.4 eval について 140
10.5 遅延された Lazy 評価 143

11 制御構造 145
11.1 順序 145
11.2 条件 146
11.3 組み合わせ条件の構築 148
11.4 パターンマッチングによる条件 149
11.4.1 `pcase`マクロ 150
11.4.2 `pcase`の拡張 156
11.4.3 バッククォートスタイルパターン 157
11.4.4 `pcase`パターンによる分解 158
11.5 繰り返し 159
11.6 ジェネレーター 160
11.7 非ローカル脱出 162
11.7.1 明示的な非ローカル脱出: `catch`と `throw` 162
11.7.2 `catch`と `throw`の例 164
11.7.3 エラー 164
11.7.3.1 エラーをシグナルする方法 165
11.7.3.2 Emacs がエラーを処理する方法 166
11.7.3.3 エラーを処理するコードの記述 167
11.7.3.4 エラーシンボルとエラー条件 171
11.7.4 非ローカル脱出のクリーンアップ 172

12 変数 174
12.1 グローバル変数 174
12.2 変更不可な変数 174
12.3 ローカル変数 175
12.4 変数が void のとき 177
12.5 グローバル変数の定義 178
12.6 堅牢な変数定義のためのヒント 180
12.7 変数の値へのアクセス 182
12.8 変数の値のセット 182
12.9 変数が変更されたときに実行される関数。 184
12.9.1 制限 184
12.10 変数のバインディングのスコーピングルール 185
12.10.1 ダイナミックバインディング 185
12.10.2 ダイナミックバインディングの正しい使用 186
12.10.3 レキシカルバインディング 187
12.10.4 レキシカルバインディングの使用 189
12.11 バッファーローカル変数 190
12.11.1 バッファーローカル変数の概要 190
12.11.2 バッファーローカルなバインディングの作成と削除 192
12.11.3 バッファーローカル変数のデフォルト値 195
12.12 ファイルローカル変数 197
12.13 ディレクトリーローカル変数 200
12.14 接続ローカル変数 201
12.15 変数のエイリアス 204
12.16 値を制限された変数 205
12.17 ジェネリック変数 205
12.17.1 `setf`マクロ 206
12.17.2 新たな `setf`フォーム 207

13 関数 209
13.1 関数とは? 209
13.2 ラムダ式 211
13.2.1 ラムダ式の構成要素 211
13.2.2 単純なラムダ式の例 212
13.2.3 引数リストのその他機能 212
13.2.4 関数のドキュメント文字列 213
13.3 関数の命名 214
13.4 関数の定義 215
13.5 関数の呼び出し 217
13.6 関数のマッピング 219
13.7 無名関数 220
13.8 ジェネリック関数 222
13.9 関数セルの内容へのアクセス 225
13.10 クロージャー 227
13.11 Emacs Lisp 関数にたいするアドバイス 227
13.11.1 アドバイスを操作するためのプリミティブ 228
13.11.2 名前つき関数にたいするアドバイス 230
13.11.3 アドバイスの構築方法 231

13.11.4 古い defadvice を使用するコードの改良 233
13.12 関数の陳腐化の宣言 234
13.13 インライン関数 Inli 235
13.14 declareフォーム 237
13.15 コンパイラーへの定義済み関数の指示 238
13.16 安全に関数を呼び出せるかどうかの判断 239
13.17 関数に関するその他トピック 240

14 マクロ 242
14.1 単純なマクロの例 242
14.2 マクロ呼び出しの展開 242
14.3 マクロとバイトコンパイル 244
14.4 マクロの定義 244
14.5 マクロ使用に関する一般的な問題 245
14.5.1 タイミング間違い 245
14.5.2 マクロ引数の多重評価 245
14.5.3 マクロ展開でのローカル変数 247
14.5.4 展開におけるマクロ引数の評価 247
14.5.5 マクロが展開される回数は? 248
14.6 マクロのインデント 249

15 カスタマイゼーション設定 250
15.1 一般的なキーワードアイテム 250
15.2 カスタマイゼーショングループの定義 252
15.3 カスタマイゼーション変数の定義 253
15.4 カスタマイゼーション型 257
15.4.1 単純型 257
15.4.2 複合型 258
15.4.3 リストへのスプライス 263
15.4.4 型キーワード 263
15.4.5 新たな型の定義 265
15.5 カスタマイゼーションの適用 266
15.6 Custom テーマ 267

16 ロード 269
16.1 プログラムがロードを行う方法 269
16.2 ロードでの拡張子 271
16.3 ライブラリー検索 272
16.4 非 ASCII文字のロード 274
16.5 autoload 274
16.5.1 プレフィックスによる autoload 278
16.5.2 autoload を使用するケース 278
16.6 多重ロード 278
16.7 名前つき機能 279
16.8 どのファイルで特定のシンボルが定義されているか 281
16.9 アンロード 282
16.10 ロードのためのフック 283
16.11 Emacs のダイナミックモジュール 284

17 バイトコンパイル 285
17.1 バイトコンパイル済みコードのパフォーマンス 285
17.2 バイトコンパイル関数 285
17.3 ドキュメント文字列とコンパイル 288
17.4 個々の関数のダイナミックロード 288
17.5 コンパイル中の評価 289
17.6 コンパイラーのエラー 290
17.7 バイトコード関数オブジェクト 291
17.8 逆アセンブルされたバイトコード 292

18 Lisp プログラムのデバッグ 296
18.1 Lisp デバッガ 296
18.1.1 エラーによるデバッガへのエンター 296
18.1.2 無限ループのデバッグ 298
18.1.3 関数呼び出しによるデバッガへのエンター 298
18.1.4 変数の変更時にデバッガにエンターする。..... 299
18.1.5 明示的なデバッガへのエントリー 300
18.1.6 デバッガの使用 300
18.1.7 バックトレース 301
18.1.8 デバッガのコマンド 301
18.1.9 デバッガの呼び出し 303
18.1.10 デバッガの内部 304
18.2 Edebug 306
18.2.1 Edebug の使用 306
18.2.2 Edebug のためのインストルメント 307
18.2.3 Edebug の実行モード 308
18.2.4 ジャンプ 310
18.2.5 その他の Edebug コマンド 311
18.2.6 ブレーク 311
18.2.6.1 Edebug のブレークポイント 312
18.2.6.2 グローバルなブレーク条件 313
18.2.6.3 ソースブレークポイント 313
18.2.7 エラーのトラップ 313
18.2.8 Edebug のビュー 314
18.2.9 評価 314
18.2.10 評価 List Buffer 315
18.2.11 Edebug でのプリント 316
18.2.12 トレースバッファー 317
18.2.13 カバレッジテスト 317
18.2.14 コンテキスト外部 318
18.2.14.1 停止するかどうかのチェック 318
18.2.14.2 Edebug の表示の更新 319
18.2.14.3 Edebug の再帰編集 319
18.2.15 Edebug とマクロ 320
18.2.15.1 マクロ呼び出しのインストルメント 320
18.2.15.2 仕様リスト 321
18.2.15.3 仕様でのバックトレース 323
18.2.15.4 仕様の例 324

18.2.16 Edebug のオプション 325
18.3 無効な Lisp 構文のデバッグ 328
18.3.1 過剰な開カッコ 328
18.3.2 過剰な閉カッコ 328
18.4 カバレッジテスト 329
18.5 プロファイリング 329

19 Lisp オブジェクトの読み取りとプリント 331
19.1 読み取りとプリントの概念 331
19.2 入力ストリーム 331
19.3 入力関数 334
19.4 出力ストリーム 335
19.5 出力関数 337
19.6 出力に影響する変数 339

20 ミニバッファー 342
20.1 ミニバッファーの概要 342
20.2 ミニバッファーでのテキスト文字列の読み取り 343
20.3 ミニバッファーでの Lisp オブジェクトの読み取り 347
20.4 ミニバッファーのヒストリー 348
20.5 入力の初期値 350
20.6 補完 350
20.6.1 基本的な補完関数 350
20.6.2 補完とミニバッファー 353
20.6.3 補完を行うミニバッファーコマンド 355
20.6.4 高レベルの補完関数 357
20.6.5 ファイル名の読み取り 359
20.6.6 補完変数 362
20.6.7 プログラムされた補完 363
20.6.8 通常バッファーでの補完 365
20.7 Yes-or-No による問い合わせ 366
20.8 複数の問いを尋ねる 368
20.9 パスワードの読み取り 369
20.10 ミニバッファーのコマンド 370
20.11 ミニバッファーのウィンドウ 371
20.12 ミニバッファーのコンテンツ 372
20.13 再帰的なミニバッファー 373
20.14 ミニバッファー、その他の事項 373

21 コマンドループ 375
21.1 コマンドループの概要 375
21.2 コマンドの定義 376
21.2.1 `interactive`の使用 376
21.2.2 `interactive`にたいするコード文字 378
21.2.3 `interactive`の使用例 381
21.2.4 コマンド候補からの選択 382
21.3 インタラクティブな呼び出し 382

21.4 インタラクティブな呼び出しの区別 384
21.5 コマンドループからの情報 385
21.6 コマンド後のポイントの調整 387
21.7 入力イベント 387
21.7.1 キーボードイベント 388
21.7.2 ファンクションキー 389
21.7.3 マウスイベント 390
21.7.4 クリックイベント 390
21.7.5 ドラッグイベント 392
21.7.6 ボタンダウンイベント 393
21.7.7 リピートイベント 393
21.7.8 モーションイベント 395
21.7.9 フォーカスイベント 395
21.7.10 その他のシステムイベント 395
21.7.11 イベントの例 397
21.7.12 イベントの分類 398
21.7.13 マウスイベントへのアクセス 400
21.7.14 スクロールバーイベントへのアクセス 402
21.7.15 文字列内へのキーボードイベントの配置 402
21.8 入力の読み取り 403
21.8.1 キーシーケンス入力 404
21.8.2 単一イベントの読み取り 405
21.8.3 入力イベントの変更と変換 408
21.8.4 入力メソッドの呼び出し 409
21.8.5 クォートされた文字の入力 410
21.8.6 その他のイベント入力の機能 410
21.9 スペシャルイベント 412
21.10 時間の経過や入力の待機 412
21.11 quit 413
21.12 プレフィクスコマンド引数 415
21.13 再帰編集 417
21.14 コマンドの無効化 418
21.15 コマンドのヒストリー 419
21.16 キーボードマクロ 420

22 キーマップ 422
22.1 キーシーケンス 422
22.2 キーマップの基礎 423
22.3 キーマップのフォーマット 423
22.4 キーマップの作成 425
22.5 継承とキーマップ 427
22.6 プレフィクスキー 428
22.7 アクティブなキーマップ 429
22.8 アクティブなキーマップの検索 431
22.9 アクティブなキーマップの制御 432
22.10 キーの照合 434
22.11 キー照合のための関数 436
22.12 キーバインディングの変更 438

22.13 コマンドのリマップ 441
22.14 イベントシーケンス変換のためのキーマップ 442
22.14.1 通常のキーマップとの対話 443
22.15 キーのバインドのためのコマンド 444
22.16 キーマップのスキャン 445
22.17 メニューキーアップ 447
22.17.1 メニューの定義 447
22.17.1.1 単純なメニューアイテム 448
22.17.1.2 拡張メニューアイテム 448
22.17.1.3 メニューセパレーター 450
22.17.1.4 メニューアイテムのエイリアス 451
22.17.2 メニューとマウス 452
22.17.3 メニューとキーボード 452
22.17.4 メニューの例 452
22.17.5 メニューバー 453
22.17.6 ツールバー 455
22.17.7 メニューの変更 458
22.17.8 easy-menu 458

23 メジャーモードとマイナーモード 461
23.1 フック 461
23.1.1 フックの実行 462
23.1.2 フックのセット 462
23.2 メジャーモード 463
23.2.1 メジャーモードの慣習 464
23.2.2 Emacs がメジャーモードを選択する方法 468
23.2.3 メジャーモードでのヘルプ入手 470
23.2.4 派生モードの定義 470
23.2.5 基本的なメジャーモード 472
23.2.6 モードフック 473
23.2.7 Tabulated List モード 474
23.2.8 ジェネリックモード 477
23.2.9 メジャーモードの例 478
23.3 マイナーモード 479
23.3.1 マイナーモード記述の規約 480
23.3.2 キーマップとマイナーモード 482
23.3.3 マイナーモードの定義 482
23.4 モードラインのフォーマット 485
23.4.1 モードラインの基礎 485
23.4.2 モードラインのデータ構造 486
23.4.3 モードライン制御のトップレベル 487
23.4.4 モードラインで使用される変数 488
23.4.5 モードラインでの%構文 491
23.4.6 モードラインでのプロパティ 492
23.4.7 ウィンドウのヘッダーライン 493
23.4.8 モードラインのフォーマットのエミュレート 493
23.5 Imenu 494
23.6 Font Lock モード 496

23.6.1 Font Lock の基礎 496
23.6.2 検索ベースのフォント化 498
23.6.3 検索ベースのフォント化のカスタマイズ 501
23.6.4 Font Lock のその他の変数 502
23.6.5 Font Lock のレベル 503
23.6.6 事前計算されたフォント化 504
23.6.7 Font Lock のためのフェイス 504
23.6.8 構文的な Font Lock 505
23.6.9 複数行の Font Lock 構造 506
23.6.9.1 複数行の Font Lock 507
23.6.9.2 バッファー変更後のリージョンのフォント化 508
23.7 コードの自動インデント 508
23.7.1 SMIE: 無邪気なインデントエンジン 509
23.7.1.1 SMIE のセットアップと機能 509
23.7.1.2 演算子順位文法 510
23.7.1.3 言語の文法の定義 511
23.7.1.4 トークンの定義 512
23.7.1.5 非力なパーサーと歩む 513
23.7.1.6 インデントルールの指定 514
23.7.1.7 インデントルールにたいするヘルパー関数 515
23.7.1.8 インデントルールの例 516
23.7.1.9 インデントのカスタマイズ 517
23.8 Desktop Save モード 518

24 ドキュメント 519

24.1 ドキュメントの基礎 519
24.2 ドキュメント文字列へのアクセス 520
24.3 ドキュメント内でのキーバインディングの置き換え 522
24.4 テキストのクォートスタイル 524
24.5 ヘルプメッセージの文字記述 525
24.6 ヘルプ関数 526

25 ファイル 530

25.1 ファイルの visit 530
25.1.1 ファイルを visit する関数 530
25.1.2 visit のためのサブルーチン 533
25.2 バッファーの保存 534
25.3 ファイルからの読み込み 537
25.4 ファイルへの書き込み 538
25.5 ファイルのロック 539
25.6 ファイルの情報 540
25.6.1 アクセシビリティのテスト 541
25.6.2 ファイル種別の区別 542
25.6.3 本当の名前 544
25.6.4 ファイルの属性 545
25.6.5 拡張されたファイル属性 548
25.6.6 標準的な場所へのファイルの配置 549

25.7 ファイルの名前と属性の変更 550
25.8 ファイルと二次媒体 554
25.9 ファイルの名前 555
25.9.1 ファイル名の構成要素 555
25.9.2 絶対ファイル名と相対ファイル名 557
25.9.3 ディレクトリーの名前 558
25.9.4 ファイル名を展開する関数 559
25.9.5 一意なファイル名の生成 561
25.9.6 ファイル名の補完 563
25.9.7 標準的なファイル名 565
25.10 ディレクトリーのコンテンツ 565
25.11 ディレクトリーの作成・コピー・削除 568
25.12 特定のファイル名の “Magic” の作成 569
25.13 ファイルのフォーマット変換 573
25.13.1 概要 573
25.13.2 ラウンドトリップ仕様 574
25.13.3 漸次仕様 576

26 バックアップと自動保存 578
26.1 ファイルのバックアップ 578
26.1.1 バックアップファイルの作成 578
26.1.2 リネームかコピーのどちらでバックアップするか? 580
26.1.3 番号つきバックアップファイルの作成と削除 581
26.1.4 バックアップファイルの命名 582
26.2 自動保存 583
26.3 リバート 586

27 バッファー 590
27.1 バッファーの基礎 590
27.2 カレントバッファー 590
27.3 バッファーの名前 592
27.4 バッファーのファイル名 594
27.5 バッファーの変更 596
27.6 バッファーの変更 Time 597
27.7 読み取り専用のバッファー 599
27.8 バッファーリスト 600
27.9 バッファーの作成 602
27.10 バッファーの kill 603
27.11 インダイレクトバッファー 605
27.12 2 つのバッファー間でのテキストの交換 606
27.13 バッファーのギャップ 606

28 ウィンドウ 608
28.1 Emacs ウィンドウの基本概念 608
28.2 ウィンドウとフレーム 609
28.3 ウィンドウのサイズ 613
28.4 ウィンドウのリサイズ 617
28.5 ウィンドウサイズの保持 621
28.6 ウィンドウの分割 622
28.7 ウィンドウの削除 625
28.8 ウィンドウの再結合 626
28.9 ウィンドウの選択 631
28.10 ウィンドウのサイクル順 633
28.11 バッファーとウィンドウ 635
28.12 ウィンドウ内のバッファーへの切り替え 637
28.13 適切なウィンドウへのバッファーの表示 639
28.13.1 バッファーを表示するウィンドウの選択 640
28.13.2 バッファー表示用のアクション関数 641
28.13.3 バッファー表示用のアクション alist 645
28.13.4 バッファー表示の追加オプション 649
28.13.5 アクション関数の優先順 651
28.13.6 バッファー表示の思想 655
28.14 ウィンドウのヒストリー 659
28.15 専用のウィンドウ 661
28.16 ウィンドウの quit 662
28.17 サイドウィンドウ 664
28.17.1 サイドウィンドウへのバッファーの表示 665
28.17.2 サイドウィンドウのオプションと関数 665
28.17.3 サイドウィンドウによるフレームのレイアウト 667
28.18 アトミックウィンドウ 669
28.19 ウィンドウとポイント 671
28.20 ウィンドウの開始位置と終了位置 671
28.21 テキスト的なスクロール 675
28.22 割り合いによる垂直スクロール 679
28.23 水平スクロール 680
28.24 座標とウィンドウ 681
28.25 マウスによるウィンドウの自動選択 685
28.26 ウィンドウの構成 686
28.27 ウィンドウのパラメーター 688
28.28 ウィンドウのスクロールと変更のためのフック 691

29 フレーム 696
29.1 フレームの作成 697
29.2 複数の端末 698
29.3 フレームのジオメトリー 701
29.3.1 フレームのレイアウト 702
29.3.2 フレームのフォント 707
29.3.3 フレームの位置 708
29.3.4 フレームのサイズ 709
29.3.5 フレームの暗黙的なリサイズ 711

29.4 フレームのパラメーター 712
29.4.1 フレームパラメーターへのアクセス 713
29.4.2 フレームの初期パラメーター 713
29.4.3 ウィンドウフレームパラメーター 714
29.4.3.1 基本パラメーター 715
29.4.3.2 位置のパラメーター 715
29.4.3.3 サイズのパラメーター 717
29.4.3.4 レイアウトのパラメーター 719
29.4.3.5 バッファーのパラメーター 721
29.4.3.6 フレームとの相互作用のためのパラメーター 721
29.4.3.7 マウスドラッグのパラメーター 723
29.4.3.8 ウィンドウ管理のパラメーター 723
29.4.3.9 カーソルのパラメーター 725
29.4.3.10 フォントとカラーのパラメーター 726
29.4.4 ジオメトリー 728
29.5 端末のパラメーター 729
29.6 フレームのタイトル 730
29.7 フレームの削除 730
29.8 すべてのフレームを探す 731
29.9 ミニバッファーとフレーム 732
29.10 入力のフォーカス 732
29.11 フレームの可視性 736
29.12 フレームの raise、lower、re-stack 737
29.13 フレーム構成 739
29.14 子フレーム 739
29.15 マウスの追跡 742
29.16 マウスの位置 743
29.17 ポップアップメニュー 744
29.18 ダイアログボックス 745
29.19 ポインターの形状 746
29.20 ウィンドウシステムによる選択 747
29.21 ドラッグアンドドロップ 748
29.22 カラー名 748
29.23 テキスト端末のカラー 749
29.24 X リソース 750
29.25 ディスプレー機能のテスト 751

30 ポジション 755

30.1 ポイント 755
30.2 モーション 756
30.2.1 文字単位の移動 756
30.2.2 単語単位の移動 757
30.2.3 バッファー終端への移動 758
30.2.4 テキスト行単位の移動 758
30.2.5 スクリーン行単位の移動 760
30.2.6 バランスのとれたカッコを越えた移動 762
30.2.7 文字のスキップ 764
30.3 エクスカーション 765

30.4 ナローイング 766

31 マーカー 768

31.1 マーカーの概要 768
31.2 マーカーのための述語 769
31.3 マーカーを作成する関数 769
31.4 マーカーからの情報 771
31.5 マーカーの挿入タイプ 771
31.6 マーカー位置の移動 772
31.7 マーク 772
31.8 リージョン 776

32 テキスト 777

32.1 ポイント近傍のテキストを調べる 777
32.2 バッファーのコンテンツを調べる 778
32.3 テキストの比較 781
32.4 テキストの挿入 781
32.5 ユーザーレベルの挿入コマンド 783
32.6 テキストの削除 784
32.7 ユーザーレベルの削除コマンド 786
32.8 kill リング 788
32.8.1 kill リングの概念 788
32.8.2 kill 用の関数 788
32.8.3 yank 789
32.8.4 yank 用の関数 790
32.8.5 低レベルの kill リング 791
32.8.6 kill リングの内部 792
32.9 アンドゥ 793
32.10 アンドゥリストの保守 796
32.11 fill 797
32.12 fill のマージン 800
32.13 Adaptive Fill モード 801
32.14 オート fill 802
32.15 テキストのソート 803
32.16 列を数える 806
32.17 インデント 807
32.17.1 インデント用のプリミティブ 807
32.17.2 メジャーモードが制御するインデント 807
32.17.3 リージョン全体のインデント 809
32.17.4 前行に相対的なインデント 810
32.17.5 調整可能なタブストップ 811
32.17.6 インデントにもとづくモーションコマンド 811
32.18 大文字小文字の変更 811
32.19 テキストのプロパティ 813
32.19.1 テキストプロパティを調べる 813
32.19.2 テキストプロパティの変更 814
32.19.3 テキストプロパティの検索関数 817

32.19.4 特殊な意味をもつプロパティ 820
32.19.5 フォーマットされたテキストのプロパティ 826
32.19.6 テキストプロパティの粘着性 827
32.19.7 テキストプロパティの lazy な計算 828
32.19.8 クリック可能なテキストの定義 828
32.19.9 フィールドの定義と使用 831
32.19.10 なぜテキストプロパティはインターバルではないのか 832
32.20 文字コードの置き換え 833
32.21 レジスター 834
32.22 テキストの交換 835
32.23 バッファーテキストの置換 836
32.24 圧縮されたデータの処理 836
32.25 Base 64 エンコーディング 837
32.26 チェックサムとハッシュ 838
32.27 GnuTLS 暗号化 839
32.27.1 GnuTLS 暗号化入力のフォーマット 839
32.27.2 GnuTLS 暗号化関数 840
32.28 HTML と XML の解析 841
32.28.1 ドキュメントオブジェクトモデル 842
32.29 JSON 値の解析と生成 844
32.30 JSONRPC による対話 846
32.30.1 概観 846
32.30.2 プロセスベースの JSONRPC 接続 847
32.30.3 JSONRPC の JSON オブジェクトフォーマット 848
32.30.4 遅延された JSONRPC リクエスト 848
32.31 グループのアトミックな変更 848
32.32 フックの変更 850

33 非 ASCII 文字 852

33.1 テキストの表現方法 852
33.2 マルチバイト文字の無効化 854
33.3 テキスト表現の変換 854
33.4 表現の選択 855
33.5 文字コード 856
33.6 文字のプロパティ 857
33.7 文字セット 861
33.8 文字セットのスキャン 863
33.9 文字の変換 863
33.10 コーディングシステム 864
33.10.1 コーディングシステムの基本概念 865
33.10.2 エンコーディングと I/O 866
33.10.3 Lisp でのコーディングシステム 867
33.10.4 ユーザーが選択したコーディングシステム 870
33.10.5 デフォルトのコーディングシステム 871
33.10.6 単一の操作にたいするコーディングシステムの指定 874
33.10.7 明示的なエンコードとデコード 876
33.10.8 端末 I/O のエンコーディング 877
33.11 入力メソッド 878

33.12 locale 879

34 検索とマッチング 881
34.1 文字列の検索 881
34.2 検索と大文字小文字 883
34.3 正規表現 883
34.3.1 正規表現の構文 884
34.3.1.1 正規表現内の特殊文字 884
34.3.1.2 文字クラス 887
34.3.1.3 正規表現内のバッククラッシュ構文 889
34.3.2 正規表現の複雑な例 892
34.3.3 正規表現の関数 892
34.4 正規表現の検索 894
34.5 POSIX 正規表現の検索 897
34.6 マッチデータ 897
34.6.1 マッチしたテキストの置換 897
34.6.2 単純なマッチデータへのアクセス 899
34.6.3 マッチデータ全体へのアクセス 900
34.6.4 マッチデータの保存とリストア 901
34.7 検索と置換 902
34.8 編集で使用される標準的な正規表現 905

35 構文テーブル 906
35.1 構文テーブルの概念 906
35.2 構文記述子 907
35.2.1 構文クラスのテーブル 907
35.2.2 構文フラグ 909
35.3 構文テーブルの関数 910
35.4 構文プロパティ 912
35.5 モーションと構文 913
35.6 式のパース 914
35.6.1 パースにもとづくモーションコマンド 914
35.6.2 ある位置のパース状態を調べる 915
35.6.3 パーサー状態 915
35.6.4 低レベルのパース 916
35.6.5 パースを制御するためのパラメーター 917
35.7 構文テーブルの内部 917
35.8 カテゴリー 918

36 abbrev と abbrev 展開 922
36.1 abbrev テーブル 922
36.2 abbrev の定義 923
36.3 ファイルへの abbrev の保存 924
36.4 略語の照会と展開 925
36.5 標準 abbrev テーブル 927
36.6 abbrev プロパティー 927
36.7 abbrev テーブルのプロパティー 928

37 スレッド 929
37.1 基本的なスレッド関数 929
37.2 ミューテックス 931
37.3 条件変数 931
37.4 スレッドリスト 932

38 プロセス 934
38.1 サブプロセスを作成する関数 934
38.2 shell 引数 936
38.3 同期プロセスの作成 937
38.4 非同期プロセスの作成 941
38.5 プロセスの削除 946
38.6 プロセスの情報 947
38.7 プロセスへの入力の送信 950
38.8 プロセスへのシグナルの送信 951
38.9 プロセスからの出力の受信 953
38.9.1 プロセスのバッファー 953
38.9.2 プロセスのフィルター関数 955
38.9.3 プロセス出力のデコード 957
38.9.4 プロセスの出力を受け取る 957
38.9.5 プロセスとスレッド 958
38.10 センチネル: プロセス状態の変更の検知 959
38.11 exit 前の問い合わせ 961
38.12 別のプセスへのアクセス 961
38.13 トランザクションキュー 963
38.14 ネットワーク接続 964
38.15 ネットワークサーバー 967
38.16 データグラム 967
38.17 低レベルのネットワークアクセス 968
38.17.1 `make-network-process` 968
38.17.2 ネットワークのオプション 971
38.17.3 ネットワーク機能の可用性のテスト 972
38.18 その他のネットワーク機能 972
38.19 シリアルポートとの対話 974
38.20 バイト配列の pack と unpack 977
38.20.1 データレイアウトの記述 977
38.20.2 バイトの unpack と pack を行う関数 979

39 Emacs のディスプレー表示 981
39.1 スクリーンのリフレッシュ 981
39.2 強制的な再表示 981
39.3 切り詰め 982
39.4 エコーエリア 983
39.4.1 エコーエリアへのメッセージの表示 983
39.4.2 処理の進捗レポート 985
39.4.3 `*Messages*`へのメッセージのロギング 988
39.4.4 エコーエリアのカスタマイズ 988

39.5 警告のレポート 989
39.5.1 警告の基礎 989
39.5.2 警告のための変数 990
39.5.3 警告のためのオプション 991
39.5.4 遅延された警告 991
39.6 不可視のテキスト 992
39.7 選択的な表示 994
39.8 一時的な表示 996
39.9 オーバーレイ 999
39.9.1 オーバーレイの管理 1000
39.9.2 オーバーレイのプロパティ 1002
39.9.3 オーバーレイにたいする検索 1006
39.10 表示されるテキストのサイズ 1007
39.11 行の高さ 1010
39.12 フェイス 1011
39.12.1 フェイスの属性 1012
39.12.2 フェイスの定義 1015
39.12.3 フェイス属性のための関数 1018
39.12.4 フェイスの表示 1021
39.12.5 フェイスのリマップ 1021
39.12.6 フェイスを処理するための関数 1023
39.12.7 フェイスの自動割り当て 1024
39.12.8 基本的なフェイス 1024
39.12.9 フォントの選択 1025
39.12.10 フォントの照会 1027
39.12.11 フォントセット 1027
39.12.12 低レベルのフォント表現 1029
39.13 フリンジ 1034
39.13.1 フリンジのサイズと位置 1034
39.13.2 フリンジのインジケーター 1035
39.13.3 フリンジのカーソル Frin 1036
39.13.4 フリンジのビットマップ 1037
39.13.5 フリンジビットマップのカスタマイズ 1038
39.13.6 オーバーレイ矢印 1039
39.14 スクロールバー 1039
39.15 ウィンドウディバイダー 1042
39.16 `display`プロパティ 1043
39.16.1 テキストを置換するディスプレー仕様 1043
39.16.2 スペースの指定 1044
39.16.3 スペースにたいするピクセル指定 1045
39.16.4 その他のディスプレー仕様 1046
39.16.5 マージン内への表示 1047
39.17 イメージ 1049
39.17.1 イメージのフォーマット 1049
39.17.2 イメージのディスクリプタ 1049
39.17.3 XBM イメージ 1053
39.17.4 XPM イメージ 1054
39.17.5 ImageMagick イメージ 1054

39.17.6 SVG イメージ 1055
39.17.7 その他のイメージタイプ 1061
39.17.8 イメージの定義 1061
39.17.9 イメージの表示 1063
39.17.10 マルチフレームのイメージ 1065
39.17.11 イメージキャッシュ 1065
39.18 埋め込みネイティブウィジェット 1066
39.19 ボタン 1068
39.19.1 ボタンのプロパティ 1068
39.19.2 ボタンのタイプ 1069
39.19.3 ボタンの作成 1069
39.19.4 ボタンの操作 1070
39.19.5 ボタンのためのバッファーコマンド 1071
39.20 抽象的なディスプレー 1072
39.20.1 抽象ディスプレーの関数 1073
39.20.2 抽象ディスプレーの例 1075
39.21 カッコの点滅 1077
39.22 文字の表示 1077
39.22.1 通常の表示の慣習 1078
39.22.2 ディスプレーテーブル 1079
39.22.3 アクティブなディスプレーテーブル 1080
39.22.4 グリフ 1081
39.22.5 グリフなし文字の表示 1081
39.23 ビープ 1083
39.24 ウィンドウシステム 1083
39.25 ツールチップ 1084
39.26 双方向テキストの表示 1085

40 オペレーティングシステムのインターフェース 1090

40.1 Emacs のスタートアップ 1090
40.1.1 要約: スタートアップ時のアクション順序 1090
40.1.2 init ファイル 1093
40.1.3 端末固有の初期化 1095
40.1.4 コマンドライン引数 1096
40.2 Emacs からの脱出 1097
40.2.1 Emacs の kill 1097
40.2.2 Emacs のサスペンド 1098
40.3 オペレーティングシステムの環境 1100
40.4 ユーザーの識別 1103
40.5 時刻 1104
40.6 タイムゾーンのルール 1105
40.7 時刻の変換 1106
40.8 時刻のパースとフォーマット 1109
40.9 プロセッサーの実行時間 1112
40.10 時間の計算 1113
40.11 遅延実行のためのタイマー 1114
40.12 アイドルタイマー 1116
40.13 端末の入力 1118

40.13.1 入力のモード ... 1118
40.13.2 入力の記録 ... 1118
40.14 端末の出力 ... 1119
40.15 サウンドの出力 ... 1120
40.16 X11 キーシンボルの処理 ... 1121
40.17 batch モード ... 1121
40.18 セッションマネージャー ... 1122
40.19 デスクトップ通知 ... 1123
40.20 ファイル変更による通知 ... 1127
40.21 動的にロードされるライブラリー ... 1130
40.22 セキュリティへの配慮 ... 1131

41 配布用 Lisp コードの準備 ... 1134
41.1 パッケージ化の基礎 ... 1134
41.2 単純なパッケージ ... 1135
41.3 複数ファイルのパッケージ ... 1136
41.4 パッケージアーカイブの作成と保守 ... 1138
41.5 アーカイブウェブサーバーとのインターフェイス ... 1139

Appendix A Emacs 26 のアンチニュース ... 1140

Appendix B GNU Free Documentation License ... 1143

Appendix C GNU General Public License ... 1151

Appendix D ヒントと規約 ... 1162
D.1 Emacs Lisp コーディング規約 ... 1162
D.2 キーバインディング規約 ... 1164
D.3 Emacs プログラミングのヒント ... 1165
D.4 コンパイル済みコードを高速化ためのヒント ... 1167
D.5 コンパイラー警告を回避するためのヒント ... 1167
D.6 ドキュメント文字列のヒント ... 1168
D.7 コメント記述のヒント ... 1171
D.8 Emacs ライブラリーのヘッダーの慣習 ... 1172

Appendix E GNU Emacs の内部 ... 1175
E.1 Emacs のビルド ... 1175
E.2 純粋ストレージ ... 1177
E.3 ガーベージコレクション ... 1178
E.4 スタックに割り当てられたオブジェクト ... 1183
E.5 メモリー使用量 ... 1183
E.6 C 方言 ... 1184
E.7 Emacs プリミティブの記述 ... 1184
E.8 動的にロードされるモジュールの記述 ... 1188

E.8.1　モジュールの初期化コード 1189
E.8.2　モジュール関数の記述 1191
E.8.3　Lisp・モジュール間の値変換 1193
E.8.4　その他の便利なモジュール用関数 1199
E.8.5　モジュールでの非ローカル脱出 1201
E.9　オブジェクトの内部 1202
E.9.1　バッファーの内部 1203
E.9.2　ウィンドウの内部 1208
E.9.3　プロセスの内部 1212
E.10　C の整数型 1214

Appendix F　標準的なエラー 1216

Appendix G　標準的なキーマップ 1220

Appendix H　標準的なフック 1223

Index 1227

1 イントロダクション

GNU Emacs テキストエディターのほとんどの部分は、Emacs Lisp と呼ばれるプログラミング言語で記述されています。新しいコードを Emacs Lisp で記述して、このエディターの拡張としてインストールできます。しかし Emacs Lisp は、単なる拡張言語を越える言語であり、それ自体で完全なコンピュータープログラミング言語です。他のプログラミング言語で行なうすべてのことに、この言語を使用できます。

Emacs Lisp はエディターの中で使用するようにデザインされているので、テキストのスキャンやパースのための特別な機能をもち、同様にファイル、バッファー、ディスプレー、サブプロセスを処理する機能をもちます。Emacs Lisp は編集機能と密に統合されています。つまり編集コマンドは Lisp プログラムから簡単に呼び出せる関数で、カスタマイズのためのパラメーターは普通の Lisp 変数です。

このマニュアルは Emacs Lisp の完全な記述を試みます。初心者のための Emacs Lisp のイントロダクションは、Free Software Foundation からも出版されている、Bob Chassell の *An Introduction to Emacs Lisp Programming* を参照してください。このマニュアルは、Emacs を使用した編集に熟知していることを前提としています。これの基本的な情報については、*The GNU Emacs Manual* を参照してください。

おおまかに言うと、前の方のチャプターでは多くのプログラミング言語の機能にたいして、Emacs Lisp での対応する機能を説明し、後の方のチャプターでは Emacs Lisp に特異な機能や、編集に特化した関連する機能を説明します。

これはエディション [No value for “VERSION”] Emacs 27.1 に対応した *GNU Emacs Lisp Reference Manual* です。

1.1 注意事項

このマニュアルは幾多のドラフトを経てきました。ほとんど完璧ではありますが、不備がないとも言えません。(ほとんどの特定のモードのように) それらが副次的であるとか、まだ記述されていないという理由により、カバーされていないトピックもあります。わたしたちがそれらを完璧に扱うことはできないので、いくつかの部分は意図的に省略しました。

このマニュアルは、それがカバーしている事柄については完全に正しくあるべきあり、故に特定の説明テキスト、チャプターやセクションの順番にたいしての批判にオープンであるべきです。判りにくかったり、このマニュアルでカバーされていない何かを学ぶためにソースを見たり実地から学ぶ必要があるなら、このマニュアルはおそらく訂正されるべきなのかもしれません。どうかわたしたちにそれを教えてください。

このマニュアルを使用するときは、訂正のためにページにマークしてください。そうすれば後でそれを探して、わたしたちに送ることができます。関数や関数グループの単純で現実的な例を思いついたときは、ぜひそれを記述して送ってください。それが妥当ならチャプター名、セクション名、関数名への参照をコメントしてください。なぜならページ番号やチャプター番号、セクション番号は変更されるので、あなたが言及しているテキストを探すのに問題が生じるかもしれないからです。あなたが訂正を求めるエディションのバージョンも示してください。

`M-x report-emacs-bug`を使用して、コメントや訂正を送ってください。

1.2 Lisp の歴史

Lisp(LISt Processing language: リスト処理言語) は、MIT(Massachusetts Institute of Technology: マサチューセッツ工科大学) で、AI(artificial intelligence: 人工知能) の研究のために、1950

年代末に最初に開発されました。Lisp 言語の強力なパワーは、編集コマンドの記述のような、他の目的にも適っています。

長年の間に何ダースもの Lisp 実装が構築されてきて、それらのそれぞれに特異な点があります。これらの多くは、1960 年代に MIT の Project MAC で記述された、Maclisp に影響を受けています。最終的に、Maclisp 後裔の実装者は共同して、Common Lisp と呼ばれる標準の Lisp システムを開発しました。その間に MIT の Gerry Sussman と Guy Steele により、簡潔ながらとても強力な Lisp 方言の、Scheme が開発されました。

GNU Emacs Lisp は Maclisp から多く、Common Lisp から少し影響を受けています。Common Lisp を知っている場合、多くの類似点に気づくでしょう。しかし Common Lisp の多くの機能は、GNU Emacs が要求するメモリー量を削減するために、省略または単純化されています。ときには劇的に単純化されているために、Common Lisp ユーザーは混乱するかもしれません。わたしたちは時折 GNU Emacs Lisp が Common Lisp と異なるか示すでしょう。Common Lisp を知らない場合、それについて心配する必要はありません。このマニュアルは、それ自体で自己完結しています。

`cl-lib`ライブラリーを通じて、Common Lisp をかなりエミュレートできます。Section “Overview” in *Common Lisp Extensions* を参照してください。

Emacs Lisp は Scheme の影響は受けていません。しかし GNU プロジェクトには Guile と呼ばれる Scheme 実装があります。拡張が必要な新しい GNU ソフトウェアーでは、Guile を使用します。

1.3 表記について

このセクションでは、このマニュアルで使用する表記規約を説明します。あなたはこのセクションをスキップして、後から参照したいと思うかもしれません。

1.3.1 用語について

このマニュアルでは、“Lisp リーダー” および “Lisp プリンター” という用語で、Lisp のテキスト表現を実際の Lisp オブジェクトに変換したり、その逆を行なう Lisp ルーチンを参照します。詳細については、Section 2.1 [Printed Representation], page 8 を参照してください。あなた、つまりこのマニュアルを読んでいる人のことはプログラマーと考えて “あなた” と呼びます。“ユーザー” とは、あなたの記述したものも含めて、Lisp プログラムを使用する人を指します。

Lisp コードの例は、`(list 1 2 3)`のようなフォーマットです。メタ構文変数 (metasyntactic variables) を表す名前や、説明されている関数の引数名前は、*first-number*のようにフォーマットされています。

1.3.2 `nil`と`t`

Emacs Lisp では、シンボル `nil`には 3 つの異なる意味があります。1 つ目は ‘`nil`’という名前のシンボル、2 つ目は論理値の *false*、3 つ目は空リスト— つまり要素が 0 のリストです。変数として使用した場合、`nil`は常に値 `nil`をもちます。

Lisp リーダーに関する限り、‘`()`’と ‘`nil`’は同一です。これらは同じオブジェクト、つまりシンボル `nil`を意味します。このシンボルを異なる方法で記述するのは、完全に人間の読み手を意図したものです。Lisp リーダーが ‘`()`’か ‘`nil`’のどちらかを読み取った後は、プログラマーが実際にどちらの表現で記述したかを判断する方法はありません。

このマニュアルでは、空リストを意味することを強調したいときは `()`と記述し、論理値の *false*を意味することを強調したいときは `nil`と記述します。この慣習は Lisp プログラムで使用してもよいでしょう。

```
(cons 'foo ())                  ; 空リストを強調
```

```
(setq foo-flag nil)             ; 論理値の false を強調
```

論理値が期待されているコンテキストでは、非 `nil` は *true* と判断されます。しかし論理値の *true* を表す好ましい方法は `t` です。*true* を表す値を選択する必要があり、他に選択の根拠がない場合は `t` を使用してください。シンボル `t` は、常に値 `t` をもちます。

Emacs Lisp での `nil` と `t` は、常に自分自身を評価する特別なシンボルです。そのためプログラムでこれらを定数として使用する場合、クォートする必要はありません。これらの値の変更を試みると、結果は `setting-constant` エラーとなります。Section 12.2 [Constant Variables], page 174 を参照してください。

`booleanp` *object* [Function]

object が 2 つの正規のブーリーン値 (`t` か `nil`) のいずれかなら、非 `nil` をリターンする。

1.3.3 評価の表記

評価できる Lisp 式のことをフォーム *(form)* と呼びます。フォームの評価により、これは結果として常に Lisp オブジェクトを生成します。このマニュアルの例では、これを '⇒' で表します:

```
(car '(1 2))
     ⇒ 1
```

これは "`(car '(1 2))` を評価すると、1 になる" と読むことができます。

フォームがマクロ呼び出しの場合、それは評価されるための新たな Lisp フォームに展開されます。展開された結果は '↦' で表します。わたしたちは展開されたフォームの評価し結果を示すこともあれば、示さない場合もあります。

```
(third '(a b c))
     ↦ (car (cdr (cdr '(a b c))))
     ⇒ c
```

1 つのフォームを説明するために、同じ結果を生成する別のフォームを示すこともあります。完全に等価な 2 つのフォームは、'≡' で表します。

```
(make-sparse-keymap) ≡ (list 'keymap)
```

1.3.4 プリントの表記

このマニュアルの例の多くは、それらが評価されるときにテキストをプリントします。(`*scratch*` バッファーのような)Lisp Interaction バッファーでコード例を実行する場合、プリントされるテキストはそのバッファーに挿入されます。(関数 `eval-region` での評価のように) 他の方法でコード例を実行する場合、プリントされるテキストはエコーエリアに表示されます。

このマニュアルの例はプリントされるテキストがどこに出力されるかに関わらず、それを '⊣' で表します。フォームを評価することにより戻される値は、'⇒' とともに後続の行で示します。

```
(progn (prin1 'foo) (princ "\n") (prin1 'bar))
     ⊣ foo
     ⊣ bar
     ⇒ bar
```

1.3.5 エラーメッセージ

エラーをシグナルする例もあります。これは通常、エコーエリアにエラーメッセージを表示します。エラーメッセージの行は ' error ' で始まります。' error ' 自体は、エコーエリアに表示されないことに注意してください。

```
(+ 23 'x)
```

error Wrong type argument: number-or-marker-p, x

1.3.6 バッファーテキストの表記

バッファー内容の変更を説明する例もあます。それらの例では、そのテキストの before(以前) と after(以後) のバージョンを示します。それらの例では、バッファー内容の該当する部分を、ダッシュを用いた 2 行の破線 (バッファー名を含む) で示します。さらに、'⋆'はポイントの位置を表します (もちろんポイントのシンボルはバッファーのテキストの一部ではなく、ポイントが現在配されている 2 つの文字の間の位置を表す)。

```
---------- Buffer: foo ----------
This is the ⋆contents of foo.
---------- Buffer: foo ----------

(insert "changed ")
     ⇒ nil
---------- Buffer: foo ----------
This is the changed ⋆contents of foo.
---------- Buffer: foo ----------
```

1.3.7 説明のフォーマット

このマニュアルでは関数 (function)、変数 (variable)、コマンド (command)、ユーザーオプション (user option)、スペシャルフォーム (special form) を、統一されたフォーマットで記述します。記述の最初の行には、そのアイテムの名前と、もしあれば引数 (argument) が続きます。そのアイテムの属するカテゴリー (function、variable など) は、ページの右マージンの隣にプリントされます。それ以降の行は説明行で、例を含む場合もあります。

1.3.7.1 関数の説明例

関数の記述では、関数の名前が最初に記述されます。同じ行に引数の名前のリストが続きます。引数の値を参照するために、引数の名前は記述の本文にも使用されます。

引数リストの中にキーワード`&optional`がある場合、その後の引数が省略可能であることを示します (省略された引数のデフォルトは `nil`)。その関数を呼び出すときは、`&optional`を記述しないでください。

キーワード`&rest`(これの後には 1 つの引数名を続けなければならない) は、その後に任意の引数を続けることができることを表します。`&rest`の後に記述された引数名の値には、その関数に渡された残りのすべての引数がリストとしてセットされます。この関数を呼び出すときは、`&rest`を記述しないでください。

以下は `foo`という架空の関数 (function) の説明です:

`foo` *integer1* **&optional** *integer2* **&rest** *integers* [Function]

関数 `foo`は *integer2*から *integer1*を減じてから、その結果に残りすべての引数を加える。*integer2*が与えられなかった場合、デフォルトして数値 19 が使用される。

```
(foo 1 5 3 9)
     ⇒ 16
(foo 5)
     ⇒ 14
```

より一般的には、

```
(foo w x y...)
≡
(+ (- x w) y...)
```

慣例として引数の名前には、(たとえば *integer*、*integer1*、*buffer*のような) 期待されるタイプ名が含まれます。(*buffers*のような) 複数形のタイプは、しばしばその型のオブジェクトのリストを意味します。*object*のような引き数名は、それが任意の型であることを表します (Emacs オブジェクトタイプのリストは Chapter 2 [Lisp Data Types], page 8 を参照)。他の名前をもつ引数 (たとえば *new-file*) はその関数に固有の引数で、関数がドキュメント文字列をもつ場合、引数のタイプはその中で説明されるべきです (Chapter 24 [Documentation], page 519 を参照)。

`&optional`や`&rest`により修飾される引数のより完全な説明は、Section 13.2 [Lambda Expressions], page 211 を参照してください。

コマンド (command)、マクロ (macro)、スペシャルフォーム (special form) の説明も同じフォーマットですが、'`Function`'が'`Command`'、'`Macro`'、'`Special Form`'に置き換えられます。コマンドはとは単に、インタラクティブ (interactive: 対話的) に呼び出すことができる関数です。マクロは関数とは違う方法 (引数は評価されない) で引数を処理しますが、同じ方法で記述します。

マクロとスペシャルフォームにたいする説明には、特定のオプション引数や繰り替えされる引数のために、より複雑な表記が使用されます。なぜなら引数リストが、より複雑な方法で別の引数に分離されるからです。'[*optional-arg*]'は *optional-arg*がオプションであることを意味し、'*repeated-args*`...`'は 0 個以上の引数を表します。カッコ (parentheses) は、複数の引数をリスト構造の追加レベルにグループ化するのに使用されます。以下は例です:

`count-loop` (*var* [*from to* [*inc*]]) *body*... [Special Form]

この架空のスペシャルフォームは、*body*フォームを実行してから変数 *var*をインクリメントするループを実装します。最初の繰り返しでは変数は値 *from*をもちます。以降の繰り返しでは、変数は 1(*inc*が与えられた場合は *inc*) 増分されます。*var*が *to*に等しい場合、*body*を実行する前にループを exit します。以下は例です:

```
(count-loop (i 0 10)
  (prin1 i) (princ " ")
  (prin1 (aref vector i))
  (terpri))
```

*from*と *to*が省略された場合、ループを実行する前に *var*に `nil`がバインドされ、繰り返しの先頭において *var*が非 `nil`の場合は、ループを exit します。以下は例です:

```
(count-loop (done)
  (if (pending)
      (fixit)
    (setq done t)))
```

このスペシャルフォームでは、引数 *from*と *to*はオプションですが、両方を指定するか未指定にするかのいずれかでなければなりません。これらの引数が与えられた場合には、オプションで *inc*も同様に指定することができます。これらの引数は、フォームのすべての残りの要素を含む *body*と区別するために、引数 *var*とともにリストにグループ化されます。

1.3.7.2 変数の説明例

変数 *(variable)* とは、オブジェクトにバインド *(bind)* される名前です (セット *(set)* とも言う)。変数がバインドされたオブジェクトのことを値 *(value)* と呼びます。このような場合には、その変数が値

をもつという言い方もします。ほとんどすべての変数はユーザーがセットすることができますが、特にユーザーが変更できる特定の変数も存在し、これらはユーザーオプション *(user options)* と呼ばれます。通常の変数およびユーザーオプションは、関数と同様のフォーマットを使用して説明されますが、それらには引数がありません。

以下は架空の変数 `electric-future-map`の説明です。

`electric-future-map` [Variable]
: この変数の値は Electric Command Future モードで使用される完全なキーマップである。このマップ内の関数により、まだ実行を考えていないコマンドの編集が可能になる。

ユーザーオプションも同じフォーマットをもちますが、'`Variable`'が'`User Option`'に置き換えられます。

1.4 バージョンの情報

以下の機能は、使用している Emacs に関する情報を提供します。

`emacs-version` **&optional** *here* [Command]
: この関数は実行している Emacs のバージョンを説明する文字列を return すす。これはバグレポートにこの文字列を含めるときに有用である。

```
(emacs-version)
  ⇒ "GNU Emacs 26.1 (build 1, x86_64-unknown-linux-gnu,
             GTK+ Version 3.16) of 2017-06-01"
```

*here*が非 `nil`ならテキストをバッファーのポイントの前に挿入して、`nil`をリターンする。この関数がインタラクティブに呼び出すと、同じ情報をエコーエリアに出力する。プレフィクス引数を与えると、*here*が非 `nil`になる。

`emacs-build-time` [Variable]
: この変数の値は Emacs がビルドされた日時を示す。値は `current-time`の形式 (Section 40.5 [Time of Day], page 1104 を参照)、その情報が利用できなければ `nil`。

```
emacs-build-time
     ⇒ (20614 63694 515336 438000)
```

`emacs-version` [Variable]
: この変数の値は実行中の Emacs のバージョンであり、`"26.1"`のような文字列。`"26.0.91"`のように 3 つの数値コンポーネントをもつ値はリリース版ではなくテストバージョンであることを示す (Emacs 26.1 より前では`"25.1.1"`のように文字列の最後に余分な整数コンポーネントが含まれていたが、これは現在は `emacs-build-number`に格納される)。

`emacs-major-version` [Variable]
: Emacs のメジャーバージョン番号を示す整数。Emacs 23.1 では値は 23。

`emacs-minor-version` [Variable]
: Emacs のマイナーバージョン番号を示す整数。Emacs 23.1 では値は 1。

`emacs-build-number` [Variable]
: これは同一のディレクトリーにおいて Emacs が (クリーニングなしで) ビルドされるたびに増分される整数。これは Emacs の開発時だけに関係のある変数。

`emacs-repository-version` [Variable]
: Emacs がビルドされたリポジトリのリビジョン番号を与える文字列。Emacs がリビジョンコントロール外部でビルドされた場合の値は `nil`。

`emacs-repository-branch` [Variable]
: Emacs がビルドされたリポジトリブランチを与える文字列。ほとんどの場合は`"master"`。Emacs がリビジョンコントロール外部でビルドされた場合の値は `nil`。

1.5 謝辞

このマニュアルは当初、Robert Krawitz、Bil Lewis、Dan LaLiberte、Richard M. Stallman、Chris Welty、および GNU マニュアルグループのボランティアにより、数年を費やして記述されました。Robert J. Chassell はこのマニュアルのレビューと編集を Defense Advanced Research Projects Agency、ARPA Order 6082 のサポートのもとに手助けしてくれ、Computational Logic, Inc の Warren A. Hunt, Jr. によりアレンジされました。それ以降も追加のセクションが Miles Bader、Lars Brinkhoff、Chong Yidong、Kenichi Handa、Lute Kamstra、Juri Linkov、Glenn Morris、Thien-Thi Nguyen、Dan Nicolaescu、Martin Rudalics、Kim F. Storm、Luc Teirlinck、Eli Zaretskii、およびその他の人たちにより記述されました。

Drew Adams、Juanma Barranquero、Karl Berry、Jim Blandy、Bard Bloom、Stephane Boucher、David Boyes、Alan Carroll、Richard Davis、Lawrence R. Dodd、Peter Doornbosch、David A. Duff、Chris Eich、Beverly Erlebacher、David Eckelkamp、Ralf Fassel、Eirik Fuller、Stephen Gildea、Bob Glickstein、Eric Hanchrow、Jesper Harder、George Hartzell、Nathan Hess、Masayuki Ida、Dan Jacobson、Jak Kirman、Bob Knighten、Frederick M. Korz、Joe Lammens、Glenn M. Lewis、K. Richard Magill、Brian Marick、Roland McGrath、Stefan Monnier、Skip Montanaro、John Gardiner Myers、Thomas A. Peterson、Francesco Potort^^c3^^ac、Friedrich Pukelsheim、Arnold D. Robbins、Raul Rockwell、Jason Rumney、Per Starb^^c3^^a4ck、Shinichirou Sugou、Kimmo Suominen、Edward Tharp、Bill Trost、Rickard Westman、Jean White、Eduard Wiebe、Matthew Wilding、Carl Witty、Dale Worley、Rusty Wright、David D. Zuhn により訂正が提供されました。

より完全な貢献者のリストは、Emacs ソースリポジトリーの関連する変更ログエントリーを参照してください。

2 Lisp のデータ型

Lisp のオブジェクト *(object)* とは、Lisp プログラムから操作されるデータです。型 *(type)* やデータ型 *(data type)* とは、可能なオブジェクトの集合を意味します。

すべてのオブジェクトは少なくとも 1 つの型に属します。同じ型のオブジェクトは同様な構造をもち、通常は同じコンテキストで使用されます。型を重複してもつことができ、オブジェクトは複数の型に属することができます。その結果として、あるオブジェクトが特定の型に属するかどうかを尋ねることはできますが、オブジェクトがその型だけに属するかどうかは決定できません。

Emacs にはいくつかの基本オブジェクト型が組み込まれています。これらの型は他のすべての型を構成するもとであり、プリミティブ型 *(primitive types:* 基本型*)* と呼ばれます。すべてのオブジェクトはただ 1 つのプリミティブ型に属します。これらの型には整数 *(integer)*、浮動小数点数 *(float)*、コンス *(cons)*、シンボル *(symbol)*、文字列 *(string)*、ベクター *(vector)*、ハッシュテーブル *(hash-table)*、サブルーチン *(subr)*、バイトコード関数 *(byte-code function)*、レコード *(record)*、および *buffer* のような編集に関連した特別な型が含まれます (Section 2.5 [Editing Types], page 25 を参照)。

プリミティブ型にはそれぞれ、オブジェクトがその型のメンバーかどうかのチェックを行なうために、それぞれ対応する Lisp 関数があります。

他の多くの言語とは異なり、Lisp のオブジェクトは自己記述 *(self-typing)* 的です。オブジェクトのプリミティブ型は、オブジェクト自体に暗に含まれます。たとえばオブジェクトがベクターなら、それを数字として扱うことはできません。Lisp はベクターが数字でないことを知っているのです。

多くの言語では、プログラマーは各変数にたいしてデータ型を宣言しなければならず、コンパイラーは型を知っていますが、データの中に型はありません。Emacs Lisp には、このような型宣言はありません。Lisp 変数は任意の型の値をもつことができ、変数に保存した値と型を記憶します (実際には特定の型の値だけをもつことができる少数の Emacs Lisp 変数がある。Section 12.16 [Variables with Restricted Values], page 205 を参照されたい)。

このチャプターでは、GNU Emacs Lisp の各標準型の意味、プリント表現 (printed representation)、入力構文 (read syntax) を説明します。これらのデータ型を使用する方法についての詳細は、以降のチャプターを参照してください。

2.1 プリント表現と読み取り構文

オブジェクトのプリント表現 (printed representation) とは、オブジェクトにたいして Lisp プリンター (関数 `prin1`) が生成する出力のフォーマットです。すべてのデータ型は一意なプリント表現をもちます。オブジェクトの入力構文 *(read syntax)* とは、オブジェクトにたいして Lisp リーダー (関数 `read`) が受け取る入力のフォーマットです。これは一意である必要はありません。多くの種類のオブジェクトが複数の構文をもちます。Chapter 19 [Read and Print], page 331 を参照してください。

ほとんどの場合、オブジェクトのプリント表現が、入力構文としても使用されます。しかし Lisp プログラム内の定数とすることに意味が無いいくつかの型には、入力構文がありません。これらのオブジェクトはハッシュ表記 *(hash notation)* でプリントされ、'`#<`'、説明的な文字列 (典型的には型名にオブジェクトの名前を続けたもの)、'`>`'で構成される文字列です。たとえば:

```
(current-buffer)
     ⇒ #<buffer objects-ja.texi>
```

ハッシュ表記は読み取ることができないので、Lisp リーダーは '`#<`'に遭遇すると常にエラー `invalid-read-syntax`をシグナルします。

他の言語では式はテキストであり、これ以外の形式はありません。Lisp では式は第一にまず Lisp オブジェクトであって、オブジェクトの入力構文であるテキストは副次的なものに過ぎません。たいていこの違いを強調する必要はありませんが、このことを心に留めておかないとたまに混乱することがあるでしょう。

インタラクティブに式を評価するとき、Lisp インタープリターは最初にそれのテキスト表現を読み取り、Lisp オブジェクトを生成してからそのオブジェクトを評価します (Chapter 10 [Evaluation], page 133 を参照)。しかし評価と読み取りは別の処理です。読み取りによりテキストにより表現された Lisp オブジェクトを読み取り、Lisp オブジェクトがリターンされます。後でオブジェクトは評価されるかもしれないし、評価されないかもしれません。オブジェクトを読み取るための基本的な関数`read`の説明は、Section 19.3 [Input Functions], page 334 を参照してください。

2.2 特別な読み取り構文

Emacs Lisp では特別なハッシュ表記を通じて多くの特別なオブジェクトと構文を表します。

‘`#<...>`’ 入力構文がないオブジェクトはこのように表される (Section 2.1 [Printed Representation], page 8 を参照)。

‘`##`’ 名前が空文字列であるようなインターン済みシンボルのプリント表現 (Section 2.4.4 [Symbol Type], page 14 を参照)。

‘`#'`’ `function`にたいするショートカット。Section 13.7 [Anonymous Functions], page 220 を参照のこと。

‘`#:`’ 名前が *foo*であるようなインターンされていないシンボルのプリント表現は ‘`#:foo`’ (Section 2.4.4 [Symbol Type], page 14 を参照)。

‘`#N`’ 循環構造のプリント時には構造が自身をループバックすることを表すためにこの構文が使用される。ここで ‘`N`’リストの開始カウント。

```
(let ((a (list 1)))
  (setcdr a a))
=> (1 . #0)
```

‘`#N=`’

‘`#N#`’ ‘`#N=`’はオブジェクト名、‘`#N#`’はそのオブジェクトを表すので、これらはコピーではなく同一オブジェクトになる (Section 2.6 [Circular Objects], page 29 を参照)。

‘`#@N`’ 次の ‘`N`’文字をスキップする (Section 2.3 [Comments], page 10 を参照)。

‘`#xN`’ 16 進表現の ‘`N`’ (‘`#x2a`’)。

‘`#oN`’ 8 進表現の ‘`N`’ (‘`#o52`’)。

‘`#bN`’ 2 進表現の ‘`N`’ (‘`#b101010`’)。

‘`#(...)`’ 文字列のテキストプロパティ(Section 2.4.8.4 [Text Props and Strings], page 21 を参照)。

‘`#^`’ 文字テーブル (Section 2.4.10 [Char-Table Type], page 21 を参照)。

‘`#s(hash-table ...)`’
ハッシュテーブル (Section 2.4.12 [Hash Table Type], page 22 を参照)。

‘`?C`’ 文字 (Section 2.4.3.1 [Basic Char Syntax], page 11 を参照)。

‘`#$`’ バイトコンパイル済みファイルのカレントファイル名 (Section 17.3 [Docs and Compilation], page 288 を参照)。これは Emacs Lisp ソースファイルで使用するためのものではない。

‘`#@N`’ 次の ‘`N`’文字をスキップする (Section 2.3 [Comments], page 10 を参照)。バイトコンパイル済みファイル内で使用されるものであって、Emacs Lisp ソースファイル内で使用するためのものではない。

2.3 コメント

コメント *(comment)* はプログラム中に記述されたテキストであり、そのプログラムを読む人間ためだけに存在するものであって、プログラムの意味には何の影響ももちません。Lisp ではそれが文字列や文字定数にある場合をのぞき、エスケープされていないセミコロン (‘`;`’) でコメントが開始されます。行の終端までがコメントになります。Lisp リーダーはコメントを破棄します。コメントは Lisp システム内でプログラムを表す Lisp オブジェクトの一部にはなりません。

‘`#@`*count*’構成は、次の *count*個の文字をスキップします。これはプログラムにより生成されたバイナリーデータを含むコメントにたいして有用です。Emacs Lisp バイトコンパイラーは出力ファイルにこれを使用します (Chapter 17 [Byte Compilation], page 285 を参照)。しかしソースファイル用ではありません。

コメントのフォーマットにたいする慣例は、Section D.7 [Comment Tips], page 1171 を参照してください。

2.4 プログラミングの型

Emacs Lisp には 2 種類の一般的な型があります。1 つは Lisp プログラミングに関わるもので、もう 1 つは編集に関わるものです。前者はさまざまな形で多くの Lisp 実装に存在します。後者は Emacs Lisp に固有です。

2.4.1 整数型

内部には *fixnums* と呼ばれる小さい整数、および *bignums* という大きい整数という 2 種類の整数が存在します。

fixnum の値の範囲はマシンに依存します、最小のレンジは −536,870,912 から 536,870,911(30 ビットでは -2^{29} から $2^{29}-1$) しかし多くのマシンはより広い範囲を提供します。

bignum は任意の精度をもつことができます。fixnum のオーバーフローする処理では、かわりに bignum をリターンされます。

`eql`や`=`ですべての数値、fixnum なら `eq`で比較することができます。整数が fixnum か bignum をテストするには `most-negative-fixnum`と `most-positive-fixnum`で比較するか、便利な述語 `fixnump`と `bignump`を任意のオブジェクトに使用できます。

整数にたいする入力構文は、(10 を基数とする) 数字のシーケンスで、オプションで先頭に符号、最後にピリオドがつきます。Lisp インタープリターにより生成されるプリント表現には、先頭の ‘`+`’ や最後の ‘`.`’はありません。

```
-1              ; 整数の −1
1               ; 整数の 1
1.              ; これも整数の 1
+1              ; これも整数の 1
```

詳細は Chapter 3 [Numbers], page 37 を参照してください。

2.4.2 浮動小数点数型

浮動小数点数は、コンピューターにおける科学表記に相当するものです。浮動小数点数を 10 の指数をともなう有理数として考えることができます。正確な有効桁数と可能な指数はマシン固有です。Emacs は値の保存に C データ型の `double`を使用し、内部的には 10 の指数ではなく、2 の指数として記録します。

浮動小数点数のプリント表現には、(後に最低 1 つの数字をともなう) 小数点と、指数のどちらか一方、または両方が必要です。たとえば‘`1500.0`’、‘`+15e2`’、‘`15.0e+2`’、‘`+1500000e-3`’、‘`.15e4`’は、いずれも浮動小数点数の 1500 を記述し、これらはすべて等価です。

詳細は Chapter 3 [Numbers], page 37 を参照してください。

2.4.3 文字型

Emacs Lisp での文字 *(character)* は、整数以外の何者でもありません。言い換えると、文字は文字コードで表現されます。たとえば文字 *A*は、整数の 65 として表現されます。

プログラムで文字を個別に使用するのは稀であり、文字のシーケンスとして構成される文字列 *(strings)* として扱われるのがより一般的です。Section 2.4.8 [String Type], page 19 を参照してください。

文字列やバッファーの中の文字は、現在のところ 0 から 4194303 の範囲 — つまり 22 ビットに制限されています (Section 33.5 [Character Codes], page 856 を参照)。0 から 127 のコードは ASCII コードで、残りは非 ASCIIです (Chapter 33 [Non-ASCII Characters], page 852 を参照)。キーボード入力を表す文字はコントロール (Control)、メタ (Meta)、シフト (Shift) などの修飾キーをエンコードするために、より広い範囲をもちます。

文字から可読なテキスト記述を生成する、メッセージ用の特別な関数が存在します。Section 24.5 [Describing Characters], page 525 を参照してください。

2.4.3.1 基本的な文字構文

文字は実際には整数なので、文字のプリント表現は 10 進数です。文字にたいする入力構文も利用可能ですが、Lisp プログラムでこの方法により文字を記述するのは、明解なプログラミングではありません。文字にたいしては、Emacs Lisp が提供する、特別な入力構文を常に使用するべきです。これらの構文フォーマットはクエスチョンマークで開始されます。

英数字にたいする通常の入力構文は、クエスチョンマークと、その後にその文字を記述します。したがって文字 *A*は‘`?A`’、文字 *B*は‘`?B`’、文字 `a`は‘`?a`’となります。

たとえば:

```
?Q ⇒ 81     ?q ⇒ 113
```

区切り文字 (punctuation characters) にも同じ構文を使用できますが、区切り文字が Lisp で特別な意味をもつ場合には‘`\\`’でクォートしなければなりません。たとえば‘`?\\(`’が開カッコを記述する方法であり、同様に文字が‘`\`’なら、‘`?\\`’のようにクォートするために 2 つ目の‘`\`’を使用しなければなりません。

Ctrl+g(control-g)、バックスペース (backspace)、タブ (tab)、改行 (newline)、垂直タブ (vertical tab)、フォームフィード (formfeed)、スペース (space)、キャリッジリターン (return)、デリート (del)、エスケープ (escape) はそれぞれ‘`?\a`’、‘`?\b`’、‘`?\t`’、‘`?\n`’、‘`?\v`’、‘`?\f`’、‘`?\s`’、‘`?\r`’、‘`?\d`’、‘`?\e`’と表すことができます (後にダッシュのついた‘`?\s`’は違う意味をもつ— これは後続の文字にたいしてスーパーによる修飾を適用する)。したがって、

```
?\a ⇒ 7                 ; control-g、C-g
?\b ⇒ 8                 ; バックスペース、BS、C-h
```

```
?\t ⇒ 9                 ; タブ、TAB、C-i
?\n ⇒ 10                ; 改行、C-j
?\v ⇒ 11                ; 垂直タブ、C-k
?\f ⇒ 12                ; フォームフィード文字、C-l
?\r ⇒ 13                ; キャリッジリターン、RET、C-m
?\e ⇒ 27                ; エスケープ文字、ESC、C-[
?\s ⇒ 32                ; スペース文字、SPC
?\\ ⇒ 92                ; バックスラッシュ文字、\
?\d ⇒ 127               ; デリート文字、DEL
```

バックスラッシュがエスケープ文字の役割を果たすので、これらのバックスラッシュで始まるシーケンスはエスケープシーケンス *(escape sequences)* とも呼ばれます。この用語法は文字 ESCとは関係ありません。‘\s’は文字定数としての使用を意図しており、文字定数の内部では単にスペースを記述します。

エスケープという特別な意味を与えずに、任意の文字の前にバックスラッシュの使用することは許されていて害もありません。したがって ‘?\+’は ‘?+’と等価です。ほとんどの文字の前にバックスラッシュを追加することに理由はありません。しかし文字 ‘()[]\;"’の前にはバックスラッシュを追加しなければならず、Lisp コードを編集する Emacs コマンドが混乱するのを避けるために ‘|'`#.,’の前にもバックスラッシュを追加するべきです。コードを読む人の混乱を避けるために、前に言及したASCII文字と類似した Unicode 文字の前にもバックスラッシュを追加する必要があります。これを奨励するために、‘ ’のような一般的には混乱を招くエスケープされていない文字をハイライトします。スペース、タブ、改行、フォームフィードのような空白文字の前にもバックスラッシュを追加できます。しかしタブやスペース space のような実際の空白文字のかわりに、‘\t’や ‘\s’のような可読性のあるエスケープシーケンスを使用するほうが明解です (スペースを後にともなうバックスラッシュを記述する場合、後続のテキストと区別するために、文字定数の後に余分なスペースを記述すること)。

2.4.3.2 一般的なエスケープ構文

特に重要なコントロール文字にたいする特別なエスケープシーケンスに加えて、Emacs は非 ASCIIテキスト文字の指定に使用できる、何種類かのエスケープ構文を提供します。

1. もしあれば Unicode 名で文字を指定できる。?\N{*NAME*}は *NAME* という名前の Unicode 文字を表す。したがって ‘?\N{LATIN SMALL LETTER A WITH GRAVE}’は?^^c3^^a0と等価であり Unicode 文字の U+00E0 を意味する。名前中のスペースを空白文字 (改行) の非空のシーケンスにして複数行文字列の入力を簡略化できる。
2. Unicode 値で文字を指定できる。?\N{U+*X*}は Unicode コードポイント X (16 進数値) を表す。また?\u*xxxx*と?\U*xxxxxxxx*はそれぞれコードポイント *xxxx*と *xxxxxxxx*を表す *x*は (1 つの 16 進数字)。たとえば?\N{U+E0}、?\u00e0、?\U000000E0は?^^c3^^a0と ‘?\N{LATIN SMALL LETTER A WITH GRAVE}’に等しい。Unicode 標準は ‘U+*10ffff*’以下のコードポイントだけを定義するので、これより大きいコードポイントでは Emacs はエラーをシグナルする。
3. 文字を 16 進の文字コードで指定できる。16 進エスケープシーケンスはバックスラッシュ、‘x’、および 16 進の文字コードにより構成される。したがって ‘?\x41’は文字 A、‘?\x1’は文字 *C*-a、?\xe0は文字^^*c3*^^*a0* (グレイブアクセントつきの a) になる。任意の 16 進数を使用できるので、この方法で任意の文字を表すことができる。
4. 8 進の文字コードにより文字を指定できる。8 進エスケープシーケンスは 3 桁までの 8 進数字をともなうバックスラッシュにより形成される。したがって ‘?\101’は文字 A、‘?\001’は文字 *C*-a、?\002は文字 *C*-bを表す。この方法で指定できるのは 8 進コード 777 までの文字のみ。

これらのエスケープシーケンスは文字列内でも使用されます。Section 2.4.8.2 [Non-ASCII in Strings], page 20 を参照してください。

2.4.3.3 コントロール文字構文

他の入力構文を使用してコントロール文字を表すことができます。これは後にバックスラッシュ、カレット、対応する非コントロール文字 (大文字か小文字) をともなうクエスチョンマークから構成されます。たとえば '`?\^I`' と '`?\^i`' はどちらも、値が 9 である文字 *`C-i`* の有効な入力構文です。

'`^`' のかわりに '`C-`' を使用することもできます。したがって '`?\C-i`' は '`?\^I`' や '`?\^i`' と等価です。

```
?\^I ⇒ 9      ?\C-I ⇒ 9
```

文字列やバッファーの中では ASCII のコントロール文字だけが許されますが、キーボード入力にたいしては '`C-`' により任意の文字をコントロール文字にすることができます。これらの非 ASCII のコントロール文字にたいするコントロール文字には非コントロール文字にたいするコードと同様に、2^{26} ビットが含まれます。通常のテキスト端末には非 ASCII コントロール文字を生成する方法がありませんが、X やその他のウィンドウシステムを使用すれば簡単に生成することができます。

歴史的な理由により、Emacs は `DEL` 文字を *`?`* のコントロール文字として扱います:

```
?\^? ⇒ 127      ?\C-? ⇒ 127
```

結果として、X では有意な入力文字である *`Control-?`* 文字を、'`\C-`' を使用して表現することは今のところできません。さまざまな Lisp ファイルがこの方法で `DEL` を参照するので、これを変更するのは簡単ではないのです。

コントロール文字の表現はファイルや文字列内で見ることができますが、わたしたちは '`^`' 構文を推奨します。キーボード入力にたいするコントロール文字に好ましいのは、'`C-`' 構文です。どちらを使用するかはプログラムの意味に影響しませんが、プログラムを読む人の理解を助けるでしょう。

2.4.3.4 メタ文字構文

メタ文字 *(meta character)* とは、`META` 修飾キーとともにタイプされた文字です。そのような文字を表す整数には 2^{27} のビットがセットされています。基本的な文字コードの広い範囲を利用可能にするために、メタやその他の修飾にたいしては上位ビットを使用します。

文字列では、メタ文字を示す ASCII 文字に、2^7 ビットが付加されます。したがって文字列に含めることができるメタ文字のコードは 1 から 255 の範囲となり、メタ文字は通常の ASCII 文字のメタ修飾されたバージョンとなります。文字列内での `META` 処理の詳細については、Section 21.7.15 [Strings of Events], page 402 を参照してください。

メタ文字の入力構文には '`\M-`' を使用します。たとえば '`?\M-A`' は *`M-A`* を意味します。8 進文字コード (以下参照) や、'`\C-`'、その他の文字にたいする他の構文とともに '`\M-`' を使用できます。したがって、*`M-A`* は '`?\M-A`' や '`?\M-\101`' と記述できます。同様に *`C-M-b`* は '`?\M-\C-b`'、'`?\C-\M-b`'、'`?\M-\002`' と記述することができます。

2.4.3.5 その他の文字修飾ビット

グラフィック文字 (graphic character) の case は文字コードで示されます。たとえば ASCII では、文字 '`a`' と文字 '`A`' は区別されます。しかし ASCII にはコントロール文字が大文字なのか小文字なのかを表現する方法がありません。コントロール文字がタイプされたときシフトキーが使用されたかを示すために、Emacs は 2^{25} のビットを使用します。この区別は X 端末や、その他の特別な端末を使用しているときだけ可能です。通常のテキスト端末は、これらの違いを報告しません。シフトをあらわすビットのための Lisp 構文は '`\S-`' です。したがって '`?\C-\S-o`' や '`?\C-\S-O`' は、Shift+Ctrl+o 文字を表します。

X ウィンドウシステムは文字にセットするために、他にも 3 つ修飾ビット — ハイパー *(hyper)*、スーパー *(super)*、アルト *(alt)* を定義します。これらのビットにたいする構文は、'`\H-`'、'`\s-`'、'`\A-`' です (これらのプレフィクスでは、case は意味がある)。したがって '`?\H-\M-\A-x`'は *`Alt-Hyper-Meta-x`*を表します ('`-`'が後にない '`\s`'はスペース文字を表すことに注意)。数値としてはビット値 2^{22} はアルト、2^{23} はスーパー、2^{24} はハイパーです。

2.4.4 シンボル型

GNU Emacs Lisp でのシンボル *(symbol)* とは、名前をもつオブジェクトです。シンボル名は、そのシンボルのプリント表現としての役割があります。Lisp の通常の使用では、1 つの obarray(Section 9.3 [Creating Symbols], page 126 を参照) により、シンボル名は一意です — 2 つのシンボルが同じ名前をもつことはありません。

シンボルは変数や関数名としての役割、プロパティーリストを保持する役割をもつことができます。データ構造内にそのようなシンボルが存在することが確実に認識できるように、他のすべての Lisp オブジェクトから区別するためだけの役割をもつ場合もあります。与えられたコンテキストにおいて、通常はこれらのうちの 1 つの使用だけが意図されます。しかし 3 つすべての方法で、1 つのシンボルを独立して使用することもできます。

名前がコロン ('`:`') で始まるシンボルはキーワードシンボル *(keyword symbol)* と呼ばれます。これらのシンボルは自動的に定数として振る舞い、通常は未知のシンボルといくつかの特定の候補を比較することだけに使用されます。Section 12.2 [Constant Variables], page 174 を参照してください。

シンボル名にはどんな文字でも含めることができます。ほとんどのシンボル名は英字、数字、'`-+=*/`' などの区切り文字で記述されます。このような名前には特別な区切り文字は必要ありません。名前が数字のように見えない限り、名前にはどのような文字も使用できます (名前が数字のように見える場合は、名前の先頭に '`\`'を記述して強制的にシンボルとして解釈させる)。文字 '`_~!@$%^&:<>{}?`'はあまり使用されませんが、これらも特別な句読点文字を必要としません。他の文字も、バックスラッシュでエスケープすることにより、シンボル名に含めることができます。しかし文字列内でのバックスラッシュの使用とは対照的に、シンボル名でのバックスラッシュは、バックスラッシュの後の 1 文字をエスケープするだけです。たとえば文字列内では、'`\t`'はタブ文字を表します。しかしシンボル名の中では、'`\t`'は英字 '`t`'をクォートするに過ぎません。名前にタブ文字をもつシンボルを記述するには、(バックスラッシュを前置した) 実際のタブを使用しなければなりません。しかし、そのようなことを行なうことは稀です。

> **Common Lisp** に関する注意:Common Lisp では明示的にエスケープされない限り、小文字は常に大文字に "フォールド (folded)" される。Emacs Lisp では大文字と小文字は区別される。

以下はシンボル名の例です。4 つ目の例の中の '`+`'は、シンボルが数字として読み取られるのを防ぐためにエスケープされていることに注意してください。6 つ目の例では、名前の残りの部分により数字としては不正なのでエスケープの必要はありません。

```
foo                   ; 'foo'という名前のシンボル
FOO                   ; 'foo'とは別の、'FOO'という名前のシンボル
1+                    ; '1+'という名前のシンボル
                      ;    (整数の '+1'ではない)
\+1                   ; '+1'という名前のシンボル
                      ;    (判読しにくい名前)
\(*\ 1\ 2\)           ; '(* 1 2)'という名前のシンボル (悪い名前)
+-*/_~!@$%^&=:<>{}    ; '+-*/_~!@$%^&=:<>{}'という名前のシンボル
                      ;    これらの文字のエスケープは不要
```

シンボル名がプリント表現としての役割をもつというルールの例外として、‘##’があります。これは、名前が空文字列の intern されたシンボルのプリント表現です。さらに ‘#:*foo*’は、intern されていない *foo*という名前のシンボルにたいするプリント表現です (通常、Lisp リーダーはすべてのシンボルを intern する。Section 9.3 [Creating Symbols], page 126 を参照されたい)。

2.4.5 シーケンス型

シーケンス *(sequence)* とは、要素の順序セットを表現する Lisp オブジェクトです。Emacs Lisp には 2 種類のシーケンス — リスト *(lists)* と配列 *(arrays)* があります。

リストはもっとも一般的に使用されるシーケンスです。リストは任意の型の要素を保持でき、要素の追加と削除により簡単に長さを変更できます。リストについては、次のサブセクションを参照してください。

配列は固定長のシーケンスです。配列はさらに文字列 (strings)、ベクター (vectors)、文字テーブル (char-tables)、ブールベクター (bool-vectors) に細分されます。ベクターは任意の型の要素を保持できますが、文字列の要素は文字でなければならず、ブールベクターの要素は `t`か `nil`でなければなりません。文字テーブルはベクターと似ていますが、有効な文字によりインデックスづけされる点が異なります。文字列内の文字は、バッファー内の文字のようにテキストプロパティーをもつことができます (Section 32.19 [Text Properties], page 813 を参照)。しかしベクターはその要素が文字のときでも、テキストプロパティーをサポートしません。

リスト、文字列、およびその他の配列型も、重要な類似点を共有します。たとえば、それらはすべて長さ *l*をもち、要素は 0 から *l*−1 でインデックスづけされます。いくつかの関数はシーケンス関数と呼ばれ、これらは任意の種類のシーケンスを許容します。たとえば、関数 `length`は、任意の種類のシーケンスの長さを報告します。Chapter 6 [Sequences Arrays Vectors], page 95 を参照してください。

シーケンスは読み取りにより常に新たに作成されるやめ、同じシーケンスを 2 回読み取るのは一般的に不可能です。シーケンスにたいする入力構文を 2 回読み取った場合には、内容が等しい 2 つのシーケンスを得ます。これには 1 つ例外があります。空リスト `()`は、常に同じオブジェクト `nil`を表します。

2.4.6 コンスセルとリスト型

コンスセル *(cons cell)* は CAR スロット、CDR スロットと呼ばれる 2 つのスロットから構成されるオブジェクトです。それぞれのスロットはには、任意の Lisp オブジェクトを保持できます。そのとき CAR スロットに保持されるオブジェクトが何であれ、わたしたちは “このコンスセルの CAR” のような言い方をします。これは CDR の場合も同様です。

リスト *(list)* はコンスセルの連続するシリーズで、各コンスセルの CDR スロットは次のコンスセル、または空リストを保持します。空リストは実際にはシンボル `nil`です。詳細については、Chapter 5 [Lists], page 71 を参照してください。ほとんどのコンスセルはリストの一部として使用されるので、わたしたちはコンスセルにより構成される任意の構造を、リスト構造 *(list structure)* という用語で参照します。

> C プログラマーにたいする注意: Lisp のリストはコンスセルにより構築されるリンクリスト *(linked list)* として機能する。Lisp ではポインターは暗黙的なので、コンスセルのスロットが値を “保持 (hold)” するのか、それとも値を “指す (point)” のかは区別しない。

コンスセルは Lisp の中核なので、コンスセルではないオブジェクトにたいする用語もあります。これらのオブジェクトはアトム *(atoms)* と呼ばれます。

リストにたいする入力構文とプリント表現は同じで左カッコ、任意の数の要素、右カコから構成されます。以下はリストの例です:

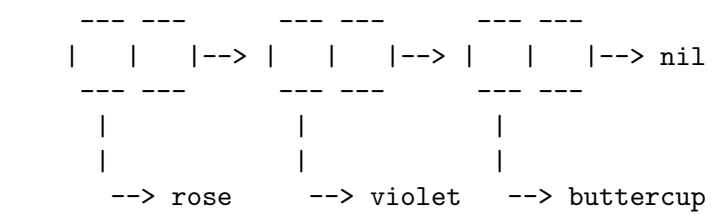
```
(A 2 "A")            ; 3 要素のリスト
()                   ; 要素がないリスト (空リスト)
nil                  ; 要素がないリスト (空リスト)
("A ()")             ; 1 要素のリスト: 文字列"A ()"
(A ())               ; 2 要素のリスト: Aと空リスト
(A nil)              ; 同上
((A B C))            ; 1 要素のリスト
                     ;    (この要素は、3 要素のリスト)
```

読み取りではカッコの内側はリストの要素になります。つまりコンスセルは各要素から作成されます。コンスセルの CAR スロットは要素を保持し、CDR スロットはリスト内の次のコンスセル (このコンスセルはリスト内の次の要素をする) を参照します。最後のコンスセルの CDR スロットは `nil`を保持するようにセットされます。

CAR や CDR という名称は Lisp の歴史に由来します。オリジナルの Lisp 実装は IBM 704 コンピューターで実行されていました。ワードを 2 つの部分、つまり “address” と呼ばれる部分と、“decrement” と呼ばれる部分に分割していて、その際 CAR は address 部から内容を取り出す命令で、CDR は decrement 部から内容を取り出す命令でした。これとは対照的に “cons cells” は、これらを作成する関数 `cons`から命名されました。この関数は関数の目的、すなわちセルを作る (construction of cells) という目的から命名されました。

2.4.6.1 ボックスダイアグラムによるリストの描写

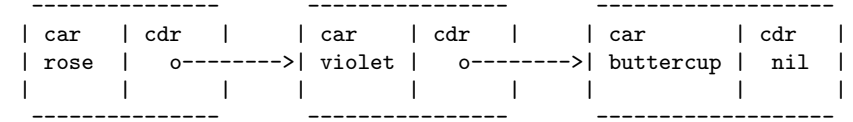
コンスセルを表現するドミノのような 1 対のボックスによる図で、リストを説明することができます (Lisp リーダーがこのような図を読み取ることはできない。人間とコンピューターが理解できるテキスト表記と異なり、ボックス図は人間だけが理解できる)。この図は 3 要素のリスト `(rose violet buttercup)`を表したものです:



この図では、ボックスは任意の Lisp オブジェクトへの参照を保持できるスロットを表します。ボックスのペアーはコンスセルを表します。矢印は Lisp オブジェクト (アトム、または他のコンスセル) への参照を表します。

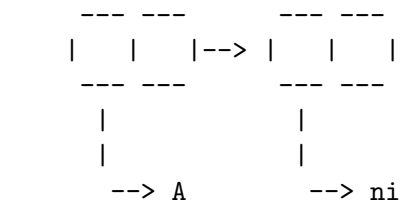
この例では、1 番目のボックスは 1 番目のコンスセルで、それの CAR は `rose`(シンボル) を参照または保持します。2 番目のボックスは 1 番目のコンスセルの CDR を保持し、次のボックスペアすなわち 2 番目のコンスセルを参照します。2 番目のコンスセルの CAR は `violet`で、CDR は 3 番目のコンスセルです。(最後の)3 番目のコンスセルの CDR は `nil`です。

同じリスト `(rose violet buttercup)`を、違うやり方で描いた別の図で表してみましょう:

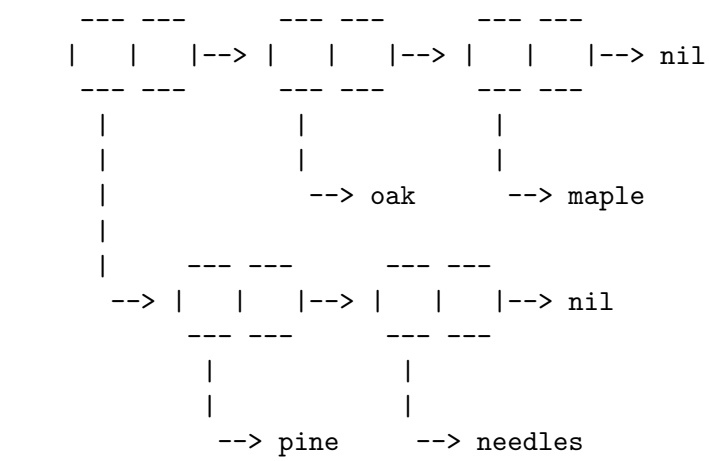


要素がないリストは空リスト *(empty list)* で、これはシンボル `nil` と同じです。言い換えると `nil` はシンボルであり、かつリストでもあります。

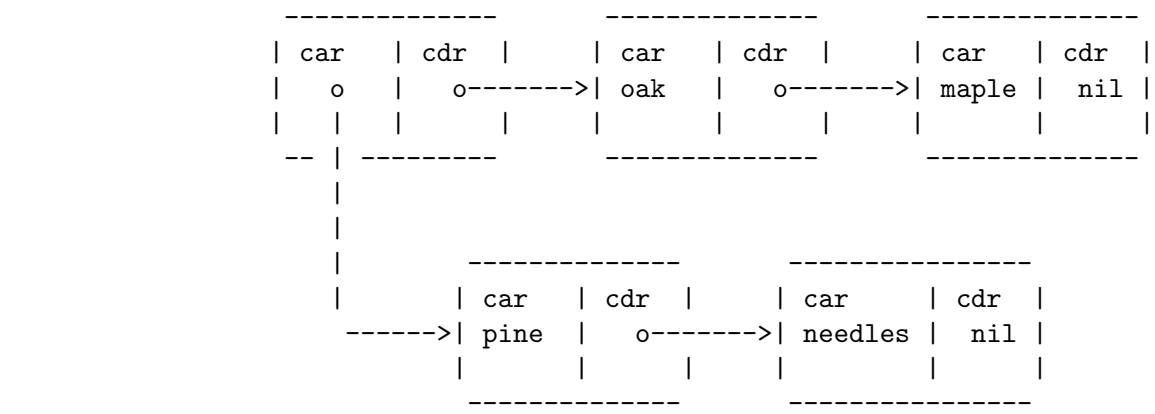
以下はリスト `(A ())`、または等価な `(A nil)` をボックスと矢印で描いたものです:



以下はもっと複雑な例です。これは 1 番目の要素が 2 要素のリストであるような、3 要素のリスト `((pine needles) oak maple)` を表します:



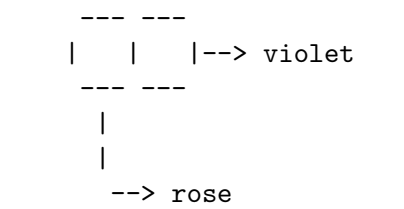
同じリストを 2 番目のボックス表記で表すと、以下のようになります:



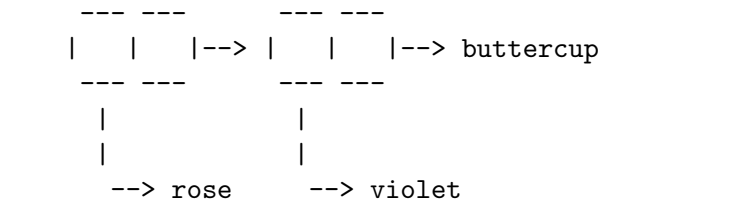
2.4.6.2 ドットペア表記

ドットペア表記 *(dotted pair notation)* は、CAR と CDR が明示的に表されたコンスセルの一般的な構文です。この構文では `(a . b)` が CAR がオブジェクト *a*、CDR がオブジェクト *b* という意味になります。CDR がリストである必要がないので、ドットペア表記はより一般的なリスト構文です。しかしリスト構文が機能するような場合には、より扱いにくくなります。ドットペア表記では、リスト '`(1 2 3)`' は '`(1 . (2 . (3 . nil)))`' と記述されます。`nil` で終端されたリストにたいしては、どちらの表記法も使用できますが、リスト表記の方が通常は明解で便利です。リストをプリントする場合には、コンスセルの CDR がリストでないときだけドットペア表記が使用されます。

以下はボックスを使用してドットペア表記を表した例です。これはペア (rose . violet)を表します:

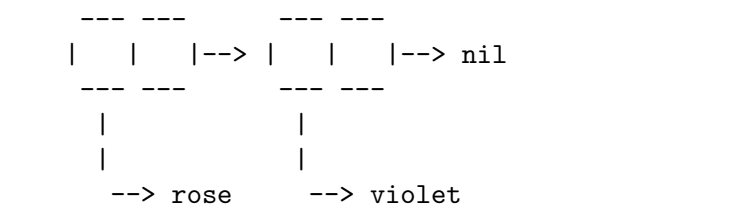


最後の CDR が非 nilのコンスセルのチェーンを表すので、ドットペア表記とリスト表記を組み合わせることができます。リストの最後の要素の後にドットを記述して、その後に最後のコンスセルの CDR を記述します。たとえば (rose violet . buttercup)は、(rose . (violet . buttercup))と等価です。オブジェクトは以下のようになります:



構文 (rose . violet . buttercup)は無効です。なぜならこれは何も意味していないからです。何か意味があるとしても、violetのために CDR がすでに使用されているコンスセルの CDR に、buttercupを置くということになります。

リスト (rose violet)は (rose . (violet))と等価であり、以下のようになります:



同様に 3 要素のリスト (rose violet buttercup)は、(rose . (violet . (buttercup)))と等価です。

2.4.6.3 連想リスト型

連想リスト *(association list)* または *alist* は、要素がコンスセルであるように特別に構成されたリストです。各要素においては、CAR がキー *(key)* で、CDR が連想値 *(associated value)* であると考えます (連想値が CDR の CAR に保存される場合もある)。リストの先頭への連想値の追加と削除は簡単なので、連想リストはスタック (stack) にしばしば使用されます。

たとえば、

```
(setq alist-of-colors
      '((rose . red) (lily . white) (buttercup . yellow)))
```

これは変数 alist-of-colorsに 3 要素の alist をセットします。最初の要素では、roseがキーで redが値になります。

alist と alist 関数についての詳細な説明は Section 5.8 [Association Lists], page 88 を参照してください。(多くのキーの操作をより高速に行なう) テーブルを照合する他の手段については Chapter 8 [Hash Tables], page 118 を参照してください。

2.4.7 配列型

配列 *(array)* は、他の Lisp オブジェクトを保持または参照する任意の数のスロットから構成され、メモリーの連続ブロックに配列されます。配列の任意の要素へのアクセス時間は大体同じです。対照的にリストの要素にたいするアクセスは、リスト内でのその要素の位置に比例した時間を要します (リストの最後の要素にアクセスするにはリストの最初の要素にアクセスするより長い時間が必要)。

Emacs は文字列 (strings)、ベクター (vectors)、ブールベクター (bool-vectors)、文字テーブル (char-tables) という 4 種の配列を定義します。

文字列は文字の配列であり、ベクターは任意のオブジェクトの配列です。ブールベクターは `t`か`nil`だけを保持できます。この種の配列はシステムアーキテクチャー制限と利用可能なメモリーにしたがったもっとも大きい fixnum までの任意の長さをもつことができます。文字テーブルは、任意の有効な文字コードによりインデックスづけされる疎な配列であり、任意のオブジェクトを保持することができます。

配列の最初の要素はインデックス 0、2 番目の要素はインデックス 1、... となります。これは *0* 基準 *(zero-origin)* のインデックスづけと呼ばれます。たとえば、4 要素の配列はインデックス 0、1、2、3 をもちます。利用できる最大のインデックス値は、配列の長さより 1 つ小さくなります。▼一度配列が作成されると、長さは固定されます。

Emacs Lisp のすべての配列は、1 次元です (他のほとんどのプログラミング言語は多次元配列をサポートするが、これらは必須ではない。ネストされた 1 次元配列により同じ効果を得ることが可能)。各種の配列は独自の入力構文をもちます。詳細は以降のセクションを参照してください。

配列型はシーケンス型のサブセットであり文字列型、ベクター型、ブールベクター型、文字テーブル型が含まれます。

2.4.8 文字列型

文字列 *(string)* とは文字の配列です。Emacs がテキストエディターであることから予想できるように、文字列はたとえば Lisp シンボルの名前、ユーザーへのメッセージ、バッファーから抽出されたテキストの表現など多くの目的のために使用されます。Lisp の文字列は定数です。文字列を評価すると、それと同じ文字列がリターンされます。

文字列を操作する関数については Chapter 4 [Strings and Characters], page 51 を参照してください。

2.4.8.1 文字列の構文

文字列にたいする入力構文は`"like this"`のようにダブルクォート、任意個数の文字、もう 1 つのダブルクォートから構成されます。文字列内にダブルクォートを含める場合は、それの前にバックスラッシュを記述します。したがって`"\""`は 1 つのダブルクォート文字だけを含む文字列です。同様にバックスラッシュを含める場合は、`"this \\ is a single embedded backslash"`のように、それの前にもう 1 つのバックスラッシュを記述します。

文字列にたいする入力構文では、改行 (newline) は特別ではありません。ダブルクォートの間に改行を記述すれば、その改行は文字列内の文字となります。しかしエスケープされた改行— 前に '\'をともなう改行 — は文字列の一部とはなりません。同様にエスケープされたスペース '\ ' も無視されます。

```
"It is useful to include newlines
in documentation strings,
but the newline is \
ignored if escaped."
```

```
     ⇒ "It is useful to include newlines
in documentation strings,
but the newline is ignored if escaped."
```

2.4.8.2 文字列内の非 **ASCII**文字

Emacd の文字列内の非 ASCII文字にたいしては 2 つのテキスト表現 — マルチバイト (multibyte) とユニバイト (unibyte) があります (Section 33.1 [Text Representations], page 852 を参照)。大まかに言うとユニバイト文字列には raw(生) バイトが保存され、マルチバイト文字列には人間が読めるテキストが保存されます。ユニバイト文字列内の各文字はバイトであり、値は 0 から 255 となります。対照的にマルチバイト文字列内の各文字は、0 から 4194303 の値をもつかもしれません (Section 2.4.3 [Character Type], page 11 を参照)。いずれも 127 より上の文字は非 ASCIIです。

文字をリテラルとして記述することにより、文字列に非 ASCII文字を含めることができます。マルチバイトのバッファーや文字列、あるいはマルチバイトとして visit されたファイル等、マルチバイトのソースから文字列定数を読み込む場合、Emacs は非 ASCII文字をマルチバイト文字として読み取り、その文字列を自動的にマルチバイト文字列にします。ユニバイトのソースから文字列定数を読み込む場合、Emacs は非 ASCII文字をユニバイト文字として読み取り、その文字列をユニバイト文字列にします。

マルチバイト文字列内にリテラルとして文字を記述するかわりに、エスケープシーケンスを使用して文字コードとして記述できます。エスケープシーケンスについての詳細は、Section 2.4.3.2 [General Escape Syntax], page 12 を参照してください。

文字列定数内で Unicode スタイルのエスケープシーケンス '`\uNNNN`'または '`\UOONNNNNN`'を使用する場合、(たとえ ASCII文字であっても)Emacs は自動的に文字列をマルチバイトとみなします。

文字列定数内で 16 進エスケープシーケンス ('`\x`*n*') と 8 進エスケープシーケンス ('`\`*n*') を使用することもできます。しかし注意してください**:** 文字列定数が 16 進または 8 進のエスケープシーケンスを含み、それらのエスケープシーケンスすべてがユニバイト文字 (256 より小) を指定していて、その文字列内に他にリテラルの非 ASCII文字または Unicode スタイルのエスケープシーケンスが存在しない場合、Emacs は自動的に文字列をユニバイト文字列とみなします。つまり文字列内のすべての非 ASCII文字は 8 ビットの raw バイトとみなされます。

16 進および 8 進のエスケープシーケンスでは、エスケープされた文字コードに可変個の数字が含まれるかもしれないので、それに続く文字で 16 進および 8 進として有効ではない最初の文字は、そのエスケープシーケンスを終了させます。文字列内の次の文字が 16 進または 8 進として解釈できる文字の場合は、'`\ `'(バックスラッシュとスペース) を記述して、エスケープシーケンスを終了できます。たとえば '`\xe0\ `' は grave accent つきの '`a`'という 1 文字を表します。文字列内の '`\ `' はバックスラッシュ改行と同様です。これは文字列内の文字とはなりませんが、先行する 16 進エスケープを終了します。

2.4.8.3 文字列内の非プリント文字

リテラル文字と同様に、文字列定数内でバックスラッシュによるエスケープシーケンスを使用できます (ただし文字定数を開始するクエスチョンマークは使用しない)。たとえば非プリント文字のタブと *C*`-a`を含む文字列は、`"\t, \C-a"`のように、それらの間にカンマとスペースを記述します。文字にたいする入力構文については Section 2.4.3 [Character Type], page 11 を参照してください。

しかしバックスラッシュによるエスケープシーケンスとともに記述できるすべての文字が、文字列内で有効というわけではありません。文字列が保持できるコントロール文字は ASCIIコントロール文字だけです。ASCIIコントロール文字では、文字列の case は区別されません。

正確に言うと、文字列はメタ文字を保持できません。しかし文字列がキーシーケンスとして使用される場合には、文字列内でメタ修飾された ASCII文字を表現するための方法を提供する特別な慣習があります。文字列定数内でメタ文字を示すために '`\M-`'構文を使用した場合、これは文字列内の文字の 2^7 のビットをセットします。その文字列が `define-key`または `lookup-key`で使用される場合、この数字コードは等価なメタ文字に変換されます。Section 2.4.3 [Character Type], page 11 を参照してください。

文字列はハイパー (hyper)、スーパー (super)、アルト (alt) で修飾された文字を保持できません。

2.4.8.4 文字列内のテキストプロパティ

文字列にはその文字自身に加えて、文字のプロパティーも保持することができます。これにより特別なことをしなくても、文字列とバッファーとの間でテキストをコピーするプログラムが、テキストプロパティーをコピーすることが可能になります。テキストプロパティーが何を意味するかについてはSection 32.19 [Text Properties], page 813 を参照してください。テキストプロパティーをもつ文字列は、特別な入力構文とプリント構文を使用します。

```
#("characters" property-data...)
```

ここで *property-data*は,3 個でグループ化された 0 個以上の要素から構成されます:

```
beg end plist
```

要素 *beg*および *end*は整数で、文字列内のインデックスの範囲を指定します。*plist*はその範囲にたいするプロパティーリストです。たとえば、

```
#("foo bar" 0 3 (face bold) 3 4 nil 4 7 (face italic))
```

これはテキスト内容が '`foo bar`'で、最初の 3 文字は `face`プロパティーに値 `bold`をもち、最後の 3 文字は `face`プロパティーに値 `italic`をもつことを表します (4 番目の文字にはテキストプロパティーはないので、プロパティーリストは `nil`。実際には範囲の中の指定されていない文字はデフォルトではプロパティーをもたないので、範囲のプロパティーリストを `nil`と指定する必要ない)。

2.4.9 ベクター型

ベクター *(vector)* は任意の型の要素からなる 1 次元の配列です。ベクター内の任意の要素へのアクセスに要す時間は一定です (リストの場合では要素へのアクセスに要す時間は、リストの先頭からその要素までの距離に比例する)。

ベクターのプリント表現は左角カッコ (left square bracket)、要素、右角カッコ (right square bracket) から構成されます。これは入力構文でもあります。数字や文字列と同様にベクターは評価において定数と判断されます。

```
[1 "two" (three)]       ; 3 要素のベクター
     ⇒ [1 "two" (three)]
```

ベクターに作用する関数についてはSection 6.4 [Vectors], page 109 を参照してください。

2.4.10 文字テーブル型

文字テーブル *(char-table)* は任意の型の要素をもつ 1 次元の配列であり、文字コードによりインデックスづけされます。文字テーブルは、文字コードに情報を割り当てることを必要とする多くの処理を簡単にするための、特別な追加の機能をもちます— たとえば文字テーブルは継承する親、デフォルト値、特別な目的のために使用する余分なスロットをいくつかもつことができます。文字テーブルは文字セット全体にたいして 1 つの値を指定することもできます。

文字テーブルのプリント表現はベクターと似ていますが、最初に余分な '`#^`'があります [1]。

[1] 副文字テーブル (sub-char-tables) に使用される '`#^^`'を目にすることがあるかもしれません。

文字テーブルを操作する特別な関数については Section 6.6 [Char-Tables], page 110 を参照してください。文字テーブルの使用には以下が含まれます:

- case テーブル (Section 4.10 [Case Tables], page 69 を参照)。
- 文字カテゴリーテーブル (Section 35.8 [Categories], page 918 を参照)。
- ディスプレーテーブル (Section 39.22.2 [Display Tables], page 1079 を参照)。
- 構文テーブル (Chapter 35 [Syntax Tables], page 906 を参照)。

2.4.11 ブールベクター型

ブールベクター *(bool-vector)* は、要素が `t`か `nil`のいずれかでなければならない 1 次元の配列です。

ブールベクターのプリント表現は文字列と似ていますが、後に長さを記述した '`#&`'で始まります。これに続く文字列定数は、ビットマップとして実際に内容を指定するブールベクターです— 文字列定数内のそれぞれの "文字" は 8 ビットを含み、これはブールベクターの次の 8 要素を指定します (1 は `t`、0 は `nil`です)。文字の最下位ビットブールベクターの最下位のインデックスに対応します。

```
(make-bool-vector 3 t)
     ⇒ #&3"^G"
(make-bool-vector 3 nil)
     ⇒ #&3"^@"
```

'`C-g`'の 2 進コードは 111、'`C-@`'はコード 0 の文字なのでこの結果は理にかなっています。

長さが 8 の倍数でなければプリント表現には余分な要素が表示されますが、これらの余分な要素に意味はありません。たとえば以下の例では、最初の 3 ビットだけが使用されるので 2 つのブールベクターは等価です:

```
(equal #&3"\377" #&3"\007")
     ⇒ t
```

2.4.12 ハッシュテーブル型

ハッシュテーブルは非常に高速な照合テーブルの一種で、キーを対応する値にマップする alist と似ていますがより高速です。ハッシュテーブルのプリント表現では、以下のようにハッシュテーブルのプロパティーと内容を指定します:

```
(make-hash-table)
     ⇒ #s(hash-table size 65 test eql rehash-size 1.5
                     rehash-threshold 0.8125 data ())
```

ハッシュテーブルについての詳細は Chapter 8 [Hash Tables], page 118 を参照してください。

2.4.13 関数型

他のプログラミング言語の関数と同様、Lisp 関数は実行可能なコードです。他の言語と異なり、Lisp の関数は Lisp オブジェクトでもあります。Lisp のコンパイルされていない関数はラムダ式— つまり 1 番目の要素がシンボル `lambda`であるリストです (Section 13.2 [Lambda Expressions], page 211 を参照)。

ほとんどのプログラミング言語では名前のない関数はありません。Lisp では関数に本質的な名前はありません。名前がなくてもラムダ式を関数として呼び出すことができます。これを強調するために、わたしたちはこれを無名関数 *(anonymous function)* とも呼びます (Section 13.7 [Anonymous Functions], page 220 を参照)。Lisp の名前つき関数は関数セルに有効な関数がセットされた単なるシンボルです (Section 13.4 [Defining Functions], page 215 を参照)。

ほとんどの場合、関数は Lisp プログラム内の Lisp 式の名前が記述されたところで呼び出されます。しかし実行時に関数オブジェクトを構築または取得してから、プリミティブ関数 `funcall`および`apply`により呼び出すことができます。Section 13.5 [Calling Functions], page 217 を参照してください。

2.4.14 マクロ型

Lisp マクロ *(Lisp macro)* は Lisp 言語を拡張するユーザー定義の構成です。これはオブジェクトとしてではなく関数のように表現されますが、引数の渡し方の意味が異なります。Lisp マクロの形式はリストです。これは最初の要素が `macro`で、CDR が Lisp 関数オブジェクト (`lambda`シンボルを含む) であるようなリストです。

Lisp マクロオブジェクトは通常、ビルトインの `defmacro`関数で定義されますが、`macro`で始まる任意のリストも Emacs にとってはマクロです。マクロを記述する方法の説明は Chapter 14 [Macros], page 242 を参照してください。

警告: Lisp マクロとキーボードマクロ (Section 21.16 [Keyboard Macros], page 420 を参照) は完全に別の物である。修飾なしで “マクロ” という単語を使用したときは、キーボードマクロではなく Lisp マクロのことを指す。

2.4.15 プリミティブ関数型

プリミティブ関数 *(primitive function)* とは、C プログラミング言語で記述された Lisp から呼び出せる関数です。プリミティブ関数は *subrs* やビルトイン関数 *(built-in functions)* とも呼ばれます (単語 “subr” は “サブルーチン (subroutine)” が由来)。ほとんどのプリミティブ関数ハ、呼び出されたときニすべての引数を評価します。すべての引数を評価しないプリミティブ関数はスペシャルフォーム *(special form)* と呼ばれます (Section 10.1.7 [Special Forms], page 137 を参照)。

呼び出す側からすれば、その関数がプリミティブ関数かどうかは問題になりません。しかしプリミティブ関数を Lisp で記述された関数で再定義した場合に問題になります。理由はそのプリミティブ関数が C コードから直接呼び出されているかもしれないからです。Lisp から再定義した関数を呼び出すと新しい定義を使用するでしょうが、C コードから呼び出すとビルトインの定義が使用されるでしょう。したがって、プリミティブ関数の再定義はしないでください。

関数 *(function)* という用語で、Lisp や C で記述されたすべての Emacs 関数を参照します。Lisp で記述された関数についての情報は Section 2.4.13 [Function Type], page 22 を参照してください。

プリミティブ関数に入力構文はなく、サブルーチン名とともにハッシュ表記でプリントします。

```
(symbol-function 'car)          ; そのシンボルの関数セルに
                                ;       アクセスする
     ⇒ #<subr car>
(subrp (symbol-function 'car))  ; これはプリミティブ関数?
     ⇒ t                        ;     そのとおり
```

2.4.16 バイトコード関数型

バイトコード関数オブジェクト *(byte-code function objects)* は、Lisp コードをバイトコンパイルすることにより生成されます (Chapter 17 [Byte Compilation], page 285 を参照)。バイトコード関数オブジェクトは、内部的にはベクターによく似ています。しかしバイトコード関数オブジェクトが関数呼び出しのように見える場合、評価プロセスによりこのデータ型は特別に処理されます。Section 17.7 [Byte-Code Objects], page 291 を参照してください。

バイトコード関数オブジェクトのプリント表現と入力構文はベクターのものと似ていますが、開き角カッコ ‘[’の前に ‘`#`’があります。

2.4.17 レコード型

レコード *(record)* は `vector`と似ていますが、最初の要素は `type-of`でリターンされる型を保持するために使用されます。レコードの主要目的はプログラマーが Emacs のビルトインではない新たな型を作成することを可能にすることです。

レコードに作用する関数については Chapter 7 [Records], page 116 を参照してください。

2.4.18 型記述子

型記述子 *(type descriptor)* は型に関する情報を保持する `record`です。レコードの 1 スロット目には型を命名するシンボルでなければならず、`type-of`は `record`オブジェクトの型をリターンするためにこれに依存しています。他の型記述子スロットを Emacs は使用しません。これらを Lisp 拡張が使用するのは自由です。

`cl-structure-class`のインスタンスはすべて型記述子の例です。

2.4.19 autoload 型

autoload オブジェクト *(autoload object)* は、最初の要素がシンボル `autoload`のリストです。これはシンボルの関数定義として保存され、実際の定義にたいする代替としての役割をもちます。autoload オブジェクトは、必要な時にロードされる Lisp コードファイルの中で実際の定義を見つけることができることを宣言します。これにはファイル名と、それに加えて実際の定義についての他のいくつかの情報が含まれます。

ファイルのロード後、そのシンボルは autoload オブジェクトではない新しい関数定義をもつはずです。新しい定義は、最初からそこにあったかのように呼び出されます。ユーザーの観点からは関数呼び出しは期待された動作、つまりロードされたファイル内の関数定義を使用します。

autoload オブジェクトは通常、シンボルの関数セルにオブジェクトを保存する関数 `autoload`により作成されます。詳細は Section 16.5 [Autoload], page 274 を参照してください。

2.4.20 ファイナライザー型

ファイナライザーオブジェクト *(finalizer object)* は、オブジェクトがもはや必要なくなった後の Lisp コードのクリーンアップを助けます。ファイナライザーは、Lisp 関数オブジェクトを保持します。ガーベージコレクションのオアス後にファイナライザーオブジェクトが到達不能になったとき、Emacs はそのファイナライザーに関連付けられた関数オブジェクトを呼び出します。ファイナライザーの到達可否の判定時、もしかしてファイナライザーオブジェクト自身が参照を離さないのではないかと心配することなくファイナライザーを使用できるように、Emacs はファイナラーオブジェト自身からの参照は勘定しません。

ファイナラーザー内でのエラーは`*Messages*`にプリントされます。その関数が失敗しても、Emacs は与えられたファイナライザーオブジェクトに関連付けられた関数を正確に 1 回実行します。

`make-finalizer` *function* [Function]
: *function*を実行するファイナライザーを作成する。*function*はガーベージコレクション後、リターンされたファイナライザーオブジェクトが到達不能になったときに実行される。そのファイナライザーオブジェクトがファイナライザーオブジェクトからの参照を通じてのみ到達可能なら、*function*の実行是非の判断時の目的にたいして、それは到達可能とみなされない。*function* はファイナライザーオブジェクトごとに 1 回実行される。

2.5 編集用の型

前セクションの型は一般的なプログラミング目的のために使用され、これらの型のほとんどは Lisp 方言のほとんどで一般的です。Emacs Lisp は編集に関する目的のために、いくつかの追加のデータ型を提供します。

2.5.1 バッファー型

バッファー *(buffer)* とは、編集されるテキストを保持するオブジェクトです (Chapter 27 [Buffers], page 590 を参照)。ほとんどのバッファーはディスクファイル (Chapter 25 [Files], page 530 を参照) の内容を保持するので編集できますが、他の目的のために使用されるものもいくつかあります。ほとんどのバッファーはユーザーにより閲覧されることも意図しているので、いつかはウィンドウ内 (Chapter 28 [Windows], page 608 を参照) に表示されます。しかしバッファーはウィンドウに表示される必要はありません。バッファーはそれぞれ、ポイント *(point)* と呼ばれる位置指定をもちます (Chapter 30 [Positions], page 755 を参照)。ほとんどの編集コマンドは、カレントバッファー内のポイントに隣接する内容を処理します。常に 1 つのバッファーがカレントバッファー *(current buffer)* です。

バッファーの内容は文字列によく似ていますが、バッファーは Emacs Lisp の文字列と同じようには使用されず、利用可能な操作は異なります。文字列にテキストを挿入するためには部分文字列の結合が必要で、結果は完全に新しい文字列オブジェクトなのるのにたいして、バッファーでは既存のバッファーに効率的にテキストを挿入してバッファーの内容を変更できます。

標準的な Emacs 関数の多くは、カレントバッファー内の文字を操作したりテストするためのものです。このマニュアルはこれらの関数の説明のために、1 つのチャプターを設けています (Chapter 32 [Text], page 777 を参照)。

他のデータ構造のいくつかは、各バッファーに関連付けられています:

- ローカル構文テーブル (Chapter 35 [Syntax Tables], page 906 を参照)。
- ローカルキーマップ (Chapter 22 [Keymaps], page 422 を参照)。
- バッファーローカルな変数バインディングのリスト (Section 12.11 [Buffer-Local Variables], page 190 を参照)。
- オーバーレイ (Section 39.9 [Overlays], page 999 を参照)。
- バッファー内のテキストにたいするテキストプロパティー (Section 32.19 [Text Properties], page 813 を参照)。

ローカルキーマップと変数リストは、グローバルなバインディングや値を個別にオーバーライドするためのエントリーを含みます。これらは実際にプログラムを変更することなく、異なるバッファーでプログラムの振る舞いをカスタマイズするために使用されます。

バッファーはインダイレクト *(indirect:* 間接*)* — つまり他のバッファーとテキストを共有するがそれぞれ別に表示する — かもしれません。Section 27.11 [Indirect Buffers], page 605 を参照してください。

バッファーに入力構文はありません。バッファーはバッファー名を含むハッシュ表記でプリントされます。

```
(current-buffer)
     ⇒ #<buffer objects-ja.texi>
```

2.5.2 マーカー型

マーカー *(marker)* は特定のバッファー内の位置を表します。したがってマーカーには 2 つの内容 — 1 つはバッファー、もう 1 つは位置— をもちます。バッファーのテキストの変更では、マーカーが常

にバッファー内の同じ 2 つの文字の間に位置することを確実にするために、必要に応じて自動的に位置の値が再配置されます。

マーカーは入力構文をもちません。マーカーはカレントの文字位置とそのバッファー名を与える、ハッシュ表記でプリントされます。

```
(point-marker)
     ⇒ #<marker at 10779 in objects-ja.texi>
```

マーカーのテスト、作成、コピー、移動の方法についての情報は Chapter 31 [Markers], page 768 を参照してください。

2.5.3 ウィンドウ型

ウィンドウ *(window)* は Emacs がバッファーを表示するために使用する端末スクリーンの部分を記述します。すべてのウィンドウは関連付けられた 1 つのバッファーをもち、バッファーの内容はそのウィンドウに表示されます。それとは対照的に、あるバッファーは 1 つのウィンドウに表示されるか表示されないか、それとも複数のウィンドウに表示されるかもしれません。

同時に複数のウィンドウが存在するかもしれませんが、常に 1 つのウィンドウが選択されたウィンドウ *(selected window)* になります。Emacs がコマンドにたいして準備できているときは、(通常は) カーソルが表示されるウィンドウが選択されたウィンドウです。選択されたウィンドウは、通常はカレントバッファー (Section 27.2 [Current Buffer], page 590 を参照) を表示しますがこれは必須ではありません。

スクリーン上でウィンドウはフレームにグループ化されます。ウィンドウはそれぞれ、ただ 1 つのフレームだけに属します。Section 2.5.4 [Frame Type], page 26 を参照してください。

ウィンドウは入力構文をもちません。ウィンドウはウィンドウ番号と表示されているバッファー名を与える、ハッシュ表記でプリントされます。与えられたウィンドウに表示されるバッファーは頻繁に変更されるかもしれないので、一意にウィンドウを識別するためにウィンドウ番号が存在します。

```
(selected-window)
     ⇒ #<window 1 on objects-ja.texi>
```

ウィンドウに作用する関数の説明は Chapter 28 [Windows], page 608 を参照してください。

2.5.4 フレーム型

フレーム *(frame)* とは 1 つ以上の Emacs ウィンドウを含むスクリーン領域です。スクリーン領域を参照するために Emacs が使用する Lisp オブジェクトを指す場合にも “フレーム” という用語を使用します。

フレームは入力構文をもちません。フレームはフレームのタイトルとメモリー内のアドレス (フレームを一意に識別するのに有用) を与えるハッシュ表記でプリントされます。

```
(selected-frame)
     ⇒ #<frame emacs@psilocin.gnu.org 0xdac80>
```

フレームに作用する関数の説明は Chapter 29 [Frames], page 696 を参照してください。

2.5.5 端末型

端末 *(terminal)* は 1 つ以上の Emacs フレーム (Section 2.5.4 [Frame Type], page 26 を参照) を表示する能力があるデバイスです。

端末は入力構文をもちません。端末はその端末の順序番号と TTY デバイスファイル名を与える、ハッシュ表記でプリントされます。

```
(get-device-terminal nil)
     ⇒ #<terminal 1 on /dev/tty>
```

2.5.6 ウィンドウ構成型

ウィンドウ構成 *(window configuration)* はフレーム内のウィンドウの位置とサイズ、内容についての情報を保持します。これにより後で同じウィンドウ配置を再作成できます。

ウィンドウ構成は入力構文をもちません。ウィンドウ構成のプリント表現は‘`#<window-configuration>`’のようになります。ウィンドウ構成に関連するいくつかの関数の説明は Section 28.26 [Window Configurations], page 686 を参照してください。

2.5.7 フレーム構成型

フレーム構成 *(frame configuration)* はすべてのフレーム内のウィンドウの位置とサイズ、内容についての情報を保持します。これは基本型ではありません — 実際のところ、これは CAR が `frame-configuration`で CDR が alist であるようなリストです。それぞれの alist 要素は、その要素の CAR に示される 1 つのフレームを記述します。

フレーム構成に関連するいくつかの関数の説明は Section 29.13 [Frame Configurations], page 739 を参照してください。

2.5.8 プロセス型

プロセス *(process)* という単語は、通常は実行中のプログラムを意味します。Emacs 自身はこの種のプロセス内で実行されます。しかし Emacs Lisp では、プロセスとは Emacs プロセスにより作成されたサブプロセスを表す Lisp オブジェクトです。シェル、GDB、ftp、コンパイラーなどのプログラムは、Emacs のサブプロセスとして実行され Emacs の能力を拡張します。さらに操作を行なうために、Emacs サブプロセスは Emacs からテキスト入力を受け取り、テキスト出力を Emacs にリターンします。Emacs がサブプロセスにシグナルを送ることもできます。

プロセスオブジェクトは入力構文をもちません。プロセスオブジェクトはプロセス名を与えるハッシュ表記でプリントされます。

```
(process-list)
     ⇒ (#<process shell>)
```

プロセスの作成、削除、プロセスに関する情報のリターン、入力やシグナルの送信、出力の受信を行なう関数についての情報は Chapter 38 [Processes], page 934 を参照してください。

2.5.9 スレッド型

Emacs でのスレッド *(thread)* とは Emacs Lisp の実行スレッドとは別のスレッドを表します。これは自身の Lisp プログラムを実行して、自身のカレントバッファーを所有して、そのスレッドにロックされたサブプロセスをもつことができます (サブプロセスの出力を受け取ることができるのはそのスレッドのみ)。Chapter 37 [Threads], page 929 を参照してください。

スレッドオブジェクトは入力構文をもちません。スレッドオブジェクトは (名前を与えられていれば) スレッド名を与えるハッシュ表記かメモリー内のアドレスでプリントされます。

```
(all-threads)
    ⇒ (#<thread 0176fc40>)
```

2.5.10 ミューテックス型

ミューテックス *(mutex)* とはスレッド間で同期をとるためにスレッドが所有と非所有することができる排他ロックです。Section 37.2 [Mutexes], page 931 を参照してください。

ミューテックスオブジェクトは入力構文をもちません。プロセスオブジェクトは (名前を与えられていれば) ミューテックス名を与えるハッシュ表記かメモリー内のアドレスでプリントされます。

```
(make-mutex "my-mutex")
    ⇒ #<mutex my-mutex>
(make-mutex)
    ⇒ #<mutex 01c7e4e0>
```

2.5.11 状態変数型

状態変数 *(condition variable)* はミューテックスがサポートするよりも複雑な非同期スレッドのためのデバイスです。スレッドは別のスレッドが状態を通知したときに再開するように状態変数を待機することができます。

状態変数オブジェクトは入力構文をもちません。プロセスオブジェクトは (名前が与えられていれば) 状態変数の名前を与えるハッシュ表記かメモリー内のアドレスでプリントされます。

```
(make-condition-variable (make-mutex))
    ⇒ #<condvar 01c45ae8>
```

2.5.12 ストリーム型

ストリーム *(stream)* とは、文字のソースまたはシンクとして — つまり入力として文字を供給したり、出力として文字を受け入れるために使用できるオブジェクトです。多くの異なるタイプ — マーカー、バッファー、文字列、関数をこの方法で使用できます。ほとんどの場合、入力ストリーム (文字列ソース) はキーボード、バッファー、ファイルから文字を受け取り、出力ストリーム (文字シンク) は文字を`*Help*`バッファーのようなバッファーやエコーエリアに文字を送ります。

オブジェクト `nil`は、他の意味に加えてストリームとして使用されることがあります。`nil`は変数 `standard-input`や `standard-output`の値を表します。オブジェクト `t`も入力としてミニバッファー (Chapter 20 [Minibuffers], page 342 を参照)、出力としてエコーエリア (Section 39.4 [The Echo Area], page 983 を参照) の使用を指定するストリームになります。

ストリームは特別なプリント表現や入力構文をもたず、それが何であれそれらの基本型としてプリントされます。

パース関数およびプリント関数を含む、ストリームに関連した関数の説明は Chapter 19 [Read and Print], page 331 を参照してください。

2.5.13 キーマップ型

キーマップ *(keymap)* はユーザーがタイプした文字をコマンドにマップします。このマップはユーザーのコマンド入力が実行される方法を制御します。キーマップは、実際には CAR がシンボル `keymap`であるようなリストです。

キーマップの作成、プレフィクスキーの処理、ローカルキーマップやグローバルキーマップ、キーバインドの変更についての情報は Chapter 22 [Keymaps], page 422 を参照してください。

2.5.14 オーバーレイ型

オーバーレイ *(overlay)* はバッファーの一部に適用するプロパティーを指定します。それぞれのオーバーレイはバッファーの指定された範囲に適用され、プロパティーリスト (プロパティー名と値が交互に記述された要素のリスト) を含みます。オーバーレイプロパティーは、バッファーの指定された一部を、一時的に異なるスタイルで表示するために使用されます。オーバーレイは入力構文をもたず、バッファー名と範囲の位置を与えるハッシュ表記でプリントされます。

オーバーレイを作成したり使用する方法についての情報は Section 39.9 [Overlays], page 999 を参照してください。

2.5.15 フォント型

font はグラフィカルな端末上のテキストを表示する方法を指定します。実際には異なる 3 つのフォント型 — フォントオブジェクト *(font objects)*、フォントスペック *(font specs)*、フォントエンティティー *(font entities)* — が存在します。これらは入力構文をもちません。これらのプリント構文は '`#<font-object>`'、'`#<font-spec>`'、'`#<font-entity>`'のようになります。これらの Lisp オブジェクトの説明は Section 39.12.12 [Low-Level Font], page 1029 を参照してください。

2.6 循環オブジェクトの読み取り構文

複雑な Lisp オブジェクトでの共有された構造や循環する構造を表すために、リーダー構成 '`#`*n*`=`'と'`#`*n*`#`'を使用することができます。

後でオブジェクトを参照するには、オブジェクトの前で`#`*n*`=`を使用します。その後で、他の場所にある同じオブジェクトを参照するために、`#`*n*`#`を使用することができます。ここで *n*は任意の整数です。たとえば以下は、1 番目の要素が 3 番目の要素にも繰り替えされるリストを作成する方法です:

```
(#1=(a) b #1#)
```

これは、以下のような通常の構文とは異なります

```
((a) b (a))
```

これは 1 番目の要素と 3 番目の要素がそっくりなリストですが、これらは同じ Lisp オブジェクトではありません。以下で違いを見ることができます:

```
(prog1 nil
  (setq x '(#1=(a) b #1#)))
(eq (nth 0 x) (nth 2 x))
     ⇒ t
(setq x '((a) b (a)))
(eq (nth 0 x) (nth 2 x))
     ⇒ nil
```

"要素" として自身を含むような循環する構造を作成するために、同じ構文を使用できます。以下は例です:

```
#1=(a #1#)
```

これは 2 番目の要素がそのリスト自身であるリストを作成します。これが実際にうまくいくのか、以下で確認できます:

```
(prog1 nil
  (setq x '#1=(a #1#)))
(eq x (cadr x))
     ⇒ t
```

変数`print-circle`を非`nil`値にバインドした場合、Lisp プリンターは、循環および共有される Lisp オブジェクトを記録するこの構文を生成することができます。Section 19.6 [Output Variables], page 339 を参照してください。

2.7 型のための述語

関数が呼び出されたとき、Emacs Lisp インタープリター自身はその関数に渡された実際の引数の型チェックは行ないません。それが行なえないのは、Lisp における関数の引数は他のプログラミング言語のようなデータ型宣言をもたないからです。したがって実際の引数が、その関数が使用できる型に属するかどうかをテストするのは、それぞれの関数に任されています。

すべてのビルトイン関数は適切なときに実際の引数の型チェックを行い、引数の型が違う場合は`wrong-type-argument`エラーをシグナルします。たとえば以下は、+の引数に+が扱うことができない引数を渡したとき何が起こるかの例です:

```
(+ 2 'a)
     error  Wrong type argument: number-or-marker-p, a
```

異なる型にたいして異なる処理をプログラムに行なわせる場合は、明示的に型チェックを行なわなければなりません。オブジェクトの型をチェックするもっとも一般的な方法は型述語 *(type predicate)* 関数の呼び出しです。Emacs はそれぞれの型にたいする型述語をもち、組み合わされた型にたいする述語もあります。

型述語関数は 1 つの引数をとり、その引数が適切な型であれば`t`、そうでなければ`nil`をリターンします。述語関数にたいする一般的な Lisp 慣習にしたがい、ほとんどの型述語の名前は '`p`'で終わります。

以下はリストにたいしてチェックを行なう述語`listp`と、シンボルにたいしてチェックを行なう述語`symbolp`の例です。

```
(defun add-on (x)
  (cond ((symbolp x)
         ;; X がシンボルなら LIST に put する
         (setq list (cons x list)))
        ((listp x)
         ;; X がリストならその要素を LIST に追加
         (setq list (append x list)))
        (t
         ;; シンボルとリストだけを処理する
         (error "Invalid argument %s in add-on" x))))
```

以下のテーブルは事前定義された型述語 (アルファベット順) と、さらに情報を得るためのリファレンスです。

`atom` Section 5.2 [List-related Predicates], page 72 を参照のこと。

`arrayp` Section 6.3 [Array Functions], page 107 を参照のこと。

`bignump` Section 3.3 [Predicates on Numbers], page 40 を参照のこと。

`bool-vector-p`
Section 6.7 [Bool-Vectors], page 113 を参照のこと。

`booleanp` Section 1.3.2 [nil and t], page 2 を参照のこと。

`bufferp` Section 27.1 [Buffer Basics], page 590 を参照のこと。

`byte-code-function-p`
Section 2.4.16 [Byte-Code Type], page 23 を参照のこと。

`case-table-p`
Section 4.10 [Case Tables], page 69 を参照のこと。

`char-or-string-p`
Section 4.2 [Predicates for Strings], page 52 を参照のこと。

`char-table-p`
Section 6.6 [Char-Tables], page 110 を参照のこと。

`commandp` Section 21.3 [Interactive Call], page 382 を参照のこと。

`condition-variable-p`
Section 37.3 [Condition Variables], page 931 を参照のこと。

`consp` Section 5.2 [List-related Predicates], page 72 を参照のこと。

`custom-variable-p`
Section 15.3 [Variable Definitions], page 253 を参照のこと。

`fixnump` Section 3.3 [Predicates on Numbers], page 40 を参照のこと。

`floatp` Section 3.3 [Predicates on Numbers], page 40 を参照のこと。

`fontp` Section 39.12.12 [Low-Level Font], page 1029 を参照のこと。

`frame-configuration-p`
Section 29.13 [Frame Configurations], page 739 を参照のこと。

`frame-live-p`
Section 29.7 [Deleting Frames], page 730 を参照のこと。

`framep` Chapter 29 [Frames], page 696 を参照のこと。

`functionp`
Chapter 13 [Functions], page 209 を参照のこと。

`hash-table-p`
Section 8.4 [Other Hash], page 122 を参照のこと。

`integer-or-marker-p`
Section 31.2 [Predicates on Markers], page 769 を参照のこと。

`integerp` Section 3.3 [Predicates on Numbers], page 40 を参照のこと。

`keymapp` Section 22.4 [Creating Keymaps], page 425 を参照のこと。

`keywordp` Section 12.2 [Constant Variables], page 174 を参照のこと。

`listp` Section 5.2 [List-related Predicates], page 72 を参照のこと。

`markerp` Section 31.2 [Predicates on Markers], page 769 を参照のこと。

`mutexp` Section 37.2 [Mutexes], page 931 を参照のこと。

`nlistp` Section 5.2 [List-related Predicates], page 72 を参照のこと。

`number-or-marker-p`
Section 31.2 [Predicates on Markers], page 769 を参照のこと。

`numberp` Section 3.3 [Predicates on Numbers], page 40 を参照のこと。

`overlayp` Section 39.9 [Overlays], page 999 を参照のこと。

`processp` Chapter 38 [Processes], page 934 を参照のこと。

`recordp` Section 2.4.17 [Record Type], page 24 を参照のこと。

`sequencep`
Section 6.1 [Sequence Functions], page 95 を参照のこと。

`string-or-null-p`
: Section 4.2 [Predicates for Strings], page 52 を参照のこと。

`stringp` Section 4.2 [Predicates for Strings], page 52 を参照のこと。

`subrp` Section 13.9 [Function Cells], page 225 を参照のこと。

`symbolp` Chapter 9 [Symbols], page 124 を参照のこと。

`syntax-table-p`
: Chapter 35 [Syntax Tables], page 906 を参照のこと。

`threadp` Section 37.1 [Basic Thread Functions], page 929 を参照のこと。

`vectorp` Section 6.4 [Vectors], page 109 を参照のこと。

`wholenump`
: Section 3.3 [Predicates on Numbers], page 40 を参照のこと。

`window-configuration-p`
: Section 28.26 [Window Configurations], page 686 を参照のこと。

`window-live-p`
: Section 28.7 [Deleting Windows], page 625 を参照のこと。

`windowp` Section 28.1 [Basic Windows], page 608 を参照のこと。

あるオブジェクトがどの型かチェックするもっとも一般的な方法は、関数 `type-of`の呼び出しです。オブジェクトは、ただ 1 つだけの基本型に属することを思い出してください。`type-of`は、それがどの型かを告げます (Chapter 2 [Lisp Data Types], page 8 を参照)。しかし `type-of`は基本型以外の型については何も知りません。ほとんどの場合では、`type-of`より型述語を使用するほうが便利でしょう。

`type-of` *object* [Function]

この関数は *object*の基本型を名前とするシンボルをリターンする。リターン値はシンボル `bool-vector`、`buffer`、`char-table`、`compiled-function`、`condition-variable`、`cons`、`finalizer`、`float`、`font-entity`、`font-object`、`font-spec`、`frame`、`hash-table`、`integer`、`marker`、`mutex`、`overlay`、`process`、`string`、`subr`、`symbol`、`thread`、`vector`、`window`、`window-configuration`のいずれか。ただし *object*がレコードなら最初のスロットで指定された型をリターンする。Chapter 7 [Records], page 116 を参照のこと。

```
(type-of 1)
     ⇒ integer
(type-of 'nil)
     ⇒ symbol
(type-of '())    ; ()は nil
     ⇒ symbol
(type-of '(x))
     ⇒ cons
(type-of (record 'foo))
     ⇒ foo
```

2.8 同等性のための述語

ここでは 2 つのオブジェクトの同一性をテストする関数を説明します。(たとえば文字列などの) 特定の型のオブジェクト同士で内容の同一性をテストするには、別の関数を使用します。これらの述語にたいしては、そのデータ型を説明する適切なチャプターを参照してください。

`eq` *object1 object2* [Function]

この関数は *object1* と *object2* が同じオブジェクトなら `t`、それ以外は `nil` をリターンする。

object1 と *object2* が同じ名前をもつシンボルなら、通常は同じオブジェクトだが例外もある。Section 9.3 [Creating Symbols], page 126 を参照のこと。他の非数値型 (リストやベクター、文字列などの) にたいしては、同じ内容 (または要素) の 2 つの引数が両者 `eq` である必要はない。これらが同じオブジェクトの場合だけ `eq` であり、その場合には一方の内容を変更するともう一方の内容にも同じ変更が反映される。

object1 と *object2* 異なるタイプや値をもつ数値なら同じオブジェクトではなく、`eq` は `nil` をリターンする。同じ値をもつ fixnum なら同じオブジェクトであり、`eq` は `t` をリターンする。別個に計算されてたまたま同じ値をもち、かつ非 fixnum タイプの同じ数値型なら、それらは同じかもしれないし違うかもしれず、Lisp インタープリターが作成したオブジェクトが 1 つか 2 つかに依存して `eq` は `t` か `nil` をリターンする。

```
(eq 'foo 'foo)
     ⇒ t

(eq ?A ?A)
     ⇒ t

(eq 3.0 3.0)
     ⇒ t または nil
;; 浮動小数にたいする eq ではオブジェクト同じかもしれず違うかもしれない

(eq (make-string 3 ?A) (make-string 3 ?A))
     ⇒ nil

(eq "asdf" "asdf")
     ⇒ t または nil
;; 文字列コンテンツにたいする eq ではオブジェクト同じかもしれず違うかもしれない

(eq '(1 (2 (3))) '(1 (2 (3))))
     ⇒ nil

(setq foo '(1 (2 (3))))
     ⇒ (1 (2 (3)))
(eq foo foo)
     ⇒ t
(eq foo '(1 (2 (3))))
     ⇒ nil

(eq [(1 2) 3] [(1 2) 3])
     ⇒ nil
```

```
(eq (point-marker) (point-marker))
     ⇒ nil
```

`make-symbol`関数は intern されていないシンボルをリターンする。これは Lisp 式内でその名前を記述したシンボルとは区別される。同じ名前の異なるシンボルは `eq`ではない。Section 9.3 [Creating Symbols], page 126 を参照のこと。

```
(eq (make-symbol "foo") 'foo)
     ⇒ nil
```

Emacs Lisp バイトコンパイラーはリテラル文字列のような等価なリテラルオブジェクトを同一オブジェクトにたいする参照に落し込む (collapse into) かもしれない。バイトコンパイルされたコードはそのようなオブジェクトを `eq`で比較するだろうが、そうでないコードは異なるという効果がある。したがってコードではオブジェクトのリテラルコンテンツが `eq`か否かではなく、以下に説明する `equal`のような関数でオブジェクトの関数を使用すること。同様にコードではリテラルオブジェクトを変更 (たとえばリテラル文字列へのテキストプロパティの put) しないこと。バイトコンパイラーがそれらの落し込みを行っていたら、同一コンテンツをもつ別のリテラルオブジェクトに影響があるかもしれない。

`equal` *object1 object2* [Function]

この関数は *object1*と *object2*が同じ構成要素をもつなら `t`、それ以外は `nil`をリターンする。`eq`が引数が同じオブジェクトなのかテストするのにたいして、`equal`は同一でない引数の内部を調べて、それらの要素または内容が同一化をテストする。したがって 2 つのオブジェクトが `eq`ならばそれらは `equal`だが、その逆は常に真ではない。

```
(equal 'foo 'foo)
     ⇒ t

(equal 456 456)
     ⇒ t

(equal "asdf" "asdf")
     ⇒ t
(eq "asdf" "asdf")
     ⇒ nil

(equal '(1 (2 (3))) '(1 (2 (3))))
     ⇒ t
(eq '(1 (2 (3))) '(1 (2 (3))))
     ⇒ nil

(equal [(1 2) 3] [(1 2) 3])
     ⇒ t
(eq [(1 2) 3] [(1 2) 3])
     ⇒ nil

(equal (point-marker) (point-marker))
     ⇒ t
```

```
(eq (point-marker) (point-marker))
     ⇒ nil
```

文字列の比較は case を区別するがテキストプロパティーは考慮しない — これは文字列内の文字だけを比較する。Section 32.19 [Text Properties], page 813 を参照のこと。テキストプロパティーも比較する場合には、`equal-including-properties`を使用すること。技術的な理由によりユニバイト文字列とマルチバイト文字列は、それらが同じ文字コードのシーケンスを含み、それらのコードがすべて 0 から 127(ASCII) の場合に限り `equal`となる (Section 33.1 [Text Representations], page 852 を参照)。

```
(equal "asdf" "ASDF")
     ⇒ nil
```

しかし 2 つの別のバッファーは、それらのテキスト内容が同じでも `equal`と判断されることはない。

`equal`では等価性は再帰的に定義されています。たとえば 2 つのコンスセル *x*と *y*を与えると、`(equal` *x y*`)`は、以下の式の両方が `t`をリターンする場合だけ `t`をリターンします:

```
(equal (car x) (car y))
(equal (cdr x) (cdr y))
```

したがって循環リストの比較はエラーとなるような深い再帰を引き起こすかもしれず、`(equal a b)`は `t`をリターンするにも関わらず `(equal b a)`はエラーをシグナルするといった直感に反する結果となることがあります。

`equal-including-properties` *object1 object2* [Function]

この関数はすべてのケースにおいて `equal`と同様に振る舞うが、2 つの文字列が `equal`になるためには、それらが同じテキストプロパティーをもつ必要がある。

```
(equal "asdf" (propertize "asdf" 'asdf t))
     ⇒ t
(equal-including-properties "asdf"
                            (propertize "asdf" 'asdf t))
     ⇒ nil
```

2.9 可変性

変更されるべきではない Lisp オブジェクトがいくつかあります。たとえば Lisp 式`"aaa"`では文字列を生成しますが、そのコンテンツを変更するべきではありません。いくつかのオブジェクトは変更できません。たとえばある数値を計算して新たな数値を作成できたとしても、Lisp は既存の数値を変更する操作を提供しません。

その他の Lisp オブジェクトは副作用をともなう破壊的な操作を通じて安全に値を変更可能な *mutable(*可変*)* オブジェクトです。たとえばマーカーを別のポイントを指すマーカーに移動することにより、既存のマーカーを変更することができます。

数値は変更不可でマーカーはすべて mutable だとしても、mutable と非 mutable のメンバーをもつタイプがいくつかあります。これらのタイプにはコンス、ベクター、文字列が含まれます。たとえば`"cons"`と `(symbol-name 'cons)`は変更するべきではない文字列を生成しますが、`(copy-sequence "cons")`と `(make-string 3 ?a)`は後から `aset`を呼び出すことを通じて変更可能な mutable 文字列を生成します。

mutable オブジェクトは評価される式の一部となったときに mutable であることを止めます。たとえば:

```
(let* ((x (list 0.5))
```

```
      (y (eval (list 'quote x))))
(setcar x 1.5) ;; プログラムはこれを行うべきではない
y)
```

作成時にリスト`(0.5)`が mutable でも、それは`eval`に与えられるので`setcar`を通じて変更するべきではありません。変更するべきではないオブジェクトが後から mutable になることは決してないので逆はあり得ません。

変更するべきではないオブジェクトの変更をプログラムが試みた際の動作は未定義です。Lisp インタープリターがエラーをシグナルするかもしれず、クラッシュしたり他の方法で予測不能な振る舞いを引き起こすかもしれません [2]。

プログラムの一部として類似した定数が出現する際には、既存の定数やそのコンポーネントの再利用によって Lisp が時間やスペースを節約できるかもしれません。たとえば`(eq "abc" "abc")`はインタープリターが文字列リテラル`"abc"`の 1 つのインスタンスを作成したら`t`、2 つのインスタンスを作成したら`nil`をリターンします。したがってこの最適化使用の有無に関わらず機能するように Lisp プログラムを記述する必要があります。

[2] これは Common Lisp や C のような言語が定数にたいして指定する挙動であり、プログラムによる immutable オブジェクト変更の試みにエラーのシグナルを要求する JavaScript や Python のようなインタープリターとは異なる。理想的には Emacs Lisp インタープリターは後者を目指して進化するであろう。

3 数値

GNU Emacs は 2 つの数値データ型 — 整数 *(integers)* と浮動小数点数 *(floating-point numbers)* をサポートします。整数は −3、0、7、13、511 などの整数です。浮動小数点数は −4.5、0.0、2.71828 などの小数部をもちます。これらは指数記数法でも表現できます— ‘`1.5e2`’は ‘`150.0`’と同じです。ここで ‘`e2`’は 10 の 2 乗を表し、それに 1.5 を乗じるという意味です。整数計算は正確です。浮動小数点数の計算では数値は固定された精度をもつので、しばしば丸め誤差 (rounding errors) が発生します。

3.1 整数の基礎

Lisp リーダーは、10 進数字のシーケンス (オプションで最初の符号記号と最後のピリオドをともなう) として整数を読み取ります。

```
 1                ; 整数 1
 1.               ; 整数 1
+1                ; これも整数 1
-1                ; 整数 −1
 0                ; 整数 0
-0                ; 整数 0
```

10 以外の基数をもつ整数の構文は ‘`#`’、基数表示 (radix indication)、その後の 1 つ以上の数字から構成されます。基数表示は 2 進数 (binary) は ‘`b`’、8 進数 (octal) は ‘`o`’、16 進数 (hex) は ‘`x`’、基数 *radix*にたいしては ‘***radix*`r`**’になります。したがって ‘**`#b`*integer***’は 2 進数、‘**`#`*radix*`r`*integer***’は基数 *radix*で *integer*を読み取ります。*radix*の値として可能な値は 2 から 36 であり、最初の *radix* 文字は ‘`0`’–‘`9`’および ‘`A`’–‘`Z`’から採択されます。英文字の case(大文字小文字) は無視されて、最初の符号と最後のピリオドはありません。たとえば:

```
#b101100 ⇒ 44
#o54 ⇒ 44
#x2c ⇒ 44
#24r1k ⇒ 44
```

整数にたいして処理を行なうさまざまな関数、特にビット演算 (Section 3.8 [Bitwise Operations], page 46 を参照) を理解するためには、数を 2 進形式で見ることが助けになることがよくあります。

10 進整数の 5 は 2 進数では以下のようになります:

```
...000101
```

(省略記号 ‘`...`’は概念的に先頭ビットにマッチする無限個数のビットを意味する。ここでは無限個の 0 ビット。後の例でも ‘`...`’表記を使用する。)

整数の −1 は以下のようになります:

```
...111111
```

−1 はすべて 1 で表現されます (*2* の補数表記と呼ばれる)。

−1 から 4 を減じることで負の整数 −5 が得られます。10 進の整数 4 は 2 進では 100 です。したがって −5 は以下のようになります:

```
...111011
```

このチャプターで説明する多くの関数は、数字の位置として引数にマーカー (Chapter 31 [Markers], page 768 を参照) を受け取ります。そのような関数にたいする実際の引数は数字かマーカーなので、わたしたちはこれらの引数に *number-or-marker*という名前を与えることがあります。引数の値がマーカーならマーカーの位置が使用され、マーカーのバッファーは無視されます。

Emacs Lisp では、テキスト文字は整数により表現されます。0 から `(max-char)`までの整数は、有効な文字として判断されます。Section 33.5 [Character Codes], page 856 を参照してください。

Emacs Lisp の整数はマシンのワードサイズに制限されません。しかしその背後には *fixnums* と呼ばれる小さい整数と、*bignums* と呼ばれる大きい整数という 2 種類の整数が存在します。Emacs Lisp コードは通常は整数が fixnum か bignum のいずれであるかに依存するべきではありませんが、Emacs の古いバージョンでは fixnum だけがサポートされており、未だに fixnum だけを受け取る Emacs 関数がいくつかあり、古い Emacs Lisp コードが bignum を受け取ると問題が起こるかもしれません。たとえば古い Emacs Lisp コードは `eq`で整数にたいする数値の等価性を安全に比較できましたが、bignum の登場により整数の比較には `eql`や`=`のような等価性にたいする述語を使うことが必要になりました。

bignum の値の範囲は主メモリー量、bignum の指数の表現に使用されるワードサイズのようなマシン特性、および `integer-width`変数により制限されます。これらの制限は通常は fixnum にたいする制限よりは寛大です。bignum が数値的に fixnum と等しくなることはありません。Emacs は fixnum 範囲内の整数を、bignum ではなく常に fixnum として表現します。

fixnum の値の範囲はマシンに依存します。最小の範囲は −536,870,912 から 536,870,911(30 ビット長の -2^{29} から $2^{29}-1$) ですが、多くのマシンはより広い範囲を提供します。

`most-positive-fixnum` [Variable]
この変数の値は Emacs Lisp が扱える “小さい” 整数の最大値。典型的な値は 32 ビットでは $2^{29}-1$ 、64 ビットでは $2^{61}-1$ 。

`most-negative-fixnum` [Variable]
この変数の値は Emacs Lisp が扱える数値的に最小の “小さい” 整数。これは負の整数。典型的な値は 32 ビットでは -2^{29} 、64 ビットでは -2^{61}、。

`integer-width` [Variable]
この変数の値は大きな整数えお計算時に Emacs が範囲エラー (range error) をシグナルするかどうかを制御する負ではない整数。絶対値が 2^n, (nはこの変数の値) が小さい整数は範囲エラーをシグナルしない。大きい整数を簡単に作成できればエラーがシグナルされない場合もあるが、通常は大きな整数の作成を試みると範囲エラーをシグナルする。この変数に大きな数値を設定すると、巨大な整数の計算にコストを要する可能性がある。

3.2 浮動小数点数の基礎

浮動小数点数は整数以外の数値の表現に有用です。浮動小数点数の範囲は使用中のシンでの C のデータ型 `double`と同じ範囲です。Emacs がサポートするすべてのコンピューターでは IEEEの 64 ビット浮動小数フォーマットであり、これは IEEE Std 754-2019 で標準化されたもので、David Goldberg の論文 “What Every Computer Scientist Should Know About Floating-Point Arithmetic (`https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`)” で更に議論されています。モダンなプラットフォームでは浮動小数処理は IEEE-754 標準に厳密にしたがいますが、特に 32 ビット X86 のような陳腐化したプラットフォームでは丸めは常に正しい訳ではありません。

浮動小数点数にたいする入力構文は、小数点と指数のどちらか 1 つ、または両方が必要とします。オプションの符号 (‘+’か ‘-’) は、その数字と指数の前に記述します。たとえば ‘`1500.0`’、‘`+15e2`’、‘`15.0e+2`’、‘`+1500000e-3`’、‘`.15e4`’は値が 1500 の浮動小数点数を記述する 5 つの方法です。これらはすべて等価です。Common Lisp と同様、Emacs Lisp は浮動小数点数の小数点の後に少なくとも 1 つの数字を必要とします。‘`1500.`’は整数であって浮動小数点数ではありません。

Emacs Lisp は=のような数学的な比較に関して、-0.0を通常の 0 と数学的に同じものとして扱います。これは、(他の処理がこれらを区別するとしても-0.0と 0.0は数学的に等しいとする)IEEE浮動小数点数規格にしたがっています。

IEEE浮動小数点数規格は浮動小数点数として、正の無限大と負の無限大をサポートします。この規格は NaN または “not a number(数字ではない)” と呼ばれる値クラスも提供します。正しい答えが存在しないような場合に、数学関数はこのような値をリターンします。たとえば (/ 0.0 0.0)は NaN をリターンします。数値的に NaN はたとえ自身と比較してもすべての値にたいして数値的にイコールになることはありません。NaN は符号と仮数をもち、非数学関数は符号と仮数が一致すれば 2 つの NaN を等しいものと扱います。NaN の仮数は文字列表現の数字のようにマシン依存です。

NaN と符号つき 0 が関係する際には eql、equal、sxhash-eql、sxhash-equal、gethash のような非数学関数はそれらが数学的にイコールかではなく、値が区別できるかどうかを判断します。たとえば xと yが同じ NaN なら数値比較を使用する (= x y)は nilをリターンするのにたいして (equal x y)は tをリターンして、反対に (= 0.0 -0.0)が tをリターンするのにたいして (equal 0.0 -0.0)は nilをリターンします。

以下は、これらの特別な浮動小数点数にたいする入力構文です:

infinity ‘1.0e+INF’と ‘-1.0e+INF’

not-a-number
‘0.0e+NaN’と ‘-0.0e+NaN’

以下の関数は浮動小数点数を扱うために特化したものです:

isnan x [Function]
この述語は浮動小数引数が NaN なら t、それ以外は nilをリターンする。

frexp x [Function]
この関数はコンスセル (s . e)をリターンする。ここで sと eは、浮動小数点数の仮数 (浮動小数点数を 2 の指数表現したときの仮数) と指数である。

xが有限なら sは 0.5 以上 1.0 未満の浮動小数点数、eは整数で、$x = s2^e$ となる。xが 0 または無限なら sは xと等しくなる。xが NaN なら sも NaN。xが 0 なら eは 0。

ldexp s e [Function]
数値の仮数 sと整数の指数 eを与えられると、この関数は浮動小数点数 $s2^e$ をリターンする。

copysign *x1 x2* [Function]
この関数は *x2*の符号を *x1*の値にコピーして結果をリターンする。*x1*と *x2*は浮動小数でなければならない。

logb x [Function]
この関数は xの 2 進指数をリターンする。より正確には xが有限かつ非 0 なら $|x|$ の 2 を底とする対数を整数に切り下げた値。xが 0 または無限なら値は無限大。xが Na なら値は NaN。

```
(logb 10)
     ⇒ 3
(logb 10.0e20)
     ⇒ 69
(logb 0)
     ⇒ -1.0e+INF
```

3.3 数値のための述語

このセクションの関数は数値や、特定の数値型にたいしてテストを行ないます。関数 integerpと floatpは、引数として任意の Lisp オブジェクト型をとることができます (でなければ、あまり使用する機会ない)。しかし述語 zeropは引数として数値を要求します。Section 31.2 [Predicates on Markers], page 769 の integer-or-marker-p、number-or-marker-pも参照してください。

bignump *object* [Function]
この述語は引数が大きい整数かどうかをテストしてもしそうなら t、それ以外は nilをリターンする。小さい整数とは異なり大きい整数は eqでなくても=や eqlになり得る。

fixnump *object* [Function]
この述語は引数が小さい整数かどうかをテストしてもしそうなら t、それ以外は nilをリターンする。小さい整数は eqで比較できる。

floatp *object* [Function]
この述語は引数が浮動小数かどうかをテストしてもしそうなら t、それ以外は nilをリターンする。

integerp *object* [Function]
この述語は引数が整数かどうかをテストしてもしそうなら t、それ以外は nilをリターンする。

numberp *object* [Function]
この述語は引数が数 (整数か浮動小数) かどうかをテストしてもしそうなら t、それ以外は nil をリターンする。

natnump *object* [Function]
この述語は引数が正の整数かどうかをテストしてもしそうなら t、それ以外は nilをリターンする (名前は “natural numberl: 自然数” が由来)。0 は整数と判断される。

wholenumpは natnumpのシノニム。

zerop *number* [Function]
この述語は引数が 0 かどうかをテストしてもしそうなら t、それ以外は nilをリターンする。引数は数でなければならない。

(zerop x)は (= x 0)と等価。

3.4 数値の比較

数値にたいして数学的な等価性をテストするには通常は eq、eql、equalのような非数値的な比較述語のかわりに=を使用するべきです。異なる浮動小数点オブジェクトと大きい整数オブジェクトを数値的に等しくすることができます。これらの比較に eqを使用した場合にはそれらが同一のオブジェクトかどうかを、eqlや equalを使用した場合にはそれらの値が区別不能かどうかをテストすることになります。対照的に=は数値比較を使用して、非数値的な比較が nilをリターンするような場合に tをリターンしたり、その逆もあり得ます。Section 3.2 [Float Basics], page 38 を参照してください。

Emacs Lisp では 2 つの fixnum が数値的に等しければ同一の Lisp オブジェクトです。つまり fixnum では eqは=と同じです。値が未知の fixnum の比較に eqを使用する方が便利な場合があります。なぜなら未知の値が数でない場合でも eqはエラーを報告しないからです。これは任意のタイプの引数を受け付けます。対照的に引数が数でもマーカーでもなければ=はエラーをシグナルします。しかし整数の比較においてさえ、使用できる場合には=を使用するのがよいプログラミング習慣です。

数の比較において、2 つの数が同じデータ型 (どちらも整数か浮動小数) では、同じ値の場合は等しい数として扱う eqlや equalのほうが便利なときもあります。対照的に=は整数と浮動小数点数を等しい数と扱うことができます。Section 2.8 [Equality Predicates], page 33 を参照してください。

他の欠点もあります。浮動小数演算は正確ではないので、浮動小数値を比較するのが悪いアイデアとなるときがよくあります。通常は近似的に等しいことをテストするほうがよいでしょう。以下はこれを行なう関数です:

```
(defvar fuzz-factor 1.0e-6)
(defun approx-equal (x y)
  (or (= x y)
      (< (/ (abs (- x y))
            (max (abs x) (abs y)))
         fuzz-factor)))
```

= *number-or-marker* **&rest** *number-or-markers* [Function]
この関数はすべての引数が数値的に等しいかどうかをテストしてもしそうなら t、それ以外は nilをリターンする。

eql *value1 value2* [Function]
この関数は eqと同様に振る舞うが引数が両方とも数のときを除く。これは数を型と数値的な値により比較するので (eql 1.0 1)は nilをリターンするが、(eql 1.0 1.0)と (eql 1 1)は tをリターンする。これは小さい整数と同様に大きい整数の比較に使用できる。

/= *number-or-marker1 number-or-marker2* [Function]
この関数は引数が数値的に等しいかどうかをテストして、もし異なる場合は t、等しい場合は nilをリターンする。

< *number-or-marker* **&rest** *number-or-markers* [Function]
この関数は、各引数それぞれを後の引数より小さいかどうかをテストしてもしそうなら t、それ以外は nilをリターンする。

<= *number-or-marker* **&rest** *number-or-markers* [Function]
この関数は、各引数それぞれが後の引数以下かどうかをテストしてもしそうなら t、それ以外は nilをリターンする。

> *number-or-marker* **&rest** *number-or-markers* [Function]
この関数は、各引数それぞれが後の引数より大きいかどうかをテストしてもしそうなら t、それ以外は nilをリターンする。

>= *number-or-marker* **&rest** *number-or-markers* [Function]
この関数は、各引数それぞれが後の引数以上かどうかをテストしてもしそうなら t、それ以外は nilをリターンする。

max *number-or-marker* **&rest** *numbers-or-markers* [Function]
この関数は最大の引数をリターンする。

```
(max 20)
     ⇒ 20
(max 1 2.5)
     ⇒ 2.5
(max 1 3 2.5)
     ⇒ 3
```

`min` *number-or-marker* **&rest** *numbers-or-markers* [Function]
この関数は最小の引数をリターンする。

```
(min -4 1)
     ⇒ -4
```

`abs` *number* [Function]
この関数は *number*の絶対値をリターンする。

3.5 数値の変換

整数を浮動少数の変換には関数 `float`を使用します。

`float` *number* [Function]
これは浮動小数点数に変換された *number*をリターンする。すでに *number*が浮動小数点数なら `float`はそれを変更せずにリターンする。

浮動小数点数を整数に変換する関数が 4 つあります。これらは浮動小数点数を丸める方法が異なります。これらはすべて引数 *number*、およびオプション引数として *divisor*を受け取ります。引数は両方とも整数か浮動小数点数です。*divisor*が `nil`のこともあります。*divisor*が `nil`または省略された場合、これらの関数は *number*を整数に変換するか、それが既に整数の場合は変更せずにリターンします。*divisor*が非 `nil`なら、これらの関数は *number*を *divisor*で除して結果を整数に変換します。*divisor*が (整数か浮動小数かに関わらず)0 の場合、Emacs は `arith-error`エラーをシグナルします。

`truncate` *number* **&optional** *divisor* [Function]
これは 0 に向かって丸めることにより整数に変換した *number*をリターンする。

```
(truncate 1.2)
     ⇒ 1
(truncate 1.7)
     ⇒ 1
(truncate -1.2)
     ⇒ -1
(truncate -1.7)
     ⇒ -1
```

`floor` *number* **&optional** *divisor* [Function]
これは下方 (負の無限大に向かって) に丸めることにより整数に変換した *number*をリターンする。

*divisor*が指定された場合には、`mod`に相当する種類の除算演算を使用して下方に丸めを行う。

```
(floor 1.2)
     ⇒ 1
(floor 1.7)
     ⇒ 1
(floor -1.2)
     ⇒ -2
(floor -1.7)
     ⇒ -2
(floor 5.99 3)
     ⇒ 1
```

`ceiling` *number* **&optional** *divisor* [Function]

これは上方 (正の無限大に向かって) に丸めることにより整数に変換した *number*をリターンする。

```
(ceiling 1.2)
     ⇒ 2
(ceiling 1.7)
     ⇒ 2
(ceiling -1.2)
     ⇒ -1
(ceiling -1.7)
     ⇒ -1
```

`round` *number* **&optional** *divisor* [Function]

これはもっとも近い整数に向かって丸めることにより、整数に変換した *number*をリターンする。2 つの整数から等距離にある値の丸めでは、偶数の整数をリターンする。

```
(round 1.2)
     ⇒ 1
(round 1.7)
     ⇒ 2
(round -1.2)
     ⇒ -1
(round -1.7)
     ⇒ -2
```

3.6 算術演算

Emacs Lisp は伝統的な 4 つの算術演算 (加減乗除)、同様に剰余と modulus の関数、および 1 の加算と減算を行う関数を提供します。`%`を除き、これらの各関数は引き数として整数か浮動小数を受け取り、浮動小数の引数がある場合は浮動小数点数をリターンします。

`1+` *number-or-marker* [Function]

この関数は *number-or-marker* + 1 をリターンする。例えば、

```
(setq foo 4)
     ⇒ 4
(1+ foo)
     ⇒ 5
```

この関数は C の演算子`++`とは異なり、変数をインクリメントしない。この関数は和を計算するだけである。したがって以下を続けて評価すると、

```
foo
     ⇒ 4
```

変数をインクリメントしたい場合は、以下のように `setq`を使用しなければならない:

```
(setq foo (1+ foo))
     ⇒ 5
```

`1-` *number-or-marker* [Function]

この関数は *number-or-marker* − 1 をリターンする。

+ &rest *numbers-or-markers* [Function]

この関数は引数すべてを加算する。引数を与えないと+は 0 をリターンする。

```
(+)
     ⇒ 0
(+ 1)
     ⇒ 1
(+ 1 2 3 4)
     ⇒ 10
```

- &optional *number-or-marker* **&rest** *more-numbers-or-markers* [Function]

-関数は 2 つの目的 — 符号反転と減算 — をもつ。-に 1 つの引数を与えると、値は引数の符号を反転したものになる。複数の引数の場合は、*number-or-marker*から *more-numbers-or-markers*までの各値を蓄積的に減算する。引数がなければ結果は 0。

```
(- 10 1 2 3 4)
     ⇒ 0
(- 10)
     ⇒ -10
(-)
     ⇒ 0
```

*** &rest** *numbers-or-markers* [Function]

この関数はすべての引数を乗じて積をリターンする。引数がなかれば*は 1 をリターンする。

```
(*)
     ⇒ 1
(* 1)
     ⇒ 1
(* 1 2 3 4)
     ⇒ 24
```

/ *number* **&rest** *divisors* [Function]

*divisors*が 1 つ以上ならこの関数は *divisors*内の除数で順に *number*を除して、その商をリターンする。*divisors*がなければ、この関数は 1/*number*、つまり *number*の逆数をリターンする。各引数には数かマーカーを指定できる。

すべての引数が整数なら、結果は各除算の後に商を 0 へ向かって丸めることにより得られる整数となる。

```
(/ 6 2)
     ⇒ 3
(/ 5 2)
     ⇒ 2
(/ 5.0 2)
     ⇒ 2.5
(/ 5 2.0)
     ⇒ 2.5
(/ 5.0 2.0)
     ⇒ 2.5
(/ 4.0)
     ⇒ 0.25
```

```
(/ 4)
     ⇒ 0
(/ 25 3 2)
     ⇒ 4
(/ -17 6)
     ⇒ -2
```

整数を整数 0 で除すると Emacs は `arith-error`エラー (Section 11.7.3 [Errors], page 164 を参照) をシグナルする。浮動小数点数の除算では、非 0 の数を 0 で除することで正の無限大または負の無限大を得る (Section 3.2 [Float Basics], page 38 を参照)。

`%` *dividend divisor* [Function]

この関数は *dividend*を *divisor*で除した後、その剰余を整数でリターンする。引数は整数かマーカーでなければならない。

任意の 2 つの整数 *dividend*と *divisor*にたいして、

```
(+ (% dividend divisor)
   (* (/ dividend divisor) divisor))
```

は、*divisor*が非 0 なら常に *dividend*と等しくなる。

```
(% 9 4)
     ⇒ 1
(% -9 4)
     ⇒ -1
(% 9 -4)
     ⇒ 1
(% -9 -4)
     ⇒ -1
```

`mod` *dividend divisor* [Function]

この関数は *dividend*の *divisor*にたいする modulo、言い換えると *dividend*を *divisor*で除した後の剰余 (ただし符号は *divisor*と同じ) をリターンする。引数は数かマーカーでなければならない。

`%`とは異なり `mod`は浮動小数の引数を許す。これは商を整数に下方 (負の無限大に向かって) へ丸めて剰余を計算するのにこの商を使用する。

`mod`は *divisor*が 0 のとき、両方の引数が整数なら `arith-error`エラーをシグナルし、それ以外は NaN をリターンする。

```
(mod 9 4)
     ⇒ 1
(mod -9 4)
     ⇒ 3
(mod 9 -4)
     ⇒ -3
(mod -9 -4)
     ⇒ -1
(mod 5.5 2.5)
     ⇒ .5
```

任意の 2 つの数 *dividend*と *divisor*にたいして、

```
(+ (mod dividend divisor)
   (* (floor dividend divisor) divisor))
```

は常に *dividend*になる (ただし引数のどちらかが浮動小数なら、丸め誤差の範囲内で等しく、かつ *dividend*が整数で *divisor*が 0 なら `arith-error`となる)。`floor`については、Section 3.5 [Numeric Conversions], page 42 を参照のこと。

3.7 丸め処理

関数 `ffloor`、`fceiling`、`fround`、`ftruncate`は浮動小数の引数をとり、値が近くの整数であるような浮動少数をリターンします。`ffloor`は一番近い下方の整数、`fceiling`は一番近い上方の整数、`ftruncate`は 0 に向かう方向で一番近い整数、`fround`は一番近い整数をリターンします。

`ffloor` *float* [Function]
この関数は *float*を次に小さい整数値に丸めて、その値を浮動小数点数としてリターンする。

`fceiling` *float* [Function]
この関数は *float*を次に大きい整数値に丸めて、その値を浮動小数点数としてリターンする。

`ftruncate` *float* [Function]
この関数は *float*を 0 方向の整数値に丸めて、その値を浮動小数点数としてリターンする。

`fround` *float* [Function]
この関数は *float*を一番近い整数値に丸めて、その値を浮動小数点数としてリターンする。2 つの整数値との距離が等しい値にたいする丸めでは、偶数の整数をリターンする。

3.8 整数にたいするビット演算

コンピューターの中では、整数はビット *(bit: 0* か *1* の数字*)* のシーケンスである 2 進数で表されます。ビットシーケンスは概念的には最上位ビットがすべて 0 か 1 であるような左側に無限なシ 0 ケンスです。ビット演算はそのようなシーケンスの中の個々のビットに作用します。たとえばシフト *(shifting)* はシーケンス全体を 1 つ以上左または右に移動して、移動されたのと同じパターンを再現します。

Emacs Lisp のビット演算は整数だけに適用されます。

`ash` *integer1 count* [Function]
`ash` (算術シフト *(arithmetic shift)*) は、*integer1*の中のビット位置を左に *count*シフトする。*count*が負なら右にシフトする。左シフトでは右側に 0 が挿入されて、右シフトでは最右ビットが破棄される。整数処理として考えると、`ash`は *integer1*を 2^{count} 乗じてから下方、負の無限大に向かって丸めることにより結果を変換する。

以下はビットパターンを 1 ビット左にシフトしてから右にシフトする例。この例で 2 進数パターンの下位ビットだけを示している。先行ビットは表示されている上位ビットにすべて一致する。確認できるように 1 ビットの左シフトは 2 を乗じて、右シフトは 2 で除してから負の無限大方向に丸められる。

```
(ash 7 1) ⇒ 14
;; 10 進の 7 は 10 進の 14 になる
...000111
     ⇒
...001110
```

```
(ash 7 -1) ⇒ 3
...000111
     ⇒
...000011

(ash -7 1) ⇒ -14
...111001
     ⇒
...110010

(ash -7 -1) ⇒ -4
...111001
     ⇒
...111100
```

以下は 2 ビット左にシフトしてから右に 2 ビットシフトする例:

```
                    ;       2 進数値
(ash 5 2)           ;   5  =  ...000101
     ⇒ 20            ;      =  ...010100
(ash -5 2)          ;  -5  =  ...111011
     ⇒ -20           ;      =  ...101100
(ash 5 -2)
     ⇒ 1             ;      =  ...000001
(ash -5 -2)
     ⇒ -2            ;      =  ...111110
```

`lsh` *integer1 count* [Function]

`lsh`は *logical shift* の略で、*integer1*のビットを左に *count*シフトする。*count*が負なら *integer1*は fixnum か正の bignum のいずれかでなければならず、`lsh`はシフト前に負の fixnum を `most-negative-fixnum`で 2 回減算してあたかも符号なしであるかのように非負の結果を生成する。この奇妙な振る舞いは Emacs が fixnums だけをサポートしていた頃の振る舞いであり、現在では `ash`がより良い選択である。

*integer1*と *count1*がいずれも負の場合を除いて `lsh`は `ash`のように振る舞うので、以下の例ではこれらの例外ケースに焦点をあてている。これらの例は 30 ビットの fixnums を想定している。

```
                    ;     2 進数値
(ash -7 -1)         ; -7 = ...11111111111111111111111111111001
     ⇒ -4            ;    = ...11111111111111111111111111111100
(lsh -7 -1)
     ⇒ 536870908 ;    = ...01111111111111111111111111111100
(ash -5 -2)         ; -5 = ...11111111111111111111111111111011
     ⇒ -2            ;    = ...11111111111111111111111111111110
(lsh -5 -2)
     ⇒ 268435454 ;    = ...00111111111111111111111111111110
```

`logand` **&rest** *ints-or-markers* [Function]

この関数は引数のビットの AND をリターンする。すべての引数の *n*番目のビットが 1 の場合に限り、結果の *n*番目のビットが 1 となる。

たとえば 13 と 12 では、4 ビット 2 進数を使用すると 1101 と 1100 のビット AND は 1100 を生成する。この 2 進数ではいずれも左の 2 ビットがセット (つまり 1) されているので、リター

ンされる値の左 2 ビットがセットされる。しかし右の 2 ビットにたいしては少なくとも 1 つの引数でそのビットが 0 なので、リターンされる値の右 2 ビットは 0 になる。

したがって、

```
(logand 13 12)
     ⇒ 12
```

`logand`に何も引数も渡さなければ、値 −1 がリターンされる。−1 を 2 進数で表すとすべてのビットが 1 なので、−1 は`logand`にたいする単位元 (identity element) である。

```
                    ;      2 進数値

(logand 14 13)      ; 14  =  ...001110
                    ; 13  =  ...001101
     ⇒ 12           ; 12  =  ...001100

(logand 14 13 4)    ; 14  =  ...001110
                    ; 13  =  ...001101
                    ;  4  =  ...000100
     ⇒ 4            ;  4  =  ...000100

(logand)
     ⇒ -1           ; -1  =  ...111111
```

`logior` **&rest** *ints-or-markers* [Function]

この関数は、引数のビット単位の包含的 OR をリターンする。少なくとも 1 つの引数で *n*番目のビットが 1 なら、結果の *n*番目のビットが 1 になる。引数を与えなければ、結果はこの処理にたいする単位元である 0 となる。`logior`に渡す引数が 1 つだけならその引数がリターンされる。

```
                    ;      2 進数値

(logior 12 5)       ; 12  =  ...001100
                    ;  5  =  ...000101
     ⇒ 13           ; 13  =  ...001101

(logior 12 5 7)     ; 12  =  ...001100
                    ;  5  =  ...000101
                    ;  7  =  ...000111
     ⇒ 15           ; 15  =  ...001111
```

`logxor` **&rest** *ints-or-markers* [Function]

この関数は、引数のビット単位の排他的 OR をリターンする。*n*番目のビットが 1 であるような引数の数が奇数個の場合のみ、結果の *n*番目のビットが 1 となる。引数を与えなければ、結果はこの処理の単位元である 0 となる。`logxor`に渡す引数が 1 つだけならその引数がリターンされる。

```
                    ;      2 進数値

(logxor 12 5)       ; 12  =  ...001100
                    ;  5  =  ...000101
     ⇒ 9            ;  9  =  ...001001

(logxor 12 5 7)     ; 12  =  ...001100
                    ;  5  =  ...000101
                    ;  7  =  ...000111
     ⇒ 14           ; 14  =  ...001110
```

`lognot` *integer* [Function]

この関数は引数のビット単位の補数 (bitwise complement) をリターンする。*integer*の *n*番目のビットが 0 の場合に限り、結果の *n*番目のビットが 1 になり、その逆も成り立つ。結果は −1 − *integer*と等価。

```
(lognot 5)
     ⇒ -6
;;  5  =  ...000101
;; becomes
;; -6  =  ...111010
```

`logcount` *integer* [Function]

この関数は *integer*のハミング重み (Hamming weight: *integer*の 2 進数表現での 1 の個数) をリターンする。*integer*が負なら、その 2 の補数の 2 進数表現での 0 ビットの個数をリターンする。結果は常に非負となる。

```
(logcount 43)     ;  43 = ...000101011
     ⇒ 4
(logcount -43)    ; -43 = ...111010101
     ⇒ 3
```

3.9 標準的な数学関数

以下の数学的関数は、引数として整数と同様に浮動小数点数も受け入れます。

`sin` *arg* [Function]
`cos` *arg* [Function]
`tan` *arg* [Function]

これらは三角関数であり、引数 *arg*はラジアン単位。

`asin` *arg* [Function]

`(asin arg)`の値は、sin の値が *arg*となるような $-\pi/2$ から $\pi/2$ (両端を含む) の数である。*arg*が範囲外 ([−1, 1] の外) なら、`asin`は NaN をリターンする。

`acos` *arg* [Function]

`(acos arg)`の値は、cos の値が *arg*となるような、0 から π (両端を含む) の数である。arg が範囲外 ([-1, 1] の外) なら `acos`は NaN をリターンする。

`atan` *y* **&optional** *x* [Function]

`(atan y)`の値は、tan の値が *y*となるような、$-\pi/2$ から $\pi/2$ (両端を含まず) の数である。オプションの第 2 引数 *x*が与えられると、`(atan y x)`の値はベクトル `[x, y]`と `X`軸が成す角度のラジアン値となる。

`exp` *arg* [Function]

これは指数関数である。この関数は *e* の指数 *arg*をリターンする。

`log` *arg* **&optional** *base* [Function]

この関数は底を *base*とする *arg*の対数をリターンする。*base*を指定しなければ、自然底 (natural base)*e* が使用される。*arg*か *base*が負なら、`log`は NaN をリターンする。

`expt` *x y* [Function]
この関数は *x*に *y*を乗じてリターンする。引数が両方とも整数で *y*が非負なら結果は整数になる。この場合オーバーフローはエラーをシグナルするので注意。*x*が有限の負数で *y*が有限の非整数なら、`expt`は NaN をリターンする。

`sqrt` *arg* [Function]
これは *arg*の平方根をリターンする。*arg*が有限で 0 より小さければ、`sqrt`は NaN をリターンする。

加えて、Emacs は以下の数学的な定数を定義します:

`float-e` [Variable]
自然対数 e(2.71828...)

`float-pi` [Variable]
円周率 pi(3.14159...)

3.10 乱数

決定論的なコンピュータープログラムでは真の乱数を生成することはできません。しかしほとんどの目的には、疑似乱数 *(pseudo-random numbers)* で充分です。一連の疑似乱数は決定論的な手法により生成されます。真の乱数ではありませんが、それらにはランダム列を模する特別な性質があります。たとえば疑似ランダム系では、すべての可能な値は均等に発生します。

疑似乱数はシード値 *(seed value)* から生成されます。与えられた任意のシードから開始することにより、`random`関数は常に同じ数列を生成します。デフォルトでは、Emacs は開始時に乱数シードを初期化することにより、それぞれの Emacs の実行において、`random`の値シーケンスは (ほとんど確実に) 異なります。

再現可能な乱数シーケンスが欲しい場合もあります。たとえば乱数シーケンスに依存するプログラムをデバッグする場合、プログラムの各実行において同じ挙動を得ることが助けになります。再現可能なシーケンスを作成するには、`(random "")`を実行します。これは特定の Emacs の実行可能ファイルにたいして、シードに定数値をセットします (しかしこの実行可能ファイルは、その他の Emacs ビルドと異なるものになるであろう)。シード値として、他のさまざまな文字列を使用することができます。

`random` **&optional** *limit* [Function]
この関数は疑似乱数の整数をリターンする。繰り返し呼び出すと一連の疑似乱数の整数をリターンする。

*limit*が正の fixnum なら、値は負ではない *limit*未満の値から選択される。それ以外なら値は Lisp で表現可能な任意の fixnum(`most-negative-fixnum`から `most-positive-fixnum` の間の任意の整数) となるだろう (Section 3.1 [Integer Basics], page 37 を参照)。

*limit*が `t`なら、あたかも Emacs が再起動されたかのように、通常はシステムのエントロピーから新たなシードが選択されることを意味する。エントロピープールを欠くシステムでは、カレント時刻のような若干揮発性が低い乱数からシードが選択される。

*limit*が文字列なら、その文字列定数にもとづいた新しいシードを選択することを意味する。

4 文字列と文字

Emacs Lisp の文字列は、文字列の順序列 (ordered sequence) を含む配列です。文字列はシンボル、バッファー、ファイルの名前に使用されます。その他にもユーザーにたいしてメッセージを送ったりバッファー間でコピーする文字列を保持したり等、多くの目的に使用されます。文字列は特に重要なので、Emacs Lisp は特別には文字列を操作するために多くの関数があります。Emacs Lisp プログラムでは個々の文字より文字列を多用します。

キーボードの文字イベントの文字列にたいする特別な考慮は、Section 21.7.15 [Strings of Events], page 402 を参照してください。

4.1 文字列と文字の基礎

文字 (character) とは、テキスト内の 1 つの文字を表す Lisp オブジェクトです。Emacs Lisp では文字は単なる整数です。ある整数が文字か文字でないかを区別するのは、それが使用される方法だけです。Emacs での文字表現についての詳細は Section 33.5 [Character Codes], page 856 を参照してください。

文字列 (string) とは固定された文字シーケンスです。これは配列 *(array)* と呼ばれるシーケンス型であり、配列長が固定で一度作成したら変更できないことを意味します (Chapter 6 [Sequences Arrays Vectors], page 95 を参照)。C とは異なり、Emacs Lisp の文字列は文字コードを判断することにより終端されません。

文字列は配列であるということは同様にシーケンスでもあるので、Chapter 6 [Sequences Arrays Vectors], page 95 にドキュメントされている一般的な配列関数やシーケンス関数で文字列を処理できます。たとえば関数 `aref`を使用して文字列内の特定の文字にアクセスしたり変更することができます (Section 6.3 [Array Functions], page 107 を参照)。

Emacs 文字列での非 ASCIIにたいすテキスト表現は 2 つ — ユニバイト (unibyte) とマルチバイト (multibyte) があります。ほとんどの Lisp プログラミングでは、これら 2 つの表現を気にする必要はありません。詳細は Section 33.1 [Text Representations], page 852 を参照してください。

キーシーケンスがユニバイト文字列で表されることがあります。ユニバイト文字列がキーシーケンスの場合、範囲 128 から 255 までの文字列要素は範囲 128 から 255 の文字コードではなく、メタ文字 (これは非常に大きな整数である) を表します。文字列はハイパー (hyper)、スーパー (super)、アルト (alt) で修飾された文字を保持できません。文字列は ASCIIコントロール文字を保持できますが、それは他のコントロール文字です。文字列は ASCIIコントロール文字の case を区別できません。そのような文字をシーケンスに保存したい場合は、文字列ではなくベクターを使用しなければなりません。キーボード入力文字についての情報は Section 2.4.3 [Character Type], page 11 を参照してください。

文字列は正規表現を保持するために便利です。`string-match` (Section 34.4 [Regexp Search], page 894 を参照) を使用して、文字列にたいして正規表現をマッチすることもできます。関数 `match-string` (Section 34.6.2 [Simple Match Data], page 899 を参照) と `replace-match` (Section 34.6.1 [Replacing Match], page 897 を参照) は、文字列にたいして正規表現をマッチした後に、文字列を分解・変更するのに便利です。

バッファーのように、文字列は文字列内の文字自身とその文字にたいするテキストプロパティーを含みます。Section 32.19 [Text Properties], page 813 を参照してください。文字列からバッファーや他の文字列にテキストをコピーする、すべての Lisp プリミティブ (Lisp primitives) はコピーされる文字のプロパティーもコピーします。

文字列の表示やバッファーにコピーする関数についての情報は Chapter 32 [Text], page 777 を参照してください。文字または文字列の構文についての情報は、Section 2.4.3 [Character Type],

page 11 と Section 2.4.8 [String Type], page 19 を参照してください。異なるテキスト表現間で変換したり、文字コードのエンコードやデコードを行う関数については Chapter 33 [Non-ASCII Characters], page 852 を参照してください。ディスプレイ上の文字列幅の計算に `length`を使用するべきではないことにも注意してください。かわりに `string-width`を使用してください (Section 39.10 [Size of Displayed Text], page 1007 を参照)。

4.2 文字列のための述語

一般的なシーケンスや配列にたいする述語についての情報は、Chapter 6 [Sequences Arrays Vectors], page 95 と Section 6.2 [Arrays], page 107 を参照してください。

`stringp` *object* [Function]
この関数は *object*が文字列なら `t`、それ以外は `nil`をリターンする。

`string-or-null-p` *object* [Function]
この関数は *object*が文字列か `nil`なら `t`、それ以外は `nil`をリターンする。

`char-or-string-p` *object* [Function]
この関数は *object*が文字列か文字 (たとえば整数) なら `t`、それ以外は `nil`をリターンする。

4.3 文字列の作成

以下の関数は新たに文字列を作成したり、文字列同士の結合による文字列の作成や、文字列の一部から文字列を作成する関数です。

`make-string` *count character* **&optional** *multibyte* [Function]
この関数は *character*を *count*回繰り返すことにより作成された文字列をリターンする。*count* が負ならエラーをシグナルする。

```
(make-string 5 ?x)
     ⇒ "xxxxx"
(make-string 0 ?x)
     ⇒ ""
```

*character*が ASCII文字なら、結果は通常はユニバイト文字列になる。しかしオプション引数 *multibyte*が非 `nil`なら、この関数はかわりにマルチバイト文字列を生成する。これは結果を後で非 ASCII文字列と結合したり、結果の中のいくつかの文字を非 ASCII文字で置換する必要がある際に有用。

この関数に対応する他の関数には `make-vector` (Section 6.4 [Vectors], page 109 を参照) や `make-list` (Section 5.4 [Building Lists], page 75 を参照) が含まれる。

`string` **&rest** *characters* [Function]
この関数は文字 *characters*を含む文字列をリターンする。

```
(string ?a ?b ?c)
     ⇒ "abc"
```

`substring` *string* **&optional** *start end* [Function]
この関数は *string*から、インデックス *start*の文字 (その文字を含む) と *end*の文字 (その文字は含まない) の間の範囲の文字で構成される、新しい文字列をリターンする。文字列の最初の文字はインデックス 0。引数が 1 つなら、この関数は単に *string*をコピーする。

```
(substring "abcdefg" 0 3)
     ⇒ "abc"
```

上記の例では‘a’のインデックスは 0、‘b’のインデックスは 1、‘c’のインデックスは 2 となる。インデックス 3 — この文字列の 4 番目の文字 — は、コピーされる部分文字列の文字位置までをマークする。したがって文字列"abcdefg"から‘abc’がコピーされる。

負の数は文字列の最後から数えることを意味するので、−1 は文字列の最後の文字のインデックスである。たとえば:

```
(substring "abcdefg" -3 -1)
     ⇒ "ef"
```

この例では‘e’のインデックスは −3、‘f’のインデックスは −2、‘g’のインデックスは −1。つまり‘e’と‘f’が含まれ、‘g’は含まれない。

*end*に nilを使用した場合、それは文字列の長さを意味する。したがって、

```
(substring "abcdefg" -3 nil)
     ⇒ "efg"
```

引数 *end*を省略した場合、それは nilを指定したのと同じである。(substring *string* 0)は *string*のすべてをコピーしてリターンする。

```
(substring "abcdefg" 0)
     ⇒ "abcdefg"
```

しかしこの目的のためには copy-sequenceを推奨する (Section 6.1 [Sequence Functions], page 95 を参照)。

*string*からコピーされた文字がテキストプロパティーをもつなら、そのプロパティーは新しい文字列へもコピーされる。Section 32.19 [Text Properties], page 813 を参照のこと。

substringの最初の引数にはベクターも指定できる。たとえば:

```
(substring [a b (c) "d"] 1 3)
     ⇒ [b (c)]
```

*start*が整数でない、または *end*が整数でも nilでもなければ、wrong-type-argumentエラーがシグナルされる。*start*が *end*の後の文字を指す、または *string*にたいして範囲外の整数をいずれかに指定すると、args-out-of-rangeエラーがシグナルされる。

この関数に対応するのは buffer-substring (Section 32.2 [Buffer Contents], page 778 を参照) で、これはカレントバッファー内のテキストの一部を含む文字列をリターンする。文字列の先頭はインデックス 0 だが、バッファーの先頭はインデックス 1 である。

`substring-no-properties` *string* **&optional** *start end* [Function]

これは substringと同じように機能するが、値のすべてのテキストプロパティーを破棄する。*start*を省略したり nilを指定することができ、その場合は 0 と等価だる。したがって (substring-no-properties *string*) は、すべてのテキストプロパティーが削除された *string*のコピーをリターンする。

`concat` **&rest** *sequences* [Function]

この関数は渡された引数内の文字からなる文字列をリターンする (もしあればテキストプロパティーも)。引数には文字列、数のリスト、数のベクターを指定できる。引数は変更されない。concatに引数を指定しなければ空文字列をリターンする。

```
(concat "abc" "-def")
     ⇒ "abc-def"
(concat "abc" (list 120 121) [122])
     ⇒ "abcxyz"
```

```
;; nilh あ空のシーケンス。
(concat "abc" nil "-def")
     ⇒ "abc-def"
(concat "The " "quick brown " "fox.")
     ⇒ "The quick brown fox."
(concat)
     ⇒ ""
```

この関数は常に新たな文字列の割り当てを行う訳ではない。呼び出し側は結果が新たな文字列であること、もしくは既存の文字列にたいして `eq`であることに依存しないよう推奨する。

特にリターン値を変更すると誤って別の文字列を変更したり、プログラム内の定数文字列の変更や、エラーを raise することさえあり得る。安全に変更できる文字列を取得するには、結果に `copy-sequence`を使用すること。

他の結合関数 (concatenation functions) についての情報は Section 13.6 [Mapping Functions], page 219 の `mapconcat`、Section 6.5 [Vector Functions], page 109 の `vconcat`、Section 5.4 [Building Lists], page 75 の `append`を参照のこと。シェルコマンドで使用される文字列の中に、個々のコマンドライン引数を結合するには、Section 38.2 [Shell Arguments], page 936 を参照されたい。

`split-string` *string* **&optional** *separators omit-nulls trim* [Function]

この関数は正規表現 *separators*(Section 34.3 [Regular Expressions], page 883 を参照) にもとづいて、*string*を部分文字列に分解する。*separators*にたいする各マッチは分割位置を定義する。分割位置の間にある部分文字列をリストにまとめてリターンする。

*separators*が `nil` (か省略) ならデフォルトは `split-string-default-separators`の値となり、関数は *omit-nulls*が `t`であるかのように振る舞う。

*omit-nulls*が `nil`(または省略) なら、連続する 2 つの *separators*へのマッチか、*string*の最初か最後にマッチしたときの空文字列が結果に含まれる。*omit-nulls*が `t`なら、これらの空文字列は結果から除外される。

オプションの引数 *trim*が非 `nil`なら、その値は各部分文字列の最初と最後からトリム (trim: 除去) するテキストにマッチする正規表現を指定する。トリムによりその部分文字列が空になるようなら、それは空文字列として扱われる。

文字列を分割して `call-process`や `start-process`に適するような、個々のコマンドライン引数のリストにする必要がある場合は Section 38.2 [Shell Arguments], page 936 を参照のこと。

以下は例:

```
(split-string "  two words ")
     ⇒ ("two" "words")
```

有用性はほとんどないであろう `("" "two" "words" "")`という結果とはならない。このような結果が必要なら *separators*に明示的な値を使用すること

```
(split-string "  two words "
              split-string-default-separators)
     ⇒ ("" "two" "words" "")

(split-string "Soup is good food" "o")
     ⇒ ("S" "up is g" "" "d f" "" "d")
(split-string "Soup is good food" "o" t)
```

```
     ⇒ ("S" "up is g" "d f" "d")
(split-string "Soup is good food" "o+")
     ⇒ ("S" "up is g" "d f" "d")
```

空のマッチはカウントされます。例外は、空でないマッチを使用することにより、すでに文字列の最後に到達しているとき、または *string*が空の時で、この場合 `split-string`は最後の空マッチを探しません。

```
(split-string "aooob" "o*")
     ⇒ ("" "a" "" "b" "")
(split-string "ooaboo" "o*")
     ⇒ ("" "" "a" "b" "")
(split-string "" "")
     ⇒ ("")
```

しかし *separators*が空文字列にマッチできるとき、通常は *omit-nulls*を `t`にすれば、前の 3 つの例の不明瞭さはほとんど発生しない:

```
(split-string "Soup is good food" "o*" t)
     ⇒ ("S" "u" "p" " " "i" "s" " " "g" "d" " " "f" "d")
(split-string "Nice doggy!" "" t)
     ⇒ ("N" "i" "c" "e" " " "d" "o" "g" "g" "y" "!")
(split-string "" "" t)
     ⇒ nil
```

空でないマッチより空のマッチを優先するような、一部の “非貪欲 (non-greedy)” な値を *separators*に指定することにより、幾分奇妙 (ではあるが予見可能) な振る舞いが発生することがある。繰り返しになるが、そのような値は実際には稀である:

```
(split-string "ooo" "o*" t)
     ⇒ nil
(split-string "ooo" "\\|o+" t)
     ⇒ ("o" "o" "o")
```

`split-string-default-separators` [Variable]
`split-string`の *separators*にたいするデフォルト値。通常の値は`"[ \f\t\n\r\v]+"`。

4.4 文字列の変更

このセクションで説明する処理を介して変更可能な文字列のコンテンツを変更できます。Section 2.9 [Mutability], page 35 を参照してください。

既存の文字列の内容を変更するもっとも基本的な方法は、`aset` (Section 6.3 [Array Functions], page 107 を参照) を使用する方法です。`(aset` ***string idx char***`)`は、*string*のインデックス *idx* に、*char*を格納します。それぞれの文字は 1 文字以上を占有しますが、すでにインデックスの場所にある文字のバイト数が *char*が要するバイト数と異なる場合、`aset`はエラーをシグナルします。

より強力な関数は `store-substring`です:

`store-substring` *string idx obj* [Function]
この関数はインデックス *idx*で開始される位置に *obj*を格納することにより、文字列 *string*の内容の一部を変更する。*obj*は文字、または (*string*より小さい) 文字列です。

既存の文字列の長さを変更するのは不可能なので、*string*の実際の長さに *obj*が収まらない、または *string*のその位置に現在ある文字のバイト数が新しい文字に必要なバイト数と異なる場合はエラーになる。

パスワードを含む文字列をクリアーするときには `clear-string`を使用します:

`clear-string` *string* [Function]
これは *string*をユニバイト文字列にして、内容を 0 にクリアーする。これにより *string*の長さも変更されるだろう。

4.5 文字および文字列の比較

`char-equal` *character1 character2* [Function]
この関数は引数が同じ文字を表すなら `t`、それ以外は `nil`をリターンする。`case-fold-search`が非 `nil`なら、この関数は case の違いを無視する。

```
(char-equal ?x ?x)
     ⇒ t
(let ((case-fold-search nil))
  (char-equal ?x ?X))
     ⇒ nil
```

`string=` *string1 string2* [Function]
この関数は、2 つの文字列の文字が正確にマッチすれば `t`をリターンする。引数にはシンボルも指定でき、この場合はそのシンボル名が使用される。`case-fold-search`とは無関係に case は常に意味をもつ。

この関数は、`equal`で 2 つの文字列を比較するのと等価である (Section 2.8 [Equality Predicates], page 33 を参照)。特に、2 つの文字列のテキストプロパティーは無視されます。テキストプロパティーだけが異なる文字列を区別する必要があるなら `equal-including-properties`を使用すること。しかし `equal`とは異なり、いずれかの引数が文字列でもシンボルでもなければ、`string=`はエラーをシグナルする。

```
(string= "abc" "abc")
     ⇒ t
(string= "abc" "ABC")
     ⇒ nil
(string= "ab" "ABC")
     ⇒ nil
```

技術的な理由によりユニバイト文字列とマルチバイト文字列が `equal`になるのは、それらが同じ文字コードのシーケンスを含み、それらすべてのコードが 0 から 127(ASCII)、または 160 から 255(`eight-bit-graphic`) のときだけである。しかしユニバイト文字列をマルチバイト文字列に変更する際、コードが 160 から 255 の範囲にあるすべての文字はより高いコードに変換され、ASCII文字は変換されないまま残る。したがってユニバイト文字列とそれを変換したマルチバイト文字列は、その文字列のすべてが ASCIIのときだけ `equal`となる。もしマルチバイト文字列中で文字コード 160 から 255 の文字があったとしても、それは完全に正しいとは言えない。結果として、すべてが ASCIIではないユニバイト文字列とマルチバイト文字列が `equal`という状況は、もしかしたら Emacs Lisp プロプラマーが直面するかもしれない、とても稀で記述的に不可解な状況だといえよう。Section 33.1 [Text Representations], page 852 を参照されたい。

`string-equal` *string1 string2* [Function]
`string-equal`は `string=`の別名である。

`string-collate-equalp` *string1 string2* **&optional** *locale ignore-case* [Function]
この関数は照合ルール (collation rules) にもとづいて *string1*と *string2*が等しければ `t`をリターンする。照合ルールは *string1*と *string2*に含まれる文字の辞書順だけではなく、それらの文字間の関係に関する他のルールにより判断される。これは通常は、Emacs 実行中の *locale*環境により決定される。

たとえば異なるコーディングポイントでも、Unicode のグレイブアクセント文字のように同じ意味なら等しいとみなされる。

```
(string-collate-equalp (string ?\uFF40) (string ?\u1FEF))
     ⇒ t
```

オプション引数 *locale*(文字列) は、照合用のカレント locale 識別子 (current locale identifier) をオーバーライドする。値はシステムに依存する。たとえば POSIX システムでは`"en_US.UTF-8"`、MS-Windows システムでは`"enu_USA.1252"`の *locale*が適用できるだろう。

*ignore-case*が非 `nil`なら、それらは比較前に小文字に変換される。

MS-Windows システムで Unicode 互換の照合をエミュレートする場合、MS-Windows では locale のコードセット部分を`"UTF-8"`にできないので、`w32-collate-ignore-punctuation`に非 `nil`値をバインドすること。

ある locale 環境をシステムがサポートしなれければ、この関数は `string-equal`と同様に振る舞う。

一般的にファイルシステムは照合ルールが実装するような文字列の言語学的な等価性を尊重しないので、この関数をファイル名の等価性の比較に使用しないこと。

`string<` *string1 string2* [Function]
この関数は 2 つの文字列を 1 文字ずつ比較する。この関数は同時に 2 つの文字列をスキャンして、対応する文字同士がマッチしない最初のペアを探す。2 つの文字列内で小さいほうの文字が *string1*の文字なら *string1*が小さいことになり、この関数は `t`をリターンする。小さいほうの文字が *string2*の文字なら *string1*が大きいことになり、この関数は `nil`をリターンする。2 つの文字列が完全にマッチしたら値は `nil`になる。

文字のペアーは文字コードで比較されル。ASCII文字セットでは英小文字は英大文字より高い数値をもつことに留意されたい。数字と区切り文字の多くは英大文字より低い数値をもつ。ASCII 文字は任意の非 ASCII文字より小さくなる。ユニバイトの非 ASCII文字は、任意のマルチバイト非 ASCII文字より常に小さくなります (Section 33.1 [Text Representations], page 852 を参照)。

```
(string< "abc" "abd")
     ⇒ t
(string< "abd" "abc")
     ⇒ nil
(string< "123" "abc")
     ⇒ t
```

文字列の長さが異なり、*string1*の長さまでマッチする場合、結果は `t`になる。*string2*の長さまでマッチする場合、結果は `nil`になる。文字を含まない文字列は、他の任意の文字列より小さくなる。

```
(string< "" "abc")
     ⇒ t
(string< "ab" "abc")
     ⇒ t
(string< "abc" "")
     ⇒ nil
(string< "abc" "ab")
     ⇒ nil
(string< "" "")
     ⇒ nil
```

引数としてシンボルを指定することもでき、この場合はシンボルのプリント名が比較される。

`string-lessp` *string1 string2* [Function]

`string-lessp`は`string<`の別名である。

`string-greaterp` *string1 string2* [Function]

この関数は逆順で *string1*と *string2*を比較した結果をリターンする。つまりこれは`(string-lessp` ***string2 string1***`)`を呼び出すのと等価である。

`string-collate-lessp` *string1 string2* **&optional** *locale ignore-case* [Function]

この関数は照合順で *string1*が *string2*より小さければ`t`をリターンする。照合順は *string1*と *string2*に含まれる文字の辞書順だけではなく、それらの文字間の関係に関するルールによっても判断される。これは通常は Emacs 実行中の *locale*環境により決定される。

たとえばソートでは区切り文字と空白文字は無視されるだろう (Section 6.1 [Sequence Functions], page 95 を参照)。

```
(sort (list "11" "12" "1 1" "1 2" "1.1" "1.2") 'string-collate-lessp)
     ⇒ ("11" "1 1" "1.1" "12" "1 2" "1.2")
```

Cygwin では locale と無関係に区切り文字と空白文字が無視されることが決してないように、この振る舞いはシステム依存である。

オプション引数 *locale*(文字列) は、照合用のカレント locale 識別子 (current locale identifier) をオーバーライドする。値はシステムに依存する。たとえば POSIX システムでは`"en_US.UTF-8"`、MS-Windows システムでは`"enu_USA.1252"`の *locale*が適用できるだろう。*locale*の値を`"POSIX"`か`"C"`にすると、`string-collate-lessp`は`string-lessp`と同様に振る舞う。

```
(sort (list "11" "12" "1 1" "1 2" "1.1" "1.2")
      (lambda (s1 s2) (string-collate-lessp s1 s2 "POSIX")))
     ⇒ ("1 1" "1 2" "1.1" "1.2" "11" "12")
```

*ignore-case*が非`nil`なら、それらは比較前に小文字に変換される。

MS-Windows システムで Unicode 互換の照合をエミュレートする場合、MS-Windows では locale のコードセット部分を`"UTF-8"`にできないので、`w32-collate-ignore-punctuation`に非`nil`値をバインドすること。

locale 環境をサポートしないシステムでは、この関数は`string-lessp`と同様に振る舞う。

`string-version-lessp` *string1 string2* [Function]

この関数は文字列を辞書順で比較するが、数字のシーケンスを 10 進数で構成されているかのように扱い、その数値を比較する。つまりたとえ辞書順で '`12`'が '`2`'より "小" だとしても、この述語に応じて '`foo12.png`'より '`foo2.png`'が "小" になる。

`string-prefix-p` *string1 string2* **&optional** *ignore-case* [Function]
この関数は *string1*が *string2*のプレフィクス (たとえば *string2*が *string1*で始まる) なら、非`nil`をリターンする。オプションの引数 *ignore-case*が非`nil`ばら、比較において case の違いは無視される。

`string-suffix-p` *suffix string* **&optional** *ignore-case* [Function]
この関数は *suffix*が *string*のサフィックス (たとえば *string*が *suffix*で終わる) なら、非`nil`をリターンする。オプションの引数 *ignore-case*が非`nil`なら、比較において case の違いは無視される。

`compare-strings` *string1 start1 end1 string2 start2 end2* **&optional** *ignore-case* [Function]
この関数は *string1*の指定部分をと *string2*指定部分を比較する。*string1*の指定部分とは、インデックス *start1*(その文字を含む) から、インデックス *end1*(その文字を含まない) まで。*start1*に`nil`を指定すると文字列の最初という意味になり、*end1*に`nil`を指定すると文字列の長さを意味する。同様に *string2*の指定部分とはインデックス *start2*からインデックス *end2*まで。

文字列は文字列内の文字の数値により比較される。たとえば *str1*と *str2*は、最初に異なる文字で *str1*の文字の数値が小さければ小さいと判断される。*ignore-case*が非`nil`なら比較を行なう前に大文字に変換される。比較用にユニバイト文字列はマルチバイト文字列に変換されるので (Section 33.1 [Text Representations], page 852 を参照)、ユニバイト文字列とそれを変換したマルチバイト文字列は常に等しくなる。

2 つの文字列の指定部分がマッチした場合、値は`t`になる。それ以外なら値は整数で、何文字が一致してどちらの文字が小さいかを示す。この値の絶対値は、2 つの文字列の先頭から一致した文字数に 1 加えた値になる。*string1*(または指定部分) のほうが小さければ符号は負になる。

`string-distance` *string1 string2* **&optional** *bytecompare* [Function]
この関数はソース文字列 *string1*とターゲット文字列 *string2*の間のレーベンシュタイン距離 *(Levenshtein distance)* をリターンする。レーベンシュタイン距離はソース文字列をターゲット文字列に変換 (削除、挿入、置換) するために必要な単一文字の個数。これは文字列間の編集距離 *(edit distance)* として使用可能な定義の 1 つである。

計算距離にとって文字列の英字の case(大文字小文字) は意味をもつが、テキストプロパティは無視される。オプション引数 *bytecompare*が非`nil`なら、この関数は文字ではなくバイトで計算する。バイト単位での比較は Emacs の内部的な文字表現を使用するので、raw バイトを含むマルチバイト文字列では不正確な結果を生成するかもしれない (Section 33.1 [Text Representations], page 852 を参照)。raw で正確な結果が必要なら、エンコードして文字列をユニバイトにすること (Section 33.10.7 [Explicit Encoding], page 876 を参照)。

`assoc-string` *key alist* **&optional** *case-fold* [Function]
この関数は`assoc`と同様に機能するが、*key*は文字列かシンボルでなければならず、比較は`compare-strings`を使用して行なわれる。テストする前にシンボルは文字列に変換される。*case-fold*が非`nil`なら、*key*と *alist*の要素は比較前に大文字に変換される。`assoc`とは異なり、この関数はコンスではない文字列またはシンボルの alist 要素もマッチできる。特に *alist*は実際の alist ではなく、文字列またはリストでも可。Section 5.8 [Association Lists], page 88 を参照のこと。

バッファー内のテキストを比較する方法として、Section 32.3 [Comparing Text], page 781 の関数`compare-buffer-substrings`も参照してください。文字列にたいして正規表現のマッチを行なう関数`string-match`も、ある種の文字列比較に使用することができます。Section 34.4 [Regexp Search], page 894 を参照してください。

4.6 文字および文字列の変換

このセクションでは文字、文字列、整数の間で変換を行なう関数を説明します。`format` (Section 4.7 [Formatting Strings], page 61 を参照) と `prin1-to-string` (Section 19.5 [Output Functions], page 337 を参照) も Lisp オブジェクトを文字列に変換できます。`read-from-string` (Section 19.3 [Input Functions], page 334 を参照) は、Lisp オブジェクトの文字列表現をオブジェクトに “変換” できます。関数 `string-to-multibyte`と `string-to-unibyte`は、テキスト表現を文字列に変換します (Section 33.3 [Converting Representations], page 854 を参照)。

テキスト文字と一般的なインプットイベントにたいするテキスト記述を生成する関数 (`single-key-description`と `text-char-description`) については、Chapter 24 [Documentation], page 519 を参照してください。これらの関数は主にヘルプメッセージを作成するために使用されます。

`number-to-string` *number* [Function]

この関数は *number*の 10 進プリント表現からなる文字列をリターンする。引数が負ならリターン値はマイナス記号から開始される。

```
(number-to-string 256)
     ⇒ "256"
(number-to-string -23)
     ⇒ "-23"
(number-to-string -23.5)
     ⇒ "-23.5"
```

`int-to-string`はこの関数にたいする半ば廃れたエイリアスである。

Section 4.7 [Formatting Strings], page 61 の関数 `format`も参照されたい。

`string-to-number` *string* **&optional** *base* [Function]

この関数は *string*内の文字の数値的な値をリターンする。*base*が非 `nil`なら値は 2 以上 16 以下でなければならず、整数はその基数に変換される。*base*が `nil`なら基数に 10 が使用される。浮動少数点数の変換は基数が 10 のときだけ機能する。わたしたちは浮動小数点数にたいして他の基数を実装しない。なぜならこれには多くの作業を要し、その割にその機能が有用には思えないからだ。*string*が整数のように見えるが、その値が Lisp の整数に収まらないほど大きな値なら、`string-to-number`は浮動小数点数の結果をリターンする。

パースでは *string*の先頭にあるスペースとタブはスキップして、与えられた基数で数字として解釈できるところまで *string*を読み取る (スペースとタブだけではなく先頭にある他の空白文字を無視するシステムもある)。*string*を数字として解釈できなければこの関数は 0 をリターンする。

```
(string-to-number "256")
     ⇒ 256
(string-to-number "25 is a perfect square.")
     ⇒ 25
(string-to-number "X256")
     ⇒ 0
(string-to-number "-4.5")
     ⇒ -4.5
(string-to-number "1e5")
     ⇒ 100000.0
```

`string-to-int`はこの関数にたいする半ば廃れたエイリアスである。

`char-to-string` *character* [Function]
この関数は 1 つの文字 *character*を含む新しい文字列をリターンする。関数 `string`のほうがより一般的であり、この関数は半ば廃れている。Section 4.3 [Creating Strings], page 52 を参照のこと。

`string-to-char` *string* [Function]
この関数は *string*の最初の文字をリターンする。これはほとんど `(aref string 0)`と同じで、例外は文字列が空のときに 0 をリターンすること (文字列の最初の文字が ASCIIコード 0 のヌル文字のときも 0 をリターンする)。この関数は残すのに充分なほど有用と思えないければ、将来削除されるかもしれない。

以下は文字列へ／からの変換に使用できるその他の関数です:

`concat` この関数はベクターまたはリストから文字列に変換する。Section 4.3 [Creating Strings], page 52 を参照のこと。

`vconcat` この関数は文字列をベクターに変換する。Section 6.5 [Vector Functions], page 109 を参照のこと。

`append` この関数は文字列をリストに変換する。Section 5.4 [Building Lists], page 75 を参照のこと。

`byte-to-string`
この関数は文字データのバイトをユニバイト文字列に変換する。Section 33.3 [Converting Representations], page 854 を参照のこと。

4.7 文字列のフォーマット

フォーマット *(formatting)* とは、定数文字列内のなまざまな場所を計算された値で置き換えることにより、文字列を構築することを意味します。この定数文字列は他の値がプリントされる方法、同様にどこに表示するかを制御します。これはフォーマット文字列 *(format string)* と呼ばれます。

表示されるメッセージを計算するためにフォーマットが便利なことがしばしばあります。実際に関数 `message`と `error`は、ここで説明する機能と同じフォーマットを提供します。これらの関数と `format-message`の違いはフォーマットされた結果を使用する方法だけです。

`format` *string* **&rest** *objects* [Function]
この関数は *string*のすべてのフォーマット仕様を、対応する *objects*を復号化したものと置換したものと等しい文字列をリターンする。引数 *objects*はフォーマットされる計算値。

(もしあれば)*string*内のフォーマット仕様以外の文字はテキストプロパティーを含めて出力に直接コピーされる。フォーマット仕様のすべてのテキストプロパティは引数 *objects*を表現する生成された文字列にコピーされる。

出力される文字列は新規に割り当てられる必要はない。たとえば `x`が文字列`"foo"`なら `(eq x (format x))`と `(eq x (format "%s" x))`はいずれも `t`となるだろう。

`format-message` *string* **&rest** *objects* [Function]
この関数は `format`と同様に機能するが、*string*内のすべてのグレイブアクセント (‘) とアポストロフィー (’) を `text-quoting-style`の各値に応じて変換する点が異なる。

フォーマット内のグレイブアクセントとアポストロフィーはマッチする curved quotes に変換される (`"Missing ‘%s’"`は`"Missing ‘foo’"`という結果になる) この変換の影響と回避については Section 24.4 [Text Quoting Style], page 524 を参照のこと。

フォーマット仕様 (format specification) は‘%’で始まる文字シーケンスです。したがって *string* 内に‘%d’がると `format`はそれを、フォーマットされる値の 1 つ (引数 *objects*のうちの 1 つ) にたいするプリント表現で置き換えます。たとえば:

```
(format "The value of fill-column is %d." fill-column)
     ⇒ "The value of fill-column is 72."
```

`format`は文字‘%’をフォーマット仕様と解釈するので、決して最初の引数に不定な文字列 (arbitrary string) を渡すべきではありません。これは特に何らかの Lisp コード ga 生成 si た文字列の場合に当てはまります。その文字列が決して文字‘%’を含まないと確信できないならば、以下で説明するように最初の引数に`"%s"`を渡して、不定な文字列を 2 番目の引数として渡します:

```
(format "%s" arbitrary-string)
```

ある種のフォーマット仕様は特定の型の値を要求します。その要求に適合しない値を与えた場合にはエラーがシグナルされます。

以下は有効なフォーマット仕様のテーブルです:

‘%s’ フォーマット仕様を、クォートなしのオブジェクトのプリント表現で置き換える (つまり `prin1`ではなく `princ`を使用して置き換える。Section 19.5 [Output Functions], page 337 を参照されたい)。したがって文字列は‘"’文字なしの文字列内容だけが表示され、シンボルは‘\’文字なしで表される。

オブジェクトが文字列なら文字列のプロパティーは出力にコピーされる。‘%s’のテキストプロパティー自身もコピーされるが、オブジェクトのテキストプロパティーが優先される。

‘%S’ フォーマット仕様を、クォートありのオブジェクトのプリント表現で置き換える (つまり `prin1`を使用して変換する。Section 19.5 [Output Functions], page 337 を参照されたい)。したがって文字列は‘"’文字で囲まれ、必要となる特別文字の前に‘\’文字が表示される。

‘%o’ フォーマット仕様を整数の 8 進表現に置き換える。負の整数はプラットフォーム依存の方法でフォーマットされる。オブジェクトは浮動小数点数 (少数部分を切り捨てて整数にフォーマット) でもよい。

‘%d’ フォーマット仕様を 10 進表現の符号すき整数で置き換える。オブジェクトは浮動小数点数 (少数部分を切り捨てて整数にフォーマット) でもよい。

‘%x’
‘%X’ フォーマット仕様を 16 進表現の整数で置き換える。負の整数はプラットフォーム依存の方法でフォーマットされる。‘%x’なら小文字、‘%X’なら大文字が使用される。オブジェクトは少数部分を切り捨てて整数にフォーマットされた浮動小数点数でもよい。

‘%c’ フォーマット仕様を与えられた値の文字で置き換える。

‘%e’ フォーマット仕様を浮動小数点数の指数表現で置き換える。

‘%f’ フォーマット仕様を浮動小数点数にたいする 10 進少数表記で置き換える。

‘%g’ 指数表記か小数点表記のいずれかを使用してフォーマット仕様を浮動小数点数にたいする表記に置き換える。指数が -4 未満または精度 (デフォルトは 6) 以上なら指数表記を使用する。デフォルトでは結果の少数部の末尾の 0 は削除されて、小数点が現れるのは後に数字が続く場合のみ。

‘%%’ フォーマット仕様を 1 つの ‘%’で置き換える。このフォーマット仕様は唯一のフォームが素の ‘%%’であり値を使用しないという点で特殊。たとえば (format "%% %d" 30)は"% 30"をリターンする。

他のフォーマット文字は ‘Invalid format operation’エラーとなります。

以下は典型的な text-quoting-styleのセッティングを想定した場合の例です:

```
(format "The octal value of %d is %o,
         and the hex value is %x." 18 18 18)
     ⇒ "The octal value of 18 is 22,
         and the hex value is 12."

(format-message
 "The name of this buffer is ‘%s’." (buffer-name))
     ⇒ "The name of this buffer is ‘strings-ja.texi’."

(format-message
 "The buffer object prints as `%s'." (current-buffer))
     ⇒ "The buffer object prints as ‘strings-ja.texi’."
```

フォーマット仕様はデフォルトでは *objects*から連続して値を引き当てます。つまり *string*内の 1 番目のフォーマット仕様は 1 番目の値、2 番目のフォーマット仕様は 2 番目の値、... を使用します。余分なフォーマット仕様 (対応する値がない場合) にはエラーとなります。フォーマットされる値が余分にある場合には無視されます。

フォーマット仕様はフィールド番号 *(field number)* をもつことができます。これは最初の ‘%’の直後に 10 進数字、その後にドル記号 ‘$’が続きます。これにより次の引数ではなく与えられた番号の引数をフォーマット仕様に変換させることができます。フィールド番号は 1 から始まります。フォーマットのフォーマット仕様が番号を含むことも含まないことも可能ですが、両方を含むことはできません。ただし例外は ‘%%’であり、これは番号付きのフォーマット仕様と混交できます。

```
(format "%2$s, %3$s, %%, %1$s" "x" "y" "z")
     ⇒ "y, z, %, x"
```

‘%’とすべてのフィールド番号の後にフラグ文字 *(flag characters)* を配置できます。

フラグ ‘+’は非負の数の前にプラス符号を挿入するので、数には常に符号がつきます。フラグとしてスペースを指定すると、非負の数の前に 1 つのスペースが挿入されます (それ以外は非負の数は最初の数字から開始される)。これらのフラグは非負の数と負数にたいして確実に同じ列数を使用させるために有用です。これらは ‘%d’、‘%e’、‘%f’、‘%g’以外では無視され、両方が指定された場合は ‘+’が優先されます。

フラグ ‘#’は代替形式 (alternate form) を指定します。これは使用するフォーマットに依存します。‘%o’にたいしては結果を ‘0’で開始させます。‘%x’と ‘%X’にたいしては非 0 の結果のプレフィクスは ‘0x’または ‘0X’になります。‘%e’、‘%f’にたいしての ‘#’フラグは、少数部が 0 のときにも小数点が含まれることを意味します。‘%g’にたいしては常に小数点が含まれるとともに、それ以外なら削除される小数点の後の末尾のすべての 0 も強制的に残されます。

フラグ ‘0’はスペースの代わりに文字 ‘0’でパディングします。このフラグは ‘%s’、‘%S’、‘%c’のような非数値のフォーマット仕様文字では無視されます。これらのフォーマット仕様文字で ‘0’フラグを指定できますが、それでもスペースでパディングされます。

フラグ ‘-’はフィールド幅指定子により挿入されるすべてのパディングに作用して、もしパディングが指定された場合には左側ではなく右側にパディングされます。‘-’と ‘0’の両方が指定されると ‘0’フラグは無視されます。

```
(format "%06d is padded on the left with zeros" 123)
     ⇒ "000123 is padded on the left with zeros"

(format "'%-6d' is padded on the right" 123)
     ⇒ "'123   ' is padded on the right"

(format "The word '%-7s' actually has %d letters in it."
        "foo" (length "foo"))
     ⇒ "The word 'foo    ' actually has 3 letters in it."
```

フォーマット仕様はフィールド幅 *(width)* をもつことができます。これはすべてのフィールド番号とフラグの後にある 10 進の数字です。オブジェクトのプリント表現がこのフィールド幅より少ない文字w含む場合には、formatはパディングによりフィールド幅に拡張します。フォーマット仕様 '%%' ではフィールド幅の指定は無視されます。フィールド幅指定子により行なわれるパディングは、通常は左側に挿入されるスペースで構成されます:

```
(format "%5d is padded on the left with spaces" 123)
     ⇒ "  123 is padded on the left with spaces"
```

フィールド幅が小さすぎる場合でも formatはオブジェクトのプリント表現を切り詰めません。したがって情報を失う危険を犯すことなく、フィールドの最小幅を指定することができます。以下の 2 つの例では '%7s' は最小幅に 7 を指定します。1 番目の例では '%7s' に挿入される文字列は 3 文字だけなので、4 つのブランクスペースによりパディングされます。2 番目の例では文字列"specification" は 13 文字ですが切り詰めはされません。

```
(format "The word '%7s' has %d letters in it."
        "foo" (length "foo"))
     ⇒ "The word '    foo' has 3 letters in it."
(format "The word '%7s' has %d letters in it."
        "specification" (length "specification"))
     ⇒ "The word 'specification' has 13 letters in it."
```

すべてのフォーマット仕様文字にはフィールド番号、フラグ、フィールド幅の後にオプションで精度 *(precision)* を指定できます。精度は小数点 '.' と、その後に桁文字列 (digit-string) を指定します。浮動少数点数のフォーマット仕様 ('%e' と '%f') では、精度は表示する小数点以下の桁数を指定します。0 なら小数点も省略されます。%g の精度が 0 か未指定なら 1 として扱われます。'%s' と '%S' では精度として与えられた幅に文字列が切り詰められるので、'%.3s' では *object* の表現の最初の 3 文字だけが表示されます。その他の仕様文字では、printfファミリーのローカルライブラリーが生成する精度の効果が表れます。

'%s' と '%S' にたいしては、文字列を精度で指定された幅に切り詰めます。したがって '%.3s' では、*object* にたいするプリント表現の最初の 3 文字だけが表示されます。他のフォーマット仕様文字にたいしては、精度の効果はローカルライブラリーの printf関数ファミリーが生成する効果となります。

フォーマット済みの値のコピーを取得するために後で readを使用する予定なら、readが値を再構築する仕様を使用してください。この逆手順で数値をフォーマットするには '%s' と '%S'、整数だけなら '%d'、非負の整数なら '#x%x' と '#o%o' も使用できます。その他のフォーマットでは問題があるかもしれません。たとえば '%d' と '%g' は NaN を誤って処理したり精度や型を失うかもしれず、'#x%x' と '#o%o' は負の整数を誤って処理するかもしれません。Section 19.3 [Input Functions], page 334 を参照してください。

このセクションでは仕様文字の固定セットを受け取る関数を説明します。次のセクションでは '%a' や '%z' のようなカスタム仕様文字を受け取ることができる関数 format-specを説明します。

4.8 カスタムフォーマット文字列

ユーザーや Lisp プログラムが、カスタムフォーマットの制御文字列を介して特定のテキストが生成される方法を制御できるようにすると便利な場合があります。たとえばフォーマット文字列は人の姓や名、email アドレスを表示する方法を制御できます。前のセクションで説明した関数 `format`を使用することにより、フォーマット文字列は`"%s %s <%s>"`のようになるかもしれません。しかしこのアプローチはどの仕様文字がどの情報に対応するかが不明瞭なのですぐに非実用的になります。

そのような場合には`"%f %l <%e>"`のようなフォーマット文字列のほうが便利かもしれません。このフォーマット文字列では仕様文字それぞれがより意味的な情報をもち、他の仕様文字に関連して簡単に再配置できるので、このようなフォーマット文字列はユーザーにより簡単にカスタマイズできます。

このセクションで説明する関数 `format-spec`は `format`と同様の機能を処理しますが、任意の仕様文字を使用するフォーマットコントロール文字列を処理する点が異なります。

`format-spec` *template spec-alist* **&optional** *only-present* [Function]

この関数は *spec-alist*で指定された変換にしたがってフォーマット文字列 *template*から生成された文字列をリターンする。*spec-alist*は `(letter . replacement)` という形式の alist(Section 5.8 [Association Lists], page 88 を参照)。*template*内の仕様`%letter`はそれぞれ結果文字列のフォーマット時に置換される。

(もしあれば)*template*内のフォーマット仕様以外の文字はテキストプロパティーを含めて出力に直接コピーされる。フォーマット仕様のすべてのテキストプロパティは置換先にコピーされる。

変換の指定に alist を使用することによって有用な特性がいくつか生成される:

- *template*内に存在する一意な仕様文字より多くの一意な *letter*キーが *spec-alist*に含まれていると、使用されないキーは単に無視される。
- 同じ *letter*にたいして複数の連想値が *spec-alist*に含まれていると、リスト先頭にもっとも近いものを使用する。
- *template*に同じ仕様文字が複数個含まれている場合には、その文字のすべての置換の基礎として *spec-alist*内で見つけた *replacement*を使用する。
- *template*内の仕様の順序は、*spec-alist*内の連想値の順序に対応する必要はない。

オプション引数 *only-present*は、*spec-alist*で見つからない *template*内の仕様文字の処理方法を示す。`nil`か省略なら、関数はエラーをシグナルする。それ以外ならこれらのフォーマット仕様および *template*内のすべての '`%%`'は、(もしあれば) テキストプロパティも含めてそのまま出力される。

`format-spec`が受け取るフォーマット仕様の構文は `format`が受け取るフォーマット仕様と似ていますが同一ではありません。いずれの場合でもフォーマット仕様は '`%`'で始まり '`s`'のようなアルファベット文字で終わる文字シーケンスです。

仕様文字の固定セットに特定の意味を割り当てる `format`とは異なり、`format-spec`は任意の仕様文字を受け取ってそれらをすべて等しく扱います。たとえば:

```
(setq my-site-info
      (list (cons ?s system-name)
            (cons ?t (symbol-name system-type))
            (cons ?c system-configuration)
            (cons ?v emacs-version)
            (cons ?e invocation-name)
            (cons ?p (number-to-string (emacs-pid)))
            (cons ?a user-mail-address)
            (cons ?n user-full-name)))

(format-spec "%e %v (%c)" my-site-info)
     ⇒ "emacs 27.1 (x86_64-pc-linux-gnu)"

(format-spec "%n <%a>" my-site-info)
     ⇒ "Emacs Developers <emacs-devel@gnu.org>"
```

フォーマット仕様には置換の様相を変更するために、'%'の直後に任意個数のフラグ文字を含めることができます。

'0' このフラグは指定された幅のパディングをスペースのかわりに '0'で構成する。

'-' このフラグは指定された幅のパディングを左側ではなく右側に挿入する。

'<' このフラグはもし幅が指定されたら置換の左側を切り捨てる。

'>' このフラグはもし幅が指定されたら置換の右側を切り捨てる。

'^' このフラグは置換されるテキストを大文字に変換する (Section 4.9 [Case Conversion], page 67 を参照)。

'_' このフラグは置換されるテキストを小文字に変換する (Section 4.9 [Case Conversion], page 67 を参照)。

矛盾したフラグ (たとえば大文字と小文字) を使用した場合の結果は未定義です。

`format`の場合と同様に、任意のフラグの後の 10 進数値として幅指定子をフォーマット仕様に含めることができます。指定した幅より置換の文字が少なければ左側がパディングされます。

```
(format-spec "%8a is padded on the left with spaces"
             '((?a . "alpha")))
     ⇒ "   alpha is padded on the left with spaces"
```

以下は前述の機能をいくつか組み合わせたより複雑な例です:

```
(setq my-battery-info
      (list (cons ?p "73")      ; パーセント表示
            (cons ?L "Battery") ; 状態
            (cons ?t "2:23")    ; 残り時間
            (cons ?c "24330")   ; 容量
            (cons ?r "10.6")))  ; 放電率

(format-spec "%>^-3L : %3p%% (%05t left)" my-battery-info)
     ⇒ "BAT :  73% (02:23 left)"

(format-spec "%>^-3L : %3p%% (%05t left)"
             (cons (cons ?L "AC")
                   my-battery-info))
     ⇒ "AC  :  73% (02:23 left)"
```

このセクションの例で示したように、`format-spec`はさまざまな情報の断片を選択的にフォーマットするために頻繁に使用されます。これはプログラムが可能にする情報のサブセットだけをユーザーが通常の構文で望む順序で選択できるように、ユーザーにカスタマイズ可能なフォーマット文字列を提供するプログラムにとって有用です。

4.9 Lisp での大文字小文字変換

case 変換関数 (character case functions) は、1 つの文字または文字列中の大文字小文字を変換します。関数は通常、アルファベット文字 (英字 '`A`'から '`Z`'と '`a`'から '`z`'、同様に非 ASCIIの英字) だけを変換し、それ以外の文字は変換しません。case テーブル (case table。Section 4.10 [Case Tables], page 69 を参照されたい) で指定することにより、case の変換に異なるマッピングを指定できます。

これらの関数は引数として渡された文字列は変更しません。

以下の例では文字 '`X`'と '`x`'を使用します。これらの ASCIIコードは 88 と 120 です。

`downcase` *string-or-char* [Function]

この関数は *string-or-char*(文字か文字列) を小文字に変換する。

*string-or-char*が文字列なら、この関数は引数の大文字を小文字に変換した新しい文字列をリターンする。*string-or-char*が文字なら、この関数は対応する小文字 (整数) をリターンする。元の文字が小文字か非英字ならリターン値は元の文字と同じ。

```
(downcase "The cat in the hat")
     ⇒ "the cat in the hat"

(downcase ?X)
     ⇒ 120
```

`upcase` *string-or-char* [Function]

この関数は *string-or-char*(文字か文字列) を大文字に変換する。

*string-or-char*が文字列なら、この関数は引数の小文字を大文字に変換した新しい文字列をリターンする。*string-or-char*が文字なら、この関数は対応する大文字 (整数) をリターンする。元の文字が大文字か非英字ならリターン値は元の文字と同じ。

```
(upcase "The cat in the hat")
     ⇒ "THE CAT IN THE HAT"
```

```
(upcase ?x)
     ⇒ 88
```

`capitalize` *string-or-char* [Function]

この関数は文字列や文字をキャピタライズ (capitalize: 先頭が大文字で残りは小文字) する。この関数は *string-or-char*が文字列なら *string-or-char*の各単語をキャピタライズした新たなコピーをリターンする。これは各単語の最初の文字が大文字に変換され、残りは小文字に変換されることを意味する。

単語の定義はカレント構文テーブル (current syntax table) の単語構成構文クラス (word constituent syntax class) に割り当てられた、連続する文字の任意シーケンスである (Section 35.2.1 [Syntax Class Table], page 907 を参照)。

*string-or-char*が文字ならこの関数は `upcase`と同じことを行なう。

```
(capitalize "The cat in the hat")
     ⇒ "The Cat In The Hat"

(capitalize "THE 77TH-HATTED CAT")
     ⇒ "The 77th-Hatted Cat"

(capitalize ?x)
     ⇒ 88
```

`upcase-initials` *string-or-char* [Function]

この関数は *string-or-char*が文字列なら、*string-or-char*の中の単語の頭文字をキャピタライズして、頭文字以外の文字は変更しない。この関数は *string-or-char*の各単語の頭文字が大文字に変換された新しいコピーをリターンする。

単語の定義はカレント構文テーブル (current syntax table) の単語構成構文クラス (word constituent syntax class) に割り当てられた、連続する文字の任意シーケンスである (Section 35.2.1 [Syntax Class Table], page 907 を参照)。

`upcase-initials`の引数が文字なら、`upcase-initials`の結果は `upcase`と同じ。

```
(upcase-initials "The CAT in the hAt")
     ⇒ "The CAT In The HAt"
```

case 変換コードポイントを 1 対 1 でマップするものではなく、結果の文字列長は引数の文字列長と異なるかもしれません。さらに文字を渡すことによりリターンされる型にも文字が強制されるので、関数は正しい置換を行えずに 1 文字の文字列を処理する場合とは結果が異なるかもしれません。たとえば:

```
(upcase "fi")  ; 注意: 1 文字の合字 "fi"
     ⇒ "FI"
(upcase ?fi)
     ⇒ 64257  ; つまり ?fi
```

これを避けるためには case 関数のいずれかに文字を渡す前に `string`関数を使用して文字列に変換しなければなりません。もちろん結果の長さについて仮定はできません。

このような特殊ケースのマッピングは `special-uppercase`、`special-lowercase`、`special-titlecase`から取得されます。Section 33.6 [Character Properties], page 857 を参照してください。

文字列を比較する関数 (case の違いを無視するものや、オプションで case の違いを無視できるもの) については、Section 4.5 [Text Comparison], page 56 を参照されたい。

4.10 case テーブル

特別な *case* テーブル *(case table)* をインストールすることにより、case の変換をカスタマイズできます。case テーブルは大文字と小文字の間のマッピングを指定します。case テーブルは Lisp オブジェクトにたいする case 変換関数 (前のセクションを参照) と、バッファー内のテキストに適用される関数の両方に影響します。それぞれのバッファーには case テーブルがあります。新しいバッファーの case テーブルを初期化するために使用される、標準の case テーブル (standard case table) もあります。

case テーブルは、サブタイプが `case-table`の文字テーブル (char-table。Section 6.6 [Char-Tables], page 110 を参照) です。この文字テーブルはそれぞれの文字を対応する小文字にマップします。case テーブルは、関連するテーブルを保持する 3 つの余分なスロットをもちます:

upcase upcase(大文字) テーブルはそれぞれの文字を対応する大文字にマップする。

canonicalize
: canonicalize(正準化) テーブルは、case に関連する文字セットのすべてを、その文字セットの特別なメンバーにマップする。

equivalences
: equivalence(同値) テーブルは、大の字小文字に関連した文字セットのそれぞれを、そのセットの次の文字にマップする。

単純な例では、小文字へのマッピングを指定することだけが必要です。3 つの関連するテーブルは、このマッピングから自動的に計算されます。

大文字と小文字が 1 対 1 で対応しない言語もいくつかあります。これらの言語では、2 つの異なる小文字が同じ大文字にマップされます。このような場合、大文字と小文字の両方にたいするマップを指定する必要があります。

追加の *canonicalize*テーブルは、それぞれの文字を正準化された等価文字にマップします。case に関連する任意の 2 文字は、同じ正準等価文字 (canonical equivalent character) をもちます。たとえば ‘`a`’と ‘`A`’は case 変換に関係があるので、これらの文字は同じ正準等価文字 (両方の文字が ‘`a`’、または両方の文字が ‘`A`’) をもつべきです。

追加の *equivalences*テーブルは、等価クラスの文字 (同じ正準等価文字をもつ文字) それぞれを循環的にマップします (通常の ASCIIでは、これは ‘`a`’を ‘`A`’に ‘`A`’を ‘`a`’にマップし、他の等価文字セットにたいしても同様にマップする)。

case テーブルを構築する際は、*canonicalize*に `nil`を指定できます。この場合、Emacs は大文字と小文字のマッピングでこのスロットを充填します。*equivalences*にたいして `nil`を指定することもできます。この場合、Emacs は *canonicalize*からこのスロットを充填します。実際に使用される case テーブルでは、これらのコンポーネントは非 `nil`です。*canonicalize*を指定せずに *equivalences*を指定しないでください。

以下は case テーブルに作用する関数です:

`case-table-p` *object* [Function]
: この述語は、*object*が有効な case テーブルなら非 `nil`をリターンする。

`set-standard-case-table` *table* [Function]
: この関数は *table*を標準 case テーブルにして、これ以降に作成される任意のバッファーにたいしてこのテーブルが使用されるようにする。

`standard-case-table` [Function]
これは標準 case テーブル (standard case table) をリターンする。

`current-case-table` [Function]
この関数はカレントバッファーの case テーブルをリターンする。

`set-case-table` *table* [Function]
これはカレントバッファーの case テーブルを *table*にセットする。

`with-case-table` *table body...* [Macro]
`with-case-table`マクロはカレント case テーブルを保存してから、*table*をカレント case テーブルにセットし、その後に *body*フォームを評価してから、最後に case テーブルをリストアします。リターン値は、*body*の最後のフォームの値です。`throw`かエラー (Section 11.7 [Nonlocal Exits], page 162 を参照) により異常終了した場合でも、case テーブルはリストアされます。

ASCII文字の case 変換を変更する言語環境 (language environment) がいくつかあります。たとえばトルコ語の言語環境では、ASCIIの大文字 'I'にたいする小文字は、トルコ語のドットがない i('`^^c4^^b1`') です。これは (ASCIIベースのネットワークプロトコル実装のような)ASCIIの通常の case 変換を要求するコードに干渉する可能性があります。このような場合には、変数 *ascii-case-table* にたいして `with-case-table`マクロを使用してください。これにより変更されていない ASCII文字セットの case テーブルが保存されます。

`ascii-case-table` [Variable]
ASCII文字セットにたいする case テーブル。すべての言語環境セッティングにおいて、これを変更するべきではない。

以下の 3 つの関数は、非 ASCII文字セットを定義するパッケージにたいして便利なサブルーチンです。これらは *case-table*に指定された case テーブルを変更します。これは標準構文テーブルも変更します。Chapter 35 [Syntax Tables], page 906 を参照してください。通常これらの関数は、標準 case テーブルを変更するために使用されます。

`set-case-syntax-pair` *uc lc case-table* [Function]
この関数は対応する文字のペアー (一方は大文字でもう一方は小文字) を指定する。

`set-case-syntax-delims` *l r case-table* [Function]
この関数は文字 *l*と *r*を、case 不変区切り (case-invariant delimiter) のマッチングペアーとする。

`set-case-syntax` *char syntax case-table* [Function]
この関数は *char*を構文 *syntax*の case 不変 (case-invariant) とする。

`describe-buffer-case-table` [Command]
このコマンドはカレントバッファーの case テーブルの内容にたいする説明を表示する。

5 リスト

リスト *(list)* は 0 個以上の要素 (任意の Lisp オブジェクト) のシーケンスを表します。リストとベクターの重要な違いは、2 つ以上のリストが構造の一部を共有できることです。加えて、リスト全体をコピーすることなく要素の挿入と削除ができます。

5.1 リストとコンスセル

Lisp でのリストは基本データ型ではありません。リストはコンスセル *(cons cells)* から構築されます (Section 2.4.6 [Cons Cell Type], page 15 を参照)。コンスセルは順序つきペアを表現するデータオブジェクトです。つまりコンスセルは 2 つのスロットをもち、それぞれのスロットは Lisp オブジェクトを保持 *(holds)* または参照 *(refers to)* します。1 つのスロットは CAR、もう 1 つは CDR です (これらの名前は歴史的なものである。Section 2.4.6 [Cons Cell Type], page 15 を参照されたい)。CDR は “could-er(クダー)” と発音します。

わたしたちは、コンスセルの CAR スロットに現在保持されているオブジェクトが何であれ、“このコンスセルの CAR は、...” のような言い方をします。これは CDR の場合でも同様です。

リストとは互いに連なる (chained together) 一連のコンスセルであり、各セルは次のセルを参照します。リストの各要素にたいして 1 つのコンスセルがあります。慣例によりコンスセルの CAR はリストの要素を保持し、CDR はリストをチェーンするのに使用されます (CAR と CDR の間の非対称性は完全に慣例的なものである。コンスセルのレベルでは CAR スロットと CDR スロットは同じようなプロパティーをもつ)。したがって、リスト内の各コンスセルの CDR スロットは次のコンスセルを参照します。

これも慣例的なものですがリスト内の最後のコンスセルの CDR は `nil`です。わたしたちはこのような `nil`で終端された構造を正リスト *(proper list)* と呼びます [1]。Emacs Lisp ではシンボル `nil`はシンボルであり、かつ要素なしのリストでもあります。便宜上、シンボル `nil`はその CDR(と CAR) に `nil`をもつと考えます。

したがって正リストの CDR は常に正リストです。空でない正リストの CDR は 1 番目の要素以外を含む正リストです。

リストの最後のコンスセルの CDR が `nil`以外の何らかの値の場合、このリストのプリント表現はドットペア表記 (dotted pair notation。Section 2.4.6.2 [Dotted Pair Notation], page 17 を参照のこと) を使用するので、わたしたちはこの構造をドットリスト *(dotted list)* と呼びます。他の可能性もあります。あるコンスセルの CDR が、そのリストのそれより前にある要素を指すかもしれません。わたしたちは、この構造を循環リスト *(circular list)* と呼びます。

ある目的においてはそのリストが正リストか循環リストなのか、あるいはドットリストなのかが問題にならない場合もあります。そのプログラムがリストを充分に辿って最後のコンスセルの CDR を確認しようとしないなら、これは問題になりません。しかしリストを処理する関数のいくつかは正リストを要求し、ドットリストの場合はエラーをシグナルします。リストの最後を探そうと試みる関数のほとんどは循環リストを与えると無限ループに突入します。

ほとんどのコンスセルはリストの一部として使用されるので、わたしたちはコンスセルで構成される任意の構造をリスト構造 *(list structure)* と呼びます。

---

[1] これは真リスト *(true list)* と呼ばれることもありますが、このマニュアルでは一般的にこの用語を使用しません。

5.2 リストのための述語

以下の述語はある Lisp オブジェクトがアトムか、コンスセルか、リストなのか、またはオブジェクトが nilかどうかテストします (これらの述語の多くは他の述語で定義することもできるが、多用されるので個別に定義する価値がある)。

`consp` *object* [Function]

この関数は *object*がコンスセルなら t、それ以外は nilをリターンする。たとえ nilがリストであっても、コンスセルではない。

`atom` *object* [Function]

この関数は *object*がアトムなら t、それ以外は nilをリターンする。シンボル nilはアトムであり、かつリストでもある。そのような Lisp オブジェクトは nilだけである。

```
(atom object) ≡ (not (consp object))
```

`listp` *object* [Function]

この関数は *object*がコンスセルか nilなら t、それ以外は nilをリターンする。

```
(listp '(1))
     ⇒ t
(listp '())
     ⇒ t
```

`nlistp` *object* [Function]

この関数は listpの反対である。*object*がリストでなければ t、それ以外は nilをリターンする。

```
(listp object) ≡ (not (nlistp object))
```

`null` *object* [Function]

この関数は *object*が nilなら t、それ以外は nilをリターンする。この関数は notと等価だが、明解にするために *object*をリストだと考えるときは null、真偽値だと考えるときは notを使用すること (Section 11.3 [Combining Conditions], page 148 の notを参照)。

```
(null '(1))
     ⇒ nil
(null '())
     ⇒ t
```

`proper-list-p` *object* [Function]

この関数は *object*が適正なリストなら *object*の長さ、それ以外は nilをリターンする (Section 5.1 [Cons Cells], page 71 を参照)。適正なリストとは listpを満足することに加えて、循環リストやドットリストでもない。

```
(proper-list-p '(a b c))
    ⇒ 3
(proper-list-p '(a b . c))
    ⇒ nil
```

5.3 リスト要素へのアクセス

`car` *cons-cell* [Function]

この関数はコンスセル *cons-cell*の 1 番目のスロットが参照する値をリターンする。言い換えるとこの関数は *cons-cell*の CAR をリターンする。

特別なケースとして *cons-cell*が `nil`の場合、この関数は `nil`をリターンする。したがってリストはすべて引数として有効である。引数がコンスセルでも `nil`でもなければエラーがシグナルされる。

```
(car '(a b c))
     ⇒ a
(car '())
     ⇒ nil
```

`cdr` *cons-cell* [Function]

この関数はコンスセル *cons-cell*の 2 番目のスロットにより参照される値をリターンする。言い換えるとこの関数は *cons-cell*の CDR をリターンする。

特別なケースとして *cons-cell*が `nil`の場合、この関数は `nil`をリターンする。したがってリストはすべて引数として有効である。引数がコンスセルでも `nil`でもければエラーがシグナルされる。

```
(cdr '(a b c))
     ⇒ (b c)
(cdr '())
     ⇒ nil
```

`car-safe` *object* [Function]

この関数により他のデータ型によるエラーを起こさずに、コンスセルの CAR を取得できり。この関数は *object*がコンスセルなら *object*の CAR、それ以外は `nil`をリターンする。この関数は、*object*がリストでなければエラーをシグナルする `car`とは対象的である。

```
(car-safe object)
≡
(let ((x object))
  (if (consp x)
      (car x)
    nil))
```

`cdr-safe` *object* [Function]

この関数により他のデータ型によるエラーを起こさずに、コンスセルの CDR を取得できる。この関数は *object*がコンスセルなら *object*の CDR、それ以外は `nil`をリターンする。この関数は、*object*がリストでないときはエラーをシグナルする `cdr`とは対象的である。

```
(cdr-safe object)
≡
(let ((x object))
  (if (consp x)
      (cdr x)
    nil))
```

`pop` *listname* [Macro]

このマクロはリストの CAR を調べて、それをリストから取り去るのを一度に行なう便利な方法を提供する。この関数は *listname*に格納されたリストにたいして処理を行なう。この関数はリストから 1 番目の要素を削除して、CDR を *listname*に保存し、その後で削除した要素をリターンする。

もっとも単純なケースは、リストに名前をつけるためのクォートされていないシンボルの場合である。この場合、このマクロは `(prog1 (car listname) (setq listname (cdr listname)))` と等価である。

```
x
     ⇒ (a b c)
(pop x)
     ⇒ a
x
     ⇒ (b c)
```

より一般的なのは *listname*が汎変数 (generalized variable) の場合である。この場合、このマクロは `setf`を使用して *listname*に保存する。Section 12.17 [Generalized Variables], page 205 を参照のこと。

リストに要素を追加する `push`マクロについては Section 5.5 [List Variables], page 79 を参照のこと。

`nth` *n list* [Function]

この関数は *list*の *n*番目の要素をリターンする。要素は 0 から数えられるので *list*の CAR は要素 0 になる。*list*の長さが *n*以下なら値は `nil`。

```
(nth 2 '(1 2 3 4))
     ⇒ 3
(nth 10 '(1 2 3 4))
     ⇒ nil

(nth n x) ≡ (car (nthcdr n x))
```

これは関数 `elt`も類似しているが、任意の種類のシーケンスに適用される。歴史的な理由によりこの関数は逆の順序で引数を受け取る。Section 6.1 [Sequence Functions], page 95 を参照のこと。

`nthcdr` *n list* [Function]

この関数は *list*の *n*番目の CDR をリターンする。言い換えると、この関数は *list*の最初の *n*個のリンクをスキップしてから、それ以降をリターンする。

*n*が 0 なら `nthcdr`は *list*全体をリターンする。*list*の長さが *n*以下なら `nthcdr`は `nil`をリターンする。

```
(nthcdr 1 '(1 2 3 4))
     ⇒ (2 3 4)
(nthcdr 10 '(1 2 3 4))
     ⇒ nil
(nthcdr 0 '(1 2 3 4))
     ⇒ (1 2 3 4)
```

`last` *list* **&optional** *n* [Function]
この関数は *list*の最後のリンクをリターンする。このリンクの `car`はこのリストの最後の要素。*list*が null なら `nil`がリターンされる。*n*が非 `nil`なら *n*番目から最後までのリンクがリターンされる。*n*が *list*の長さより大きければ *list*全体がリターンされる。

`safe-length` *list* [Function]
この関数はエラーや無限ループの危険なしで、*list*の長さをリターンする。この関数は一般的に、リスト内のコンスセルの個数をリターンする。しかし循環リストでは単に上限値が値となるため、非常に大きくなる場合があります。

*list*が `nil`] とコンスセルのいずれでもなければ `safe-length`は 0 をリターンする。

循環リストを考慮しなくてもよい場合にリストの長さを計算するもっとも一般的な方法は、`length`を使う方法です。Section 6.1 [Sequence Functions], page 95 を参照してください。

`caar` *cons-cell* [Function]
これは `(car (car cons-cell))`と同じ。

`cadr` *cons-cell* [Function]
これは `(car (cdr cons-cell))`か `(nth 1 cons-cell)`と同じ。

`cdar` *cons-cell* [Function]
これは `(cdr (car cons-cell))`と同じ。

`cddr` *cons-cell* [Function]
これは `(cdr (cdr cons-cell))`か `(nthcdr 2 cons-cell)`と同じ。

上記に加えて `cxxxr`や `cxxxxr`のような `car`と `cdr`で構成される 24 の関数が定義されています。ここで *x*は `a`か `d`のいずれかです。`cadr`と `caddr`と `cadddr`はそれぞれリストの 2 つ目、3 つ目、4 つ目の要素です。`cl-lib`は同じものを `cl-second`、`cl-third`、`cl-fourth`という名前で提供しています。Section “List Functions” in *Common Lisp Extensions* を参照してください。

`butlast` *x* **&optional** *n* [Function]
この関数はリスト *x*から、最後の要素か最後の *n*個の要素を削除してリターンする。*n*が 0 より大きければこの関数はリストのコピーを作成するので、元のリストに影響はない。一般的に `(append (butlast x n) (last x n))`は、*x*と等しいリストをリターンする。

`nbutlast` *x* **&optional** *n* [Function]
この関数はリストのコピーを作成するのではなく、`cdr`を適切な要素に変更することにより破壊的に機能するバージョンの `butlast`である。

5.4 コンスセルおよびリストの構築

リストは Lisp の中核にあたる機能なので、リストを構築するために多くの関数があります。`cons`はリストを構築する基本的な関数です。しかし Emacs のソースコードでは、`cons`より `list`のほうが多く使用されているのは興味深いことです。

`cons` *object1 object2* [Function]
この関数は新しいリスト構造を構築するための、もっとも基本的な関数である。この関数は *object1*を CAR、*object2*を CDR とする新しいコンスセルを作成して、それから新しいコンス

セルをリターンする。引数 *object1*と *object2*には任意の Lisp オブジェクトを指定できるが、ほとんどの場合 *object2*はリストである。

```
(cons 1 '(2))
     ⇒ (1 2)
(cons 1 '())
     ⇒ (1)
(cons 1 2)
     ⇒ (1 . 2)
```

リストの先頭に 1 つの要素を追加するために、`cons`がよく使用される。これをリストに要素をコンスすると言います。[2] たとえば:

```
(setq list (cons newelt list))
```

この例で使用されている`list`という名前の変数と、以下で説明する`list`という名前の関数は競合しないことに注意されたい。すべてのシンボルが、変数ト関数の両方の役割を果たすことができる。

`list` **&rest** *objects* [Function]

この関数は *objects*を要素とするリストを作成する。結果となるリストは常に`nil`終端される。*objects*を指定しないと空リストがリターンされる。

```
(list 1 2 3 4 5)
     ⇒ (1 2 3 4 5)
(list 1 2 '(3 4 5) 'foo)
     ⇒ (1 2 (3 4 5) foo)
(list)
     ⇒ nil
```

`make-list` *length object* [Function]

この関数は各要素が *object*であるような、*length*個の要素からなるリストを作成する。`make-list`と`make-string`(Section 4.3 [Creating Strings], page 52 を参照) を比較してみよ。

```
(make-list 3 'pigs)
     ⇒ (pigs pigs pigs)
(make-list 0 'pigs)
     ⇒ nil
(setq l (make-list 3 '(a b)))
     ⇒ ((a b) (a b) (a b))
(eq (car l) (cadr l))
     ⇒ t
```

`append` **&rest** *sequences* [Function]

この関数は *sequences*のすべての要素を含むリストを return します。*sequences*にはリスト、ベクター、ブールベクター、文字列も指定できるが、通常は最後にリストを指定すること。最

[2] リストの最後に要素を追加するための、これと完全に同等な方法はありません。*listname*をコピーすることにより新しいリストを作成してから、*newelt*をそのリストの最後に追加する (`append` *listname* (`list` *newelt*))を使用することができます。すべての CDR を辿って終端の`nil`を置き換える、(`nconc` *listname* (`list` *newelt*))を使用することもできます。コピーも変更も行なわずにリストの先頭に要素を追加する`cons`と比較してみてください。

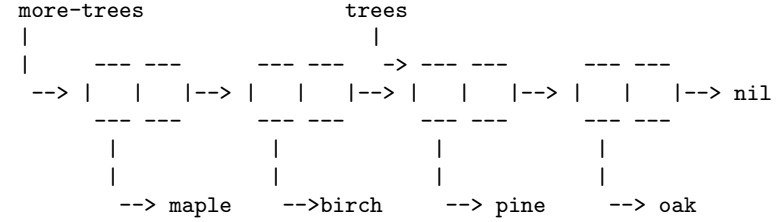
後の引数を除くすべての引数はコピーされるので、変更される引数はない (コピーを行なわずにリストを結合する方法については Section 5.6.3 [Rearrangement], page 84 の nconcを参照のこと)。

より一般的には appendにたいする最後の引数は任意の Lisp オブジェクトを指定できる。最後の引数のコピーや変換は行わない。最後の引数は新しいリストの最後のコンスセルの CDR となる。最後の引数もリストならば、このリストの要素は実質的には結果リストの要素になる。最後の要素がリストでなければ、最後の CDR が (正リストで要求される)nilではないので結果はドットリストになる (Section 5.1 [Cons Cells], page 71 を参照)。

以下は appendを使用した例です:

```
(setq trees '(pine oak))
     ⇒ (pine oak)
(setq more-trees (append '(maple birch) trees))
     ⇒ (maple birch pine oak)

trees
     ⇒ (pine oak)
more-trees
     ⇒ (maple birch pine oak)
(eq trees (cdr (cdr more-trees)))
     ⇒ t
```

appendがどのように機能するか、ボックスダイアグラムで確認できます。変数 treesはリスト (pine oak)にセットされ、それから変数 more-treesにリスト (maple birch pine oak)がセットされます。しかし変数 treesは継続して元のリストを参照します:



空のシーケンスは appendによりリターンされる値に寄与しません。この結果、最後の引数に nilを指定すると、それより前の引数のコピーを強制することになります。

```
trees
     ⇒ (pine oak)
(setq wood (append trees nil))
     ⇒ (pine oak)
wood
     ⇒ (pine oak)
(eq wood trees)
     ⇒ nil
```

関数 copy-sequenceが導入される以前は, これがリストをコピーする通常の方法でした。Chapter 6 [Sequences Arrays Vectors], page 95 を参照してください。

以下は appendの引数としてベクターと文字列を使用する例です:

```
(append [a b] "cd" nil)
     ⇒ (a b 99 100)
```

`apply` (Section 13.5 [Calling Functions], page 217 を参照) の助けを借りることにより、リストのリストの中のすべてのリストを append できます。

```
(apply 'append '((a b c) nil (x y z) nil))
     ⇒ (a b c x y z)
```

*sequences*が与えられなければ`nil`がリターンされます:

```
(append)
     ⇒ nil
```

以下は最後の引数がリストでない場合の例です:

```
(append '(x y) 'z)
     ⇒ (x y . z)
(append '(x y) [z])
     ⇒ (x y . [z])
```

2 番目の例は最後の引数はリストではないシーケンスの場合で、このシーケンスの要素は、結果リストの要素にはなりません。かわりに最後の引数がリストでないときと同様、シーケンスが最後の CDR になります。

`copy-tree` *tree* **&optional** *vecp* [Function]

この関数はツリー *tree*のコピーをリターンする。*tree*がコンスセルなら同じ CAR と CDR をもつ新しいコンスセルを作成してから、同じ方法によって CAR と CDR を再帰的にコピーする。

*tree*がコンスセル以外の場合、通常は`copy-tree`は単に *tree*をリターンする。しかし *vecp* が非`nil`なら、この関数はベクターでもコピーします (そしてベクターの要素を再帰的に処理する)。

`flatten-tree` *tree* [Function]

この関数は *tree*を “平坦化” したコピー (*tree*をルートとするコンスセルのツリーのすべての非`nil`な終端 node と leave) をリターンする。リターンされたリストの leave の順序は *tree*での順序と同じ。

```
(flatten-tree '(1 (2 . 3) nil (4 5 (6)) 7))
    ⇒(1 2 3 4 5 6 7)
```

`number-sequence` *from* **&optional** *to separation* [Function]

この関数は *from*から *separation*づつインクリメントして、*to*の直前で終わる数字のリストをリターンする。*separation*には正か負の数を指定でき、デフォルトは 1。*to*が`nil`、または数値的に *from*と等しければ、値は 1 要素のリスト`(`*from*`)`になる。*separation*が正で *to*が *from* より小さい、または *separation*が負で *to*が *from*より大きければ、これらの引数は空のシーケンスを指示することになるので、値は`nil`になる。

*separation*が 0 で、*to*が`nil`でもなく、数値的に *from*とも等しくまければ、これらの引数は無限シーケンスを指示することになるので、エラーがシグナルされる。

引数はすべて数字である。浮動少数点数の計算は正確ではないので、浮動少数点数の引数には注意する必要がある。たとえばマシンへの依存により、`(number-sequence 0.4 0.8 0.2)` が 3 要素のリストをリターンして、`(number-sequence 0.4 0.6 0.2)`が 1 要素のリスト `(0.4)`をリターン n することがよく起こる。リストの *n*番目の要素は、厳密に `(+ `*from*` (* n `*separation*`))`という式により計算される。リストに確実に *to*が含まれるようにするために、この式に適切な型の *to*を渡すことができる。別の方法として *to*を少しだけ大きな値 (*separation* が負なら少しだけ小さな値) に置き換えることもできる。

例をいくつか示す:

```
(number-sequence 4 9)
     ⇒ (4 5 6 7 8 9)
(number-sequence 9 4 -1)
     ⇒ (9 8 7 6 5 4)
(number-sequence 9 4 -2)
     ⇒ (9 7 5)
(number-sequence 8)
     ⇒ (8)
(number-sequence 8 5)
     ⇒ nil
(number-sequence 5 8 -1)
     ⇒ nil
(number-sequence 1.5 6 2)
     ⇒ (1.5 3.5 5.5)
```

5.5 リスト変数の変更

以下の関数と 1 つのマクロは、変数に格納されたリストを変更する便利な方法を提供します。

`push` *element listname* [Macro]

このマクロは CAR が *element* で、CDR が *listname* のリストであるような新しいリストを作成して、そのリストを *listname* に保存する。*listname* がリストに名前をつけるクォートされていないシンボルのときは単純で、この場合マクロは `(setq` *`listname`* `(cons` *`element listname`*`))` と等価になる。

```
(setq l '(a b))
     ⇒ (a b)
(push 'c l)
     ⇒ (c a b)
l
     ⇒ (c a b)
```

より一般的なのは `listname` が汎変数の場合である。この場合、このマクロは `(setf` *`listname`* `(cons` *`element listname`*`))` と等価になる。Section 12.17 [Generalized Variables], page 205 を参照のこと。

リストから 1 番目の要素を取り出す `pop` マクロについては、Section 5.3 [List Elements], page 73 を参照されたい。

以下の 2 つの関数は、変数の値であるリストを変更します。

`add-to-list` *symbol element* **&optional** *append compare-fn* [Function]

この関数は *element* が *symbol* の値のメンバーでなければ、*symbol* に *element* をコンスすることにより、変数 *symbol* をセットする。この関数はリストが更新されているか否かに関わらず、結果のリストをリターンする。*symbol* の値は呼び出し前にすでにリストであることが望ましい。*element* がリストの既存メンバーか比較するために、`add-to-list` は *compare-fn* を使用する。*compare-fn* が `nil` なら `equal` を使用する。

element が追加される場合は、通常は *symbol* の前に追加されるが、オプションの引数 *append* が非 `nil` なら最後に追加される。

引数 *symbol*は暗黙にクォートされない。`setq`とは異なり `add-to-list`は `set`のような通常の関数である。クォートしたい場合には自分で引数をクォートすること。

*symbol*がレキシカル変数を参照する際にはこの関数を使用しないこと。

以下に `add-to-list`を使用する方法をシナリオで示します:

```
(setq foo '(a b))
     ⇒ (a b)

(add-to-list 'foo 'c)     ;; cを追加
     ⇒ (c a b)

(add-to-list 'foo 'b)     ;; 効果なし
     ⇒ (c a b)

foo                       ;; fooが変更された
     ⇒ (c a b)
```

以下は `(add-to-list '`*var value*`)`と等価な式です:

```
(if (member value var)
    var
  (setq var (cons value var)))
```

`add-to-ordered-list` *symbol element* **&optional** *order* [Function]

この関数は古い値の *order* (リストであること) で指定された位置に、*element*を挿入して変数 *symbol*をセットする。*element*がすでにこのリストのメンバなら、リスト内の要素の位置は *order*にしたがって調整される。メンバーか否かは `eq`を使用してテストされる。この関数は更新されているかどうかに関わらず、結果のリストをリターンする。

*order*は通常は数字 (整数か浮動小数点数) で、リストの要素はその数字の昇順で並べられる。

*order*は省略または `nil`を指定できる。これによりリストに *element*がすでに存在するなら、*element*の数字順序は変更されない。それ以外なら *element*は数字順序をもたない。リストの数字順序をもたない要素はリストの最後に配置され、特別な順序はつかない。

*order*に他の値を指定すると、*element*がすでに数字順序をもつときは数字順序が削除される。それ以外はなら `nil`と同じ。

引数 *symbol*は暗黙にクォートされない。`add-to-ordered-list`は `setq`などとは異なり、`set`のような通常の関数である。必要なら引数を自分でクォートすること。

順序の情報は *symbol*の `list-order`プロパティーにハッシュテーブルで保存される。*symbol*はレキシカル変数を参照できない。

以下に `add-to-ordered-list`を使用する方法をシナリオで示します:

```
(setq foo '())
     ⇒ nil

(add-to-ordered-list 'foo 'a 1)     ;; aを追加
     ⇒ (a)

(add-to-ordered-list 'foo 'c 3)     ;; cを追加
     ⇒ (a c)
```

```
(add-to-ordered-list 'foo 'b 2)     ;; bを追加
     ⇒ (a b c)

(add-to-ordered-list 'foo 'b 4)     ;; bを移動
     ⇒ (a c b)

(add-to-ordered-list 'foo 'd)       ;; dを後に追加
     ⇒ (a c b d)

(add-to-ordered-list 'foo 'e)       ;; eを追加
     ⇒ (a c b e d)

foo                          ;; fooが変更された
     ⇒ (a c b e d)
```

5.6 既存のリスト構造の変更

プリミティブ `setcar`と `setcdr`でコンスセルの CAR および CDR のコンテンツを変更できます。これらは既存のリスト構造を変更するので破壊的な操作です。破壊的操作は mutable(変更可能) なリスト、すなわち `cons`、`list`、または類似の操作により構築される必要があります。クォートにより作成されたリストはプログラムの一部であり、破壊的な操作により変更するべきではありません。Section 2.9 [Mutability], page 35 を参照してください。

> **Common Lisp** に関する注意**:** Common Lisp はリスト構造の変更に `rplaca`と `rplacd` を使用する。これらは `setcar`や `setcdr`と同じ方法でリスト構造を変更するが、`setcar` と `setcdr`は新しい CAR や CDR をリターンするのにたいして、Common Lisp の関数はコンスセルをリターンする。

5.6.1 `setcar`によるリスト要素の変更

コンスセルの CAR の変更は `setcar`で行ないます。リストにたいして使用すると `setcar`はリストの 1 つの要素を別の要素に置き換えます。

`setcar` *cons object* [Function]

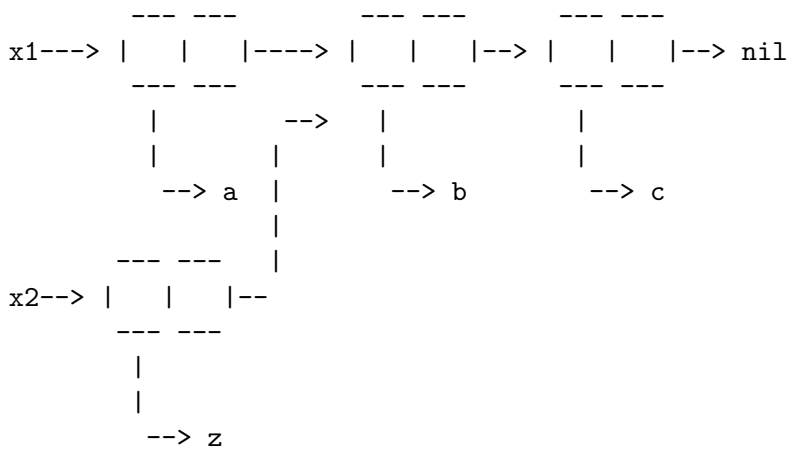
この関数は以前の CAR を置き換えて、*cons*の新しい CAR に *object*を格納する。言い換えると、この関数は *cons*の CAR スロットを *object*を参照するように変更する。この関数は値 *object* をリターンする。たとえば:

```
(setq x (list 1 2))
     ⇒ (1 2)
(setcar x 4)
     ⇒ 4
x
     ⇒ (4 2)
```

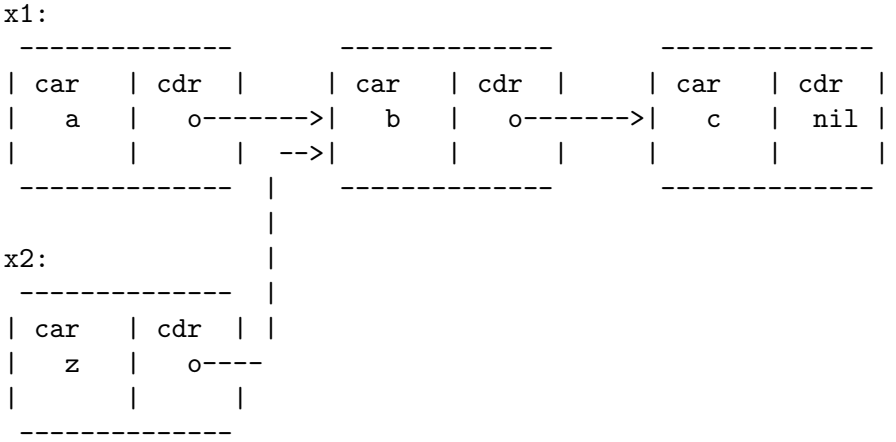
コンスセルが複数のリストを共有する構造の一部なら、コンスに新しい CAR を格納することにより、これら共有されたリストの各 1 つの要素を変更します。以下は例です:

```
;; 部分的に共有された 2 つのリストを作成
(setq x1 (list 'a 'b 'c))
     ⇒ (a b c)
(setq x2 (cons 'z (cdr x1)))
     ⇒ (z b c)

;; 共有されたリンクの CAR を置き換え
(setcar (cdr x1) 'foo)
     ⇒ foo
x1                              ; 両方のリストが変更された
     ⇒ (a foo c)
x2
     ⇒ (z foo c)

;; 共有されていないリンクの CAR を置き換え
(setcar x1 'baz)
     ⇒ baz
x1                              ; 1 つのリストだけが変更された
     ⇒ (baz foo c)
x2
     ⇒ (z foo c)
```

なぜ bを置き換えると両方が変更されるのかを説明するために、変数 x1と x2の 2 つのリストによる共有構造を視覚化してみましょう:



同じ関係を別のボックス図で示すと、以下のようになります:



5.6.2 リストの CDR の変更

CDR を変更するもっとも低レベルのプリミティブ関数は `setcdr`です:

`setcdr` *cons object* [Function]

この関数は前の CDR を置き換えて、*cons*の新しい CDR に *object*を格納する。言い換えると、この関数は *cons*の CDR が *object*を参照するように変更する。この関数は値 *object*をリターンする。

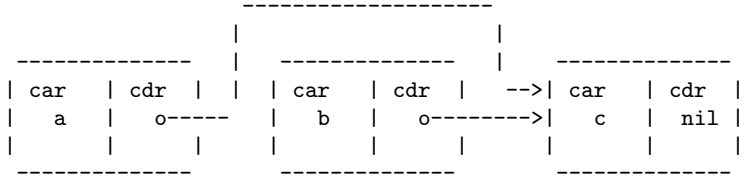
以下はリストの CDR を、他のリストに置き換える例です。1 番目の要素以外のすべての要素は、別のシーケンスまたは要素のために取り除かれます。1 番目の要素はリストの CAR なので変更されず、CDR を通じて到達することもできないからです。

```
(setq x (list 1 2 3))
     ⇒ (1 2 3)
(setcdr x '(4))
     ⇒ (4)
x
     ⇒ (1 4)
```

リスト内のコンスセルの CDR を変更することにより、リストの途中から要素を削除できます。たとえば以下では、1 番目のコンスセルの CDR を変更することにより、2 番目の要素 `b`をリスト `(a b c)`から削除します。

```
(setq x1 (list 'a 'b 'c))
     ⇒ (a b c)
(setcdr x1 (cdr (cdr x1)))
     ⇒ (c)
x1
     ⇒ (a c)
```

以下に結果をボックス表記で示します:

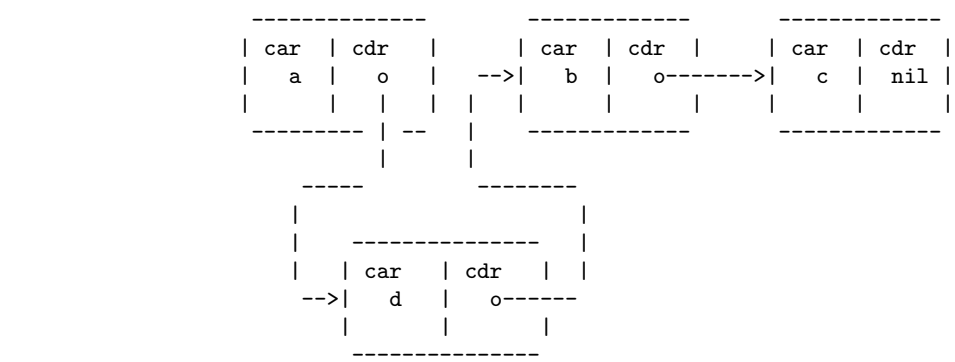


以前は要素 bを保持していた 2 番目のコンスセルは依然として存在し、その CAR も bのままですが、すでにこのリストの一部を形成していません。

CDR を変更して新しい要素を挿入するのも同じくらい簡単です:

```
(setq x1 (list 'a 'b 'c))
     ⇒ (a b c)
(setcdr x1 (cons 'd (cdr x1)))
     ⇒ (d b c)
x1
     ⇒ (a d b c)
```

以下に結果をボックス表記で示します:



5.6.3 リストを再配置する関数

以下ではリストの構成要素であるコンスセルの CDR を変更することにより、リストを破壊的に再配置する関数をいくつか示します。これらの関数が破壊的だという理由は、これらの関数が引数として渡された元のリストを処理してリターン値となる新しいリストを形成するために、リストのコンスセルを再リンクするからです。

以降のセクションで説明する関数 delqは、破壊的にリストを操作する別の例です。

nconc &rest *lists* [Function]

この関数は *lists*の要素すべてを含むリストをリターンする。append (Section 5.4 [Building Lists], page 75 を参照) とは異なり、*lists*はコピーされない。かわりに *lists*の各リストの最後の CDR が次のリストを参照するように変更される。*lists*の最後のリストは変更されない。たとえば:

```
(setq x (list 1 2 3))
     ⇒ (1 2 3)
(nconc x '(4 5))
     ⇒ (1 2 3 4 5)
x
     ⇒ (1 2 3 4 5)
```

nconcの最後の引数は変更されないので、上記の例のように'(4 5)のような定数リストを使用するのが合理的である。また同じ理由により最後の引数がリストである必要はない。

```
(setq x (list 1 2 3))
     ⇒ (1 2 3)
(nconc x 'z)
     ⇒ (1 2 3 . z)
```

```
x
     ⇒ (1 2 3 . z)
```

しかし他の (最後を除くすべての) 引数は mutable リストでなければなければならない。

一般的な落とし穴としては、nconcにたいしてリスト定数を最後以外の引数として使用した場合である。これを行なった場合の結果としての挙動は未定義である。実行するごとにプログラムはリスト定数を変更する可能性がある! (必ず発生する保証はないが) 以下のようなことが起こり得る:

```
(defun add-foo (x)            ; この関数では foo
  (nconc '(foo) x))           ;   を引数の前に追加したい

(symbol-function 'add-foo)
     ⇒ (lambda (x) (nconc '(foo) x))

(setq xx (add-foo '(1 2)))    ; 動いているように見える
     ⇒ (foo 1 2)
(setq xy (add-foo '(3 4)))    ; 何が起きているのか?
     ⇒ (foo 1 2 3 4)
(eq xx xy)
     ⇒ t

(symbol-function 'add-foo)
     ⇒ (lambda (x) (nconc '(foo 1 2 3 4) x))
```

5.7 集合としてのリストの使用

リストは順序なしの数学的集合 — リスト内に要素があれば集合の要素の値としてリスト内の順序は無視される — を表すことができます。2 つの集合を結合 (union) するには、(重複する要素を気にしなければ)appendを使用します。equalである重複を取り除くには delete-dupsを使用します。集合にたいする他の有用な関数には memqや delqや、それらの equalバージョンである memberと deleteが含まれます。

> **Common Lisp** に関する注意**:** 集合を処理するために Common Lisp には関数 union (要素の重複がない) と intersectionがある。Emacs Lisp では cl-libがこれらの変種を提供する。Section “Lists as Sets” in *Common Lisp Extensions* を参照のこと。

memq *object list* [Function]

この関数は *object*が *list*のメンバーかどうかをテストする。メンバーなら memqは、*object*で最初に見つかった要素から開始されるリストをリターンする。メンバーでなければ nilをリターンする。memqの文字 ‘q’は、この関数が *object*とリスト内の要素の比較に eqを使用することを示す。たとえば:

```
(memq 'b '(a b c b a))
     ⇒ (b c b a)
(memq '(2) '((1) (2)))    ; 2 つの (2)が eqである必要はない
     ⇒ 未定義; nilか ((2))かも
```

delq *object list* [Function]

この関数は *list*から *object*と eqであるような、すべての要素を破壊的に取り除いて結果のリストをリターンする。delqの文字 ‘q’は、この関数が *object*とリスト内の要素の比較に eqを使用することを示す (memqや remqと同様)。

delqを呼び出すときは、通常は元のリストを保持していた変数にリターン値を割り当てて使用する必要がある (理由は以下参照)。

delq関数がリストの先頭にある要素を削除する場合は、単にリストを読み進めてこの要素の後から開始される部分リストをリターンします。つまり:

```
(delq 'a '(a b c)) ≡ (cdr '(a b c))
```

リストの途中にある要素を削除するときは、必要な CDR (Section 5.6.2 [Setcdr], page 83 を参照) を変更することで削除を行います。

```
(setq sample-list (list 'a 'b 'c '(4)))
     ⇒ (a b c (4))
(delq 'a sample-list)
     ⇒ (b c (4))
sample-list
     ⇒ (a b c (4))
(delq 'c sample-list)
     ⇒ (a b (4))
sample-list
     ⇒ (a b (4))
```

(delq 'a sample-list)は何も取り除きませんが (単に短いリストをリターンする)、(delq 'c sample-list)は 3 番目の要素を取り除いて sample-listを変更することに注意してください。引数 *list*を保持するように形成された変数が、実行後にもっと少ない要素になるとか、元のリストを保持すると仮定しないでください! かわりに delqの結果を保存して、それを使用してください。元のリストを保持していた変数に結果を書き戻すことはよく行なわれます。

```
(setq flowers (delq 'rose flowers))
```

以下の例では、delqが比較しようとしている (list 4)と sample-list内の (4)は、equalですが eqではありません:

```
(delq (list 4) sample-list)
     ⇒ (a c (4))
```

与えられた値と equalな要素を削除したい場合には、delete (以下参照) を使用してください。

`remq` *object list* [Function]

この関数は *object*と eqなすべての要素が除かれた、*list*のコピーをリターンする。remqの文字 'q'は、この関数が *object*とリスト内の要素の比較に eqを使用することを示す。

```
(setq sample-list (list 'a 'b 'c 'a 'b 'c))
     ⇒ (a b c a b c)
(remq 'a sample-list)
     ⇒ (b c b c)
sample-list
     ⇒ (a b c a b c)
```

`memql` *object list* [Function]

関数 memqlは eql(浮動少数点数の要素は値で比較される) を使用してメンバーと eqlを比較することにより、*object*が *list*のメンバーかどうかをテストする。*object*がメンバーなら、memql は *list*内で最初に見つかった要素から始まるリスト、それ以外なら nilをリターンする。

memqと比較してみよう:

```
(memql 1.2 '(1.1 1.2 1.3))  ; 1.2と 1.2は eql。
     ⇒ (1.2 1.3)
(memq 1.2 '(1.1 1.2 1.3))  ; 2 つの 1.2が eqである必要はない
     ⇒ 未定義; nilか (1.2 1.3)かもしれない
```

以下の 3 つの関数は memq、delq、remqと似ていますが、要素の比較に eqではなく equalを使用します。Section 2.8 [Equality Predicates], page 33 を参照してください。

member *object list* [Function]

関数 memberは、メンバーと *object*を equalを使用して比較して、*object*が *list*のメンバーかどうかをテストする。*object*がメンバーなら、memberは *list*で最初に見つかったところから開始されるリスト、それ以外なら nilをリターンする。

memqと比較してみよう:

```
(member '(2) '((1) (2)))  ; (2) and (2) are equal.
     ⇒ ((2))
(memq '(2) '((1) (2)))    ; 2 つの (2)が eqである必要はない
     ⇒ 未定義; nilか (2)かもしれない
;; 同じ内容の 2 つの文字列は equal
(member "foo" '("foo" "bar"))
     ⇒ ("foo" "bar")
```

delete *object sequence* [Function]

この関数は *sequence*から *object*と equalな要素を取り除いて、結果のシーケンスをリターンする。

*sequence*がリストなら、deleteが delqに対応するように、memberは memqに対応する。つまりこの関数は memberと同様、要素と *object*の比較に equalを使用する。マッチする要素が見つかったら、delqが行なうようにその要素を取り除く。delqと同様、通常は元のリストを保持していた変数にリターン値を割り当てて使用する。

sequenceがベクターか文字列なら、deleteは objectと equalなすべての要素を取り除いた sequenceのコピーをリターンする。

たとえば:

```
(setq l (list '(2) '(1) '(2)))
(delete '(2) l)
     ⇒ ((1))
l
     ⇒ ((2) (1))
;; lの変更に信頼性を要するときは
;; (setq l (delete '(2) l))と記述する。
(setq l (list '(2) '(1) '(2)))
(delete '(1) l)
     ⇒ ((2) (2))
l
     ⇒ ((2) (2))
;; このケースでは lのセットの有無に違い
;; はないが他のケースに倣ってセットするべき
(delete '(2) [(2) (1) (2)])
     ⇒ [(1)]
```

remove *object sequence* [Function]

この関数は deleteに対応する非破壊的な関数である。この関数は objectと equalな要素を取り除いた、sequence(リスト、ベクター、文字列) のコピーをリターンする。たとえば:

```
(remove '(2) '((2) (1) (2)))
     ⇒ ((1))
(remove '(2) [(2) (1) (2)])
     ⇒ [(1)]
```

Common Lisp に関する注意**:** GNU Emacs Lisp の関数 `member`、`delete`、`remove` は Common Lisp ではなく、Maclisp を継承する。Common Lisp では比較に `equal` を使用しない。

`member-ignore-case` *object list* [Function]
この関数は `member`と同様だが、*object*が文字列で case とテキスト表現の違いを無視する。文字の大文字と小文字は等しいものとして扱われ、比較に先立ちユニバイト文字列はマルチバイト文字列に変換される。

`delete-dups` *list* [Function]
この関数は *list*からすべての `equal`な重複を破壊的に取り除いて、結果を *list*に保管してそれをリターンする。*list*内の要素に `equal`な要素がいくつかあるなら、`delete-dups`は最初の要素を残す。

変数に格納されたリストへの要素の追加や、それを集合として使用する方法については、Section 5.5 [List Variables], page 79 の関数 `add-to-list`も参照してください。

5.8 連想リスト

連想配列 *(association list*、短くは *alist)* は、キーと値のマッピングを記録します。これは連想 *(associations)* と呼ばれるコンスセルのリストです。各コンスセルにおいて CAR はキー *(key)* で、CDR は連想値 *(associated value)* となります。[3]

以下は alist の例です。キー `pine`は値 `cones`、キー `oak`は `acorns`、キー `maple`は `seeds`に関連付けられます。

```
((pine . cones)
 (oak . acorns)
 (maple . seeds))
```

alist 内の値とキーには、任意の Lisp オブジェクトを指定できます。たとえば以下の alist0 では、シンボル `a`は数字 `1`、文字列`"b"`はリスト `(2 3)`(alist 要素の CDR) に関連付けられます。

```
((a . 1) ("b" 2 3))
```

要素の CDR の CAR に連想値を格納するように alist デザインするほうがよい場合があります。以下はそのような alist です。

```
((rose red) (lily white) (buttercup yellow))
```

この例では、`red`が `rose`に関連付けられる値だと考えます。この種の alist の利点は、CDR の CDR の中に他の関連する情報— 他のアイテムのリストでさえも — を格納することができることです。不利な点は、与えられた値を含む要素を見つけるために `rassq`(以下参照) を使用できないことです。これらを検討することが重要でない場合には、すべての与えられた alist にたいして一貫している限り、選択は好みの問題といえます。

[3] ここでの “キー (key)” の使い方は、用語 “キーシーケンス (key sequence)” とは関係ありません。キーはテーブルにあるアイテムを探すために使用される値という意味です。この場合、テーブルは alist であり alist はアイテムに関連付けられます。

上記で示したのと同じ alist は、要素の CDR に連想値をもつと考えることができます。この場合、`rose`に関連付けられる値はリスト `(red)`になるでしょう。

連想リストは新しい連想値を簡単にリストの先頭に追加できるので、スタックに保持したいような情報を記録するのによく使用されます。連想リストから与えられたキーにたいして連想値を検索する場合、それが複数ある場合は、最初に見つかったものが return されます。

Emacs Lisp では、連想リストがコンスセルでなくても、それはエラーではありません。alist 検索関数は、単にそのような要素を無視します。多くの他のバージョンの Lisp では、このような場合はエラーをシグナルします。

いくつかの観点において、プロパティーリストは連想リストと似ていることに注意してください。それぞれのキーが一度だけ出現するような場合、プロパティーリストは連想リストと同様に振る舞います。プロパティーリストと連想リストの比較については、Section 5.9 [Property Lists], page 92 を参照してください。

`assoc` *key alist* **&optional** *testfn* [Function]

この関数は alist 要素にたいして非`nil`なら *testfn*、それ以外ならを使用して、*alist*内から *key* をもつ最初の連想をリターンする。CAR が *key* と `equal`であるような連想値が *alist*になければ、この関数は`nil`をリターンする。たとえば:

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
     ⇒ ((pine . cones) (oak . acorns) (maple . seeds))
(assoc 'oak trees)
     ⇒ (oak . acorns)
(cdr (assoc 'oak trees))
     ⇒ acorns
(assoc 'birch trees)
     ⇒ nil
```

以下はキーと値がシンボルでない場合の例である:

```
(setq needles-per-cluster
      '((2 "Austrian Pine" "Red Pine")
        (3 "Pitch Pine")
        (5 "White Pine")))

(cdr (assoc 3 needles-per-cluster))
     ⇒ ("Pitch Pine")
(cdr (assoc 2 needles-per-cluster))
     ⇒ ("Austrian Pine" "Red Pine")
```

関数`assoc-string`は`assoc`と似ていますが、文字列間の特定の違いを無視する点が異なります。Section 4.5 [Text Comparison], page 56 を参照してください。

`rassoc` *value alist* [Function]

この関数は *alist*の中から値 *value*をもつ最初の連想をリターンする。CDR が *value*と `equal`であるような連想値が *alist*になければ、この関数は`nil`をリターンする。

`rassoc`は`assoc`と似てイルが、CAR ではなく *alist*の連想値の CDR を比較する。この関数は与えられた値に対応するキーを探す、`assoc`の逆バージョンと考えることができよう。

`assq` *key alist* [Function]

この関数は *alist*から *key*をもつ最初の連想値をリターンする点は`assoc`と同様だが、比較に`eq`を使用する点が異なる。CAR が *key*と `eq`であるような連想値が *alist*内に存在しなければ`assq`は`nil`をリターンする。`eq`は`equal`より高速であり、ほとんどの alist はキーにシンボ

ルを使用するので、この関数は assocより多用される。Section 2.8 [Equality Predicates], page 33 を参照のこと。

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
     ⇒ ((pine . cones) (oak . acorns) (maple . seeds))
(assq 'pine trees)
     ⇒ (pine . cones)
```

逆にキーがシンボルではない alist では、通常は assqは有用ではない:

```
(setq leaves
      '(("simple leaves" . oak)
        ("compound leaves" . horsechestnut)))

(assq "simple leaves" leaves)
     ⇒ 未定義; nilか ("simple leaves" . oak)かもしれない
(assoc "simple leaves" leaves)
     ⇒ ("simple leaves" . oak)
```

`alist-get` *key alist* **&optional** *default remove testfn* [Function]

この関数は assqと似ている。これは *alist*の要素の *key*を比較して最初の連想 (*key* . *value*) を見つける。連想が見つからなければ、関数は *default*をリターンする。*alist*にたいする *key* の比較には *testfn*で指定された関数を使用する (デフォルトは eq)。

これは setfでの値の変更に使用できる汎変数 (Section 12.17 [Generalized Variables], page 205 を参照)。値の値へのセットにこれを使用する際にオプション引数 *remove*が nilの場合は、新たな値が *default*と eqlなら *alist*から *key*の連想を削除することを意味する。

`rassq` *value alist* [Function]

この関数は、*alist*内から値 *value*をもつ最初の連想値をリターンする。*alist*内に CDR が *value* と eqであるような連想値が存在しないなら nilをリターンする。

rassqは assqと似ていますが、CAR ではなく *alist*の各連想の CDR を比較します。この関数を、与えられた値に対応するキーを探す assqの逆バージョンと考えることができます。

たとえば:

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))

(rassq 'acorns trees)
     ⇒ (oak . acorns)
(rassq 'spores trees)
     ⇒ nil
```

rassqは要素の CDR の CAR に保管された値の検索はできません:

```
(setq colors '((rose red) (lily white) (buttercup yellow)))

(rassq 'white colors)
     ⇒ nil
```

この場合、連想 (lily white)の CDR は whiteではなくリスト (white)です。これは連想をドットペア表記で記述すると明確になります:

```
(lily white) ≡ (lily . (white))
```

`assoc-default` *key alist* **&optional** *test default* [Function]

この関数は、*key*にたいするマッチを *alist*から検索する。*alist*の各要素にたいして、この関数は *key*と要素 (アトムの場合)、または要素の CAR(コンスの場合) を比較する。比較は *test*に 2 つの引数— 要素 (か要素の CAR) と *key* — を与えて呼び出すことにより行なわれる。引数はこの順番で渡されるので、正規表現 (Section 34.4 [Regexp Search], page 894 を参照) を含

む alist では、`string-match`を使用することにより有益な結果を得ることができる。*test*が省略または`nil`なら比較に `equal`が使用される。

alist の要素がこの条件により *key*とマッチすると、`assoc-default`はその要素の値をリターンする。要素がコンスなら値は要素の CDR、それ以外ならリターン値は *default*となる。

*key*にマッチする要素が alist に存在しないければ、`assoc-default`は`nil`をリターンする。

`copy-alist` *alist* [Function]

この関数は深さのレベルが 2 の *alist*のコピーをリターンする。この関数は各連想の新しいコピーを作成するので、元の alist を変更せずに新しい alist を変更できる。

```
(setq needles-per-cluster
      '((2 . ("Austrian Pine" "Red Pine"))
        (3 . ("Pitch Pine"))
        (5 . ("White Pine"))))
⇒
((2 "Austrian Pine" "Red Pine")
 (3 "Pitch Pine")
 (5 "White Pine"))

(setq copy (copy-alist needles-per-cluster))
⇒
((2 "Austrian Pine" "Red Pine")
 (3 "Pitch Pine")
 (5 "White Pine"))

(eq needles-per-cluster copy)
     ⇒ nil
(equal needles-per-cluster copy)
     ⇒ t
(eq (car needles-per-cluster) (car copy))
     ⇒ nil
(cdr (car (cdr needles-per-cluster)))
     ⇒ ("Pitch Pine")
(eq (cdr (car (cdr needles-per-cluster)))
    (cdr (car (cdr copy))))
     ⇒ t
```

以下の例は、どのようにして `copy-alist`が他に影響を与えずにコピーの連想を変更可能なのかを示す:

```
(setcdr (assq 3 copy) '("Martian Vacuum Pine"))
(cdr (assq 3 needles-per-cluster))
     ⇒ ("Pitch Pine")
```

`assq-delete-all` *key alist* [Function]

この関数は、`delq`を使用してマッチする要素を 1 つずつ削除するときのように、CAR が *key*と `eq`であるようなすべての要素を *alist*から削除する。この関数は短くなった alist をリターンし、*alist*の元のリスト構造を変更することもよくある。正しい結果を得るために、*alist*に保存された値ではなく `assq-delete-all`のリターン値を使用すること。

```
(setq alist (list '(foo 1) '(bar 2) '(foo 3) '(lose 4)))
     ⇒ ((foo 1) (bar 2) (foo 3) (lose 4))
(assq-delete-all 'foo alist)
     ⇒ ((bar 2) (lose 4))
alist
     ⇒ ((foo 1) (bar 2) (lose 4))
```

`assoc-delete-all` *key alist* **&optional** *test* [Function]
この関数は `assq-delete-all`と同様だが、オプション引数 *test* (*alist*内のキーを比較するための述語関数) を受け取る点が異なる。*test*が省略か `nil`ならデフォルトは `equal`。この関数は `assq-delete-all`のように、多くの場合は *alist*の元のリスト構造を変更する。

`rassq-delete-all` *value alist* [Function]
この関数は、*alist*から CDR が *value*と `eq`であるようなすべての要素を削除する。この関数は短くなったリストをリターンし、*alist*の元のリスト構造を変更することもよくある。`rassq-delete-all`は `assq-delete-all`と似ているが、CAR ではなく *alist*の各連想の CDR を比較する。

`let-alist` *alist body* [Macro]
連想リスト *alist*のキーとして使用される先頭にドットを付したシンボルそれぞれにたいしてバインディングを作成する。これは同じ連想リスト内の複数のアイテムにアクセスする際に有用かもしれない。理解するためにもっともよいのは以下のシンプルな例だろう:

```
(setq colors '((rose . red) (lily . white) (buttercup . yellow)))
(let-alist colors
  (if (eq .rose 'red)
      .lily))
=> white
```

*body*をコンパイル時に検査して、*body*内に出現する先頭文字として '.'を付したシンボルだけがバインドされる。キーの検索は `assq`、この `assq`のリターン値の `cdr`がそのバインディングにたいする値として割り当てられる。

ネストされた連想リストをサポートする:

```
(setq colors '((rose . red) (lily (belladonna . yellow) (brindisi . pink))))
(let-alist colors
  (if (eq .rose 'red)
      .lily.belladonna))
=> yellow
```

互いに内部に `let-alist`をネストすることが可能だが、内側の `let-alist`は外側の `let-alist`がバインドする変数にはアクセスできない。

5.9 プロパティリスト

プロパティーリスト *(property list*、短くは *plist)* は、ペアになった要素のリストです。各ペアはプロパティー名 (通常はシンボル) とプロパティー値を対応づけます。以下はプロパティーリストの例です:

```
(pine cones numbers (1 2 3) color "blue")
```

このプロパティーリストは `pine`を `cones`、`numbers`を `(1 2 3)`、`color`を `"blue"`に関連づけます。プロパティー名とプロパティー値には任意の Lisp オブジェクトを指定できますが、通常プロパティー名は (この例のように) シンボルです。

いくつかのコンテキストでプロパティーリストが使用されます。たとえば関数 `put-text-property`はプロパティーリストを引数にとり、文字列やバッファー内のテキストにたいして、テキストプロパティーとテキストに適用するプロパティー値を指定します。Section 32.19 [Text Properties], page 813 を参照してください。

プロパティーリストが頻繁に使用される他の例は、シンボルプロパティーの保管です。すべてのシンボルはシンボルに関する様々な情報を記録するために、プロパティーのリストを処理します。これらのプロパティーはプロパティーリストの形式で保管されます。Section 9.4 [Symbol Properties], page 129 を参照してください。

5.9.1 プロパティリストと連想リスト

連想リスト (Section 5.8 [Association Lists], page 88 を参照) は、プロパティーリストとよく似ています。連想リストとは対照的にプロパティー名は一意でなければならないので、プロパティーリスト内でペアの順序に意味はありません。

様々な Lisp 関数や変数に情報を付加するためには、連想リストよりプロパティーリストの方が適しています。プログラムでこのような情報すべてを 1 つの連想リストに保持する場合は、特定の Lisp 関数や変数にたいする連想をチェックする度にリスト全体を検索する必要が生じ、それにより遅くなる可能性があります。対照的に関数名や変数自体のプロパティーリストに同じ情報を保持すれば、検索ごとにそのプロパティーリストの長さだけを検索するようになり、通常はこちらの方が短時間で済みます。変数のドキュメントが `variable-documentation`という名前のプロパティーに記録されているのはこれが理由です。同様にバイトコンパイラーも、特別に扱う必要がある関数を記録するためにプロパティーを使用します。

とはいえ連想リストにも独自の利点があります。アプリケーションに依存しますが、プロパティーを更新するより連想リストの先頭に連想を追加する方が高速でしょう。シンボルにたいするすべてのプロパティーは同じプロパティーリストに保管されるので、プロパティー名を異なる用途のために使用すると衝突の可能性があります (この理由により、そのプログラムで通常の変数や関数の名前につけるプレフィクスをプロパティー名の先頭につけて、一意と思われるプロパティー名を選ぶのはよいアイデアだと言える)。連想リストは、連想をリストの先頭に push して、その後にある連想は無視されるので、スタックと同様に使用できます。これはプロパティーリストでは不可能です。

5.9.2 プロパティリストと外部シンボル

以下の関数はプロパティーリストを操作するために使用されます。これらの関数はすべて、プロパティー名の比較に `eq`を使用します。

`plist-get` *plist property* [Function]

この関数はプロパティーリスト *plist*に保管された、プロパティー *property*の値をリターンする。この関数には不正な形式 (malformed) の *plist*引数を指定できる。*plist*で *property*が見つからないと、この関数は `nil`をリターンする。たとえば、

```
(plist-get '(foo 4) 'foo)
     ⇒ 4
(plist-get '(foo 4 bad) 'foo)
     ⇒ 4
(plist-get '(foo 4 bad) 'bad)
     ⇒ nil
(plist-get '(foo 4 bad) 'bar)
     ⇒ nil
```

`plist-put` *plist property value* [Function]

この関数はプロパティーリスト *plist*に、プロパティー *property*の値として *value*を保管する。この関数は *plist*を破壊的に変更するかもしれず、元のリスト構造を変更せずに新しいリストを構築することもある。この関数は変更されたプロパティーリストをリターンするので、*plist*を取得した場所に書き戻すことができる。たとえば、

```
(setq my-plist (list 'bar t 'foo 4))
     ⇒ (bar t foo 4)
(setq my-plist (plist-put my-plist 'foo 69))
     ⇒ (bar t foo 69)
```

```
(setq my-plist (plist-put my-plist 'quux '(a)))
     ⇒ (bar t foo 69 quux (a))
```

`lax-plist-get` *plist property* [Function]
`plist-get`と同様だがプロパティーの比較に `eq`ではなく `equal`を使用する。

`lax-plist-put` *plist property value* [Function]
`plist-put`と同様だがプロパティーの比較に `eq`ではなく `equal`を使用する。

`plist-member` *plist property* [Function]
この関数は与えられた *property*が *plist*に含まれるなら非 `nil`をリターンする。`plist-get`とは異なりこの関数は存在しないプロパティーと、値が `nil`のプロパティーを区別できる。実際にリターンされる値は、`car`が *property*で始まる *plist*の末尾部分である。

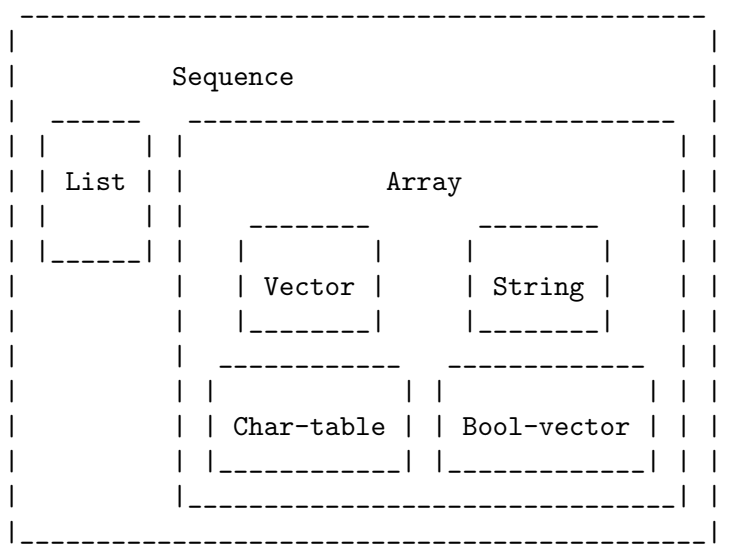
6 シーケンス、配列、ベクター

シーケンス *(sequence)* 型は 2 つの異なる Lisp 型 — リストと配列 — を結合した型です。言い換えると任意のリストはシーケンスであり任意の配列はシーケンスです。すべてのシーケンスがもつ共通な属性は、それぞれが順序づけされた要素のコレクションであることです。

配列 *(array)* はスロットがその要素であるような、固定長のオブジェクトです。すべての要素に一定時間でアクセスできます。配列の 4 つの型として文字列、ベクター、文字テーブル、ブールベクターがあります。

リストは要素のシーケンスですが、要素は単一の基本オブジェクトではありません。リストはコンスセルにより作られ、要素ごとに 1 つのセルをもちます。*n*番目の要素を探すには *n*個のコンスセルを走査する必要があるので、先頭から離れた要素ほどアクセスに時間を要します。しかしリストは要素の追加や削除が可能です。

以下の図はこれらの型の関連を表しています:



6.1 シーケンス

このセクションでは任意の種類のシーケンスを許す関数を説明します。

`sequencep` *object* [Function]

この関数は *object*がリスト、ベクター、文字列、ブールベクター、文字テーブルなら `t`、それ以外は `nil`をリターンする。以下の `seqp`も参照のこと。

`length` *sequence* [Function]

この関数は *sequence*内の要素の数をリターンする。*sequence*がシーケンス以外、またはドットリストなら `wrong-type-argument`エラーをシグナルする。引数が循環リストなら `circular-list`エラーをシグナルする。文字テーブルでは Emacs の最大文字コードより 1 大きい値が常にリターンされる。

関連する関数 `safe-length`については [Definition of safe-length], page 75 を参照のこと。

```
(length '(1 2 3))
    ⇒ 3
```

```
(length ())
    ⇒ 0
(length "foobar")
    ⇒ 6
(length [1 2 3])
    ⇒ 3
(length (make-bool-vector 5 nil))
    ⇒ 5
```

Section 33.1 [Text Representations], page 852 の `string-bytes`も参照されたい。

ディスプレー上での文字列の幅を計算する必要があるなら、文字数だけを数えて各文字のディスプレー幅を計算しない `length`ではなく、`string-width` (Section 39.10 [Size of Displayed Text], page 1007 を参照) を使用すること。

`elt` *sequence index* [Function]

この関数は *index*によりインデックスづけされた、*sequence*の要素をリターンする。*index*の値として妥当なのは、0 から *sequence*の長さより 1 小さい数までの範囲の整数。*sequence*がリストなら範囲外の値は `nth`と同じように振る舞う。[Definition of nth], page 74 を参照のこと。それ以外なら範囲外の値は `args-out-of-range`エラーを引き起こす。

```
(elt [1 2 3 4] 2)
     ⇒ 3
(elt '(1 2 3 4) 2)
     ⇒ 3
;; eltがどの文字を return するか明確にするために stringを使用
(string (elt "1234" 2))
     ⇒ "3"
(elt [1 2 3 4] 4)
     error  Args out of range: [1 2 3 4], 4
(elt [1 2 3 4] -1)
     error  Args out of range: [1 2 3 4], -1
```

この関数は `aref` (Section 6.3 [Array Functions], page 107 を参照) と `nth` ([Definition of nth], page 74 を参照) を一般化したものである。

`copy-sequence` *seqr* [Function]

この関数は *seqr* (シーケンスかレコードであること) のコピーをリターンする。コピーはオリジナルと同じオブジェクト型であり、同じ要素を同じ順序でもつ。しかし *seqr*が空なら長さが 0 の文字列やベクターと同じように関数がリターンする値はコピーではないかもしれないが、*seqr*と同じ型の空のオブジェクトである。

コピーに新しい要素を格納するのは元の *seqr*に影響を与えずその逆も真である。しかし新しいシーケンス内の要素はコピーではなく、元のシーケンスの要素と同一 (`eq`) になる。したがってコピーされたシーケンスを介して見つかった要素を変更するとオリジナルでも変更を見ることができる。

引数がテキストプロパティーをもつ文字列なら、コピー内のプロパティーリスト自身もコピーとなり、元のシーケンスのプロパティーリストと共有はされない。しかしプロパティーリストの実際の値は共有される。Section 32.19 [Text Properties], page 813 を参照のこと。

この関数はドットリストでは機能しない。循環リストのコピーは無限ループを起こすだろう。

シーケンスをコピーする別の方法については Section 5.4 [Building Lists], page 75 の `append`、Section 4.3 [Creating Strings], page 52 の `concat`、Section 6.5 [Vector Functions], page 109 の `vconcat` も参照されたい。

```
(setq bar (list 1 2))
     ⇒ (1 2)
(setq x (vector 'foo bar))
     ⇒ [foo (1 2)]
(setq y (copy-sequence x))
     ⇒ [foo (1 2)]

(eq x y)
     ⇒ nil
(equal x y)
     ⇒ t
(eq (elt x 1) (elt y 1))
     ⇒ t

;; 一方のシーケンスの要素を置き換え
(aset x 0 'quux)
x ⇒ [quux (1 2)]
y ⇒ [foo (1 2)]

;; 共有された要素の内部を変更
(setcar (aref x 1) 69)
x ⇒ [quux (69 2)]
y ⇒ [foo (69 2)]
```

`reverse` *sequence* [Function]

この関数は *sequence* の要素を反転した要素をもつ新たなシーケンスを作成する。元となる引数 *sequence* は変更されない。文字テーブルは反転できないことに注意。

```
(setq x '(1 2 3 4))
     ⇒ (1 2 3 4)
(reverse x)
     ⇒ (4 3 2 1)
x
     ⇒ (1 2 3 4)
(setq x [1 2 3 4])
     ⇒ [1 2 3 4]
(reverse x)
     ⇒ [4 3 2 1]
x
     ⇒ [1 2 3 4]
(setq x "xyzzy")
     ⇒ "xyzzy"
```

```
(reverse x)
     ⇒ "yzzyx"
x
     ⇒ "xyzzy"
```

`nreverse` *sequence* [Function]

この関数は *sequence*の要素を反転する。`reverse`とは異なり、元となる *sequence*は変更されるかもしれない。

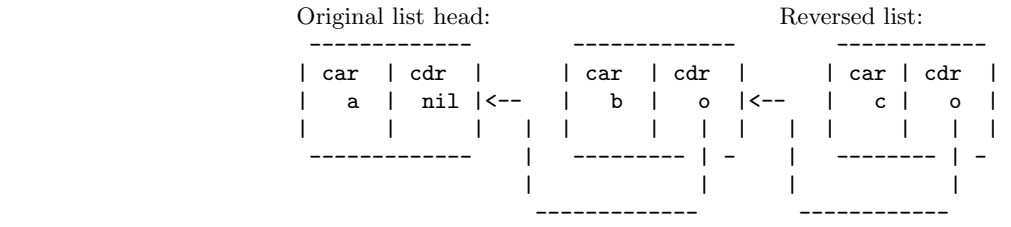
たとえば:

```
(setq x (list 'a 'b 'c))
     ⇒ (a b c)
x
     ⇒ (a b c)
(nreverse x)
     ⇒ (c b a)
;; 先頭にあったコンスセルが末尾となった
x
     ⇒ (a)
```

混乱しないように、通常は元となるリストを保持する同じ変数に、`nreverse`の結果を書き戻す:

```
(setq x (nreverse x))
```

お馴染の例 `(a b c)`の `nreverse`を以下に図示する:



setq が不要なのでベクターはより単純になる:

```
(setq x (copy-sequence [1 2 3 4]))
     ⇒ [1 2 3 4]
(nreverse x)
     ⇒ [4 3 2 1]
x
     ⇒ [4 3 2 1]
```

`reverse`とは異なり、この関数は文字列では機能しない。`aset`を使用して文字列データを変更できても、たとえ mutable であったとしても文字列は不変として扱うことを強く推奨する。Section 2.9 [Mutability], page 35 を参照のこと。

`sort` *sequence predicate* [Function]

この関数は *sequence*を安定ソートする。この関数はすべてのシーケンスにたいしては機能せず、リストとベクターにたいしてのみ使用できることに注意されたい。*sequence*がリストなら破壊的に変更される。この関数はソートされた *sequence*をリターンして、要素の比較には *predicate*を使用する。安定ソートでは、ソートキーが等しい要素の相対順序がソートの前後で保たれる。この安定性は異なる条件により要素を並べ替えるために、連続してソートを行う場合に重要となる。

引数 *predicate*は 2 つの引数を受け取る関数でなければならない。これは *sequence*の 2 つの要素で呼び出される。昇順でソートするなら、1 つ目の要素が 2 つ目の要素より “小” なら非 `nil`、それ以外なら `nil`をリターンすること。

比較関数 *predicate*は、少なくとも `sort`の単一の呼び出しにおいて、与えられた任意の引数ペアにたいして信頼できる結果をリターンしなければならない。これは非対照的 *(antisymmetric)*、すなわち *a*が *b*より小なら、*b*が *a*より小であってはならず、推移律 *(transitive)*、すなわち *a*が *b*より小、かつ *b*が *c*より小なら、*a*は *c*より小でなければならない。これらの要件に合致しない比較関数を使用すると、`sort`の結果は予想できない。

`sort`のリストにたいする破壊的側面は、CDR を変更することにより *sequence*を形成するコンスセルを再配置することにある。非破壊ソート関数は、それらのソート順に要素を格納するために、新たなコンスセルを作成するだろう。オリジナルを破壊せずにソートしたコピーを望むなら、まず `copy-sequence`でコピーしてからソートすること。

ソートにより *sequence*のコンスセルの CAR は変化しない。元々*sequence*内で要素 `a`を含むコンスセルは、ソート後もその CAR に `a`を保持する。しかし CDR の変更により、ソート後には異なる位置に出現する。たとえば:

```
(setq nums (list 1 3 2 6 5 4 0))
     ⇒ (1 3 2 6 5 4 0)
(sort nums #'<)
     ⇒ (0 1 2 3 4 5 6)
nums
     ⇒ (1 2 3 4 5 6)
```

警告 `nums`内のリストが 0 を含まないことに注意。これはソート前と同じコンスセルだがもはやリストの先頭ではない。ソート前に引数を保持していた変数がソート済みリスト全体を保持すると仮定してはならない! かわりに `sort`の結果を保存して、それを使うこと。ほとんどの場合、わたしたちは元のリストを保持していた変数に結果を書き戻すようにしている。

```
(setq nums (sort nums #'<))
```

安定ソートの何たるかをより理解するには、以下のベクターのサンプルを考えてみよ。ソート後、`car`が 8 であるようなすべてのアイテムは `vector`の先頭にグループ化されるが、それらの相対的な順序は保たれる。`car`が 9 であるようなすべてのアイテムは `vector`の末尾にグループ化されるが、それらの相対的な順序も保たれる。

```
(setq
  vector
  (vector '(8 . "xxx") '(9 . "aaa") '(8 . "bbb") '(9 . "zzz")
          '(9 . "ppp") '(8 . "ttt") '(8 . "eee") '(9 . "fff")))
     ⇒ [(8 . "xxx") (9 . "aaa") (8 . "bbb") (9 . "zzz")
        (9 . "ppp") (8 . "ttt") (8 . "eee") (9 . "fff")]
(sort vector (lambda (x y) (< (car x) (car y))))
     ⇒ [(8 . "xxx") (8 . "bbb") (8 . "ttt") (8 . "eee")
        (9 . "aaa") (9 . "zzz") (9 . "ppp") (9 . "fff")]
```

ソートを行う他の関数については Section 32.15 [Sorting], page 803 を参照のこと。`sort`の有用な例は、Section 24.2 [Accessing Documentation], page 520 の `documentation`を参照されたい。

`seq.el`ライブラリーは、以下のようなプレフィクス `seq-`がついたシーケンス操作用の追加のマクロと関数を提供します。それらを使用するには、最初に `seq`ライブラリーをロードしなければなりません。

このライブラリー内で定義されたすべての関数は、副作用をもちません。これらは引数として渡されたすべてのシーケンス (リスト、ベクター、文字列) を変更しません。特に明記しなければ、結果は入力と同じ型のシーケンスです。述語を受け取る関数では、それらは単一の関数である必要があります。

`seq.el`ライブラリーは、シーケンシャルなデータ構造の追加型で機能するように拡張可能です。そのためにすべての関数は `cl-defgeneric`を使用して定義されています。`cl-defgeneric`を使用した拡張の追加に関する詳細は、Section 13.8 [Generic Functions], page 222 を参照してください。

`seq-elt` *sequence index* [Function]

この関数は *index*(有効な範囲は 0 から *sequence*の長さより 1 少ない整数) で指定された *sequence*の要素をリターンする。ビルトインのシーケンス型にたいする範囲外 (out-of-range) の値にたいして、`seq-elt`は `elt`と同様に振る舞う。詳細は [Definition of elt], page 96 を参照のこと。

```
(seq-elt [1 2 3 4] 2)
⇒ 3
```

`seq-elt`は `setf`を使用してセット可能な place をリターンする (Section 12.17.1 [Setting Generalized Variables], page 206 を参照)。

```
(setq vec [1 2 3 4])
(setf (seq-elt vec 2) 5)
vec
⇒ [1 2 5 4]
```

`seq-length` *sequence* [Function]

この関数は *sequence*内の要素の個数をリターンする。ビルトインのシーケンス型にたいして `seq-length`は `length`と同様に振る舞う。[Definition of length], page 95 を参照のこと。

`seqp` *object* [Function]

この関数は *object*がシーケンス (リストか配列)、または `seq.el`のジェネリック関数を通じて定義されたすべての追加シーケンス型なら非 `nil`をリターンする。これは `sequencep`の拡張された変種である。

```
(seqp [1 2])
⇒ t
(seqp 2)
⇒ nil
```

`seq-drop` *sequence n* [Function]

この関数は *sequence*の最初の *n*個 (整数) を除く、すべての要素をリターンする.*n*が 0 以下なら結果は *sequence*。

```
(seq-drop [1 2 3 4 5 6] 3)
⇒ [4 5 6]
(seq-drop "hello world" -4)
⇒ "hello world"
```

`seq-take` *sequence n* [Function]

この関数は *sequence*の最初の *n*個 (整数) の要素をリターンする。*n*が 0 以下なら結果は `nil`。

```
(seq-take '(1 2 3 4) 3)
⇒ (1 2 3)
```

```
(seq-take [1 2 3 4] 0)
⇒ []
```

`seq-take-while` *predicate sequence* [Function]

この関数は *sequence*のメンバーを順にリターンし、*predicate*が最初に `nil`をリターンした要素の前で停止する。

```
(seq-take-while (lambda (elt) (> elt 0)) '(1 2 3 -1 -2))
⇒ (1 2 3)
(seq-take-while (lambda (elt) (> elt 0)) [-1 4 6])
⇒ []
```

`seq-drop-while` *predicate sequence* [Function]

この関数は *predicate*が最初に `nil`をリターンした要素から、*sequence*のメンバーを順にリターンする。

```
(seq-drop-while (lambda (elt) (> elt 0)) '(1 2 3 -1 -2))
⇒ (-1 -2)
(seq-drop-while (lambda (elt) (< elt 0)) [1 4 6])
⇒ [1 4 6]
```

`seq-do` *function sequence* [Function]

この関数は *sequence*の各要素にたいして、(恐らくは副作用を得るために) 順番に *function*を適用して、*sequence*をリターンする。

`seq-map` *function sequence* [Function]

この関数は *sequence*の各要素に *function*を適用した結果をリターンする。リターン値はリスト。

```
(seq-map #'1+ '(2 4 6))
⇒ (3 5 7)
(seq-map #'symbol-name [foo bar])
⇒ ("foo" "bar")
```

`seq-map-indexed` *function sequence* [Function]

この関数は *sequence*の各要素および *seq*であるようなインデックスに *function*を適用した結果をリターンする。リターン値はリスト。

```
(seq-map-indexed (lambda (elt idx)
                   (list idx elt))
                 '(a b c))
⇒ ((0 a) (b 1) (c 2))
```

`seq-mapn` *function* **&rest** *sequences* [Function]

この関数は *sequences*の各要素に *function*を適用した結果をリターンする。 *function*の arity (関数が受け取れる引数の個数。Section 13.1 [What Is a Function], page 209 を参照) はシーケンスの個数にマッチしなければならない。マッピングは最短のシーケンス終端で停止する。リターン値はリスト。

```
(seq-mapn #'+ '(2 4 6) '(20 40 60))
⇒ (22 44 66)
(seq-mapn #'concat '("moskito" "bite") ["bee" "sting"])
⇒ ("moskitobee" "bitesting")
```

`seq-filter` *predicate sequence* [Function]

この関数は *predicate*が非 `nil`をリターンした *sequence*内のすべての要素のリストをリターンする。

```
(seq-filter (lambda (elt) (> elt 0)) [1 -1 3 -3 5])
⇒ (1 3 5)
(seq-filter (lambda (elt) (> elt 0)) '(-1 -3 -5))
⇒ nil
```

`seq-remove` *predicate sequence* [Function]

この関数は *predicate*が `nil`をリターンした *sequence*内のすべての要素のリストをリターンする。

```
(seq-remove (lambda (elt) (> elt 0)) [1 -1 3 -3 5])
⇒ (-1 -3)
(seq-remove (lambda (elt) (< elt 0)) '(-1 -3 -5))
⇒ nil
```

`seq-reduce` *function sequence initial-value* [Function]

この関数は *initial-value*と *sequence*の 1 つ目の要素で *function*を呼び出し、次にその結果と *sequence*の 2 つ目の要素で *function*を呼び出し、その次にその結果と *sequence*の 3 つ目の要素で、... と呼び出した結果をリターンする。*function*は引数が 2 つの関数であること。*sequence*が空なら、これは *function*を呼び出さずに *initial-value*をリターンする。

```
(seq-reduce #'+ [1 2 3 4] 0)
⇒ 10
(seq-reduce #'+ '(1 2 3 4) 5)
⇒ 15
(seq-reduce #'+ '() 3)
⇒ 3
```

`seq-some` *predicate sequence* [Function]

この関数は *sequence*の各要素に順に *predicate*を適用してリターンされた、最初の非 `nil`値をリターンする。

```
(seq-some #'numberp ["abc" 1 nil])
⇒ t
(seq-some #'numberp ["abc" "def"])
⇒ nil
(seq-some #'null ["abc" 1 nil])
⇒ t
(seq-some #'1+ [2 4 6])
⇒ 3
```

`seq-find` *predicate sequence* **&optional** *default* [Function]

この関数は *predicate*が非 `nil`をリターンした、*sequence*内の最初の要素をリターンする。*predicate*にマッチする要素がなければ、この関数は *default*をリターンする。

この関数は見つかった要素が *default*と等しい場合、要素が見つかったかどうかを知る術がないので曖昧さをもつことに注意。

```
(seq-find #'numberp ["abc" 1 nil])
⇒ 1
```

```
(seq-find #'numberp ["abc" "def"])
⇒ nil
```

`seq-every-p` *predicate sequence* [Function]

この関数は *sequence*の各要素に *predicate*を適用して、すべてが非 `nil`をリターンしたら非 `nil`をリターンする。

```
(seq-every-p #'numberp [2 4 6])
⇒ t
(seq-every-p #'numberp [2 4 "6"])
⇒ nil
```

`seq-empty-p` *sequence* [Function]

この関数は *sequence*が空なら `nil`をリターンする。

```
(seq-empty-p "not empty")
⇒ nil
(seq-empty-p "")
⇒ t
```

`seq-count` *predicate sequence* [Function]

この関数は *sequence*内で *predicate*が非 `nil`をリターンした要素の個数をリターンする。

```
(seq-count (lambda (elt) (> elt 0)) [-1 2 0 3 -2])
⇒ 2
```

`seq-sort` *function sequence* [Function]

この関数は *function*に応じてソートされた *sequence*のコピーをリターンする。*function*は 2 つの引数を受け取り、1 つ目の引数が 2 つ目より前にソートされるべきなら非 `nil`をリターンする。

`seq-sort-by` *function predicate sequence* [Function]

この関数は `seq-sort`と似ているがソート前に *sequence*の要素に *function*を適用して変換する点が異なる。*function*は単一の引数を受け取る関数。

```
(seq-sort-by #'seq-length #'> ["a" "ab" "abc"])
⇒ ["abc" "ab" "a"]
```

`seq-contains-p` *sequence elt* **&optional** *function* [Function]

この関数は *sequence*内の少なくとも 1 つの要素が *elt*と equal なら非 `nil`をリターンする。オプション引数 *function*が非 `nil`なら、それはデフォルトの `equal`のかわりに使用する 2 つの引数を受け取る関数であること。

```
(seq-contains '(symbol1 symbol2) 'symbol1)
⇒ symbol1
(seq-contains '(symbol1 symbol2) 'symbol3)
⇒ nil
```

`seq-set-equal-p` *sequence1 sequence2* **&optional** *testfn* [Function]

この関数は順序とは無関係に *sequence1*と *sequence2*が同じ要素を含むかどうかをチェックする。オプション引数 *testfn*が非 `nil`なら、デフォルトの `equal`のかわりに使用する 2 つの引数を受け取る関数であること。

```
(seq-set-equal-p '(a b c) '(c b a))
⇒ t
```

```
(seq-set-equal-p '(a b c) '(c b))
⇒ nil
(seq-set-equal-p '("a" "b" "c") '("c" "b" "a"))
⇒ t
(seq-set-equal-p '("a" "b" "c") '("c" "b" "a") #'eq)
⇒ nil
```

`seq-position` *sequence elt* **&optional** *function* [Function]

この関数は *elt*と `equal`であるような *sequence*内の最初の要素のインデックスをリターンする。オプション引数 *function*が非 `nil`なら、それはデフォルトの `equal`のかわりに使用する 2 つの引数を受け取る関数であること。

```
(seq-position '(a b c) 'b)
⇒ 1
(seq-position '(a b c) 'd)
⇒ nil
```

`seq-uniq` *sequence* **&optional** *function* [Function]

この関数は重複を削除した *sequence*の要素のリストをリターンする。オプション引数 *function* が非 `nil`なら、それはデフォルトの `equal`のかわりに使用する 2 つの引数を受け取る関数であること。

```
(seq-uniq '(1 2 2 1 3))
⇒ (1 2 3)
(seq-uniq '(1 2 2.0 1.0) #'=)
⇒ (1 2)
```

`seq-subseq` *sequence start* **&optional** *end* [Function]

この関数は *sequence*の *start*から *end*(いずれも整数) までのサブセットをリターンする (*end* のデフォルトは最後の要素)。*start*か *end*が負なら *sequence*の最後から数える。

```
(seq-subseq '(1 2 3 4 5) 1)
⇒ (2 3 4 5)
(seq-subseq '[1 2 3 4 5] 1 3)
⇒ [2 3]
(seq-subseq '[1 2 3 4 5] -3 -1)
⇒ [3 4]
```

`seq-concatenate` *type* **&rest** *sequences* [Function]

この関数は *sequences*を結合して作成された *type*型のシーケンスをリターンする。*type*は `vector`、`list`、`string`のいずれか。

```
(seq-concatenate 'list '(1 2) '(3 4) [5 6])
⇒ (1 2 3 4 5 6)
(seq-concatenate 'string "Hello " "world")
⇒ "Hello world"
```

`seq-mapcat` *function sequence* **&optional** *type* [Function]

この関数は *sequence*の各要素に *function*を適用した結果に、`seq-concatenate`を適用した結果をリターンする。結果は *type*型のシーケンス、または *type*が `nil`ならリストである。

```
(seq-mapcat #'seq-reverse '((3 2 1) (6 5 4)))
⇒ (1 2 3 4 5 6)
```

`seq-partition` *sequence n* [Function]
この関数は長さ *n*のサブシーケンスへグループ化した *sequence*の要素のリストをリターンする。最後のシーケンスに含まれる要素は *n*より少ないかもしれない。*n*は整数であること。*n*が 0 以下の整数ならリターン値は `nil`。

```
(seq-partition '(0 1 2 3 4 5 6 7) 3)
⇒ ((0 1 2) (3 4 5) (6 7))
```

`seq-intersection` *sequence1 sequence2* **&optional** *function* [Function]
この関数は *sequence1*と *sequence2*の両方に出現する要素のリストをリターンする。オプション引数 *function*が非 `nil`なら、それはデフォルトの `equal`のかわりに比較に使用する 2 つの引数を受け取る関数であること。

```
(seq-intersection [2 3 4 5] [1 3 5 6 7])
⇒ (3 5)
```

`seq-difference` *sequence1 sequence2* **&optional** *function* [Function]
この関数は *sequence1*に出現するが *sequence2*に出現しない要素のリストをリターンする。オプション引数 *function*が非 `nil`なら、それはデフォルトの `equal`のかわりに比較に使用する 2 つの引数を受け取る関数であること。

```
(seq-difference '(2 3 4 5) [1 3 5 6 7])
⇒ (2 4)
```

`seq-group-by` *function sequence* [Function]
この関数は *sequence*の各要素に *function*を適用して、その結果をキーとして *sequence*を alist に分割する。キーの比較には `equal`を使用する。

```
(seq-group-by #'integerp '(1 2.1 3 2 3.2))
⇒ ((t 1 3 2) (nil 2.1 3.2))
(seq-group-by #'car '((a 1) (b 2) (a 3) (c 4)))
⇒ ((b (b 2)) (a (a 1) (a 3)) (c (c 4)))
```

`seq-into` *sequence type* [Function]
この関数はシーケンス *sequence*を *type*型のシーケンスに変換する。*type*は `vector`、`string`、`list`のいずれかであること。

```
(seq-into [1 2 3] 'list)
⇒ (1 2 3)
(seq-into nil 'vector)
⇒ []
(seq-into "hello" 'vector)
⇒ [104 101 108 108 111]
```

`seq-min` *sequence* [Function]
この関数は *sequence*の最小の要素をリターンする。*sequence*の要素は数字かマーカー (Chapter 31 [Markers], page 768 を参照) でなければならない。

```
(seq-min [3 1 2])
⇒ 1
(seq-min "Hello")
⇒ 72
```

`seq-max` *sequence* [Function]

この関数は *sequence*の最大の要素をリターンする。*sequence*の要素は数字かマーカーでなければならない。

```
(seq-max [1 3 2])
⇒ 3
(seq-max "Hello")
⇒ 111
```

`seq-doseq` (*var sequence*) *body*... [Macro]

このマクロは `dolist` (Section 11.5 [Iteration], page 159 を参照) と同様だが、*sequence*にリスト、ベクター、文字列のいずれかを指定できる点が異なる。これ主な利点は副作用である。

`seq-let` *var-sequence val-sequence body*... [Macro]

このマクロは *var-sequence*内で定義される変数に *val-sequence*の対応する要素をバインドする。これは分割代入 *(destructuring binding)* として知られている。*var-sequence*の要素は、ネストされた非構造化を許容することにより自身にシーケンスを含むことができる。

*var-sequence*シーケンスには、*val-sequence*の残りにバインドされる変数名が後続するような `&rest`マーカーを含めることもできる。

```
(seq-let [first second] [1 2 3 4]
  (list first second))
⇒ (1 2)
(seq-let (_ a _ b) '(1 2 3 4)
  (list a b))
⇒ (2 4)
(seq-let [a [b [c]]] [1 [2 [3]]]
  (list a b c))
⇒ (1 2 3)
(seq-let [a b &rest others] [1 2 3 4]
  others)
⇒ [3 4]
```

`pcase`パターンは分割代入にたいする代替えの機能を提供する。Section 11.4.4 [Destructuring with pcase Patterns], page 158 を参照のこと。

`seq-random-elt` *sequence* [Function]

この関数は *sequence*の要素をランダムにリターンする。

```
(seq-random-elt [1 2 3 4])
⇒ 3
(seq-random-elt [1 2 3 4])
⇒ 2
(seq-random-elt [1 2 3 4])
⇒ 4
(seq-random-elt [1 2 3 4])
⇒ 2
(seq-random-elt [1 2 3 4])
⇒ 1
```

*sequence*が空ならこの関数はエラーをシグナルする。

6.2 配列

配列 *(array)* オブジェクトは、いくつかの Lisp オブジェクトを保持するスロットをもち、これらのオブジェクトは配列の要素と呼ばれます。配列内の任意の要素は一定時間でアクセスされます。対照的にリスト内の要素のアクセスに要する時間は、その要素がリスト内のどの位置にあるかに比例します。

Emacs は 4 つの配列型 —文字列 *(strings、Section 2.4.8 [String Type], page 19* を参照*)*、ベクター *(vectors、Section 2.4.9 [Vector Type], page 21* を参照*)*、ブールベクター *(bool-vectors、Section 2.4.11 [Bool-Vector Type], page 22* を参照*)*、文字テーブル *(char-tables、Section 2.4.10 [Char-Table Type], page 21* を参照*)* — を定義しており、これらはすべて 1 次元です。ベクターと文字テーブルは任意の型の要素を保持できますが、文字列は文字だけ、ブールベクターは `t`か `nil`しか保持できません。

4 種のすべての配列はこれらの特性を共有します:

- 配列の 1 番目の要素はインデックス 0、2 番目はインデックス 1、... となる。これは *0* 基準 *(zero-origin)* のインデックスづけと呼ばれる。たとえば 4 要素の配列のインデックスは 0、1、2、3。
- 配列の長さは一度配列が作成されたら固定されるので、既存の配列の長さは変更できない。
- 評価において配列は定数 — つまりそれ自身へと評価される。
- 配列の要素は関数 `aref`で参照したり、関数 `aset`で変更できる (Section 6.3 [Array Functions], page 107 を参照)。

配列を作成したとき、文字テーブル以外では長さを指定しなければなりません。文字テーブルの長さは文字コードの範囲により決定されるので長さを指定できません。

原則として、テキスト文字の配列が欲しい場合は、文字列とベクターのどちらかを使用できます。実際のところ 4 つの理由により, そのような用途にたいしては、わたしたちは常に文字列を選択します:

- 文字列は同じ要素をもつベクターと比較して占めるスペースが 1/4 である。
- 文字列の内容はテキストとして、より明解な方法によりプリントされる。
- 文字列はテキストプロパティーを保持できる。Section 32.19 [Text Properties], page 813 を参照のこと。
- Emacs の特化した編集機能と I/O 機能の多くが文字列だけに適用される。たとえば文字列をバッファーに挿入する方法では、文字のベクターをバッファーに挿入できない。Chapter 4 [Strings and Characters], page 51 を参照のこと

対照的に、(キーシーケンスのような) キーボード入力文字の配列では、多くのキーボード入力文字は文字列に収まる範囲の外にあるので、ベクターが必要になるでしょう。Section 21.8.1 [Key Sequence Input], page 404 を参照してください。

6.3 配列を操作する関数

このセクションではすべての型の配列に適用される関数を説明します。

`arrayp` *object* [Function]

この関数は *object*が配列 (ベクター、文字列、ブールベクター、文字テーブル) なら `t`をリターンする。

```
(arrayp [a])
     ⇒ t
(arrayp "asdf")
     ⇒ t
(arrayp (syntax-table))    ;; 文字テーブル
     ⇒ t
```

`aref` *arr index* [Function]

この関数は *arr* (配列かレコード) の *index*番目の要素をリターンする。1 番目の要素のインデクスは 0。

```
(setq primes [2 3 5 7 11 13])
     ⇒ [2 3 5 7 11 13]
(aref primes 4)
     ⇒ 11
(aref "abcdefg" 1)
     ⇒ 98            ; 'b'の ASCIIコードは 98
```

Section 6.1 [Sequence Functions], page 95 の関数 `elt`も参照されたい。

`aset` *array index object* [Function]

この関数は *array*の *index*番目の要素を *object*にセットする。この関数は *object*をリターンする。

```
(setq w (vector 'foo 'bar 'baz))
     ⇒ [foo bar baz]
(aset w 0 'fu)
     ⇒ fu
w
     ⇒ [fu bar baz]

;; copy-sequenceは後で変更する文字列をコピーする
(setq x (copy-sequence "asdfasfd"))
     ⇒ "asdfasfd"
(aset x 3 ?Z)
     ⇒ 90
x
     ⇒ "asdZasfd"
```

*array*は mutable であること。Section 2.9 [Mutability], page 35 を参照のこと。

*array*が文字列で *object*が文字でなければ、結果は `wrong-type-argument`エラーとなる。この関数は文字列の挿入で必要なら、ユニバイト文字列をマルチバイト文字列に変換する。

`fillarray` *array object* [Function]

この関数は配列 *array*を *object*で充填するので、*array*のすべての要素は *object*になる。この関数は *array*をリターンする。

```
(setq a (copy-sequence [a b c d e f g]))
     ⇒ [a b c d e f g]
(fillarray a 0)
     ⇒ [0 0 0 0 0 0 0]
a
     ⇒ [0 0 0 0 0 0 0]
(setq s (copy-sequence "When in the course"))
     ⇒ "When in the course"
(fillarray s ?-)
     ⇒ "------------------"
```

*array*が文字列で *object*が文字でなければ、結果は `wrong-type-argument`エラーとなる。

配列と判っているオブジェクトにたいしては、一般的なシーケンス関数 `copy-sequence`と `length` が有用なときがよくあります。Section 6.1 [Sequence Functions], page 95 を参照してください。

6.4 ベクター

ベクター *(vector)* とは任意の Lisp オブジェクトを要素にもつことができる、一般用途のための配列です (対照的に文字列の要素は文字のみ。Chapter 4 [Strings and Characters], page 51 を参照)。Emacs ではベクターはキーシーケンス (Section 22.1 [Key Sequences], page 422 を参照)、シンボル検索用のテーブル (Section 9.3 [Creating Symbols], page 126 を参照)、バイトコンパイルされた関数表現の一部 (Chapter 17 [Byte Compilation], page 285 を参照) などの多くの目的で使用されます。

他の配列と同様、ベクターは 0 基準のインデックスづけを使用し、1 番目の要素はインデックス 0 になります。

ベクターは角カッコ (square brackets) で囲まれた要素としてプリントされます。したがってシンボル `a`、`b`、`a`を要素にもつベクターは、`[a b a]`とプリントされます。Lisp 入力として同じ方法でベクターを記述できます。

文字列や数値と同様にベクターは定数として評価され、評価された結果は同じベクターになります。ベクターの要素は評価も確認もされません。Section 10.1.1 [Self-Evaluating Forms], page 134 を参照してください。角カッコ (square brackets) で記述されたベクターを `aset`や他の破壊的操作を通じて修正しないでください。Section 2.9 [Mutability], page 35 を参照してください。

以下はこれらの原理を表す例です:

```
(setq avector [1 two '(three) "four" [five]])
     ⇒ [1 two '(three) "four" [five]]
(eval avector)
     ⇒ [1 two '(three) "four" [five]]
(eq avector (eval avector))
     ⇒ t
```

6.5 ベクターのための関数

ベクターに関連した関数をいくつか示します:

`vectorp` *object* [Function]

この関数は *object*がベクターなら `t`をリターンする。

```
(vectorp [a])
     ⇒ t
(vectorp "asdf")
     ⇒ nil
```

`vector` **&rest** *objects* [Function]

この関数は引数 *objects*を要素にもつベクターを作成してリターンする。

```
(vector 'foo 23 [bar baz] "rats")
     ⇒ [foo 23 [bar baz] "rats"]
(vector)
     ⇒ []
```

`make-vector` *length object* [Function]

この関数は各要素が *object*に初期化された、*length*個の要素からなる新しいベクターをリターンする。

```
(setq sleepy (make-vector 9 'Z))
     ⇒ [Z Z Z Z Z Z Z Z Z]
```

`vconcat` **&rest** *sequences* [Function]

この関数は *sequences*のすべての要素を含む新しいベクターをリターンする。引数 *sequences* は正リスト、ベクター、文字列、ブールベクター。*sequences*が与えられければ空のベクターがリターンされる。

値は空のベクター、またはすべての既存ベクターと `eq`ではないような空ではない新しいベクターのいずれか。

```
(setq a (vconcat '(A B C) '(D E F)))
     ⇒ [A B C D E F]
(eq a (vconcat a))
     ⇒ nil
(vconcat)
     ⇒ []
(vconcat [A B C] "aa" '(foo (6 7)))
     ⇒ [A B C 97 97 foo (6 7)]
```

`vconcat`関数は、引数としてバイトコード関数オブジェクトも受け取ることができる。これはバイトコード関数オブジェクトの内容全体にアクセスするのを容易にするための特別な機能である。Section 17.7 [Byte-Code Objects], page 291 を参照のこと。

結合を行なう他の関数については Section 13.6 [Mapping Functions], page 219 の `mapconcat`、Section 4.3 [Creating Strings], page 52 の `concat`、Section 5.4 [Building Lists], page 75 の `append`を参照されたい。

`append`関数はベクターを同じ要素をもつリストに変換する方法も提供します:

```
(setq avector [1 two (quote (three)) "four" [five]])
     ⇒ [1 two '(three) "four" [five]]
(append avector nil)
     ⇒ (1 two '(three) "four" [five])
```

6.6 文字テーブル

文字テーブル (char-table) はベクターとよく似ていますが、文字テーブルは文字コードによりインデックスづけされます。文字テーブルのインデックスには、修飾キーをともなわない任意の有効な文字コードを使用できます。他の配列と同様に、`aref`と `aset`で文字テーブルの要素にアクセスできます。加えて、文字テーブルは追加のデータを保持するために、特定の文字コードに関連づけられていないエキストラスロット *(extra slots)* をもつことができます。ベクターと同様、文字テーブルは定数として評価され、任意の型の要素を保持できます。

文字テーブルはそれぞれサブタイプ *(subtype)* をもち、これは 2 つの目的をもつシンボルです:

- サブタイプはそれがなんのための文字テーブルなのかを簡単に表す方法を提供する。たとえばディスプレーテーブル (display tables) はサブタイプが `display-table`の文字テーブルであり、構文テーブル (syntax tables) はサブタイプが `syntax-table`の文字テーブル。以下で説明するように関数 `char-table-subtype`を使用してサブタイプの問い合わせが可能。

- サブタイプは文字テーブル内のいくつかのエキストラスロット *(extra slots)* を制御する。エキストラスロットの数は、そのサブタイプの `char-table-extra-slots`シンボルプロパティー (Section 9.4 [Symbol Properties], page 129 を参照) により指定され、値は 0 から 10 の整数。サブタイプにそのようなシンボルプロパティーがなければ、その文字テーブルはエキストラスロットをもたない。

文字テーブルは親 *(parent)* をもつことができ、これは他の文字テーブルです。文字テーブルが親をもつ場合、その文字テーブルで特定の文字 *c*にたいして `nil`が指定されていたら、親と指定された文字テーブルで指定された値を継承します。言い方を変えると、文字テーブル *char-table*で *c*に `nil`が指定されていたら、`(aref char-table c)`は *char-table*の親の値をリターンします。

文字テーブルはデフォルト値 *(default value)* をもつこともできます。デフォルト値をもつとき、文字テーブル *char-table*が *c*にたいして `nil`値を指定すると、`(aref char-table c)`はデフォルト値をリターンします。

`make-char-table` *subtype* **&optional** *init* [Function]
サブタイプ *subtype*(シンボル) をもつ、新たに作成された文字テーブルをリターンする。各要素は *init*に初期化され、デフォルトは `nil`。文字テーブルが作成された後で、文字テーブルのサブタイプを変更することはできない。

すべての文字テーブルは、インデックスとなる任意の有効な文字テーブルのための空間をもつので、文字テーブルの長さを指定する引数はない。

*subtype*がシンボルプロパティー `char-table-extra-slots`をもつなら、それはその文字列テーブル内のエキストラスロットの数を指定する。値には 0 から 10 の整数を指定し、これ以外なら `make-char-table`はエラーとなる。*subtype*がシンボルプロパティー `char-table-extra-slots`(Section 5.9 [Property Lists], page 92 を参照) をもたなければ、その文字テーブルはエキストラスロットをもたない。

`char-table-p` *object* [Function]
この関数は *object*が文字テーブルなら `t`、それ以外は `nil`をリターンする。

`char-table-subtype` *char-table* [Function]
この関数は *char-table*のサブタイプのシンボルをリターンする。

文字テーブルのデフォルト値にアクセスするための特別な関数は存在しません。これを行なうには `char-table-range`を使用します (以下参照)。

`char-table-parent` *char-table* [Function]
この関数は *char-table*の親をリターンする。親は常に `nil`か他の文字テーブルである。

`set-char-table-parent` *char-table new-parent* [Function]
この関数は *char-table*の親を *new-parent*にセットする。

`char-table-extra-slot` *char-table n* [Function]
この関数は *char-table*のエキストラスロット *n* (0 基準) の内容をリターンする。文字テーブルのエキストラスロットの数は文字テーブルのサブタイプにより決定される。

`set-char-table-extra-slot` *char-table n value* [Function]
この関数は *char-table*のエキストラスロット *n* (0 基準) に *value*を格納する。

文字テーブルは 1 つの文字コードにたいして 1 つの要素値 (element value) を指定できます。文字テーブルは文字セット全体にたいして値を指定することもできます。

`char-table-range` *char-table range* [Function]
この関数は文字範囲 *range*にたいして *char-table*で指定された値をリターンする。可能な *range* は以下のとおり:

`nil` デフォルト値への参照。

char 文字 *char*にたいする要素への参照 (*char*は有効な文字コードであると仮定)。

`(`*from* `.` *to*`)`
包括的な範囲 '`[`*from*`..`*to*`]`'内のすべての文字を参照するコンスセル。

`set-char-table-range` *char-table range value* [Function]
この関数は *char-table*内の文字範囲 *range*にたいして値をセットする。可能な *range*は以下のとおり:

`nil` デフォルト値への参照。

`t` 文字コード範囲の全体を参照。

char 文字 *char*にたいする要素への参照 (*char*は有効な文字コードであると仮定)。

`(`*from* `.` *to*`)`
包括的な範囲 '`[`*from*`..`*to*`]`'内のすべての文字を参照するコンスセル。

`map-char-table` *function char-table* [Function]
この関数は *char-table*の非 `nil`値ではない各要素にたいして引数 *function*を呼び出す。*function*の呼び出しでは 2 つの引数 (key と value) が指定される。key は `char-table-range`にたいする可能な *range* (有効な文字か、同じ値を共有する文字範囲を指定するコンスセル `(`*from* `.` *to*`)`)。value は `(char-table-range` *char-table key*`)`がリターンする値。

全体として、*function*に渡される key-value のペアは *char-table*に格納されたすべての値を表す。

リターン値は常に `nil`である。`map-char-table`呼び出しを有用にするために *function*は副作用をもつこと。たとえば以下は構文テーブルを調べる方法:

```
(let (accumulator)
   (map-char-table
    (lambda (key value)
      (setq accumulator
            (cons (list
                   (if (consp key)
                       (list (car key) (cdr key))
                     key)
                   value)
                  accumulator)))
    (syntax-table))
   accumulator)
⇒
(((2597602 4194303) (2)) ((2597523 2597601) (3))
 ... (65379 (5 . 65378)) (65378 (4 . 65379)) (65377 (1))
 ... (12 (0)) (11 (3)) (10 (12)) (9 (0)) ((0 8) (3)))
```

6.7 ブールベクター

ブールベクター (bool-vector) はベクターとよく似ていますが、値に `t`と `nil`しか格納できません。ブールベクターの要素に非 `nil`値の格納を試みたると、そこには `t`が格納されます。すべての配列と同様、ブールベクターのインデックスは 0 から開始され、一度ブールベクターが作成されたら長さを変更することはできません。ブールベクターは定数として評価されます。

ブールベクターを処理する特別な関数がいくつかあります。その関数以外にも、他の種類の配列に使用されるのと同じ関数でブールベクターを操作できます。

`make-bool-vector` *length initial* [Function]
*initial*に初期化された *length*要素の新しいブールベクターをリターンする。

`bool-vector` **&rest** *objects* [Function]
この関数は引数 *objects*を要素にもつブールベクターを作成してリターンする。

`bool-vector-p` *object* [Function]
この関数は *object*がブールベクターであれば `t`、それ以外は `nil`をリターンする。

以下で説明するように、ブールベクターのセット処理を行なう関数がいくつかあります:

`bool-vector-exclusive-or` *a b* **&optional** *c* [Function]
ブールベクター *a*と *b*のビットごとの排他的論理和 *(bitwise exclusive or)* をリターンする。オプション引数 *c*が与えられたら、この処理の結果は *c*に格納される。引数にはすべて同じ長さのブールベクターを指定すること。

`bool-vector-union` *a b* **&optional** *c* [Function]
ブールベクター *a*と *b*のビットごとの論理和 *(bitwise or)* をリターンする。オプション引数 *c*が与えられたら、この処理の結果は *c*に格納される。引数にはすべて同じ長さのブールベクターを指定すること。

`bool-vector-intersection` *a b* **&optional** *c* [Function]
ブールベクター *a*と *b*のビットごとの論理積 *(bitwise and)* をリターンする。オプション引数 *c*が与えられたら、この処理の結果は *c*に格納される。引数にはすべて同じ長さのブールベクターを指定すること。

`bool-vector-set-difference` *a b* **&optional** *c* [Function]
ブールベクター *a*と *b*の差集合 *(set difference)* をリターンする。オプション引数 *c*が与えられたら、この処理の結果は *c*に格納される。引数にはすべて同じ長さのブールベクターを指定すること。

`bool-vector-not` *a* **&optional** *b* [Function]
ブールベクター *a*の補集合 *(set complement)* をリターンする。オプション引数 *b*が与えられたら、この処理の結果は *b*に格納される。引数にはすべて同じ長さのブールベクターを指定すること。

`bool-vector-subsetp` *a b* [Function]
*a*内のすべての `t`値が *b*でも `t`値なら `t`、それ以外は `nil`をリターンする。引数にはすべて同じ長さのブールベクターを指定すること。

`bool-vector-count-consecutive` *a b i* [Function]
*i*から始まる *a*の、*b*と等しい連続する要素の数をリターンする。`a`はブールベクターで、*b*は `t`か `nil`、*i*は `a`のインデックス。

`bool-vector-count-population` *a* [Function]
ブールベクター *a*から `t`であるような要素の数をリターンする。

長さ 8 以下のブール値のプリント表記は 1 文字で表されます。

```
(bool-vector t nil t nil)
     ⇒ #&4"^E"
(bool-vector)
     ⇒ #&0""
```

他のベクター同様、`vconcat`を使用してブールベクターをプリントできます:

```
(vconcat (bool-vector nil t nil t))
     ⇒ [nil t nil t]
```

以下はブールベクターを作成、確認、更新する別の例です:

```
(setq bv (make-bool-vector 5 t))
     ⇒ #&5"^_"
(aref bv 1)
     ⇒ t
(aset bv 3 nil)
     ⇒ nil
bv
     ⇒ #&5"^W"
```

control-_の 2 進コードは 11111、control-W は 10111 なので、この結果は理にかなっています。

6.8 オブジェクト用固定長リングの管理

リング *(ring)* は挿入、削除、ローテーション、剰余 (modulo) でインデックスづけされた、参照と走査 (traversal) をサポートする固定長のデータ構造です。`ring`パッケージにより効率的なリングデータ構造が実装されています。このパッケージは、このセクションにリストした関数を提供します。

Emacs にある kill リングやマークリングのようないくつかのリングは、実際には単なるリストとして実装されていることに注意してください。したがってこれらのリングにたいしては、以下の関数は機能しないでしょう。

`make-ring` *size* [Function]
この関数は *size*オブジェクトを保持できる、新しいリングをリターンする。*size*は整数。

`ring-p` *object* [Function]
この関数は *object*がリングなら `t`、それ以外は `nil`をリターンする。

`ring-size` *ring* [Function]
この関数は *ring*の最大の要素数をリターンする。

`ring-length` *ring* [Function]
この関数は *ring*に現在含まれるオブジェクトの数をリターンする。値が `ring-size`のリターンする値を超えることはない。

`ring-elements` *ring* [Function]
この関数は *ring*内のオブジェクトのリストをリターンする。リストの順序は新しいオブジェクトが先頭になる。

`ring-copy` *ring* [Function]
この関数は新しいリングとして *ring*のコピーをリターンする。新しいリングは *ring*と同じ (`eq` な) オブジェクトを含む。

`ring-empty-p` *ring* [Function]
この関数は *ring*が空なら `t`、それ以外は `nil`をリターンする。

リング内の 1 番新しい要素は常にインデックス 0 をもちます。より大きいインデックスは、より古い要素に対応します。インデックスはリング長の modulo により計算されます。インデックス −1 は 1 番古い要素、−2 は次に古い要素、... となります。

`ring-ref` *ring index* [Function]
この関数はインデックス *index*にある *ring*内のオブジェクトをリターンする。*index*には負やリング長より大きい数を指定できる。*ring*が空なら `ring-ref`はエラーをシグナルする。

`ring-insert` *ring object* [Function]
この関数は 1 番新しい要素として *object*を *ring*に挿入して *object*をリターンする。

リングが満杯なら新しい要素用の空きを作るために、挿入により 1 番古い要素が削除される。

`ring-remove` *ring* **&optional** *index* [Function]
*ring*からオブジェクトを削除してそのオブジェクトをリターンする。引数 *index*はどのアイテムを削除するかを指定する。これが `nil`なら、それは 1 番古いアイテムを削除することを意味する。*ring*が空なら `ring-remove`はエラーをシグナルする。

`ring-insert-at-beginning` *ring object* [Function]
この関数は 1 番古い要素として *object*を *ring*に挿入する。リターン値に意味はない。

リングが満杯なら、この関数は挿入される要素のための空きを作るために 1 番新しい要素を削除する。

`ring-resize` *ring size* [Function]
*ring*のサイズを *size*にセットする。新たなサイズのほうが小さければリング内の古いアイテムは破棄される。

リングサイズを超過しないよう注意すれば、そのリングを FIFO(first-in-first-out: 先入れ先出し) のキューとして使用することができます。たとえば:

```
(let ((fifo (make-ring 5)))
  (mapc (lambda (obj) (ring-insert fifo obj))
        '(0 one "two"))
  (list (ring-remove fifo) t
        (ring-remove fifo) t
        (ring-remove fifo)))
     ⇒ (0 t one t "two")
```

7 レコード

レコードの目的は Emacs にビルトインされていない新たな型のオブジェクトをプログラマーが作成できるようにすることです。これらは `cl-defstruct`や `defclass`のインスタンスを表現する基礎として使用されています。

レコードオブジェクトは内部的にはベクターと似ています。レコードオブジェクトのスロットは `aref`を使用してアクセス可能であり、`copy-sequence`を使用してコピーできます。しかし最初のスロットは `type-of`がリターンする型を保持するために使用されます。更に現実装ではレコードは最大で 4096 スロットを所有できますが、ベクターはそれより多くのスロットを所有できます。レコードは配列と同じように 0 基準のインデックスを使用するので、最初のスロットはインデックスが 0 になります。

型スロット (type slot) はシンボルか型記述子であるべきです。型記述子なら型を命名するシンボルがリターンされます。Section 2.4.18 [Type Descriptors], page 24 を参照してください。その他の種類のオブジェクトはすべてそのままリターンされます。

レコードのプリント表現では '`#s`'の後にコンテンツを指定するリストが続きます。リストの最初の要素はそのレコードの型である必要があります。後続の要素はレコードのスロットです。

レコードの型を新たに定義する Lisp プログラムは他の型名との衝突を避けるために、通常はレコード型を導入する部分では型名にパッケージの命名規約を使用するべきです。衝突する可能性がある型名は、パッケージが定義したレコード型のロード時には不明かもしれないことに注意してください。これらは将来のある時点でロードされる可能性があります。

レコードは評価にたいして定数とみなされます。評価すると結果は同じレコードになります。レコードは評価されずにスロットのチェックさえ行われません。Section 10.1.1 [Self-Evaluating Forms], page 134 を参照してください。

7.1 レコード関数

`recordp` *object* [Function]

この関数は *object*がレコードなら `t`をリターンする。

```
(recordp #s(a))
     ⇒ t
```

`record` *type* **&rest** *objects* [Function]

この関数は型が *type*であり、残りのスロットが残りの引数 *objects*であるようなレコードを作成してリターンする。

```
(record 'foo 23 [bar baz] "rats")
     ⇒ #s(foo 23 [bar baz] "rats")
```

`make-record` *type length object* [Function]

この関数は型が *type*、*object*で初期化されたスロット数が *length*の新たなレコードをリターンする。

```
(setq sleepy (make-record 'foo 9 'Z))
     ⇒ #s(foo Z Z Z Z Z Z Z Z Z)
```

7.2 後方互換

レコードを使用しない古いバージョンの `cl-defstruct`でコンパイルされたコードを新しい Emacs で実行したときに問題が発生するかもしれません。この魔問題を軽減するために、Emacs は古い `cl-defstruct`の使用を検知した際に古い構造体オブジェクトがレコードであるかのように処理するモードを有効にします。

`cl-old-struct-compat-mode` *arg* [Function]

*arg*が正なら旧スタイルの構造体にたいする後方互換を有効にする。

8 ハッシュテーブル

ハッシュテーブル (hash table) は非常に高速なルックアップテーブルの一種で、キーに対応する値をマップするという点では alist(Section 5.8 [Association Lists], page 88 を参照) に似ています。ハッシュテーブルは以下の点で alist と異なります:

- ハッシュテーブルでのルックアップ (lookup: 照合) は、巨大なテーブルにたいして非常に高速である — 実際のところルックアップに必要な時間は、そのテーブルに格納されている要素数とは基本的に無関係である。ハッシュテーブルには一定のオーバーヘッドが多少あるので、小さいテーブル (数十の要素) では alist のほうが高速だろう。
- ハッシュテーブル内の対応関係に特定の順序はない。
- 2 つの alist で共通の末尾 (tail) を共有させるような、2 つのハッシュテーブル間で構造を共有する方法はない。

Emacs Lisp は一般的な用途のハッシュテーブルデータ型とともに、それらを処理する一連の関数を提供します。ハッシュテーブルは '`#s`'、その後にハッシュテーブルのプロパティーと内容を指定するリストが続く、特別なプリント表現をもちます。Section 8.1 [Creating Hash], page 118 を参照してください (ハッシュ表記の最初に使用される '`#`'文字は、読み取り表現をもたないオブジェクトのプリント表現であり、これはハッシュテーブルに何も行わない。Section 2.1 [Printed Representation], page 8 を参照のこと)。

obarray(オブジェクト配列) もハッシュテーブルの一種ですが、これらは異なる型のオブジェクトであり、intern(インターン) されたシンボルを記録するためだけに使用されます (Section 9.3 [Creating Symbols], page 126 を参照)。

8.1 ハッシュテーブルの作成

ハッシュテーブルを作成する基本的な関数は `make-hash-table`です。

`make-hash-table` **&rest** *keyword-args* [Function]

この関数は指定された引数に対応する新しいハッシュテーブルを作成する。引数はキーワード (特別に認識される独自のシンボル) と、それに対応する値を交互に指定することで構成される。

`make-hash-table`ではいくつかのキーワードが意味をもつが、実際に知る必要があるのは `:test`と`:weakness`の 2 つだけである。

`:test` *test*

これはそのハッシュテーブルにたいしてキーを照合する方法を指定する。デフォルトは `eql`であり他の代替としては `eq`や `equal`がある:

`eql` キーが数字ならそれらが `equal`、つまりそれらの値が等しくどちらも整数か浮動少数点数なら同一。それ以外なら別の 2 つのオブジェクトは決して同一とならない。

`eq` 別の 2 つの Lisp オブジェクトはすべて別のキーになる。

`equal` 別の 2 つの Lisp オブジェクトで、それらが `equal`なら同一のキーである。

*test*にたいして追加の選択肢を定義するために、`define-hash-table-test` (Section 8.3 [Defining Hash], page 121 を参照) を使用することができる。

`:weakness` *weak*

ハッシュテーブルの weakness(強度) は、ハッシュテーブル内に存在するキーと値をガーベージコレクションから保護するかどうかを指定する。

値 *weak*には `nil`、`key`、`value`、`key-or-value`、`key-and-value`、または `t`(`key-and-value`のエイリアス) のいずれかを指定しなければならない。*weak* が `key`ならそのハッシュテーブルは、(キーが他の場所で参照されていなければ) ハッシュテーブルのキーがガーベージコレクトされるのを妨げられない。ある特定のキーがガーベージコレクトされると、それに対応する連想はハッシュテーブルから削除される。

*weak*が `value`ならそのハッシュテーブルは、(値が他の場所で参照されていなければ) ハッシュテーブルの値がガベージコレクトされるのを妨げませんられない。ある特定の値がガーベージコレクトされると、それに対応する連想はハッシュテーブルから削除される。

*weak*が `key-and-value`(か `t`) なら、その連想を保護するためにはキーと値の両方が生きていなければならない。したがってそのハッシュテーブルは、キーと値の一方だけをガーベージコレクトから守ることはしない。キーか値のどちらか一方がガーベージコレクトされたら、その連想は削除される。

*weak*が `key-or-value`nara、キーか値のどちらか一方で、その連想を保護することができる。したがってキーと値の両方がガベージコレクトされたときだけ (それがハッシュテーブル自体にたいする参照でなければ)、ハッシュテーブルからその連想が削除される。

*weak*のデフォルトは `nil`なので、ハッシュテーブルから参照されているキーと値はすべてガーベージコレクションから保護される。

`:size` *size*

これはそのハッシュテーブルに保管しようとしている、連想の数にたいするヒントを指定する。数が概算で判っていれば、この方法でそれを指定して処理を若干効率的にすることができる。小さすぎるサイズを指定すると、そのハッシュテーブルは必要に応じて自動的に拡張されるが、これを行なうために時間が余計にかかる。

デフォルトのサイズは 65。

`:rehash-size` *rehash-size*

ハッシュテーブルに連想を追加するとき、そのテーブルが満杯ならテーブルを自動的に拡張する。この値はその際にどれだけハッシュテーブルを拡張するかを指定する。

*rehash-size*が整数 (正であること) なら、通常のサイズに *rehash-size*に近い値を加えることによりハッシュテーブルが拡張される。*rehash-size*が浮動小数 (1 より大きい方がよい) なら、古いサイズにその数に近い値を乗じることによりハッシュテーブルが拡張される。

デフォルト値は 1.5。

`:rehash-threshold` *threshold*

これはハッシュテーブルが一杯 (なのでもっと大きく拡張する必要がある) だと判断される基準を指定する。*threshold*の値は 1 以下の正の浮動小数点数であること。実際のエントリー数が通常のサイズにたいする指定した割合に近い値を超えるとハッシュテーブルは一杯 (full) になる。*threshold*のデフォルトは 0.8125。

ハッシュテーブルのプリント表現を使用して、新しいハッシュテーブルを作成することもできます。指定されたハッシュテーブル内の各要素が、有効な入力構文 (Section 2.1 [Printed Representation], page 8 を参照) をもっていれば、Lisp リーダーはこのプリント表現を読み取ることができます。たとえば以下は値 `val1`(シンボル) と `300`(数字) に関連づけられた、キー `key1`と `key2`(両方ともシンボル) を、新しいハッシュテーブルに指定します。

```
#s(hash-table size 30 data (key1 val1 key2 300))
```

ハッシュテーブルのプリント表現は '`#s`'と、その後の '`hash-table`'で始まるリストにより構成されます。このリストの残りの部分はそのハッシュテーブルのプロパティーと初期内容を指定する、0 個以上のプロパティーと値からなるペアで構成されるべきです。プロパティーと値はそのまま読み取られます。有効なプロパティー名は `size`、`test`、`weakness`、`rehash-size`、`rehash-threshold`、`data`です。`data`プロパティーは、初期内容にたいするキーと値のペアからなるリストであるべきです。他のプロパティーは、上記で説明した `make-hash-table`のキーワード (`:size`、`:test`など) と同じ意味をもちます。

バッファーやフレームのような、入力構文をもたないオブジェクトを含んだ初期内容をもつハッシュテーブルを指定できないことに注意してください。そのようなオブジェクトは、ハッシュテーブルを作成した後に追加します。

8.2 ハッシュテーブルへのアクセス

このセクションではハッシュテーブルにアクセスしたり、連想を保管する関数を説明します。比較方法による制限がない限り、一般的には任意の Lisp オブジェクトをハッシュキーとして使用できます。

`gethash` *key table* **&optional** *default* [Function]
この関数は *table*の *key*を照合してそれに関連づけられた *value*、*table*内に *key*をもつ連想が存在しなければ *default*をリターンする。

`puthash` *key value table* [Function]
この関数は *table*内に値 *value*をもつ *key*の連想を挿入します。*table*がすでに *key*の連想をもつなら、*value*で古い連想値を置き換える。

`remhash` *key table* [Function]
この関数は *table*に *key*の連想があればそれを削除する。*key*が連想をもたなければ `remhash`は何も行なわない。

Common Lisp に関する注意**:** Common Lisp では `remhash`が実際に連想を削除したときは非 `nil`、それ以外は `nil`をリターンする。Emacs Lisp では `remhash`は常に `nil`をリターンする。

`clrhash` *table* [Function]
この関数はハッシュテーブル *table*からすべての連想を削除するので、そのハッシュテーブルは空になる。これはハッシュテーブルのクリーニング *(clearing)* とも呼ばれる。

Common Lisp に関する注意**:** Common Lisp では `clrhash`は空の *table*をリターンする。Emacs Lisp では `nil`をリターンする。

`maphash` *function table* [Function]
この関数は *table*内の各連想にたいして一度ずつ *function*を呼び出す。関数 *function*は 2 つの引数 — *table*にリストされた *key*と、それに関連づけられた *value* — を受け取ること。`maphash`は `nil`をリターンする。

8.3 ハッシュの比較の定義

`define-hash-table-test`でキーを照合する新しい方法を定義できます。この機能を使用するにはハッシュテーブルの動作方法と、ハッシュコード *(hash code)* の意味を理解する必要があります。

概念的にはハッシュテーブルを 1 つの連想を保持できるスロットがたくさんある巨大な配列として考えることができます。キーを照合するにはまず、`gethash`がキーから整数のハッシュコードを計算します。配列の長さを modulo としてこの整数を減らして、配列内のインデックスを生成することができます。それからキーが見つかったかどうか確認するためにそのスロット、もし必要なら近くのスロットを探します。

しあがってキーを照合する新たな方法を定義するにはキーからハッシュコードを計算する関数、および 2 つのキーを直接比較する関数の両方を指定する必要があります。この 2 つの関数は互いに一貫性をもつ必要があります。すなわちキーを比較して equal なら、2 つのキーのハッシュコードは同一であるべきです。さらに (ガーベージコレクターからの呼び出しのように)2 つの関数は任意のタイミングで呼び出される可能性があるので、関数が副作用をもたないこと、すぐにリターンすること、そしてこれらの関数の挙動はそのキーの不変の性質だけに依存する必要があります。

`define-hash-table-test` *name test-fn hash-fn* [Function]

この関数は *name*という名前の新たなハッシュテーブルテストを定義します。

この方法で *name*を定義した後は、`make-hash-table`の引数 *test*にこれを使用することができる。これを行なう際は、そのハッシュテーブルのキー値の比較に *test-fn*、キー値からハッシュコードを計算するために *hash-fn*を使用することになる。

関数 *test-fn*は 2 つの引数 (2 つのキー) をとり、それらが同一と判断されたときは非`nil`をリターンする。

関数 *hash-fn*は 1 つの引数 (キー) を受け取り、そのキーのハッシュコード (整数) をリターンすること。よい結果を得るために、その関数は負の fixnum を含む fixnum の全範囲をハッシュコードに使用すること。

指定された関数は、プロパティー`hash-table-test`の配下の、*name*というプロパティーリストに格納される。そのプロパティーの値形式は (*`test-fn hash-fn`*)。

`sxhash-equal` *obj* [Function]

この関数は Lisp オブジェクト *obj*のハッシュコードをリターンする。リターン値は *obj*と、それが指す別の Lisp オブジェクトの内容を表す整数。

2 つのオブジェクト *obj1*と *obj2*が等しければ (`sxhash-equal` *`obj1`*)と (`sxhash-equal` *`obj2`*)は同じ整数になる。

2 つのオブジェクトが`equal`でなければ通常は`sxhash-equal`が return する値は異なるが、常に異なるとは限らない。稀 (運次第) に`sxhash-equal`が同じ結果を与える、2 つの異なって見えるオブジェクトに遭遇するかもしれない。

Common Lisp に関する注意**:** Common Lisp ではこれに似た関数は`sxhash`と呼ばれる。Emacs は互換性のために`sxhash-equal`にたいするエイリアスとしてこの名前を提供している。

`sxhash-eq` *obj* [Function]

この関数は Lisp オブジェクト *obj*にたいするハッシュコードをリターンする。結果は *obj*の識別値でありコンテンツではない。

2 つのオブジェクト *obj1*と *obj2*が`eq`なら (`sxhash-eq` *`obj1`*)と (`sxhash-eq` *`obj2`*)は同じ整数。

`sxhash-eql` *obj* [Function]
この関数は `eql`による比較に適した Lisp オブジェクト *obj*にたいするハッシュコードをリターンする。つまり浮動小数点数と bignum 以外の *obj*なら、それにたいする識別値 (浮動小数点数ならその値にたいするハッシュコード) を生成する。

2 つのオブジェクト *obj1*と *obj2*が `eql`なら `(sxhash-eql obj1)`と `(sxhash-eql obj2)`は同じ整数。

以下は lcase を区別しない文字列のキーをもつハッシュテーブルを作成する例です。

```
(defun case-fold-string= (a b)
  (eq t (compare-strings a nil nil b nil nil t)))
(defun case-fold-string-hash (a)
  (sxhash-equal (upcase a)))

(define-hash-table-test 'case-fold
  'case-fold-string= 'case-fold-string-hash)

(make-hash-table :test 'case-fold)
```

以下は事前に定義されたテスト値 `equal`と等価なテストを行なうハッシュテーブルを定義できるという例です。キーは任意の Lisp オブジェクトで、equal に見えるオブジェクトは同じキーと判断されます。

```
(define-hash-table-test 'contents-hash 'equal 'sxhash-equal)

(make-hash-table :test 'contents-hash)
```

ハッシュ関数の実装はセッション中に変更されたり、アークテクチャーごとに異なるかもしれなオブジェクトストレージのいくつかの詳細を使用するので、Lisp プログラムは Emacs セッションの間はハッシュコードが保存されることに依存するべきではありません。

8.4 ハッシュテーブルのためのその他関数

以下はハッシュテーブルに作用する他の関数です。

`hash-table-p` *table* [Function]
この関数は *table*がハッシュテーブルオブジェクトなら非 `nil`をリターンする。

`copy-hash-table` *table* [Function]
この関数は *table*のコピーを作成してリターンする。そのテーブル自体がコピーされたものである場合のみ、キーと値が共有される。

`hash-table-count` *table* [Function]
この関数は *table*内の実際のエントリー数をリターンする。

`hash-table-test` *table* [Function]
この関数はハッシュを行なう方法と、キーを比較する方法を指定するために、*table*作成時に与えられた *test*の値をリターンする。Section 8.1 [Creating Hash], page 118 の `make-hash-table`を参照されたい。

`hash-table-weakness` *table* [Function]
この関数はハッシュテーブル *table*に指定された *weak*の値をリターンする。

`hash-table-rehash-size` *table* [Function]
この関数は *table*の rehash-size をリターンする。

`hash-table-rehash-threshold` *table* [Function]
この関数は *table*の rehash-threshold をリターンする。

`hash-table-size` *table* [Function]
この関数は *table*の現在の定義されたサイズをリターンする。

9 シンボル

シンボル *(symbol)* は一意な名前をもつオブジェクトです。このチャプターではシンボル、シンボルの構成要素とプロパティーリスト、およびシンボルの作成とインターンする方法を説明します。別のチャプターではシンボルを変数として使用したり、関数名として使用する方法が説明されています。Chapter 12 [Variables], page 174 と Chapter 13 [Functions], page 209 を参照してください。シンボルの正確な入力構文については、Section 2.4.4 [Symbol Type], page 14 を参照してください。

`symbolp`を使用して、任意の Lisp オブジェクトがシンボルかどうかをテストできます:

`symbolp` *object* [Function]
: この関数は *object*がシンボルなら `t`、それ以外は `nil`をリターンする。

9.1 シンボルの構成要素

各シンボルは 4 つの構成要素 (もしくは “セル”) をもち、構成要素はそれぞれ別のオブジェクトを参照します:

プリント名 (print name)
: そのシンボルの名前。

値 (value)　変数としてのそのシンボルの現在値。

関数 (function)
: そのシンボルの関数定義。シンボル、キーマップ、キーボードマクロも保持できる。

プロパティーリスト (property list)
: そのシンボルのプロパティーリスト。

プリント名のセルは常に文字列を保持し、それを変更することはできません。他の 3 つのセルには、任意の Lisp オブジェクトをセットすることができます。

プリント名のセルはシンボルの名前となる文字列を保持します。シンボルはシンボル名によりテキストとして表されるので、2 つのシンボルが同じ名前をもたないことが重要です。Lisp リーダーはシンボルを読み取るごとに、それを新規作成する前に、指定されたシンボルがすでに存在するかを調べます。シンボルの名前を得るには関数 `symbol-name`(Section 9.3 [Creating Symbols], page 126 を参照) を使用します。

値セルは変数としてのシンボルの値 (そのシンボル自身が Lisp 式として評価されたときに得る値) を保持します。ローカルバインディング *(local binding)* やスコーピングルール *(scoping rules)* 等のような複雑なものを含めて、変数のセットや取得方法については Chapter 12 [Variables], page 174 を参照してください。ほとんどのシンボルは値として任意の Lisp オブジェクトをもつことができますが、一部の特別なシンボルは変更できない値をもちます。これらには `nil`、`t`、および名前が ‘`:`’で始まるすべてのシンボル (キーワード *(keyword)* と呼ばれる) が含まれます。Section 12.2 [Constant Variables], page 174 を参照してください。

関数セルはシンボルの関数定義を保持します。実際はには `foo`の関数セルの中に保管されている関数を意味するときに、“関数 `foo`” といってそれを参照することがよくあります。わたしたちは必要なときだけ、これを明確に区別することにします。関数セルは通常は関数 (Chapter 13 [Functions], page 209 を参照) か、マクロ (Chapter 14 [Macros], page 242 を参照) を保持するために使用されます。しかし関数セルはシンボル (Section 10.1.4 [Function Indirection], page 135 を参照)、キーボードマクロ (Section 21.16 [Keyboard Macros], page 420 を参照)、キーマップ (Chapter 22 [Keymaps], page 422 を参照)、またはオートロードオブジェクト (Section 10.1.8 [Autoloading],

page 138 を参照) を保持するためにも使用できます。シンボルの関数セルの内容を得るには、関数 `symbol-function` (Section 13.9 [Function Cells], page 225 を参照) を使用します。

プロパティーリストのセルは、通常は正しくフォーマットされたプロパティーリストを保持するべきです。シンボルのプロパティーリストを得るには関数 `symbol-plist`を使用します。Section 9.4 [Symbol Properties], page 129 を参照してください。

マクロセルと値セルが *void(空)* のときもあります。void とはそのセルがどのオブジェクトも参照していないことを意味します (これはシンボル `void`を保持するのともシンボル `nil`を保持するのとも異なる)。void の関数セルまたは値セルを調べようとすると結果は '`Symbol's value as variable is void`'のようなエラーとなります。

各シンボルは値セルと関数セルを別個にもつので、変数名と関数名が衝突することはありません。たとえばシンボル `buffer-file-name`が値 (カレントバッファーで visit されているファイルの名前) をもつと同様に、関数定義 (ファイルの名前をリターンするプリミティブ関数) をもつことができます:

```
buffer-file-name
     ⇒ "/gnu/elisp/symbols-ja.texi"
(symbol-function 'buffer-file-name)
     ⇒ #<subr buffer-file-name>
```

9.2 シンボルの定義

定義 *(definition)* とは、特別な方法での使用の意図を宣言する特別な種類の Lisp 式です。定義とは通常はシンボルにたいする値を指定するか、シンボルにたいする 1 つの種類の使用についての意味とその方法で使用する際のシンボルの意味のドキュメントを指定します。したがってシンボルを変数として定義すると、その変数の初期値に加えてその変数のドキュメントを提供できます。

`defvar`と `defconst`はグローバル変数 *(global variable)* — Lisp プログラムの任意の箇所からアクセスできる変数 — として定義するためのスペシャルフォームです。変数についての詳細は Chapter 12 [Variables], page 174 を参照してください。カスタマイズ可能な変数を定義するには `defcustom` (サブルーチンとして `defvar`も呼び出す) を使用します (Chapter 15 [Customization], page 250 を参照)。

最初にシンボルが変数として定義されているかどうかに関わらず、原則として `setq`で任意のシンボルに値を割り当てることができます。しかし使用したいグローバル変数それぞれにたいして変数定義を記述するべきです。さもないとレキシカルスコープ (Section 12.10 [Variable Scoping], page 185 を参照) が有効なときに変数が評価されると、Lisp プログラムが正しく動作しないかもしれません。

`defun`はラムダ式 (lambda expression) を生成して、そのシンボルの関数セルに格納することにより、そのシンボルを関数として定義します。したがってこのシンボルの関数定義は、そのラムダ式になります (関数セルの内容を意味する用語 "関数定義 (function definition)" は、`defun`がシンボルに関数としての定義を与えるというアイデアに由来する)。Chapter 13 [Functions], page 209 を参照してください。

`defmacro`はシンボルをマクロとして定義します。これはマクロオブジェクトを作成してシンボルの関数セルにそれを格納します。シンボルにはマクロと関数を与えることができますが、マクロと関数定義はどちらも関数セルに保持されるのにたいし、関数セルに保持できるのは常にただ 1 つの Lisp オブジェクトなので、一度に両方を行なうことはできないことに注意してください。Chapter 14 [Macros], page 242 を参照してください。

前に注記したように Emacs Lisp ではシンボルを (たとえば `defvar`で) 変数として定義して、同じシンボルを (たとえば `defun`で) 関数やマクロとして両方定義することができます。このような定義は衝突しません。

これらの定義は、プログラミングツールのガイドを果たすこともできます。たとえば、*C-h f*および *C-h v*コマンドは、関係ある変数、関数、マクロ定義へのリンクを含むヘルプバッファーを作成します。Section “Name Help” in *The GNU Emacs Manual* を参照してください。

9.3 シンボルの作成と intern

GNU Emacs Lisp でシンボルが作成される方法を理解するには、Lisp がシンボルを読み取る方法を理解しなければなりません。Lisp は、同じ文字綴りを読み取ったら、毎回同じシンボルを見つけることを保証しなければなりません。これに失敗すると、完全な混乱を招くでしょう。

Lisp リーダーがシンボルに出会うと、Lisp リーダーは名前のすべての文字を読み取ります。その後 Lisp リーダーは *obarray(*オブジェクト配列*)* と呼ばれるテーブル内のインデックスを決めるために、これらの文字をハッシュ(hash) します。ハッシュ化 (hashing) は何かを照合するのに効果的な方法です。たとえば Jan Jones を見つけるときは、電話帳を表紙から 1 頁ずつ探すのではなく J の頁から探し始めます。これはハッシュ化の簡単なバージョンです。obarray の各要素は与えられたハッシュコードとともに、すべてのシンボルを保持するバケット *(bucket)* です。与えられた名前を探すためには、バケットの中からハッシュコードがその名前であるような、すべてのシンボルを探すのが効果的です (同じアイデアは一般的な Emacs のハッシュテーブルでも使用されていがこれらはデータ型が異なる。Chapter 8 [Hash Tables], page 118 を参照されたい)。

探している名前のシンボルが見つかったら、リーダーはそのシンボルを使用します。obarray にその名前のシンボルが含まれなければ、リーダーは新しいシンボルを作成してそれを obarray に追加します。特定の名前のシンボルを探して追加することをインターン *(intern)* と言い、これが行なわれた後はそのシンボルはインターンされたシンボル *(interned symbol)* と呼ばれます。

インターンすることによりある特定の名前のシンボルは、各 obarray に 1 つだけであることが保証されます。同じ名前のシンボルが他に存在するかもしれませんが、同じ obarray には存在しません。したがってリーダーは、(同じ obarray を読みつづける限り) 同じ名前にたいして同じシンボルを取得します。

インターンは通常はリーダー内で自動的に発生しますが、他のプログラムがこれを行なう必要がある場合もあります。たとえば *M-x*コマンドはその後にミニバッファーを使用してコマンド名を文字列として取得して、その文字列をインターンしてからインターンされたその名前のシンボルを得ます。

すべてのシンボルを含む obarray はありません。実際にどの obarray にも含まれないシンボルがいくつかあります。これらはインターンされていないシンボル *(uninterned symbols)* と呼ばれます。インターンされていないシンボルも、他のシンボルと同じく 4 つのセルをもちます。しかしインターンされていないシンボルへのアクセスを得る唯一の方法は、他の何らかのオブジェクトとして探すか、変数の値として探す方法だけです。

インターンされていないシンボルの作成は、Lisp コードを生成するとき有用です。なぜなら作成されたコード内で変数として使用されているインターンされていないシンボルは、他の Lisp プログラムで使用されている任意の変数と競合することはありえないからです。

Emacs Lisp では obarray はベクターです。ベクター内の各要素がバケットになります。要素の値は、名前がそのバケットにハッシュされるようなインターンされたシンボル、またはバケットが空のときは 0 です。インターンされたシンボルは、そのバケット内の次のシンボルへの内部リンク (ユーザーからは見えない) をもちます。これらのリンクは不可視なので、`mapatoms` (以下参照) を使用する方法をのぞき、obarray 内のすべてのシンボルを探す方法はありません。バケット内のシンボルの順番に意味はありません。

空の obarray ではすべての要素が 0 なので、`(make-vector` *length* `0)`で obarray を作成することができます。**obarray** を作成する有効な方法はこれだけです。長さに素数を指定するとよいハッシュ化がされる傾向があります。2 の累乗から 1 減じた長さもよい結果を生む傾向があります。

自分で **obarray** にシンボルを置かないでください。これはうまくいきません — obarray に正しくシンボルを入力できるのは `intern`だけです。

Common Lisp に関する注意**:** Common Lisp とは異なり Emacs Lisp は 1 つのシンボルを複数の obarray にインターンする方法を提供しない。

以下の関数のほとんどは、引数に名前と obarray をとります。名前が文字列以外、または obarray がベクター以外なら `wrong-type-argument`エラーがシグナルされます。

`symbol-name` *symbol* [Function]

この関数は *symbol*の名前を文字列としてリターンする。たとえば:

```
(symbol-name 'foo)
     ⇒ "foo"
```

警告**:** 文字の置き換えにより文字列を変更すると、それはシンボルの名前を変更しますが、obarray の更新には失敗するので行なわないこと!

`make-symbol` *name* [Function]

この関数は新たに割り当てられた、名前が *name*(文字列でなければならない) であるような、インターンされていないシンボルをリターンする。このシンボルの値と関数は void で、プロパティーリストは `nil`。以下の例では `sym`の値は `foo`と `eq`ではない。なぜならこれは名前が ‘`foo`’という、インターンされていないシンボルだからである。

```
(setq sym (make-symbol "foo"))
     ⇒ foo
(eq sym 'foo)
     ⇒ nil
```

`gensym` **&optional** *prefix* [Function]

この関数は `make-symbol`を使用して *prefix*に `gensym-counter`を付加した名前のシンボルをリターンする。プレフィックスのデフォルトは`"g"`。

`intern` *name* **&optional** *obarray* [Function]

この関数は名前が *name*であるような、インターンされたシンボルをリターンする。オブジェクト配列 *obarray*の中にそのようなシンボルが存在しなければ、`intern`は新たにシンボルを作成して obarray に追加してそれをリターンする。*obarray*が省略されると、グローバル変数 `obarray`の値が使用される。

```
(setq sym (intern "foo"))
     ⇒ foo
(eq sym 'foo)
     ⇒ t

(setq sym1 (intern "foo" other-obarray))
     ⇒ foo
(eq sym1 'foo)
     ⇒ nil
```

Common Lisp に関する注意**:** Common Lisp では既存のシンボルを obarray にインターンできる。Emacs Lisp では `intern`の引数はシンボルではなく文字列なのでこれを行なうことはできない。

`intern-soft` *name* **&optional** *obarray* [Function]

この関数は *obarray*内の名前が *name*のシンボル、*obarray*にその名前のシンボルが存在しなければ `nil`をリターンする。したがって与えられた名前のシンボルがすでにインターンされているかテストするために、`intern-soft`を使用することができる。*obarray*が省略されるとグローバル変数 `obarray`の値が使用される。

引数 *name*にはシンボルも使用できる。この場合、指定された obarray に *name*がインターンされていれば *name*、それ以外なら `nil`をリターンする。

```
(intern-soft "frazzle")         ; そのようなシンボルは存在しない
     ⇒ nil
(make-symbol "frazzle")         ; インターンされていないシンボルを作成する
     ⇒ frazzle
(intern-soft "frazzle")         ; そのようなシンボルは見つからない
     ⇒ nil
(setq sym (intern "frazzle"))   ; インターンされたシンボルを作成する
     ⇒ frazzle
(intern-soft "frazzle")         ; シンボルが見つかった!
     ⇒ frazzle
(eq sym 'frazzle)               ; そしてそれは同じシンボル
     ⇒ t
```

`obarray` [Variable]

この変数は `intern`と `read`が使用する標準の obarray である。

`mapatoms` *function* **&optional** *obarray* [Function]

この関数はオブジェクト配列 *obarray*の中の各シンボルにたいして、*function*を一度呼び出しその後 `nil`をリターンする。*obarray*が省略されると、通常のシンボルにたいする標準のオブジェクト配列 `obarray`の値がデフォルトになる。

```
(setq count 0)
     ⇒ 0
(defun count-syms (s)
  (setq count (1+ count)))
     ⇒ count-syms
(mapatoms 'count-syms)
     ⇒ nil
count
     ⇒ 1871
```

`mapatoms`を使用する他の例については、Section 24.2 [Accessing Documentation], page 520 の `documentation`を参照のこと。

`unintern` *symbol obarray* [Function]

この関数はオブジェクト配列 *obarray*から *symbol*を削除する。obarray の中に `symbol`が存在ければ、`unintern`は何も行なわない。*obarray*が `nil`なら現在の obarray が使用される。

*symbol*にシンボルではなく文字列を与えると、それはシンボルの名前を意味する。この場合、`unintern`は (もしあれば)obarray からその名前のシンボルを削除する。そのようなシンボルが存在するなら `unintern`は何も行なわない。

`unintern`がシンボルを削除したら `t`、それ以外は `nil`をリターンする。

9.4 シンボルのプロパティ

シンボルはそのシンボルについての様々な情報を記録するために使用される、任意の数のシンボルプロパティー *(symbol properties)* をもつことができます。たとえばシンボルの`risky-local-variable`プロパティーが`nil`なら、その変数の名前が危険なファイルローカル変数 (Section 12.12 [File Local Variables], page 197 を参照) であることを意味します。

シンボルのプロパティーとプロパティー値はそれぞれ、シンボルのプロパティーリストセル (Section 9.1 [Symbol Components], page 124 を参照) に、プロパティーリスト形式 (Section 5.9 [Property Lists], page 92 を参照) で格納されます。

9.4.1 シンボルのプロパティへのアクセス

以下の関数を使用してシンボルプロパティーにアクセスできます。

`get` *symbol property* [Function]

この関数は *symbol*のプロパティーリスト内の、名前が *property*というプロパティーの値をリターンする。そのようなプロパティーが存在しなければ`nil`をリターンする。したがって値が`nil`のときとプロパティーが存在しないときの違いはない。

名前 *property*は`eq`を使用して既存のプロパティーと比較されるので、すべてのオブジェクトがプロパティーとして適正である。

`put`の例を参照のこと。

`put` *symbol property value* [Function]

この関数は *symbol*のプロパティーリストの、プロパティー名 *property*に *value*を put して、前のプロパティー値を置き換える。`put`関数は *value*をリターンする。

```
(put 'fly 'verb 'transitive)
     ⇒'transitive
(put 'fly 'noun '(a buzzing little bug))
     ⇒ (a buzzing little bug)
(get 'fly 'verb)
     ⇒ transitive
(symbol-plist 'fly)
     ⇒ (verb transitive noun (a buzzing little bug))
```

`symbol-plist` *symbol* [Function]

この関数は *symbol*のプロパティーリストをリターンする。

`setplist` *symbol plist* [Function]

この関数は *symbol*のプロパティーリストを *plist*にセットする。*plist*は通常は適正なプロパティーリストであるべきだが、これは強制ではない。リターン値は *plist*です。

```
(setplist 'foo '(a 1 b (2 3) c nil))
     ⇒ (a 1 b (2 3) c nil)
(symbol-plist 'foo)
     ⇒ (a 1 b (2 3) c nil)
```

通常の用途には使用されない特別な obarray 内のシンボルでは、非標準的な方法でプロパティーリストセルを使用することに意味があるかもしれない。実際に abbrev(Chapter 36 [Abbrevs], page 922 を参照) のメカニズムでこれを行なっている。

以下のように setplistと plist-putで putを定義できる:

```
(defun put (symbol prop value)
  (setplist symbol
            (plist-put (symbol-plist symbol) prop value)))
```

`function-get` *symbol property* **&optional** *autoload* [Function]
この関数は getと等価だが *symbol*が関数のエイリアス名なら。実際の関数を命名するシンボルのプロパティリストを照合する点が異なる。Section 13.4 [Defining Functions], page 215 を参照のこと。オプション引数 *autoload*が非 nilで、*symbol*が自動ロードされていれば、その自動ロードにより *symbol*の *property*がセットされるかもしれないので、この関数はそれの自動ロードを試みるだろう。*autoload*がシンボル macroなら、*symbol*が自動ロードされたマクロのときだけ自動ロードを試みる。

`function-put` *function property value* [Function]
この関数は *function*の *property*に *value*をセットする。*function*はシンボルであること。関数のプロパティのセットには、putよりこの関数を呼び出すほうがよい。この関数を使用すれば、いつか古いプロパティから新しいプロパティへのリマップを実装することができるからである。

9.4.2 シンボルの標準的なプロパティ

Emacs で特別な目的のために使用されるシンボルプロパティーを以下に一覧します。以下のテーブルで、“命名される関数 (the named function)” と言うときは、関数名がそのシンボルであるような関数を意味します。“命名される変数 (the named variable)” 等の場合も同様です。

`:advertised-binding`
このプロパティーリストは、命名される関数のドキュメントを表示する際に優先されるキーバインディングを指定する。Section 24.3 [Keys in Documentation], page 522 を参照のこと。

`char-table-extra-slots`
値が非 nilなら、それは命名される文字テーブル型の追加スロットの数を指定する。Section 6.6 [Char-Tables], page 110 を参照のこと。

`customized-face`
`face-defface-spec`
`saved-face`
`theme-face`
これらのプロパティーはフェイスの標準のフェイス仕様 (face specs) と、フォント仕様の saved-fase、customized-face、themed-face を記録するために使用される。これらのプロパティーを直接セットしないこと。これらのプロパティーは defaceと関連する関数により管理される。Section 39.12.2 [Defining Faces], page 1015 を参照のこと。

`customized-value`
`saved-value`
`standard-value`
`theme-value`
これらのプロパティーは、カスタマイズ可能な変数の standard-value、saved-value、customized-value(しかし保存はされない)、themed-value を記録するために使用される。これらのプロパティーを直接セットしないこと。これらは defcustomと関連する関数により管理される。Section 15.3 [Variable Definitions], page 253 を参照のこと。

`disabled` 値が非`nil`なら命名される関数はコマンドとして無効になる。Section 21.14 [Disabling Commands], page 418 を参照のこと。

`face-documentation`
値には命名されるフェイスのドキュメント文字列が格納される。これは`defface`により自動的にセットされる。Section 39.12.2 [Defining Faces], page 1015 を参照のこと。

`history-length`
値が非`nil`なら、命名されるヒストリーリスト変数のミニバッファーヒストリーの最大長を指定する。Section 20.4 [Minibuffer History], page 348 を参照のこと。

`interactive-form`
この値は命名される関数のインタラクティブ形式である。通常はこれを直接セットするべきではない。かわりにスペシャルフォーム`interactive`を使用すること。Section 21.3 [Interactive Call], page 382 を参照されたい。

`menu-enable`
この値は命名されるメニューアイテムが、メニュー内で有効であるべきか否かを決定するための式である。Section 22.17.1.1 [Simple Menu Items], page 448 を参照のこと。

`mode-class`
値が`special`なら命名されるメジャーモードは special(特別) である。Section 23.2.1 [Major Mode Conventions], page 464 を参照のこと。

`permanent-local`
値が非`nil`なら命名される変数はバッファーローカル変数となり、メジャーモードの変更によって変数の値はリセットされない。Section 12.11.2 [Creating Buffer-Local], page 192 を参照のこと。

`permanent-local-hook`
値が非`nil`なら、命名される関数はメジャーモード変更時にフック変数のローカル値から削除されない。Section 23.1.2 [Setting Hooks], page 462 を参照のこと。

`pure` 値が非`nil`の場合には、名づけられた関数は純粋 (pure) だとみなされる。定数の引数で呼び出された場合には、コンパイル時に評価することができる。これは実行時のエラーをコンパイル時へとシフトする。純粋ストレージ (pure storage) と混同しないこと (Section E.2 [Pure Storage], page 1177 を参照)。

`risky-local-variable`
値が非`nil`なら、命名される変数はファイルローカル変数としては危険だとみなされる。Section 12.12 [File Local Variables], page 197 を参照のこと。

`safe-function`
値が非`nil`なら、命名される関数は評価において一般的に安全だとみなされます。Section 13.16 [Function Safety], page 239 を参照のこと。

`safe-local-eval-function`
値が非`nil`なら、命名される関数はファイルローカルの評価フォーム内で安全に呼び出すことができる。Section 12.12 [File Local Variables], page 197 を参照のこと。

`safe-local-variable`
値は命名される変数の、安全なファイルローカル値を決定する関数を指定する。Section 12.12 [File Local Variables], page 197 を参照のこと。

`side-effect-free`
: 非`nil`値は命名される関数は副作用 (Section 13.1 [What Is a Function], page 209 を参照) をもたないので、バイトコンパイラーは値が使用されない呼び出しを無視する。このプロパティの値が`error-free`なら、バイトコンパイラーはそのような呼び出しの削除すら行うかもしれない。バイトコンパイラーの最適化に加えて、このプロパティは関数の安全性を判断するためにも使用される (Section 13.16 [Function Safety], page 239 を参照)。

`undo-inhibit-region`
: 非`nil`の場合には、命名される関数の直後に`undo`が呼び出されると、`undo`操作をアクティブなリージョンに限定することを抑止する。Section 32.9 [Undo], page 793 を参照のこと。

`variable-documentation`
: 非`nil`なら、それは命名される変数のドキュメント文字列を指定する。ドキュメント文字列は`defvar`と関連する関数により自動的にセットされる。Section 39.12.2 [Defining Faces], page 1015 を参照のこと。

10 評価

Emacs Lisp での式の評価 *(evaluation)* は、*Lisp* インタープリター — 入力として Lisp オブジェクトを受け取り、それの式としての値 *(value as an expression)* を計算する — により処理されます。評価を行なう方法はそのオブジェクトのデータ型に依存していて、それはこのチャプターで説明するルールにより行なわれます。インタープリターはプログラムの一部を評価するために自動的に実行されますが、Lisp プリミティブ関数の `eval`を通じて明示的に呼び出すこともできます。

評価を意図した Lisp オブジェクトはフォーム *(form)*、または式 *(expression)* と呼ばれます [1]。フォームはデータオブジェクトであって単なるテキストではないという事実は、Lisp 風の言語と通常のプログラミング言語との間にある基本的な相違点の 1 つです。任意のオブジェクトを評価できますが、実際に評価される事が非常に多いのは数字、シンボル、リスト、文字列です。

以降のセクションでは、各種フォームにたいしてそれを評価することが何を意味するかの詳細を説明します。

Lisp フォームを読み取ってそのフォームを評価するのは、非常に一般的なアクティビティーですが、読み取りと評価は別のアクティビティーであって、どちらか一方を単独で処理することができます。読み取っただけでは何も評価されません。読み取りは Lisp オブジェクトのプリント表現をそのオブジェクト自体に変換します。そのオブジェクトが評価されるべきフォームなのか、そのれともまったく違う目的をもつかを指定するのは、`read`の呼び出し元の役目です。Section 19.3 [Input Functions], page 334 を参照してください。

評価とは再帰的な処理であり、あるフォームを評価するとそのフォームの一部が評価されるといったことがよくあります。たとえば `(car x)`のような関数呼び出し *(function call)* のフォームを評価する場合、Emacs は最初にその引数 (サブフォーム `x`) を評価します。引数を評価した後、Emacs はその関数 (`car`) を実行 *(executes)* します。その関数が Lisp で記述されていれば、関数の *body(*本文*)* を評価することによって実行が行なわれます (しかしこの例で使用している `car`は Lisp 関数ではなく C で実装されたプリミティブ関数である)。関数と関数呼び出しについての情報は Chapter 13 [Functions], page 209 を参照してください。

評価は環境 *(environment)* と呼ばれるコンテキストの内部で行なわれます。環境はすべての Lisp 変数 (Chapter 12 [Variables], page 174 を参照) のカレント値とバインディングにより構成されます。[2] フォームが新たなバインディングを作成せずに変数を参照する際、その変数はカレントの環境から与えられる値へと評価されます。フォームの評価は、変数のバインディングによって一時的にその環境を変更することもあります (Section 12.3 [Local Variables], page 175 を参照)。

フォームの評価が永続する変更を行なうこともあります。これらの変更は副作用 *(side effects)* と呼ばれます。副作用を生成するフォームの例は `(setq foo 1)`です。

コマンドキー解釈での評価と混同しないでください。エディターのコマンドループはアクティブなキーマップを使用して、キーボード入力をコマンド (インタラクティブに呼び出すことができる関数) に変換してからそのコマンドを実行するために、`call-interactively`を使用します。そのコマンドが Lisp で記述されていれば、そのコマンドの実行には通常は評価を伴います。しかしこのステップはコマンドキー解釈の一部とは考えません。Chapter 21 [Command Loop], page 375 を参照してください。

---

[1] *S* 式 *(S-expression)*、短くは *sexp* という言葉でも呼ばれることがありますが、わたしたちはこのマニュアル内では通常はこの用語は使用しません。

[2] "環境" にたいするこの定義は、プログラムの結果に影響し得るすべてのデータを特に意図したものではありません。

10.1 フォームの種類

評価される事を意図した Lisp オブジェクトはフォーム *(form)*、または式 *(expression)*) と呼ばれます。Emacs がフォームを評価する方法はフォームのデータ型に依存します。Emacs は 3 種の異なるフォーム— シンボル、リスト、およびその他すべての型 — をもち、それらが評価される方法は異なります。このセクションではまず最初に自己評価フォームのその他の型から開始して、3 つの種類をすべて 1 つずつ説明します。

10.1.1 自己評価を行うフォーム

自己評価フォーム *(self-evaluating form)* はリストやシンボルではないすべてのフォームです。自己評価フォームはそのフォーム自身を評価します。評価の結果は評価されたオブジェクトと同じです。したがって数字の 25 は 25、文字列`"foo"`は文字列`"foo"`に評価されます。同様にベクターの評価では、ベクターの要素の評価は発生しません— 内容が変更されずに同じベクターがリターンされます。

```
'123                ; 評価されずに表示される数字
     ⇒ 123
123                 ; 通常どおり評価され、同じものが return される
     ⇒ 123
(eval '123)         ; 手動での評価 — 同じものが return される
     ⇒ 123
(eval (eval '123)) ; 2 度評価しても何も変わらない。
     ⇒ 123
```

自己評価フォームはプログラムの一部となる値を生成します。これを`setcar`や`aset`、その他の類似操作を通じて変更しようと試みるべきではありません。Lisp インタープリターがプログラム中の自己評価フォームにより生成される定数を統合して、これらの定数が構造を共有するかもしれません。Section 2.9 [Mutability], page 35 を参照してください。

自己評価されるという事実による利点から数字、文字、文字列、そしてベクターでさえ Lisp コード内で記述されるのが一般的です。しかし入力構文がない型にたいしてこれを行なうのは極めて異例です。なぜなら、これらをテキスト的に記述する方法がないからです。Lisp プログラムを使用してこれらの型を含む Lisp 式を構築することは可能です。以下は例です:

```
;; バッファーオブジェクトを含む式を構築する。
(setq print-exp (list 'print (current-buffer)))
     ⇒ (print #<buffer eval-ja.texi>)
;; それを評価する。
(eval print-exp)
     ⊣ #<buffer eval-ja.texi>
     ⇒ #<buffer eval-ja.texi>
```

10.1.2 シンボルのフォーム

シンボルが評価されるときは変数として扱われます。それが値をもつなら結果はその変数の値になります。そのシンボルが変数としての値をもたなければ、Lisp インタープリターはエラーをシグナルします。変数の使用法についての情報は Chapter 12 [Variables], page 174 を参照してください。

以降の例では`setq`でシンボルに値をセットしています。その後シンボルを評価してからを`setq`に戻します。

```
(setq a 123)
     ⇒ 123
```

```
(eval 'a)
     ⇒ 123
a
     ⇒ 123
```

シンボル nilと tは特別に扱われるので、nilの値は常に nil、tの値は常に tになります。これらに他の値をセットしたり、他の値にバインドすることはできません。したがってこの 2 つのシンボルは、(たとえ evalがそれらを他の任意のシンボルと同様に扱うとはいえ) 自己評価フォームと同じように振る舞います。名前が ':'で始まるシンボルも同じ方法で自己評価されます。そして、(通常は) 値を変更できない点も同じです。Section 12.2 [Constant Variables], page 174 を参照してください。

10.1.3 リストフォームの分類

空ではないリストフォームは関数呼び出し、マクロ呼び出し、スペシャルフォームのいずれかで、それは 1 番目の引数にしたがいます。これら 3 種のフォームは、以下で説明するように異なる方法で評価されます。残りの要素は関数、マクロ、またはスペシャルフォームにたいする引数 *(arguments)* を構成します。

空ではないリストを評価する最初のステップは、1 番目の要素の確認です。この要素は単独でそのリストがどの種類のフォームなのかと、残りの引数をどのように処理するがを決定します。Scheme のような Lisp 方言とは異なり、1 番目の要素は評価されません。

10.1.4 シンボル関数インダイレクション

リストの最初の要素がシンボルなら、評価はそのシンボルの関数セルを調べて、元のシンボルの代わりに関数セルの内容を使用します。その内容が他のシンボルなら、シンボルではないものが得られるまでこのプロセスが繰り返されます。このプロセスのことをシンボル関数インダイレクション *(symbol function indirection: indirection* は間接の意*)* と呼びます。シンボル関数インダイレクションについての情報は Section 13.3 [Function Names], page 214 を参照してください。

このプロセスの結果、シンボルの関数セルが同じシンボルを参照する場合には、無限ループを起こす可能性があります。それ以外なら最終的には非シンボルにたどりつき、それは関数か他の適切なオブジェクトである必要があります。

適切なオブジェクトとは、より正確には Lisp 関数 (ラムダ式)、バイトコード関数、プリミティブ関数、Lisp マクロ、スペシャルフォーム、またはオートロードオブジェクトです。これらそれぞれの型については以降のセクションで説明します。これらの型以外のオブジェクトなら Emacs は invalid-functionエラーをシグナルします。

以下の例はシンボルインダイレクションのプロセスを説明するものです。わたしたちはシンボルの関数セルへの関数のセットに fset、関数セルの内容 (Section 13.9 [Function Cells], page 225 を参照) の取得に symbol-functionを使用します。具体的には firstの関数セルにシンボル carを格納して、シンボル firstを ersteの関数セルに格納します。

```
;; この関数セルのリンクを構築する:
;;   -------------       -----        -------        -------
;;  | #<subr car> | <-- | car |  <-- | first |  <-- | erste |
;;   -------------       -----        -------        -------
(symbol-function 'car)
     ⇒ #<subr car>
(fset 'first 'car)
     ⇒ car
(fset 'erste 'first)
     ⇒ first
```

```
(erste '(1 2 3))   ; ersteにより参照される関数を呼び出す
     ⇒ 1
```

対照的に、以下の例ではシンボル関数インダイレクションを使用せずに関数を呼び出しています。なぜなら 1 番目の要素はシンボルではなく、無名 Lisp 関数 (anonymous Lisp function) だからです。

```
((lambda (arg) (erste arg))
 '(1 2 3))
     ⇒ 1
```

関数自身を実行するとその関数の body を評価します。ここでは `erste`を呼び出すとき、シンボル関数インダイレクションが行なわれています。

このフォームが使用されるのは稀であり、現在では推奨されていません。かわりに以下のように記述するべきです:

```
(funcall (lambda (arg) (erste arg))
         '(1 2 3))
```

または単に

```
(let ((arg '(1 2 3))) (erste arg))
```

ビルトイン関数の `indirect-function`は、明示的にシンボル関数インダイレクションを処理するための簡単な方法を提供します。

`indirect-function` *function* **&optional** *noerror* [Function]

この関数は *function*が意味するものを関数としてリターンする。*function*がシンボルなら *function*の関数定義を探して、その値で最初からやり直す。*function*がシンボルでなければ *function* 自身をリターンする。

この関数は最終的なシンボルがバインドされていなければ `nil`をリターンする。特定のシンボル内にループがれば、この関数は `cyclic-function-indirection`エラーをシグナルする。

オペション引数 *noerror*は廃れており、後方互換のためだけのもので効果はない。

以下は Lisp で `indirect-function`を定義する例である:

```
(defun indirect-function (function)
  (if (symbolp function)
      (indirect-function (symbol-function function))
    function))
```

10.1.5 関数フォームの評価

リストの 1 番目の要素が Lisp の関数オブジェクト、バイトコードオブジェクト、プリミティブ関数オブジェクトのいずれかと評価されると、そのリストは関数呼び出し *(function call)* になります。たとえば、以下は関数`+`を呼び出します:

```
(+ 1 x)
```

関数呼び出しを評価する最初のステップでは、そのリストの残りの要素を左から右に評価します。結果は引数の実際の値で、リストの各要素にたいして 1 つの値となります。次のステップでは関数 `apply`(Section 13.5 [Calling Functions], page 217 を参照) を使用して、引数のリストでその関数を呼び出します。関数が Lisp で記述されていたら引数はその関数の引数変数にバインドするために使用されます。その後に関数 body 内のフォームが順番に評価されて、リストの body フォームの値が関数呼び出しの値になります。

10.1.6 Lisp マクロの評価

リストの最初の要素がマクロオブジェクトと評価されると、そのリストはマクロ呼び出し *(macro call)* になります。マクロ呼び出しが評価されるとき、リストの残りの要素は最初は評価されません。そのかわりこれらの要素自体がマクロの引数に使用されます。そのマクロ定義は、元のフォームが評価される場所で置換フォームを計算します。これをマクロの展開 *(expansion)* と言います。展開した結果は、任意の種類のフォーム— 自己評価定数、シンボル、リストになります。展開した結果自体がマクロ呼び出しなら、結果が他の種類のフォームになるまで、繰り返し展開処理が行なわれます。

通常のマクロ展開は、その展開結果を評価することにより終了します。しかし他のプログラムもマクロ呼び出しを展開し、それらが展開結果を評価するか、あるいは評価しないかもしれないので、そのマクロ展開が即時または最終的に評価される必要がない場合があります。

引数式は通常はマクロ展開の計算の一部としては評価されませんが、展開の部分として出現するので、展開結果が評価されるときに計算されます。

たとえば以下のようなマクロ定義が与えられたとします:

```
(defmacro cadr (x)
  (list 'car (list 'cdr x)))
```

`(cadr (assq 'handler list))`のような式はマクロ呼び出しであり、展開結果は以下のようになります:

```
(car (cdr (assq 'handler list)))
```

引数 `(assq 'handler list)`が展開結果に含まれることに注意してください。

Emacs Lisp マクロの完全な説明は Chapter 14 [Macros], page 242 を参照してください。

10.1.7 スペシャルフォーム

スペシャルフォーム *(special form)* とは、特別だとマークされたプリミティブ関数であり、その引数のすべては評価されません。もっとも特別なフォームは制御構文の定義や変数バインディングの処理等、関数ではできないことを行ないます。

スペシャルフォームはそれぞれ、どの引数を評価して、どの引数を評価しないかについて独自のルールをもちます。特定の引数が評価されるかどうかは、他の引数を評価した結果に依存します。

式の最初のシンボルがスペシャルフォームなら、式はそのスペシャルフォームのルールにしたがう必要があります。それ以外なら Emacs の挙動は (たとえクラッシュしなくいとしても) 未定義です。たとえば `((lambda (x) x . 3) 4)`は `lambda`で始まるサブ式を含みますが、これは適正な `lambda`式ではないので、Emacs はエラーをシグナルするかもしれないし、3 や 4 や `nil`をリターンしたり、もしかしたら他の挙動を示すかもしれません。

`special-form-p` *object* [Function]
: この述語は引数がスペシャルフォームかをテストして、スペシャルフォームなら `t`、それ以外なら `nil`をリターンする。

以下に Emacs Lisp のスペシャルフォームすべてと、それらがどこで説明されているかのリファレンスをアルファベット順でリストします。

`and` see Section 11.3 [Combining Conditions], page 148,

`catch` see Section 11.7.1 [Catch and Throw], page 162,

`cond` see Section 11.2 [Conditionals], page 146,

`condition-case`
see Section 11.7.3.3 [Handling Errors], page 167,

`defconst` see Section 12.5 [Defining Variables], page 178,

`defvar` see Section 12.5 [Defining Variables], page 178,

`function` see Section 13.7 [Anonymous Functions], page 220,

`if` see Section 11.2 [Conditionals], page 146,

`interactive`
see Section 21.3 [Interactive Call], page 382,

`lambda` see Section 13.2 [Lambda Expressions], page 211,

`let`
`let*` see Section 12.3 [Local Variables], page 175,

`or` see Section 11.3 [Combining Conditions], page 148,

`prog1`
`prog2`
`progn` see Section 11.1 [Sequencing], page 145,

`quote` see Section 10.2 [Quoting], page 139,

`save-current-buffer`
see Section 27.2 [Current Buffer], page 590,

`save-excursion`
see Section 30.3 [Excursions], page 765,

`save-restriction`
see Section 30.4 [Narrowing], page 766,

`setq` see Section 12.8 [Setting Variables], page 182,

`setq-default`
see Section 12.11.2 [Creating Buffer-Local], page 192,

`unwind-protect`
see Section 11.7 [Nonlocal Exits], page 162,

`while` see Section 11.5 [Iteration], page 159,

> **Common Lisp** に関する注意**:** GNU Emacs と Common Lisp のスペシャルフォームを比較する。`setq`、`if`、`catch`は Emacs Lisp と Common Lisp の両方でスペシャルフォームである。`save-excursion`は Emacs Lisp ではスペシャルフォームだが、Common Lisp には存在しない。`throw`は Common Lisp ではスペシャルフォーム (なぜなら複数の値を throw できなければならない) だが、Emacs Lisp では (複数の値をもたない) 関数である。

10.1.8 自動ロード

オートロード *(autoload)* 機能により、まだ関数定義が Emacs にロードされていない関数 (またはマクロ) を呼び出すことができます。オートロードは定義がどのファイルに含まれるかを指定します。オートロードオブジェクトがシンボルの関数定義にある場合は、関数としてそのシンボルを呼び出すことにより、自動的に指定されたファイルがロードされます。その後にファイルからロードされた実際の定義を呼び出します。シンボル内の関数定義としてオートロードオブジェクトをアレンジする方法は Section 16.5 [Autoload], page 274 で説明します。

10.2 クォート

スペシャルフォーム quoteは、単一の引数を記述されたままに評価せずにリターンします。これはプログラムに自己評価オブジェクトではない、定数シンボルや定数リストを含める方法を提供します (数字、文字列、ベクターのような自己評価オブジェクトをクォートする必要はない)。

`quote` *object* [Special Form]
このスペシャルフォームは *object*を評価せずにリターンする。リターン値は共有されるかもしれないので変更しないこと。Section 10.1.1 [Self-Evaluating Forms], page 134 を参照のこと。

quoteはプログラム中で頻繁に使用されるので、Lisp はそれにたいする便利な入力構文を提供します。アポストロフィー文字 (‘'’) に続けて Lisp オブジェクト (の入力構文) を記述すると、それは 1 番目の要素が quote、2 番目の要素がそのオブジェクトであるようなリストに展開されます。つまり入力構文'xは (quote x)の略記になります。

以下に quoteを使用した式の例をいくつか示します:

```
(quote (+ 1 2))
     ⇒ (+ 1 2)
(quote foo)
     ⇒ foo
'foo
     ⇒ foo
''foo
     ⇒ 'foo
'(quote foo)
     ⇒ 'foo
['foo]
     ⇒ ['foo]
```

(list '+ 1 2)と'(+ 1 2)の 2 つの式はいずれも (+ 1 2)と equal なリストを生成しますが前者は mutable リストを新たに作成するのにたいして、後者は共有される可能性のある変更すべきではないコンスから構築したリストを作成します。Section 10.1.1 [Self-Evaluating Forms], page 134 を参照してください。

他のクォート構文としては、コンパイル用に Lisp で記述された無名のラムダ式の元となる functionを (Section 13.7 [Anonymous Functions], page 220 を参照)、リストを計算して置き換える際にリストの一部だけをクォートするために使用される ‘`’(Section 10.3 [Backquote], page 139 を参照) があります。

10.3 バッククォート

バッククォート構文 *(backquote constructs)* を使用することにより、リストをクォートしてそのリストのある要素を選択的に評価することができます。もっとも簡単な使い方ではスペシャルフォームquoteと同じです。たとえば以下の 2 つのフォームは同じ結果を生みます:

```
`(a list of (+ 2 3) elements)
     ⇒ (a list of (+ 2 3) elements)
'(a list of (+ 2 3) elements)
     ⇒ (a list of (+ 2 3) elements)
```

バッククォートする引数の内側でスペシャルマーカー‘,’を使用すると、それは値が定数でないことを示します。Emacs Lisp エバリュエーターは‘,’がついた引数を放置して、リスト構文内にその値を配置します:

```
`(a list of ,(+ 2 3) elements)
     ⇒ (a list of 5 elements)
```

‘,’による置き換えを、リスト構文のより深いレベルでも使用できます。たとえば:

```
`(1 2 (3 ,(+ 4 5)))
     ⇒ (1 2 (3 9))
```

スペシャルマーカー‘,@’を使用すれば、評価された値を結果リストに継ぎ足す *(splice)* こともできます。継ぎ足されたリストの要素は、結果リスト内の他の要素と同じレベルになります。‘`’を使用しない等価なコードは読むのが困難なことがよくあります。以下にいくつかの例を示します:

```
(setq some-list '(2 3))
     ⇒ (2 3)
(cons 1 (append some-list '(4) some-list))
     ⇒ (1 2 3 4 2 3)
`(1 ,@some-list 4 ,@some-list)
     ⇒ (1 2 3 4 2 3)

(setq list '(hack foo bar))
     ⇒ (hack foo bar)
(cons 'use
  (cons 'the
    (cons 'words (append (cdr list) '(as elements)))))
     ⇒ (use the words foo bar as elements)
`(use the words ,@(cdr list) as elements)
     ⇒ (use the words foo bar as elements)
```

バッククォート構文の部分式に置換や継ぎ足し (splice) がなければ、これは共有される可能性があり変更するべきではないコンス、ベクター、文字列での `quote`のように振る舞います。Section 10.1.1 [Self-Evaluating Forms], page 134 を参照してください。

10.4 `eval` について

フォームはほとんどの場合、実行されるプログラム内に出現することにより自動的に評価されます。ごく稀に実行時 — たとえば編集されているテキストやプロパティーリストから取得したフォームを読み取った後 — に計算されるようにフォームを評価するコードを記述する必要があるかもしれません。このようなときは `eval`関数を使用します。`eval`が不必要だったり、かわりに他の何かを使用すべきときがよくあります。たとえば変数から値を取得するには `eval`も機能しますが、`symbol-value`のほうが適しています。`eval`で評価するためにプロパティーリストに式を格納するかわりに、`funcall`に渡すように関数を格納した方がよいでしょう。

このセクションで説明する関数と変数はフォームの評価、評価処理の制限の指定、最後にリターンされた値の記録を行なうものです。ファイルのロードでも評価が行なわれます (Chapter 16 [Loading], page 269 を参照)。

データ構造に式を格納して評価するより、データ構造に関数を格納して `funcall`や `apply`で呼び出すほうが、より明解で柔軟です。関数を使用することにより、引数に情報を渡す能力が提供されます。

`eval` *form* **&optional** *lexical* [Function]
これは式を評価する基本的な関数である。この関数はカレント環境内で *form*を評価して、その結果をリターンする。*form*オブジェクトの型はそれが評価される方法を決定します。Section 10.1 [Forms], page 134 を参照のこと。

引数 *lexical*は、ローカル変数にたいするスコープ規則 (Section 12.10 [Variable Scoping], page 185 を参照) を指定する。これが省略または `nil`ならデフォルトのダイナミックスコープ規則を使用して *form*を評価することを意味する。`t`ならレキシカルスコープ規則が使用されることを意味する。*lexical*の値にはレキシカルバインディングでの特定のレキシカル環境 *(lexical environment)* を指定する空ではない alist も指定できる。しかしこの機能は Emacs Lisp デバッガーのような、特別な用途にたいしてのみ有用。Section 12.10.3 [Lexical Binding], page 187 を参照のこと。

`eval`は関数なので `eval`呼び出しに現れる引数式は 2 回 — `eval`が呼び出される前の準備で一度、`eval`関数自身によりもう一度 — 評価される。以下に例を示す:

```
(setq foo 'bar)
     ⇒ bar
(setq bar 'baz)
     ⇒ baz
;; evalが引数 fooを受け取る
(eval 'foo)
     ⇒ bar
;; evalが、fooの値である、引数 barを受け取る
(eval foo)
     ⇒ baz
```

`eval`で現在アクティブな呼び出しの数は `max-lisp-eval-depth`に制限される (以下参照)。

`eval-region` *start end* **&optional** *stream read-function* [Command]
この関数はカレントバッファー内の、位置 *start*と *end*で定義されるリージョン内のフォームを評価する。この関数はリージョンからフォームを読み取って `eval`を呼び出す。これはリージョンの最後に達するか、処理されないエラーがシグナルされるまで行なわれる。

デフォルトでは `eval-region`は出力を何も生成しない。しかし *stream*が非 `nil`なら出力関数 (Section 19.5 [Output Functions], page 337 を参照) で生成された任意の出力、同様にリージョン内の式を評価した結果の値が、*stream*を使用してプリントされる。Section 19.4 [Output Streams], page 335 を参照のこと。

*read-function*が非 `nil`なら、`read`のかわりに 1 つずつ式を読み取るために使用する関数を指定すること。これは入力を読み取るストリームを指定する、1 つの引数で呼び出される関数である。この関数を指定するために変数 `load-read-function`([How Programs Do Loading], page 271 を参照) も使用できるが、引数 *read-function*を使用するほうが堅実である。

`eval-region`はポイントを移動しない。常に `nil`をリターンする。

`eval-buffer` **&optional** *buffer-or-name stream filename unibyte print* [Command]
この関数は `eval-region`と似ているが、引数は異なるオプション機能を提供する。`eval-buffer`はバッファー *buffer-or-name*のアクセス可能な部分 (Section "Narrowing" in *The GNU Emacs Manual* を参照) の全体を処理する。*buffer-or-name*にはバッファー名 (文字列) を指定でき、`nil`(または省略) のときはカレントバッファーを意味する。*stream*が非 `nil`、または *print*が `nil`なら、`eval-region`のように *stream*が使用される。この場合には

式の評価結果の値は依然として破棄されるが、出力関数による出力はエコーエリアにプリントされる。*filename*は `load-history` (Section 16.9 [Unloading], page 282 を参照) に使用されるファイル名であり、デフォルトは `buffer-file-name` (Section 27.4 [Buffer File Name], page 594 を参照)。*unibyte*が非 `nil`なら `read`可能な限りは文字列をユニコードに変換する。

`eval-current-buffer`はこのコマンドのエイリアスである。

`max-lisp-eval-depth` [User Option]

この変数はエラー (エラーメッセージは`"Lisp nesting exceeds max-lisp-eval-depth"`) がシグナルされる前に `eval`、`apply`、`funcall`の呼び出しで許容される最大の深さを定義する。

制限を超過時のエラーを付随するこの制限は、誤って定義された関数による無限再帰を Emacs Lisp が回避する方法の 1 つである。`max-lisp-eval-depth`の値を過大に増加させると、そのようなコードはかわりにスタックオーバーフローを起こすだろう。オーバーフローを処理できるシステムがいくつかある。この場合には通常の Lisp 評価は割り込まれて、制御はトップレベルのコマンドループ (`top-level`) に戻される。この状況では Emacs Lisp デバッガにエンターする手段は存在しないことに注意されたい。Section 18.1.1 [Error Debugging], page 296 を参照のこと。

Lisp 式に記述された関数の呼び出し、関数呼び出しの引数と関数 body フォームにたいする再帰評価、Lisp コード内での明示的な呼び出し等では内部的に `eval`、`apply`、`funcall`を使用して深さ制限を計数する。

この変数のデフォルト値は 800。この値を 100 未満にセットした場合には、値が与えられた値に達すると Lisp はそれを 100 にリセットする。空きが少なければデバッガー自身を実行するために空きが必要になるので、Lisp デバッガーに入ったときは値が増加される。

`max-specpdl-size`はネストの他の制限を提供する。[Local Variables], page 177 を参照のこと。

`values` [Variable]

この変数の値は読み取り、評価、プリントを行なった標準的な Emacs コマンドにより、バッファー (ミニバッファーを含む) からリターンされる値のリストである (これには`*ielm*`バッファーでの評価や、`lisp-interaction-mode`での *C-j*や *C-x C-e*、類似の評価コマンドを使用した評価は含まれないことに注意)。要素の順番はもっとも最近のものが最初になる。

```
(setq x 1)
     ⇒ 1
(list 'A (1+ 2) auto-save-default)
     ⇒ (A 3 t)
values
     ⇒ ((A 3 t) 1 ...)
```

この変数は最近評価されたフォームの値を後で参照するのに有用。`values`自体の値のプリントは、値がおそらく非常に長くなるので通常は悪いアイデアである。かわりに以下のように特定の要素を調べること:

```
;; もっとも最近評価された結果を参照する
(nth 0 values)
     ⇒ (A 3 t)
;; これは新たな要素を put するので
;;    すべての要素が 1 つ後に移動する
(nth 1 values)
     ⇒ (A 3 t)
```

```
;; これは次に新しい、この例の前の次に新しい要素を取得する
(nth 3 values)
     ⇒ 1
```

10.5 遅延された Lazy 評価

たとえばプログラムの将来において計算結果が不要なら時間を要する計算処理を回避したい等、式の評価を遅延できれば便利な場合があります。そのような遅延評価 *(deferred evaluation)* をサポートするために、thunkは以下の関数とマクロを提供します。

`thunk-delay` *forms...* [Macro]

*forms*を評価するための *thunk* をリターンする (訳注: thunk とは別のサブルーチンに計算を追加で挿入するために使用するサブルーチンであり、計算結果が必要になるまで計算を遅延したり、別のサブルーチンの先頭や最後に処理を挿入するために使用される。英語版 Wikipedia より)。thunk は `thunk-delay`呼び出しの lexical 環境を継承するクロージャである (Section 13.10 [Closures], page 227 を参照)。このマクロの使用には `lexical-binding`が必要。

`thunk-force` *thunk* [Function]

thunk を作成した `thunk-delay`で指定されたフォームの評価を *thunk*に強制する。最後のフォームの評価結果をリターンする。*thunk*が強制されたことも "記憶" される。同一の *thunk* にたいする以降の `thunk-force`呼び出しでは、フォームを再度評価せずに同じ結果をリターンする。

`thunk-let` (*bindings...*) *forms...* [Macro]

このマクロは `let`の類似だが "lazy(遅延された)" 変数バインディングを作成する。すべてのバインディングは (*symbol value-form*) という形式をもつ。`let`とは異なり、すべての *value-form*の評価は *forms*を最初に評価する際に、対応する *symbol*のバインディングが使用されるまで遅延される。すべての *value-form*は最大でも 1 回評価される。このマクロの使用には `lexical-binding`が必要。

例:

```
(defun f (number)
  (thunk-let ((derived-number
               (progn (message "Calculating 1 plus 2 times %d" number)
                      (1+ (* 2 number)))))
    (if (> number 10)
        derived-number
      number)))

(f 5)
⇒ 5

(f 12)
⊣ Calculating 1 plus 2 times 12
⇒ 25
```

遅延バインドされた変数の特性として、それらにたいする (`setq`による) セットはエラーになります。

`thunk-let*` (*bindings...*) *forms...* [Macro]

これは `thunk-let`と似ているが、*bindings*内の任意の式がこの `thunk-let*`フォーム内の先行するバインディングの参照を許されている点が異なる。このマクロの使用には `lexical-binding`が必要。

```
(thunk-let* ((x (prog2 (message "Calculating x...")
                    (+ 1 1)
                  (message "Finished calculating x")))
             (y (prog2 (message "Calculating y...")
                    (+ x 1)
                  (message "Finished calculating y")))
             (z (prog2 (message "Calculating z...")
                    (+ y 1)
                  (message "Finished calculating z")))
             (a (prog2 (message "Calculating a...")
                    (+ z 1)
                  (message "Finished calculating a"))))
  (* z x))

⊣ Calculating z...
⊣ Calculating y...
⊣ Calculating x...
⊣ Finished calculating x
⊣ Finished calculating y
⊣ Finished calculating z
⇒ 8
```

`thunk-let`と `thunk-let*`は thunk を暗黙に使用します。これらの拡張はヘルパーシンボルを作成してバインディング式をラップする thunk にバインドします。*forms*本体中の元の変数にたいするすべての参照は、対応するヘルパー変数を引数とする `thunk-force`呼び出し式に置き換えられます。したがって `thunk-let`や `thunk-let*`を使用するコードは thunk を使用するように書き換えが可能ですが、多くの場合には明示的に thunk を使用するよりこれらのマクロを使用するほうが優れたコードになるでしょう。

11 制御構造

Lisp プログラムは一連の式、あるいはフォーム (Section 10.1 [Forms], page 134 を参照) により形成されます。これらのフォームの実行順は制御構造 *(control structures)* で囲むことによって制御します。制御構造とはその制御構造が含むフォームをいつ、どのような条件で、何回実行するかを制御するスペシャルフォームです。

もっとも単純な実行順は 1 番目は a、2 番目は b、... というシーケンシャル実行 (sequential execution: 順番に実行) です。これは関数の body 内の連続する複数のフォームや、Lisp コードのファイル内のトップレベルを記述したときに発生します — つまりフォームは記述した順に実行されます。わたしたちはこれをテキスト順 *(textual order)* と呼びます。たとえば関数の body が 2 つのフォーム *a*と *b*から構成される場合、関数の評価は最初に *a*、次に *b*を評価します。*b*を評価した結果がその関数の値となります。

明示的に制御構造を使用することにより、非シーケンシャルな順番での実行が可能になります。

Emacs Lisp は他の様々な順序づけ、条件、繰り返し、(制御された) ジャンプを含む複数の種類の制御構造を提供しており、以下ではそれらのすべてを記述します。ビルトインの制御構造は制御構造のサブフォームが評価される必要がなかったり、順番に評価される必要がないのでスペシャルフォームです。独自の制御構造を構築するためにマクロを使用することができます (Chapter 14 [Macros], page 242 を参照)。

11.1 順序

フォームを出現順に評価するのは、あるフォームから別のフォームに制御を渡すもっとも一般的な制御です。関数の body のようなコンテキストにおいては自動的にこれが行なわれます。他の場所ではこれを行なうために制御構造を使用しなければなりません。Lisp で一単純な制御構造は `progn`です。

スペシャルフォーム `progn`は以下のようなものです:

```
(progn a b c ...)
```

これは順番に *a*、*b*、*c*、... を実行するよう指定します。これらは `progn`フォームの *body* と呼ばれます。body 内の最後のフォームの値が `progn`全体の値になります。`(progn)`は `nil`をリターンします。

初期の Lisp では `progn`は、連続で複数のフォームを実行して最後のフォームの値を使用する唯一の方法でした。しかしプログラマーは関数の body の、(その時点では)1 つのフォームだけが許される場所で `progn`を使用する必要が多いことに気づきました。そのため関数の body を暗黙の `progn`にして、`progn`の body のように複数のフォームを記述できるようにしました。他の多くの制御構造も暗黙の `progn`を同様に含みます。結果として昔ほど `progn`は多用されなくなりました。現在では `progn`が必要になるのは `unwind-protect`、`and`、`or`、または `if`の *then*パートの中であることがほとんどです。

`progn` *forms...* [Special Form]

このスペシャルフォームは *forms*のすべてをテキスト順に評価してフォームの結果をリターンする。

```
(progn (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The third form"
```

他の 2 つの構文は一連のフォームを同様に評価しますが、異なる値をリターンします:

`prog1` *form1 forms...* [Special Form]

このスペシャルフォームは *form1*と *forms*のすべてをテキスト順に評価して *form1*の結果をリターンする。

```
(prog1 (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The first form"
```

以下の例は変数 `x`のリストから 1 番目の要素を削除して、削除した 1 番目の要素の値をリターンする:

```
(prog1 (car x) (setq x (cdr x)))
```

`prog2` *form1 form2 forms...* [Special Form]

このスペシャルフォームは *form1*、*form2*、その後の *forms*のすべてをテキスト順で評価して *form2*の結果をリターンする。

```
(prog2 (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The second form"
```

11.2 条件

条件による制御構造は候補の中から選択を行ないます。Emacs Lisp は 5 つの条件フォームをもちます。`if`は他の言語のものとほとんど同じです。`when`と `unless`は `if`の変種です。`cond`は一般化された case 命令です。`cond`を汎用化したものが `pcase`です (Section 11.4 [Pattern-Matching Conditional], page 149 を参照)。

`if` *condition then-form else-forms...* [Special Form]

`if`は *condition*の値にもとづき *then-form*と *else-forms*を選択する。評価された *condition*が非 `nil`なら *then-form*が評価されて結果がリターンされる。それ以外なら *else-forms*がテキスト順に評価されて最後のフォームの値がリターンされる (`if`の *else*パートは暗黙の `progn`の例である。Section 11.1 [Sequencing], page 145 を参照)。

*condition*の値が `nil`で *else-forms*が与えられなければ、`if`は `nil`をリターンする。

選択されなかったブランチは決して評価されない — 無視される — ので、`if`はスペシャルフォームである。したがって以下の例では `print`が呼び出されることはないので `true`はプリントされない。

```
(if nil
    (print 'true)
  'very-false)
⇒ very-false
```

`when` *condition then-forms...* [Macro]

これは *else-forms*がなく、複数の *then-forms*が可能な `if`の変種である。特に、

```
(when condition a b c)
```

は以下と完全に等価である

```
(if condition (progn a b c) nil)
```

`unless` *condition forms...* [Macro]

これは *then-form*がない `if`の変種です:

```
(unless condition a b c)
```

は以下と完全に等価である

```
(if condition nil
   a b c)
```

`cond` *clause...* [Special Form]

`cond`は任意個数の選択肢から選択を行なう。`cond`内の各 *clause*はリストでなければならない。このリストの CAR は *condition*で、(もしあれば) 残りの要素は *body-forms*となる。したがって clause は以下のようになる:

```
(condition body-forms...)
```

`cond`は各 clause の *condition*を評価することにより、テキスト順で clause を試みる。*condition* の値が非 `nil`ならその clause は成り立つ。その後に `cond`はその clause の *body-forms*を評価して、*body-forms*の最後の値をリターンする。残りの clause は無視される。

*condition*の値が `nil`ならその clause は失敗して、`cond`は次の clause に移動してそれの *condition*を試みる。

clause は以下のようにも見えるかもしれない:

```
(condition)
```

*condition*がテストされたときに非 `nil`なら、`cond`フォームは *condition*の値をリターンする。

すべての *condition*が `nil`に評価された場合 — つまりすべての clause が不成立なら、`cond` は `nil`をリターンする。

以下の例は `x`の値が数字、文字列、バッファー、シンボルなのかをテストする 4 つの clause をもつ:

```
(cond ((numberp x) x)
      ((stringp x) x)
      ((bufferp x)
       (setq temporary-hack x) ; 1 つの clause に
       (buffer-name x))        ; 複数 body フォーム
      ((symbolp x) (symbol-value x)))
```

前の clause が不成立のとき最後の条項を実行したいときがよくある。これを行なうには `(t body-forms)`のように、*condition*の最後の clause に `t`を使用する。フォーム `t`は `t`に評価され決して `nil`にならないので、この clause が不成立になることはなく最終的に `cond`はこの clause に到達する。たとえば:

```
(setq a 5)
(cond ((eq a 'hack) 'foo)
      (t "default"))
⇒ "default"
```

この `cond`式は `a`の値が `hack`なら `foo`、それ以外は文字列`"default"`をリターンする。

すべての条件構文は condかifのいずれかで表すことができます。したがってどちらを選択するかはスタイルの問題になります。たとえば:

```
(if a b c)
≡
(cond (a b) (t c))
```

11.3 組み合わせ条件の構築

このセクションでは複雑な条件を表現するために ifや condとともによく使用される構文を説明します。andと orの構文は、ある種の複数条件の構文として個別に使用することもできます。

not *condition* [Function]

この関数は *condition*が偽であることをテストする。この関数は *condition*が nilなら t、それ以外は nilをリターンする。関数 notは nullと等価であり、空のリストをテストする場合は nullの使用を推奨する。

and *conditions...* [Special Form]

スペシャルフォーム andは、すべての *conditions*が真かどうかをテストする。この関数は *conditions*を記述順に 1 つずつ評価することにより機能する。

ある *conditions*が nilに評価されると、残りの *conditions*に関係なく、andは nilをリターンしなければならない。この場合 andは即座に nilをリターンして、残りの *conditions*は無視される。

すべての *conditions*が非 nilなら、それらの最後の値が andフォームの値になる。*conditions*がない単独の (and)は tをリターンする。なぜならすべての *conditions*が非 nilとなるので、これは適切である (考えてみてみよ、非 nilでない *conditions*はどれか?)。

以下に例を示す。1 番目の条件は整数 1 をリターンし、これは nilではまい。同様に 2 番目の条件は整数 2 をリターンし、これも nilではない。3 番目の条件は nilなので、のこりの条件が評価されることは決してない。

```
(and (print 1) (print 2) nil (print 3))
     ⊣ 1
     ⊣ 2
⇒ nil
```

以下は andを使用した、より現実的な例である:

```
(if (and (consp foo) (eq (car foo) 'x))
    (message "foo is a list starting with x"))
```

(consp foo)が nilをリターンすると、(car foo)は実行されないのでエラーにならないことに注意。

ifか condのいずれかを使用して、and式を記述することもできる。以下にその方法を示す:

```
(and arg1 arg2 arg3)
≡
(if arg1 (if arg2 arg3))
≡
(cond (arg1 (cond (arg2 arg3))))
```

or *conditions...* [Special Form]

スペシャルフォーム orは、少なくとも 1 つの *conditions*が真かどうかをテストする。この関数はすべての *conditions*を 1 つずつ、記述された順に評価することにより機能する。

ある *conditions*が非 `nil`値に評価されたら、`or`の結果は非 `nil`でなければならない。この場合 `or`は即座にリターンし、残りの *conditions*は無視される。この関数がリターンする値は、非 `nil`値に評価された条件の値そのものである。

すべての *conditions*が `nil`なら、`or`式は `nil`をリターンします。*conditions*のない単独の `(or)` は `nil`をリターンする。なぜならすべての *conditions*が `nil`になるのでこれは適切である (考えてみよ、`nil`でない *conditions*はどれか?)。

たとえば以下の式は、`x`が `nil`か整数 0 かどうかをテストする:

```
(or (eq x nil) (eq x 0))
```

`and`構文と同様に、`or`を `cond`に置き換えて記述することができる。たとえば:

```
(or arg1 arg2 arg3)
≡
(cond (arg1)
      (arg2)
      (arg3))
```

ほとんどの場合は、`or`を `if`に置き換えて記述できるが完全ではない:

```
(if arg1 arg1
  (if arg2 arg2
    arg3))
```

これは完全に同一ではない。なぜなら *arg1*か *arg2*を 2 回評価するかもしれないからである。対照的に `(or` *arg1 arg2 arg3*`)`が 2 回以上引数を評価することは決してない。

`xor` *condition1 condition2* [Function]

この関数は *condition1*と *condition2*の排他的論理和をリターンする。つまり `xor`は引数がいずれも `nil`あるいは非 `nil`なら `nil`をリターンする。それ以外なら非 `nil`の引数の値をリターンする。

`or`とは対照的に引数はどちらも常に評価されることに注意。

11.4 パターンマッチングによる条件

4 つの基本的な条件フォームとは別に、Emacs Lisp には `cond`と `cl-case` (Section “Conditionals” in *Common Lisp Extensions* を参照) の合成物とも言うべき、`pcase`マクロというパターンマッチングによる条件フォームがあります。これは `cond`と `cl-case`の制限を克服して、パターンマッチングによるプログラミングスタイル *(pattern matching programming style)* を導入するものです。その `pcase`が克服する制限とは:

- `cond`フォームは clause それぞれにたいして述語 *condition*を評価して候補から選択を行う (Section 11.2 [Conditionals], page 146 を参照)。*condition*内での let バインドされた変数が clause の *body-forms*で利用できないのが主な制限。

  もう 1 つの煩しい (制限というより不便な) 点は一連の *condition*述語が等価なテストを実装する際にはコードが多数回繰り返されること (`cl-case`はこの不便さを解決している)。

- `cl-case`マクロは最初の引数と特定の値セットの等価性を評価することにより候補から選択を行う。

  制限は 2 つ:

  1. 等価性のテストに `eql`を使用。
  2. 値は既知でありあらかじめ記述されていなければならない。

これらの制限は文字列や複合データ構造にたいして `cl-case`を不適格にする (このような制限は `cond`にはないが上述のように別の制限をもつ)。

`pcase`マクロはパターンマッチング *(pattern matching)* の変種であるような等価性テストを汎化したものによる *condition*の置き換え、clause の述語を簡潔に表現できるような機能の追加、clause の述語と *body-forms*の間で let バインディングを共有するようなアレンジにより、概念的には最初の引数のフォーカスでは `cl-case`、clause の処理フローでは `cond`を借用しています。

この述語の簡潔な表現はパターン *(pattern)* として知られています。最初の引数の値にたいして呼び出される述語が非 `nil`をリターンしたときには、“パターンが値にマッチした” といいます (“値がパターンにマッチした” ということもある)。

11.4.1 `pcase`マクロ

背景は Section 11.4 [Pattern-Matching Conditional], page 149 を参照してください。

`pcase` *expression* **&rest** *clauses* [Macro]

*clauses*内の clause は `(pattern body-forms...)` という形式をもつ。

*expression*の値 (*expval*) を決定するために評価する。*pattern*が *expval*にマッチするような最初の clause を探して、その clause の *body-forms*に制御を渡す。

マッチが存在すれば `pcase`の値はマッチが成功した clause の *body-forms*の最後の値、それ以外なら `pcase`は `nil`に評価される。

各 *pattern*は *pcase* パターン *(pcase pattern)* である必要があります。これは以下に定義されるコアパターンのいずれか、または `pcase-defmacro` (Section 11.4.2 [Extending pcase], page 156 を参照) を通じて定義されるパターンの 1 つを使用できます。

このサブセクションの残りの部分ではコアパターンの異なるフォームをいくつかの例を交えて説明して、いくつかのパターンフォームが提供する let バイディング機能を使用する上で重要な注意点で締めくくります。コアパターンは以下のフォームをもつことができます:

`_` 任意の *expval*にマッチ。これは *don't care*、あるいはワイルドカード *(wildcard)* としても知られる。

`'val` *expval*と *val*が等しければマッチ。比較は `equal`のように行われる (Section 2.8 [Equality Predicates], page 33 を参照)。

`keyword`
`integer`
`string` *expval*がリテラルオブジェクトと等しければマッチ。これは上述の`'val`の特殊なケースであり、これらのタイプのリテラルオブジェクトが自己クォート (self-quoting) を行うために可能となる。

`symbol` 任意の *expval*にマッチするとともに、追加で *symbol*を *expval*に let バインドする。このようなバインディングは *body-forms*内でも利用できる (Section 12.10.1 [Dynamic Binding], page 185 を参照)。

*symbol*が (以下の `and`を使用することにより) シーケンスパターン *seqpat*の一部なら、*symbol*に後続する *seqpat*部分でもバインディングは利用可能。この使用法にはいくつかの注意点がある。[caveats], page 153 を参照のこと。

使用を避けるべき 2 つのシンボルは`_` (上述) と同様に振る舞う非推奨の `t`と `nil`(エラーをシグナルする)。同様にキーワードシンボルへのバインドは無意味 (Section 12.2 [Constant Variables], page 174 を参照)。

`(pred` *function*`)`

*expval*にたいして述語 *function*が非 `nil`をリターンしたらマッチ。述語 *function*は以下のフォームのいずれかが可能:

関数名 (シンボル)
: *expval*を単一の引数として名前つきの関数を呼び出す。

  例: `integerp`

lambda 式
: *expval*を単一の引数として無名関数を呼び出す (Section 13.2 [Lambda Expressions], page 211 を参照)。

  例: `(lambda (n) (= 42 n))`

*n*個の引数での関数呼び出し
: 関数 (関数呼び出しの 1 つ目の要素) を *n*個の引数 (残りの要素)、および *expval*を *n*+1 番目の追加の引数としてで呼び出す。

  例: `(= 42)`
  この例では関数が`=`、*n*が 1 であり、実際の関数呼び出しは `(= 42` *expval*`)` になる。

`(app` *function pattern*`)`

*expval*にたいして呼び出した *function*が *pattern*にマッチする値をリターンすればマッチ。*function*は上述の `pred`で説明したフォームのいずれかが可能。しかし `pred`とは異なり、`app`はブーリーンの真値ではなく *pattern*にたいして結果をテストする点が異なる。

`(guard` *boolean-expression*`)`

*boolean-expression*が非 `nil`に評価されればマッチ。

`(let` *pattern expr*`)`

*exprval*を取得するために *expr*を評価して、*exprval*が *pattern*にマッチすればマッチ (*pattern*は *symbol*を使用することでシンボルに値をバインドできるので `let`と呼ばれる)。

*seqpat*としても知られるシーケンスパターン *(sequencing pattern)* はサブパターンを順に処理するパターンです。`pcase`にたいしては `and`と `or`の 2 つが存在します。これらは同じ名前のスペシャルフォーム (Section 11.3 [Combining Conditions], page 148 を参照) と同様の方式で振る舞いますが、値ではなくサブパターンを処理します。

`(and` *pattern1...*`)`

pattern1...のいずれかがマッチに失敗するまで順にマッチを試みる。この場合には `and`もマッチに失敗して、残りのサブパターンはテストしない。すべてのサブパターンがマッチすれば `and`はマッチする。

`(or` *pattern1 pattern2...*`)`

pattern1、*pattern2*、...のいずれかがマッチに成功するまで順にマッチを試みる。この場合には `or`もマッチして、残りのサブパターンはテストしない (少なくともサブパターンが 2 つなければならにことに注意。単なる `(or` *pattern1*`)` はエラーをシグナルする)。

*body-forms*に一定の環境 (⟨undefined⟩ [Intro Eval], page ⟨undefined⟩ を参照) を与えるためには、シンボルセットを let バインドするサブパターンがある場合には、それらは同一のシンボルセットすべてをバインドしなければならない。

例: cl-caseにたいする利点

以下はcl-case (Section "Conditionals" in *Common Lisp Extensions* を参照) にたいするpcaseの利点のいくつかを強調する例です。

```
(pcase (get-return-code x)
  ;; 文字列
  ((and (pred stringp) msg)
   (message "%s" msg))
  ;; シンボル
  ('success       (message "完了!"))
  ('would-block   (message "残念、今は実行できない"))
  ('read-only     (message "shmliblick は読み取り専用"))
  ('access-denied (message "必要な権限がない"))
  ;; default
  (code           (message "未知のリターンコード %S" code)))
```

cl-caseではget-return-codeのリターン値を保持するためにローカル変数codeを宣言する必要があります。さらにcl-caseは比較にeqlを使用するので文字列の使用も難しくなります。

例: andの使用

後続のサブパターン (と同様に body フォーム) にバインディングを提供する 1 つ以上の *symbol*サブパターンをもつandで開始するのが、パターンを記述する際の一般的なイディオムです。たとえば以下のパターンは 1 桁の整数にマッチします。

```
(and
  (pred integerp)
  n                       ; nに expvalをバインド
  (guard (<= -9 n 9)))
```

まずpredは(integerp *expval*)が非nilに評価されればマッチになります。次にnはすべてにマッチする *symbol*パターンであり、nに *expval*がバインドされます。最後にguardはブーリーン式(<= -9 n 9)が非nilに評価されればマッチになります (nへの参照に注意)。これらすべてのサブパターンがマッチすればandがマッチになります。

例: pcaseによる書き換え

以下はシンプルなマッチングタスクを伝統的な実装 (関数grok/traditionalから、pcaseを使用する実装 (関数grok/pcase) に書き換える別の例です。これらの関数の docstring はいずれも "If OBJ is a string of the form "key:NUMBER", return NUMBER (a string). Otherwise, return the list ("149" default)." です。最初は伝統的な実装です (Section 34.3 [Regular Expressions], page 883 を参照):

```
(defun grok/traditional (obj)
  (if (and (stringp obj)
           (string-match "^key:\\([[:digit:]]+\\)$" obj))
      (match-string 1 obj)
    (list "149" 'default)))

(grok/traditional "key:0")   ⇒ "0"
(grok/traditional "key:149") ⇒ "149"
(grok/traditional 'monolith) ⇒ ("149" default)
```

この書き換えでは *symbol*バインディング、および `or`、`and`、`pred`、`app`、`let`を実演します。

```
(defun grok/pcase (obj)
  (pcase obj
    ((or                                        ; L1
      (and                                      ; L2
       (pred stringp)                           ; L3
       (pred (string-match                      ; L4
              "^key:\\([[:digit:]]+\\)$"))      ; L5
       (app (match-string 1)                    ; L6
            val))                               ; L7
      (let val (list "149" 'default)))          ; L8
     val)))                                     ; L9

(grok/pcase "key:0")   ⇒ "0"
(grok/pcase "key:149") ⇒ "149"
(grok/pcase 'monolith) ⇒ ("149" default)
```

`grok/pcase`は主に 1 つの `pcase`フォームの clause、L1 から L8 のパターン、L9 の (1 つの) フォームからなります。パターンは引数であるサブパターンにたいして順にマッチを試みる `or`です。これは最初に `and` (L2 から L7)、次に `let` (L8) のように、いずれかが成功するまで順にマッチを試みます。

前出の例 ([Example 1], page 152 を参照) のように、`and`は以降のサブパターンが正しいタイプのオブジェクト (この場合は文字列) に作用することを保証するために `pred`サブパターンで始まります。`(stringp` *expval*`)` が `nil`なら `pred`は失敗となり、したがって `and`も失敗となります。

次の `pred` (L4 から L5) は `(string-match RX` *expval*`)` を評価して結果が非 `nil` (*expval*が期待するフォーム `key:NUMBER`であることを意味する) ならマッチになります。これが失敗すると再び `pred`は失敗となり、したがって `and`も失敗となります。

(`and`の一連のサブパターンでは) 最後の `app`は一時的な値 *tmp* (部分文字列 “NUMBER”) を取得するために `(match-string 1` *expval*`)` (L6) を評価して、パターン `val` (L7) と *tmp*のマッチを試みます。これは *symbol*パターンなので無条件でマッチして、`val`に *tmp*を追加でバインドします。

ついに `app`がマッチしたので、`and`のすべてのサブパターンがマッチして、`and`がマッチとなります。同じように一度 `and`がマッチすれば `or`がマッチするので、サブパターン `let` (L8) の処理が試みられることはありません。

`obj`が文字列以外、あるいは間違った形式の文字列の場合を考えてみましょう。この場合にはいずれかの `pred` (L3 から L5) がマッチに失敗するので、`and` (L2) がマッチに失敗して、`or` (L1) がサブパターン `let` (L8) の処理を試みます。

まず `let`は `("149" default)` を取得するために `(list "149" 'default)` を評価して、それからパターン `val`にたいして *exprval*のマッチを試みます。これは *symbol*パターンなので無条件にマッチして、追加で `val`に *exprval*をバインドします。これで `let`がマッチしたので、`or`がマッチとなります。

`and`と `let`のサブパターンはどちらも同じ方法、すなわち `val`をバインドする過程で *symbol*パターンの `val`に (常に成功する) マッチを試みることにより完了することに注意してください。したがって `or`は常にマッチして、常に制御を body フォーム (L9) に渡します。マッチが成功した `pcase`の clause としては最後の body なので、これは `pcase`の値となり、同様に `grok/pcase`のリターン値になります (Section 13.1 [What Is a Function], page 209 を参照)。

シーケンスパターンにおける ***symbol***の注意点

前出の例のすべてでは、何らかの方法により *symbol*サブパターンが含まれるシーケンスパターンが使用されています。以下に使用法に関する重要な詳細をいくつか挙げます。

1. *seqpat*内に *symbol*が複数回出現する場合には、2 回目以降に出現してもリバインドには展開されないが、かわりに eqを使用した等価性テストに展開される。

   以下の例には 2 つの clause と 2 つの *seqpat* (A と B) を使用している。A と B はいずれも最初に (predを使用することにより)*expval*がペアであることをチェックして、それから (それぞれにたいして appを使用することにより) *expval*の carと cdrにシンボルをバインドする。

   A ではシンボル stが 2 回記述されているので、2 つ目の記述は eqを使用した等価性チェックになる。一方で B は s1と s2という別個のシンボルを使用するので、独立したバインディングになる。

```
(defun grok (object)
  (pcase object
    ((and (pred consp)        ; seqpat A
          (app car st)        ; st: 1 回目
          (app cdr st))       ; st: 2 回目
     (list 'eq st))
    ((and (pred consp)        ; seqpat B
          (app car s1)        ; s1: 1 回目
          (app cdr s2))       ; s2: 1 回目
     (list 'not-eq s1 s2))))

(let ((s "yow!"))
  (grok (cons s s)))      ⇒ (eq "yow!")
(grok (cons "yo!" "yo!")) ⇒ (not-eq "yo!" "yo!")
(grok '(4 2))             ⇒ (not-eq 4 (2))
```

2. *symbol*を参照するコードの副作用は未定義。無視する。たとえば以下 2 つの関数は類似している。いずれの関数も and、*symbol*、guardを使用する:

```
(defun square-double-digit-p/CLEAN (integer)
  (pcase (* integer integer)
    ((and n (guard (< 9 n 100))) (list 'yes n))
    (sorry (list 'no sorry))))

(square-double-digit-p/CLEAN 9) ⇒ (yes 81)
(square-double-digit-p/CLEAN 3) ⇒ (no 9)

(defun square-double-digit-p/MAYBE (integer)
  (pcase (* integer integer)
    ((and n (guard (< 9 (incf n) 100))) (list 'yes n))
    (sorry (list 'no sorry))))

(square-double-digit-p/MAYBE 9) ⇒ (yes 81)
(square-double-digit-p/MAYBE 3) ⇒ (yes 9)  ; WRONG!
```

   違いは guard内の *boolean-expression*である。CLEANは単に直接 nを参照するのにたいして、MAYBEは式 (incf n)の中で副作用により nを参照している。integerの際には以下のようなことが発生している:

- 最初の nは *expval* (評価した結果である (* 3 3)、つまり 9) にバインドされる。
- *boolean-expression*が評価される:

```
start:   (< 9 (incf n)        100)
becomes: (< 9 (setq n (1+ n)) 100)
becomes: (< 9 (setq n (1+ 9)) 100)
becomes: (< 9 (setq n 10)     100)
                                   ; ここで副作用!
becomes: (< 9       n         100) ; nは 10 にバインドされている
becomes: (< 9      10         100)
becomes: t
```

- 結果は非 nilなので guardがマッチして andがマッチとなり、制御はその clause の body フォームに渡される。

MAYBEには 9 が 2 桁の整数だと判定してしまう数学的な誤り以外にも問題がある。body フォームは nの更新された値 (10) を確認せずに参照を複数回行う。するとどうなるか?

要約すると (guardでの) *boolean-expression*だけではなく (letでの) *expr*、(predと appでの) *function*でも副作用をもつ *symbol*パターンにたいする参照は完全に避けることが最良である。

3. マッチでは clause の body フォームは let バインドされたパターンのシンボルセットを参照できる。このシンボルセットは *seqpat*が andの際には、各サブパターンで let バインドされるすべてのシンボルそれぞれを結合したものになる。andのマッチではすべてのサブパターンがマッチしなければならないので、これには意味がある。

   *seqpat*が orなら事情は異なる。orは最初にマッチしたサブパターンでマッチとなり、残りのサブパターンは無視される。body フォームにはどのサブパターンがマッチして異なるセットの中からどれが選択されたかを知る術はないので、各シンボルが異なるシンボルセットを let バインドすることに意味はない。たとえば以下は無効:

```
(require 'cl-lib)
(pcase (read-number "Enter an integer: ")
  ((or (and (pred cl-evenp)
            e-num)      ; e-numを expvalにバインド
       o-num)           ; o-numを expvalにバインド
   (list e-num o-num)))

Enter an integer: 42
error  Symbol's value as variable is void: o-num
Enter an integer: 149
error  Symbol's value as variable is void: e-num
```

   body フォーム (list e-num o-num) の評価によりエラーがシグナルされる。サブパターンを区別するために、すべてのサブパターンごとに異なる値をもつ同一名のシンボルを使用できる。上記の例を書き換えると:

```
(require 'cl-lib)
(pcase (read-number "Enter an integer: ")
  ((and num                                ; L1
        (or (and (pred cl-evenp)           ; L2
                 (let spin 'even))         ; L3
            (let spin 'odd)))              ; L4
   (list spin num)))                       ; L5

Enter an integer: 42
⇒ (even 42)
Enter an integer: 149
⇒ (odd 149)
```

L1 では *expval*のバインディング (この場合は `num`) を `and`と *symbol*で “分解” している。L2 では前と同じ方法で `or`は始まるが、異なるシンボルにバインドするかわりに、両方のサブパターン内で同一のシンボル `spin`に回バインドするために 2 回 `let`を使用している (L3 から L4)。`spin` の値によりサブパターンは区別される。そして body フォームでは両方のシンボルを参照している (L5)。

11.4.2 `pcase`の拡張

`pcase`マクロは数種類のパターンをサポートします (Section 11.4 [Pattern-Matching Conditional], page 149 を参照)。`pcase-defmacro`を使用すれば違う種類のパターンにたいするサポートを追加できます。

`pcase-defmacro` *name args* [*doc*] **&rest** *body* [Macro]

`(`*name actual-args*`)` のように呼び出すために新たな種類の `pcase`用のパターンを定義する。`pcase`マクロは *body*を評価する呼び出しへと展開する。このマクロの役割は *args*を *actual-args*にバインドした環境下において、呼び出されたパターンを別の何らかのパターンに書き換えることである。

さらに `pcase`のドキュメント文字列とともに *doc*が表示されるように計らう。*doc*では慣例により *expression*の評価結果を示すために `EXPVAL`を使用すること。

*body*は通常はより基本的なパターンを使用して呼び出されたパターンを書き換える。最終的にはすべてのパターンはコアパターンに絞り込まれるが、`body`がすぐにコアパターンを使用する必要はない。以下の例では `less-than`と `integer-less-than`という 2 つのパターンを定義している。

```
(pcase-defmacro less-than (n)
  "Matches if EXPVAL is a number less than N."
  `(pred (> ,n)))

(pcase-defmacro integer-less-than (n)
  "Matches if EXPVAL is an integer less than N."
  `(and (pred integerp)
        (less-than ,n)))
```

args (このケースでは `n`の 1 つだけ) に言及する docstring は通常の方法、`EXPVAL`では慣例にもとづく方法であることに注意してください。1 つ目の書き換え (`less-than`の *body*) では `pred`というコアパターンだけが使用されていて、2 つ目では 2 つのコアパターン `and`と `pred`と新たに定義したパターン `less-than`が使用されています。そしていずれの書き換えにおいてもシングルバッククォート構文が使用されています (Section 10.3 [Backquote], page 139 を参照)。

11.4.3 バッククォートスタイルパターン

このサブセクションでは構造化マッチングを容易にするビルトインパターンバッククォートスタイルパターン *(backquote-style patterns)* について説明します。背景については Section 11.4 [Pattern-Matching Conditional], page 149 を参照してください。

バッククォートスタイルパターンは (`pcase-defmacro`を使用して作成された) 強力な `pcase`パターン拡張であり、その構造 *(structure)* の仕様にたいする *expval*をマッチを容易にします。

たとえば 1 つ目の要素が特定の文字列、2 つ目の要素が任意の値であるような 2 要素リストの *expval*にたいするマッチはコアパターンを使用して記述できます:

```
(and (pred listp)
     ls
     (guard (= 2 (length ls)))
     (guard (string= "first" (car ls)))
     (let second-elem (cadr ls)))
```

しかし等価なバッククォートスタイルパターンで記述することもできます:

```
`("first" ,second-elem)
```

バッククォートスタイルパターンはより簡潔かつ *expval*の構造と似ており、`ls`のバインドを要しません。

バッククォートスタイルパターンは`` `*qpat* ``のような形式をもちます。ここで *qpat*は以下の形式をもつことができます:

`(`*qpat1* `.` *qpat2*`)`
: *expval*が (`car`が *qpat1*、`cdr`が *qpat2*にマッチする) コンスセルならマッチ。`(`*qpat1 qpat2* `...)` のように容易に一般化できる。

`[`*qpat1 qpat2* `...` *qpatm*`]`
: *expval*が長さ *m*の (`0`から `(`*m*`-1)`番目の要素が *qpat1*、*qpat2*、...、*qpatm*にマッチする) ベクターならマッチ。

symbol
keyword
number
string
: *expval*の対応する要素が指定されたリテラルオブジェクトと `equal`ならマッチ。

`,`*pattern*
: *expval*の対応する要素が *pattern*にマッチすればマッチ。*pattern*は `pcase`がサポートするすべての種類のパターンであることに注意 (上記の例では `second-elem`は *symbol* コアパターンであり、これはすべてにマッチして `second-elem`を let でバインドする)。

対応する要素 *(corresponding element)* とはバッククォートスタイルパターン *qpat*にたいする構造的な位置に等しいような *expval*の構造的な位置部分のことです (上記の例では `second-elem`の対応する要素は *expval*の 2 つ目の要素)。

以下は小さな式言語用の単純なインタープリターの実装用に `pcase`を使用する例です (`body`と `arg`を正しくキャプチャーするには `fn`の clause 内で lambda 式にレキシカルバインディングが必要なことに注意):

```
(defun evaluate (form env)
  (pcase form
    (`(add ,x ,y)       (+ (evaluate x env)
                           (evaluate y env)))
```

```
(`(call ,fun ,arg)  (funcall (evaluate fun env)
                             (evaluate arg env)))
(`(fn ,arg ,body)   (lambda (val)
                      (evaluate body (cons (cons arg val)
                                           env))))
((pred numberp)     form)
((pred symbolp)     (cdr (assq form env)))
(_                  (error "Syntax error: %S" form))))
```

最初の 3 つの clause ではバッククォートスタイルパターンが使用されています。`(add ,x ,y)はformがリテラルシンボル addから始まる 3 要素リストであることをチェックしてから 2 つ目と 3 つ目の要素を取り出してシンボル xと yにバインドします。clause の body では xと yを評価して結果を加算します。同じように call clause は関数呼び出しを実装して、fn clause は無名関数定義を実装します。

残りの clause ではコアパターンが使用されています。(pred numberp)は formが数値ならマッチします。マッチした場合には body がそれを評価します。(pred symbolp)は formがシンボルならマッチします。マッチした場合には body は env内のシンボルを照合して、それの連想値をリターンします。最後の_はすべてにマッチする catch-all パターンなので、構文エラーの報告に適しています。

以下は評価した結果を含む、この小さな言語のサンプルプログラムの例です:

```
(evaluate '(add 1 2) nil)                 ⇒ 3
(evaluate '(add x y) '((x . 1) (y . 2)))  ⇒ 3
(evaluate '(call (fn x (add 1 x)) 2) nil) ⇒ 3
(evaluate '(sub 1 2) nil)                 ⇒ error
```

11.4.4 pcaseパターンによる分解

pcase のパターンはあるオブジェクトがマッチ可能なフォーム上の条件を表現するだけではなく、それらのオブジェクトのサブフィールドの抽出もできます。たとえば以下のコードにより、変数 my-list の値であるリストから 2 つの要素を抽出できます:

```
(pcase my-list
  (`(add ,x ,y)  (message "Contains %S and %S" x y)))
```

これは xと yを抽出すだけではなく、加えて my-listが正確に 3 つの要素を含むリストであり、最初の要素がシンボル addかどうかをテストします。これらのテストのいずれかが失敗したら、pcase は messageを呼び出さずに即座に nilをリターンします。

あるオブジェクトから格納された複数の値を抽出する処理は分割 *(destructuring)* という処理としても知られています。pcaseパターンの使用によりバインディングの分割 *(destructuring binding)* を処理することが可能になります。これはローカルバインディング (Section 12.3 [Local Variables], page 175 を参照) と似ていますが、互換性のあるオブジェクトから値を抽出することにより、変数の複数要素に値を与えることができます。

このセクションで説明したマクロはバインディングを分割するために pcaseパターンを使用しています。オブジェクト構造に互換性があるという条件は、そのオブジェクトがパターンにマッチしなければならないことを意味しています。なぜならマッチした場合のみオブジェクトのサブフィールドが抽出可能になるからです。たとえば:

```
(pcase-let ((`(add ,x ,y) my-list))
  (message "Contains %S and %S" x y))
```

これは最初に my-listが正しい個数の要素をもつリストであり、かつ 1 つ目の要素が addか検証せずに、my-listから直接 xと yの抽出を行う点を除いて前の例と同じことを行います。実際にオブジェ

クトがパターンにマッチしない場合には、たとえ body が暗黙にスキップされることはないとしても、その振る舞いは未定義でありエラーがシグナルされるか、あるいはいくつかの変数が `nil`のような任意の値にバインドされた状況で body が実行されるかもしれません。

バインディングの分割に有用な pcase パターンとしては、マッチされるオブジェクト構造の仕様を表現する Section 11.4.3 [Backquote Patterns], page 157 で説明したパターンが一般的です。

バインディングの分割にたいする代替え機能については [seq-let], page 106 を参照してください。

`pcase-let` *bindings body...* [Macro]

*bindings*に応じて変数のバインディング分割を行い、それから *body*を評価する。

*bindings*は `(pattern exp)` という形式のバインディングのリスト。ここで *exp*は評価する式、*pattern*は `pcase`パターン。

*exp*はすべて最初に評価されて、その後で対応する *pattern*にマッチされて、*body*の内部で使用可能な変数バインディングが導入される。この変数バインディングは *pattern*の要素を、評価された *exp*の対応する要素の値に分割してのバインディングすることにより生成される。

`pcase-let*` *bindings body...* [Macro]

*bindings*に応じて変数のバインディング分割を行い、それから *body*を評価する。

*bindings*は `(pattern exp)`という形式のバインディングのリスト。ここで *exp*は評価する式、*pattern*は `pcase`パターン。この変数バインディングは *pattern*の要素を、評価された *exp*の対応する要素の値に分割してのバインディングすることにより生成される。

`pcase-let`とは異なり (しかし `let*`と同じように)、各 *exp*は *bindings*の次要素の処理前に対応する *pattern*にたいしてマッチされるので、各 *bindings*のいずれかによって導入される変数バインディングは *body*内で利用可能になるのに加えて、その後に続く *bindings*の *exp*内で利用可能になる。

`pcase-dolist` (*pattern list*) *body...* [Macro]

繰り返しごとに *pattern*の変数を *list*の要素の対応するサブフィールドに分割バインディングしながら、*list*の各要素ごとに一度 *body*を実行するこのバインディングは `pcase-let`の場合のように行われる。*pattern*が単なる変数なら `dolist`と等価 (Section 11.5 [Iteration], page 159 を参照)。

11.5 繰り返し

繰り返し (iteration) とは、プログラムの一部を繰り返し実行することを意味します。たとえばリストの各要素、または 0 から *n*の整数にたいして、繰り返し一度ずつ何らかの計算を行いたいとしましょう。Emacs Lisp ではスペシャルフォーム `while`でこれを行なうことができます:

`while` *condition forms...* [Special Form]

`while`は最初に *condition*を評価する。結果が非 `nil`なら *forms*をテキスト順に評価する。その後に *condition*を再評価して結果が非 `nil`なら、再度 *forms*を評価する。この処理は *condition*が `nil`に評価されるまで繰り返される。

繰り返し回数に制限はない。このループは *condition*が `nil`に評価されるか、エラーになるか、または `throw`で抜け出す (Section 11.7 [Nonlocal Exits], page 162 を参照) まで継続される。

`while`フォームの値は常に `nil`である。

```
(setq num 0)
     ⇒ 0
```

```
(while (< num 4)
  (princ (format "Iteration %d." num))
  (setq num (1+ num)))
     ⊣ Iteration 0.
     ⊣ Iteration 1.
     ⊣ Iteration 2.
     ⊣ Iteration 3.
     ⇒ nil
```

各繰り返しごとに何かを実行して、その後も終了テストを行なう repeat-until ループを記述するには、以下のように `while`の 1 番目の引数として body の後に終了テストを記述して、それを `progn`の中に配置する:

```
(while (progn
         (forward-line 1)
         (not (looking-at "^$"))))
```

これは 1 行前方に移動して、空行に達するまで行単位の移動を継続する。独特な点は `while` が body をもたず、終了テスト (これはポイント移動という実処理も行なう) だけを行うことである。

マクロ `dolist`および `dotimes`は、2 つの一般的な種類のループを記述する、便利な方法を提供します。

`dolist` (*var list* [*result*]) *body*. . . [Macro]

この構文は *list*の各要素に一度 *body*を実行して、カレント要素をローカルに保持するように、変数 *var*にバインドする。その後に *result*を評価した値、*result*が省略された場合は `nil`をリターンする。たとえば以下は `reverse`関数を定義するために `dolist`を使用する方法の例である:

```
(defun reverse (list)
  (let (value)
    (dolist (elt list value)
      (setq value (cons elt value)))))
```

`dotimes` (*var count* [*result*]) *body*. . . [Macro]

この構文は 0 以上 *count*未満の各整数にたいして、一度 *body*を実行してから、繰り返しのカレント回数となる整数を変数 *var*にバインドする。その後に *result*の値、*result*が省略された場合は `nil`をリターンする。*result*の使用は推奨しない。以下は `dotimes`を使用して、何らかの処理を 100 回行なう例:

```
(dotimes (i 100)
  (insert "I will not obey absurd orders\n"))
```

11.6 ジェネレーター

ジェネレーター *(generator)* とは、潜在的に無限な値ストリームを生成する関数です。毎回その関数が値を生成するごとに、呼び出し側が次の値を要求するまで、自身をサスペンドします。

`iter-defun` *name args* [*doc*] [*declare*] [*interactive*] *body*. . . [Macro]

`iter-defun`はジェネレーター関数を定義する。ジェネレーター関数は通常の関数と同様の signature をもつが、異なるように機能する。ジェネレーター関数は呼び出し時に *body*を実

行するのではなく、かわりに iterator オブジェクトをリターンする。この iterator は値を生成するために *body*を実行、値を発行すると `iter-yield`か `iter-yield-from`が出現するまで一時停止する。*body*が正常にリターンした際に、`iter-next`がコンディションデータとなる*body*の結果とともに、`iter-end-of-sequence`をシグナルする。

*body*内部では任意の種類の Lisp コードが有効だが、`iter-yield`と `iter-yield-from`は `unwind-protect`フォームの内部にあってはならない。

`iter-lambda` *args* [*doc*] [*interactive*] *body*. . . [Macro]
`iter-lambda`は `iter-defun`で生成されたジェネレーター関数と同様な、無名のジェネレーター関数を生成する。

`iter-yield` *value* [Macro]
`iter-yield`がジェネレーター関数内部で出現した際には、カレント iterator が一時停止して `iter-next`から *value*をリターンすることを示す。`iter-yield`は、次回 `iter-next`呼び出しの `value`パラメーターへと評価される。

`iter-yield-from` *iterator* [Macro]
`iter-yield-from`は *iterator*が生成するすべての値を生成して、その *iterator*のジェネレーター関数が通常リターンする値へと評価される。これが制御を得ている間、*iterator*は `iter-next`を使用して送信された値を受け取る。

ジェネレーター関数を使用するには、まずそれを普通に呼び出して *iterator* オブジェクトを生成します。iterator はジェネレーターの固有のインスタンスです。その後でこの iterator から値を取得するために `iter-next`を使用します。iterator から取得する値がなくなると、`iter-next`はその iterator の最終値とともに `iter-end-of-sequence`のコンディションを raises します。

ジェネレーター関数の body は、`iter-next`の呼び出しの内側でのみ実行されることに注意することが重要です。`iter-defun`で定義された関数の呼び出しは iterator を生成します。何か興味があることが発生したら、`iter-next`でこの iterator を制御しなければなりません。ジェネレーター関数の個々の呼び出しは、それぞれが独自に状態をもつ別個の iterator を生成します。

`iter-next` *iterator value* [Function]
*iterator*から次の値を取得する。(*iterator*のジェネレーター関数がリターンしていて) 生成される値が存在しない場合、`iter-next`はコンディション `iter-end-of-sequence`をシグナルする。このコンディションに関連付けられるデータ値は、*iterator*のジェネレーター関数がリターンした値である。

*value*は iterator に送信されて、`iter-yield`を評価した値になる。*iterator*のジェネレーター関数の開始時には、ジェネレーター関数は `iter-yield`フォームを何も評価していないので、与えられた iterator にたいする最初の `iter-next`呼び出しでは *value*は無視される。

`iter-close` *iterator* [Function]
*iterator*が `unwind-protect`の `bodyform`フォーム内でサスペンドされていたら、ガーベージコレクション処理後に Emacs が最終的に unwind ハンドラーを実行する (`unwind-protect`の `unwindforms`内部では `iter-yield`は不当であることに注意)。その前に確実にこれらのハンドラーを実行するには、`iter-close`を使用すること。

iterator を簡単に連携できるように、便利な関数がいくつか提供されています:

`iter-do` (*var iterator*) *body* . . . [Macro]
*iterator*が生成する各値を *var*にバインドしつつ *body*を実行する。

Common Lisp のループ機能には iterator とともに機能も含まれています。Section “Loop Facility” in *Common Lisp Extensions* を参照してください。

以下のコード片は iterator との連携における重要な原則を示すものです。

```
(require 'generator)
(iter-defun my-iter (x)
  (iter-yield (1+ (iter-yield (1+ x))))
   ;; 普通にリターンする
  -1)

(let* ((iter (my-iter 5))
       (iter2 (my-iter 0)))
  ;; 6 をプリント
  (print (iter-next iter))
  ;; 9 をプリント
  (print (iter-next iter 8))
  ;; 1 をプリント
  ;; iter と iter は異なる状態をもつ
  (print (iter-next iter2 nil))

  ;; ここで iter シーケンスの終了を待機
  (condition-case x
      (iter-next iter)
    (iter-end-of-sequence
      ;; my-iter が通常の方法でリターンした-1 をプリント
      (print (cdr x)))))
```

11.7 非ローカル脱出

非ローカル脱出 *(nonlocal exit)* とは、プログラム内のある位置から別の離れた位置へ制御を移します。Emacs Lisp ではエラーの結果として非ローカル脱出が発生することがあります。明示的な制御の下で非ローカル脱出を使用することもできます。非ローカル脱出は脱出しようとしている構文により作成された、すべての変数バインディングのバインドを解消します。

11.7.1 明示的な非ローカル脱出: `catch`と`throw`

ほとんどの制御構造は、その構文自身の内部の制御フローだけに影響します。関数 `throw`は、この通常のプログラム実行でのルールの例外です。これはリクエストにより非ローカル脱出を行ないます (他にも例外はあるがそれらはエラー処理用のものだけ)。`throw`は `catch`の内部で使用され、`catch`に制御を戻します。たとえば:

```
(defun foo-outer ()
  (catch 'foo
    (foo-inner)))

(defun foo-inner ()
  ...
  (if x
      (throw 'foo t))
  ...)
```

throwフォームが実行されると、対応する catchに制御を移して、catchは即座にリターンします。throwの後のコードは実行されません。throwの 2 番目の引数は catchのリターン値として使用されます。

関数 throwは 1 番目の引数にもとづいて、それにマッチする catchを探します。throwは 1 番目の引数が、throwで指定されたものと eqであるような catchを検索します。複数の該当する catchがある場合には、最内のものが優先されます。したがって上記の例では throwが fooを指定していて、foo-outer内の catchが同じシンボルを指定しているので、(この間に他のマッチする catchは存在しないと仮定するなら) その catchが該当します。

throwの実行により、マッチする catchまでのすべての Lisp 構文 (関数呼び出しを含む) を脱出します。この方法により letや関数呼び出しのようなバインディング構文を脱出する場合には、これらの構文を正常に exit したときのように、そのバインディングは解消されます (Section 12.3 [Local Variables], page 175 を参照)。同様に throwは save-excursion(Section 30.3 [Excursions], page 765 を参照) によって保存されたバッファーと位置を復元します。throwがスペシャルフォーム unwind-protectを脱出した場合には、unwind-protectにより設定されたいくつかのクリーンアップも実行されます。

ジャンプ先となる catch内にレキシカル (局所的) である必要はありません。throwは catch内で呼び出された別の関数から、同じようにに呼び出すことができます。throwが行なわれたのが、時系列的に catchに入った後で、かつ exit する前である限り、その throwは catchにアクセスできます。エディターのコマンドループから戻る exit-recursive-editのようなコマンドで、throwが使用されるのはこれが理由です。

> **Common Lisp** に関する注意**:** Common Lisp を含む、他のほとんどのバージョンの Lisp は非シーケンシャルに制御を移すいくつかの方法 — たとえば return、return-from、go — をもつ。Emacs Lisp は throwのみ。cl-libライブラリーはこれらのうちいくつかを提供する。Section “Blocks and Exits” in *Common Lisp Extensions* を参照のこと。

`catch` *tag body...* [Special Form]

catchは throw関数にたいするリターン位置を確立する。リターン位置は *tag*により、この種の他のリターン位置と区別される。*tag*は nil以外の任意の Lisp オブジェクト。リターン位置が確立される前に、引数 *tag*は通常どおり評価される。

リターン位置が効果をもつことにより、catchは *body*のフォームをテキスト順に評価する。フォームが (エラーや非ローカル脱出なしで) 通常に実行されたなら、body の最後のフォームの値が catchからリターンされる。

*body*の実行の間に throwが実行された場合、*tag*と同じ値を指定すると catchフォームは即座に exit する。リターンされる値は、それが何であれ throwの 2 番目の引数に指定された値である。

`throw` *tag value* [Function]

throwの目的は、以前に catchにより確立されたリターン位置に戻ることである。引数 *tag*は、既存のさまざまなリターン位置からリターン位置を選択するために使用される。複数のリターン位置が *tag*にマッチしたら、最内のものが使用される。

引数 *value*は catchからリターンされる値として使用される。

タグ *tag*のリターン位置が存在しなければ、データ `(tag value)`とともに no-catchエラーがシグナルされます。

11.7.2 catchとthrowの例

2 重にネストされたループから脱出する 1 つの方法は、catchと throwを使うことです (これはほとんどの言語では gotoにより行なわれるだろう)。ここでは *i*と *j*を 0 から 9 に変化させて、(foo *i j*) を計算します:

```
(defun search-foo ()
  (catch 'loop
    (let ((i 0))
      (while (< i 10)
        (let ((j 0))
          (while (< j 10)
            (if (foo i j)
                (throw 'loop (list i j)))
            (setq j (1+ j))))
        (setq i (1+ i))))))
```

fooが非 nilをリターンしたら即座に処理を中止して、*i*と *j*のリストをリターンしています。fooが常に nilをリターンする場合には、catchは通常どおりリターンして、その値は whileの結果である nilとなります。

以下では 2 つのリターン位置を一度に表す、微妙に異なるトリッキーな例を 2 つ示します。まず同じタグ hackにたいして 2 つのリターン位置があります:

```
(defun catch2 (tag)
  (catch tag
    (throw 'hack 'yes)))
⇒ catch2

(catch 'hack
  (print (catch2 'hack))
  'no)
⊣ yes
⇒ no
```

どちらのリターン位置も throwにマッチするタグをもつので内側のもの、つまり catch2で確立された catch へ goto します。したがって catch2は通常どおり値 yesをリターンして、その値がプリントされます。最後に外側の catchの 2 番目の body、つまり 'noが評価されて外側の catchからそれがリターンされます。

ここで catch2に与える引数を変更してみましょう:

```
(catch 'hack
  (print (catch2 'quux))
  'no)
⇒ yes
```

この場合も 2 つのリターン位置がありますが、今回は外側だけがタグ hackで、内側はかわりにタグ quuxをもちます。したがって throwにより、外側の catchが値 yesをリターンします。関数 printが呼び出されることはなく body のフォーム 'noも決して評価されません。

11.7.3 エラー

Emacs Lisp が何らかの理由で評価できないようなフォームの評価を試みると、エラー *(error)* がシグナル *(signal)* されます。

エラーがシグナルされるとエラーメッセージを表示して、カレントコマンドの実行を終了するのがEmacs デフォルトの反応です。たとえばバッファーの最後で *C-f*とタイプしたときのように、ほとんどの場合にはこれは正しい反応になります。

複雑なプログラムでは単なる終了が望ましくない場合もあるでしょう。たとえばそのプログラムがータ構造に一時的に変更を行なっていたり、プログラム終了前に削除する必要がある一時バッファーを作成しているかもしれません。このような場合には、エラー時に評価されるクリーンアップ式 *(cleanup expressions)* を設定するために、`unwind-protect`を使用するでしょう (Section 11.7.4 [Cleanups], page 172 を参照)。サブルーチン内のエラーにもかかわらずに、プログラムの実行を継続したいときがあるかもしれません。このような場合には、エラー時のリカバリーを制御するエラーハンドラー *(error handlers)* を設定するために `condition-case`を使用するでしょう。

エラーハンドラーを使用せずにプログラムの一部から別の部分へ制御を移すためには、`catch`と`throw`を使用します。Section 11.7.1 [Catch and Throw], page 162 を参照してください。

11.7.3.1 エラーをシグナルする方法

エラーのシグナリング *(signaling)* とは、エラーの処理を開始することを意味します。エラー処理は通常は実行中のプログラムのすべて、または一部をアボート (abort) してエラーをハンドルするためにセットアップされた位置にリターンします。ここではエラーをシグナルする方法を記述します。

ほとんどのエラーは、たとえば整数にたいして CAR の取得を試みたり、バッファーの最後で 1 文字前方に移動したときなどのように、他の目的のために呼び出した Lisp プリミティブ関数の中で自動的にシグナルされます。関数 `error`と `signal`で明示的にエラーをシグナルすることもできます。

ユーザーが *C-g*をタイプしたときに発生する quit はエラーとは判断されませんが、ほとんどはエラーと同様に扱われます。Section 21.11 [Quitting], page 413 を参照してください。

すべてのエラーメッセージはそれぞれ、何らかのエラーメッセージを指定します。そのメッセージは何が悪いのか (“File does not exist”)、物事がどうしてそうあるべきではない (“File must exist”) かを示すべきです。Emacs Lisp の慣習ではエラーメッセージは大文字で開始され、区切り文字で終わるべきではありません。

`error` *format-string* **&rest** *args* [Function]

この関数は *format-string*と *args*にたいして、`format-message` (Section 4.7 [Formatting Strings], page 61 を参照) を適用して構築されたエラーメッセージとともに、エラーをシグナルする。

以下は `error`を使用する典型的な例である:

```
(error "That is an error -- try something else")
     error  That is an error -- try something else

(error "Invalid name `%s'" "A%%B")
     error  Invalid name ‘A%%B’
```

2 つの引数 — エラーシンボル `error`と `format-message`がリターンするる文字列を含むリスト — で `signal`を呼び出すことにより `error`は機能する。

`"Missing ‘%s’"`が`"Missing ‘foo’"`となるように、通常はフォーマット内の grave accent と apostrophe はマッチする curved quotes に変換される。この変換の効果や抑制については Section 24.4 [Text Quoting Style], page 524 を参照のこと。

警告: エラーメッセージとして固定の文字列を使用したい場合、単に `(error` *`string`*`)`とは記述しないこと。もし *string*が‘`%`’、‘`` ` ``’、‘`'`’を含んでいると、再フォーマットされて望む結果は得られないだろう。かわりに、`(error "%s"` *`string`*`)`を使用すること。

`signal` *error-symbol data* [Function]

この関数は *error-symbol*で命名されるエラーをシグナルする。引数 *data*はエラー状況に関連する追加の Lisp オブジェクトのリスト。

引数 *error-symbol*はエラーシンボル *(error symbol)* — `define-error`で定義されたシンボル — でなければならない。これは Emacs Lisp が異なる種類のエラーをクラス分けする方法である。エラーシンボル (error symbol)、エラーコンディション (error condition)、コンディション名 (condition name) の説明については Section 11.7.3.4 [Error Symbols], page 171 を参照のこと。

エラーが処理されない場合には、エラーメッセージをプリントするために 2 つの引数が使用される。このエラーメッセージは通常、*error-symbol*の `error-message`プロパティーにより提供される。*data*が非 `nil`なら、その後にコロンと *data*の未評価の要素をカンマで区切ったリストが続く。`error`にたいするエラーメッセージは *data*の CAR である (文字列であること)。サブカテゴリー `file-error`は特別に処理される。

*data*内のオブジェクトの数と意味は *error-symbol*に依存する。たとえば `wrong-type-argument`エラーではリスト内に 2 つのオブジェクト— 期待する型を記述する述語とその型への適合に失敗したオブジェクト — であること。

エラーを処理する任意のエラーハンドラーにたいして *error-symbol*と *data*の両方を利用できる。`condition-case`はローカル変数を `(`*`error-symbol . data`*`)`というフォームでバインドする (Section 11.7.3.3 [Handling Errors], page 167 を参照)。

関数 `signal`は決してリターンしない。

```
(signal 'wrong-number-of-arguments '(x y))
     error  Wrong number of arguments: x, y

(signal 'no-such-error '("My unknown error condition"))
     error  peculiar error: "My unknown error condition"
```

`user-error` *format-string* **&rest** *args* [Function]

この関数は、`error`とまったく同じように振る舞うが、`error`ではなくエラーシンボル `user-error`を使用する。名前が示唆するように、このエラーはコード自身のエラーではなく、ユーザー側のエラーの報告を意図している。たとえば Info の閲覧履歴の開始を超えて履歴を遡るためにコマンド `Info-history-back` (`l`) を使用した場合、Emacs は `user-error`をシグナルする。このようなエラーでは、たとえ `debug-on-error`が非 `nil`であっても、デバッガーへのエントリーは発生しない。Section 18.1.1 [Error Debugging], page 296 を参照のこと。

Common Lisp に関する注意**:** Emacs Lisp には Common Lisp のような継続可能なエラーのような概念は存在しない。

11.7.3.2 Emacs がエラーを処理する方法

エラーがシグナルされたとき、`signal`はそのエラーにたいするアクティブなハンドラー *(handler)* を検索します。ハンドラーとは、Lisp プログラムの一部でエラーが発生したときに実行するよう意図された Lisp 式のシーケンスです。そのエラーが適切なハンドラーをもっていればそのハンドラーが実行され、そのハンドラーの後から実行が再開されます。ハンドラーはそのハンドラーが設定された `condition-case`の環境内で実行されます。`condition-case`内のすべての関数呼び出しはすでに終了しているので、ハンドラーがそれらにリターンすることはありません。

そのエラーにたいする適切なハンドラーが存在しなければ、カレントコマンドを終了してエディターのコマンドループに制御をリターンします (コマンドループにはすべての種類のエラーにたいす

る暗黙のハンドラーがある)。コマンドループのハンドラーは、エラーメッセージをプリントするためにエラーシンボルと、それに関連付けられたデータを使用します。変数 `command-error-function` を使用して、これが行なわれる方法を制御できます:

`command-error-function` [Variable]
この変数が非`nil`なら、それはEmacsのコマンドループに制御をリターンしたエラーの処理に使用する関数を指定する。この関数は3つの引数を受け取る。1つ目の *data*は、`condition-case` が自身の変数にバインドするのと同じフォーム。2つ目の *context*はエラーが発生した状況を記述する文字列か、(大抵は)`nil`。3つ目の *caller*はエラーをシグナルしたプリミティブ関数を呼び出したLisp関数。

明示的なハンドラーがないエラーは、Lispデバッガーを呼び出すかもしれません。変数`debug-on-error` (Section 18.1.1 [Error Debugging], page 296 を参照) が非`nil`ならデバッガーが有効です。エラーハンドラーと異なり、デバッガーはそのエラーの環境内で実行されるので、エラー時の変数の値を正確に調べることができます。

11.7.3.3 エラーを処理するコードの記述

エラーをシグナルすることによる通常の効果は、実行されていたコマンドを終了してEmacsエディターのコマンドループに即座にリターンすることです。スペシャルフォーム `condition-case`を使用してエラーハンドラーを設定することにより、プログラム内の一部で発生するエラーのをトラップを調整することができます。以下は単純な例です:

```
(condition-case nil
    (delete-file filename)
  (error nil))
```

これは *filename*という名前のファイルを削除して、任意のエラーをcatch、エラーが発生した場合は`nil`をリターンします (このような単純なケースではマクロ `ignore-errors`を使用することもできる。以下を参照のこと)。

`condition-case`構文は、`insert-file-contents`呼び出しによるファイルオープンの失敗のような、予想できるエラーをトラップするために多用されます。`condition-case`構文はユーザーから読み取った式を評価するプログラムのような、完全には予測できないエラーのトラップにも使用されます。

`condition-case`の2番目の引数は保護されたフォーム *(protected form)* と呼ばれます (上記の例では保護されたフォームは `delete-file`の呼び出し)。このフォームの実行が開始されるとエラーハンドラーが効果をもち、このフォームがリターンすると不活性になります。その間のすべてにおいてエラーハンドラーは効果をもちます。特にこのフォームで呼び出された関数とそのサブルーチン等を実行する間、エラーハンドラーは効果をもちます。厳密にいうと保護されたフォーム自身ではなく、保護されたフォームから呼び出されたLispプリミティブ関数 (`signal`と `error`を含む) だけがシグナルされるというのは、よいことと言えます。

保護されたフォームの後の引数はハンドラーです。各ハンドラーはそれぞれ、どのエラーを処理するかを指定する1つ以上のコンディション名 (シンボル) をリストします。エラーがシグナルされたとき、エラーシンボルはコンディション名のリストも定義します。エラーが共通のコンディション名をもつ場合、そのハンドラーがそのエラーに適用されます。上記の例では1つのハンドラーがあり、それはすべてのエラーをカバーするコンディション名 `error`を指定しています。

適切なハンドラーの検索は、もっとも最近に設定されたハンドラーから始まり、設定されたすべてのハンドラーをチェックします。したがってネストされた `condition-case`フォームに同じエラー処理がある場合には、内側のハンドラーがそれを処理します。

何らかの condition-caseによりエラーが処理されると、debug-on-errorでエラーによりデバッガーが呼び出されるようにしていても、通常はデバッガーの実行が抑制されます。

condition-caseで補足されるようなエラーをデバッグできるようにしたいなら、変数debug-on-signalに非nil値をセットします。以下のようにコンディション内にdebugを記述することにより、最初にデバッガーを実行するような特定のハンドラーを指定することもできます:

```
(condition-case nil
    (delete-file filename)
  ((debug error) nil))
```

ここでの debugの効果は、デバッガー呼び出しを抑制する condition-caseを防ぐことだけです。debug-on-errorとその他のフィルタリングメカニズムがデバッガーを呼び出すように指定されているときだけ、エラーによりデバッガーが呼び出されます。Section 18.1.1 [Error Debugging], page 296を参照してください。

condition-case-unless-debug *var protected-form handlers...* [Macro]
: マクロ condition-case-unless-debugは、そのようなフォームのデバッギングを処理する、別の方法を提供する。このマクロは変数 debug-on-errorが nil、つまり任意のエラーを処理しないようなケース以外は、condition-caseとまったく同様に振る舞う。

特定のハンドラーがそのエラーを処理すると Emacs が判断すると、Emacs は制御をそのハンドラーに return します。これを行うために、Emacs はそのとき脱出しつつあるバインディング構成により作成されたすべての変数のバインドを解き、そのとき脱出しつつあるすべての unwind-protectフォームを実行します。制御がそのハンドラーに達すると、そのハンドラーの body が通常どおり実行されます。

そのハンドラーの body を実行した後、condition-caseフォームから実行が return されます。保護されたフォームは、そのハンドラーの実行の前に完全に exit しているので、そのハンドラーはそのエラーの位置から実行を再開することはできず、その保護されたフォーム内で作られた変数のバインディングを調べることもできません。ハンドラーが行なえることは、クリーンアップと、処理を進行させることだけです。

エラーのシグナルとハンドルにはthrowと catch (Section 11.7.1 [Catch and Throw], page 162を参照) に類似する点がいくつかありますが、これらは完全に別の機能です。エラーは catchでキャッチできず、throwをエラーハンドラーで処理することはできません (しかし対応する catchが存在しないときに throwを使用することによりシグナルされるエラーは処理できる)。

condition-case *var protected-form handlers...* [Special Form]
: このスペシャルフォームは *protected-form*の実行を囲い込むエラーハンドラー *handlers*を確立する。エラーなしで *protected-form*が実行されると、リターンされる値は condition-caseフォームの値になる。この場合、condition-caseは効果をもたない。*protected-form*の間にエラーが発生すると、condition-caseフォームは違いを生じる。

  *handlers*はそれぞれ、(*conditions body...*)というフォームのリストである。ここで *conditions*はハンドルされるエラーコンディション名、またはそのハンドラーの前にデバッガーを実行するためのコンディション名 (debugを含む)。tというコンディション名はすべてのコンディションにマッチする。*body*はこのハンドラーがエラーを処理するときに実行される 1 つ以上の Lisp 式。

```
(error nil)

(arith-error (message "Division by zero"))

((arith-error file-error)
 (message
  "Either division by zero or failure to open a file"))
```

発生するエラーはそれぞれ、それが何の種類のエラーかを記述するエラーシンボル *(error symbol)* をもち、これはコンディション名のリストも記述する (Section 11.7.3.4 [Error Symbols], page 171 を参照)。Emacs は 1 つ以上のコンディション名を指定するハンドラーにたいして、すべてのアクティブな `condition-case`フォームを検索する。`condition-case`の最内のマッチがそのエラーを処理する。`condition-case`内では、最初に適合したハンドラーがそのエラーを処理する。

ハンドラーの body を実行した後、`condition-case`は通常どおりリターンして、ハンドラーの body の最後の値をハンドラー全体の値として使用する。

引数 *var*は変数である。*protected-form*を実行するとき、`condition-case`はこの変数をバインドせず、エラーを処理するときだけバインドする。その場合には、*var*をエラー記述 *(error description)* にバインドする。これはエラーの詳細を与えるリストである。このエラー記述は `(`*`error-symbol . data`*`)`というフォームをもつ。ハンドラーは何を行なうか決定するために、このリストを参照することができる。たとえばファイルオープンの失敗にたいするエラーなら、ファイル名が *data*(エラー記述の 3 番目の要素) の 2 番目の要素になる。

*var*が `nil`なら、それはバインドされた変数がないことを意味する。この場合、エラーシンボルおよび関連するデータは、そのハンドラーでは利用できない。

より外側のレベルのハンドラーに catch させるために、`condition-case`により catch されたシグナルを再度 throw する必要がある場合もある。以下はこれを行なう方法である:

```
(signal (car err) (cdr err))
```

ここで `err`はエラー記述変数 (error description variable) で、`condition-case`の 1 番目の引数は、再 throw したいエラーコンディション。[Definition of signal], page 166 を参照のこと。

`error-message-string` *error-descriptor* [Function]

この関数は与えられたエラー記述子 (error descriptor) にたいするエラーメッセージ文字列をリターンする。これはそのエラーにたいする通常のエラーメッセージをプリントすることにより、エラーを処理したい場合に有用。[Definition of signal], page 166 を参照のこと。

以下は 0 除算の結果によるエラーを処理するために、`condition-case`を使用する例です。このハンドラーは、(beep なしで) エラーメッセージを表示して、非常に大きい数をリターンします。

```
(defun safe-divide (dividend divisor)
  (condition-case err
      ;; 保護されたフォーム
      (/ dividend divisor)
    ;; ハンドラー
    (arith-error                        ; コンディション
     ;; このエラーにたいする、通常のメッセージを表示する
     (message "%s" (error-message-string err))
     1000000)))
⇒ safe-divide
```

```
(safe-divide 5 0)
     ⊣ Arithmetic error: (arith-error)
⇒ 1000000
```

このハンドラーはコンディション名 arith-errorを指定するので、division-by-zero(0 除算) エラーだけを処理します。他の種類のエラーは (この condition-caseによっては)、処理されません。したがって:

```
(safe-divide nil 3)
     error  Wrong type argument: number-or-marker-p, nil
```

以下は errorによるエラーを含む、すべての種類のエラーを catch する condition-caseです:

```
(setq baz 34)
     ⇒ 34

(condition-case err
    (if (eq baz 35)
        t
      ;; 関数 errorの呼び出し
      (error "Rats!  The variable %s was %s, not 35" 'baz baz))
  ;; フォームではないハンドラー
  (error (princ (format "The error was: %s" err))
         2))
⊣ The error was: (error "Rats!  The variable baz was 34, not 35")
⇒ 2
```

ignore-errors *body...* [Macro]

この構文は、それの実行中に発生する任意のエラーを無視して *body*を実行する。その実行中にエラーがなければ、ignore-errorsは *body*内の最後のフォームの値を、それ以外は nilをリターンする。

以下はこのセクションの最初の例を ignore-errorsを使用して記述する例である:

```
(ignore-errors
 (delete-file filename))
```

ignore-error *condition body...* [Macro]

このマクロは ignore-errorsと同様だが、指定した特定のエラーコンディションだけを無視する点が異なる。

```
(ignore-error end-of-file
  (read ""))
```

*condition*はエエラーコンディションのリストでも可。

with-demoted-errors *format body...* [Macro]

このマクロはいわば ignore-errorsの穏やかなバージョンである。これはエラーを完全に抑止するのではなく、エラーをメッセージに変換する。これはメッセージのフォーマットに、文字列 *format*を使用する。*format*は"Error: %S"のように、単一の '%'シーケンスを含むこと。エラーをシグナルするとは予測されないが、もし発生した場合は堅牢であるべきようなコードの周囲に with-demoted-errorsを使用する。このマクロは condition-caseではなく、condition-case-unless-debugを使用することに注意。

11.7.3.4 エラーシンボルとエラー条件

エラーをシグナルするとき、想定するエラーの種類を指定するためにエラーシンボル *(error symbol)* を指定します。エラーはそれぞれ、それをカテゴリー分けするために単一のエラーシンボルをもちます。これは Emacs Lisp 言語で定義されるエラーを分類する、もっともよい方法です。

これらの狭義の分類はエラー条件 *(error conditions)* と呼ばれる、より広義のクラス階層にグループ化され、それらはコンディション名 *(condition names)* により識別されます。そのようなもっとも狭義なクラスは、エラーシンボル自体に属します。つまり各エラーシンボルは、コンディション名でもあるのです。すべての種類のエラー (`quit`を除く) を引き受けるコンディション名 `error`に至る、より広義のクラスにたいするコンディション名も存在します。したがって各エラーは 1 つ以上のコンディション名をもちます。つまり `error`、`error`とは区別されるエラーシンボル、もしかしたらその中間に分類されるものかもしれません。

`define-error` *name message* **&optional** *parent* [Function]
: シンボルをエラーシンボルとするために、シンボルは親コンディションを受け取る `define-error`で定義されなければならない。この親はその種のエラーが属するコンディションを定義する。親の推移的な集合は、常にそのエラーシンボルとシンボル `error`を含む。quit はエラーと判断されないので、`quit`の親の集合は単なる `(quit)`である。

親のコンディションに加えてエラーシンボルはメッセージ *(message)*をもち、これは処理されないエラーがシグナルされたときプリントされる文字列です。そのメッセージが有効でなければ、エラーメッセージ ‘`peculiar error`’が使用されます。[Definition of signal], page 166 を参照してください。

内部的には親の集合はエラーシンボルの `error-conditions`プロパティーに格納され、メッセージはエラーシンボルの `error-message`プロパティーに格納されます。

以下は新しいエラーシンボル `new-error`を定義する例です:

```
(define-error 'new-error "A new error" 'my-own-errors)
```

このエラーは複数のコンディション名 — もっとも狭義の分類 `new-error`、より広義の分類を想定する `my-own-errors`、および `my-own-errors`のコンディションすべてを含む `error`であり、これはすべての中でもっとも広義なものです。

エラー文字列は大文字で開始されるべきですが、ピリオドで終了すべきではありません。これは Emacs の他の部分との整合性のためです。

もちろん Emacs 自身が `new-error`をシグナルすることはありません。あなたのコード内で明示的に `signal` ([Definition of signal], page 166 を参照) を呼び出すことにより、これを行なうことができます。

```
(signal 'new-error '(x y))
     error  A new error: x, y
```

このエラーは、エラーの任意のコンディション名により処理することができます。以下の例は `new-error`とクラス `my-own-errors`内の他の任意のエラーを処理します:

```
(condition-case foo
    (bar nil t)
  (my-own-errors nil))
```

エラーが分類される有効な方法はコンディション名による方法で、その名前はハンドラーのエラーのマッチに使用されます。エラーシンボルは意図されたエラーメッセージと、コンディション名のリストを指定する便利な方法であるという役割をもつだけです。1 つのエラーシンボルではなく、コンディション名のリストを `signal`に与えるのは面倒でしょう。

対照的にコンディション名を伴わずにエラーシンボルだけを使用すると、それは`condition-case`の効果を著しく減少させるでしょう。コンディション名はエラーハンドラーを記述するとき、一般性のさまざまなレベルにおいて、エラーをカテゴリー分けすることを可能にします。エラーシンボルを単独で使用することは、もっとも狭義なレベルの分類を除くすべてを捨ててしまうことです。

主要なエラーシンボルとそれらのコンディションについては、Appendix F [Standard Errors], page 1216 を参照してください。

11.7.4 非ローカル脱出のクリーンアップ

`unwind-protect`構文は、データ構造を一時的に不整合な状態に置くときに重要です。これはエラーや throu のイベントにより、再びデータを整合された状態にすることができます (バッファー内容の変更だけに使用される他のクリーンアップ構成はアトミックな変更グループである。Section 32.31 [Atomic Changes], page 848 を参照)。

`unwind-protect` *body-form cleanup-forms...* [Special Form]

`unwind-protect`は制御が *body-form*を離れる場合に、*cleanup-forms*が評価されるという保証の下において、何が起こったかに関わらず *body-form*を実行する。*body-form*は通常どおり完了するかもしれず、`unwind-protect`の外側で`throw`の実行やエラーが発生するかもしれないが、*cleanup-forms*は評価される。

*body-form*が正常に終了したら、`unwind-protect`は *cleanup-forms*を評価した後に、*body-form*の値をリターンする。*body-form*が終了しなかったら、`unwind-protect`は通常の意味におけるような値はリターンしない。

`unwind-protect`で保護されるのは *body-form*だけである。*cleanup-forms*自体の任意のフォームが、(`throw`またはエラーにより) 非ローカルに exit すると、`unwind-protect`は残りのフォームが評価されることを保証しない。*cleanup-forms*の中の 1 つが失敗することが問題となるようなら、そのフォームの周囲に他の `unwind-protect`を配置して保護すること。

現在アクティブな `unwind-protect`フォーム数とローカルの変数バインディング数の和は、`max-specpdl-size` ([Local Variables], page 177 を参照) により制限される。

たとえば以下は一時的な使用のために不可視のバッファーを作成して、終了する前に確実にそのバッファーを kill する例です:

```
(let ((buffer (get-buffer-create " *temp*")))
  (with-current-buffer buffer
    (unwind-protect
        body-form
      (kill-buffer buffer))))
```

`(kill-buffer (current-buffer))`のように記述して、変数`buffer`を使用せずに同様のことを行えると思うかもしれません。しかし上の例は、別のバッファーにスイッチしたときに *body-form*でエラーが発生した場合、より安全なのです (一時的なバッファーを kill するとき、そのバッファーがカレントとなることを確実にするために、かわりに *body-form*の周囲に `save-current-buffer`を記述することもできる)。

Emacs には上のコードとおおよそ等しいコードに展開される、`with-temp-buffer`という標準マクロが含まれます ([Current Buffer], page 592 を参照)。このマニュアル中で定義されるいくつかのマクロは、この方法で `unwind-protect`を使用します。

以下は FTP パッケージ由来の実例です。これはリモートマシンへの接続の確立を試みるために、プロセス (Chapter 38 [Processes], page 934 を参照) を作成しています。関数`ftp-login`は関数のライター (writer) が予想できないことによる多くの問題から非常に影響を受けるので、失敗イベント

でプロセスの削除を保証するフォームで保護されています。そうしないと Emacs は無用なサブプロセスで一杯になってしまうでしょう。

```
(let ((win nil))
  (unwind-protect
      (progn
        (setq process (ftp-setup-buffer host file))
        (if (setq win (ftp-login process host user password))
            (message "Logged in")
          (error "Ftp login failed")))
    (or win (and process (delete-process process)))))
```

この例には小さなバグがあります。ユーザーが quit するために *C-g*とタイプすると、関数`ftp-setup-buffer`のリターン後に即座に quit が発生しますが、それは変数`process`がセットされる前なので、そのプロセスは kill されないでしょう。このバグを簡単に訂正する方法はありませんが、少なくともこれは非常に稀なことだと言えます。

12 変数

変数 *(variable)* とはプログラム内で値を表すために使用される名前です。Lisp では変数はそれぞれ Lisp シンボルとして表されます (Chapter 9 [Symbols], page 124 を参照)。変数名は単にそのシンボルの名前であり、変数の値はそのシンボルの値セル (value cell) に格納されます [1]。Section 9.1 [Symbol Components], page 124 を参照してください。Emacs Lisp ではシンボルを変数として使用することは、同じシンボルを関数名として使用することと関係ありません。

このマニュアルで前述したとおり、Lisp プログラムはまず第 1 に Lisp オブジェクトとして表され、副次的にテキストとして表現されます。Lisp プログラムのテキスト的な形式は、そのプログラムを構成する Lisp オブジェクトの入力構文により与えられます。したがって Lisp プログラム内の変数のテキスト的な形式は、その変数を表すシンボルの入力構文を使用して記述されます。

12.1 グローバル変数

変数を使用するための一番シンプルな方法は、グローバル *(globally)* を使用する方法です。これはある時点でその変数はただ 1 つの値をもち、その値が (少なくともその時点では)Lisp システム全体で効果をもつことを意味します。あらたな値を指定するまでその値が効果をもちます。新しい値で古い値を置き換えるとき、古い値を追跡する情報は変数内に残りません。

シンボルの値は `setq`で指定します。たとえば、

```
(setq x '(a b))
```

これは変数 `x`に値 `(a b)`を与えます。`setq`はスペシャルフォームであることに注意してください。これは 1 番目の引数 (変数の名前) は評価しませんが、2 番目の引数 (新しい値) は評価します。

変数が一度値をもつと、そのシンボル自身を式として使用することによって参照することができます。したがって、

```
x ⇒ (a b)
```

これは上記の `setq`フォームが実行された場合です。

同じ変数を再びセットすると、古い値は新しい値で置き換えられます:

```
x
     ⇒ (a b)
(setq x 4)
     ⇒ 4
x
     ⇒ 4
```

12.2 変更不可な変数

Emacs Lisp では特定のシンボルは、通常は自分自身に評価されます。これらのシンボルには `nil`と `t`、同様に名前が‘`:`’で始まる任意のシンボル (これらはキーワードと呼ばれる) が含まれます。これらのシンボルはリバインドや、値の変更はできません。`nil`や `t`へのセットやリバインドは、`setting-constant` エラーをシグナルします。これはキーワード (名前が‘`:`’で始まるシンボル) についても当てはまりま

[1] 正確に言うとデフォルトのダイナミックスコープ *(dynamic scoping)* のルールでは、値セルは常にその変数のカレント値を保持しますが、レキシカルスコープ *(lexical scoping)* では異なります。詳細は Section 12.10 [Variable Scoping], page 185 を参照してください。

す。ただしキーワードが標準の obarray に intern されていれば、そのようなシンボルを自分自身にセットしてもエラーになりません。

```
nil ≡ 'nil
     ⇒ nil
(setq nil 500)
error  Attempt to set constant symbol: nil
```

`keywordp` *object* [Function]

この関数は *object*が‘`:`’で始まる名前のシンボルであり、標準の obarray に intern されていれば `t`、それ以外は `nil`をリターンする。

これらの定数はスペシャルフォーム `defconst`(Section 12.5 [Defining Variables], page 178 を参照) を使用して定義された定数 (constant) とは根本的に異なります。`defconst`フォームは、人間の読み手に値の変更を意図しない変数であることを知らせる役目は果たしますが、実際にそれを変更しても Emacs はエラーを起こしません。

現実的な種々の理由により、追加で少数のシンボルが読み取り専用になります。これらには `enable-multibyte-characters`、`most-positive-fixnum`、`most-negative-fixnum`の他にいくつかのシンボルが含まれます。これらにたいしてセットやバインドを試みると、すべて `setting-constant`エラーがシグナルされます。

12.3 ローカル変数

グローバル変数は新しい値で明示的に置き換えるまで値が持続します。変数にローカル値 *(local value)* — Lisp プログラム内の特定の部分で効果をもつ — を与えると便利なときがあります。変数がローカル値をもつとき、わたしたちは変数がその値にローカルにバインド *(locally bound)* されていると言い、その変数をローカル変数 *(local variable)* と呼びます。

たとえば関数が呼び出されるとき、関数の引数となる変数はローカル値 (その関数の呼び出しにおいて実際の引数に与えられた値) を受け取ります。これらのローカルバインディングは関数の body 内で効果をもちます。他にもたとえばスペシャルフォーム `let`は特定の変数にたいして明示的にローカルなバインディングを確立して、これは `let`フォームの body 内だけで効果を持ちます。

これにたいしてグローバルなバインディング *(global binding)* とは、(概念的には) グローバルな値が保持される場所です。

ローカルバインディングを確立すると、その変数の以前の値は他の場所に保存されます (または失われる)。わたしたちはこれを、以前の値がシャドー *(shadowed)* されたと言います。シャドーはグローバル変数とローカル変数の両方で発生し得ます。ローカルバインディングが効果を持つときには、ローカル変数に `setq`を使用することにより、指定した値をローカルバインディングに格納します。ローカルバインディングが効果を持たなくなったとき、以前にシャドーされた値が復元されます (または失われる)。

変数は同時に複数のローカルバインディングを持つことができます (たとえばその変数をバインドするネストされた `let`)。カレントバインディング *(current binding)* とは、実際に効果を持つローカルバインディングのことです。カレントバインディングは、その変数の評価によりリターンされる値を決定し、`setq`により影響を受けるバインディングです。

ほとんどの用途において、最内 (innermost) のローカルバインディングとローカルバインディングをもたないグローバルバインディングを、カレントバインディングと考えることができます。より正確に言うと、スコープルール *(scoping rule)* と呼ばれるルールは、プログラム内でローカルバインディングが効果を持つ任意の与えられた場所を決定します。Emacs Lisp のスコープルールはダイナ

ミックスコープ *(dynamic scoping)* と呼ばれ、これは単に実行中のプログラム内の与えられた位置でのカレントバインディングを示しており、その変数がまだ存在すれば、その変数にたいしてもっとも最近作成されたバインディングです。ダイナミックスコープについての詳細、およびその代替であるレキシカルスコープ *(lexical scoping)* と呼ばれるスコープルールについては、Section 12.10 [Variable Scoping], page 185 を参照してください。

スペシャルフォーム `let`と `let*`は、ローカルバインディングを作成するために存在します:

`let` (*bindings...*) *forms...* [Special Form]

このスペシャルフォームは *bindings*により指定される特定の変数セットにたいするローカルバインディングをセットアップしてから、*forms*のすべてをテキスト順に評価する。これは *forms* 内の最後のフォームの値をリターンする。`let`がセットアップしたローカルバインディングは *forms*の body 内でのみ効果をもつ。

*bindings*の各バインディングは 2 つの形式のいずれかである。(i) シンボルなら、そのシンボルは `nil`にローカルにバインドされる。(ii) フォーム (*symbol value-form*)のリストなら、*symbol*は *value-form*を評価した結果へローカルにバインドされる。*value-form*が省略されたら `nil`が使用される。

*bindings*内のすべての *value-form*は、シンボルがそれらにバインドされる前に、記述された順番に評価される。以下の例では `z`は `y`の新しい値 (つまり 1) にではなく、古い値 (つまり 2) にバインドされる。

```
(setq y 2)
     ⇒ 2

(let ((y 1)
      (z y))
  (list y z))
     ⇒ (1 2)
```

その一方で *bindings* の順序は指定されない。以下の例では 1 か 2 のどちらかがプリントされる。

```
(let ((x 1)
      (x 2))
  (print x))
```

したがって単一の `let`フォーム内で変数を複数回バインディングするのは避けること。

`let*` (*bindings...*) *forms...* [Special Form]

このスペシャルフォームは `let`と似ているが、次の変数値にたいするローカル値を計算する前に、ローカル値を計算してそれを変数にバインドする。したがて *bindings*内の式は、この `let*` フォーム内の前のシンボルのバインドを参照できる。以下の例を上記 `let`の例と比較されたい。

```
(setq y 2)
     ⇒ 2

(let* ((y 1)
       (z y))    ; yの値に今計算されたばかりの値を使用する
  (list y z))
     ⇒ (1 1)
```

`letrec` (*bindings...*) *forms...* [Special Form]

このスペシャルフォームは `let*`と同様だが、ローカル値を計算する前にすべての変数をバインドする。値はその後にバインドされた変数にローカルに割り当てられる。これはレキシカル

バインディングが効力をもち、let*の使用しても有効にならないバインディングを参照するクロージャを作成したい場合のみ有用。

たとえば以下は一度実行後にフックから自身を削除するクロージャ:

```
(letrec ((hookfun (lambda ()
                    (message "Run once")
                    (remove-hook 'post-command-hook hookfun))))
  (add-hook 'post-command-hook hookfun))
```

以下はローカルバインディングを作成する他の機能のリストです:

- 関数呼び出し (Chapter 13 [Functions], page 209 を参照)。
- マクロ呼び出し (Chapter 14 [Macros], page 242 を参照)。
- condition-case (Section 11.7.3 [Errors], page 164 を参照)。

変数はバッファーローカルなバインディングを持つこともできます (Section 12.11 [Buffer-Local Variables], page 190 を参照)。数は多くありませんが、端末ローカル (terminal-local) なバインディングをもつ変数もあります (Section 29.2 [Multiple Terminals], page 698 を参照)。この種のバインディングは、通常のローカルバインディングのように機能することもありますが、これらは Emacs 内のどこにいるかに依存してローカルになります。

max-specpdl-size [User Option]

この変数はローカルな変数バインディングと、unwind-protectにゆるクリーンアップ (Section 11.7.4 [Cleaning Up from Nonlocal Exits], page 172 を参照) の総数にたいする制限を定義し、この変数を越えると Emacs は (データ"Variable binding depth exceeds max-specpdl-size"とともに) エラーをシグナルする。

このリミットは、もし超過したときにエラーが関連付けられていれば、誤って定義された関数による無限再起を避けるための 1 つの手段になる。ネストの深さにたいする他の制限としては、max-lisp-eval-depthがある。[Eval], page 142 を参照のこと。

デフォルト値は 1600。Lisp デバッガーのエントリーしたとき、もし残りが少なければ、デバッガーを実行するための空きを作るために値が増加される。

12.4 変数が void のとき

シンボルの値セル (Section 9.1 [Symbol Components], page 124 を参照) に値が割り当てられていない場合、その変数は void(空) であると言います。

Emacs Lisp のデフォルトであるダイナミックスコープルール (Section 12.10 [Variable Scoping], page 185 を参照) の下では、値セルはその変数のカレント値 (ローカルまたはグローバル) を保持します。値が割り当てられていない値セルは、値セルに nilをもつのとは異なることに注意してください。シンボル nilは Lisp オブジェクトであり、他のオブジェクトと同様に変数の値となることができます。nilは値なのです。変数が void の場合にその変数の評価を試みると、値をリターンするかわりに、void-variableエラーがシグナルされます。

オプションであるレキシカルスコープルール (lexical scoping rule) の下では、値セル保持できるのはその変数のグローバル値 — 任意のレキシカルバインディング構造の外側の値だけです。変数がレキシカルにバインドされている場合、ローカル値はそのレキシカル環境により決定されます。したがってこれらのシンボルの値セルに値が割り当てられていなくても、変数はローカル値を持つことができます。

`makunbound` *symbol* [Function]

この関数は *symbol*の値セルを空にして、その変数を void にする。この関数は *symbol*をリターンする。

*symbol*がダイナミックなローカルバインディングをもつなら、`makunbound`はカレントのバインディングを void にして、そのローカルバインディングが効果を持つ限り void にする。その後で以前にシャドーされたローカル値 (またはグローバル値) が再び有効になって、再び有効になった値が void でなければ、その変数は void ではなくなる。

いくつか例を示す (ダイナミックバインディングが有効だとする):

```
(setq x 1)                ; グローバルバインディングに値をセットする
     ⇒ 1
(let ((x 2))              ; それをローカルにバインドする
  (makunbound 'x)         ; ローカルバインディングを void にする
  x)
error  Symbol's value as variable is void: x
x                         ; グローバルバインディングは変更されない
     ⇒ 1

(let ((x 2))              ; ローカルにバインドする
  (let ((x 3))            ; もう一度
    (makunbound 'x)       ; 最内のローカルバインディングを void にする
    x))                   ; それを参照すると、void
error  Symbol's value as variable is void: x

(let ((x 2))
  (let ((x 3))
    (makunbound 'x))      ; 内側のバインディングを void にしてから取り除く
  x)                      ; 外側の letバインディングが有効になる
     ⇒ 2
```

`boundp` *variable* [Function]

この関数は *variable*(シンボル) が void でなければ `t`、void なら `nil`をリターンする。

いくつか例を示す (ダイナミックバインディングが有効だとする):

```
(boundp 'abracadabra)          ; 最初は void
     ⇒ nil
(let ((abracadabra 5))         ; ローカルにバインドする
  (boundp 'abracadabra))
     ⇒ t
(boundp 'abracadabra)          ; グローバルではまだ void
     ⇒ nil
(setq abracadabra 5)           ; グローバルで非 void にする
     ⇒ 5
(boundp 'abracadabra)
     ⇒ t
```

12.5 グローバル変数の定義

変数定義 *(variable definition)* とは、そのシンボルをグローバル変数として使用する意図を表明する構文です。これには以下で説明するスペシャルフォーム `defvar`や `defconst`が使用されます。

変数宣言は 3 つの目的をもちます。1 番目はコードを読む人にたいして、そのシンボルが特定の方法 (変数として) 使用されることを意図したものだと知らせることです。2 番目は Lisp システムにたいしてオプションで初期値とドキュメント文字列を与えて、これを知らせることです。3 番目は `etags`のようなプログラミングツールにたいして、その変数が定義されている場所を見つけられるように情報を提供することです。

`defconst`と `defvar`の主な違いは、人間の読み手に値が変更されるかどうかを知らせることにあります。Emacs Lisp は実際に、`defconst`で定義された変数の値の変更を妨げません。この 2 つのフォームの特筆すべき違いは、`defconst`は無条件で変数を初期化して、`defvar`は変数が元々void のときだけ初期化することです。

カスタマイズ可能な変数を定義する場合は、`defcustom`を使用するべきです (これはサブルーチンとして `defvar`を呼び出す)。Section 15.3 [Variable Definitions], page 253 を参照してください。

`defvar` *symbol* [*value* [*doc-string*]] [Special Form]

このスペシャルフォームは変数として *symbol*を定義する。*symbol*が評価されないことに注意。シンボルは `defvar`フォーム内に明示的に表記して定義される必要がある。この変数は特別だとマークされて、これは常に変数がダイナミックにバインドされることを意味する (Section 12.10 [Variable Scoping], page 185 を参照)。

*value*が指定されていて *symbol*が void(たとえばこのシンボルがダイナミックにバインドされた値を持たないとき。Section 12.4 [Void Variables], page 177 を参照) なら *value*が評価されて、その結果が *symbol*にセットされる。しかし *symbol*が void でなければ、*value*は評価されず *symbol*の値は変更されない。*value*が省略された場合は、いかなる場合も *symbol*の値は変更されない。

たとえ `nil`であっても値を指定することにより、その変数は特別だと永続的にマークされることに注意。一方で *value*が省略されると変数はローカル (カレントのレキシカルスコープまたはトップレベルにあればファイル) でのみ特別だとマークされる。これはバイトコンパイルの警告を抑止するために有用。Section 17.6 [Compiler Errors], page 290 を参照のこと。

*symbol*がカレントバッファー内でバッファーローカルなバインディングをもつ場合、`defvar`はデフォルト値に作用する。デフォルト値はバッファーローカルなバインディングではなく、バッファーにたいして独立である。デフォルト値が void のときはデフォルト値をセットする。Section 12.11 [Buffer-Local Variables], page 190 を参照のこと。

すでに *symbol*がレキシカルにバインドされている場合 (たとえばレキシカルバインドが有効な状態で `let`フォーム内に `defvar`があるような場合)、`defvar`はダイナミックな値をセットする。バインディング構文を抜けるまで、レキシカルバインディングは効果をもつ。Section 12.10 [Variable Scoping], page 185 を参照のこと。

Emacs Lisp モードで *C-M-x* (`eval-defun`) でトップレベルの `defvar`を評価するとき、`eval-defun`の特別な機能はその値が void であるかテストすることなく、その変数を無条件にセットする。

引数 *doc-string*が与えられたら、それは変数にたいするドキュメント文字列を指定する (そのシンボルの `variable-documentation`プロパティーに格納される)。Chapter 24 [Documentation], page 519 を参照のこと。

以下にいくつか例を示す。これは `foo`を定義するが初期化は行わない:

```
(defvar foo)
     ⇒ foo
```

以下の例は `bar`の値を 23に初期化してドキュメント文字列を与える:

```
(defvar bar 23
  "The normal weight of a bar.")
     ⇒ bar
```

`defvar`フォームは *symbol*をリターンするが、これは通常は値が問題にならないファイル内のトップレベルで使用される。

値をもたない `defvar`のより詳細な使用例は [Local defvar example], page 189 を参照のこと。

`defconst` *symbol value* [*doc-string*] [Special Form]

このスペシャルフォームはある値で *symbol*を定義して、それを初期化する。これはコードを読む人に、*symbol*がここで設定される標準的なグローバル値をもち、ユーザーや他のプログラムがそれを変更すべきではないことを知らせる。*symbol*が評価されないことに注意。定義されるシンボルは `defconst`内に明示的に記されなければならない。

`defvar`と同様、`defconst`は変数を特別 — この変数が常にダイナミックにバインドされているという意味 — であるとマークする (Section 12.10 [Variable Scoping], page 185 を参照)。加えてこれはその変数を危険であるとマークする (Section 12.12 [File Local Variables], page 197 を参照)。

`defconst`は常に *value*を評価して、その結果を *symbol*の値にセットする。カレントバッファー内で *symbol*がバッファーローカルなバインディングをもつなら、`defconst`はデフォルト値ではなくバッファーローカルな値をセットする (しかし `defconst`で定義されたシンボルにたいしてバッファーローカルなバインディングを作らないこと)。

`defconst`の使い方の例は、Emacs の `float-pi` — (たとえインディアナ州議会が何を試みようと) 何者かにより変更されるべきではない数学定数 *pi* にたいする定義である。しかし 2 番目の `defconst`の例のように、これは単にアドバイス的なものである。

```
(defconst float-pi 3.141592653589793 "The value of Pi.")
     ⇒ float-pi
(setq float-pi 3)
     ⇒ float-pi
float-pi
     ⇒ 3
```

警告**:** 変数がローカルバインディングをもつとき (`let`により作成された、または関数の引数の場合) に、スペシャルフォーム `defconst`または `defvar`を使用すると、これらのフォームはグローバルバインディングではなく、ローカルバインディングをセットします。これは通常は、あなたが望むことではないはずです。これを防ぐには、これらのスペシャルフォームをファイル内のトップレベルで使用します。この場所は通常、何のローカルバインディングも効果をもたないので、その変数にたいするローカルバインディングが作成される前にファイルがロードされることが確実だからです。

12.6 堅牢な変数定義のためのヒント

値が関数 (または関数のリスト) であるような変数を定義するときには、変数の名前の最後に‘`-function`’(または‘`-functions`’) を使用します。

他にも変数名に関する慣習があります。以下はその完全なリストです:

‘`...-hook`’
: 変数はノーマルフック (Section 23.1 [Hooks], page 461 を参照)。

‘`...-function`’
: 値は関数。

‘`...-functions`’
: 値は関数のリスト。

‘`...-form`’
: 値はフォーム (式)。

‘`...-forms`’
: 値はフォーム (式) のリスト。

‘`...-predicate`’
: 値は述語 (predicate) — 1 つの引数をとる関数 — であり成功なら非 `nil`、失敗なら `nil`をリターンする。

‘`...-flag`’
: `nil`か否かだけが意味をもつような値。結局そのような変数は、やがては多くの値をもつことが多いので、この慣習を強く推奨はしない。

‘`...-program`’
: 値はプログラム名。

‘`...-command`’
: 値は完全なシェルコマンド。

‘`...-switches`’
: 値はコマンドにたいして指定するオプション。

‘*`prefix`*`--...`’
: これは内部的な使用を意図した変数でありファイル *`prefix`*`.el`内で定義される (他の規約にしたがうかもしれない 2018 年以前に貢献された Emacs コードは段階的に廃止される)。

‘`...-internal`’
: これは内部的な使用を意図した変数でありファイル C コード内で定義される (他の規約にしたがうかもしれない 2018 年以前に貢献された Emacs コードは段階的に廃止される)。

変数を定義するときは、その変数を安全 (safe) とマークすべきか、それとも危険 (risky) とマークすべきかを常に考慮してください。Section 12.12 [File Local Variables], page 197 を参照してください。

複雑な値を保持する変数 (バインディングをもつ keymap など) の定義や初期化を行う場合は、以下のように値の計算をすべて `defvar`の中に配置するのが最良です:

```
(defvar my-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\C-c\C-a" 'my-command)
    ...
    map)
  docstring)
```

この方法にはいくつかの利点があります。1 つ目はファールをロード中にユーザーが中断した場合、変数はまだ初期化されていないか、初期化されているかのどちらかであり、その中間ということはありません。まだ初期化されていなければ、ファイルをリロードすれば正しく初期化されます。2 つ目は一度初期化された変数は、ファイルをリロードしても変更されないことです。コンテンツの一部を変更 (たとえばキーのリバインド) するフックをユーザーが実行した場合などに、これは重要です。3 つ目は *`C-M-x`*で `defvar`を評価すると、そのマップは完全に再初期化されることです。

`defvar`フォーム内に多すぎるコードを配置することが不利な点が 1 つあります。ドキュメント文字列が変数の名前から離れた場所に配置されることです。これを避ける安全な方法は以下の方法です:

```
(defvar my-mode-map nil
  docstring)
(unless my-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\C-c\C-a" 'my-command)
```

```
...
(setq my-mode-map map)))
```

これは初期化を defvarの内側に配置した場合とまったく同じ利点をもちますが、変数を再度初期化したい場合は、各フォームにたいして 1 回ずつ、*C-M-x*を 2 回タイプしなければならない点が異なります。

12.7 変数の値へのアクセス

変数を参照する通常の方法は、それに名前をつけるシンボルを記述する方法です。Section 10.1.2 [Symbol Forms], page 134 を参照してください。

実行時にのみ決定される変数を参照したいときがあるかもしれません。そのような場合、プログラム中のテキストで変数名を指定することはできません。そのような値を抽出するために symbol-value を使うことができます。

`symbol-value` *symbol* [Function]

この関数は *symbol*の値セルに格納された値をリターンする。これはその変数の (ダイナミックな) カレント値が格納された場所である。その変数がローカルバインディングをもたなければ単にその変数のグローバル値になる。変数が void なら void-variableはエラーをシグナルする。

その変数がレキシカルにバインドされていれば、symbol-valueが報告する値は、その変数のレキシカル値と同じである必要はない。レキシカル値はそのシンボルの値セルではなく、レキシカル環境により決定される。Section 12.10 [Variable Scoping], page 185 を参照のこと。

```
(setq abracadabra 5)
     ⇒ 5
(setq foo 9)
     ⇒ 9

;; ここでシンボル abracadabra
;;    は値がテストされるシンボル
(let ((abracadabra 'foo))
  (symbol-value 'abracadabra))
     ⇒ foo

;; ここでは abracadabraの値、
;;    つまり fooが値を
;;    テストされるシンボル
(let ((abracadabra 'foo))
  (symbol-value abracadabra))
     ⇒ 9

(symbol-value 'abracadabra)
     ⇒ 5
```

12.8 変数の値のセット

ある変数の値を変更する通常の方法は、スペシャルフォーム setqを使用する方法です。実行時に変数選択を計算する必要がある場合には関数 setを使用します。

`setq` [*symbol form*]... [Special Form]

このスペシャルフォームは、変数の値を変更するためのもっとも一般的な方法である。*symbol* にはそれぞれ、新しい値 (対応する *form*が評価された結果) が与えられる。そのシンボルのカレントバインディングは変更される。

`setq`は *symbol*を評価せずに、記述されたシンボルをセットする。この引数のことを自動的にクォートされた *(automatically quoted)* と呼ぶ。`setq`の '`q`'は "quoted(クォートされた)" が由来。

`setq`フォームの値は最後の *form*の値となる。

```
(setq x (1+ 2))
     ⇒ 3
x                    ; ここで xはグローバル値をもつ
     ⇒ 3
(let ((x 5))
  (setq x 6)         ; xのローカルバインディングをセット
  x)
     ⇒ 6
x                    ; グローバル値は変更されない
     ⇒ 3
```

1 番目の *form*が評価されてから 1 番目の *symbol*がセット、次に 2 番目の *form*が評価されてから *symbol*が評価されて、... となることに注意:

```
(setq x 10           ; ここで、xがセットされるのは
      y (1+ x))      ;    yの計算前であることに注目
     ⇒ 11
```

`set` *symbol value* [Function]

この関数は *symbol*の値セルに *value*を配置する。これはスペシャルフォームではなく関数なので、シンボルにセットするために *symbol*に記述された式は評価される。リターン値は *value*。

ダイナミックな変数バインドが有効 (デフォルト) なら、`set`は自身の引数 *symbol*を評価するが、`setq`は評価しないという点を除き、`set`は `setq`と同じ効果をもつ。しかし変数がレキシカルバインドなら、`set`は変数のダイナミックな値に、`setq`は変数のカレント値 (レキシカル値) に影響する。Section 12.10 [Variable Scoping], page 185 を参照のこと。

```
(set one 1)
error  Symbol's value as variable is void: one
(set 'one 1)
     ⇒ 1
(set 'two 'one)
     ⇒ one
(set two 2)          ; twoはシンボル oneに評価される
     ⇒ 2
one                  ; したがって oneがセットされる
     ⇒ 2
(let ((one 1))       ; oneのこのバインディングがセットされる
  (set 'one 3)       ;    のであってグローバル値はセットされない
  one)
     ⇒ 3
```

```
one
     ⇒ 2
```

*symbol*が実際のシンボルでなければ `wrong-type-argument`エラーがシグナルされる。

```
(set '(x y) 'z)
error  Wrong type argument: symbolp, (x y)
```

12.9 変数が変更されたときに実行される関数。

変数の値が変化したときに何らかのアクションを行えれば便利なときがあります。変数 *watchpoint(variable watchpoint)* 機能はそのための機能を提供します。この機能の有効な利用方法としては変数セッティングと表示の同期、変数への予期せぬ変更を追跡するためのデバッガの呼び出しが含まれます (Section 18.1.4 [Variable Debugging], page 299 を参照)。

以下の関数は関数にたいする watch 関数の操作や問い合わせに使用できます。

`add-variable-watcher` *symbol watch-function* [Function]

この関数は*symbol*が変化したときは常に *watch-function*が呼び出されるようにアレンジする。エイリアスを介した変更にも同じ効果をもつ (Section 12.15 [Variable Aliases], page 204 を参照)。

*watch-function*は *symbol*の値の変更直前に *symbol*、*newval*、*operation*、*where*という 4 つの引数で呼び出される。*symbol*は変更される変数、*newval*は変更後の値 (*watch-function* では *newval*に変更される前なので古い値は *symbol*の値で利用可能)、*operation*は変更の種類を表すシンボルであり `set`、`let`、`unlet`、`makunbound`、`defvaralias`のいずれか。*where* は変数のバッファーローカルな値が変更される場合にはバッファー、それ以外は `nil`。

`remove-variable-watcher` *symbol watch-function* [Function]

この関数は *symbol*の watcher リストから *watch-function*を削除する。

`get-variable-watchers` *symbol* [Function]

この関数は *symbol*のアクティブな watcher 関数のリストをリターンする。

12.9.1 制限

watchpoint をトリガーせずに変数が変更される (または少なくとも変更されたように見える) 方法がいくつかあります。

watchpoint はシンボルにアタッチされるので変数内に含まれるオブジェクトの変更 (リスト変更関数による変更。Section 5.6 [Modifying Lists], page 81 を参照のこと) はこのメカニズムにより検出されません。

さらに C のコードは watchpoint メカニズムをバイパスして変数の値を直接変更できます。

繰り返しになりますがこれはシンボルをターゲットとするので、この機能のマイナーな制限はダイナミックなスコープをもつ変数だけをウォッチできるということです。レキシカル変数への変更は変数スコープ内のコードを調べれば容易に発見できるので、これが問題をもたらすことは稀でしょう (結局のところいかなるコードからも変更され得るダイナミック変数とは異なる。Section 12.10 [Variable Scoping], page 185 を参照のこと)。

12.10 変数のバインディングのスコーピングルール

ある変数にたいするローカルバインディングを作成するとき、そのバインディングはプログラムの限られた一部だけに効果をもちます (Section 12.3 [Local Variables], page 175 を参照)。このセクションでは、これが正確には何を意味するかについて説明します。

ローカルバインディングはそれぞれ、個別にスコープ *(scope:* 範囲という意味*)* とエクステント *(extent:* これも範囲を意味する*)* をもちます。スコープはそのバインディングにアクセスできるのが、テキストのソースコードのどこ *(where)* であるかを示します。エクステントはプログラムの実行中に、そのバインディングが存在するのがいつ *(when)* であるかを示します。

デフォルトでは Emacs が作成したローカルバインディングは、ダイナミックバインディング *(dynamic binding)* です。このようなバインディングはダイナミックスコープ *(dynamic scope)* をもち、それはプログラムの任意の範囲が、その変数バインディングにアクセスするかもしれないことを意味します。これはダイナミックエクステント *(dynamic extent)* ももっています。これはそのバインディング構造 (`let`フォームの body など) が実行される間だけ、そのバインディングが存続することを意味します。

Emacs はオプションでレキシカルバインディング *(lexical binding)* を作成することができます。レキシカルバインディングはレキシカルスコープ *(lexical scope)* をもち、これはその変数にたいするすべての参照が、バインディング構文内にテキスト的に配置されなければならないことを意味します[2]。レキシカルバインディングは不定エクステント *(indefinite extent)* ももっています。これはある状況下において、クロージャー *(closures)* と呼ばれるスペシャルオブジェクトにより、バインディング構造が実行を終えた後でさえも、存続し続けることを意味します。

以降のサブセクションでは、ダイナミックバインディングとレキシカルバインディング、および Emacs Lisp プログラムでレキシカルバインディングを有効にする方法についてより詳細に説明します。

12.10.1 ダイナミックバインディング

デフォルトでは、Emacs により作成されるローカル変数のバインディングはダイナミックバインディングです。ある変数がダイナミックにバインドされていると、Lisp プログラムの実行における任意のポイントでのカレントバインディングは、単にそのシンボルにたいしてもっとも最近作成されたダイナミックなローカルバインディング、またはそのようなローカルバインディングが存在しなければグローバルバインディングになります。

以下の例のように、ダイナミックバインディングはダイナミックスコープとダイナミックエクステントをもちます:

[2] これにはいくつか例外があります。たとえばレキシカルバインディングは、Lisp デバッガーからもアクセスできます。

```
(defvar x -99)  ; xは初期値として −99 を受け取る

(defun getx ()
  x)            ; この関数内では xは自由に使用される

(let ((x 1))    ; xはダイナミックにバインドされている
  (getx))
     ⇒ 1

;; letフォームが終了した後に
;; xは前の値 −99 にリバートされる

(getx)
     ⇒ -99
```

関数 getxは xを参照します。defun構文自体の中に xにたいするバインディングが存在しないという意味において、これはフリーな参照です。xが (ダイナミックに) バインドされている letフォーム内から getxを呼び出すと、ローカル値 (つまり 1) が取得されます。しかしその後 letフォームの外側から getxを呼び出すと、グローバル値 (つまり −99) が取得されます。

以下は setqを使用してダイナミックに変数をバインドする例です:

```
(defvar x -99)      ; xは初期値として −99 を受け取る

(defun addx ()
  (setq x (1+ x)))  ; xに 1 加算して新しい値をリターンする

(let ((x 1))
  (addx)
  (addx))
     ⇒ 3            ; addxを 2 回呼び出すと xに 2 回加算される

;; letフォームが終了した後に
;; xは前の値 −99 にリバートされる

(addx)
     ⇒ -98
```

Emacs Lisp でのダイナミックバインディングは、シンプルな方法で実装されています。シンボルはそれぞれ、シンボルのカレントのダイナミック値 (または値の不在) を指定する値セルをもちます。Section 9.1 [Symbol Components], page 124 を参照してください。あるシンボルがダイナミックなローカル値を与えられたとき、Emacs は値セルの内容 (または値の不在) をスタックに記録して、新しいローカル値を値セルに格納します。バインディング構文が実行を終えたとき、Emacs はスタックから古い値を pop して値セルにそれを配置します。

12.10.2 ダイナミックバインディングの正しい使用

ダイナミックバインディングは、プログラムにたいしてテキスト的なローカルスコープ内で定義されていない変数を参照することを許容する、強力な機能です。しかし無制限に使用した場合には、プログラムの理解を困難にしてしまうこともあります。このテクニックを使用するために 2 つの明解な方法があります:

- ある変数がグローバルな定義をもたなければ、ローカル変数としてバインディング構文内 (その変数がバインドされる `let`フォームの body などの場所) だけでそれを使用する。プログラムでこの慣習に一貫してしたがえば、プログラム内の他の場所で同じ変数シンボルを任意に使用しても、その変数の値に影響を与えたり、影響を受けることがなくなる。
- それ以外では `defvar`、`defconst` (Section 12.5 [Defining Variables], page 178 を参照)、`defcustom` (Section 15.3 [Variable Definitions], page 253 を参照) で変数を定義する。この定義は通常は Emacs Lisp ファイル内のトップレベルであること。この定義には可能な限り変数の意味と目的を説明するドキュメント文字列を含めること。また名前の衝突を避けるように変数を命名すること (Section D.1 [Coding Conventions], page 1162 を参照)。

  そうすればプログラム内のどこか別の場所で、それが何に影響するか確信をもって変数をバインドすることができます。その変数にどこで出会っても、(たとえば変数の定義が Emacs にロードされていれば *`C-h v`*コマンドを通じて) 定義を参照するのが簡単になります。Section “Name Help” in *The GNU Emacs Manual* を参照してください。

  たとえば `case-fold-search`のようなカスタマイズ可能な変数にたいしてローカルバインディングを使用するのは一般的です:

```
(defun search-for-abc ()
  "Search for the string \"abc\", ignoring case differences."
  (let ((case-fold-search t))
    (re-search-forward "abc")))
```

12.10.3 レキシカルバインディング

Emacs のバージョン 24.1 からオプションの機能としてレキシカルバインディングが導入されました。わたしたちはこの機能の重要性が時とともに増加することを期待します。レキシカルバインディングは最適化の機会をより広げるので、この機能を使用するプログラムはおそらく将来の Emacs バージョンで高速に実行されるようになるでしょう。レキシカルバインディングは、わたしたちがバージョン 26。1 の Emacs で追加した並列性 (concurrency) とも互換があります。

レキシカルにバインドされた変数はレキシカルスコープ *(lexical scope)* をもちます。これはその変数にたいする参照が、そのバインディング構文内にテキスト的に配置されなければならないことを意味しています。以下は例です (実際にレキシカルバインディングを有功にする方法は、次のサブセクションを参照のこと):

```
(let ((x 1))     ; xはレキシカルにバインドされる
  (+ x 3))
     ⇒ 4

(defun getx ()
  x)             ; この関数内では xは自由に使用される

(let ((x 1))     ; xはレキシカルにバインドされる
  (getx))
error  Symbol's value as variable is void: x
```

ここでは `x`はグローバル値をもちません。`let`フォーム内でレキシカルにバインドされたとき、この変数は `let`のテキスト境界内で使用できます。しかしこの `let`内から呼び出される `getx`関数からは、`getx`の関数定義が `let`フォームの外側なので使用することができません。

レキシカルバインディングが機能する方法を説明します。バインディング構文はぞれぞれ、その構文内でローカル値にバインドする変数を指定する、レキシカル環境 *(lexical environment)* を定義し

ます。Lisp の評価機能 (Lisp evaluator) が、ある変数のカレント値を得たいときは、最初にレキシカル環境内を探します。そこで変数が指定されていなければ、ダイナミック値が格納されるシンボルの値セルを探します。

(内部的にはレキシカル環境はシンボルと値がペアになった alist でり、alist の最後の要素はコンスセルではなくシンボル `t`である。そのような alist はフォームを評価するためのレキシカル環境を指定するために、`eval`関数の 2 番目の引数として渡すことができる。Section 10.4 [Eval], page 140を参照のこと。しかしほとんどの Emacs Lisp プログラムは、この方法で直接レキシカル環境を使用するべきではない。デバッガーのような特化されたプログラムだけが使用すること。)

レキシカルバインディングは不定エクステント (indefinite extent) をもちます。バインディング構造が終了した後でも、そのレキシカル環境はクロージャー *(closures)* と呼ばれる Lisp オブジェクト内に “保持” されるかもしれ、あせん。クロージャーはレキシカルバインディングが有効な、名前つきまたは無名 (anonymous) の関数が作成されたときに作成されます。詳細は Section 13.10 [Closures], page 227 を参照してください。

クロージャーが関数として呼び出されたとき、その関数の定義内のレキシカル変数にたいする任意の参照は、維持されたレキシカル環境を使用します。以下は例です:

```
(defvar my-ticker nil)   ; クロージャーを格納するために
                         ; この変数を使用する

(let ((x 0))             ; xはレキシカルにバインドされる
  (setq my-ticker (lambda ()
                    (setq x (1+ x)))))
    ⇒ (closure ((x . 0) t) ()
        (setq x (1+ x)))

(funcall my-ticker)
    ⇒ 1

(funcall my-ticker)
    ⇒ 2

(funcall my-ticker)
    ⇒ 3

x                        ; xはグローバル値をもたないことに注意
error  Symbol's value as variable is void: x
```

`let`バインディングは、内部に変数 `x`をもつレキシカル環境を定義して、変数は 0 にローカルにバインドされます。このバインディング構文内で `x`を 1 増加して、増加された値をリターンするクロージャーを定義しています。このラムダ式は自動的にクロージャーとなり、たとえ `let`構文を抜けた後でも、その内部ではレキシカル環境が存続します。クロージャーを評価するときは、毎回レキシカル環境内の `x`のバインディングが使用されて、`x`が加算されます。

シンボルオブジェクト自体に束縛されるダイナミック変数と異なり、レキシカル変数とシンボルの関係はインタープリター (かコンパイラー) 内にのみ存在します。したがって (`symbol-value`、`boundp`、`set`のような) シンボル引数を受け取る関数ができるのは、変数のダイナミックなバインディング (そのシンボルの値セルの内容) の取得と変更だけです。

12.10.4 レキシカルバインディングの使用

Emacs Lisp ファイルのロードや Lisp バッファーを評価するとき、バッファーローカルな変数 `lexical-binding`が非 `nil`なら、レキシカルバインディングが有効になります:

`lexical-binding` [Variable]
: このバッファーローカルな変数が非 `nil`なら、Emacs Lisp ファイルとバッファーはダイナミックバインディングではなくレキシカルバインディングを使用して評価される (しかし特別な変数はダイナミックにバインドされたまま。以下を照)。`nil`ならすべてのローカル変数にたいしてダイナミックバインディングが使用される。この変数は、通常はファイルローカル変数として、Emacs Lisp ファイル全体にたいしてセットされる (Section 12.12 [File Local Variables], page 197 を参照)。他のファイルローカル変数などとは異なり、ファイルの最初の行でセットされなければならないことに注意。

`eval`呼び出しを使用して Emacs Lisp コードを直接評価するとき、`eval`の *lexical*引数が非 `nil`なら、レキシカルバインディングが有効になります。Section 10.4 [Eval], page 140 を参照してください。

レキシカルバインディングは`*scratch*`バッファーで使用される Lisp Interaction モード、および`*ielm*`バッファーで使用される IELM モードでも有効であり、*`M-:`* (`eval-expression`) を通じた式の評価や、Emacs と `emacsclient` (Section “emacsclient Options” in *The GNU Emacs Manual* を参照) の`--eval`コマンドラインオプション (Section “Action Arguments” in *The GNU Emacs Manual* を参照) を処理する際にも有効です。

レキシカルバインディングが有効な場合でも、特定の変数はダイナミックにバインドされたままです。これらはスペシャル変数 *(special variable)* と呼ばれます。`defvar`、`defcustom`、`defconst`で定義されたすべての変数はスペシャル変数です (Section 12.5 [Defining Variables], page 178 を参照)。その他のすべての変数はレキシカルバインディングの対象になります。

値なしで `defvar`を使用することにより、他の場所ではレキシカルにバインドされている状態のまま、単一ファイルやファイルの一部だけで変数をダイナミックにバインドすることが可能になります。たとえば:

```
(let (_)
  (defvar x)      ; xへの let バインドはこの let 内ではダイナミック
  (let ((x -99))  ; これは xのダイナミックバインド
    (defun get-dynamic-x ()
      x)))

(let ((x 'lexical)) ; これは xのレキシカルバインド
  (defun get-lexical-x ()
    x))

(let (_)
  (defvar x)
  (let ((x 'dynamic))
    (list (get-lexical-x)
          (get-dynamic-x))))
    ⇒ (lexical dynamic)
```

`special-variable-p` *symbol* [Function]

この関数は *symbol*がスペシャル変数 (つまり変数が `defvar`、`defcustom`、`defconst`による定義をもつ) なら非 `nil`をリターンする。、それ以外ならリターン値は `nil`。

これは関数なので永続的にスペシャルな変数には非 `nil`をリターンできるが、カレントレキシカルスコープでのみスペシャルな変数では異なることに注意。

関数内での通常の引数としてのスペシャル変数の使用は、推奨されません。レキシカルバインディングモードが有効なときにこれを行うと、(あるときはレキシカルバインディング、またあるときはダイナミックバインディングのような) 不定な動作が起こります。

Emacs Lisp プログラムをレキシカルバインディングに変換するのは簡単です。最初に Emacs Lisp ソースファイルのヘッダー行で `lexical-binding`を `t`にして、ファイルローカル変数を追加します (Section 12.12 [File Local Variables], page 197 を参照)。次に意図せずレキシカルにバインドしてしまわないように、ダイナミックなバインドをもつ必要がある変数が変数定義をもつことを各変数ごとにチェックします。

どの変数が変数定義をもつ必要があるか見つけるシンプルな方法は、ソースファイルをバイトコンパイルすることです。Chapter 17 [Byte Compilation], page 285 を参照してください。`let`フォームの外側で非スペシャル変数が使用されていれば、バイトコンパイラーはフリーな変数にたいする参照や割り当てについて警告するでしょう。非スペシャル変数がバインドされているが `let`フォーム内で使用されていなければ、バイトコンパイラーは使用されないレキシカル変数に関して警告するでしょう。バイトコンパイラーは、スペシャル変数を関数の引数として使用している場合も問題を警告します。

(使用されていない変数についての警告を抑制するためには単に変数名をアンダースコアーで開始する。そうすればバイトコンパイラーは変数が使用されないことを示すと解釈する。)

12.11 バッファーローカル変数

グローバルおよびローカルな変数バインディングは、いずれかの形式をほとんどのプログラミング言語で見つけることができます。しかし Emacs は 1 つのバッファーだけに適用されるバッファーローカル *(buffer-local)* なバインディング用に、普通には存在しない類の変数バインディングもサポートしています。ある変数にたいして異なるバッファーごとに別の値をもつのは、カスタマイズでの重要な手法です (変数は端末ごとにローカルなバインディングをもつこともできる。Section 29.2 [Multiple Terminals], page 698 を参照)。

12.11.1 バッファーローカル変数の概要

バッファーローカル変数は特定のバッファーに関連づけられた、バッファーローカルなバインディングをもちます。このバインディングはそのバッファーがカレントのときに効果をもち、カレントでないときには効果がありません。バッファーローカルなバインディングが効力をもつときにその変数をセットすると、そのバインディングは新しい値をもちますが他のバインディングは変更されません。これはバッファーローカルなバインディングを作成したバッファーだけで変更が見えることを意味します。

その変数にたいする特定のバッファーに関連しない通常のバインディングは、デフォルトバインディング *(default binding)* と呼ばれます。これはほとんどの場合はグローバルバインディングです。

変数はあるバッファーではバッファーローカルなバインディングをもつことができ、他のバッファーではもたないことができます。デフォルトバインディングは、その変数にたいして自身のバインディングをもたないすべてのバッファーで共有されます (これには新たに作成されたバッファーが含まれる)。ある変数にたいして、その変数のバッファーローカルなバインディングをもたないバッファーでその変数をセットすると、それによりデフォルトバインディングがセットされるので、デフォルトバインディングを参照するすべてのバッファーで新しい値を見ることになります。

バッファーローカルなバインディングのもっとも一般的な使用は、メジャーモードがコマンドの動作を制御するために変数を変更する場合です。たとえばC モードや Lisp モードは、空行だけがパラグラフの区切りになるように変数 `paragraph-start`をセットします。これらのモードは、C モードや Lisp モードになるようなバッファー内でこの変数をバッファーローカルにすることでこれを行って、その後そのモードにたいする新しい値をセットします。Section 23.2 [Major Modes], page 463 を参照してください。

バッファーローカルなバインディングを作成する通常の方法は、`make-local-variable`による方法で、これは通常はメジャーモードが使用します。これはカレントバッファーだけに効果があります。その他すべてのバッファー (まだ作成されていないバッファーを含む) は、それらのバッファー自身が明示的にバッファーローカルなバインディングを与えるまでデフォルト値を共有し続けます。

変数を自動的にバッファーローカルになるようにマークする、より強力な操作は`make-variable-buffer-local`を呼び出すことにより行われます。これはたとえその変数がまだ作成されていなくても、変数をすべてのバッファーにたいしてローカルにすると考えることができます。より正確には変数を自動的にセットすることにより、その変数がカレントバッファーにたいしてローカルでなくても、変数をローカルにする効果があります。すべてのバッファーは最初は通常のようにデフォルト値を共有しますが、変数をセットすることでカレントバッファーにたいしてバッファーローカルなバインディングを作成します。新たな値はバッファーローカルなバインディングに格納され、デフォルトバインディングは変更されずに残ります。これは任意のバッファーで `setq`によりデフォルト値を変更できないことを意味します。変更する唯一の方法は `setq-default`だけです。

警告**:** ある変数が 1 つ以上のバッファーでバッファーローカルなバインディングをもつ際に、`let`はそのとき効力がある変数のバインディングをリバインドします。たとえばカレントバッファーがバッファーローカルな値をもつなら、`let`は一時的にそれをリバインドします。効力があるバッファーローカルなバインディングが存在しなければ `let`はデフォルト値をリバインドします。`let`の内部で、別のバインディングが効力をもつ別のバッファーをカレントバッファーにすると、それ以上 `let`バインディングを参照できなくなります。他のバッファーにいる間に `let`を抜けると、(たとえそれが正しくても) バインディングの解消を見ることはできません。以下にこれを示します:

```
(setq foo 'g)
(set-buffer "a")
(make-local-variable 'foo)
(setq foo 'a)
(let ((foo 'temp))
  ;; foo ⇒ 'temp  ; バッファー 'a'内での let バインディング
  (set-buffer "b")
  ;; foo ⇒ 'g     ; foo は 'b'にたいしてローカルではないためグローバル値
  body...)
foo ⇒ 'g         ; exit によりバッファー 'a'のローカル値が復元されるが
                 ; バッファー 'b'では見ることができない
(set-buffer "a") ; ローカル値が復元されたことを確認
foo ⇒ 'a
```

*body*内の `foo`にたいする参照は、バッファー 'b'のバッファーローカルなバインディングにアクセスすることに注意してください。

あるファイルがローカル変数の値をセットする場合、これらの変数はファイルを visit するときバッファーローカルな値になります。Section “File Variables” in *The GNU Emacs Manual* を参照してください。

バッファーローカル変数を端末ローカル (terminal-local) にすることはできません (Section 29.2 [Multiple Terminals], page 698 を参照)。

12.11.2 バッファーローカルなバインディングの作成と削除

`make-local-variable` *variable* [Command]

この関数はカレントバッファー内で、*variable*(シンボル) にたいするバッファーローカルなバインディングを作成する。他のバッファーは影響を受けない。リターンされる値は *variable*。

*variable*のバッファーローカルな値は、最初は以前に *variable*がもっていた値と同じ値をもつ。*variable*が void のときは void のまま。

```
;; バッファー 'b1'で行う:
(setq foo 5)                ; すべてのバッファーに影響する。
     ⇒ 5
(make-local-variable 'foo)  ; 'b1'内でローカルになった
     ⇒ foo
foo                         ; 値は変更されない
     ⇒ 5
(setq foo 6)                ; 'b1'内で値を変更
     ⇒ 6
foo
     ⇒ 6

;; バッファー 'b2'では、値は変更されていない
(with-current-buffer "b2"
  foo)
     ⇒ 5
```

変数を `let`バインディングでバッファーローカルにしても、`let`への出入り時の両方でこれを行うバッファーがカレントでなければ信頼性はない。これは `let`がバインディングの種類を区別しないからである。`let`に解るのはバインディングが作成される変数だけである。

定数や読み取り専用の変数をバッファーローカルにするとエラーになる。Section 12.2 [Constant Variables], page 174 を参照のこと。

変数が端末ローカル (Section 29.2 [Multiple Terminals], page 698 を参照) なら、この関数はエラーをシグナルする。そのような変数はバッファーローカルなバインディングをもつことができない。

警告: フック変数にたいして `make-local-variable`を使用しないこと。フック変数は `add-hook`か `remove-hook`の *local*引数を使用すると、必要に応じて自動でバッファーローカルになる。

`setq-local` **&rest** *pairs* [Macro]

*pairs*は変数と値のペアからなるリスト。このマクロはカレントバッファー内で変数それぞれにたいしてバッファーローカルなバインディングを作成して、それにバッファーローカルな値を与える。このマクロは各変数にたいして `make-local-variable`の後に `setq`を呼び出すのと等価。変数はクォートされていないシンボルであること。

```
(setq-local var1 "value1"
            var2 "value2")
```

`make-variable-buffer-local` *variable* [Command]
このコマンドは *variable*(シンボル) が自動的にバッファーローカルになるようにマークするので、それ以降にその変数へのセットを試みると、その時点でカレントのバッファーにローカルになる。しばしば混乱を招く `make-local-variable`とは異なり、これが取り消されることはなく、すべてのバッファー内での変数の挙動に影響する。

この機能特有の欠点は、(`let`やその他のバインディング構文による) 変数のバインディングが、その変数にたいするバッファーローカルなバインディングを作成しないことである。(`set`や`setq`による) 変数のセットだけは、その変数がカレントバッファーで作成された `let`スタイルのバインディングをもたないので、ローカルなバインディングを作成する。

*variable*がデフォルト値をもたない場合、このコマンドの呼び出しは `nil`のデフォルト値を与える。*variable*がすでにデフォルト値をもつなら、その値は変更されずに残る。それ以降に *variable*にたいして `makunbound`を呼び出すと、バッファーローカル値を void にして、デフォルト値は影響を受けずに残る。

▼リターン値は *variable*。

定数や読み取り専用の変数をバッファーローカルにするとエラーになる。Section 12.2 [Constant Variables], page 174 を参照のこと。

警告: ユーザーオプション変数では、ユーザーは異なるバッファーにたいして異なるカスタマイズを望むかもしれないので、`make-variable-buffer-local`を使うべきだと決め込むべきではない。ユーザーは望むなら任意の変数をローカルにできる。その選択の余地を残すほうがよい。

`make-variable-buffer-local`を使用すべきなのは、複数のバッファーが同じバインディングを共有しないことが自明な場合である。たとえばバッファーごとに個別な値をもつことに依存する Lisp プログラム内の内部プロセスにたいして変数が使用されるときは、`make-variable-buffer-local`の使用が最善の解決策になるかもしれない。

`defvar-local` *variable value* **&optional** *docstring* [Macro]
このマクロは *variable*を初期値 *value*と *docstring*の変数として定義して、それを自動的にバッファーローカルとマークする。これは `defvar`の後につづけて `make-variable-buffer-local`を呼び出すのと同じ。*variable*はクォートされていないシンボル。

`local-variable-p` *variable* **&optional** *buffer* [Function]
これは *variable*がバッファー *buffer*(デフォルトはカレントバッファー) 内でバッファーローカルなら `t`、それ以外は `nil`をリターンする。

`local-variable-if-set-p` *variable* **&optional** *buffer* [Function]
これは *variable*がバッファー *buffer*内でバッファーローカル値をもつ、または自動的にバッファーローカルになるなら `t`、それ以外は `nil`をリターンする。*buffer*が省略または `nil`の場合のデフォルトはカレントバッファー。

`buffer-local-value` *variable buffer* [Function]
この関数はバッファー *buffer*内の、*variable*(シンボル) のバッファーローカルなバインディングをリターンする。*variable*がバッファー *buffer*内でバッファーローカルなバインディングをもたなければ、かわりに *variable*のデフォルト値 (Section 12.11.3 [Default Value], page 195 を参照) をリターンする。

`buffer-local-variables` **&optional** *buffer* [Function]
この関数はバッファー *buffer*内のバッファーローカル変数を表すリストをリターンする (*buffer*が省略された場合はカレントバッファーが使用される)。リストの各要素は通常は `(sym . val)`

という形式をもつ。ここで *sym*はバッファーローカル変数 (シンボル)、*val*はバッファーローカル値。しかし *buffer*内のある変数のバッファーローカルなバインディングが void なら、その変数に対応するリスト要素は単に *sym*となる。

```
(make-local-variable 'foobar)
(makunbound 'foobar)
(make-local-variable 'bind-me)
(setq bind-me 69)
(setq lcl (buffer-local-variables))
    ;; 最初はすべてのバッファー内でローカルなビルトイン変数:
⇒ ((mark-active . nil)
    (buffer-undo-list . nil)
    (mode-name . "Fundamental")
    ...
    ;; 次にビルトインでないバッファーローカル変数
    ;; This one is buffer-local and void:
    foobar
    ;; これはバッファーローカルで void ではない:
    (bind-me . 69))
```

このリスト内のコンスセルの CDR に新たな値を格納しても、その変数のバッファーローカル値は変化しないことに注意。

`kill-local-variable` *variable* [Command]

この関数はカレントバッファー内の *variable*(シンボル) にたいするバッファーローカルなバインディング (もしあれば) を削除する。その結果として、このバッファー内で *variable*のデフォルトバインディングが可視になる。これは通常は *variable*の値を変更する。デフォルト値は削除されたバッファーローカル値とは異なるのが普通だからである。

セットしたとき自動的にバッファーローカルになる変数のバッファーローカルなバインディングを kill すると、これによりカレントバッファー内でデフォルト値が可視になる。しかし変数を再度セットすると、その変数にたいするバッファーローカルなバインディングが再作成される。

`kill-local-variable`は *variable*を return します。

この関数はコマンドである。なぜならバッファーローカル変数のインタラクティブな作成が有用な場合があるように、あるバッファーローカル変数のインタラクティブな kill が有用な場合があるからである。

`kill-all-local-variables` [Function]

この関数は permanent(永続的) とマークされた変数と`permanent-local-hook`プロパティーに非`nil`をもつローカルフック関数 (Section 23.1.2 [Setting Hooks], page 462 を参照) を除き、カレントバッファーのすべてのバッファーローカルなバインディングを解消する。結果として、そのバッファーのほとんどの変数がデフォルト値を参照するようになる。

この関数はそのバッファーに関連する他の特定の情報もリセットする。これはローカルキーマップを `nil`、構文テーブルを `(standard-syntax-table)`の値、case テーブルを `(standard-case-table)`、abbrev テーブルを `fundamental-mode-abbrev-table`の値にセットする。

この関数が 1 番最初に行うのはノーマルフック `change-major-mode-hook`(以下参照) の実行である。

すべてのメジャーモードコマンドは、Fundamental モードにスイッチする効果をもち、以前のメジャーモードのほとんどの効果を消去する、この関数を呼び出すことによって開始されます。この関数が処理を行うのを確実にするために、メジャーモードがセットする変数は permanent とマークすべきではない。

`kill-all-local-variables` returns `nil`.

`change-major-mode-hook` [Variable]

関数`kill-all-local-variables`は、何か他のことを行う前にまずこのノーマルフックを実行する。この関数はメジャーモードにたいして、ユーザーが他のメジャーモードにスイッチした場合に行われる何か特別なことを準備する方法を与える。この関数はユーザーがメジャーモードを変更した場合に忘れられるべき、バッファー固有のマイナーモードにたいしても有用。

最善の結果を得るために、この変数をバッファーローカルにすれば、処理が終了したときに消えるので、以降のメジャーモードに干渉しなくなる。Section 23.1 [Hooks], page 461 を参照のこと。

変数名 (シンボル) が非`nil`の`permanent-local`プロパティーをもつなら、そのバッファーローカル変数は *permanent(永続的)* です。そのような変数は`kill-all-local-variables`の影響を受けず、したがってメジャーモードの変更によりそれらのローカルバインディングは作成されません。permanent なローカル変数はファイルの内容を編集する方法ではなく、どこから読み込んだファイルか、あるいはどのように保存するかといったことに関連するデータに適しています。

12.11.3 バッファーローカル変数のデフォルト値

バッファーローカルなバインディングをもつ変数のグローバル値もデフォルト値 *(default)* 値と呼ばれます。なぜならその変数にたいしてカレントバッファーや選択されたフレームもバインディングをもたなければ、その値が常に効果をもつからです。

関数`default-value`と`setq-default`は、カレントバッファーがバッファーローカルなバインディングをもつかどうかに関わらず、その変数のデフォルト値にアクセスまたは変更します。たとえばほとんどのバッファーにたいして、`paragraph-start`のデフォルトのセッティングを変更するために、`setq-default`を使用できます。そしてこの変数にたいするバッファーローカルな値をもつ C モードや Lisp モードにいるときでさえ、これは機能します。

スペシャルフォーム`defvar`と`defconst`もバッファーローカルな値ではなく、(もし変数にセットする場合は) デフォルト値をセットします。

`default-value` *symbol* [Function]

この関数は *symbol*のデフォルト値をリターンする。これはこの変数にたいして独自の値をもたないバッファーやフレームから参照される値である。*symbol*がバッファーローカルでなければ、これは`symbol-value`(Section 12.7 [Accessing Variables], page 182 を参照) と同じ。

`default-boundp` *symbol* [Function]

関数`default-boundp`は *symbol*のデフォルト値が void でないか報告する。`(default-boundp 'foo)`が`nil`をリターンした場合には`(default-value 'foo)`はエラーになる。

`default-boundp`は`default-value`、`boundp`は`symbol-value`にたいする述語である。

`setq-default` [*symbol form*]... [Special Form]

このスペシャルフォームは各 *symbol*に新たなデフォルト値として、対応する *form*を評価した結果を与える。これは *symbol*を評価しないが *form*は評価する。`setq-default`フォームの値は最後の *form*の値。

カレントバッファーにたいして *symbol*がバッファーローカルでなく、自動的にバッファーローカルにマークされていなければ、`setq-default`は`setq`と同じ効果をもつ。カレントバッファーにたいして *symbol*がバッファーローカルなら、(バッファーローカルな値をもたない) 他のバッファーから参照できる値を変更するが、それはカレントバッファーが参照する値ではない。

```
;; バッファー‘foo’で行う:
(make-local-variable 'buffer-local)
     ⇒ buffer-local
(setq buffer-local 'value-in-foo)
     ⇒ value-in-foo
(setq-default buffer-local 'new-default)
     ⇒ new-default
buffer-local
     ⇒ value-in-foo
(default-value 'buffer-local)
     ⇒ new-default

;; (新しい) バッファー‘bar’で行う:
buffer-local
     ⇒ new-default
(default-value 'buffer-local)
     ⇒ new-default
(setq buffer-local 'another-default)
     ⇒ another-default
(default-value 'buffer-local)
     ⇒ another-default

;; バッファー‘foo’に戻って行う:
buffer-local
     ⇒ value-in-foo
(default-value 'buffer-local)
     ⇒ another-default
```

`set-default` *symbol value* [Function]

この関数は`setq-default`と似ているが、*symbol*は通常の引数として評価される。

```
(set-default (car '(a b c)) 23)
     ⇒ 23
(default-value 'a)
     ⇒ 23
```

ある変数に値を let バインドできます (Section 12.3 [Local Variables], page 175 を参照)。このバインディングにより変数のグローバル値はシャドーされます。`default-value`はグローバル値ではなくそのバインディングの値をリターンして、`set-default`によるグローバル値のセットは防がれます (かわりに let バインドされた値が変更される)。以下の 2 つの関数により let バインドでグローバル値がシャドーされていてもグローバル値を参照できます。

`default-toplevel-value` *symbol* [Function]

この関数は *symbol*にたいするすべての let バインディングの外部の値としてトップレベルのデフォルト値をリターンする。

```
(defvar variable 'global-value)
    ⇒ variable
(let ((variable 'let-binding))
  (default-value 'variable))
    ⇒ let-binding
(let ((variable 'let-binding))
  (default-toplevel-value 'variable))
    ⇒ global-value
```

`set-default-toplevel-value` *symbol value* [Function]
この関数は *symbol*のトップレベルのデフォルト値に指定された *value*をセットする。これはコードが *symbol*の let バインディングのコンテキスト下で実行中かどうかとは無関係に *symbol* のグローバル値をセットしたいときに便利。

12.12 ファイルローカル変数

ファイルにローカル変数の値を指定できます。そのファイルを visit しているバッファー内で、これらの変数にたいしてバッファーローカルなバインディングを作成するために、Emacs はこれらを使用します。ファイルローカル変数の基本的な情報については、Section “Local Variables in Files” in *The GNU Emacs Manual* を参照してください。このセクションではファイルローカル変数が処理される方法に影響する関数と変数を説明します。

ファイルローカル変数が勝手に関数や、後で呼び出される Lisp 式を指定できたら、ファイルの visit によって Emacs が乗っ取られてしまうかもしれません。Emacs は既知のファイルローカル変数だけにたいして、指定された値が安全だと自動的にセットすることにより、この危険から保護します。これ以外のファイルローカル変数は、ユーザーが同意した場合のみセットされます。

追加の安全策として Emacs がファイルローカル変数を読み込むとき、一時的に `read-circle`を `nil`にバインドします (Section 19.3 [Input Functions], page 334 を参照)。これは循環認識と共有された Lisp 構造から Lisp リーダーを保護します (Section 2.6 [Circular Objects], page 29 を参照)。

`enable-local-variables` [User Option]
この変数はファイルローカル変数を処理するかどうかを制御する。以下の値が利用できる:

`t`(デフォルト)
: 安全な変数をセット、安全でない変数は問い合わせる (1 回)。

`:safe`
: 安全な変数だけをセット、問い合わせはしない。

`:all`
: 問い合わせをせずに、すべての変数をセット。

`nil`
: 変数をセットしない。

その他
: すべての変数にたいして問い合わせる (1 回)。

`inhibit-local-variables-regexps` [Variable]
これは正規表現のリストである。ファイルがこのリストの要素にマッチする名前をもつなら、すべてのファイルローカル変数のフォームはスキャンされない。どんなときにこれを使いたいかの例は、Section 23.2.2 [Auto Major Mode], page 468 を参照のこと。

`hack-local-variables` **&optional** *handle-mode* [Function]
この関数はカレントバッファーの内容により指定された任意のローカル変数にたいしてパースを行い、適切にバインドと評価を行う。変数 `enable-local-variables`はここでも効果をもつ。

しかしこの関数は‘-*-’行の、‘mode:’ローカル変数を探さない。set-auto-modeはこれを行って enable-local-variablesも考慮する (Section 23.2.2 [Auto Major Mode], page 468 を参照)。

この関数は file-local-variables-alist内に格納された alist を調べて、各ローカル変数を順に適用することにより機能する。この関数は変数に適用する前 (か後) に、before-hack-local-variables-hook(か hack-local-variables-hook) を呼び出す。alist が非 nilの場合のみ、事前のフック (before-hook) を呼び出し、その他のフックは常に呼び出す。この関数はそのバッファーがすでにもつメジャーモードと同じメジャーモードが指定された場合は‘mode’要素を無視する。

オプションの引数 *handle-mode*が tなら、この関数が行うのはメジャーモードを指定するシンボルをリターンすることだけであり、‘-*-’行やローカル変数リストがメジャーモードを指定していればそのモード、それ以外は nilをリターンする。この関数はモードや他のファイルローカル変数をセットしない。*handle-mode*の値が nilと tのいずれでもなければ‘-*-’行の‘mode’に関するすべてのセッティングとローカル変数リストは無視されて、別のセッティングが適用される。*handle-mode*が nilならすべてのファイルローカル変数がセットされる。

file-local-variables-alist [Variable]
このバッファーローカルな変数は、ファイルローカル変数のセッティングの alist を保持する。alist の各要素は (*var* . *value*) という形式で、*var*はローカル変数のシンボル、*value*はその値である。Emacs がファイルを visit するとき、最初にすべてのファイルローカル変数をこの alist に収集して、その後で変数に 1 つずつ関数 hack-local-variablesを適用する。

before-hack-local-variables-hook [Variable]
Emacs は file-local-variables-alistに格納されたファイルローカル変数を適用する直前にこのフックを呼び出す。

hack-local-variables-hook [Variable]
Emacs は file-local-variables-alistに格納されたファイルローカル変数を適用し終えた直後にこのフックを呼び出す。

ある変数にたいして safe-local-variableプロパティーによって安全な値を指定できます。このプロパティーは引数を 1 つとる関数です。与えられた値にたいして、その関数が非 nilをリターンしたらその値は安全です。一般的に目にするファイル変数の多くは、safe-local-variableプロパティーをもちます。これらのファイル変数には fill-column、fill-prefix、indent-tabs-mode が含まれます。ブーリーン値の変数にたいしては、プロパティーの値に booleanpを使用します。

C ソースコード内で定義された変数に safe-local-variableプロパティを定義したければ、それらの変数の名前とプロパティを files.elのセクション “Safe local variables” のリストに追加してください。

defcustomを使用してユーザーオプションを定義する際には、defcustomに引数:safe *function*を追加して safe-local-variableプロパティをセットできます (Section 15.3 [Variable Definitions], page 253 を参照)。しかし:safeを使用して定義された安全性の述語は、その defcustomを含むパッケージのロード時の一度だけ認識されるものであり、それでは遅すぎることがしばしばあります。代替策としては、以下のようにオプションに安全性の述語を割り当てるために autoload クッキー (Section 16.5 [Autoload], page 274 を参照) を使用できます:

```
;;;###autoload (put 'var 'safe-local-variable 'pred)
```

autoloadで指定された安全な値の定義は、そのパッケージの autoload ファイル (Emacs に同梱されたパッケージのほとんどでは loaddefs.el) にコピーされて、セッションの開始から Emacs により認識されます。

`safe-local-variable-values` [User Option]
この変数はある変数の値が安全であることをマークする、別の方法を提供する。これはコンスセル (*var* . *val*)のリストであり *var*は変数名、*val*はその変数にたいして安全な値である。

Emacs が一連のファイルローカル変数にしたがうかどうかユーザーに尋ねるとき、ユーザーはそれらの変数が安全だとマークすることができる。安全とマークすると`safe-local-variable-values`にこれらの variable/value ペアーが追加されて、ユーザーのカスタムファイルに保存する。

`safe-local-variable-p` *sym val* [Function]
この関数は上記の条件に基づき、*sym*に値 *val*を与えても安全ななら非`nil`をリターンする。

いくつかの変数は危険 *(risky)* だと判断されます。ある変数が危険なら、その変数が`safe-local-variable-values`に自動的に追加されることはありません。ユーザーが`safe-local-variable-values`を直接カスタマイズすることで明示的に値を許さない限り、危険な変数をセットする前に Emacs は常に確認を求めます。

名前が非 `nil`の `risky-local-variable`プロパティーをもつすべての変数は危険だと判断されます。`defcustom`を使用してユーザーオプションを定義するとき、`defcustom`に引数`:risky` *value*を追加することにより、ユーザーオプションに `risky-local-variable`プロパティーをセットできます。それに加えて名前が '`-command`'、'`-frame-alist`'、'`-function`'、'`-functions`'、'`-hook`'、'`-hooks`'、'`-form`'、'`-forms`'、'`-map`'、'`-map-alist`'、'`-mode-alist`'、'`-program`'、'`-predicate`'で終わるすべての変数は自動的に危険だと判断されます。後に数字をともなう変数 '`font-lock-keywords`'と '`font-lock-keywords`'、さらには '`font-lock-syntactic-keywords`'も危険だと判断されます。

`risky-local-variable-p` *sym* [Function]
この関数は *sym*が上記の条件にもとづき危険な変数なら非`nil`をリターンする。

`ignored-local-variables` [Variable]
この変数はファイルによりローカル値を与えらるべきではない変数のリストを保持する。これらの変数に指定された任意の値は、完全に無視される。

"変数"'`Eval:`'も抜け道になる可能性があるので、Emacs は通常はそれを処理する前に確認を求めます。

`enable-local-eval` [User Option]
この変数は '`-*-`'の行中、または visit されるファイル内のローカル変数リストにたいする、'`Eval:`'の処理を制御する。値`t`は無条件に実行し、`nil`はそれらを無視することを意味します。それ以外なら各ファイルにたいして何を行うか、ユーザーに確認を求めることを意味する。デフォルト値は `maybe`。

`safe-local-eval-forms` [User Option]
この変数はファイルローカル変数リスト内で '`Eval:`'"変数" が見つかった際に評価しても安全な式のリストを保持する。

式が関数呼び出しであり、その関数が `safe-local-eval-function`プロパティーをもつなら、その式の評価が安全かどうかはそのプロパティー値が決定します。プロパティー値はその式をテストするための述語 (predicate)、そのような述語のリスト (成功した述語があれば安全)、または`t`(引数が定数である限り常に安全) を指定できます。

テキストプロパティーには、それらの値に関数呼び出しを含めることができるので抜け道になる可能性があります。したがって Emacs はファイルローカル変数にたいして指定された文字列値から、テキストプロパティーを取り除きます。

12.13 ディレクトリーローカル変数

ディレクトリーは、そのディレクトリー内のすべてのファイルに共通なローカル変数値を指定することができます。Emacs はそのディレクトリー内の任意のファイルを visit しているバッファー内で、それらの変数にたいするバッファーローカルなバインディングを作成するためにこれを使用します。これはそのディレクトリー内のファイルが何らかのプロジェクトに属していて、同じローカル変数を共有するときなどに有用です。

ディレクトリーローカル変数を指定するために 2 つの異なる方法があります: 1 つは特別なファイルにそれを記述する方法、もう 1 つはそのディレクトリーにプロジェクトクラス *(project class)* を定義する方法です。

`dir-locals-file` [Constant]

この定数は Emacs がディレクトリーローカル変数を見つけることができると期待するファイルの名前。ファイル名は`.dir-locals.el`[3]。ディレクトリー内でその名前をもつファイルにより Emacs はディレクトリー内の任意のファイル、または任意のサブディレクトリー (オプションでサブディレクトリーを除外できる。以下参照し) にセッティングを適用する。独自に`.dir-locals.el`をもつサブディレクトリーがある場合には、Emacs はサブディレクトリーで見つかったもっとも深いファイルのディレクトリーからディレクトリーツリーを上方に移動しながら、もっとも深いファイルのセッティングを使用する。この定数は 2 番目の dir-locals ファイル`.dir-locals-2.el`の名前を導出するためにも使用される。この 2 番目の dir-locals ファイルが与えられた場合には、そのファイルが`.dir-locals.el`に加えてロードされる。これは`.dir-locals.el`がバージョンコントロールの共有リポジトリの配下にあって個人のカスタマイズ用に使用できないときに有用。このファイルはローカル変数をフォーマットされたリストとして指定する。詳細は Section “Per-directory Local Variables” in *The GNU Emacs Manual* を参照のこと。

`hack-dir-local-variables` [Function]

この関数は`.dir-locals.el`ファイルを読み込み、そのディレクトリー内の任意のファイルを visit しているバッファーにローカルな`file-local-variables-alist`内に、それらを適用することなくディレクトリーローカル変数を格納する。この関数はディレクトリーローカルなセッティングも`dir-locals-class-alist`(`.dir-locals.el`ファイルが見つかったディレクトリーにたいする特別なクラスを定義する) 内に格納する。この関数は以下で説明するように、`dir-locals-set-class-variables`と`dir-locals-set-directory-class`を呼び出すことにより機能する。

`hack-dir-local-variables-non-file-buffer` [Function]

この関数はディレクトリーローカル変数を探して、即座にそれらをカレントバッファーに適用する。これは Dired バッファーのような、非ファイルバッファーをディレクトリーローカル変数のセッティングにしたがわせるために、モードコマンド呼び出しの中から呼び出されることを意図したものである。非ファイルバッファーにたいしては、Emacs は`default-directory`とその親ディレクトリーの中から、ディレクトリーローカル変数を探す。

[3] MS-DOS 版の Emacs は DOS ファイルシステムの制限により、かわりに`_dir-locals.el`という名前を使用します。

`dir-locals-set-class-variables` *class variables* [Function]
この関数は *class*という名前がつけられたシンボルにたいして一連の変数セッティングを定義する。その後はこのクラスを 1 つ以上のディレクトリーに割り当てることができるので、Emacs はこれらの変数セッティングをディレクトリー内のすべてのファイルに適用する。*variables*内のリストは 2 つの形式— (`major-mode . alist`)、または (`directory . list`) — のうちのいずれかをもつことができる。1 番目の形式ではそのファイルのバッファーが *major-mode* を継承するモードに切り替わるときに、連想リスト *alist*内のすべての変数が適用される。*alist* は (`name . value`)という形式。*major-mode*にたいする特別な値 `nil`は、そのセッティングが任意のモードに適用できることを意味する。*alist*内では特別な *name*として `subdirs`を使用することができる。連想値が `nil`なら alist は関連するディレクトリー内のファイルだけに適用されて、それらのサブディレクトリーには適用されない。

*variables*の 2 番目の形式では、*directory*がそのファイルのディレクトリーの最初のサブディレクトリーなら、上記のルールにしたがい *list*が再帰的に適用される。*list*はこの関数の *variables* で指定できる 2 つの形式のうち 1 つを指定する。

`dir-locals-set-directory-class` *directory class* **&optional** *mtime* [Function]
この関数は `directory`とサブディレクトリー内のすべてのファイルに *class*を割り当てる。その後、*class*にたいして指定されたすべての変数セッティングは、*directory*とその子ディレクトリー内で visit されたすべてのファイルに適用される。*class*は事前に `dir-locals-set-class-variables`で定義されていなければならない。

Emacs が`.dir-locals.el`ファイルからディレクトリー変数をロードする際、内部的にこの関数を使用する。その場合、オプションの引数 *mtime*はファイルの修正日時 (modification time。`file-attributes`によりリターンされる) を保持する。Emacs は記憶されたローカル変数がまだ有効化チェックするために、この日時を使用する。ファイルを介さず直接クラスを割り当てる場合、この引数は `nil`になる。

`dir-locals-class-alist` [Variable]
この alist はクラスシンボル (class symbol) とそれに関連づけられる変数のセッティングを保持する。これは `dir-locals-set-class-variables`により更新される。

`dir-locals-directory-cache` [Variable]
この alist はディレクトリー名、それらに割り当てられたクラス名、およびこのエントリーに関連するディレクトリーローカル変数ファイルの修正日時を保持する。関数 `dir-locals-set-directory-class`はこの list を更新する。

`enable-dir-local-variables` [Variable]
`nil`ならディレクトリーローカル変数は無視される。この変数はファイルローカル変数 (Section 12.12 [File Local Variables], page 197 を参照) にはしたがうが、ディレクトリーローカル変数は無視したいモードにたいして有用かもしれない。

12.14 接続ローカル変数

接続ローカル変数 (connection-local variable) はリモート接続をもつバッファにおいて、異なる変数セッティングにたいする汎用のメカニズムを提供します。これはそのリモート接続に専用のバッファーに応じてバインドおよびセットされる変数です。

`connection-local-set-profile-variables` *profile variables* [Function]
この関数は接続 *profile* (シンボル) にたいする一連の変数セッティングを定義する。この接続プロファイルに後から 1 つ以上のリモート接続を割り当てることができ、Emacs はそれらの

接続にたいするすべてのプロセスバッファーにそれらの変数セッティングを適用するだろう。*variables*内のリストは`(name . value)`という形式の alist。たとえば:

```
(connection-local-set-profile-variables
  'remote-bash
  '((shell-file-name . "/bin/bash")
    (shell-command-switch . "-c")
    (shell-interactive-switch . "-i")
    (shell-login-switch . "-l")))

(connection-local-set-profile-variables
  'remote-ksh
  '((shell-file-name . "/bin/ksh")
    (shell-command-switch . "-c")
    (shell-interactive-switch . "-i")
    (shell-login-switch . "-l")))

(connection-local-set-profile-variables
  'remote-null-device
  '((null-device . "/dev/null")))
```

`connection-local-profile-alist` [Variable]

この alist は接続プロファイルシンボルと連想変数セッティングを保持する。これは`connection-local-set-profile-variables`により更新される。

`connection-local-set-profiles` *criteria* **&rest** *profiles* [Function]

この関数は *criteria*で識別されるすべてのリモート接続に *profiles* (シンボル) を割り当てる。*criteria*は接続を識別する plist であり、アプリケーションはその接続を使用する。プロパティ名は`:application`、`:protocol`、`:user`、`:machine`のいずれか。`:application`のプロパティ値はシンボル、それ以外のプロパティ値は文字列。プロパティはすべてオプション。*criteria*が`nil`なら常に適用される。たとえば:

```
(connection-local-set-profiles
  '(:application 'tramp :protocol "ssh" :machine "localhost")
  'remote-bash 'remote-null-device)

(connection-local-set-profiles
  '(:application 'tramp :protocol "sudo"
    :user "root" :machine "localhost")
  'remote-ksh 'remote-null-device)
```

*criteria*が`nil`ならすべてのリモート接続に適用される。したがって上記の例は以下と等価

```
(connection-local-set-profiles
  '(:application 'tramp :protocol "ssh" :machine "localhost")
  'remote-bash)

(connection-local-set-profiles
  '(:application 'tramp :protocol "sudo"
    :user "root" :machine "localhost")
  'remote-ksh)
```

```
(connection-local-set-profiles
  nil 'remote-null-device)
```

*profiles*のすべてのプロファイルは`connection-local-set-profile-variables`で定義済みでなければならない。

`connection-local-criteria-alist` [Variable]
この alist は接続の criteria(判断基準) それに割り当てられたと profile の名前を含む。関数`connection-local-set-profiles`はこのリストを更新する。

`hack-connection-local-variables` *criteria* [Function]
この関数は`connection-local-variables-alist`内の*criteria*に関連する適用可能な接続ローカル変数を適用することなく収集する。たとえば:

```
(hack-connection-local-variables
  '(:application 'tramp :protocol "ssh" :machine "localhost"))

connection-local-variables-alist
     ⇒ ((null-device . "/dev/null")
        (shell-login-switch . "-l")
        (shell-interactive-switch . "-i")
        (shell-command-switch . "-c")
        (shell-file-name . "/bin/bash"))
```

`hack-connection-local-variables-apply` *criteria* [Function]
この関数は*criteria*に対応する接続ローカル変数を探してカレントバッファーに即座に適用する。

`with-connection-local-variables` **&rest** *body* [Macro]
`default-directory`で指定されるすべての接続ローカル変数を適用する。

その後に *body*を実行して接続ローカル変数を非バインド化する。たとえば:

```
(connection-local-set-profile-variables
  'remote-perl
  '((perl-command-name . "/usr/local/bin/perl")
    (perl-command-switch . "-e %s")))

(connection-local-set-profiles
  '(:application 'tramp :protocol "ssh" :machine "remotehost")
  'remote-perl)

(let ((default-directory "/ssh:remotehost:/working/dir/"))
  (with-connection-local-variables
    do something useful))
```

`enable-connection-local-variables` [Variable]
`nil`なら接続ローカル変数を無視する。この変数は特殊なモード内でのみ一時的に変更されるべきである。

12.15 変数のエイリアス

シノニムとして 2 つの変数を作成できれば便利なときがあります。2 つの変数は常に同じ値をもち、どちらか一方を変更すると、もう一方も変更されます。変数の名前を変更— 古い名前はよく考慮して選択されたものではなかったとか、変数の意味が部分的に変更された等の理由で — するとき、互換性のために新しい名前のエイリアス *(alias)* として古い名前を維持できれば便利なときがあるかもしれません。`defvaralias`によってこれを行うことができます。

`defvaralias` *new-alias base-variable* **&optional** *docstring* [Function]

この関数はシンボル *base-variable*のエイリアスとして、シンボル *new-alias*を定義する。これは *new-alias*から値を取得すると *base-variable*の値がリターンされ、*new-alias*の値を変更すると *base-variable*の値が変更されることを意味する。エイリアスされた 2 つの変数名は、常に同じ値と同じバインディングを共有する。

*docstring*引数が非`nil`なら、それは *new-alias*のドキュメント文字列を指定する。それ以外なら、エイリアスは (もしあれば)*base-variable*と同じドキュメント文字列となる。ただしそれは *base-variable*自体がエイリアスではない場合で、エイリアスなら *new-alias*はエイリアスチェーンの最後の変数のドキュメント文字列になる。

この関数は *base-variable*をリターンする。

変数のエイリアスは、変数にたいする古い名前を新しい名前に置き換える便利な方法です。`make-obsolete-variable`は古い名前を陳腐化 (obsolete) していると宣言して。それが将来のある時点で削除されるかもしれないことを宣言します。

`make-obsolete-variable` *obsolete-name current-name when* [Function]
&optional *access-type*

この関数はバイトコンパイラーに変数 *obsolete-name*が陳腐化していると警告させる。*current-name*がシンボルなら、それはこの変数の新たな名前である。警告メッセージはその後、*obsolete-name*のかわりに *current-name*を使用するよう告げるようになる。*current-name*が文字列なら、それはメッセージであり、置き換えられる変数はない。*when*はその変数が最初に陳腐化するのがいつかを示す文字列 (通常はバージョン番号文字列)。

オプションの引数 *access-type*が非`nil`なら、それは陳腐化の警告を引き起こすアクセスの種類を指定すること。`get`か`set`を指定できる。

2 つの変数シノニムを作成してマクロ`define-obsolete-variable-alias`を使用することにより、1 つが陳腐化していると同時に宣言できます。

`define-obsolete-variable-alias` *obsolete-name current-name* [Macro]
&optional *when docstring*

このマクロは変数 *obsolete-name*が陳腐化しているとマークして、それを変数 *current-name* にたいするエイリアスにする。これは以下と等価である:

```
(defvaralias obsolete-name current-name docstring)
(make-obsolete-variable obsolete-name current-name when)
```

`indirect-variable` *variable* [Function]

この関数は *variable*のエイリアスチェーンの最後の変数をリターンする。*variable*がシンボルでない、または *variable*がエイリアスとして定義されていなければ、この関数は *variable*をリターンする。

この関数はシンボルのチェーンがループしていたら、`cyclic-variable-indirection`エラーをシグナルする。

```
(defvaralias 'foo 'bar)
(indirect-variable 'foo)
     ⇒ bar
(indirect-variable 'bar)
     ⇒ bar
(setq bar 2)
bar
     ⇒ 2
foo
     ⇒ 2
(setq foo 0)
bar
     ⇒ 0
foo
     ⇒ 0
```

12.16 値を制限された変数

通常の Lisp 変数には、有効な Lisp オブジェクトである任意の値を割り当てることができます。しかし Lisp ではなく C で定義された Lisp 変数もあります。これらの変数のほとんどは、`DEFVAR_LISP`を使用して C コードで定義されています。Lisp で定義された変数と同様、これらは任意の値をとることができます。しかしいくつかの変数は`DEFVAR_INT`や`DEFVAR_BOOL`を使用して定義されています。C 実装の概要的な議論は、[Writing Emacs Primitives], page 1186、特に `syms_of_`*`filename`*型の関数の説明を参照してください。

`DEFVAR_BOOL`型の変数は、値に `nil`か `t`しかとることができません。他の値の割り当てを試みると `t`がセットされます:

```
(let ((display-hourglass 5))
  display-hourglass)
     ⇒ t
```

`byte-boolean-vars` [Variable]

この変数は`DEFVAR_BOOL`型のすべての変数のリストを保持する。

`DEFVAR_INT`型の変数は、整数値だけをとることができます。他の値の割り当てを試みると結果はエラーになります:

```
(setq undo-limit 1000.0)
error  Wrong type argument: integerp, 1000.0
```

12.17 ジェネリック変数

ジェネリック変数 *(generalized variable*: 汎変数*)* または *place form* は`setf`マクロ (Section 12.17.1 [Setting Generalized Variables], page 206 を参照) を使用して値が格納される、Lisp メモリー内の多くの場所のうちの 1 つです。1 番シンプルな place form は通常の Lisp 変数です。しかしリストの CAR と CDR、配列の要素、シンボルのプロパティー、その他多くのロケーション (location) も Lisp 値が格納される場所です。

ジェネリック変数は、C 言語の lvalues(左辺値) と類似しています。C 言語の lvalue では '`x = a[i]`'で配列から要素を取得し、同じ表記を使用して '`a[i] = x`'で要素を格納します。C では `a[i]`の

ような特定のフォームが lvalue になれるように、Lisp でジェネリック変数になることができる一連のフォームが存在します。

12.17.1 `setf`マクロ

`setf`マクロはジェネリック変数を操作するもっとも基本的な方法です。`setf`フォームは`setq`と似ていますが、シンボルだけでなく左辺の任意の place form を受け入れます。たとえば`(setf (car a) b)`は`a`の car を`b`にセットして`(setcar a b)`と同じ操作を行いますが、このタイプの place にセットやアクセスするために 2 つの関数を個別に覚える必要はありません。

`setf` [*place form*]... [Macro]
このマクロは *form*を評価して、それを *place*に格納する。*place*は有効なジェネリック変数フォームでなければならない。複数の *place*/*form*ペアーがある場合、割り当ては`setq`の場合と同様。`setf`は最後の *form*の値をリターンする。

以下の Lisp フォームは Emacs ではジェネリック変数として機能するフォームなので、`setf`の *place*引数にすることができます:

- シンボル。言い換えると、`(setf x y)`は完全に`(setq x y)`と正に等しく、厳密に言うと`setq`自体は`setf`が存在するので冗長です。これは純粋にスタイルと歴史的な理由によりますが、ほとんどのプログラマーは依然として単純な変数へのセットには`setq`の方を好みます。マクロ`(setf x y)`は実際には`(setq x y)`に展開されるので、コンパイルされたコードでこれを使用することにパフォーマンス的な不利はありません。
- 以下の標準的な Lisp 関数の呼び出し:

```
aref      cddr      symbol-function
car       elt       symbol-plist
caar      get       symbol-value
cadr      gethash
cdr       nth
cdar      nthcdr
```

- 以下の Emacs 特有な関数の呼び出し:

```
alist-get                   process-get
frame-parameter             process-sentinel
terminal-parameter          window-buffer
keymap-parent               window-display-table
match-data                  window-dedicated-p
overlay-get                 window-hscroll
overlay-start               window-parameter
overlay-end                 window-point
process-buffer              window-start
process-filter              default-value
```

どのように処理すれば良いか未知な *place*フォームを渡すと、`setf`はエラーをシグナルします。

`nthcdr`の場合、関数のリスト引数はそれ自体が有効な *place*フォームでなければならないことに注意してください。たとえば`(setf (nthcdr 0 foo) 7)`は、`foo`自体に 7 をセットするでしょう。

マクロ`push`(Section 5.5 [List Variables], page 79 を参照) と`pop`(Section 5.3 [List Elements], page 73 を参照) は、リストだけでなくジェネリック変数を操作できます。`(pop` *place*`)`は *place*内に格納されたリストの最初の要素を削除してリターンします。これは`(prog1 (car` *place*`) (setf` *place* `(cdr` *place*`)))`と似ていますが、すべてのサブフォームを一度だけ評価します。`(push` *x place*`)`は *place*内に格納されたリストの 1 番前に *x*を挿入します。これは`(setf` *place* `(cons` *x place*`))`と似ていますが、サブフォームの評価が異なります。`nthcdr` place への`push`と`pop`は、リスト内の任意の位置での挿入ち削除に使用できることに注意してください。

cl-libライブラリーでは追加の setf place を含む、ジェネリック変数にたいするさまざまな拡張が定義されています。Section “Generalized Variables” in *Common Lisp Extensions* を参照してください。

12.17.2 新たな setfフォーム

このセクションでは、setfが操作できる新たなフォームの定義方法を説明します。

gv-define-simple-setter *name setter* **&optional** *fix-return* [Macro]

このマクロは単純なケースで setfメソッドを簡単に定義することを可能にする。*name*は関数、マクロ、スペシャルフォームの名前。*name*がそれを更新するための対応する *setter*関数をもつなら、このマクロを使用できる (たとえば (gv-define-simple-setter car setcar))。

このマクロは以下のフォームの呼び出しを

```
(setf (name args...) value)
```

以下のように変換する。

```
(setter args... value)
```

このような setfの呼び出しは *value*をリターンするとドキュメントされている。これは carと setcarでは問題はない。setcarはそれがセットする値をリターンするからである。*setter*関数が *value*をリターンしない場合には、gv-define-simple-setterの *fix-return*引数に、非 nil値を使用すること。これは以下のようなものに展開される

```
(let ((temp value))
  (setter args... temp)
  temp)
```

これで正しい結果がリターンされることが保証される。

gv-define-setter *name arglist* **&rest** *body* [Macro]

このマクロは上述のフォームより複雑な setf展開を可能にする。たとえば呼び出すべきシンプルな setter 関数が存在しないときや、もしそれが存在しても place フォームとは異なる引数を要求するなら、このフォームを使う必要があるかもしれない。

このマクロは最初に setf引数フォーム (*value args...*)を *arglist*にバインドして、その後 *body*を実行することによって、フォーム (setf (*name args...*) *value*)を展開する。*body*は割り当てを行う Lisp フォームをリターンして、最終的にはセットされた値をリターンすること。以下はこのマクロの使用例:

```
(gv-define-setter caar (val x) `(setcar (car ,x) ,val))
```

gv-define-expander *name handler* [Macro]

展開をより詳細に制御するために gv-define-expanderマクロが使用できる。たとえばセット可能な substringは以下の方法で実装できる:

```
(gv-define-expander substring
  (lambda (do place from &optional to)
    (gv-letplace (getter setter) place
      (macroexp-let2* nil ((start from) (end to))
        (funcall do `(substring ,getter ,start ,end)
                 (lambda (v)
                   (funcall setter `(cl--set-substring
                                     ,getter ,start ,end ,v))))))))
```

`gv-letplace` (*getter setter*) *place* **&rest** *body* [Macro]

マクロ `gv-letplace`は `setf`のような処理を行うマクロを定義するのに有用。たとえば Common Lisp の `incf`マクロは以下の方法で実装できる:

```
(defmacro incf (place &optional n)
  (gv-letplace (getter setter) place
    (macroexp-let2 nil v (or n 1)
      (funcall setter `(+ ,v ,getter)))))
```

*getter*は *place*の値をリターンするコピー可能な式にバインドされる。*setter*は式 *v*を受け取り、*place*に *v*をセットする新たな式をリターンする関数にセットされる。*body*は *getter*と *setter*を介して *place*を操作する Emacs Lisp 式をリターンすること。

詳細は `gv.el`のソースファイルを参照。

Common Lisp に関する注意**:** Common Lisp は関数としての `setf`、すなわち関数名がシンボルではなくリスト `(setf` *`name`*`)`であるような `setf`関数の挙動を指定するために別の方法を定義する。たとえば `(defun (setf foo) ...)`は、`setf`が `foo`に適用されるときに使用される関数を定義する。Emacs はこれをサポートしない。適切な展開が定義されていないフォームに `setf`を使用するとコンパイル時エラーとなる。Common Lisp では後で関数 `(setf` *`func`*`)`が定義されるのでエラーにならない。

13 関数

Lisp プログラムは主に Lisp 関数で構成されます。このチャプターはで関数とは何か、引数を受け取る方法、そして関数を定義する方法を説明します。

13.1 関数とは?

一般的な意味では関数とは引数 *(arguments)* と呼ばれる与えられた入力値の計算を担うルールです。計算の結果は関数の値 *(value)*、または *return* 値 *(return value)* と呼ばれます。計算は変数の値やデータ構造の内容を変更する等の副作用をもつこともできます ([Definition of side effect], page 133 を参照)。純粋関数 *(pure function)* とは、それらに加えて副作用をもたず、機種別やシステム状態のような外部要因とは無関係に同じ条件の引数にたいして常に同一の値をリターンする関数のことです。

ほとんどのコンピューター言語では、関数はそれぞれ名前をもちます。しかし Lisp では厳密な意味において関数は名前をもちません。関数はオブジェクトであり、関数の名前の役割を果たすシンボルに関連づけることができますが (たとえば `car`)、それはオプションです。Section 13.3 [Function Names], page 214 を参照してください。関数が名前を与えられたとき、通常はそのシンボルを “関数” として参照します (たとえば関数 `car`のように参照する)。このマニュアルでは、関数名と関数オブジェクト自身との間の区別は通常は重要ではありませんが、それが意味をもつような場合には注記します。

スペシャルフォーム *(special form)*、マクロ *(macro)* と呼ばれる関数 like なオブジェクトがいくつかあり、それらも引数を受け取って計算を行います。しかし以下で説明するように Emacs Lisp ではこれらは関数とはみなされません。

以下は関数と関数 like なオブジェクトにたいする重要な条件です:

lambda expression
: Lisp で記述された関数 (厳密には関数オブジェクト)。これらについては以降のセクションで説明します。

primitive
: Lisp から呼び出すことができるが実際には C で記述されている。プリミティブはビルトイン関数 *(built-in functions)* とかサブルーチン *(subr)* のようにも呼ばれる。それらの例には関数 like な `car`や `append`が含まれる。加えてすべてのスペシャルフォーム (以下参照) もプリミティブとみなされる。

  関数は Lisp の基礎となる部分 (たとえば `car`) であり、オペレーティングシステムのサービスにたいして低レベルのインターフェースを与え、高速に実行される必要があるために、通常はプリミティブとして実装されている。Lisp で定義された関数と異なり、プリミティブの修正や追加には、C ソースの変更と Emacs のリコンパイルが必要となる。Section E.7 [Writing Emacs Primitives], page 1184 を参照のこと。

special form
: プリミティブは関数と似ているが、すべての引数が通常の方法で評価されない。いくつかの引数だけが評価されるかもしれず、通常ではない順序で評価されるか、複数回評価されるかもしれない。プリミティブの例には `if`、`and`、`while`が含まれる。Section 10.1.7 [Special Forms], page 137 を参照のこと。

macro
: ある Lisp 式をオリジナルの式のかわりに評価される別の式に変換する、関数とは別の Lisp で定義された構文。マクロはスペシャルフォームが行う一連のことを、Lisp プログラマーが行うのを可能にする。Chapter 14 [Macros], page 242 を参照のこと。

command プリミティブ `command-execute`を通じて呼び出すことができるオブジェクトで、通常はそのコマンドにバインドされたキーシーケンスをユーザーがタイプすることにより呼び出される。Section 21.3 [Interactive Call], page 382 を参照のこと。コマンドは通常は関数である。その関数が Lisp で記述されていれば、関数の定義内の `interactive` フォームによってコマンドとなる (Section 21.2 [Defining Commands], page 376 を参照)。関数であるコマンドは他の関数と同様、Lisp 式から呼び出すこともできる。

キーボードマクロ (文字列かベクター) は関数ではないが、これらもコマンドである。Section 21.16 [Keyboard Macros], page 420 を参照のこと。シンボルの関数セルにコマンドが含まれてれば、わたしたちはそのシンボルをコマンドと言う (Section 9.1 [Symbol Components], page 124 を参照)。そのような名前つきコマンド *(named command)* は `M-x`で呼び出すことができる。

closure ラムダ式とよく似た関数オブジェクトだが、クロージャーはレキシカル変数バインディングの環境にも囲われている。Section 13.10 [Closures], page 227 を参照のこと。

byte-code function
バイトコンパイラーによりコンパイル済みの関数。Section 2.4.16 [Byte-Code Type], page 23 を参照のこと。

autoload object
実際の関数のプレースホルダー。autoload オブジェクトが呼び出されると、Emacs は実際の関数の定義を含むファイルをロードした後に実際の関数を呼び出す。Section 16.5 [Autoload], page 274 を参照のこと。

関数 `functionp`を使用して、あるオブジェクトが関数かどうかテストできます:

`functionp` *object* [Function]
この関数は *object*が任意の種類の関数 (`funcall`に渡すことができる) なら `t`をリターンする。`functionp`は関数を名づけるシンボルにたいしては `t`、スペシャルフォームにたいしては `nil` をリターンすることに注意。

任意の関数が期待する引数の個数を調べることもできます:

`func-arity` *function* [Function]
この関数は指定された *function*の引数リストに関する情報を提供する。リターン値は `(`*min* `.` *max*`)` という形式のコンスセル。ここで *min*は引数の最小個数、*max*は引数の最大個数、または`&rest`引数をもつ関数では `many`、*function*がスペシャルフォームならシンボル `unevalled`。

以下のようにある状況下ではこの関数は不正確な結果をリターンすることに注意:

- `apply-partially`を使用して定義された関数 (Section 13.5 [Calling Functions], page 217 を参照)。
- `advice-add`を使用して定義された関数 (Section 13.11.2 [Advising Named Functions], page 230 を参照)。
- コードの一部として引数リストを直接判断する関数。

`functionp`と異なり、以下の 3 つの関数はシンボルをそれの関数定義としては扱いません。

`subrp` *object* [Function]
この関数は *object*がビルトイン関数 (たとえば Lisp プリミティブ) なら `t`をリターンする。

```
(subrp 'message)            ; messageはシンボルであり、
     ⇒ nil                  ;    subr オブジェクトではない
```

```
(subrp (symbol-function 'message))
     ⇒ t
```

`byte-code-function-p` *object* [Function]

この関数は *object*がバイトコード関数なら `t`をリターンする。たとえば:

```
(byte-code-function-p (symbol-function 'next-line))
     ⇒ t
```

`subr-arity` *subr* [Function]

これは `func-arity`と同様だがシンボルインダイレクションなしのビルトイン関数にたいしてのみ機能する。非ビルトイン関数にたいしてはエラーをシグナルする。かわりに `func-arity`の使用を推奨する。

13.2 ラムダ式

ラムダ式 (lambda expression) は Lisp で記述された関数オブジェクトです。以下は例です:

```
(lambda (x)
  "X の双曲線コサインを return する"
  (* 0.5 (+ (exp x) (exp (- x)))))
```

Emacs Lisp ではこのようなリストは、関数オブジェクトに評価される有効な式です。

ラムダ式自身は名前をもたない無名関数 *(anonymous function)* です。ラムダ式をこの方法で使用できますが (Section 13.7 [Anonymous Functions], page 220 を参照)、名前付き関数 *(named functions)* を作成するためにシンボルに関連付けられる方が一般的です (Section 13.3 [Function Names], page 214 を参照)。これらの詳細に触れる前に以下のサブセクションではラムダ式の構成要素と、それらが行うことについて説明します。

13.2.1 ラムダ式の構成要素

ラムダ式は以下のようなリストです:

```
(lambda (arg-variables...)
  [documentation-string]
  [interactive-declaration]
  body-forms...)
```

ラムダ式の 1 番目の要素は常にシンボル `lambda`です。これはそのリストが関数を表すことを示します。`lambda`で関数定義を開始する理由は、別の目的での使用が意図された他のリストが、意図せずに関数として評価されないようにするためです。

2 番目の要素はシンボル — 引数変数名のリストです。これはラムダリスト *(lambda list)* と呼ばれます。Lisp 関数が呼び出されたとき、引数値はラムダリスト内の変数と対応付けされます。ラムダリストには、与えられた値にたいするローカルバインディングが付与されます。Section 12.3 [Local Variables], page 175 を参照してください。

ドキュメント文字列 (documentation string) は Emacs Lisp のヘルプ機能にたいして、その関数を説明する関数定義に配された Lisp の文字列オブジェクトです。Section 13.2.4 [Function Documentation], page 213 を参照してください。

インタラクティブ宣言 (interactive declaration) は、`(interactive` *code-string*`)`という形式のリストです。これはこの関数が対話的に使用された場合に引数を提供する方法を宣言します。この宣言をもつ関数は、コマンド *(command)* と呼ばれます。コマンドは *M-x*を使用したり、キーにバインドして呼び出すことができます。この方法で呼び出されることを意図しない関数は、インタラクティ

ブ宣言を持つべきではありません。インタラクティブ定義を記述する方法は、Section 21.2 [Defining Commands], page 376 を参照してください。

残りの要素はその関数の *body(本体)* — その関数が処理を行うための Lisp コード (Lisp プログラマーは “評価される Lisp フォームのリスト” と言うだろう) です。この関数からリターンされる値は、body の最後の要素によりリターンされる値です。

13.2.2 単純なラムダ式の例

以下の例を考えてみてください:

```
(lambda (a b c) (+ a b c))
```

以下のように `funcall`に渡すことにより、この関数を呼び出すことができます:

```
(funcall (lambda (a b c) (+ a b c))
         1 2 3)
```

この呼び出しは変数 `a`に 1、`b`に 2、`c`に 3 をバインドして、ラムダ式の body を評価します。body の評価によってこれら 3 つの数が加算されて、6 が結果として生成されます。したがってこの関数呼び出しにより 6 がリターンされます。

以下のように引数は他の関数の結果であってもよいことに注意してください:

```
(funcall (lambda (a b c) (+ a b c))
         1 (* 2 3) (- 5 4))
```

これは引数 `1`、`(* 2 3)`、`(- 5 4)`を左から右に評価します。その後ラムダ式に引数 1、6、1 を適用して値 8 が生成されます。

これらの例が示すように、ローカル変数を作成してそれらに値を与えるフォームとして、CAR がラムダ式であるようなフォームを使用することができます。古い時代の Lisp では、この方法がローカル変数をバインドして初期化する唯一の方法でした。しかし現在ではこの目的にはフォーム `let`を使用するほうが明解です (Section 12.3 [Local Variables], page 175 を参照)。ラムダ式は主に他の関数の引数として渡される無名関数 (Section 13.7 [Anonymous Functions], page 220 を参照) として、あるいは名前つき関数 (Section 13.3 [Function Names], page 214 を参照) を生成するためにシンボルの関数定義に格納するために使用されます。

13.2.3 引数リストのその他機能

シンプルなサンプル関数 `(lambda (a b c) (+ a b c))`は 3 つの引数変数を指定しているので、3 つの引数で呼び出されなければなりません。引数を 2 つしか指定しなかったり 4 つ指定した場合には `wrong-number-of-arguments`エラーとなります (Section 11.7.3 [Errors], page 164 を参照)。

特定の引数を省略できる関数を記述できると便利なこともあります。たとえば関数 `substring`は 3 つの引数 — 文字列、開始インデックス、終了インデックス — を受け取りますが、3 つ目の引数を省略すると、デフォルトでその文字列の *length*となります。関数 `list`や`+`が行うように、特定の関数にたいして不定個の引数を指定できると便利なときもあります。

関数が呼び出されるとき省略されるかもしれないオプションの引数を指定するには、オプションの引数の前にキーワード`&optional`を含めるだけです。0 個以上の追加引数のリストを指定するには、最後の引数の前にキーワード`&rest`を含めます。

したがって引数リストの完全な構文は以下のようになります:

```
(required-vars...
 [&optional [optional-vars...]]
 [&rest [rest-var]])
```

角カッコ (square bracket) は`&optional`と`&rest`、およびそれらに続く変数が省略できることを示します。

この関数の呼び出しでは *required-vars*のそれぞれにたいして、実際の引数が要求されます。0 個以上の *optional-vars*にたいして実際の引数があるかもしれませんが、ラムダ式が`&rest`を使用していなければ、その個数を超えて実際の引数を記述することはできません。`&rest`が記述されていれば、追加で任意の個数の実際の引数があるかもしれません。

optiona や rest 変数にたいして実際の引数が省略されると、それらのデフォルトは常に `nil`になります。関数にたいして引数に明示的に `nil`が使用されたのか、引数が省略されたのかを区別することはできません。しかし関数の body が、`nil`を他の有意な値が省略されたと判断することは自由です。`substring`はこれを行います。`substring`の 3 つ目の引数が `nil`なら、それは文字列の長さを使用することを意味します。

> **Common Lisp** に関する注意**:** Common Lisp ではオプションの引数が省略されたときに使用するデフォルト値を指定できる。Emacs Lisp では、引数が明示的に渡されたかを調べる `supplied-p`変数はサポートされない。

例えば引数リストは以下のようになります:

```
(a b &optional c d &rest e)
```

これは `a`と `b`は最初の 2 つの実引数となり、これらは必須です。さらに 1 つまたは 2 つの引数が指定された場合には、それらは順番に `c`と `d`にバインドされます。1 つ目から 4 つ目の引数の後の引数はリストにまとめられて、`e`にそのリストがバインドされます。したがって 2 つしか引数が指定されなかった場合には `c`、`d`、`e`は `nil`になります。2 つまたは 3 つの引数の場合には `d`と `e`は `nil`です。引数が 4 つ以下の場合には、`e`は `nil`になります。正確に 5 つの引数に明示的に `nil`が指定された場合には、`e`にたいして他の単一値が与えられたときのように、引数の `nil`が 1 要素のリスト `(nil)`として `e`に与えられます。

オプションの引数の後ろに必須の引数を指定する方法はありません — これは意味を成さないからです。なぜそうなるかは、この例で `c`がオプションで `d`が必須な場合を考えてみてください。実際に 3 つの引数が与えられたとします。3 番めの引数は何を指定したのでしょうか? この引数は *c*なのでしょうか、それとも *d*に使用されるのでしょうか? 両方の場合が考えられます。同様に`&rest`引数の後に、さらに引数 (必須またはオプション) をもつことも意味を成しません。

以下に引数リストと、それを正しく呼び出す例をいくつか示します:

```
(funcall (lambda (n) (1+ n))          ; 1 つの必須:
         1)                           ; これは正確に 1 つの引数を要求する
     ⇒ 2
(funcall (lambda (n &optional n1)     ; 1 つは必須で、1 つはオプション:
           (if n1 (+ n n1) (1+ n)))   ; 1 つまたは 2 つの引数
         1 2)
     ⇒ 3
(funcall (lambda (n &rest ns)         ; 1 つは必須で、後は残り:
           (+ n (apply '+ ns)))       ; 1 つ以上の引数
         1 2 3 4 5)
     ⇒ 15
```

13.2.4 関数のドキュメント文字列

ラムダ式はラムダリストの直後に、オプションでドキュメント文字列 *(documentation string)* をもつことができます。この文字列は、その関数の実行に影響を与えません。これはコメントの一種ですが Lisp 機構に内在するシステム化されたコメントであり。Emacs のヘルプ機能で使用できます。ド

キュメント文字列にアクセスする方法は、Chapter 24 [Documentation], page 519 を参照してください。

たとえその関数があなたのプログラム内だけで呼び出される関数だとしても、すべての関数にドキュメント文字列を与えるのはよいアイデアです。ドキュメント文字列はコメントと似ていますが、コメントより簡単にアクセスできます。

ドキュメント文字列の 1 行目は、関数自体にもとづくものであるべきです。なぜなら `apropos`は、最初の 1 行目だけを表示するからです。ドキュメント文字列の 1 行目は、その関数の目的を要約する 1 つか 2 つの完全なセンテンスで構成されるべきです。

ドキュメント文字列の開始は通常、ソースファイル内ではインデントされていますが、ドキュメント文字列の開始のダブルクォート文字の前にインデントのスペースがあるので、インデントはドキュメント文字列の一部にはなりません。ドキュメント文字列の残りの行がプログラムソース内で揃うようにインデントする人がいます。これは間違いです。後続の行のインデントは文字列の内部にあります。これはソースコード内での見栄えはよくなりますが、ヘルプコマンドで表示したとき見栄えが悪くなります。

ドキュメント文字列がなぜオプションになるのか不思議に思うかもしれません。なぜならドキュメント文字列の後には必須となる関数の構成要素である body が続くからです。文字列を評価するとその文字列自身がリターンされるので、それが body 内の最後のフォームでない限りなんの効果もありません。したがって実際は body の 1 行目とドキュメント文字列で混乱が生じることはありません。body の唯一のフォームが文字列なら、それはリターン値とドキュメントの両方の役目を果たします。

ドキュメント文字列の最後の行には、実際の関数引数とは異なる呼び出し規約を指定できます。これは以下のようなテキストを記述します

```
\(fn arglist)
```

そのテキストの後に空行を配置して、テキスト自身は行頭から記述、ドキュメント文字列内でこのテキストの後に改行が続かないように記述します (‘\’は Emacs の移動コマンドが混乱するのを避けるために使用する)。この方法で指定された呼び出し規約は、ヘルプメッセージ内で関数の実引数から生成される呼び出し例と同じ場所に表示されます。

マクロ定義内に記述された引数は、ユーザーがマクロ呼び出しの一部だと考える方法とは合致しない場合がしばしばあるので、この機能はマクロ定義で特に有用です。

呼び出し規約を廃止して上記の仕様で示す規約を公開したければ、この機能を使用してはなりません。かわりに廃止された呼び出し規約を使用する Lisp プログラムのバイトコンパイル時に警告メッセージを発する宣言 `advertised-calling-convention` (Section 13.14 [Declare Form], page 237 を参照) か `set-advertised-calling-convention` (Section 13.12 [Obsolete Functions], page 234 を参照) を使用してください。

13.3 関数の命名

シンボルは関数の名前となることができます。これはそのシンボルの関数セル *(function cell*: Section 9.1 [Symbol Components], page 124 を参照) が、関数オブジェクト (たとえばラムダ式) を含むときに起こります。するとそのシンボル自身が呼び出し可能な有効な関数、つまりそのシンボルの関数セルの関数と等価になります。

関数セルの内容はそのシンボルの関数定義 *(function definition)* と呼ぶこともできます。そのシンボルのかわりにシンボルの関数定義を使う手続きのことをシンボル関数インダイレクション *(symbol function indirection)* と呼びます。Section 10.1.4 [Function Indirection], page 135 を参照。与えられたシンボルに関数定義がなければシンボルの関数セルは *void* と呼ばれ、それを関数として使用することはできません。

実際のところほとんどすべての関数は名前をもち、その名前により参照されます。ラムダ式を定義することで名前つきの Lisp 関数を作成、それを関数セル (Section 13.9 [Function Cells], page 225 を参照) に置くことができます。しかしより一般的なのは `defun`スペシャルフォーム (次のセクションで説明) を使う方法です。

わたしたちが関数に名前を与えるのは、Lisp 式内で関数を名前で参照するのが便利だからです。また名前つきの関数は簡単に自分自身を — 再帰的 (recursive) に参照することができます。さらにプリミティブはテキスト的な名前だけで参照することができます。なぜならプリミティブ関数は入力構文 (read syntax) をもたないオブジェクトだからです (Section 2.4.15 [Primitive Function Type], page 23 を参照)。

関数が一意な名前をもつ必要はありません。与えられた関数オブジェクトは通常は 1 つのシンボルの関数セルだけに存在しますが、これは単に慣習的なものです。`fset`を使用すれば関数を複数のシンボルに格納するのは簡単です。それらのシンボルはそれぞれ、同じ関数にたいする有効な名前となります。

関数として使用しているシンボルを、変数としても利用できることに注意してください。シンボルのこれら 2 つの利用法は独立しており、競合はしません (これは Schema のような他のいくつかの Lisp 方言には当てはまらない)。

慣例により関数のシンボルが '`--`'で分割される 2 つの名前で構成される場合には、その関数は内部的な使用を意図しており、名前の最初の部分は関数を定義するファイルです。たとえば `vc-git--rev-parse`という名前の関数は `vc-git.el`で定義される内部関数です。C で記述された内部関数は `bury-buffer-internal`のように名前が '`-internal`'で終わります。2018 年より前に貢献された Emacs コードは内部的な使用にたいして別の命名規約を使用するかもしれませんが、これらは徐々に廃止されます。

13.4 関数の定義

わたしたちは通常は関数を最初に作成したときに名前を与えます。これは関数の定義 *(defining a function)* と呼ばれており、通常は `defun`マクロにより行われます。このセクションでは関数を定義する別の方法も説明します。

`defun` *name args* [*doc*] [*declare*] [*interactive*] *body*. . . [Macro]

`defun`は新たな Lisp 関数を定義する通常の方法である。これは引数リスト *args*、および *body* により与えられる body フォームとともに、シンボル *name*を関数として定義する。*name*と *args*をクォートする必要はない。

*doc*が与えられたら、それはその関数のドキュメント文字列を指定する文字列であること (Section 13.2.4 [Function Documentation], page 213 を参照)。*declare*が与えられたら、それは関数のメタデータを指定する `declare`フォームであること (Section 13.14 [Declare Form], page 237 を参照)。*interactive*が与えられたら、それは関数が対話的に呼び出される方法を指定する `interactive`フォームであるこ (Section 21.3 [Interactive Call], page 382 を参照)。

`defun`のリターン値は定義されていません。

以下にいくつか例を示す:

```
(defun foo () 5)
(foo)
     ⇒ 5
```

```
(defun bar (a &optional b &rest c)
    (list a b c))
(bar 1 2 3 4 5)
     ⇒ (1 2 (3 4 5))
(bar 1)
     ⇒ (1 nil nil)
(bar)
error  Wrong number of arguments.

(defun capitalize-backwards ()
  "Upcase the last letter of the word at point."
  (interactive)
  (backward-word 1)
  (forward-word 1)
  (backward-char 1)
  (capitalize-word 1))
```

意図せず既存の関数を再定義しないように注意されたい。`defun`は`car`のようなプリミティブ関数でさえ、問い合わせせずに躊躇なく再定義する。Emacs がこれを妨げることはない。なぜなら関数の再定義は故意に行われることがあり、そのような意図した再定義を、意図しない再定義と見分ける方法はがないからである。

`defalias` *name definition* **&optional** *doc* [Function]

この関数は定義 *definition*(任意の有効な Lisp 関数) とともに、シンボル *name*を関数として定義する。この関数のリターン値は未定義。

*doc*が非`nil`なら、それは関数 *name*のドキュメントとなる。それ以外なら *definition*により提供されるドキュメントが使用される。

内部的には`defalias`は、通常は定義のセットに`fset`を使用する。しかし *name*が`defalias-fset-function`プロパティーをもつなら、`fset`を呼び出すかわりにそれに割り当てられた値を使用する。

`defalias`を使う正しい場所は、特定の関数名が正に定義される場所 — 特にソースファイルがロードされるとき明示的にその名前が出現する場所である。これは`defalias`が`defun`と同じように、どれが関数を定義するファイルなのか記録するからである (Section 16.9 [Unloading], page 282 を参照)。

それとは対象的に他の目的のために関数を操作するプログラムでは、そのような記録を保持しない`fset`を使用するほうがよいだろう。Section 13.9 [Function Cells], page 225 を参照のこと。

`defun`や`defalias`で新たなプリミティブ関数を作成することはできませんが、任意の関数定義を変更するのに使用することができ、通常の定義がプリミティブである`car`や`x-popup-menu`のような関数でさえ変更することができます。しかしこれは危険なことです。たとえば Lisp の完全性を損なうことなく、`car`を再定義するのはほとんど不可能だからです。それほど有名ではない`x-popup-menu`のような関数の再定義では、危険は減少しますが、それでも期待したとおりに機能しないかもしれません。C コードにそのプリミティブの呼び出しがあれば、それは直接そのプリミティブの C 定義を呼び出すので、シンボル定義を変更してもそれらに影響はありません。

`defsubst`も参照してください。これは`defun`のように関数を定義して、それのインライン展開を処理するよう Lisp コンパイラーに指示します。Section 13.13 [Inline Functions], page 235 を参照してください。

関数名を未定義にするには`fmakunbound`を使用します。Section 13.9 [Function Cells], page 225 を参照してください。

13.5 関数の呼び出し

関数を定義しただけでは半分しか終わっていません。関数はそれを呼び出す *(call)* — たとえば実行 (run) するまでは何も行いません。関数の call は *invocation* としても知られています。

関数を呼び出すもっとも一般的な方法は、リストの評価によるものです。たとえばリスト`(concat "a" "b")`を評価することにより、関数`concat`が引数`"a"`と`"b"`で呼び出されます。評価については Chapter 10 [Evaluation], page 133 を参照してください。

プログラム内で式としてリストを記述するときは、プログラム内にテキストでどの関数を呼び出すか、いくつの引数を与えるかを指定します。通常はこれが行いたいことです。どの関数を呼び出すかを実行時に計算する必要がある場合もあります。これを行うには関数`funcall`を使用します。実行時にいくつの引数を渡すか決定する必要があるときは`apply`を使用します。

`funcall` *function* **&rest** *arguments* [Function]

`funcall`は関数 *function*を引数 *arguments*で呼び出して、*function*がリターンした値をリターンする。

`funcall`は関数なので、*function*を含むすべての引数は`funcall`の呼び出し前に評価される。これは呼び出される関数を得るために任意の式を使用できることを意味している。また`funcall`が *arguments*に記述した式ではなく、その値だけを見ることを意味している。これらの値は *function*呼び出し中では、2 回目は評価されない。`funcall`の処理は関数の通常の呼び出し手続きと似ており、すでに評価された引数は評価されない。

引数 *function*は Lisp 関数かプリミティブ関数でなければならない。つまりスペシャルフォームやマクロは、未評価の引数式を与えられたときだけ意味があるので、指定することはできない。上述したように最初の場所で`funcall`がそれらを知らないので、`funcall`がそれらを提供することはできない。

コマンドの呼び出しに`funcall`を使用して、それがインタラクティブに呼び出されたように振る舞うようにする必要があるなら、`funcall-interactively`を使用すること (Section 21.3 [Interactive Call], page 382 を参照)。

```
(setq f 'list)
     ⇒ list
(funcall f 'x 'y 'z)
     ⇒ (x y z)
(funcall f 'x 'y '(z))
     ⇒ (x y (z))
(funcall 'and t nil)
error  Invalid function: #<subr and>
```

これらの例を`apply`の例と比較されたい。

`apply` *function* **&rest** *arguments* [Function]

`apply`は関数 *function*を引数 *arguments*で呼び出す。これは`funcall`と同様だが 1 つ違いがある。*arguments*の最後はオブジェクトのリストである。これは 1 つのリストではなく、個別の引数として *function*に渡される。わたしたちはこれを、`apply`がこのリストを展開 *(spread)*(個々の要素が引数となるので) すると言う。

`apply`は *function*を呼び出した結果をリターンする。`funcall`と同様、*function*は Lisp 関数かプリミティブ関数でなければならない。つまりスペシャルフォームやマクロは `apply`では意味をもたない。

```
(setq f 'list)
     ⇒ list
(apply f 'x 'y 'z)
error  Wrong type argument: listp, z
(apply '+ 1 2 '(3 4))
     ⇒ 10
(apply '+ '(1 2 3 4))
     ⇒ 10

(apply 'append '((a b c) nil (x y z) nil))
     ⇒ (a b c x y z)
```

`apply`を使用した興味深い例は [Definition of mapcar], page 219 を参照のこと。

ある関数にたいして、その関数のある引数を特定の値に固定して、他の引数は実際に呼びだされたときの値にできれば便利なことがあります。関数のいくつかの引数を固定することは、その関数の部分適用 *(partial application)* と呼ばれます [1]。これの結果は残りの引数をとる新たな関数で、すべての引数を合わせて元の関数を呼び出します。

Emacs Lisp で部分適用を行う方法を示します:

`apply-partially` *func* **&rest** *args* [Function]

この関数は新たな関数をリターンする。この新しい関数は呼びだされたときに *args*、および呼び出し時に指定された追加の引数から成る引数リストで *func*を呼び出す関数である。*func*に *n* 個の引数を指定できる場合、*m* < *n* 個の引数で `apply-partially`を呼び出すと、*n* − *m* 個の新たな関数を生成する。

以下はビルトイン関数 `1+`が存在しないものとして、`apply-partially`と他のビルトイン関数 `+`を使用して `1+`を定義する例である:

```
(defalias '1+ (apply-partially '+ 1)
  "Increment argument by one.")
(1+ 10)
     ⇒ 11
```

引数として関数を受け取ったり、データ構造 (特にフック変数やプロパティーリスト) から関数を探す関数は Lisp では一般的で、それらは `funcall`や `apply`を使用してそれらの関数を呼び出します。引数として関数をとる関数は、ファンクショナル *(functional)* と呼ばれるときもあります。

ファンクショナルを呼び出すとき、引数として no-op 関数 (何も行わない関数) を指定できると便利なときがあります。以下に 2 つの異なる no-op 関数を示します:

`identity` *argument* [Function]

この関数は *argument*をリターンする。副作用はない。

---

[1] これはカリー化 *(currying)* と関連しますが異なる機能です。カーリングは複数の引数を受け取る関数を、関数チェーンとして呼び出せるような 1 つの引数を取る個々の関数に変換するような方法です。

`ignore` **&rest** *arguments* [Function]
この関数はすべての *arguments*を無視して `nil`をリターンする。

関数のいくつかはユーザーに可視なコマンドで、これらは (通常はキーシーケンスを介して) 対話的に呼び出すことができます。そのようなコマンドは、`call-interactively`関数を使用することにより、対話的に呼びだされたときと同様に呼び出すことができます。Section 21.3 [Interactive Call], page 382 を参照してください。

13.6 関数のマッピング

マップ関数 *(mapping function)* は与えられた関数 (スペシャルフォームやマクロではない) をリストや他のコレクションの各要素に適用します。Emacs Lisp にはそのような関数がいくつかあります。このセクションではリストにたいしてマッピングを行う `mapcar`、`mapc`、`mapconcat`、`mapcan`を説明します。obarray 内のシンボルにたいしてマッピングを行う関数 `mapatoms`は [Definition of mapatoms], page 128 を参照してください。ハッシュテーブル内の key/value 関係にたいしてマッピングを行う関数 `maphash`は [Definition of maphash], page 120 を参照してください。

これらのマップ関数は文字テーブル (char-table) には適用されません。なぜなら文字テーブルは非常に広い範囲の疎な配列だからです。疎な配列であるという性質に適う方法で文字テーブルにマッピングするには、関数 `map-char-table`を使用します (Section 6.6 [Char-Tables], page 110 を参照)。

`mapcar` *function sequence* [Function]
`mapcar`は関数 *function*を *sequence*の各要素にたいして順番に適用して、その結果をリストでリターンする。

引数 *sequence*には、文字テーブルを除く任意の種類のシーケンス — つまりリスト、ベクター、ブールベクター、文字列を指定できる。結果は常にリストになる。結果の長さは *sequence*の長さと同じ。たとえば:

```
(mapcar 'car '((a b) (c d) (e f)))
     ⇒ (a c e)
(mapcar '1+ [1 2 3])
     ⇒ (2 3 4)
(mapcar 'string "abc")
     ⇒ ("a" "b" "c")

;; my-hooks内の各関数を呼び出す
(mapcar 'funcall my-hooks)

(defun mapcar* (function &rest args)
  "Apply FUNCTION to successive cars of all ARGS.
Return the list of results."
  ;; リストが消費されていなければ
  (if (not (memq nil args))
      ;; CAR に関数を適用する
      (cons (apply function (mapcar 'car args))
            (apply 'mapcar* function
                   ;; 残りの要素のための再帰
                   (mapcar 'cdr args)))))
```

```
(mapcar* 'cons '(a b c) '(1 2 3 4))
     ⇒ ((a . 1) (b . 2) (c . 3))
```

`mapcan` *function sequence* [Function]

この関数は `mapcar`のように *sequence*の各要素に *function*を適用するが、結果をリストに収集するかわりに結果を変更 (`nconc`を使用。see Section 5.6.3 [Rearrangement], page 84 を参照) して結果のすべての要素を単一のリストでリターンする。`mapcar`と同様に *sequence*には文字テーブルを除く任意のタイプのシーケンスを指定できる。

```
;; 以下と:
(mapcar 'list '(a b c d))
     ⇒ ((a) (b) (c) (d))
;; 以下を比較してみよ:
(mapcan 'list '(a b c d))
     ⇒ (a b c d)
```

`mapc` *function sequence* [Function]

`mapc`は `mapcar`と似ているが、*function*は副作用のためだけに使用される — つまり *function*がリターンする値は無視されてリストに収集されない。`mapc`は常に *sequence*をリターンする。

`mapconcat` *function sequence separator* [Function]

`mapconcat`は *sequence*のそれぞれの要素に *function*を適用する。結果は文字のシーケンス (文字列、ベクター、リスト) でなければならず、単一の文字列に結合されてリターン値となる。`mapconcat`は結果シーケーンスの各ペアの間に *separator*の文字を挿入する。これも文字列、または文字のベクターかリストでなければならない。Chapter 6 [Sequences Arrays Vectors], page 95 を参照のこと。

引数 *function*は 1 つの引数を受け取り文字のシーケンス、すなわち文字列、ベクター、リストのいずれかをリターンする。引数 *sequence*は文字テーブル以外の任意の種類のシーケンス、すなわちリスト、ベクター、ブールベクター、または文字列を指定できる。

```
(mapconcat 'symbol-name
           '(The cat in the hat)
           " ")
     ⇒ "The cat in the hat"

(mapconcat (lambda (x) (format "%c" (1+ x)))
           "HAL-8000"
           "")
     ⇒ "IBM.9111"
```

13.7 無名関数

関数は通常は `defun`により定義されて、同時に名前が与えられますが、明示的にラムダ式を使う — 無名関数 *(anonymous function)* のほうが便利なときもあります。無名関数は名前つき関数が有効な場所ならどこでも有効です。無名関数は変数や関数の引数に割り当てられることがよくあります。たとえばある関数をリストの各要素に適用する `mapcar`の *function*引数に渡すかもしれません (Section 13.6 [Mapping Functions], page 219 を参照)。現実的な例は [describe-symbols example], page 520 を参照してください。

無名関数として使用するためのラムダ式を定義するとき、原則的にはリストを構築する任意の手法を使用できます。しかし通常はマクロ `lambda`、スペシャルフォーム `function`、または入力構文`#'`を使用するべきです。

`lambda` *args* [*doc*] [*interactive*] *body*... [Macro]

このマクロは引数リスト *args*、(もしあれば) ドキュメント文字列 *doc*、(もしあれば) インタラクティブ指定 *interactive*、および *body*で与えられる body フォームをもつ無名関数をリターンする。

ダイナミックバインディングの下では、このマクロは`lambda`フォームを効果的に “自己クォート (self-quoting)” する。つまり CAR が`lambda`であるようなフォームはフォーム自身を得る。

```
(lambda (x) (* x x))
     ⇒ (lambda (x) (* x x))
```

レキシカルバインディングの下で評価した際には、結果はクロージャオブジェクトになることに注意 (Section 13.10 [Closures], page 227 を参照)。

`lambda`フォームは別の 1 つの効果をもつ。このマクロは`function`(以下参照) をサブルーチンとして使用することにより、Emacs 評価機能 (Emacs evaluator) とバイトコンパイラーに、その引数が関数であることを告げる。

`function` *function-object* [Special Form]

このスペシャルフォームは評価を行わずに *function-object*をリターンする。この点では`quote`(Section 10.2 [Quoting], page 139 を参照) と似ている。しかし`quote`とは異なり、Emacs 評価機能とバイトコンパイラーに、これを関数として使用する意図を告げる役割をもつ。*function-object*が有効なラムダ式と仮定すると、これは 2 つの効果をもつ:

- そのコードがバイトコンパイルされているとき、*function-object*はバイトコード関数オブジェクトにコンパイルされる (Chapter 17 [Byte Compilation], page 285 を参照)。
- レキシカルバインドが有効なら *function-object*はクロージャーに変換される。Section 13.10 [Closures], page 227 を参照のこと。

*function-object*がシンボルかつバイトコンパイル済みコードの場合には、その関数が未定義、あるいは実行時に認識されていなければバイトコンパイラーは警告を発する。

入力構文`#'`は`function`の使用の略記です。以下のフォームは等価です:

```
(lambda (x) (* x x))
(function (lambda (x) (* x x)))
#'(lambda (x) (* x x))
```

以下の例では 3 つ目の引数に関数をとる `change-property`関数を定義して、その後の`change-property`で無名関数を渡してこれを使用しています:

```
(defun change-property (symbol prop function)
  (let ((value (get symbol prop)))
    (put symbol prop (funcall function value))))

(defun double-property (symbol prop)
  (change-property symbol prop (lambda (x) (* 2 x))))
```

`lambda`フォームをクォートしていないことに注意してください。

上記のコードをコンパイルすると無名関数もコンパイルされます。リストをクォートすることにより無名関数を構築した場合にはコンパイルはされません。

```
(defun double-property (symbol prop)
  (change-property symbol prop '(lambda (x) (* 2 x))))
```

この場合、無名関数はコンパイルされたコード内のラムダ式に保持されます。バイトコンパイラーは`change-property`が関数としての使用を意図していることを知ることができないので、たとえこの関数が関数のように見えるとしても、このリストが関数であると決め込むことができません。

13.8 ジェネリック関数

`defun`を使用して定義された関数は、その引数の型と期待する値に関して、ハードコードされた一連の仮定をもちます。たとえば数字か数字のリストを引数値として処理するようにデザインされた関数は、ベクターや文字列のような他の型の値で呼び出されると失敗したりエラーをシグナルするでしょう。これはその関数実装が、デザイン時に想定した以外の型に対応しないために発生します。

対照的に多相型関数 *(polymorphic functions)* を使用したオブジェクト指向プログラムでは、同一の名前をもつ一連の特化した関数のそれぞれが、特定の引数型セットにたいして記述されます。どの関数が実際に呼び出されるかは、実際の引数の型にもとづいて実行時に決定されます。

Emacs はポリモーフィズム (polymorphism) にたいするサポートを提供します。他の Lisp 環境、特に Common Lisp と Common Lisp オブジェクトシステム (CLOS) と同じように、このサポートはジェネリック関数 *(generic functions)* を基礎としています。Emacs のジェネリック関数は同一名の使用を含む CLOSに密接にしたがっているので、CLOSの経験があればこのセクションの残りの部分は非常に身近に感じるでしょう。

ジェネリック関数は、その名前と引数のリストを指定して、(通常は) 実装されていない抽象操作 (abstract operation) を指定します。引数のいくつかの固有クラスにたいする実際の実装はメソッド *(methods)* により提供され、これは個別に定義されるべきです。ジェネリック関数を実装するそれぞれのメソッドはジェネリック関数としてとして同じ名前をもちますが、そのジェネリック関数で定義された引数のスペシャライジング *(specializing)* により、メソッドの定義はどの種類の引数を処理可能かを示します。これらの引数スペシャライザー *(argument specializers)* は多少の差はあれ特化したものにできます。たとえば`string`型は`sequence`のようなより一般的な型より特化した型です。

C++や Simula のようなメッセージベースの OO 言語と異なり、ジェネリック関数を実装するメソッドはクラスに属さずに、それらが実装するジェネリック関数に属することに注意してください。

ジェネリック関数が呼び出されると、呼び出し側に渡された実際の引数と各メソッドの引数スペシャライザーを比較することにより、適用可能なメソッドを呼び出します。その呼び出しの実際の引数がメソッドのスペシャライザーと互換性があれば、そのメソッドが適用可能です。複数のメソッドが適用可能ならば、それらは以下で説明する特定のルールにより合成されて、その組み合わせが呼び出しを処理します。

`cl-defgeneric` *name arguments* [*documentation*] [Macro]
[*options-and-methods...*] **&rest** *body*

このマクロは指定した *name*と *arguments*でジェネリック関数を定義する。*body*が与えられたなら、それは実装のデフォルトを与える。(常に与えられるべきであるが)*documentation*が与えられたなら、それは (`:documentation` ***docstring***)の形式でそのジェネリック関数のドキュメント文字列を指定する。オプションの *options-and-methods*は以下のフォームのいずれかを指定できる:

(`declare` ***declarations***)

Section 13.14 [Declare Form], page 237 で説明するような declare フォーム。

`(:argument-precedence-order &rest args)`
: このフォームは適用可能なメソッド合成にたいするソート順に影響を与える。合成において 2 つのメソッドを比較する際、メソッドの引数は通常は左から右に試験されて、引数スペシャライザーがより特化した最初のメソッドが他のメソッドより前になる。このフォームで定義された順序はそれをオーバーライドして、左から右ではなくこのフォームの順に応じて試験される。

`(:method [qualifiers...] args &rest body)`
: このメソッドは `cl-defmethod`が行うようなメソッドを定義する。

`cl-defmethod` *name* [*qualifier*] *arguments* [**&context** (*expr* **spec**)...] [Macro]
&rest [*docstring*] *body*

このマクロは *name*と呼ばれるジェネリック関数の、特定の実装を定義する。実装コードは *body* で与えられる。もし与えられたら *docstring*はそのメソッドのドキュメント文字列である。リスト *arguments*はジェネリック関数を実装するすべてのメソッドで等しく、その関数の引数リストとマッチしなければならず、`(arg spec)`という形式の引数スペシャライザーを提供する。ここで *arg*は `cl-defgeneric`呼び出しで指定された引数名、*spec*は以下のスペシャライザーフォームのいずれかであること:

type
: このスペシャライザーは、引数が *type*のいずれかであることを要求する。*type* は以下で説明する型ヒエラルキーのいずれかの型である。

`(eql object)`
: このスペシャライザーは、引数が *object*と `eql`であることを要求する。

`(head object)`
: 引数は `car`が *object*と `eql`であるようなコンスセルでなければならない。

struct-type
: 引数は `cl-defstruct` (Section "Structures" in *Common Lisp Extensions for GNU Emacs Lisp* を参照) で定義された *struct-type*という名前のクラス、またはその子クラスのインスタンスでなければならない。

メソッド定義は新たな引数リストのキーワード`&context`を使用できる。これはメソッド実行時に環境をテストする余分なスペシャライザーを導入する。このキーワードは必須の引数リストの後、かつすべての`&rest`と`&optional`キーワードの前に記述すること。`&context`スペシャライザーは正規の引数スペシャライザー (*expr spec*) と非常によく似ているが、*expr*はカレントコンテキストで評価される式であり、*spec*は比較対象となる値となる。たとえば`&context (overwrite-mode (eql t))`は、メソッドを `overwrite-mode`がオンのとだけ適用可能にする。`&context`キーワードの後には任意個数のコンテキストスペシャライザーを続けることができる。コンテキストスペシャライザーはジェネリック関数の引数 signature の一部ではないので、これらを必要としないメソッドでは省略できる。

型スペシャライザー `(arg type)`は以下のリストのシステム型 *(system types)* のいずれかを指定できる。親の型が指定されたときは、型がより特化した子型、孫型、曾孫型、... のいずれかであるような任意の引数も互換となるだろう。

`integer` 親型: `number`。

`number`

`null` 親型: `symbol`。

`symbol`

`string` 親型: `array`。

`array` 親型: `sequence`。

`cons` 親型: `list`。

`list` 親型: `sequence`。

`marker`

`overlay`

`float` 親型: `number`。

`window-configuration`
`process`

`window`

`subr`

`compiled-function`
`buffer`

`char-table`
親型: `array`。

`bool-vector`
親型: `array`。

`vector` 親型: `array`。

`frame`

`hash-table`
`font-spec`
`font-entity`
`font-object`

オプションの *qualifier*は複数の適用可能なメソッドの合成を許容する。与えられなければ定義されるメソッドは *primary*(主) メソッドとなり、スペシャライズされた引数にたいする主要な実装の提供に責任を有する。*qualifier*として以下の値のいずれかを使用して *auxiliary*(副) メソッドも定義できる:

`:before` この auxiliary メソッドは primary メソッドの前に実行される。より正確にはすべての`:before`メソッドは、より特化したメソッドが最初になる順で、primary メソッドの前に実行される。

`:after` この auxiliary メソッドは primary メソッドの後に実行される。より正確にはすべてのこの類のメソッドは、より特化したメソッドが最後になる順で、primary メソッドの後に実行される。

`:around` この auxiliary メソッドは primary メソッドの代替えとして実行される。この類のメソッドでもっとも特化したものが他のメソッドより前に実行される。このようなメソッドは他の auxiliary メソッドや primary メソッドを呼び出すために、通常は以下で説明する `cl-call-next-method`を使用する。

`:extra` *string*
: これにより同一の specializer と qualifier にたいして、*string*で区別されるメソッドを追加できる。

`cl-defmethod`を使用して定義した関数をインタラクティブにすることはできない。つまり`interactive`フォームを追加してコマンドにすることはできない (Section 21.2 [Defining Commands], page 376 を参照)。多相型コマンド (polymorphic command) が必要なら、`cl-defgeneric`と`cl-defmethod`を通じて定義した多相型関数 (polymorphic function) を呼び出す通常のコマンドを定義することを推奨する。

ジェネリック関数が呼び出されると、毎回その関数にたいして定義された適用可能なメソッドを合成することによってその呼び出しを処理する *effective* メソッド *(effective method)* を構築します。適用可能なメソッドを探して effective メソッドを生成するプロセスは *dispatch* と呼ばれます。その呼び出しの実際の引数と互換性があるスペシャライザーをもつすべてのメソッドが、互換性のあるメソッドです。すべての引数がスペシャライザーと互換でなければならないので、それらはすべてメソッドが適用可能かどうか判定します。複数の引数に明示的に特化したメソッドを *multiple-dispatch* メソッド *(multiple-dispatch methods)* と呼びます。

適用可能なメソッドはそれらが合成される順にソートされます。最左の引数スペシャライザーがもっとも特化したものであるようなメソッドが、順序の最初になります (上述したように`cl-defmethod`の一部として`:argument-precedence-order`を指定することによりこれをオーバーライドできる)。そのメソッドの body が`cl-call-next-method`を呼び出すと、もっとも特化した次のメソッドが実行されます。適用可能な`:around`メソッドがあれば、それらのうちもっとも特化したメソッドが実行されます。そのメソッドはより特化していない任意の`:around`メソッドを実行するために、`cl-call-next-method`を呼び出すべきです。次に`:before`メソッドがその特化した順に、その後に specificity メソッドが実行されます。そして後に`:after`メソッドがその特化した順と逆順で実行されます。

`cl-call-next-method` **&rest** *args* [Function]
: primary メソッドか`:around` auxiliary メソッド内のレキシカル body 内で呼び出されると、同じジェネリック関数にたいして適用可能な次のメソッドを呼び出す。通常これは引数なしで呼び出され、これは次の適用可能なメソッドを呼び出すメソッドが、呼び出されたときと同じ引数で次のメソッドを呼び出すことを意味する。それ以外ならかわりに指定された引数が使用される。

`cl-next-method-p` [Function]
: primary メソッドか`:around` auxiliary メソッドのレキシカル body 内からこの関数を呼び出したときは、呼び出す次のメソッドが存在すれば非`nil`をリターンする。

13.9 関数セルの内容へのアクセス

シンボルの関数定義 *(function definition)* とは、そのシンボルの関数セルに格納されたオブジェクトのことです。ここではシンボルの関数セルへのアクセスやテスト、それをセットする関数を説明します。

[Definition of indirect-function], page 136 の関数`indirect-function`も参照してください。

`symbol-function` *symbol* [Function]
: これは *symbol*の関数セル内のオブジェクトをリターンする。これはリターンされたオブジェクトが本物のの関数であるかチェックしない。

関数セルが void ならリターン値は `nil`。関数セルが void のときと `nil`がセットされているときを区別するには `fboundp`(以下参照) を使用する。

```
(defun bar (n) (+ n 2))
(symbol-function 'bar)
     ⇒ (lambda (n) (+ n 2))
(fset 'baz 'bar)
     ⇒ bar
(symbol-function 'baz)
     ⇒ bar
```

シンボルに何の関数定義も与えていなければ、そのシンボルの関数セルは *void* だと言います。言い換えると、その関数セルはどんな Lisp オブジェクトも保持しません。そのシンボルを関数として呼びだそうとすると、Emacs は `void-function`エラーをシグナルします。

void は `nil`やシンボル `void`とは異なることに注意してください。シンボル `nil`と `void`は Lisp オブジェクトであり、他のオブジェクトと同じように関数セルに格納することができます (これらのシンボルは `defun`を使用して有効な関数になることができる)。void であるような関数セルは、どのようなオブジェクトも含んでいません。

`fboundp`を使用して任意のシンボルの関数定義が void かどうかテストすることができます。シンボルに関数定義を与えた後は、`fmakunbound`を使用して再び void にすることができます。

`fboundp` *symbol* [Function]

この関数はそのシンボルが関数セルにオブジェクトをもっていれば `t`、それ以外は `nil`をリターンする。これはそのオブジェクトが本物の関数であるかチェックしない。

`fmakunbound` *symbol* [Function]

この関数は *symbol*の関数セルを void にする。そのためこれ以降に関数セルへのアクセスを試みると、`void-function`エラーが発生する。これは *symbol*をリターンします (Section 12.4 [Void Variables], page 177 の `makunbound`も参照)。

```
(defun foo (x) x)
(foo 1)
     ⇒1
(fmakunbound 'foo)
     ⇒ foo
(foo 1)
error  Symbol's function definition is void: foo
```

`fset` *symbol definition* [Function]

この関数は *symbol*の関数セルに *definition*を格納する。結果は *definition*。*definition*は通常は関数か関数の名前であるべきだが、これはチェックされない。引数 *symbol*は通常のどおり評価される引数である。

この関数は主に関数を定義したり変更して構築を行う、`defun`や `advice-add`のようなものからサブルーチンとして使用される。たとえばキーボードマクロ (Section 21.16 [Keyboard Macros], page 420 を参照) のような、関数ではない関数定義をシンボルに与えるためにも使用することができる:

```
;; 名前つきのキーボードマクロを定義する。
(fset 'kill-two-lines "\^u2\^k")
     ⇒ "\^u2\^k"
```

関数にたいして別の名前を作成するために `fset`を使いたいなら、かわりに `defalias`の使用を考慮すること。[Definition of defalias], page 216 を参照。

13.10 クロージャー

Section 12.10 [Variable Scoping], page 185 で説明したように、Emacs はオプションで変数のレキシカルバインディングを有効にできます。レキシカルバインディングが有効な場合は、(たとえば `defun`などで) 作成したすべての名前つき関数、同様に `lambda`マクロや `function`スペシャルフォーム、`#'`構文を使用して作成したすべての無名関数 (Section 13.7 [Anonymous Functions], page 220 を参照) が、自動的にクロージャー *(closure)* に変換されます。

クロージャーとはその関数が定義されたどときに存在したレキシカル環境の記録をあわせもつ関数です。クロージャーが呼び出されたとき、定義内のレキシカル変数の参照には、その保持されたレキシカル環境が使用されます。他のすべての点では、クロージャーは通常の関数と同様に振る舞います。特にクロージャーは通常の関数と同じ方法で呼び出すことができます。

クロージャー使用する例は Section 12.10.3 [Lexical Binding], page 187 を参照してください。

現在のところ Emacs Lisp のクロージャーオブジェクトは、1 つ目の要素にシンボル `closure`をもつリストとして表現されます。そのリストは 2 つ目の要素としてレキシカル環境、残りの要素で引数リストと body フォームを表します:

```
;; レキシカルバインディングが有効
(lambda (x) (* x x))
     ⇒ (closure (t) (x) (* x x))
```

しかし実際にはクロージャーの内部構造は、内部的な実装の詳細と判断される残りの Lisp 界を晒け出すものだと言えます。この理由により、クロージャーオブジェクトの構造を直接調べたり変更することは推奨しません。

13.11 Emacs Lisp 関数にたいするアドバイス

他のライブラリーの関数定義を変更する必要があるとき、および *foo*`-function`o のようなフックやプロセスフィルター (process filter) や、関数を値としてもつ任意の変数やオブジェクトを変更する必要があるときには、名前つきの関数には `fset`か `defun`、フック変数には `setq`、プロセスフィルターには `set-process-filter`のように、適切なセッター関数 (setter function) を使用することができます。しかしこれらが以前の値を完全に破棄してしまうのが好ましくない場合もあります。

アドバイス *(advice)* 機能によって関数にアドバイスすることにより、既存の関数定義に機能を追加できます。これは関数全体を再定義するより明解な手法です。

Emacs のアドバイスシステムは 2 つのプリミティブセットを提供します。コアとなるセットは変数やオブジェクトのフィールドに保持された関数値にたいするものです (対応するプリミティブは `add-function`と `remove-function`)。もう 1 つのセットは名前つき関数の最上位のレイヤーとなるものです (主要なプリミティブは `advice-add`と `advice-remove`)。

簡単な例として、以下は関数の呼び出し時に毎回リターン値を変更するアドバイスを追加する方法です:

```
(defun my-double (x)
  (* x 2))
(defun my-increase (x)
  (+ x 1))
(advice-add 'my-double :filter-return #'my-increase)
```

このアドバイスの追加後に '3'で `my-double`を呼び出すとリターン値は '7'になるでしょう。このアドバイスを削除するには、以下のようにします

```
(advice-remove 'my-double #'my-increase)
```

より高度な例として、以下はプロセス *proc*のプロセスフィルターの呼び出しをトレースする例です:

```
(defun my-tracing-function (proc string)
  (message "Proc %S received %S" proc string))

(add-function :before (process-filter proc) #'my-tracing-function)
```

これによりそのプロセスの出力は元のプロセスフィルターに渡される前に、`my-tracing-function`に渡されるようになります。`my-tracing-function`は元の関数と同じ引数を受け取ります。これを行えば以下のようにしてトレースを行う前の振る舞いにリバートすることができます。

```
(remove-function (process-filter proc) #'my-tracing-function)
```

同様に `display-buffer`という名前つきの関数の実行をトレースしたいなら以下を使用できます:

```
(defun his-tracing-function (orig-fun &rest args)
  (message "display-buffer called with args %S" args)
  (let ((res (apply orig-fun args)))
    (message "display-buffer returned %S" res)
    res))

(advice-add 'display-buffer :around #'his-tracing-function)
```

ここで `his-tracing-function`は元の関数のかわりに呼び出されて、元の関数 (に加えてその関数の引数) を引数として受け取るので、必要な場合はそれを呼び出すことができます。出力を確認し終えたら、以下のようにしてトレースを行う前の振る舞いにリバートできます:

```
(advice-remove 'display-buffer #'his-tracing-function)
```

上記の例で使用されている引数`:before`と`:around`は、2 つの関数が構成される方法を指定します (これを行う多くの方法があるから)。追加された関数もアドバイス *(advice)* と呼ばれます。

13.11.1 アドバイスを操作するためのプリミティブ

`add-function` *where place function* **&optional** *props* [Macro]

このマクロは *place*(Section 12.17 [Generalized Variables], page 205 を参照) に格納された関数に、アドバイス *function*を追加する手軽な方法である。

*where*は既存の関数のどこ — たとえば元の関数の前や後 — に *function*が構成されるかを決定する。2 つの関数を構成するために利用可能な方法のリストは、Section 13.11.3 [Advice Combinators], page 231 を参照のこと。

(通常は名前が`-function`で終わる) 変数を変更するときには、*function*がグローバルに使用されるか、あるいはカレントバッファーだけに使用されるか選ぶことができる。*place*が単にシンボルなら *function*は *place*のグローバル値に追加される。*place*が `(local` *symbol*`)`というフォームなら、*symbol*はその変数の名前をリターンする式なので、*function*はカレントバッファーだけに追加される。最後にレキシカル変数を変更したければ、`(var` *variable*`)`を使用する必要があるだろう。

`add-function`で追加されたすべての関数は、自動的にプロパティー *props*の連想リストに加えることができる。現在のところ特別な意味をもつのは以下の 2 つのプロパティーのみ:

`name` これはアドバイスの名前を与える。この名前は`remove-function`が取り除く関数を識別するのに使用できます。これは通常は *function*が無名関数のときに使用されます。

`depth` これは複数のアドバイスが与えられたときにアドバイスを順番づける方法を指定します。depth のデフォルト 0 です。depth が 100 のときにはアドバイスが可能な限りの深さを保持すべきことを、-100 のときは最外のアドバイスに留めることを意味します。同じ depth で 2 つのアドバイスが指定された場合には、もっとも最近に追加されたアドバイスが最外になります。

`:before`アドバイスにたいしては、最外 (outermost) になるということは、このアドバイスが他のすべてのアドバイスの前、つまり 1 番目に実行されることを意味し、最内 (innermost) とは元の関数が実行される直前、すなわちこのアドバイスと元の関数の間に実行されるアドバイスは存在しないことを意味する。同様に`:after`アドバイスにたいしては、最内とは元の関数の直後、つまりこの元の関数とアドバイスの間に実行される他のアドバイスは存在せず、最外とは他のすべてのアドバイスが実行された直後にこのアドバイスが実行されることを意味する。`:override`の最内アドバイスは、元の関数だけをオーバーライドし、他のアドバイスはそれに適用されるが、`:override`の最外アドバイスは元の関数だけではなく、その他すべての適用済みのアドバイスをも同様にオーバーライドする。

*function*がインタラクティブでなければ合成された関数は、(もしあれば) 元の関数のインタラクティブ仕様 (interactive spec) を継承します。それ以外なら合成された関数はインタラクティブとなり *function*のインタラクティブ仕様を使用します。1 つ例外があります。*function*のインタラクティブ仕様が関数 (式や文字列ではない`lambda`式や`fbound`シンボル) なら、元の関数のインタラクティブ仕様を唯一の引数としてその関数を呼び出して、それが合成された関数のインタラクティブ指定になります。引数として受け取ったインタラクティブ仕様を解釈するためには`advice-eval-interactive-spec`を使用します。

注意: *function*のインタラクティブ指定は結合された関数に適用されるべきであり、*function*ではなく結合された関数の呼び出し規約にしたがうこと。これらは多くの場合には等しいので差異は生じないが、*place*に格納されたオリジナルの関数とは異なる引数を受け取る *function*では`:around`、`:filter-args`、`:filter-return`では重要になる。

`remove-function` *place function* [Macro]

このマクロは *place*に格納された関数から *function*を取り除く。これは`add-function`を使用して *function*が *place*に追加されたときだけ機能する。

*function*は *place*に追加された関数にたいして、ラムダ式にたいしても機能するように`equal`を使用して比較を試みる。これは追加で *place*に追加された関数の`name`プロパティーも比較する。これは`equal`を使用してラムダ式を比較するより信頼性がある。

`advice-function-member-p` *advice function-def* [Function]

*advice*がすでに *function-def*内にあれば非`nil`をリターンする。上記の`remove-function`と同様、実際の関数 *advice*のかわりにアドバイスの`name`も使用できる。

`advice-function-mapc` *f function-def* [Function]

*function-def*に追加されたすべてのアドバイスにたいして、関数 *f*を呼び出す。*f*は 2 つの引数 — アドバイス関数とそれのプロパティーで呼びだされる。

`advice-eval-interactive-spec` *spec* [Function]

そのようなインタラクティブ仕様で関数がインタラクティブに呼び出されたように *spec*を評価して、構築された引数のリストに対応するリストをリターンする。たとえば`(advice-eval-`

`interactive-spec "r\nP"`)はリージョンの境界、カレントプレフィクス引数を含む、3 つの要素からなるリストをリターンする。

たとえば *C-x m* (`compose-mail`) コマンドに‘`From:`’ヘッダーの入力を求めるさせるようにするには、以下のようにすればよい:

```
(defun my-compose-mail-advice (orig &rest args)
  "Read From: address interactively."
  (interactive
   (lambda (spec)
     (let* ((user-mail-address
             (completing-read "From: "
                              '("one.address@example.net"
                                "alternative.address@example.net")))
            (from (message-make-from user-full-name
                                     user-mail-address))
            (spec (advice-eval-interactive-spec spec)))
       ;; Put the From header into the OTHER-HEADERS argument.
       (push (cons 'From from) (nth 2 spec))
       spec)))
  (apply orig args))

(advice-add 'compose-mail :around #'my-compose-mail-advice)
```

13.11.2 名前つき関数にたいするアドバイス

アドバイスは名前つき関数やマクロにたいして使用するのが一般的な使い方です。これは単に`add-function`を使用して以下のように行うことができます:

```
(add-function :around (symbol-function 'fun) #'his-tracing-function)
```

しかしかわりに `advice-add`と `advice-remove`を使うべきです。この異なる関数セットは名前つき関数に適用されるアドバイスを操作するためのもので、`add-function`と比較して以下の追加機能があります。まずこれらはマクロとオートロードされた関数を扱う方法を知っています。次に`describe-function`にたいして追加されたアドバイスと同様に、元のドキュメント文字列を維持します。さらに関数が定義される前でも、アドバイスの追加と削除ができます。

既存の関数全体を再定義せずに既存の呼び出しを変更するために、`advice-add`が有用になります。しかしその関数の既存の呼び出し元は古い振る舞いを前提としているかもしれず、アドバイスによりその振る舞いが変更されたときに正しく機能しないかもしれないので、これはソー内スのバグにもなり得ます。アドバイスはデバッグを難しくする可能性もあります。デバッグを行う人はその関数がアドバイスにより変更されたことに気づかなかったり、失念しているかもしれません。

これらの理由により、他の方法で関数の振る舞いを変更できない場合に備えるために、アドバイスの使用は控えるべきです。フックを通じて同じことが行えるならフック (Section 23.1 [Hooks], page 461 を参照) の使用が望ましい方法です。特定のキーが行う何かを変更したいだけなら、新しいコマンドを記述して、古いコマンドのキーバインドを新しいコマンドにリマップ (Section 22.13 [Remapping Commands], page 441 を参照) するのが、おそらくより優れた方法です。

他の人が使用するリリース用のコードを記述する場合には、アドバイスを含めることを避けるよう試みてください。アドバイスしたい関数にその処理を行うフックがなければ、適切なフックの追加について Emacs 開発者に相談してください。特に Emacs 自身のソースファイルでは、Emacs 関数にアドバイスを配置するべきではありません (現在のところこの慣習にたいするいくつかの例外がある

が修正する予定)。一般的には `foo`にアドバイスとして `bar`を配置するよりも、`foo`内に新たなフックを作成して `bar`にそのフックを使用させるほうが明快です。

スペシャルフォーム (Section 10.1.7 [Special Forms], page 137 を参照) はアドバイスできませんが、マクロは関数と同じ方法でアドバイスできます。もちろんこれはすでにマクロ展開されたコードには影響しないため、マクロ展開前にアドバイスが確実にインストールされる必要があります。

プリミティブ (Section 13.1 [What Is a Function], page 209 を参照) にアドバイスするのは可能ですが、2 つの理由により通常は行うべきではありません。1 つ目の理由はいくつかのプリミティブがアドバイスのメカニズム内で使用されているため、それらにたいしてアドバイスを行うと無限再帰が発生するからです。2 つ目の理由は多くのプリミティブが C から直接呼び出されていて、そのような呼び出しはアドバイスを無視するからです。したがってプリミティブにたいしてアドバイスの使用を控えることにより、ある呼び出しはアドバイスにしたがい (Lisp コードから呼びだされたため)、他の呼び出しではアドバイスにしたがわない (C コードから呼び出されたため) という混乱した状況を解決できます。

`define-advice` *symbol* (*where lambda-list* **&optional** *name depth*) [Macro]
&rest *body*

このマクロはアドバイスを定義して *symbol*という名前の関数に追加する。*name*が `nil` か `symbol@name`という名前の関数ならアドバイスは無名関数。他の引数についての説明は `advice-add`を参照のこと。

`advice-add` *symbol where function* **&optional** *props* [Function]

名前つき関数 *symbol*にアドバイス *function*を追加する。*where*と *props*は `add-function`(Section 13.11.1 [Core Advising Primitives], page 228 を参照) のときと同じ意味をもつ。

`advice-remove` *symbol function* [Function]

名前つき関数 *symbol*からアドバイス *function*を取り除く。*function*にアドバイスの `name`を指定することもできる。

`advice-member-p` *function symbol* [Function]

名前つき関数 *symbol*内にすでにアドバイス *function*があれば非`nil`をリターンする。*function* にアドバイスの `name`を指定することもできる。

`advice-mapc` *function symbol* [Function]

名前つき関数 *symbol*にすでに追加されたすべての関数にたいして *function*を呼び出す。*function*はアドバイス関数とそのプロパティーという 2 つの引数で呼び出される。

13.11.3 アドバイスの構築方法

以下は `add-function`と `advice-add`の *where*引数に可能な値であり、そのアドバイス *function*と元の関数が構成される方法を指定します。

`:before` 古い関数の前に *function*を呼び出す。関数は両方とも同じ引数を受け取り、2 つの関数の結合のリターン値は古い関数のリターン値である。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (apply function r) (apply oldfun r))
```

`(add-function :before` *funvar function*`)`は ノ ー マ ル フック に た い す る `(add-hook '`*hookvar function*`)`のような 1 関数のフックと同等。

`:after` 古い関数の後に *function*を呼び出す。関数は両方とも同じ引数を受け取り、2 つの関数の結合のリターン値は古い関数のリターン値である。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (prog1 (apply oldfun r) (apply function r)))
```

`(add-function :after` *funvar function*`)`は ノ ー マ ル フック に た い す る `(add-hook '`*hookvar function* `'append)`のような 1 関数のフックと同等。

`:override`
これは古い関数を新しい関数に完全に置き換える。もちろん `remove-function`を呼び出した後に古い関数が復元される。

`:around` 古い関数のかわりに *function*を呼び出すが、古い関数は *function*の追加の引数になる。これはもっとも柔軟な結合である。たとえば古い関数を異なる引数で呼び出したり、複数回呼び出したり、let バインディングで呼び出したり、あるときは古い関数に処理を委譲し、またあるときは完全にオーバーライドすることが可能になる。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (apply function oldfun r))
```

`:before-while`
古い関数の前に *function*を呼び出し、*function*が `nil`をリターンしたら古い関数を呼び出さない。関数は両方とも同じ引数を受け取り、2 つの関数の結合のリターン値は古い関数のリターン値である。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (and (apply function r) (apply oldfun r)))
```

`(add-function :before-while` *funvar function*`)`は `run-hook-with-args-until-failure`を通じて *hookvar*が実行されたときの `(add-hook '`*hookvar function*`)`のような 1 関数のフックと同等。

`:before-until`
古い関数の前に *function*を呼び出し、*function*が `nil`をリターンした場合だけ古い関数を呼び出す。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (or (apply function r) (apply oldfun r)))
```

`(add-function :before-until` *funvar function*`)`は `run-hook-with-args-until-success`を通じて *hookvar*が実行されたときの `(add-hook '`*hookvar function*`)`のような 1 関数のフックと同等。

`:after-while`
古い関数が非 `nil`をリターンした場合だけ、古い関数の後に *function*を呼び出す。関数は両方とも同じ引数を受け取り、2 つの関数の結合のリターン値は *function*のリターン値である。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (and (apply oldfun r) (apply function r)))
```

`(add-function :after-while` *funvar function*`)`は `run-hook-with-args-until-failure`を通じて *hookvar*が実行されたときの `(add-hook '`*hookvar function* `'append)`のような 1 関数のフックと同等。

`:after-until`
古い関数が `nil`をリターンした場合だけ、古い関数の後に *function*を呼び出す。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (or  (apply oldfun r) (apply function r)))
```

(add-function :after-until *funvar function*)は run-hook-with-args-until-successを通じて *hookvar*が実行されたときの (add-hook '*hookvar function* 'append)のような 1 関数のフックと同等。

`:filter-args`
: 最初に *function*を呼び出し、その結果 (リスト) を新たな引数として古い関数に渡す。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (apply oldfun (funcall function r)))
```

`:filter-return`
: 最初に古い関数を呼び出し、その結果を *function*に渡す。より正確に言うと 2 つの関数の結合は以下のように振る舞う:

```
(lambda (&rest r) (funcall function (apply oldfun r)))
```

13.11.4 古い defadvice を使用するコードの改良

多くのコードは古い defadviceメカニズムを使用しており、これらの大半は advice-addによって陳腐化しました。advice-addの実装とセマンティックは非常にシンプルです。

古いアドバイスは以下のようなものです:

```
(defadvice previous-line (before next-line-at-end
                                  (&optional arg try-vscroll))
  "Insert an empty line when moving up from the top line."
  (if (and next-line-add-newlines (= arg 1)
           (save-excursion (beginning-of-line) (bobp)))
      (progn
        (beginning-of-line)
        (newline))))
```

新しいアドバイスメカニズムを使用すれば、これを通常の関数に変換できます:

```
(defun previous-line--next-line-at-end (&optional arg try-vscroll)
  "Insert an empty line when moving up from the top line."
  (if (and next-line-add-newlines (= arg 1)
           (save-excursion (beginning-of-line) (bobp)))
      (progn
        (beginning-of-line)
        (newline))))
```

これが実際の previous-lineを変更しないことは明確です。古いアドバイスには以下が必要です:

```
(ad-activate 'previous-line)
```

一方、新しいアドバイスメカニズムでは以下が必要です:

```
(advice-add 'previous-line :before #'previous-line--next-line-at-end)
```

ad-activateはグローバルな効果をもつことに注意してください。これは指定された関数にたいして、アドバイスのすべてを有効にします。特定のアドバイスだけをアクティブ、または非アクティブにしたいなら、ad-enable-adviceか ad-disable-adviceでアドバイスを有効か無効にする必要があります。新しいメカニズムではこの区別はなくなりました。

以下のような around のアドバイスがあるとします:

```
(defadvice foo (around foo-around)
  "Ignore case in `foo'."
```

```
  (let ((case-fold-search t))
    ad-do-it))
(ad-activate 'foo)
```

これは以下のように変換できます:

```
(defun foo--foo-around (orig-fun &rest args)
  "Ignore case in `foo'."
  (let ((case-fold-search t))
    (apply orig-fun args)))
(advice-add 'foo :around #'foo--foo-around)
```

アドバイスのクラスについて、新たな`:before`は古い `before`は完全に等価ではないことに注意してください。なぜなら古いアドバイス内では、(たとえば `ad-set-arg`を使って) その関数の引数を変更でき、それは元の関数が参照する引数値に影響します。しかし新しい`:before`は、`setq`を通じてアドバイス内の引数を変更して、その変更は元の関数からの参照に影響しません。この振る舞いにもとづいて `before`アドバイスを移行するときは、代わりにそれを新たなアドバイス`:around`か`:filter-args`に変更する必要があるでしょう。

同様に古い `after`アドバイスは、`ad-return-value`を変更することによりリターン値を変更できますが、新しい`:after`は変更できないので、そのような `after`を移行するときは、かわりにそれらを新しいアドバイス`:around`か`:filter-return`に変更する必要があるでしょう。

13.12 関数の陳腐化の宣言

名前つき関数を陳腐化している *(obsolete)* とマークすることができます。これはその関数が将来のある時点で削除されるかもしれないことを意味します。陳腐化しているとマークされた関数を含むコードをバイトコンパイルしたとき、Emacs は警告を発します。またその関数のヘルプドキュメントは表示されなくなります。他の点では陳腐化した関数は他の任意の関数と同様に振る舞います。

関数を陳腐化しているとマークするもっとも簡単な方法は、その関数の `defun`定義に `(declare (obsolete ...))`を配置することです。Section 13.14 [Declare Form], page 237 を参照してください。かわりに以下で説明している `make-obsolete`関数を使うこともできます。

`make-obsolete`を使用してマクロ (Chapter 14 [Macros], page 242 を参照) を陳腐化しているとマークすることもできます。これは関数のときと同じ効果をもちます。関数やマクロにたいするエイリアスも、陳腐化しているとマークできます。これはエイリアス自身をマークするのであって、名前解決される関数やマクロにたいしてではありません。

`make-obsolete` *obsolete-name current-name when* [Function]

この関数は *obsolete-name*を陳腐化しているとマークする。*obsolete-name*には関数かマクロを命名するシンボル、または関数やマクロにたいするエイリアスを指定する。

*current-name*がシンボルなら *obsolete-name*のかわりに *current-name*の使用を促す警告メッセージになる。*current-name*が *obsolete-name*のエイリアスである必要はない。似たような機能をもつ別の関数かもしれない。*current-name*には警告メッセージとなる文字列も指定できる。メッセージは小文字で始まりピリオドで終わること。`nil`も指定でき、この場合には警告メッセージに追加の詳細は提供されない。

引数 *when*は最初にその関数が陳腐化する時期を示す文字列 — たとえば日付やリリース番号を指定する。

`define-obsolete-function-alias` *obsolete-name current-name when* **&optional** *doc* [Macro]

この便利なマクロは関数 *obsolete-name*を陳腐化しているとマークして、それを関数 *current-name*のエイリアスにする。これは以下と等価:

```
(defalias obsolete-name current-name doc)
(make-obsolete obsolete-name current-name when)
```

加えて陳腐化した関数にたいする特定の呼び出し規約をマークできます。

`set-advertised-calling-convention` *function signature when* [Function]

この関数は *function*を呼び出す正しい方法として、引数リスト *signature*を指定する。これにより Emacs Lisp プログラムが他の方法で *function*を呼び出していたら、Emacs のバイトコンパイラーが警告を発する (それでもコードはバイトコンパイルされる)。*when*にはその変数が最初に陳腐化するときを示す文字列 (通常はバージョン番号) を指定する。

たとえば古いバージョンの Emacs では、`sit-for`には以下のように 3 つの引数を指定していた

```
(sit-for seconds milliseconds nodisp)
```

しかしこの方法による `sit-for`の呼び出しは陳腐化していると判断される (Section 21.10 [Waiting], page 412 を参照)。以下のように古い呼び出し規約は推奨されない:

```
(set-advertised-calling-convention
  'sit-for '(seconds &optional nodisp) "22.1")
```

13.13 インライン関数 Inli

インライン関数 *(inline function)* は関数と同様に機能しますが、1 つ例外があります。その関数の呼び出しがバイトコンパイルされると (Chapter 17 [Byte Compilation], page 285 を参照)、その関数の定義が呼び出し側に展開されます。

インライン関数を定義するには、`defun`のかわりに `defsubst`を記述するのがシンプルに方法です。定義の残りの部分は同一に見えますが、`defsubst`の使用によりバイトコンパイルにそれをインラインにするように指示します。

`defsubst` *name args* [*doc*] [*declare*] [*interactive*] *body...* [Macro]

このマクロはインライン関数を定義する。マクロの構文は `defun`とまったく同じ (Section 13.4 [Defining Functions], page 215 を参照)。

関数をインラインにすることにより、その関数の呼び出しが高速になる場合があります、が欠点もありその 1 つは柔軟性の減少です。その関数の定義を変更すると、すでにインライン化された呼び出しは、リコンパイルを行うまで古い定義を使用することになります。

もう 1 つの欠点は、大きな関数をインライン化することにより、コンパイルされたコードのファイル上およびメモリー上のサイズが増大することです。スピード面でのインライン化の有利性は小さい関数で顕著なので、一般的に大きな関数をインライン化するべきではありません。

インライン関数はデバッグ、トレース、アドバイス (Section 13.11 [Advising Functions], page 227 を参照) に際してうまく機能しません。デバッグの容易さと関数の再定義の柔軟さは Emacs の重要な機能なので、スピードがとても重要であって `defun`の使用が実際に性能の面で問題となるのか検証するためにすでにコードをチューニングしたのでなければ、たとえその関数が小さくてもインライン化するべきではありません。

インライン関数を定義した後そのインライン展開はマクロ同様、同じファイル内の後の部分で処理されます。

インライン関数が実行するのと同じコードに展開されるマクロ (Chapter 14 [Macros], page 242 を参照してください) を定義するために `defmacro`を使用できます。しかし式内でのマクロの直接の使用には制限があります— `apply`、`mapcar`などでマクロを呼び出すことはできません。通常の関数からマクロへの変換には余分な作業が必要になります。通常の関数をインライン関数に変換するのは簡単です。`defun`を `defsubst`に置き換えるだけです。インライン関数の引数はそれぞれ正確に 1 回評価されるので、マクロのときのように body で引数を何回使用するかを心配する必要はありません。

かわりにコンパイラーマクロとしてインライン展開されるコードを記述することにより関数を定義できます。以下のマクロがこれを可能にします。

`define-inline` *name args* [*doc*] [*declare*] *body...* [Macro]
: 自身をインライン化するコードを提供することにより、コンパイラーマクロとして関数 *name* を定義する。この関数は引数リスト *args*を受け取り、指定された *body*をもつ。

  *doc*が与えられたなら、それは関数のドキュメント文字列であること (Section 13.2.4 [Function Documentation], page 213 を参照)。*declare*が与えられたなら、それは関数のメタデータを指定する `declare`フォームであること (Section 13.14 [Declare Form], page 237 を参照)。

`define-inline`で定義された関数は、`defsubst`や `defmacro`で定義されたマクロにたいして複数の利点をもちます。

- `mapcar`に渡すことができる (Section 13.6 [Mapping Functions], page 219 を参照)。
- より効率的である。
- 値を格納するための *place* フォーム *(place forms)* として使用できる (Section 12.17 [Generalized Variables], page 205 を参照)。
- `cl-defsubst`より予測可能な方法で振る舞う (Section “Argument Lists” in *Common Lisp Extensions for GNU Emacs Lisp* を参照)。

`defmacro`と同様に、`define-inline`でインライン化された関数は、呼び出し側からダイナミックかレキシカルいずれかのスコーピングルールを継承します。Section 12.10 [Variable Scoping], page 185 を参照してください。

以下のマクロは `define-inline`で定義された関数の body 内で使用する必要があります。

`inline-quote` *expression* [Macro]
: `define-inline`にたいして *expression*をクォートする。これはバッククォート (Section 10.3 [Backquote], page 139 を参照) と似ているが、コードをクォートして`,`だけを受け入れ、`,@`は受け入れない。

`inline-letevals` (*bindings...*) *body...* [Macro]
: これは `let` (Section 12.3 [Local Variables], page 175 を参照) と似ている。*bindings*で指定されたローカル変数をセットアップしてから、その効力の下で *body*を評価する。*bindings* の要素はそれぞれシンボルか `(var expr)` という形式のリストであること。*expr*を評価した結果を *var*にバインドしたものが結果となる。*bindings*の終端の要素は `nil`か引数リストを保持するシンボルが可能。シンボルなら引数はそれぞれ評価されて結果となるリストにシンボルがバインドされる。

`inline-const-p` *expression* [Macro]
: *expression*の値が既知なら非 `nil`をリターンする。

`inline-const-val` *expression* [Macro]
: *expression*の値をリターンする。

`inline-error` *format* **&rest** *args* [Macro]
*format*に応じて *args*をフォーマットしてエラーをシグナルする。

以下は `define-inline`を使用した例です:

```
(define-inline myaccessor (obj)
  (inline-letevals (obj)
    (inline-quote (if (foo-p ,obj) (aref (cdr ,obj) 3) (aref ,obj 2)))))
```

これは以下と等価です

```
(defsubst myaccessor (obj)
  (if (foo-p obj) (aref (cdr obj) 3) (aref obj 2)))
```

13.14 `declare`フォーム

`declare`(宣言) は特別なマクロで、関数やマクロにメタプロパティーを追加するために使用できます。たとえば陳腐化しているとマークしたり、Emacs Lisp モード内の特別な `TAB`インデント規約を与えることができます。

`declare` *specs...* [Macro]
このマクロは引数を無視して `nil`として評価されるので、実行時の効果はない。しかし `defun`や `defsubst`(Section 13.4 [Defining Functions], page 215 を参照)、または `defmacro`マクロ (Section 14.4 [Defining Macros], page 244 を参照) の定義の *declare*引数に `declare`フォームがある場合は、*specs*で指定されたプロパティーを関数またはマクロに追加します。これは `defun`、`defsubst`、`defmacro`により特別に処理される。

*specs*内の各要素は `(`*`property args...`*`)`というフォームをもつこと。またそれあをクォートしないこと。これらは以下の効果をもつ:

`(advertised-calling-convention` *signature when*`)`
これは `set-advertised-calling-convention`(Section 13.12 [Obsolete Functions], page 234 を参照) の呼び出しと同じように振る舞う。*signature*にはその関数 (またはマクロ) にたいする正しい引数リスト、*when*には古い引数リストが最初に陳腐化する時期を示す文字列を指定する。

`(debug` *edebug-form-spec*`)`
これはマクロだけに有効である。Edebug でそのマクロ入ったときに、*edebug-form-spec*を使用する。Section 18.2.15.1 [Instrumenting Macro Calls], page 320 を参照のこと。

`(doc-string` *n*`)`
自身が関数やマクロ、変数のようなエンティティーを定義するために使用されるような関数やマクロを定義するときにこれが使用される。これは *n*番目の引数というこを示し、もしそれがあれば、それはドキュメント文字列とみなされる。

`(indent` *indent-spec*`)`
この関数 (かマクロ) にたいするインデント呼び出しは、*indent-spec*にしたがう。これは関数でも機能するが、通常はマクロで使用される。Section 14.6 [Indenting Macros], page 249 を参照のこと。

`(interactive-only` *value*`)`
その関数の `interactive-only`プロパティに *value*をセットする。[The interactive-only property], page 376 を参照のこと。

`(obsolete current-name when)`
: `make-obsolete`(Section 13.12 [Obsolete Functions], page 234 を参照) と同様に、関数 (かマクロ) が陳腐化しているとマークする。*current-name*にはシンボル (かわりにこのシンボルを使うことを促す警告メッセージになる)、文字列 (警告メッセージを指定)、または `nil`(警告メッセージには追加の詳細が含まれない) を指定すること。*when*にはその関数 (かマクロ) が最初に陳腐化する時期を示す文字列を指定すること。

`(compiler-macro expander)`
: これは関数だけに使用でき、最適化関数 (optimization function) として *expander*を使用するようコンパイラーに告げる。`(function args...)`のようなその関数への呼び出しフォームに出会うと、マクロ展開機能 (macro expander) は *args*. . . と同様のフォームで *expander*を呼び出す。*expander*はその関数呼び出しのかわりに使用するための新しい式、または変更されていないフォーム (その関数呼び出しを変更しないことを示す) のどちらかをリターンすることができる。*expander*にはシンボルかフォーム `(lambda (arg) body)`を指定できる。フォームなら *arg*は元の関数呼び出し式を保持して、その関数の形式に適う引数を使用することにより、その関数にたいする (評価されていない) 引数にアクセスができる。

`(gv-expander expander)`
: *expander*が `gv-define-expander`と同様、ジェネリック変数としてマクロ (か関数) にたいする呼び出しを処理する関数であることを宣言する。*expander*はシンボルかフォーム `(lambda (arg) body)`を指定できる。フォームなら、その関数は追加でそのマクロ (か関数) の引数にアクセスできる。

`(gv-setter setter)`
: *setter*がジェネリック変数としてマクロ (か関数) にたいする呼び出しを処理する関数であることを宣言する。*setter*はシンボルかフォームを指定できる。シンボルなら、そのシンボルは `gv-define-simple-setter`に渡される。フォームなら `(lambda (arg) body)`という形式で、その関数は追加でマクロ (か関数) の引数にアクセスでき、それは `gv-define-setter`に渡される。

13.15 コンパイラーへの定義済み関数の指示

あるファイルをバイトコンパイルするとき、コンパイラーが知らない関数について警告が生成されるときがあります (Section 17.6 [Compiler Errors], page 290 を参照してください)。実際に問題がある場合もありますが、問題となっている関数がそのコードの実行時にロードされる他のファイルで定義されている場合が通常です。たとえば以前は `simple.el`をバイトコンパイルすると以下のような警告が出ていました:

```
simple.el:8727:1:Warning: the function ‘shell-mode’ is not known to be
    defined.
```

実際のところ `shell-mode`は `(require 'shell)`を実行する関数内の `shell-mode`を呼び出す前でのみ使用されるので、`shell-mode`は実行時に正しく定義されるでしょう。そのような警告が実際には問題を示さないことを知っているときには警告を抑制したほうがよいでしょう。そうすれば実際に問題があることを示す新しい警告の識別性が良くなります。これは `declare-function`を使用して行うことができます。

必要なのは問題となっている関数を最初に使用する前に `declare-function`命令を追加するだけです:

```
(declare-function shell-mode "shell" ())
```

これは `shell-mode`が `shell.el` (‘`.el`’は省略可) の中で定義していることを告げます。コンパイラーは関数がそのファイルで実際に定義されているとみなしてチェックを行いません。

3 つ目の引数はオプションであり `shell-mode`の引数リストを指定します。この例では引数はありません (`nil`と値を指定しないのは異なる)。それ以外の場合には `(file &optional overwrite)`のようになります。引数リストを指定する必要はありませんが、指定すればコンパイラーはその呼び出しが宣言と合致するかチェックできます。

`declare-function` *function file* **&optional** *arglist fileonly* [Macro]

ファイル *file*内で *function*が定義されているとみなすようにバイトコンパイラーに告げる。オプションの 3 つ目の引数 *arglist*は `t` (引数リストが未指定という意味)、または `defun`と同スタイルな正式パラメーターリストのいずれか。*arglist*を省略した際のデフォルトは `nil`ではなく `t`。これは引数省略時の非定形な挙動であり、3 つ目の引数を指定せずに 4 つ目引数を与える場合には通常の `nil`のかわりに 3 つ目の引数のプレースホルダーに `t`を指定しなければならないことを意味する。オプションの 4 つ目の引数 *fileonly*が非 `nil`なら実際に *function*が定義されているかではなく *file*の存在だけをチェックすることを意味する。

これらの関数が `declare-function`が告げる場所で実際に宣言されているかどうかを検証するには、`check-declare-file`を使用して 1 つのソースファイル中のすべての `declare-function`呼び出しをチェックするか、`check-declare-directory`を使用して特定のディレクトリー配下のすべてのファイルをチェックする。

これらのコマンドは、`locate-library`で使用する関数の定義を含むはずのファイルを探す。ファイルが見つからなければ、これらのコマンドは `declare-function`の呼び出しを含むファイルがあるディレクトリーからの相対ファイル名に、定義ファイル名を展開する。

‘`.c`’や‘`.m`’で終わるファイル名を指定することにより、プリミティブ関数を指定することもできる。これが有用なのは特定のシステムだけで定義されるプリミティブを呼び出す場合だけである。ほとんどのプリミティブは常に定義されているので、それらについて警告を受け取ることはありえないはずである。

あるファイルがオプションとして外部のパッケージの関数を使う場合もある。`declare-function`命令内のファイル名のプレフィクスを‘`ext:`’にすると、そのファイルが見つかった場合はチェックして、見つからない場合はエラーとせずにスキップする。

‘`check-declare`’が理解しない関数定義もいくつか存在する (たとえば `defstruct`やその他いくつかのマクロ)。そのような場合は `declare-function`の *fileonly*引数に非 `nil`を渡すことができる。これはファイルの存在だけをチェックして、その関数の実際の定義はチェックしないことを意味する。これを行うなら引数リストを指定する必要はないが、*arglist*引数には `t`をセットする必要があることに注意 (なぜなら `nil`は引数リストが指定されなかったという意味ではなく空の引数リストを意味するため)。

13.16 安全に関数を呼び出せるかどうかの判断

SES のようないくつかのメジャーモードは、ユーザーファイル内に格納された関数を呼び出します (SES の詳細は See Info file `ses`, node ‘`Top`’を参照)。 ユーザーファイルは素性があやふやな場合があります — 初対面の人から受け取ったスプレッドシートかもしれず、会ったことのない誰かから受け取った e メールかもしれません。そのためユーザーファイルに格納されたソースコードの関数を呼び出すのは、それが安全だと決定されるすまでは危険です。

`unsafep` *form* **&optional** *unsafep-vars* [Function]
: *form*が安全 *(safe)* な Lisp 式なら `nil`、危険ならなぜその式が危険かもしれないのか説明するリストをリターンする。引数 *unsafep-vars*は、この時点で一時的なバインドだと判っているシンボルのリスト。これは主に内部的な再帰呼び出しで使用される。カレントバッファーは暗黙の引数になり、これはバッファーローカルなバインディングのリストを提供する。

高速かつシンプルにするために、`unsafep`は非常に軽量な分析を行うので、実際には安全な多くの Lisp 式を拒絶します。安全ではない式にたいして `unsafep`が `nil`をリターンするケースは確認されていません。しかし安全な Lisp 式は `display`プロパティーと一緒に文字列をリターンでき、これはその文字列がバッファーに挿入された後に実行される、割り当てられた Lisp 式を含むことができます。割り当てられた式はウィルスかもしれません。安全であるためにはバッファーへ挿入する前に、ユーザーコードで計算されたすべての文字列からプロパティーを削除しなければなりません。

13.17 関数に関するその他トピック

以下のテーブルは関数呼び出しと関数定義に関連したことを行ういくつかの関数です。これらは別の場所で説明されているので、ここではクロスリファレンスを提供します。

`apply` Section 13.5 [Calling Functions], page 217 を参照のこと。

`autoload` Section 16.5 [Autoload], page 274 を参照のこと。

`call-interactively`
Section 21.3 [Interactive Call], page 382 を参照のこと。

`called-interactively-p`
Section 21.4 [Distinguish Interactive], page 384 を参照のこと。

`commandp` Section 21.3 [Interactive Call], page 382 を参照のこと。

`documentation`
Section 24.2 [Accessing Documentation], page 520 を参照のこと。

`eval` Section 10.4 [Eval], page 140 を参照のこと。

`funcall` Section 13.5 [Calling Functions], page 217 を参照のこと。

`function` Section 13.7 [Anonymous Functions], page 220 を参照のこと。

`ignore` Section 13.5 [Calling Functions], page 217 を参照のこと。

`indirect-function`
Section 10.1.4 [Function Indirection], page 135 を参照のこと。

`interactive`
Section 21.2.1 [Using Interactive], page 376 を参照のこと。

`interactive-p`
Section 21.4 [Distinguish Interactive], page 384 を参照のこと。

`mapatoms` Section 9.3 [Creating Symbols], page 126 を参照のこと。

`mapcar` Section 13.6 [Mapping Functions], page 219 を参照のこと。

`map-char-table`
Section 6.6 [Char-Tables], page 110 を参照のこと。

`mapconcat`
: Section 13.6 [Mapping Functions], page 219 を参照のこと。

`undefined`
: Section 22.11 [Functions for Key Lookup], page 436 を参照のこと。

14 マクロ

マクロ *(macros)* により新たな制御構造や、他の言語機能の定義を可能にします。マクロは関数のように定義されますが、値の計算方法を指定するかわりに、値を計算する別の Lisp 式を計算する方法を指示します。わたしたちはこの式のことをマクロの展開 *(expansion)* と呼んでいます。

マクロは関数が行うように引数の値を処理するのではなく、引数にたいする未評価の式を処理することによって、これを行うことができます。したがってマクロは、これらの引数式かその一部を含む式を構築することができます。

て通常の関数が行えることをマクロを使用して行う場合、単にそれが速度面の理由ならばかわりにインライン関数の使用を考慮してください。Section 13.13 [Inline Functions], page 235 を参照してください。

14.1 単純なマクロの例

C の++演算子のように、変数の値をインクリメントするための Lisp 構造を定義したいとしましょう。`(inc x)`のように記述すれば、`(setq x (1+ x))`という効果を得たいとします。以下はこれを行うマクロ定義です:

```
(defmacro inc (var)
   (list 'setq var (list '1+ var)))
```

これを `(inc x)`のように呼び出すと、引数 *var*はシンボル `x`になります — 関数のときのように`x`の値ではありません。このマクロの body はこれを展開の構築に使用して、展開形は `(setq x (1+ x))`になります。マクロが一度この展開形をリターンすると Lisp はそれを評価するので、`x`がインクリメントされます。

`macrop` *object* [Function]
この述語はその引数がマクロかどうかテストして、もしマクロなら `t`、それ以外は `nil`をリターンする。

14.2 マクロ呼び出しの展開

マクロ呼び出しは関数の呼び出しと同じ外観をもち、マクロの名前で始まるリストで表されます。そのリストの残りの要素はマクロの引数になります。

マクロ呼び出しの評価は 1 つの重大な違いを除いて、関数の評価と同じように開始されます。重要な違いとはそのマクロの引数はマクロ呼び出し内で実際の式として現れます。これらの引数はマクロ定義に与えられる前には評価されません。対象的に関数の引数はその関数の呼び出しリストの要素を評価した結果です。

こうして得た引数を使用して、Lisp は関数呼び出しのようにマクロ定義を呼び出します。マクロの引数変数はマクロ呼び出しの引数値にバインドされるか、a `&rest`引数の場合は引数地のリストになります。そしてそのマクロの body が実行されて、関数 body が行うようにマクロ body の値をリターンします。

マクロと関数の 2 つ目の重要な違いは、マクロの body からリターンされる値が代替となる Lisp 式であることで、これはマクロの展開 *(expansion)* としても知られています。Lisp インタープリターはマクロから展開形が戻されると、すぐにその展開形の評価を行います。

展開形は通常の方法で評価されるので、もしかしたらその展開形は他のマクロの呼び出しを含むかもしれません。一般的ではありませんが、もしかすると同じマクロを呼び出すかもしれません。

Emacs はコンパイルされていない Lisp ファイルをロードするときに、マクロの展開を試みることに注意してください。これは常に利用可能ではありませんが、もし可能ならそれ以降の実行の速度を改善します。Section 16.1 [How Programs Do Loading], page 269 を参照してください。

`macroexpand`を呼び出すことにより、与えられたマクロ呼び出しにたいする展開形を確認することができます。

`macroexpand` *form* **&optional** *environment* [Function]

この関数はそれがマクロ呼び出しなら *form*を展開する。結果が他のマクロ呼び出しなら、結果がマクロ呼び出しでなくなるまで順番に展開を行う。これが`macroexpand`からリターンされる値になる。*form*がマクロ呼び出しで開始されなければ、与えられた *form*をそのままリターンする。

`macroexpand`は、(たとえいくつかのマクロ定義がそれを行っているとしても)*form*の部分式 (subexpression) を調べないことに注意。たとえ部分式自身がマクロ呼び出しでも、`macroexpand`はそれらを展開しない。

関数 `macroexpand`はインライン関数の呼び出しを展開しない。なぜならインライン関数の呼び出しは、通常の関数呼び出しと比較して理解が難しい訳ではないので、通常はそれを行う必要がないからである。

*environment*が与えられたら、それはそのとき定義されているマクロをシャドーするマクロの alist を指定する。バイトコンパイルではこの機能を使用している。

```
(defmacro inc (var)
    (list 'setq var (list '1+ var)))

(macroexpand '(inc r))
     ⇒ (setq r (1+ r))

(defmacro inc2 (var1 var2)
    (list 'progn (list 'inc var1) (list 'inc var2)))

(macroexpand '(inc2 r s))
     ⇒ (progn (inc r) (inc s))  ; ここでは incは展開されない
```

`macroexpand-all` *form* **&optional** *environment* [Function]

`macroexpand-all`は`macroexpand`と同様にマクロを展開するが、ドップレベルだけではなく *form*内のすべてのマクロを探して展開する。展開されたマクロがなければリターン値は *form* と `eq`になる。

上記`macroexpand`で使用した例を`macroexpand-all`に用いると、`macroexpand-all`が`inc`に埋め込まれた呼び出しの展開を行うことを確認できる

```
(macroexpand-all '(inc2 r s))
     ⇒ (progn (setq r (1+ r)) (setq s (1+ s)))
```

`macroexpand-1` *form* **&optional** *environment* [Function]

この関数は`macroexpand`のようにマクロを展開するが、展開の 1 ステップだけを行う。結果が別のマクロ呼び出しなら`macroexpand-1`はそれを展開しない。

14.3 マクロとバイトコンパイル

なぜわざわざマクロにたいする展開形を計算して、その後に展開形を評価する手間をかけるのか、不思議に思うかもしれません。なぜマクロ body は直接望ましい結果を生成しないのでしょうか? それはコンパイルする必要があるからです。

コンパイルされる Lisp プログラム内にマクロ呼び出しがあるとき、Lisp コンパイラーはインタープリターが行うようにマクロ定義を呼び出して展開形を受け取ります。しかし展開形を評価するかわりに、コンパイラーは展開形が直接プログラム内にあるかのようにコンパイルを行います。結果としてコンパイルされたコードはそのマクロにたいする値と副作用を生成しますが、実行速度は完全にコンパイルされたときと同じになります。もしマクロ body 自身が値と副作用を計算したら、このようには機能しません— コンパイル時に計算されることになり、それは有用ではありません。

マクロ呼び出しのコンパイルが機能するためには、マクロを呼び出すコードがコンパイルされるとき、そのマクロが Lisp 内ですでに定義されていなければなりません。コンパイラーにはこれを行うのを助ける特別な機能があります。コンパイルされるファイルが `defmacro`フォームを含むなら、そのファイルの残りの部分をコンパイルするためにそのマクロが一時的に定義されます。

ファイルをバイトコンパイルすると、ファイル内のトップレベルにあるすべての `require`呼び出しも実行されるので、それらを定義しているファイルを require することにより、コンパイルの間に必要なマクロ定義が利用できることが確実になります (Section 16.7 [Named Features], page 279 を参照)。誰かがコンパイルされたプログラムを実行するときに、マクロ定義ファイルのロードをしないようにするには、`require`呼び出しの周囲に `eval-when-compile`を記述します (Section 17.5 [Eval During Compile], page 289 を参照)。

14.4 マクロの定義

Lisp のマクロオブジェクトは、CAR が `macro`で CDR が関数であるようなリストです。マクロの展開形はマクロ呼び出しから、評価されていない引数のリストに、(`apply`を使って) 関数を適用することにより機能します。

無名関数のように無名 Lisp マクロを使用することも可能ですが、無名マクロを `mapcar`のような関数に渡すことに意味がないので、これが行われることはありません。実際のところすべての Lisp マクロは名前をもち、ほとんど常に `defmacro`マクロで定義されます。

`defmacro` *name args* [*doc*] [*declare*] *body*... [Macro]

`defmacro`はシンボル *name*(クォートはしない) を、以下のようなマクロとして定義する:

```
(macro lambda args . body)
```

(このリストの CDR はラムダ式であることに注意。) このマクロオブジェクトは *name*の関数セルに格納される。*args*の意味は関数の場合と同じで、キーワード`&rest`や`&optional`が使用されることもある (Section 13.2.3 [Argument List], page 212 を参照)。*name*と *args*はどちらもクォートされるべきではない。`defmacro`のリターン値は未定義。

*doc*が与えられたら、それはマクロのドキュメント文字列を指定する文字列であること。*declare*が与えられたら、それはマクロのメタデータを指定する `declare`フォームであること (Section 13.14 [Declare Form], page 237 を参照)。マクロを対話的に呼び出すことはできないので、インタラクティブ宣言をもつことはできないことに注意。

マクロが定数部と非定数部の混合体から構築される巨大なリスト構造を必要とする場合があります。これを簡単に行うためには ‘`’構文 (Section 10.3 [Backquote], page 139 を参照) を使用します。たとえば:

```
(defmacro t-becomes-nil (variable)
  `(if (eq ,variable t)
       (setq ,variable nil)))

(t-becomes-nil foo)
     ≡ (if (eq foo t) (setq foo nil))
```

14.5 マクロ使用に関する一般的な問題

マクロ展開が直感に反する結果となることがあり得ます。このセクションでは問題になりやすい重要な結果と、問題を避けるためにしたがうべきルールをいくつか説明します。

14.5.1 タイミング間違い

マクロを記述する際のもっとも一般的な問題として、展開形の中ではなくマクロ展開中に早まって実際に何らかの作業を行ってしまうことがあります。たとえば実際のパッケージが以下のマクロ定義をもつとします:

```
(defmacro my-set-buffer-multibyte (arg)
  (if (fboundp 'set-buffer-multibyte)
      (set-buffer-multibyte arg)))
```

この誤ったマクロ定義は解釈 (interpret) されるときは正常に機能しますがコンパイル時に失敗します。このマクロ定義はコンパイル時に `set-buffer-multibyte`を呼び出してしまいますが、それは間違っています。その後でコンパイルされたパッケージを実行しても何も行いません。プログラマーが実際に望むのは以下の定義です:

```
(defmacro my-set-buffer-multibyte (arg)
  (if (fboundp 'set-buffer-multibyte)
      `(set-buffer-multibyte ,arg)))
```

このマクロは、もし適切なら `set-buffer-multibyte`の呼び出しに展開され、それはコンパイルされたプログラム実行時に実行されるでしょう。

14.5.2 マクロ引数の多重評価

マクロを定義する場合、展開形が実行されるときに引数が何回評価されるか注意を払わなければなりません。以下の (繰り返し処理を用意にする) マクロで、この問題を示してみましょう。このマクロで for-loop 構文を記述できます。

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  (list 'let (list (list var init))
        (cons 'while
              (cons (list '<= var final)
                    (append body (list (list 'inc var)))))))

(for i from 1 to 3 do
   (setq square (* i i))
   (princ (format "\n%d %d" i square)))
↦
```

```
(let ((i 1))
  (while (<= i 3)
    (setq square (* i i))
    (princ (format "\n%d %d" i square))
    (inc i)))

     ⊣1       1
     ⊣2       4
     ⊣3       9
⇒ nil
```

マクロ内の引数 `from`、`to`、`do`は構文糖 (syntactic sugar) であり完全に無視されます。このアイデアはマクロ呼び出し中で (`from`、`to`、`do`のような) 余計な単語をこれらの位置に記述できるようにするというものです。

以下はバッククォートの使用により、より単純化された等価の定義です:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  `(let ((,var ,init))
     (while (<= ,var ,final)
       ,@body
       (inc ,var))))
```

この定義のフォームは両方 (バッククォートのあるものとないもの) とも、各繰り返しにおいて毎回 *final*が評価されるという欠点をもちます。*final*が定数のときは問題がありません。しかしこれがより複雑な、たとえば `(long-complex-calculation x)`のようなフォームならば、実行速度は顕著に低下し得ます。*final*が副作用をもつなら、複数回実行するとおそらく誤りになります。

うまく設計されたマクロ定義は、繰り返し評価することがそのマクロの意図された目的でない限り、引数を正確に 1 回評価を行う展開形を生成することで、この問題を避けるためのステップを費やします。以下は `for`マクロの正しい展開形です:

```
(let ((i 1)
      (max 3))
  (while (<= i max)
    (setq square (* i i))
    (princ (format "%d      %d" i square))
    (inc i)))
```

以下はこの展開形を生成するためのマクロ定義です:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  `(let ((,var ,init)
         (max ,final))
     (while (<= ,var max)
       ,@body
       (inc ,var))))
```

残念なことにこの訂正により以下のセクションで説明する、別の問題が発生します。

14.5.3 マクロ展開でのローカル変数

forの新しい定義には新たな問題があります。この定義はユーザーが意識していない、maxという名前のローカル変数を導入しています。これは以下の例で示すようなトラブルを招きます:

```
(let ((max 0))
  (for x from 0 to 10 do
    (let ((this (frob x)))
      (if (< max this)
          (setq max this)))))
```

forの body 内部の maxへの参照は、maxのユーサーバインディングの参照を意図したものですが、実際には forにより作られたバインディングにアクセスします。

これを修正する方法は、maxのかわりに intern されていない (uninterned) シンボルを使用することです (Section 9.3 [Creating Symbols], page 126 を参照)。intern されていないシンボルは他のシンボルと同じようにバインドして参照することができますが、forにより作成されるので、わたしたちはすでにユーザーのプログラムに存在するはずがないことを知ることができます。これは intern されていないので、プログラムの後続の部分でそれを配置する方法はありません。これは forにより配置された場所をのぞき、他の場所で配置されることがないのです。以下はこの方法で機能する for の定義です:

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  (let ((tempvar (make-symbol "max")))
    `(let ((,var ,init)
           (,tempvar ,final))
       (while (<= ,var ,tempvar)
         ,@body
         (inc ,var)))))
```

作成された intern されていないシンボルの名前は maxで、これを通常の intern されたシンボル max のかわりに、式内のその位置に記述します。

14.5.4 展開におけるマクロ引数の評価

マクロ定義自体が eval(Section 10.4 [Eval], page 140 を参照) の呼び出しなどによりマクロ引数式を評価した場合には別の問題が発生します。その引数がユーザーの変数を参照する場合、ユーザーがマクロ引数と同じ名前で変数を使用しようとした場合に問題となるでしょう。マクロの body 内では、マクロ引数のバインディングはその変数のもっともローカルなバインディングなので、そのフォーム内部の任意の参照はそれを参照するように評価されます。以下は例です:

```
(defmacro foo (a)
  (list 'setq (eval a) t))
(setq x 'b)
(foo x) ↦ (setq b t)
     ⇒ t                    ; bがセットされる
;; but
(setq a 'c)
(foo a) ↦ (setq a t)
     ⇒ t                    ; しかし cではなく aがセットされる
```

ユーザーの変数の名前が aか xかということで違いが生じています。これは aがマクロの引数変数 aと競合しているからです。

マクロ定義内での evalの呼び出しにまつわる別の問題は、それがおそらくコンパイル時にあなたが意図したことを行わないだろうということです。バイトコンパイラーは、そのプログラム自身の (あなたが evalでアクセスしたいと望む) 計算が発生しない、ローカル変数バインディングも存在しないプログラムのコンパイル時にマクロ定義を実行します。

この問題を避けるためには、マクロ展開形の計算では引数式を評価しないでください。かわりにその式をマクロ展開形の中に置き換えれば、その値は展開形の実行の一部として計算されます。これは、このチャプターの他の例が機能する方法です。

14.5.5 マクロが展開される回数は?

逐次解釈される関数で毎回マクロ呼び出しが展開されるが、コンパイルされた関数では (コンパイル時の)1 回だけしか展開されないという事実にもとづく問題が時折発生します。そのマクロ定義が副作用をもつなら、そのマクロが何回展開されたかによって、それらのマクロは異なる動作をとるでしょう。

したがってあなたが何をしているか本当に判っていないのであれば、マクロ展開形の計算での副作用は避けるべきです。

避けることのできない特殊な副作用が 1 つあります。それは Lisp オブジェクトの構築です。ほとんどすべてのマクロ展開形にはリストの構築が含まれます。リスト構築はほとんどのマクロの核心部分です。これは通常は安全です。用心しなければならないケースが 1 つだけあります。それは構築するオブジェクトがマクロ展開形の中でクォートされた定数の一部となるときです。

そのマクロが 1 回だけ — コンパイル時 — しか展開されないなら、そのオブジェクトの構築もコンパイル時の 1 回です。しかし逐次実行では、そのマクロはマクロ呼び出しが実行されるたびに展開され、これは毎回新たなオブジェクトが構築されることを意味します。

クリーンな Lisp コードのほとんどでは、この違いは問題になりません。しかしマクロ定義によるオブジェクト構築の副作用を処理する場合には、問題になるかもしれません。したがって問題を避けるために、マクロ定義によるオブジェクト構築の副作用を避けてください。以下は副作用により問題が起こる例です:

```
(defmacro empty-object ()
  (list 'quote (cons nil nil)))

(defun initialize (condition)
  (let ((object (empty-object)))
    (if condition
        (setcar object condition))
    object))
```

initializeが解釈されると、initializeが呼び出されるたびに新しいリスト (nil)が構築されます。したがって各呼び出しの間において副作用は存続しません。しかし initializeがコンパイルされると、マクロ empty-objectはコンパイル時に展開され、これは 1 つの定数 (nil)を生成し、この定数は initializeの呼び出しの各回で再利用、変更されます。

このような異常な状態を避ける 1 つの方法は、empty-objectをメモリー割り当て構造ではなく一種の奇妙な変数と考えることです。'(nil)のような定数にたいして setcarを使うことはないでしょうから、当然 (empty-object)にも使うことはないでしょう。

14.6 マクロのインデント

マクロ定義ではマクロ呼び出しを TABがどのようにインデントすべきか指定するために、`declare` フォーム (Section 14.4 [Defining Macros], page 244 を参照) を使うことができます。インデント指定は以下のように記述します:

```
(declare (indent indent-spec))
```

この `lisp-indent-function`プロパティ内の結果はマクロの名前にセットされます。

以下は利用できる *indent-spec*です:

`nil` これはプロパティーを指定しない場合と同じ — 標準的なインデントパターンを使用する。

`defun` この関数を '`def`'構文 — 2 番目の行が *body* の開始 — と同様に扱う。

整数: *number*
関数の最初の *number*個の引数は区別され、残りは式の body と判断される。その式の中の行は、最初の引数が区別されているかどうかにしたがってインデントされる。引数が body の一部なら、その行はこの式の先頭の開カッコ (open-parenthesis) よりも `lisp-body-indent`だけ多い列にインデントされる。引数が区別されていて 1 つ目か 2 つ目の引数なら、*2* 倍余分にインデントされる。引数が区別されていて 1 つ目か 2 つ目以外の引数なら、その行は標準パターンによってインデントされる。

シンボル: *symbol*
*symbol*は関数名。この関数はこの式のインデントを計算するために呼び出される関数。この関数は 2 つの引数をとる:

pos その行のインデントが開始される位置。

state その行の開始まで解析したとき、`parse-partial-sexp`(インデントとネスト深さの計算のための Lisp プリミティブ) によりリターンされる値。

これは数 (その行のインデントの列数)、またはそのような数が car であるようなリストをリターンすること。数とリストの違いは、数の場合は同じネスト深さの後続のすべての行はこの数と同じインデントとなる。リストなら、後続の行は異なるインデントを呼び出すかもしれない。これは *C-M-q*によりインデントが計算されるときに違いが生じる。値が数なら *C-M-q*はリストの終わりまでの後続の行のインデントを再計算する必要はない。

15 カスタマイゼーション設定

Emacs のユーザーは Customize インターフェースにより、Lisp コードを記述することなく変数とフェースをカスタマイズできます。Section “Easy Customization” in *The GNU Emacs Manual* を参照してください。このチャプターでは Customize インターフェースを通じて、ユーザーとやりとりするためのカスタマイズアイテム（*customization items*）を定義する方法を説明します。

カスタマイズアイテムには `defcustom`マクロで定義されるカスタマイズ可能変数 `defface`(Section 39.12.2 [Defining Faces], page 1015 で個別に説明) で定義されるカスタマイズ可能フェイス、および `defgroup` で定義されるカスタマイゼーショングループ *(customization groups)* が含まれ、これは関連するカスタマイゼーションアイテムのコンテナとして振る舞います。

15.1 一般的なキーワードアイテム

以降のセクションで説明するカスタマイゼーション宣言 (customization declaration) — `defcustom`、`defgroup`などはすべてさまざまな情報を指定するためのキーワード引数 (Section 12.2 [Constant Variables], page 174 を参照) を受け取ります。このセクションではカスタマイゼーション宣言のすべての種類に適用されるキーワードを説明します。

`:tag`以外のすべてのキーワードは、与えられたアイテムにたいして複数回使用できます。キーワードの使用はそれぞれ独立した効果をもちます。例外は`:tag`で、これはすべての与えられたアイテムは 1 つの名前だけを表示できるからです。

`:tag` *`label`*
: *label*を使用すると、カスタマイゼーションメニュー (customization menu) とカスタマイゼーションバッファー (customization buffer) のアイテムのラベルづけに、そのアイテムの名前のかわりに指定された文字列を使用します。混乱を招くのでそのアイテムの実際の名前と大きく異なる名前は使用しないでください。

`:group` *`group`*
: このカスタマイゼーションアイテムをグループ *group*に put する。カスタマイゼーションアイテムからこのキーワードが欠落していると、アイテムは最後に定義された同じグループ内に配置されるだろう。

  `defgroup`内で`:group`を使用すると、そのアイテムは新しいグループ (`:group`のサブグループ) になる。

  このキーワードを複数回使用すると、1 つのアイテムを複数のグループに配置することができる。それらのグループのいずれかを表示すると、このアイテムが表示される。煩雑になるので多用しないこと。

`:link` *`link-data`*
: このアイテムのドキュメント文字列の後に外部リンクを含める。これは他のドキュメントを参照するセンテンスを含んだボタンである。

  *link-data*に使用できる複数の選択肢がある:

  `(custom-manual` *`info-node`*`)`
  : info ノードへのリンク。*info-node*は`"(emacs)Top"`のような、ノード名を示す文字列である。このリンクはカスタマイゼーションバッファーの‘`[Manual]`’に表示され、*info-node*にたいしてビルトインの info リーダーを起動する。

`(info-link` *`info-node`*`)`
: `custom-manual`と同様だが、カスタマイゼーションバッファーにはその info ノード名が表示される。

`(url-link` *`url`*`)`
: ウェブページへのリンク。*url*は URLを指定する文字列である。カスタマイゼーションバッファーに表示されるリンクは`browse-url-browser-function`で指定された WWW ブラウザーを呼び出す。

`(emacs-commentary-link` *`library`*`)`
: ライブラリーのコメントセクション (commentary section) へのリンク。*library*はライブラリー名を指定する文字列である。Section D.8 [Library Headers], page 1172 を参照のこと。

`(emacs-library-link` *`library`*`)`
: Emacs Lisp ライブラリーファイルへのリンク。*library*はライブラリー名を指定する文字列である。

`(file-link` *`file`*`)`
: ファイルへのリンク。*file*はユーザーがこのリンクを呼び出したときに`find-file`で visit するファイルの名前を指定する文字列である。

`(function-link` *`function`*`)`
: 関数のドキュメントへのリンク。*function*はユーザーがこのリンクを呼び出したときに`describe-function`で説明を表示する関数の名前を指定する文字列である。

`(variable-link` *`variable`*`)`
: 変数のドキュメントへのリンク。*variable*はユーザーがこのリンクを呼び出したときに`describe-variable`で説明を表示する変数の名前を指定する文字列である。

`(custom-group-link` *`group`*`)`
: 他のカスタマイゼーショングループへのリンク。このリンクを呼び出すことにより *group*にたいする新たなカスタマイゼーションバッファーが作成される。

*link-data*の 1 つ目の要素の後に`:tag` *`name`*を追加することにより、カスタマイゼーションバッファーで使用するテキストを指定できます。たとえば`(info-link :tag "foo" "(emacs)Top")`は、そのバッファーで‘`foo`’と表示される Emacs manual へのリンクを作成します。

複数のリンクを追加するために、このキーワードを複数回使用することができます。

`:load` *`file`*
: そのカスタマイゼーションアイテムを表示する前にファイル *file*をロードする (Chapter 16 [Loading], page 269 を参照)。ロードは`load`により行われ、そのファイルがまだロードされていないときだけロードを行う。

`:require` *`feature`*
: 保存したカスタマイゼーションがこのアイテム値をセットするとき、`(require '`*`feature`*`)`が実行される。*feature*はシンボル。

`:require`を使用するもっとも一般的な理由は、ある変数がマイナーモードのような機能を有効にするとき、そのモードを実装するコードがロードされていなければ、変数のセットだけでは効果がないからである。

`:version` *version*
: このキーワードはそのアイテムが最初に導入された Emacs バージョン *version*か、そのアイテムのデフォルト値がそのバージョンで変更されたことを指定する。値 *version*は文字列でなければならない。

`:package-version '(`*package* `.` *version*`)`
: このキーワードはそのアイテムが最初に導入された *package*のバージョン *version*か、アイテムの意味 (またはデフォルト値) が変更されたバージョンを指定する。このキーワードは`:version`より優先される。

  *package*にはそのパッケージの公式名をシンボルとして指定すること (たとえば`MH-E`)。*version*には文字列であること。パッケージ *package*が Emacs の一部としてリリースされたなら、*package*と *version*の値は`customize-package-emacs-version-alist`の値に表示されるはずである。

Emacs の一部として配布された`:package-version`キーワードを使用するパッケージは、`customize-package-emacs-version-alist`変数も更新しなければなりません。

`customize-package-emacs-version-alist` [Variable]

これは`:package-version`キーワード内でリストされたパッケージのバージョンに関連付けられた Emacs のバージョンにたいして、マッピングを提供する alist である。この alist の要素は:

```
(package (pversion . eversion)...)
```

それぞれの *package*(シンボル) にたいして、パッケージバージョン *pversion*を含む 1 つ以上の要素と、それに関連付けられる Emacs バージョン *eversion*が存在する。これらのバージョンは文字列である。たとえば MH-E パッケージは以下により alist を更新する:

```
(add-to-list 'customize-package-emacs-version-alist
             '(MH-E ("6.0" . "22.1") ("6.1" . "22.1") ("7.0" . "22.1")
                    ("7.1" . "22.1") ("7.2" . "22.1") ("7.3" . "22.1")
                    ("7.4" . "22.1") ("8.0" . "22.1")))
```

*package*の値は一意である必要があり、`:package-version`キーワード内に現れる *package*の値とマッチする必要がある。おそらくユーザーはエラーメッセージからこの値を確認するので、MH-E や Gnus のようなパッケージの公式名を選択するのがよいだろう。

15.2 カスタマイゼーショングループの定義

Emacs Lisp パッケージはそれぞれ、1 つのメインのカスタマイゼーショングループ (main customization group) をもち、それにはすべてのオプションとフェイス、そのパッケージ内の他のグループが含まれるべきです。そのパッケージに少数のオプションとフェイスしかなければ、1 つのグループだけを使用してその中にすべてを配置します。20 以上のオプションやフェイスがあるなら、それらをサブグループ内に構造化して、そのサブグループをメインのカスタマイゼーショングループの下に配置します。そのパッケージ内の任意のオプションやフェイスを、サブグループと並行してメイングループに配置しても問題はありません。

そのパッケージのメイングループ (または唯一のグループ) は、1 つ以上の標準カスタマイゼーショングループ (standard customization group) のメンバーであるべきです (これらの完全なリストを表示するには *`M-x customize`*を使用する)。それらの内から 1 つ以上 (多すぎないこと) を選択して、`:group`を使用してあなたのグループをそれらに追加します。

新しいカスタマイゼーショングループは `defgroup`で宣言します。

`defgroup` *group members doc* [*keyword value*]... [Macro]
*members*を含むカスタマイゼーショングループとして *group*を宣言する。シンボル *group*はクォートしない。引数 *doc*はそのグループにたいするドキュメント文字列を指定する。

引数 *members*はそのグループのメンバーとなるカスタマイゼーションアイテムの初期セットを指定するリストである。しかしほとんどの場合は *members*を `nil`にして、メンバーを定義するときに`:group`キーワードを使用することによってそのグループのメンバーを指定する。

*members*を通じてグループのメンバーを指定したければ、要素はそれぞれ`(`*name widget*`)`という形式で指定すること。ここで *name*はシンボル、*widget*はそのシンボルを編集するウィジェット型 (widget type) である。変数には `custom-variable`、フェイスにはる `custom-face`、グループには `custom-group`が有用なウィジェットである。

Emacs に新しいグループを導入するときは `defgroup`内で`:version`キーワードを使用する。そうすればグループの個別のメンバーにたいしてそれを使用する必要がなくなる。

一般的なキーワード (Section 15.1 [Common Keywords], page 250 を参照) に加えて、`defgroup`内では以下のキーワードも使用できる:

`:prefix` *prefix*
: グループ内のアイテムの名前が *prefix*で始まり、カスタマイズ変数 `custom-unlispify-remove-prefixes`が非 `nil`なら、そのアイテムのタグから *prefix*が省略される。グループは任意の数のプレフィクスをもつことができる。

変数およびグループのサブグループはグループのシンボルの `custom-group`プロパティに格納される。Section 9.4.1 [Symbol Plists], page 129 を参照のこと。このプロパティの値は `car`が変数またはサブグループのシンボル、`cdr`が `custom-variable`か `custom-group`のいずれかであるようなリスト。

`custom-unlispify-remove-prefixes` [User Option]
この変数が非 `nil`ならグループの`:prefix`キーワードで指定されたプレフィクスは、ユーザーがグループをカスタマイズするときは常にタグ名から省略される。

デフォルト値は `nil`、つまりプレフィクス省略 (prefix-discarding) の機能は無効となる。これはオプションやフェイスの名前にたいするプレフィクスの省略が混乱を招くことがあるからである。

15.3 カスタマイゼーション変数の定義

カスタマイズ可能変数 *(customizable variable)* はユーザーオプション *(user option)* とも呼ばれ、これは Customize インターフェースを通じてセットできるグローバルな Lisp 変数です。`defvar`(Section 12.5 [Defining Variables], page 178 を参照) デ定義される他のグローバル変数と異なり、カスタマイズ可能変数は `defcustom`マクロを使用して定義されます。サブルーチンとして `defvar`を呼び出すことに加えテ、`defcustom`は Customize インターフェースでその変数が表示される方法や、その変数がとることができる値などを明示します。

`defcustom` *option standard doc* [*keyword value*]... [Macro]
このマクロはユーザーオプション (かカスタマイズ可能変数) として *option*を宣言する。*option*はクォートしないこと。

引数 *standard*は *option*の標準値を指定する式である。`defcustom`フォームの評価により *standard*が評価されるが、その値にそのオプションをバインドする必要はない。*option*がすでにデ

フォルト値をもつなら、それは変更されずに残る。ユーザーがすでに *option*にたいするカスタマイゼーションを保存していれば、ユーザーによりカスタマイズされた値がデフォルト値としてインストールされる。それ以外なら *standard*を評価した結果がデフォルト値としてインストールされる。

`defvar`と同様、このマクロは `option`をスペシャル変数 — 常にダイナミックにバインドされることを意味する — としてマークする。*option*がすでにレキシカルバインドをもつなら、そのレキシカルバインドはバインディング構文を抜けるまで効果をもつ。Section 12.10 [Variable Scoping], page 185 を参照のこと。

式 *standard*は別の様々な機会 — カスタマイゼーション機能が *option*の標準値を知る必要があるときは常に — にも評価される可能性がある。そのため任意回数の評価を行ても安全な式を使用するように留意されたい。

引数 *doc*はその変数にたいするドキュメント文字列を指定する。

`defcustom`が何も `:group`を指定しなければ、同じファイル内で `defgroup`によって最後に定義されたグループが使用される。この方法ではほとんどの `defcustom`は明示的な `:group`が不必要になる。

Emacs Lisp モードで *C-M-x*(`eval-defun`) で `defcustom`フォームを評価するとき、`eval-defun`の特別な機能は変数の値が void かどうかテストせずに、無条件に変数をセットするよう段取りする (同じ機能は `defvar`にも適用される。Section 12.5 [Defining Variables], page 178 を参照)。すでに定義された defcustom で `eval-defun`を使用することにより、(もしあれば)`:set`関数 (以下参照) が呼び出される。

事前ロード (pre-loaded) された Emacs Lisp ファイル (Section E.1 [Building Emacs], page 1175 を参照) に `defcustom`を配置すると、ダンプ時にインストールされた標準値は正しくない — たとえば依存している他の変数がまだ正しい値を割り当てられていない— かもしれない。この場合は Emacs 起動後に標準値を再評価するために、以下で説明する `custom-reevaluate-setting`を使用する。

Section 15.1 [Common Keywords], page 250 にリストされたキーワードに加えて、このマクロには以下のキーワードを指定できる

`:type` *type*
: このオプションのデータ型として *type*を使用する。これはどんな値が適正なのか、その値をどのように表示するかを指定する (Section 15.4 [Customization Types], page 257 を参照)。`defcustom`はそれぞれこのキーワードにたいする値を指定すること。

`:options` *value-list*
: このオプションに使用する適正な値のリストを指定する。ユーザーが使用できる値はこれらの値に限定されないが、これらは便利な値の選択肢を提示する。

  これは特定の型にたいしてのみ意味をもち現在のところ `hook`、`plist`、`alist`が含まれる。`:options`を使用する方法は個別の型の定義を参照のこと。

`:set` *setfunction*
: Customize インターフェースを使用してこのオプションの値を変更する方法として *setfunction*を指定する。関数 *setfunction*は 2 つの引数— シンボル (オプション名) と新しい値 — を受け取り、このオプションにたいして正しく値を更新するために必要なことは何であれ行うこと (これはおそらく Lisp 変数として単にオプションをセットすることを意味しない)。この関数は引数の値を破壊的に変更しないことが望ましい。*setfunction*のデフォルトは `set-default`。

このキーワードを指定すると、その変数のドキュメント文字列には手入力の Lisp コードで同じことを行う方法が記載されること。

`:get` *getfunction*
: このオプションの値を抽出する方法として *getfunction*を指定する。関数 *getfunction* は 1 つの引数 (シンボル) を受け取り、カスタマイズがそのシンボル (シンボルの Lisp 値である必要はない) にたいするカレント値としてそれを使うべきかどうかをリターンすること。デフォルトは `default-value`。

  `:get`を正しく使用するためには、Custom の機能を真に理解する必要がある。これは変数として Custom 内で扱われる値のためのものだが、実際には Lisp 変数には格納されない。実際に Lisp 変数に格納されている値に *getfunction*を指定するのは、ほとんどの場合は誤りである。

`:initialize` *function*
: *function*は `defcustom`が評価されるときに変数を初期化するために使用される関数であること。これは 2 つの引数 — オプション名 (シンボル) と値を受け取る。この方法での使用のために事前定義された関数がいくつかある:

  `custom-initialize-set`
  : 変数の初期化にその変数の`:set`関数を使用するが、値がすでに非 void なら再初期化を行わない。

  `custom-initialize-default`
  : `custom-initialize-set`と同様だが、その変数の`:set`のかわりに関数`set-default`を使用して変数をセットする。これは変数の`:set`関数がマイナーモードを有効または無効にする場合の通常の選択である。この選択により変数の定義ではマイナーモード関数を呼び出しは行わないが、変数をカスタマイズしたときはマイナーモード関数を呼び出すだろう。

  `custom-initialize-reset`
  : 変数の初期化に常に`:set`関数を使用する。変数がすでに非 void なら、(`:get`メソッドでリターンされる) カレント値を使用して`:set`関数を呼び出して変数をリセットする。これはデフォルトの`:initialize`関数である。

  `custom-initialize-changed`
  : 変数がすでにセットされている、またはカスタマイズされているなら、変数の初期化のために`:set`関数を使用して、それ以外なら単に `set-default` を使用する。

  `custom-initialize-delay`
  : この関数は `custom-initialize-set`と同様に振る舞うが、実際の初期化を Emacs の次回起動時に遅延させる。これはビルド時ではなく実行時のコンテキストで初期化を行わせるように、事前ロードされるファイル (や autoload される変数) で使用すること。これは (遅延された) 初期化が`:set`関数で処理されるという副作用ももつ。Section E.1 [Building Emacs], page 1175 を参照のこと。

`:local` *value*
: *value*が`t`なら *option*をバッファーローカルと自動的にマークする。値が`permanent`なら *option*の `permanent-local`プロパティも `t`にセットする。Section 12.11.2 [Creating Buffer-Local], page 192 を参照のこと。

`:risky` *value*
: その変数の`risky-local-variable`プロパティーを *value*にセットする (Section 12.12 [File Local Variables], page 197 を参照)。

`:safe` *function*
: その変数の `safe-local-variable`プロパティーを *function*にセットします (Section 12.12 [File Local Variables], page 197 を参照)。

`:set-after` *variables*
: 保存されたカスタマイゼーションに合わせて変数をセッティングするときは、その前に変数 *variables*確実にセット — つまりこれら他のものが処理される後までセッティングを遅延 — すること。これら他の変数が意図された値をもっていない場合に、この変数のセッティングが正しく機能しなければ`:set-after`を使用すること。

特定の機能をオンに切り替えるオプションには、`:require`キーワードを指定すると便利です。これはその機能がまだロードされていないときには、そのオプションがセットされれば Emacs がその機能をロードするようにします。Section 15.1 [Common Keywords], page 250 を参照してください。以下は例です:

```
(defcustom frobnicate-automatically nil
  "Non-nil means automatically frobnicate all buffers."
  :type 'boolean
  :require 'frobnicate-mode
  :group 'frobnicate)
```

あるカスタマイゼーションアイテムが`:options`がサポートする `hook`や `alist`のような型をもつなら、`custom-add-frequent-value`を呼び出すことによって `defcustom`宣言の外部から別途値を追加できます。たとえば `emacs-lisp-mode-hook`から呼び出されることを意図した関数 `my-lisp-mode-initialization`を定義する場合は、`emacs-lisp-mode-hook`にたいする正当な値として、その定義を編集することなくその関数をリストに追加したいと思うかもしれません。これは以下のようにして行うことができます:

```
(custom-add-frequent-value 'emacs-lisp-mode-hook
   'my-lisp-mode-initialization)
```

`custom-add-frequent-value` *symbol value* [Function]
: カスタマイズオプション *symbol*にたいして正当な値のリストに *value*を追加する。

  追加による正確な効果は *symbol*のカスタマイズ型に依存する。

`defcustom`は内部的に、標準値にたいする式の記録にシンボルプロパティ`standard-value`、カスタマイゼーションバッファーでユーザーが保存した値の記録に `saved-value`、カスタマイゼーションバッファーでユーザーがセットして未保存の値の記録に `customized-value`を使用します。Section 9.4 [Symbol Properties], page 129 を参照してください。加えてテーマによりセットされた値の記録に使用される `themed-value`も存在します (Section 15.6 [Custom Themes], page 267 を参照)。これらのプロパティは、car がその値を評価する式であるようなリストです。

`custom-reevaluate-setting` *symbol* [Function]
: この関数は `defcustom`を通じて宣言されたユーザーオプション *symbol*の標準値を再評価する。変数がカスタマイズされたなら、この関数はかわりに保存された値を再評価する。それからこの関数はその値に、(もし定義されていればそのオプションの`:set`プロパティーを使用して) ユーザーオプションをセットする。

これは値が正しく計算される前に定義されたカスタマイズ可能オプションにたいして有用である。たとえば startup の間、Emacs は事前ロードされた Emacs Lisp ファイルで定義されたユーザーオプションにたいしてこの関数を呼び出すが、これらの初期値は実行時だけ利用可能な情報に依存する。

`custom-variable-p` *arg* [Function]
この関数は *arg*がカスタマイズ可能変数なら非`nil`をリターンする。カスタマイズ可能変数とは、`standard-value`か`custom-autoload`プロパティーをもつ (通常は`defcustom`で宣言されたことを意味する) 変数、または別のカスタマイズ可能変数にたいするエイリアスのことである。

15.4 カスタマイゼーション型

`defcustom`でユーザーオプションを定義するときは、ユーザーオプションのカスタマイゼーション型 *(customization type)* を指定しなければなりません。これは (1) どの値が適正か、および (2) 編集のためにカスタマイゼーションバッファーで値を表示する方法を記述する Lisp オブジェクトです。

カスタマイゼーション型は`defcustom`内の`:type`キーワードで指定します。`:type`の引数は評価されますが、`defcustom`が実行されるときに 1 回だけ評価されるので、さまざまな値をとる場合には有用でありません。通常はクォートされた定数を使用します。たとえば:

```
(defcustom diff-command "diff"
  "The command to use to run diff."
  :type '(string)
  :group 'diff)
```

一般的にカスタマイゼーション型は最初の要素が以降のセクションで定義されるカスタマイゼーション型の 1 つであるようなリストです。このシンボルの後にいくつかの引数があり、それはそのシンボルに依存します。型シンボルと引数の間にはオプションで keyword-value ペアー (Section 15.4.4 [Type Keywords], page 263 を参照) を記述できます。

いくつかの型シンボルは引数を使用しません。これらはシンプル型 *(simple types)* と呼ばれます。シンプル型では keyword-value ペアーを使用しないなら、型シンボルの周囲のカッコ (parentheses) を省略できます。たとえばカスタマイゼーション型として単に`string`と記述すると、それは`(string)`と等価です。

すべてのカスタマイゼーション型はウィジェットとして実装されます。詳細は、Section “Introduction” in *The Emacs Widget Library* を参照してください。

15.4.1 単純型

このセクションではすべてのシンプルデータ型を説明します。これらのカスタマイゼーション型のうちのいくつかにたいして、カスタマイゼーションウィジェットは *C-M-i*か *M-TAB*によるインライン補完を提供します。

`sexp` 値はプリントと読み込みができる任意の Lisp オブジェクト。より特化した型を使用するために時間をとりたくなければ、すべてのオプションにたいするフォールバックとして`sexp`を使用することができる。

`integer` 値は整数でなければならない。

`number` 値は数 (浮動小数点数か整数) でなければならない。

`float` 値は浮動小数点数でなければならない。

`string` 値は文字列でなければならない。カスタマイゼーションバッファーはその文字列を区切り文字 '"'文字と '\'クォートなしで表示する。

`regexp` `string`文字と同様だがその文字列は有効な正規表現でなければならない。

`character`
値は文字コードでなければならない。文字コードは実際には整数だが、この型は数字を表示せずにバッファー内にその文字を挿入することにより値を表示する。

`file` 値はファイル名でなければならない。ウィジェットは補完を提供する。

`(file :must-match t)`
値は既存のファイル名でなければならない。ウィジェットは補完を提供する。

`directory`
値はディレクトリーでなければならない。ウィジェットは補完を提供する。

`hook` 値は関数のリストでなければならない。このカスタマイゼーション型はフック変数にたいして使用される。フック内で使用を推奨される関数のリストを指定するために、フック変数の `defcustom`内で`:options`キーワードを使用できる。Section 15.3 [Variable Definitions], page 253 を参照のこと。

`symbol` 値はシンボルでなければならない。これはカスタマイゼーションバッファー内でシンボル名として表示される。ウィジェットは補完を提供する。

`function` 値はラムダ式か関数名でなければならない。ウィジェットは関数名にたいする補完を提供する。

`variable` 値は変数名でなければならない。ウィジェットは補完を提供する。

`face` 値はフェイス名のシンボルでなければならない。ウィジェットは補完を提供する。

`boolean` 値は真偽値 — `nil`か `t`である。`choice`と `const`を合わせて使用することにより (次のセクションを参照)、値は `nil`か `t`でなければならないが、それら選択肢に固有の意味に適合する方法でそれぞれの値を説明するテキストを指定することもできる。

`key-sequence`
値はキーシーケンス。カスタマイゼーションバッファーは *`kbd`*関数と同じ構文を使用してキーシーケンスを表示する。Section 22.1 [Key Sequences], page 422 を参照のこと。

`coding-system`
値はコーディングシステム名でなければならず、*`M-TAB`*で補完することができる。

`color` 値は有効なカラー名でなければならない。ウィジェットはカラー名にたいする補完と、同様に`*Colors*`バッファーに表示されるカラーサンプルとカラー名のリストからカラー名を選択するボタンを提供する。

15.4.2 複合型

適切なシンプル型がなければ複合型 (composite types) を使用することができます。複合型は特定のデータにより、他の型から新しい型を構築します。指定された型やデータは、その複合型の引数 *(argument)* と呼ばれます。複合型は通常は以下のようなものです:

```
(constructor arguments...)
```

しかし以下のように引数の前に keyword-value ペアーを追加することもできます。

```
(constructor {keyword value}... arguments...)
```

以下のテーブルに、コンストラクター (constructor) と複合型を記述するためにそれらを使用する方法を示します:

`(cons` *`car-type cdr-type`*`)`
: 値はコンスセルでなければならず CAR は *car-type*、CDR は *cdr-type*に適合していなければならない。たとえば `(cons string symbol)`は、`("foo" . foo)`のような値にマッチするデータ型となる。

  カスタマイゼーションバッファーでは、CAR と CDR はそれぞれ特定のデータ型に応じて個別に表示と編集が行われる。

`(list` *`element-types...`*`)`
: 値は *element-types*で与えられる要素と数が正確に一致するリストでなければならず、リストの各要素はそれぞれ対応する *element-type*に適合しなければならない。

  たとえば `(list integer string function)`は 3 つの要素のリストを示し、1 つ目の要素は整数、2 つ目の要素は文字列、3 つ目の要素は関数である。

  カスタマイゼーションバッファーでは、各要素はそれぞれ特定のデータ型に応じて個別に表示と編集が行われる。

`(group` *`element-types...`*`)`
: これは `list`と似ているが、Custom バッファー内でのテキストのフォーマットが異なる。`list`は各要素の値をそのタグでラベルづけするが、`group`はそれを行わない。

`(vector` *`element-types...`*`)`
: これは `list`と似ているが、リストではなくベクターでなければならない。各要素は `list` の場合と同様に機能する。

`(alist :key-type` *`key-type`* `:value-type` *`value-type`*`)`
: 値はコンスセルのリストでなければならず、各セルの CAR はカスタマイゼーション型 *key-type*のキーを表し、同じセルの CDR はカスタマイゼーション型 *value-type*の値を表す。ユーザーは key/value ペアーの追加や削除ができ、各ペアのキーと値の両方を編集することができる。

  省略された場合の *key-type*と *value-type*のデフォルトは `sexp`。

  ユーザーは指定された key-type にマッチする任意のキーを追加できるが、`:options`(Section 15.3 [Variable Definitions], page 253 を参照) で指定することにより、あるキーを優先的に扱うことができる。指定されたキーは、(適切な値とともに) 常にカスタマイゼーションバッファーに表示される。また alist に key/value を含めるか、除外するか、それとも無効にするかを指定するチェックボックスも一緒に表示される。ユーザーは`:options`キーワード引数で指定された値を変更できない。

  `:options`キーワードにたいする引数は、alist 内の適切なキーにたいする仕様のリストであること。これらは通常は単純なアトムであり、それらは自身を意味します。たとえば:

  ```
  :options '("foo" "bar" "baz")
  ```

  これは名前が`"foo"`、`"bar"`、`"baz"`であるような 3 つの既知のキーがあることを指定し、それらは常に最初に表示される。

  たとえば`"bar"`キーに対応する値を整数だけにするというように、特定のキーに対して値の型を制限したいときがあるかもしれない。これはリスト内でアトムのかわりにリストを使用することにより指定することができる。前述のように 1 つ目の要素はそのキー、2 つ目の要素は値の型を指定する。たとえば:

  ```
  :options '("foo" ("bar" integer) "baz")
  ```

最後にキーが表示される方法を変更したいときもあるだろう。デフォルトでは:options キーワードで指定された特別なキーはユーザーが変更できないので、キーは単に const として表示される。しかしたとえばそれが関数バインディングをもつシンボルであることが既知なら、function-itemのようにあるキーの表示のためにより特化した型を使用したいと思うかもしれない。これはキーにたいしてシンボルを使うかわりに、カスタマイゼーション型指定を使用することにより行うことができる。

```
:options '("foo"
           ((function-item some-function) integer)
           "baz")
```

多くの alist はコンスセルのかわりに 2 要素のリストを使用する。たとえば、

```
(defcustom cons-alist
  '(("foo" . 1) ("bar" . 2) ("baz" . 3))
  "Each element is a cons-cell (KEY . VALUE).")
```

のかわりに以下を使用する

```
(defcustom list-alist
  '(("foo" 1) ("bar" 2) ("baz" 3))
  "Each element is a list of the form (KEY VALUE).")
```

リストはコンスセルの最上位に実装されているため、上記の list-alistをコンスセルの alist(値の型が実際の値を含む 1 要素のリスト) として扱うことができる。

```
(defcustom list-alist '(("foo" 1) ("bar" 2) ("baz" 3))
  "Each element is a list of the form (KEY VALUE)."
  :type '(alist :value-type (group integer)))
```

listのかわりに groupを使用するのは、それが目的に適したフォーマットだという理由だけである。

同様に以下のようなトリックの類を用いることにより、より多くの値が各キー連づけられた alist を得ることができる:

```
(defcustom person-data '(("brian"  50 t)
                         ("dorith" 55 nil)
                         ("ken"    52 t))
  "Alist of basic info about people.
Each element has the form (NAME AGE MALE-FLAG)."
  :type '(alist :value-type (group integer boolean)))
```

(plist :key-type *key-type* :value-type *value-type*)

このカスタマイゼーション型は alist(上記参照) と似ているが、(1) 情報がプロパティーリスト (Section 5.9 [Property Lists], page 92 を参照) に格納されていて、(2)*key-type* が省略された場合のデフォルトは sexpではなく symbolになる。

(choice *alternative-types*...)

値は *alternative-types*のうちのいずれかに適合しなければならない。たとえば (choice integer string)では整数か文字列が許容される。

カスタマイゼーションバッファーでは、ユーザーはメニューを使用して候補を選択して、それらの候補にたいして通常の方法で値を編集できる。

通常はこの選択からメニューの文字列が自動的に決定される。しかし候補の中に:tag キーワードを含めることにより、メニューにたいして異なる文字列を指定できる。たと

えば空白の数を意味する整数と、その通りに使用したいテキストにたいする文字列なら、以下のような方法でカスタマイゼーション型を記述したいと思うかもしれない

```
(choice (integer :tag "Number of spaces")
        (string :tag "Literal text"))
```

この場合のメニューは‘Number of spaces’と‘Literal text’を提示する。

const以外のnilが有効な値ではない選択肢には、:valueキーワードを使用して有効なデフォルト値を指定すること。Section 15.4.4 [Type Keywords], page 263 を参照のこと。

複数の候補によりいくつかの値が提供されるなら、カスタマイズは適合する値をもつ最初の候補を選択する。これは常にもっとも特有な型が最初で、もっとも一般的な型が最後にリストされるべきことを意味する。以下は適切な使い方の例である

```
(choice (const :tag "Off" nil)
        symbol (sexp :tag "Other"))
```

この使い方では特別な値nilはその他のシンボルとは別に扱われ、シンボルは他のLisp式とは別に扱われる。

(radio *element-types...*)
: これは choiceと似ているが、選択はメニューではなくラジオボタンで表示される。これは該当する選択にたいしてドキュメントを表示できる利点があるので、関数定数(function-itemカスタマイゼーション型)の選択に適している場合がある。

(const *value*)
: 値は *value*でなければならず他は許容されない。

constは主に choiceの中で使用される。たとえば(choice integer (const nil))では整数かnilが選択できる。

choiceの中では:tagとともに constが使用される場合がある。たとえば、

```
(choice (const :tag "Yes" t)
        (const :tag "No" nil)
        (const :tag "Ask" foo))
```

これはtがyes、nilがno、fooが“ask”を意味することを示す。

(other *value*)
: この選択肢は任意のLisp値にマッチできるが、ユーザーがこの選択肢を選択したら値 *value*が選択される。

otherは主に choiceの最後の要素に使用される。たとえば、

```
(choice (const :tag "Yes" t)
        (const :tag "No" nil)
        (other :tag "Ask" foo))
```

これはtがyes、nilがno、それ以外は“ask”を意味することを示す。ユーザーが選択肢メニューから‘Ask’を選択したら、値 fooが指定される。しかしその他の値(t、nil、fooを除く)なら fooと同様に‘Ask’が表示される。

(function-item *function*)
: constと同様だが値が関数のときに使用される。これはドキュメント文字列も関数名と同じように表示する。ドキュメント文字列は:docで指定した文字列か *function*自身のドキュメント文字列。

`(variable-item` *variable*`)`
: `const`と同様だが値が変数名のときに使用される。これはドキュメント文字列も変数名と同じように表示する。ドキュメント文字列は`:doc`で指定した文字列か *variable*自身のドキュメント文字列。

`(set` *types*`...)`
: 値はリストでなければならず指定された *types*のいずれかにマッチしなければならない。

これはカスタマイゼーションバッファーではチェックリストとして表示されるので、*types* はそれぞれ対応する要素を 1 つ、あるいは要素をもたない。同じ 1 つの *types*にマッチするような、異なる 2 つの要素を指定することはできない。たとえば `(set integer symbol)`はリスト内で 1 つの整数、および/または 1 つのシンボルが許容されて、複数の整数や複数のシンボルは許容されない。結果として `set`内で `integer`のような特化していない型を使用するのは稀である。

以下のように `const`型は `set`内の *types*でよく使用される:

```
(set (const :bold) (const :italic))
```

alist 内で利用できる要素を示すために使用されることもある:

```
(set (cons :tag "Height" (const height) integer)
     (cons :tag "Width" (const width) integer))
```

これによりユーザーにオプションで height と width の値を指定させることができる。

`(repeat` *element-type*`)`
: 値はリストでなければならず、リストの各要素は型 *element-type*に適合しなければならない。カスタマイゼーションバッファーでは要素のリストとして表示され、‘`[INS]`’と‘`[DEL]`’ボタンで要素の追加や削除が行われる。

`(restricted-sexp :match-alternatives` *criteria*`)`
: これはもっとも汎用的な複合型の構築方法である。値は *criteria*を満足する任意の Lisp オブジェクト。*criteria*はリストで、リストの各要素は以下のうちのいずれかを満たす必要がある:

- 述語 — つまり引数は 1 つで、引数に応じて `nil`か非 `nil`のどちらかをリターンする関数。リスト内での述語の使用により、その述語が非 `nil`をリターンするようなオブジェクトが許されることを意味する。
- クォートされた定数 — つまり`'`*object*。リスト内でこの要素は *object*自身が許容される値であることを示す。

たとえば、

```
(restricted-sexp :match-alternatives
                 (integerp 't 'nil))
```

これは整数、`t`、`nil`を正当な値として受け入れる。

カスタマイゼーションバッファーは適切な値をそれらの入力構文 de 表示して、ユーザーはこれらをテキストとして編集できる。

以下は複合型でキーワード/値ペアーとして使用できるキーワードのテーブルです:

`:tag` *tag*
: *tag*はユーザーとのコミュニケーションのために、その候補の名前として使用される。`choice`内に出現する型にたいして有用。

`:match-alternatives` *criteria*
: *criteria*は可能な値とのマッチに使用される。`restricted-sexp`内でのみ有用。

`:args` *argument-list*
: 型構築の引数として *argument-list* の要素を使用する。たとえば `(const :args (foo))` は `(const foo)` と等価である。明示的に `:args` と記述する必要があるのは稀である。なぜなら最後のキーワード/値ペアーの後に続くものは何であれ、引数として認識されるからである。

15.4.3 リストへのスプライス

`:inline` 機能により可変個の要素を、カスタマイゼーション型の `list` や `vector` の途中にスプライス (splice: 継ぎ足す) することができます。`list` や `vector` 記述を含む型にたいして `:inline t` を追加することによってこれを使用します。

`list` や `vector` 型の仕様は、通常は単一の要素型を表します。しかしエントリーが `:inline t` を含むなら、マッチする値は含まれるシーケンスに直接マージされます。たとえばエントリーが 3 要素のリストにマッチするなら、全体が 3 要素のシーケンスになります。これはバッククォート構文 (Section 10.3 [Backquote], page 139 を参照) の '`,@`' に類似しています。

たとえば最初の要素が `baz` で、残りの引数は 0 個以上の `foo` か `bar` でなければならないようなリストを指定するには、以下のカスタマイゼーション型を使用します:

```
(list (const baz) (set :inline t (const foo) (const bar)))
```

これは `(baz)`、`(baz foo)`、`(baz bar)`、`(baz foo bar)` のような値にマッチします。

要素の型が `choice` なら、`choice` 自身の中で `:inline` を使用せずに、`choice` の選択肢 (の一部) の中で使用します。たとえば最初がファイル名で始まり、その後にシンボル `t` か 2 つの文字列を続けなければならないようなリストにマッチさせるには、以下のカスタマイゼーション型を使用します:

```
(list file
      (choice (const t)
              (list :inline t string string)))
```

選択においてユーザーが選択肢の 1 つ目を選んだ場合はリスト全体が 2 つの要素をもち、2 つ目の要素は `t` になります。ユーザーが 2 つ目の候補を選んだ場合にはリスト全体が 3 つの要素をもち、2 つ目と 3 つ目の要素は文字列でなければなりません。

ウィジェットは `:match-inline` 要素でインライン値がウィジェットにマッチするかどうかを告げる述語を指定できます。

15.4.4 型キーワード

カスタマイゼーション型内の型名シンボルの後にキーワード/引数ペアーを指定できます。以下は使用できるキーワードとそれらの意味です:

`:value` *default*
: デフォルト値を提供する。

  その候補にたいして `nil` が有効な値でなければ、`:value` に有効なデフォルトを指定することが必須となる。

  `choice` の内部の選択肢として出現する型にたいしてこれを使用するなら、ユーザーがカスタマイゼーションバッファー内のメニューでその選択肢を選択したときに使用するデフォルト値を最初に指定する。

  もちろんオプションの実際の値がこの選択肢に適合するなら、*default* ではなく実際の値が表示される。

`:format` *format-string*
: この文字列はその型に対応する値を記述するために、バッファーに挿入される。*format-string*内では以下の '`%`'エスケープが利用できる:

  '`%[`*button*`%]`'
  : ボタンとしてマークされたテキスト *button*を表示する。`:action`属性はユーザーがそれを呼び出したときに、そのボタンが何を行うか指定する。この属性の値は 2 つの引数— ボタンが表示されるウィジェットとイベント— を受け取る関数である。

    異なるアクションを行う 2 つの異なるボタンを指定する方法はない。

  '`%{`*sample*`%}`'
  : `:sample-face`により指定されたスペシャルフェイス内の *sample*を表示する。

  '`%v`' : そのアイテムの値を代替えする。その値がどのように表示されるかはアイテムの種類と、(カスタマイゼーション型にたいしては) カスタマイゼーション型にに依存する。

  '`%d`' : そのアイテムのドキュメント文字列を代替えする。

  '`%h`' : '`%d`'と同様だが、ドキュメント文字列が複数行なら、ドキュメント文字列全体か最初の行だけかを制御するボタンを追加する。

  '`%t`' : その位置でタグに置き換える。`:tag`キーワードでタグを指定する。

  '`%%`' : リテラル '`%`'を表示する。

`:action` *action*
: ユーザーがボタンをクリックしたら *action*を実行する。

`:button-face` *face*
: '`%[...%]`'で表示されたボタンテキストにたいして、フェイス *face*(フェイス名、またはフェイス名のリスト) を使用する。

`:button-prefix` *prefix*
`:button-suffix` *suffix*
: これらはボタンの前か後に表示されるテキストを指定する。以下が指定できる:

  `nil` : テキストは挿入されない。

  文字列 : その文字列がリテラルに挿入される。

  シンボル : そのシンボルの値が使用される。

`:tag` *tag* : この型に対応する値 (または値の一部) にたいするタグとして *tag*(文字列) を使用する。

`:doc` *doc* : この型に対応する値 (か値の一部) にたいするドキュメント文字列として *doc*を使用する。これが機能するためには`:format`にたいする値を指定して、その値にたいして '`%d`' か '`%h`'を使用しなければならない。

ある型にたいしてドキュメント文字列を指定するのは `choice`内の選択肢の型や、他の複合型の一部について情報を提供するのが通常の理由。

`:help-echo` *motion-doc*
: `widget-forward`や`widget-backward`でこのアイテムに移動したときに、エコーエリアに文字列 *motion-doc*を表示する。さらにマウスの`help-echo`文字列として *motion-doc*が使用され、これには実際には」ヘルプ文字列を生成するために評価される関数か

フォームを指定できる。もし関数ならそれは 1 つの引数 (そのウィジェット) で呼び出される。

`:match` *function*
: 値がその型にマッチするか判断する方法を指定する。対応する値 *function*は 2 つの引数 (ウィジェットと値) を受け取る関数であり、値が適切なら非 `nil`をリターンすること。

`:match-inline` *function*
: インライン値がその型にマッチするか判断する方法を指定する。対応する値 *function*は 2 つの引数 (ウィジェットとインライン値) を受け取る関数であり、値が適切なら非 `nil`をリターンすること。インライン値に関する詳細な情報は Section 15.4.3 [Splicing into Lists], page 263 を参照のこと。

`:validate` *function*
: 入力にたいして検証を行う関数を指定する。*function*は引数としてウィジェットを受け取り、そのウィジェットのカレント値がウィジェットにたいして有効なら `nil`をリターンすること。それ以外なら無効なデータを含むウィジェットをリターンして、そのウィジェットの`:error`プロパティに、そのエラーを記述する文字列をセットすること。

15.4.5 新たな型の定義

前のセクションでは、`defcustom`にたいして型の詳細な仕様を作成する方法を説明しました。そのような型仕様に名前を与えたい場合があるかもしれません。理解しやすいケースとしては、多くのユーザーオプションに同じ型を使用する場合などです。各オプションにたいして仕様を繰り返すよりその型に名前を与えて、`defcustom`それぞれにその名前を使用することができます。他にもユーザーオプションの値が再帰的なデータ構造のケースがあります。あるデータ型がそれ自身を参照できるようにするためには、それが名前をもつ必要があります。

カスタマイゼーション型はウィジェットとして実装されているめ、新しいカスタマイゼーション型を定義するには、新たにウィジェット型を定義します。ここではウィジェットインターフェイスの詳細は説明しません。Section “Introduction” in *The Emacs Widget Library* を参照してください。かわりにシンプルな例を用いて、カスタマイゼーション型を新たに定義するために必要な最小限の機能について説明します。

```
(define-widget 'binary-tree-of-string 'lazy
  "A binary tree made of cons-cells and strings."
  :offset 4
  :tag "Node"
  :type '(choice (string :tag "Leaf" :value "")
                 (cons :tag "Interior"
                       :value ("" . "")
                       binary-tree-of-string
                       binary-tree-of-string)))

(defcustom foo-bar ""
  "Sample variable holding a binary tree of strings."
  :type 'binary-tree-of-string)
```

新しいウィジェットを定義するための関数は`define-widget`と呼ばれます。1 つ目の引数は新たなウィジェット型にしたいシンボルです。2 つ目の引数は既存のウィジェットを表すシンボルで、新しいウィジェットではこの既存のウィジェットと異なる部分を定義することになります。新たなカスタマイゼーション型を定義する目的にたいしては`lazy`ウィジェットが最適です。なぜならこれは`defcustom`

にたいするキーワード引数と同じ構文と名前でキーワード引数:typeを受け取るからです。3 つ目の引数は新しいウィジェットにたいするドキュメント文字列です。この文字列は *M-x widget-browse RET binary-tree-of-string RET*コマンドで参照することができます。

これらの必須の引数の後にキーワード引数が続きます。もっとも重要なのは:typeで、これはこのウィジェットにマッチさせたいデータ型を表します。上記の例では binary-tree-of-stringは文字列、または car と cdr が binary-tree-of-stringであるようなコンスセルです。この定義中でのウィジェット型への参照に注意してください。:tag属性はユーザーインターフェイスでウィジェット名となる文字列、:offset引数はカスタマイゼーションバッファーでのツリー構造の外観で，子ノードと関連する親ノードの間に 4 つのスペースを確保します。

defcustomは通常のカスタマイゼーション型に使用される方法で新しいウィジェットを表示します。

lazyという名前の由来は、他のウィジェットではそれらがバッファーでインスタンス化されるとき、他の合成されたウィジェットが下位のウィジェットを内部形式に変換するからです。この変換は再帰的なので、下位のウィジェットはそれら自身の下位ウィジェットへと変換されます。データ構造自体が再帰的なら、その変換は無限再帰 (infinite recursion) となります。lazyウィジェットは、:type引数を必要なときだけ変換することによってこの再帰を防ぎます。

15.5 カスタマイゼーションの適用

以下の関数には変数とフェイスにたいして、そのユーザーのカスタマイゼーション設定をインストールする役目をもちます。それらの関数はユーザーが Customize インターフェイスで 'Save for future sessions'を呼び出したとき、次回の Emacs 起動時に評価されるように custom-set-variables フォーム、および/または custom-set-facesフォームがカスタムファイルに書き込まれることによって効果をもちます。

custom-set-variables **&rest** *args* [Function]

この関数は *args*により指定された変数のカスタマイゼーションをインストールする。*args*内の引数はそれぞれ、以下のようなフォームであること

```
(var expression [now [request [comment]]])
```

*var*は変数名 (シンボル)、*expression*はカスタマイズされた値に評価される式である。

この custom-set-variables呼び出しより前に *var*にたいして defcustomフォームが評価されたら即座に *expression*が評価されて、その変数の値にその結果がセットされる。それ以外ならその変数の saved-valueプロパティに *expression*が格納されて、これに関係する defcustom が呼び出されたとき (通常はその変数を定義するライブラリーが Emacs にロードされたとき) に評価される。

now、*request*、*comment*エントリーは内部的な使用に限られており、省略されるかもしれない。*now*がもし非 nilなら、たとえその変数の defcustomフォームが評価されていなくても、その変数の値がそのときセットされる。*request*は即座にロードされる機能のリストである (Section 16.7 [Named Features], page 279 を参照)。*comment*はそのカスタマイゼーションを説明する文字列。

custom-set-faces **&rest** *args* [Function]

この関数は *args*により指定されたフェイスのカスタマイゼーションをインストールする。*args* 内の引数はそれぞれ以下のようなフォームであること

```
(face spec [now [comment]])
```

*face*はフェイス名 (シンボル)、*spec*はそのフェイスにたいするカスタマイズされたフェイス仕様 (Section 39.12.2 [Defining Faces], page 1015 を参照)。

now、*request*、*comment*エントリーは内部的な使用に限られており、省略されるかもしれない。*now*がもし非`nil`なら、たとえ`defface`フォームが評価されていなくても、そのフェイス仕様がそのときセットされる。*comment*はそのカスタマイズを説明する文字列。

15.6 Custom テーマ

Custom テーマ *(Custom themes)* とはユニットとして有効や無効にできるセッティングのコレクションです。Section “Custom Themes” in *The GNU Emacs Manual* を参照してください。Custom テーマはそれぞれ Emacs Lisp ソースファイルにより定義され、それらはこのセクションで説明する慣習にしたがう必要があります (Custom テーマを手作業で記述するかわりに、Customize 風のインターフェイスを使用して作成することもできる。Section “Creating Custom Themes” in *The GNU Emacs Manual* を参照)。

Custom テーマファイルは *foo*`-theme.el`のように命名すること。ここで *foo*はテーマの名前。このファイルでの最初の Lisp フォームは`deftheme`の呼び出しで、最後のフォームは`provide-theme`にすること。

`deftheme` *theme* **&optional** *doc* [Macro]

このマクロは Custom テーマの名前として *theme*(シンボル) を宣言する。オプション引数 *doc* は、そのテーマを説明する文字列であること。この文字列はユーザーが`describe-theme`コマンドを呼び出したり、‘`*Custom Themes*`’バッファーで`?`をタイプしたときに表示される。

2 つの特別なテーマ名は禁止されている (使用するとエラーになる)。`user`はそのユーザーの直接的なカスタマイズ設定を格納するためのダミーのテーマである。そし`changed`は Customize システムの外部で行われた変更を格納するためのダミーのテーマである。

`provide-theme` *theme* [Macro]

このマクロは完全に仕様が定められたテーマ名 *theme*を宣言する。

`deftheme`と`provide-theme`の違いは、そのテーマセッティングを規定する Lisp フォームです (通常は`custom-theme-set-variables`の呼び出し、および/または`custom-theme-set-faces`の呼び出し)。

`custom-theme-set-variables` *theme* **&rest** *args* [Function]

この関数は Custom テーマ *theme*の変数のセッティングを規定する。*theme*はシンボル。*args* 内の各引数はフォームのリスト。

```
(var expression [now [request [comment]]])
```

ここでリストエントリーは`custom-set-variables`のときと同じ意味をもつ。Section 15.5 [Applying Customizations], page 266 を参照のこと。

`custom-theme-set-faces` *theme* **&rest** *args* [Function]

この関数は Custom テーマ *theme*のフェイスのセッティングを規定する。*theme*はシンボル。*args*内の各引数はフォームのリスト。

```
(face spec [now [comment]])
```

ここでリストエントリーは`custom-set-faces`のときと同じ意味をもつ。Section 15.5 [Applying Customizations], page 266 を参照のこと。

原則的にテーマファイルは他の Lisp フォームを含むこともでき、それらはそのテーマがロードされるときに評価されるでしょうが、これは悪いフォームです。悪意のあるコードを含むテーマのロードを防ぐために最初に非ビルトインテーマをロードする前に、Emacs はソースファイルを表示して

ユーザーに確認を求めます。このようにテーマは通常はバイトコンパイルされずに Emacs がテーマをロードする際には常にソースファイルが優先されます。

以下の関数は、テーマをプログラム的に有効または無効にするのに有用です:

`custom-theme-p` *theme* [Function]
この関数は *theme*(シンボル) が Custom テーマの名前 (たとえばそのテーマが有効かどうかにかかわらず、Custom テーマが Emacs にロードされている) なら非 `nil`をリターンする。それ以外は `nil`をリターンする。

`custom-known-themes` [Variable]
この変数の値は Emacs にロードされたテーマのリストである。テーマはそれぞれ Lisp シンボル (テーマ名) により表される。この変数のデフォルト値は 2 つのダミーテーマ `(user changed)` を含む。`changed`テーマには Custom テーマが適用される前に行われたセッティング (たとえば Custom の外部での変数のセット) が格納されている。`user`テーマにはそのユーザーがカスタマイズして保存したセッティングが格納されている。`deftheme`マクロで宣言されたすべての追加テーマは、このリストの先頭に追加される。

`load-theme` *theme* **&optional** *no-confirm no-enable* [Command]
この関数は *theme*という名前の Custom テーマを、変数 `custom-theme-load-path`で指定されたディレクトリーから探して、ソースファイルからロードする。Section “Custom Themes” in *The GNU Emacs Manual* を参照のこと。またそのテーマの変数とフェイスのセッティングが効果を及ぼすようにテーマを *enables* にする (オプション引数 *no-enable*が `nil`の場合)。さらにオプション引数 *no-confirm*が `nil`なら、そのテーマをロードする前にユーザーに確認を求める。

`enable-theme` *theme* [Command]
この関数は *theme*という名前の Custom テーマを有効にする。そのようなテーマがロードされていなければ、エラーをシグナルする。

`disable-theme` *theme* [Command]
この関数は *theme*という名前の Custom テーマを無効にする。テーマはロードされたまま残るので、続けて `enable-theme`を呼び出せばテーマは再び有効になる。

16 ロード

Lisp コードのファイルをロードすることは、その内容を Lisp オブジェクト形式で Lisp 環境に取り込むことを意味します。Emacs はファイルを探してオープンして、テキストを読み込んで各フォームを評価してから、そのファイルをクローズします。そのようなファイルは *Lisp* ライブラリー *(Lisp library)* とも呼ばれます。

`eval-buffer`関数がバッファー内のすべての式を評価するのと同様に、load 関数はファイル内のすべての式を評価します。異なるのは Emacs バッファー内のテキストではなく、load 関数はディスク上で見つかったファイル内のテキストを読み込んで評価することです。

ロードされたファイルは、ソースコードかバイトコンパイルされたコードとして Lisp 式を含んでいなければなりません。このファイル内の各フォームはトップレベルフォーム *(top-level form)* と呼ばれます。ロード可能なファイル内のフォームにたいする特別なフォーマットはありません。ファイル内のフォームはどれも同じように直接バッファーにタイプされて、そこで評価されるでしょう (実際ほとんどのコードはこの方法でテストされる)。多くの場合はそのフォームは関数定義と変数定義です。

Emacs はコンパイルされたダイナミックモジュールも同様にロードできます。これは Emacs Lisp プログラム内で、Emacs Lisp で記述されたパッケージと同様に使用するための、追加機能を提供する共有ライブラリーです。ダイナミックモジュールのロード時に、Emacs はそのモジュールが実装する必要がある特殊な名前の初期化関数を呼び出して、Emacs Lisp プログラムに追加の関数と変数を公開します。

特定の Emacs プリミティブで必要となるとあらかじめ判明している外部ライブラリーのオンデマンドローディングについては、Section 40.21 [Dynamic Libraries], page 1130 を参照してください。

16.1 プログラムがロードを行う方法

Emacs Lisp にはロードのためのインターフェイスがいくつかあります。たとえば `autoload`はファイル内で定義された関数にたいしてプレースホルダーとなるオブジェクトを作成します。この関数はオートロードされる関数を呼び出すために、ファイルからその関数の実際の定義の取得を試みます (Section 16.5 [Autoload], page 274 を参照)。`require`はファイルがまだロードされていない場合にファイルをロードします (Section 16.7 [Named Features], page 279 を参照)。これらすべての関数は処理を行うために最終的に `load`を呼び出します。

`load` *filename* **&optional** *missing-ok nomessage nosuffix must-suffix* [Function]

この関数は Lisp コードのファイルを見つけてオープンして、その中のすべてのフォームを評価してそのファイルをクローズする。

`load`はまずファイルを見つけるために、*`filename`*`.elc`という名前、つまり *filename*に拡張子‘`.elc`’を足した名前のファイルを探す。このようなファイルが存在したらそれをロードする。Emacs がダイナミックモジュール (Section 16.11 [Dynamic Modules], page 284 を参照) のサポートつきでコンパイルされていれば、、次に `load`は *`filename.ext`*という名前のファイルを探す。ここで *ext*は共有ライブラリーのシステム依存のファイル名拡張子である。その名前のファイルが存在しなければ、`load`は *`filename`*`.el`という名前のファイルを探す。このファイルが存在したらそれをロードする。最後に、もしこれらの名前がいずれも見つからなければ、`load`は何も付け足さない *filename*という名前のファイルを探してそれが存在したらロードする (`load`関数に *filename*を認識する賢さはない。`foo.el.el`のような正しくない名前のファイルでも、`(load "foo.el")`を評価してそれを見つけてしまうだろう)。

Auto Compression モードが有効 (残念ながらデフォルトでは有効) なら、`load`は他のファイル名を試みる前に圧縮されたバージョンのファイル名を探すのでファイルを見つけることがで

きない。圧縮されたファイルが存在したら、それを解凍してロードする。loadはファイル名にjka-compr-load-suffixes内の各サフィックスを足して圧縮されたバージョンを探す。この変数の値は文字列のリストでなければならない。標準的な値は (".gz")。

オプション引数 *nosuffix*が非 nilなら、loadはサフィックス '.elc'と '.el'のロードを試みない。この場合はロードしたいファイルの正確な名前を指定しなければならない。ただし Auto Compression モードが有効なら loadは圧縮されたバージョンを探すために、jka-compr-load-suffixesを使用する。正確なファイル名を指定して、*nosuffix*に tを使用することにより、foo.el.elのような名前のファイルにたいするロードの試みを抑止できる。

オプション引数 *must-suffix*が非 nilの場合、ロードに使用されるファイルの名前に明示的にディレクトリー名が含まれていなければ、loadはファイル名が '.el'か '.elc'、または共有ライブラリーの拡張子で終わること (もしかしたら圧縮による拡張子が付加されているかもしれない) を要求する。

オプション load-prefer-newerが非 nilなら、loadはサフィックスを検索するとき、どんなファイル ('.elc'、'.el'等) であっても、もっとも最近変更されたファイルのバージョンを選択する。

*filename*が fooや baz/foo.barのような相対ファイル名なら、loadは変数 load-pathを使用してそのファイルを探す。これは load-path内にリストされた各ディレクトリーに *filename* を追加して、最初に見つかった名前のマッチするファイルをロードする。デフォルトディレクトリーを意味する nilが load-pathで措定されたときだけ、カレントデフォルトディレクトリーを試みる。loadは load-path内の最初のディレクトリーで利用可能な 3 つのサフィックスすべてを試行してから、2 つ目のディレクトリーで 3 つのサフィックスすべてを試行する、...というようにファイルを探す。Section 16.3 [Library Search], page 272 を参照のこと。

最終的に見つかったファイル、および Emacs がそのファイルを見つけたディレクトリーが何であれ、Emacs はそのファイル名を変数 load-file-nameの値にセットする。

foo.elcが foo.elより古いと警告されたら、それは foo.elのリコンパイルを考慮すべきことを意味する。Chapter 17 [Byte Compilation], page 285 を参照のこと。

(コンパイルされていない) ソースファイルをロードしたとき、Emacs がファイルを visit したときと同じように loadは文字セットの変換を行う。Section 33.10 [Coding Systems], page 864 を参照のこと。

コンパイルされていないファイルをロードするとき、Emacs はそのファイルに含まれるすべてのマクロ (Chapter 14 [Macros], page 242 を参照) を展開する。わたしたちはこれを *eager* マクロ展開 *(eager macro expansion)* と呼んでいる。(関連するコードを実行するまで展開を延期しないで) これを行うことにより、コンパイルされていないコードの実行スピードが明らかに向上する。循環参照によりこのマクロ展開を行うことができないときもある。これの一番簡単な例は、ロードしようとしているファイルが他のファイルで定義されているマクロを参照しているが、そのファイルはロードしようとしているファイルを必要としている場合である。これは一般的には無害である。Emacs は問題の詳細を与えるために警告 ('Eager macro-expansion skipped due to cycle...') をプリントするが、単にその時点ではマクロを展開せずにそのファイルをロードする。あなたはこの問題が発生しないようにコードをリストラクチャーしたいと思うかもしれない。コンパイル済みファイルではマクロ展開はコンパイル時に行われるので、ロード時のマクロ展開は行われない。Section 14.3 [Compiling Macros], page 244 を参照のこと。

*nomessage*が非 nilでなければ、エコーエリアに 'Loading foo...'や 'Loading foo...done'のようなメッセージがロードの間に表示される。

ファイルをロードする間のハンドルされないエラーはロードを終了させる。autoloadのためのロードの場合、ロードの間に定義された任意の関数定義は元に戻される。

loadがロードするファイルを見つけられなかった場合には、通常は (‘Cannot open load file *filename*’のメッセージとともに) file-errorがシグナルされる。しかし *missing-ok*が非nilなら loadは単に nilをリターンする。

式の読み取りにたいして loadが readのかわりに使用する関数を指定するために、変数load-read-functionを使用できる。以下を参照されたい。

ファイルが正常にロードされたら、loadは tをリターンする。

`load-file` *filename* [Command]
このコマンドはファイル *filename*をロードする。*filename*が相対ファイル名のなら、それはカレントデフォルトディレクトリーを指定したとみなされる。このコマンドは load-pathを使用せず、サフィックスの追加もしない。しかし (Auto Compression モードが有効なら) 圧縮されたバージョンの検索を行う。ロードするファイル名を正確に指定したければ、このコマンドを使用すること。

`load-library` *library* [Command]
このコマンドは *library*という名前のライブラリーをロードする。このコマンドは引数を読み取る方法がインタラクティブであることを除き loadと同じ。Section “Lisp Libraries” in *The GNU Emacs Manual* を参照のこと。

`load-in-progress` [Variable]
この変数は Emacs がファイルをロード中なら非 nil、それ以外は nilである。

`load-file-name` [Variable]
このセクションの最初に説明した検索で Emacs がファイルを見つけて、そのファイルをロード中のとき、この変数の値はそのファイルの名前である。

`load-read-function` [Variable]
この変数は loadと eval-regionが式を読み取るために、readのかわりに使用する関数を指定する。指定する関数は readと同様、引数が 1 つの関数であること。

デフォルトではこの変数の値は read。Section 19.3 [Input Functions], page 334 を参照のこと。

この変数を使用するかわりに別の新たな方法を使用するほうが明確である。それは eval-regionの *read-function*引数にその関数を渡す方法である。[Eval], page 141 を参照のこと。

Emacs のビルドで loadがどのように使用されているかについての情報は、Section E.1 [Building Emacs], page 1175 を参照のこと。

16.2 ロードでの拡張子

ここでは loadが試行するサフィックスについて、技術的な詳細を説明します。

`load-suffixes` [Variable]
これは (ソースまたはコンパイル済みの)Emacs Lisp ファイルを示すサフィックスのリストである。空の文字列が含まないこと。loadは指定されたファイル名に Lisp ファイルのサフィックスを追加するときに、これらのサフィックスを使用する。標準的な値は (".elc" ".el")で、これは前のセクションで説明した振る舞いとなる。

`load-file-rep-suffixes` [Variable]

これは同じファイルにたいして異なる表現を示すサフィックスのリストである。このリストは空の文字列から開始されること。`load`はファイルを検索するときは、他のファイルを検索する前にこのリストのサフィックスを順番にファイル名に追加する。

Auto Compression モードを有効にすることにより `jka-compr-load-suffixes`のサフィックスがこのリストに追加され、無効にすると再びリストから取り除かれる。`load-file-rep-suffixes`の標準的な値は、Auto Compression モードが無効なら `("")`。`jka-compr-load-suffixes`の標準的な値が `(".gz")`であることを考慮すると、Auto Compression モードが有効な場合の `load-file-rep-suffixes`の標準的な値は `("" ".gz")`である。

`get-load-suffixes` [Function]

この関数は *must-suffix*引数が非 `nil`のときは、`load`が試みるべきすべてのサフィックスを順番にしたがったリストでリターンする。この関数は `load-suffixes`と `load-file-rep-suffixes`の両方を考慮する。`load-suffixes`、`jka-compr-load-suffixes`、`load-file-rep-suffixes`がすべて標準的な値の場合、この関数は Auto Compression モードが有効なら `(".elc" ".elc.gz" ".el" ".el.gz")`、無効なら `(".elc" ".el")`をリターンする。

まとめると、`load`は通常まず `(get-load-suffixes)`の値のサフィックスを試み、次に `load-file-rep-suffixes`を試みる。*nosuffix*が非 `nil`なら前者がスキップされ、*must-suffix*が非 `nil`なら後者がスキップされる。

`load-prefer-newer` [User Option]

このオプションが非 `nil`なら、ファイルが見つかった最初のサフィックスで停止せずに、`load`はすべてのサフィックスをテストして、一番新しいファイルを使用する。

16.3 ライブラリー検索

Emacs が Lisp ライブラリーをロードするときは、変数 `load-path`により指定されるディレクトリー内のライブラリーを検索します。

`load-path` [Variable]

この変数の値は `load`でファイルをロードするとき検索するディレクトリーのリスト。リストの各要素は文字列 (ディレクトリーでなければならない)、または `nil` (カレントワーキングディレクトリーを意味する)。

Emacs は起動時にいくつかのステップにより `load-path`の値をセットアップする。最初に Emacs がコンパイルされたときのデフォルトロケーションセット (default locations set) を使用して、`load-path`を初期化する。通常これは以下のようなディレクトリーである

`"/usr/local/share/emacs/`*version*`/lisp"`

(以降の例ではあなたがインストールした Emacs のインストールプレフィクスに合うように `/usr/local`を置き換えること。) これらのディレクトリーには、Emacs とともにインストールされた標準的な Lisp ファイルが含まれる。Emacs がこれらを見つけられなければ正常に起動しないだろう。

Emacs をビルドしたディレクトリーから起動した場合 ーーー つまり正式にインストールされた実行形式ではない Emacs を起動した場合— 、Emacs はビルドされたディレクトリーのソースの `lisp`ディレクトリーを使用して `load-path`を初期化する。ソースとは別のディレクトリーで Emacs

をビルドした場合は、ビルドしたディレクトリーの `lisp`ディレクトリーも追加する (いずれも要素は絶対ファイル名になる)。

`--no-site-lisp`オプションで Emacs を起動した場合を除き、`load-path`の先頭にさらに 2 つの `site-lisp`を追加する。これらはローカルにインストールされた Lisp ファイルで、通常は:

```
"/usr/local/share/emacs/version/site-lisp"
```

および

```
"/usr/local/share/emacs/site-lisp"
```

の形式である。1 つ目は特定のバージョンの Emacs にたいしてローカルにインストールされたものである。2 つ目はインストールされたすべてのバージョンの Emacs が使用することを意図してローカルにインストールされたものである (インストールされたものでない Emacs が実行されると、もし存在すればソースディレクトリーとビルドディレクトリーの `site-lisp`ディレクトリーも追加される。これらのディレクトリーは通常は `site-lisp`ディレクトリーを含まない)。

環境変数 `EMACSLOADPATH`がセットされていたら、上述の初期化プロセスが変更される。Emacs はこの環境変数の値にもとづいて `load-path`を初期化する。

`EMACSLOADPATH`の構文は `PATH`で使用される構文と同様。ディレクトリーは '`:`' (オペレーティングシステムによっては '`;`') で区切られる。以下は (`sh`スタイルのシェルから)`EMACSLOADPATH`変数をセットする例である:

```
export EMACSLOADPATH=/home/foo/.emacs.d/lisp:
```

環境変数の値内の空の要素は、(上記例のような) 末尾、先頭、中間のいずれにあるかに関わらず、標準の初期化処理により決定される `load-path`のデフォルト値に置き換えられる。そのような空要素が存在しなければ `EMACSLOADPATH`により `load-path`全体が指定される。空要素、または標準の Lisp ファイルを含むディレクトリーへの明示的なパスのいずれかを含めなければならない。さもないと Emacs が関数を見つけられなくなる (`load-path`を変更する他の方法は、Emacs 起動時にコマンドラインオプション`-L`を使用する方法である。以下参照)。

`load-path`内の各ディレクトリーにたいして、Emacs はそのディレクトリーがファイル `subdirs.el`を含むか確認して、もしあればそれをロードする。`subdirs.el`ファイルは、`load-path`のディレクトリーにたいして任意のサブディレクトリーを追加するためのコードが含まれており、Emacs がビルド/インストールされたときに作成される。サブディレクトリーと複数階層下のレベルのサブディレクトリーの両方が直接追加される。ただし名前の最初が英数字でないディレクトリー、名前が `RCS`または `CVS`のディレクトリー、名前が`.nosearch`というファイルを含むディレクトリーは除外される。

次に Emacs はコマンドラインオプション`-L`(Section "Action Arguments" in *The GNU Emacs Manual* を参照) で指定したロードディレクトリーを追加する。もしあればオプションパッケージ (Section 41.1 [Packaging Basics], page 1134 を参照) がインストールされた場所も追加する。

init ファイル (Section 40.1.2 [Init File], page 1093 を参照) で `load-path`に 1 つ以上のディレクトリーを追加するコードを記述するのは一般的に行なわれている。たとえば:

```
(push "~/.emacs.d/lisp" load-path)
```

Emacs のダンプには `load-path`の特別な値を使用する。ダンプされた Emacs をカスタマイズするために `site-load.el`か `site-init.el`を使用する場合、これらのファイルが行った `load-path`にたいする変更はすべてダンプ後に失われる。

`locate-library` *library* **&optional** *nosuffix path interactive-call* [Command]

このコマンドはライブラリー *library*の正確なファイル名を探す。`load`と同じ方法でライブラリーを検索を行い、引数 *nosuffix*も `load`の場合と同じ意味をもつ。*library*に指定する名前にはサフィックス '`.elc`'または '`.el`'を追加しないこと。

*path*が非 nilなら load-pathのかわりにそのディレクトリーのリストが使用される。

locate-libraryがプログラムから呼び出されたときはファイル名を文字列としてリターンする。ユーザーがインタラクティブに locate-libraryを実行したときは、引数 *interactive-call* が tとなり、これは locate-libraryにたいしてファイル名をエコーエリアに表示するよう指示する。

`list-load-path-shadows` **&optional** *stringp* [Command]

このコマンドはシャドー *(shadowed)* された Emacs Lisp ファイルを表示する。シャドーされたファイルとは、load-pathのディレクトリーに存在するにも関わらず、load-pathのディレクトリーリスト内で前の位置にある他のディレクトリーに同じ名前のファイルが存在するため、通常はロードされないファイルのことである。

たとえば以下のように load-pathがセットされていたとする

```
("/opt/emacs/site-lisp" "/usr/share/emacs/23.3/lisp")
```

そして両方のディレクトリーに foo.elという名前のファイルがあるとする。この場合、(require 'foo)は決して 2 つ目のディレクトリーのファイルをロードしない。このような状況は Emacs がインストールされた方法に問題があることを示唆する。

Lisp から呼び出されたると、この関数はシャドーされたファイルリストをバッファー内に表示するかわりに、それのメッセージをプリントする。オプション引数 stringpが非 nilなら、かわりにシャドーされたファイルを文字列としてリターンする。

16.4 非 ASCII文字のロード

Emacs Lisp プログラムが非 ASCII文字の文字列定数を含むとき、Emacs はそれらをユニバイト文字列かマルチバイト文字列のいずれかで表現する場合があります。どちらの表現が使用されるかは、そのファイルがどのように Emacs に読み込まれたかに依存します。マルチバイト表現へのデコーディングとともに読み込まれた場合、Lisp プログラム内のテキストはマルチバイトのテキストとなり、ファイル内の文字列定数はマルチバイト文字列になります。(たとえば)Latin-1 文字を含むファイルをデコーディングなしで読み込むと、そのプログラムのテキストはユニバイトのテキストとなり、ファイル内の文字列定数はユニバイト文字列になります。Section 33.10 [Coding Systems], page 864 を参照してください。

マルチバイト文字列がユニバイトバッファーに挿入されるときは自動的にユニバイトに変換されるため、大部分の Emacs Lisp プログラムにおいて、マルチバイト文字列が非 ASCII文字列であるという事実を意識させないようにするべきです。しかしこれが行われことにより違いが生じる場合には、ローカル変数セクションに '`coding: raw-text`'と記述することにより、特定の Lisp ファイルを強制的にユニバイトとして解釈させることができます。この識別子により、そのファイルは無条件でユニバイトとして解釈されます。これは`?v`*literal*で記述された非 ASCII文字にキーバインドするとき重要になります。

16.5 autoload

オートロード *(autoload:* 自動ロード*)* の機能により、定義されているファイルをロードすることなく関数やマクロの存在を登録できます。関数の最初の呼び出しで実際の定義およびその他の関連するコードをインストールするために適切なライブラリーを自動的にロードして、すべてがすでにロードされていたかのように実際の定義を実行します。関数やマクロのドキュメントの参照 (Section 24.1 [Documentation Basics], page 519 を参照)、変数名や関数名の補完 (以下の Section 16.5.1 [Autoload by Prefix], page 278 を参照) によってもオートロードが発生します。

オートロードされた関数をセットアップするには 2 つの方法があります。それは autoloadを呼び出す方法と、ソースの実際の定義の前に、“マジック” コメントを記述する方法です。autoloadはオートロードのための低レベルのプリミティブです。任意の Lisp プログラムが、任意のタイミングで autoloadを呼び出すことができます。Emacs とともにインストールされるパッケージにとって、マジックコメントは関数をオートロードできるようににするための一番便利な方法です。そのコメント自身は何も行いませんが、コマンド update-file-autoloadsにたいするガイドの役目を果たします。このコマンドは autoloadの呼び出しを構築して、Emacs ビルド時に実行されるようにアレンジします。

autoload *function filename* **&optional** *docstring interactive type* [Function]

この関数は *filename*から自動的にロードされるように、*function*という名前の関数 (かマクロ) を定義する。文字列 *filename*には *function*の実際の定義を取得するファイルを指定する。

*filename*がディレクトリー名、またはサフィックス .elと .elcのいずれも含まなければ、この関数はこれらのサフィックスのいずれかを強制的に追加して、サフィックスがないただの *filename* という名前のファイルはロードしない (変数 load-suffixesにより要求される正確なサフィックスが指定される)。

引数 *docstring*はその関数のドキュメント文字列である。autoloadの呼び出しでドキュメント文字列を指定することにより、その関数の実際の定義をロードせずにドキュメントを見ることが可能になる。この引数の値は通常は関数定義のドキュメント文字列と等しいこと。もし等しくなければ、その関数定義のドキュメント文字列がロード時に有効になる。

*interactive*が非 nilなら、その関数はインタラクティブに呼び出すことが可能になる。これにより *function*の実際の定義をロードせずに、*M-x*による補完が機能するようになる。ここでは完全なインタラクティブ仕様は与えられない。完全な仕様はユーザーが実際に *function*を呼び出すまで必要ない。ユーザーが実際に呼び出したときに、実際の定義がロードされる。

普通の関数と同様、マクロとキーマップをオートロードできる。*function*が実際にはマクロなら *type*に macro、キーマップのなら *type*に keymapを指定する。Emacs のさまざまな部分では、実際の定義をロードせずにこれらの情報を知ることが必要とされる。

オートロードされたキーマップは、あるプレフィクスキーがシンボル *function*にバインドされているとき、キーを探す間に自動的にロードされる。そのキーマップにたいする他の類のアクセスではオートロードは発生しない。特に Lisp プログラムが変数の値からそのキーマップを取得して define-keyを呼び出した場合には、たとえその変数の名前がシンボル *function*と同じであってもオートロードは発生しない。

*function*が非 void のオートロードされたオブジェクトではない関数定義をもつなら、その関数は何も行わずに nilをリターンする。それ以外ならオートロードされたオブジェクト (Section 2.4.19 [Autoload Type], page 24 を参照) を作成して、それを *function*にたいする関数定義として格納する。オートロードされたオブジェクトは以下の形式をもつ:

```
(autoload filename docstring interactive type)
```

たとえば、

```
(symbol-function 'run-prolog)
     ⇒ (autoload "prolog" 169681 t nil)
```

このような場合、"prolog"はロードするファイルの名前、169681 は emacs/etc/DOCファイル (Section 24.1 [Documentation Basics], page 519 を参照) 内のドキュメント文字列への参照で、tはその関数がインタラクティブであること、nilはそれがマクロやキーマップでないことを意味する。

`autoloadp` *object* [Function]

この関数は *object*がオートロードされたオブジェクトなら非`nil`をリターンする。たとえば`run-prolog`がオートロードされたオブジェクトかチェックするには以下を評価する

```
(autoloadp (symbol-function 'run-prolog))
```

オートロードされたファイルは、通常は他の定義を含み 1 つ以上の機能を必要としたり、あるいは提供するかもしれません。(内容の評価でのエラーにより) そのファイルが完全にロードされていなければ、そのロードの間に行われた関数定義や`provide`の呼び出しはアンドゥされます。これはそのファイルからオートロードされる関数にたいして再度呼び出しを試みたときに、そのファイルを確実に再ロードさせるためです。こうしないと、そのファイル内のいくつかの関数はアボートしたロードにより定義されていて、それらはロードされない修正後のファイルで提供される正しいサブルーチンを欠くため、正しく機能しないからです。

オートロードされたファイルが意図した Lisp 関数またはマクロの定義に失敗すると、データ`"Autoloading failed to define function` *`function-name`*`"`とともにエラーがシグナルされます。

オートロードのマジックコメント (*autoload cookie* とも呼ばれる) は、オートロード可能なソースファイル内の実際の定義の直前にある、‘`;;;###autoload`’だけの行から構成されます。コマンド *`M-x update-file-autoloads`*は、対応する`autoload`呼び出しを`loaddefs.el`内に書き込みます (autoload cookie となる文字列と`update-file-autoloads`で生成されるファイルの名前は上述のデフォルトから変更可能です。以下参照)。Emacs のビルドでは`loaddefs.el`をロードするために`autoload`を呼び出します。*`M-x update-directory-autoloads`*はより強力です。このコマンドはカレントディレクトリー内のすべてのファイルにたいするオートロードを更新します。

このマジックコメントは任意の種類のフォームを`loaddefs.el`内にコピーできます。このマジックコメントに続くフォームはそのままコピーされます。しかしオートロード機能が特別に処理するフォームの場合は除外されます (たとえば`autoload`内への変換)。以下はそのままコピーされないフォームです:

関数や関数風オブジェクトの定義:

> `defun`と `defmacro`。`cl-defun`と `cl-defmacro`(Section “Argument Lists” in *Common Lisp Extensions* を参照)、および `define-overloadable-function` (`mode-local.el`内のコメントを参照) も該当する。

メジャーモードとマイナーモードの定義:

> `define-minor-mode`、`define-globalized-minor-mode`、`define-generic-mode`、`define-derived-mode`、`easy-mmode-define-minor-mode`、`easy-mmode-define-global-mode`、`define-compilation-mode`、`define-global-minor-mode`。

その他のタイプの定義:

> `defcustom`、`defgroup`、`defclass` (*EIEIO* を参照)、および `define-skeleton` (*Autotyping* を参照)。

ビルド時にそのファイル自身をロードするときにフォームを実行しないようにするためにマジックコメントを使用することもできます。これを行なうにはマジックコメントと同じ行にフォームを記述します。これはコメントなのでソースファイルをロードするときには何も行いません。ただし *`M-x update-file-autoloads`*では、Emacs ビルド時に実行されたものは *`M-x update-file-autoloads`*にコピーします。

以下はマジックコメントによるオートロードのために`doctor`を準備する例です:

```
;;;###autoload
```

```
(defun doctor ()
  "Switch to *doctor* buffer and start giving psychotherapy."
  (interactive)
  (switch-to-buffer "*doctor*")
  (doctor-mode))
```

これにより以下が `loaddefs.el`内に書き込まれます:

```
(autoload 'doctor "doctor" "\
Switch to *doctor* buffer and start giving psychotherapy.

\(fn)" t nil)
```

ダブルクォートの直後のバックスラッシュと改行は、`loaddefs.el`のようなプリロードされた未コンパイルの Lisp ファイルだけに使用される慣習です。これは `make-docfile`にたいして、ドキュメント文字列を `etc/DOC`ファイルに配置するよう指示します。Section E.1 [Building Emacs], page 1175 を参照してください。また `lib-src/make-docfile.c`内のコメントも参照してください。ドキュメント文字列の使い方 (usage part) の中の‘`(fn)`’は、種々のヘルプ関数 (Section 24.6 [Help Functions], page 526 を参照) が表示するときに、その関数の名前に置き換えられます。

関数定義手法として既知ではなく、認められてもいないような、通常とは異なるマクロにより関数定義を記述した場合、通常のオートロードのマジックコメントの使用によって定義全体が `loaddefs.el`内にコピーされるでしょう。これは期待した動作ではありません。かわりに以下を記述することにより、意図した `autoload`呼び出しを `loaddefs.el`内に配置することができます。

```
;;;###autoload (autoload 'foo "myfile")
(mydefunmacro foo
  ...)
```

autoload cookie としてデフォルト以外の文字列を使用して、デフォルトの `loaddefs.el`とは異なるファイル内に対応するオートロード呼び出しを記述できます。これを制御するために Emacs は 2 つの変数を提供します:

`generate-autoload-cookie` [Variable]
: この変数の値は Lisp コメントの文法に準じた文字列である。*`M-x update-file-autoloads`* はその cookie の後の Lisp フォームを、cookie が生成したオートロードファイル内にコピーします。この変数のデフォルト値は`";;;###autoload"`。

`generated-autoload-file` [Variable]
: この変数の値は、オートロード呼び出しが書き込まれる Emacs Lisp ファイルを命名する。デフォルト値は `loaddefs.el`だが、(たとえば`.el`ファイル内のセクション Local Variables)) をオーバーライドできる。オートロードファイルは、フォームフィード文字で開始される終端を含んでいると仮定される。

以下の関数はオートロードオブジェクトにより指定されたライブラリーを明示的にロードするために使用されるかもしれません:

`autoload-do-load` *autoload* **&optional** *name macro-only* [Function]
: この関数はオートロードオブジェクト *autoload*により指定されたロードを処理する。オプション引数 *name*に非 `nil`を指定するなら、関数値が *autoload*となるシンボルを指定すること。この場合、この関数のリターン値がそのシンボルの新しい関数値になる。オプション引数 *macro-only* の値が `macro`なら、この関数は関数ではなくマクロのロードだけを有効にする。

16.5.1 プレフィックスによる autoload

コマンド `describe-variable`と `describe-function`の補完の間に、Emacs は補完されるプレフィクスにマッチする定義を含むファイルのロードを試みます。変数 `definition-prefixes`にはプレフィクスとそれをロードするファイルのリストをマップするハッシュテーブルが保持されています。このマッピングのエントリーは、`update-file-autoloads` (Section 16.5 [Autoload], page 274 を参照) が生成する `register-definition-prefixes`を呼び出すことによって追加されます。ロードする価値のある定義を含んでいないファイル (たとえばテストファイル) ではファイルローカル変数として `autoload-compute-prefixes`を `nil`にセットする必要があります。

16.5.2 autoload を使用するケース

本当に必要でなければ autoload コメントを追加しないでください。コードを autoload することは、それが常にグローバルに可視になることを意味しています。一度あるアイテムが autoload されれば、autoload 以前の状態 (autoload 後には明示的なロードなしで通常のように使用できる) に戻るための互換性のある手段はありません。

- autoload するアイテムとしてもっとも一般的なライブラリーにたいするインタラクティブなエントリーポイント。たとえば Python コードの編集用のメジャーモードを定義するライブラリーが `python.el`なら、`python-mode`関数の定義を autoload すればユーザーは単に *`M-x python-mode`*を使用してライブラリーをロードできる。
- 変数は通常は autoload する必要はない。例外はライブラリー定義全体をロードしなくても変数自体が有用な場合 (これの例としては `find-exec-terminator`のようなものが該当するだろう)。
- ユーザーがセットできるように、ユーザーオプションを autoload しないこと。
- 別ファイルでのコンパイラー警告を抑制するために autoload のコメントを決して追加してはならない。警告の発生元となるファイルでは未定義の変数にたいする警告の抑制には `(defvar foo)`、未定義の関数にたいする警告の抑制には `declare-function` (Section 13.15 [Declaring Functions], page 238 を参照) を使用するか、あるいは使用する関連ライブラリーを require したり明示的に autoload の命令文を使用すること。

16.6 多重ロード

1 つの Emacs セッション内でファイルを複数回ロードできます。たとえばバッファーで関数定義を編集して再インストールした後に元のバージョンに戻したいときがあるかもしれません。これは元のファイルをリロードすることにより行なうことができます。

ファイルのロードやリロードを行う際、`load`と `load-library`関数は未コンパイルのファイルではなく、バイトコンパイルされた同名のファイルを自動的にロードすることに留意してください。ファイルを再記述して保存後に再インストールする場合には、新しいバージョンをバイトコンパイルする必要があります。さもないと Emacs は新しいソースではなく、古いバイトコンパイルされたファイルをロードしてしまうでしょう! この場合にはファイルロード時に表示されるメッセージに、そのファイルのリコンパイルを促す‘`(compiled; note, source is newer)`’というメッセージが含まれます。

Lisp ライブラリーファイル内にフォームを記述するときは、そのファイルが複数回ロードされるかもしれないことに留意してください。たとえば、そのライブラリーをリロードするときには、各変数が再初期化されるべきかどうか考慮してください。。変数がすでに初期化されていれば、`defvar`はその変数の値を変更しません (Section 12.5 [Defining Variables], page 178 を参照)。

alist に要素を追加するもっともシンプルな方法は、以下のようなものでしょう:

```
(push '(leif-mode " Leif") minor-mode-alist)
```

しかしこれはそのライブラリーがリロードされると、複数の要素を追加してしまうでしょう。この問題を避けるには add-to-list(Section 5.5 [List Variables], page 79 を参照) を使用します:

```
(add-to-list 'minor-mode-alist '(leif-mode " Leif"))
```

時にはライブラリーが既にロード済みか、明示的にテストしたいときがあるでしょう。そのライブラリーが provideを使用して名前付きフィーチャ(named feature) を提供していれば、featurepを使用して以前に provideが実行されているかテストすることができます。かわりに以下のようにすることもできます:

```
(defvar foo-was-loaded nil)

(unless foo-was-loaded
  execute-first-time-only
  (setq foo-was-loaded t))
```

16.7 名前つき機能

provideと requireは、autoloadにかわってファイルを自動的にロードする関数です。これらは名前付きのフィーチャ *(feature:* 機能*)* という面で機能します。オートロードは特定の関数の呼び出しをトリガーにしますが、フィーチャは最初は他のプログラムが名前により問い合わせたときにロードされます。

フィーチャ名とは関数や変数などのコレクションを表すシンボルです。これらを定義するファイルは、そのフィーチャをプロバイド *(provide:* 提供*)* するべきです。これらのフィーチャを使用する他のプログラムは、その機能をリクワイア *(require:* 要求*)* することによって、それらが定義されているか確認できるでしょう。これは定義がまだロードされていなければ、定義ファイルをロードします。

フィーチャをリクワイアするには、フィーチャ名を引数として requireを呼び出します。requireは意図する機能がすでにプロバイドされているか確認するために、グローバル変数 featuresを調べます。もしプロバイドされていなければ、適切なファイルからそのフィーチャをロードします。このファイルはそのフィーチャを featuresに追加するために、トップレベルで provideを呼び出すべきです。これに失敗すると requireはエラーをシグナルします。

たとえば idlwave.el内の idlwave-complete-filenameにたいする定義には以下のコードが含まれます:

```
(defun idlwave-complete-filename ()
  "Use the comint stuff to complete a file name."
   (require 'comint)
   (let* ((comint-file-name-chars "~/A-Za-z0-9+@:_.$#%={}\\-")
          (comint-completion-addsuffix nil)
          ...)
       (comint-dynamic-complete-filename)))
```

式 (require 'comint)は comint.elがまだロードされていなければ、comint-dynamic-complete-filenameが確実に定義されるようにそのファイルをロードします。フィーチャは通常はそれらを提供するファイルにしたがって命名されるため、requireにファイル名を与える必要はありません (requireの命令文が letの body の外側にあるのが重要なことに注意。変数が let バインドされているライブラリーをロードすることにより、意図せぬ結果、つまり let を exit した後にその変数がアンバインドされる)。

comint.elには以下のトップレベル式が含まれます:

```
(provide 'comint)
```

これは comintをグローバルなリスト featuresに追加するので、(require 'comint)は今後何も行う必要がないことを知ることができます。

ファイルのトップレベルで requireが使用されたときは、それをロードしたときと同様、そのファイルをバイトコンパイル (Chapter 17 [Byte Compilation], page 285 を参照) するときにも効果が表れます。これはリクワイアされたパッケージがマクロを含んでいて、バイトコンパイラーがそれを知らなければならない場合です。これは requireによりロードされるファイルで定義される関数と変数にへのバイトコンパイラーの警告も無効にします。

バイトコンパイルの間にトップレベルの requireが評価されるとしても、provide呼び出しは評価されません。したがって以下の例のように provideの後に同じ機能にたいする requireを含めることにより、バイトコンパイル前に定義しているファイルを確実にロードできます。

```
(provide 'my-feature)  ; バイトコンパイラーには無視され
                       ;   loadには評価される
(require 'my-feature)  ; バイトコンパイラーにより評価される。
```

コンパイラーは provideを無視して、その後に対象のファイルをロードすることにより requireが処理されます。ファイルのロードは provide呼び出しを実行するので、後続の requireはファイルがロードされていれば何も行いません。

`provide` *feature* **&optional** *subfeatures* [Function]

この関数はカレント Emacs セッションに *feature*がロードされたこと、あるいはロードされつつあることをアナウンスする。これは *feature*に関連する機能が他の Lisp プログラムから利用可能できる、あるいは利用可能になることを意味する。

provide呼び出にによる直接的な効果は、リスト *feature*内にまだ追加されていなければ *feature*の先頭にそれを追加して、それを必要としている eval-after-loadコードを呼び出すことである (Section 16.10 [Hooks for Loading], page 283 を参照)。引数 *feature*はシンボルでなければならない。provideは *feature*をリターンする。

*subfeatures*が与えられたら、それは *feature*の当該バージョンによりプロバイドされる特定のサブフィーチャのセットを示すシンボルのリストであること。featurepを使用して、サブフィーチャの存在をテストできる。そのパッケージがロードされるかどうか、あるいは与えられるバージョンで存在するかどうか不明であるようなあるパッケージ (1 つの *feature*) において、パッケージの種々の部分やパッケージ機能に命名することでそのパッケージを使いやすくするのが困難なほど複雑なときに使用するというのがサブフィーチャのアイデアである。Section 38.17.3 [Network Feature Testing], page 972 の例を参照されたい。

```
features
     ⇒ (bar bish)

(provide 'foo)
     ⇒ foo
features
     ⇒ (foo bar bish)
```

オートロードによりあるファイルがロードされて、その内容の評価エラーによりストップしたときは、そのロードの間に発生した関数定義や provide呼び出しはアンドゥされる。Section 16.5 [Autoload], page 274 を参照のこと。

`require` *feature* **&optional** *filename noerror* [Function]

この関数はカレント Emacs セッションにおいて、*feature*が存在するかどうかを ((`featurep` *feature*)を使用する。以下参照) をチェックする。引数 *feature*はシンボルでなければならない。

そのフィーチャが存在しなければ、requireはloadによって *filename*をロードする。*filename*が与えられなければ、シンボル *feature*の名前がロードするファイル名のベースとして使用される。しかしこの場合、requireは *feature*を探すためにサフィックス‘.el’と‘.elc’の追加を強制する (圧縮ファイルのサフィックスに拡張されるかもしれない)。名前がただの *feature*というファイルは使用されない (変数load-suffixesは要求される Lisp サフィックスを正確に指定する)。

*noerror*が非 nilなら、ファイルの実際のロードにおけるエラーを抑止する。この場合はそのファイルのロードが失敗するとrequireはnilをリターンする。通常ではrequireは *feature*をリターンする。

ファイルのロードは成功したが *feature*をプロバイドしていない場合には、requireは欠落している機能に関するエラーをシグナルする。

`featurep` *feature* **&optional** *subfeature* [Function]

この関数はカレント Emacs セッションで *feature*がプロバイドされていれば (たとえば *feature*が featuresのメンバーなら)tをリターンする。*subfeature*が非 nilなら、この関数はサブフィーチャも同様にプロバイドされているとき (たとえば *subfeature*がシンボル *feature*のプロパティsubfeatureのメンバーのとき) だけ tをリターンする。

`features` [Variable]

この変数の値はシンボルのリストであり、そのシンボルはカレント Emacs セッションにロードされたフィーチャである。シンボルはそれぞれ provideを呼び出すことにより、このリストに put されたものである。リスト features内の要素の順番に意味はない。

16.8 どのファイルで特定のシンボルが定義されているか

`symbol-file` *symbol* **&optional** *type* [Function]

この関数は *symbol*を定義しているファイルの名前をリターンする。*type*が nilなら、どのようなタイプの定義も受け入れる。*type*が defunなら関数定義、defvarは変数定義、defaceはフェイス定義だけを指定する。

値は通常は絶対ファイル名である。定義がどのファイルにも関係しなければ nilになることもある。*symbol*がオートロード関数を指定するなら、値が拡張子なしの相対ファイル名になることもある。

symbol-fileは変数 load-historyの値にもとづく。

`load-history` [Variable]

この変数の値はロードされたライブラリーファイルの名前を、それらが定義する関数と変数の名前、およびそれらがプロバイドまたはリクワイアするフィーチャに関連付ける alist である。

この alist 内の各要素は、1 つのロード済みライブラリー (スタートアップ時にプリロードされたライブラリーを含む) を記述する。要素は CAR がライブラリーの絶対ファイル名 (文字列) であるようなリストである。残りのリスト要素は以下の形式をもつ:

var シンボル *var*が変数として定義された。

(defun . *fun*)
関数 *fun*が定義された。

(t . *fun*) 関数 *fun*はそのライブラリーが関数として再定義する前はオートロードとして定義されていた。後続の要素は常に (defun . *fun*)であり、これは *fun*を関数として定義する。

`(autoload . fun)`
関数 *fun*はオートロードとして定義された。

`(defface . face)`
フェイス *face*が定義された。

`(require . feature)`
フィーチャ *feature*がリクワイアされた。

`(provide . feature)`
フィーチャ *feature*がプロバイドされた。

`(cl-defmethod method specializers)`
`cl-defmethod`を使用してスペシャライザー *specializers*とともに *method*という名前が定義された。

`(define-type . type)`
型 *type*が定義された。

`load-history`の値には、CAR が `nil`であるような要素が 1 つ含まれるかもしれない。この要素はファイルを visit していないバッファーで `eval-buffer`により作成された定義を記述する。

コマンド `eval-region`は `load-history`を更新しますが、要素を置き換えずに、visit されているファイルの要素にたいして定義されたシンボルを追加します。Section 10.4 [Eval], page 140 を参照してください。

16.9 アンロード

他の Lisp オブジェクト用にメモリーを回収するために、ライブラリーによりロードされた関数や変数を破棄することができます。これを行うには関数 `unload-feature`を使用します:

`unload-feature` *feature* **&optional** *force* [Command]
このコマンドはフィーチャ *feature*をプロバイドしていたライブラリーをアンロードする。そのライブラリー内の `defun`、`defalias`、`defsubst`、`defmacro`、`defconst`、`defvar`、`defcustom`によって定義されたすべての関数、マクロ、変数は未定義になる。その後に、それらのシンボルにたいして事前に関連付けられていたオートロードをリストアする (ロードはシンボルの `autoload`プロパティにこれらを保存している)。

以前の定義をリストアする前に、特定のフックからそのライブラリー内の関数を取り除くために、`unload-feature`は `remove-hook`を実行する。これらのフックには名前が '`-hook`'(または廃止されたサフィックス '`-hooks`') で終わる変数、加えて `unload-feature-special-hooks`、同様に `auto-mode-alist`にリストされた変数も含まれる。これは重要なフックがすでに定義されていない関数を参照をすることにより、Emacs の機能が停止することを防ぐためである。

標準的なアンロードアクティビティでは、そのライブラリー内の関数の ELP プロファイリング、そのライブラリーによりプロバイドされたフィーチャ、そのライブラリーで定義された変数に保持されたタイマーを取り消す。

これらの基準が機能不全を防ぐのに十分でなければ、ライブラリーは *feature*`-unload-function`という名前の明示的なアンローダーを定義できる。そのシンボルが関数として定義されていたら、`unload-feature`は何かを行う前にまず引数なしでそれを呼び出す。これはライブラリーのアンロードのために適切なすべてのことを行うことができる。これが `nil`をリターンしたら、`unload-feature`は通常のアンロードアクションを処理する。それ以外ならアンロード処理は完了したとみなす。

`unload-feature`は通常は他のライブラリーが依存するライブラリーのアンロードを拒絶する (ライブラリー *b*にたいする`require`がライブラリー *a*に含まれるなら、*a*は *b*に依存している)。オプション引数 *force*が非`nil`なら依存関係は無視されて、どのようなライブラリーもアンロードできる。

`unload-feature`関数はLispで記述されており、その動作は変数`load-history`にもとづきます。

`unload-feature-special-hooks` [Variable]
この変数はライブラリー内で定義された関数を取り除くために、ライブラリーをアンロードする前にスキャンするフックのリストを保持する。

16.10 ロードのためのフック

変数`after-load-functions`を使用することにより、Emacsがライブラリーをロードするたびにコードを実行させることができます:

`after-load-functions` [Variable]
このアブノーマルフック (abnormal hook) は、ファイルをロードした後に実行される。フック内の各関数は 1 つの引数 (ロードされたファイルの絶対ファイル名) で呼び出される。

特定のライブラリーのロード後にコードを実行したければ、マクロ`with-eval-after-load`を使用します:

`with-eval-after-load` *library body...* [Macro]
このマクロは *library*がロードされるたびに、ファイル *library*のロードの最後で *body*が評価されるよう準備する。*library*がすでにロード済みなら即座に *body*を評価する。

ファイル名 *library*にディレクトリーや拡張子を与える必要はない。通常は以下のようにファイル名だけを与える:

```
(with-eval-after-load "edebug" (def-edebug-spec c-point t))
```

どのファイルが評価をトリガーするか制限するには、ディレクトリーか拡張子、またはその両方を *library*に含める。実際のファイル名 (シンボリックリンク名はすべて除外される) が、与えられた名前すべてにマッチするファイルだけがマッチとなる。以下の例ではどこかのディレクトリー`..../foo/bar`にある`my_inst.elc`や`my_inst.elc.gz`は評価をトリガーするが、`my_inst.el`は異なる。:

```
(with-eval-after-load "foo/bar/my_inst.elc" ...)
```

*library*はフィーチャ(たとえばシンボル) でもよく、その場合には`(provide` *`library`*`)`を呼び出す任意のファイルの最後に *body*が評価される。

*body*内でのエラーはロードをアンドゥしないが、*body*の残りの実行を防げる。

上手く設計されたLispプログラムは、通常は`with-eval-after-load`を使用するべきではありません。(外部からの使用を意図した) 他のライブラリーで定義された変数を調べたりセットする必要があるなら、それは即座に行うことができます−−−そのライブラリーがロードされるのを待つ必要はありまん。そのライブラリーで定義された関数を呼び出す必要があるならそのライブラリーをロードすべきであって、それには`require`(Section 16.7 [Named Features], page 279 を参照) が適しています。

16.11 Emacs のダイナミックモジュール

ダイナミック *Emacs* モジュール *(dynamic Emacs module:* 動的 *Emacs* モジュール*)* とは、Emacs Lisp で記述されたパッケージのように、Emacs Lisp プログラムで使用するための追加機能を提供する共有ライブラリーです。

Emacs Lisp パッケージをロードする関数は、ダイナミックモジュールのロードもできます。これらの関数はファイル名の拡張子、いわゆる “サフィックス” を調べることによってダイナミックモジュールを認識します。

`module-file-suffix` [Variable]
: この変数はモジュールファイルにたいするシステム依存なファイル名拡張子を保持する。POSIX ホストでは`.so`、MS-Windows では`.dll`。

すべてのダイナミックモジュールは`emacs_module_init`という名前の C から呼び出し可能な関数をエクスポートする必要があります。Emacs はそのモジュールをロードする`load`や`require`の呼び出しの一部として、その関数を呼び出します。自身のコードが GPL または互換ライセンスの下にリリースされていることを示すために、`plugin_is_GPL_compatible`という名前のシンボルもエクスポートするべきです。このようなシンボルをエクスポートしないモジュールをプログラムがロードしようとすると、Emacs はエラーをシグナルするでしょう。

あるモジュールが Emacs の関数の呼び出しを必要とするなら、Emacs ディストリビューションに同梱されているヘッダーファイル`emacs-module.h`で定義および文書化された API (Application Programming Interface)を介して行う必要があります。独自モジュールを記述する際におけるこの API 使用の詳細は Section E.8 [Writing Dynamic Modules], page 1188 を参照してください。

モジュールはモジュールが定義する C 構造体への埋め込みポインターとなる Lisp オブジェクト`user-ptr`を作成できます。これはモジュールが作成する複雑なデータ構造を維持してモジュールの関数に渡して戻す場合に有用です。user-ptr オブジェクトはそれに関連付けられるファイナライザー *(finalizer:* オブジェクトがガーベージコレクトされる際に呼び出される関数*)* をもつこともできます。これはメモリー、オープンしたファイル記述子等のデータ構造の背後に割り当てられたリソースの解放に有用です。Section E.8.3 [Module Values], page 1193 を参照してください。

`user-ptrp` *object* [Function]
: この関数は引数が`user-ptr`オブジェクトなら`t`をリターンする。

`module-load` *file* [Function]
: Emacs は指定された *file*からモジュールをロードして、そのモジュールに必要な初期化えを行うために、この低レベルのプリミティブを呼び出す。これはモジュールがシンボル`plugin_is_GPL_compatible`をエクスポートすることを保証して、モジュールの`emacs_module_init`関数を呼び出し、その関数のリターン値がエラーを示したりユーザーが初期化中に *C-g*をタイプしたらエラーをシグナルする。初期化が成功すると`module-load`は`t`をリターンする。この関数は`load`とは異なり既知の拡張子をもつファイルの検索を試みないので、*file*は正確なファイル名拡張子をもたなければならないことに注意。

  `load`とは異なり、`module-load`は`load-history`にモジュールを記録せず、何もメッセージをプリントしないし、再帰ロードにたいする保護も行わない。したがってほとんどのユーザーは`module-load`のかわりに`load`、`load-file`、`load-library`、`require`を使用するべきである。

Emacs のロード可能モジュールは、configure 時にオプション*`--with-modules`*を使用することにより有効になります。

17 バイトコンパイル

Emacs Lisp には Lisp で記述された関数をより効率的に実行できる、バイトコード *(byte-code)* と呼ばれる特別な表現に翻訳するコンパイラー *(compiler)* があります。コンパイラーは Lisp の関数定義をバイトコードに置き換えます。バイトコード関数が呼び出されたとき、その定義はバイトコードインタープリター *(byte-code interpreter)* により評価されます。

バイトコンパイルされたコードは、(本当のコンパイル済みコードのように) そのマシンのハードウェアによって直接実行されるのではなく、バイトコンパイラーによって評価されるため、バイトコードはリコンパイルしなくてもマシン間での完全な可搬性を有します。しかし本当にコンパイルされたコードほど高速ではありません。

一般的に任意のバージョンの Emacs はそれ以前のバージョンの Emacs により生成されたバイトコンパイル済みコードを実行できますが、その逆は成り立ちません。

ある Lisp ファイルを常にコンパイルせずに実行したい場合は、以下のように `no-byte-compile` をバインドするファイルローカル変数を配置します:

```
;; -*-no-byte-compile: t; -*-
```

17.1 バイトコンパイル済みコードのパフォーマンス

バイトコンパイルされた関数は C で記述されたプリミティブ関数ほど効率的ではありませんが、Lisp で記述されたバージョンよりは高速に実行されます。以下は例です:

```
(defun silly-loop (n)
  "Return the time, in seconds, to run N iterations of a loop."
  (let ((t1 (float-time)))
    (while (> (setq n (1- n)) 0))
    (- (float-time) t1)))
⇒ silly-loop

(silly-loop 50000000)
⇒ 10.235304117202759

(byte-compile 'silly-loop)
⇒ [コンパイルされたコードは表示されない]

(silly-loop 50000000)
⇒ 3.705854892730713
```

この例ではインタープリターによる実行には 10 秒を要しますが、バイトコンパイルされたコードは 4 秒未満です。これは典型的な結果例ですが、実際の結果はさまざまでしょう。

17.2 バイトコンパイル関数

`byte-compile`により、関数やマクロを個別にバイトコンパイルできます。`byte-compile-file`でファイル全体、`byte-recompile-directory`または `batch-byte-compile`で複数ファイルをコンパイルできます。

バイトコンパイラーが警告および/またはエラーメッセージを生成することもあります (詳細は Section 17.6 [Compiler Errors], page 290 を参照)。これらのメッセージは通常は Compilation モードが使用する`*Compile-Log*`と呼ばれるバッファーに記録されます。Section “Compilation

Mode" in *The GNU Emacs Manual* を参照してください。しかし変数 `byte-compile-debug` が非 `nil`ならエラーメッセージは Lisp エラーとしてシグナルされます (Section 11.7.3 [Errors], page 164 を参照)。

バイトコンパイルを意図したファイル内にマクロ呼び出しを記述する際には注意が必要です。マクロ呼び出しはコンパイル時に展開されるので、そのマクロは Emacs にロードされる必要があります (さもないとバイトコンパイラーが正しく処理しないだろう)。これを処理する通常の方法は、必要なマクロ定義を含むファイルを `require`フォームで指定することです。バイトコンパイラーは通常はコンパイルするコードを評価しませんが、`require`フォームは指定されたライブラリーをロードすることにより特別に扱われます。誰かがコンパイルされたプログラムを実行する際にマクロ定義ファイルのロードを回避するためには、`require`呼び出しの周囲に `eval-when-compile`を記述します (Section 17.5 [Eval During Compile], page 289 を参照)。詳細は Section 14.3 [Compiling Macros], page 244 を参照してください。

インライン関数 (`defsubst`) はこれほど面倒ではありません。定義が判明する前にそのような関数呼び出しをコンパイルした場合でも、その呼び出しは低速になるだけで正しく機能するでしょう。

`byte-compile` *symbol* [Function]

この関数は *symbol*の関数定義をバイトコンパイルして、以前の定義をコンパイルされた定義に置き換える。*symbol*の関数定義は、その関数にたいする実際のコードでなければならない。`byte-compile`はインダイレクト関数を処理しない。リターン値は、*symbol*のコンパイルされた定義であるようなバイトコード関数ブジェクト (Section 17.7 [Byte-Code Objects], page 291 を参照)。

```
(defun factorial (integer)
  "INTEGER の階乗を計算する。"
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
⇒ factorial

(byte-compile 'factorial)
⇒
#[(integer)
  "^H\301U\203^H^@\301\207\302^H\303^HS!\"\207"
  [integer 1 * factorial]
  4 "Compute factorial of INTEGER."]
```

*symbol*の定義がバイトコード関数オブジェクトなら、`byte-compile`は何も行わず `nil`をリターンする。そのシンボルの関数セル内の (コンパイルされていない) オリジナルのコードはすでにバイトコンパイルされたコードに置き換えられているので、シンボルの定義の再コンパイルはしない。

`byte-compile`の引数として `lambda`式も指定できる。この場合、関数は対応するコンパイル済みコードをリターンするが、それはどこにも格納されない。

`compile-defun` **&optional** *arg* [Command]

このコマンドはポイントを含む defun を読み取りそれをコンパイルして結果を評価する。実際に関数定義であるような defun でこれを使用した場合には、その関数のコンパイル済みバージョンをインストールする効果がある。

`compile-defun`は通常は評価した結果をエコーエリアに表示するが、*arg*が非 `nil`ならフォームをコンパイルした後にカレントバッファーに結果を挿入する。

`byte-compile-file` *filename* **&optional** *load* [Command]

この関数は *filename*という名前の Lisp コードファイルを、バイトコードのファイルにコンパイルする。出力となるファイルの名前は、サフィックス‘`.el`’を‘`.elc`’に変更することにより作成される。*filename*が‘`.el`’で終了しない場合には、‘`.elc`’を *filename*の最後に付け足す。

コンパイルは入力ファイルから 1 つのフォームを逐次読み取ることにより機能する。フォームが関数かマクロなら、コンパイル済みの関数かマクロが書き込まれる。それ以外のフォームはまとめられて、まとめられたものごとにコンパイルされて、そのファイルが読まれたとき実行されるようにコンパイルされたコードが書き込まれる。入力ファイルを読み取る際には、すべてのコメントは無視される。

このコマンドはエラーがなければ`t`、それ以外は`nil`をリターンする。インタラクティブに呼び出されたときは、ファイル名の入力をもとめる。

*load*が非`nil`なら、このコマンドはコンパイルした後にコンパイルしたファイルをロードする。インタラクティブに呼び出された場合、*load*はプレフィクス引数である。

```
$ ls -l push*
-rw-r--r-- 1 lewis lewis 791 Oct  5 20:31 push.el

(byte-compile-file "~/emacs/push.el")
     ⇒ t

$ ls -l push*
-rw-r--r-- 1 lewis lewis 791 Oct  5 20:31 push.el
-rw-rw-rw- 1 lewis lewis 638 Oct  8 20:25 push.elc
```

`byte-recompile-directory` *directory* **&optional** *flag force* [Command]

このコマンドは *directory*(またはそのサブディレクトリー) 内の、リコンパイルを要するすべての‘`.el`’ファイルをリコンパイルする。‘`.elc`’ファイルが存在して、それが‘`.el`’より古いファイルは、リコンパイルが必要となる。

‘`.el`’ファイルに対応する‘`.elc`’ファイルが存在しない場合に何を行うかを *flag*で指定する。`nil`なら、このコマンドはこれらのファイルを無視する。*flag*が 0 なら、それらをコンパイルする。`nil`と 0 以外なら、それらのファイルをコンパイルするかユーザーに尋ねて、同様にそれぞれのサブディレクトリーについても尋ねる。

インタラクティブに呼び出されると、`byte-recompile-directory`は *directory*の入力を求めて、*flag*はプレフィクス引数となる。

*force*が非`nil`なら、このコマンドは‘`.elc`’ファイルが存在するすべての‘`.el`’ファイルをリコンパイルする。

リターン値は不定。

`batch-byte-compile` **&optional** *noforce* [Function]

この関数はコマンドラインで指定されたファイルにたいして`byte-compile-file`を実行する。この関数は処理が完了すると Emacs を kill するので、Emacs のバッチ実行でのみ使用しなければならない。1 つのファイルでエラーが発生しても、それによって後続のファイルにたいする処理が妨げられることはないが、そのファイルにたいする出力ファイルは生成されず、Emacs プロセスは 0 以外のステータスコードで終了する。

*noforce*が非`nil`なら、この関数は最新の‘`.elc`’ファイルがあるファイルをリコンパイルしない。

```
$ emacs -batch -f batch-byte-compile *.el
```

17.3 ドキュメント文字列とコンパイル

Emacs がバイトコンパイルされたファイルから関数や変数をロードする際、通常はメモリー内にそれらのドキュメント文字列をロードしません。それぞれのドキュメント文字列は、必要なときだけバイトコンパイルされたファイルからダイナミック (dynamic: 動的) にロードされます。ドキュメント文字列の処理をスキップすることにより、メモリーが節約されてロードが高速になります。

この機能には欠点があります。コンパイル済みのファイルを削除や移動、または (新しいバージョンのコンパイル等で) 変更した場合、Emacs は以前にロードした関数や変数のドキュメント文字列にアクセスできなくなるでしょう。このような問題は通常なら、あなた自身が Emacs をビルドしたときに、その Lisp ファイルを編集および/またはリコンパイルしたときだけ発生します。この問題は、リコンパイル後にそれぞれのファイルをリロードするだけで解決します。

バイトコンパイルされたファイルからのドキュメント文字列のダイナミックロードは、バイトコンパイルされたファイルごとにコンパイル時に解決されます。これはオプション `byte-compile-dynamic-docstrings`で無効にできます。

`byte-compile-dynamic-docstrings` [User Option]

これが非 `nil`なら、バイトコンパイラーはドキュメント文字列をダイナミックロードするようにセットアップしたコンパイル済みファイルを生成する。

特定のファイルでダイナミックロード機能を無効にするには、以下のようにヘッダー行でこのオプションに `nil`をセットする (Section “Local Variables in Files” in *The GNU Emacs Manual* を参照)。

```
-*-byte-compile-dynamic-docstrings: nil;-*-
```

これは主として、あるファイルを変更しようとしていて、そのファイルをすでにロード済みの Emacs セッションがファイルを変更した際にも正しく機能し続けることを望む場合に有用である。

内部的にはドキュメント文字列のダイナミックロードは、特殊な Lisp リーダー構文 ‘`#@`*count*’とともにコンパイル済みファイルに書き込むことによって達成される。この構文は次の *count*文字をスキップする。さらに ‘`#$`’構文も使用され、これはこのファイルの名前 (文字列) を意味する。これらの構文を Lisp ソースファイル内で使用しないこと。これらは人間がファイルを読む際に明確であるようにデザインされていない。

17.4 個々の関数のダイナミックロード

ファイルをコンパイルするとき、オプションでダイナミック関数ロード *(dynamic function loading)* 機能 (*laxy* ロード *(lazy loading)* とも呼ばれる) を有効にできます。ダイナミック関数ロードでは、ファイルのロードでファイル内の関数定義は完全には読み込まれません。かわりに各関数定義にはそのファイルを参照するプレースホルダーが含まれます。それぞれ関数が最初に呼び出されるときにそのプレースホルダーを置き換えるために、ファイルから完全な定義が読み込まれます。

ファイルのロードがより高速になるだろうというのがダイナミック関数ロードの利点です。ユーザーが呼び出せる関数を多く含むファイルにとって、それらの関数のうち 1 つを使用したら多分残りの関数も使用するというのでなければ、これは利点になります。多くのキーボードコマンドを提供する特化したモードは、このパターンの使い方をする場合があります。ユーザーはそのモードを呼び出すかもしれませんが、使用するのはそのモードが提供するコマンドのわずか一部です。

ダイナミックロード機能には不利な点がいくつかあります:

- ロード後にコンパイル済みファイルの削除や移動を行うと、Emacs はまだロードされていない残りの関数定義をロードできなくなる。

- (新しいバージョンのコンパイル等で) コンパイル済みファイルを変更した場合に、まだロードされていない関数のロードを試みると通常は無意味な結果となる。

このような問題は通常の状況でインストールされた Emacs ファイルでは決して発生しません。しかしあなたが変更した Lisp ファイルでは発生し得ます。それぞれのファイルをリコンパイルしたらすぐに新たなコンパイル済みファイルをリロードするのが、これらの問題を回避する一番簡単な方法です。

ダイナミックな関数ロードの使用により提供される利点がほとんど計測できないという経験から、この機能は *Emacs 27.1* 以降は廃止されます。

コンパイル時に変数 `byte-compile-dynamic`が非 `nil`なら、バイトコンパイラーはダイナミック関数ロード機能を使用します。ダイナミックロードが望ましいのは特定のファイルにたいしてだけなので、この変数をグローバルにセットしないでください。そのかわりに、特定のソースファイルのファイルローカル変数でこの機能を有効にしてください。たとえばソースファイルの最初の行に以下のテキストを記述することにより、これを行うことができます:

```
-*-byte-compile-dynamic: t;-*-
```

`byte-compile-dynamic` [Variable]
: これが非 `nil`なら、バイトコンパイラーはダイナミック関数ロード用にセットアップされたコンパイル済みファイルを生成する。

`fetch-bytecode` *function* [Function]
: *function*がバイトコード関数オブジェクトなら、それがまだ完全にロードされていなければ、バイトコンパイル済みのファイルからの *function*のバイトコードのロードを完了させる。それ以外なら何も行わない。この関数は常に *function*をリターンする。

17.5 コンパイル中の評価

これらの機能によりプログラムのコンパイル中に評価されるコードを記述できます。

`eval-and-compile` *body...* [Special Form]
: このフォームはそれを含むコードがコンパイルされるとき、および (コンパイルされているかいないかに関わらず) 実行されるときの両方で *body*が評価されるようにマークする。

*body*を別のファイルに配置して、そのファイルを `require`で参照すれば同様の結果が得られる。これは *body*が大きいときに望ましい方法である。事実上、`require`は自動的に `eval-and-compile`されて、そのパッケージはコンパイル時と実行時の両方でロードされる。

`autoload`も実際は `eval-and-compile`される。これはコンパイル時に認識されるので、そのような関数の使用により警告 “not known to be defined” は生成されない。

ほとんどの `eval-and-compile`の使用は、完全に妥当であると言えよう。

あるマクロがマクロの結果を構築するためのヘルパー関数をもち、そのマクロがそのパッケージにたいしてローカルと外部の両方で使用される場合には、コンパイル時と後の実行時にそのヘルパー関数を取得するために `eval-and-compile`を使用すること。

これは関数がプログラム的に (`fset`で) 定義されている場合には、コンパイル時と実行時にプログラム的な定義を行わせてそれらの関数の呼び出しをチェックするためにも使用できる (“not known to be defined” の警告は抑制される)。

`eval-when-compile` *body...* [Special Form]
: このフォームは *body*がコンパイル時に評価され、コンパイルされたプログラムがロードされるときは評価されないようにマークする。コンパイラーによる評価の結果はコンパイル済みの

プログラム内の定数となる。ソースファイルをコンパイルではなくロードすると、*body*は通常どおり評価される。

生成するために何らかの計算が必要な定数があるなら、`eval-when-compile`はコンパイル時にそれを行なうことができる。たとえば、

```
(defvar my-regexp
  (eval-when-compile (regexp-opt '("aaa" "aba" "abb"))))
```

他のパッケージを使用しているが、そのパッケージのマクロ (バイトコンパイラーはそれらを展開します) だけが必要なら、それらを実行せずにコンパイル用にロードさせるために `eval-when-compile`を使用できる。たとえば、

```
(eval-when-compile
  (require 'my-macro-package))
```

これらの事項は、マクロと `defsubst`関数がローカルに定義されていて、そのファイル内だけで使用されることを要求する。これらはそのファイルのコンパイルに必要だが、コンパイル済みファイルの実行には、ほとんどの場合必要ない。たとえば、

```
(eval-when-compile
  (unless (fboundp 'some-new-thing)
    (defmacro 'some-new-thing ()
      (compatibility code))))
```

これは大抵は他のバージョンの Emacs との互換性の保証のためのコードにたいしてのみ有用である。

Common Lisp に関する注意**:** トップレベルでは、`eval-when-compile`は Common Lisp のイディオム `(eval-when (compile eval) ...)`に類似する。トップレベル以外では、Common Lisp のリーダーマクロ '`#.`'(ただし解釈時を除く) が、`eval-when-compile`と近いことを行う。

17.6 コンパイラーのエラー

バイトコンパイルのエラーメッセージと警告メッセージは、`*Compile-Log*`という名前のバッファーにプリントされます。これらのメッセージには、問題となる箇所を示すファイル名と行番号が含まれます。これらのメッセージにたいして、コンパイラー出力を操作する通常の Emacs コマンドが使用できます。

あるエラーがプログラムのシンタックスに由来する場合、バイトコンパイラーはエラーの正確な位置の取得に際して混乱するかもしれません。バッファー `*Compiler Input*`. にスイッチするのは、これを調べ 1 つの方法です (このバッファー名はスペースで始まるので、Buffer Menu に表示されない)。このバッファーにはコンパイルされたプログラムと、バイトコンパイラーが読み取った箇所からポイントがどれほど離れているかが含まれ、エラーの原因はその近傍の可能性があります。シンタックスエラーを見つけるヒントについては、Section 18.3 [Syntax Errors], page 328 を参照してください。

定義されていない関数や変数の使用は、バイトコンパイラーにより報告される警告のタイプとしては一般的です。そのような警告では、定義されていない関数や変数を使用した位置ではなく、そのファイルの最後の行の行番号が報告されるので、それを見つけるには手作業で検索しなければなりません。

定義のない関数や変数の警告が間違いだと確信できる場合には、警告を抑制する方法がいくつかあります:

- 関数 *func*への特定の呼び出しにたいする警告は、それを条件式 `fboundp`でテストすることで抑制できる:

```
(if (fboundp 'func) ...(func ...)...)
```

*func*への呼び出しは`if`文の *then-form*内になければならず、*func*は`fboundp`呼び出し内でクォートされていなければならない (この機能は`cond`でも同様に機能する)。

- 同じように、変数 *variable*の特定の使用についの警告を、条件式内の `boundp`テストで抑制できる:

```
(if (boundp 'variable) ...variable...)
```

*variable*への参照は`if`文の *then-form*内になければならず、*variable*は `boundp`呼び出し内でクォートされていなければならない。

- コンパイラーに関数が `declare-function`を使用して定義されていると告げることができる。Section 13.15 [Declaring Functions], page 238 を参照のこと。
- 同じように変数が初期値なしの `defvar`を使用して定義されているとコンパイラーに告げることができる (カレントレキシカルスコープ、またはトップレベルにあればファイルでのみダイナミックにバインドされているとして変数を特別な変数としてマークすることに注意。Section 12.5 [Defining Variables], page 178 を参照のこと。

`with-suppressed-warnings`マクロを使用して特定の式にたいするコンパイラーの警告を抑制することもできます:

`with-suppressed-warnings` *warnings body...* [Special Form]
これは実行においては `(progn body...)`と等価だが、コンパイラーは *body*内の指定したコンディションにたいする警告を発しない。*warnings*は警告シンボルと、それらを適用する関数/変数シンボルの連想リスト。たとえば`foo`という時代遅れ (obsolete) の関数を呼び出したいがコンパイル時の警告を抑止したければ、以下のようにする:

```
(with-suppressed-warnings ((obsolete foo))
  (foo ...))
```

コンパイラー警告の抑制をより粗く行うには `with-no-warnings`構文を使用できます:

`with-no-warnings` *body...* [Special Form]
これは実行時には `(progn body...)`と等価だが、コンパイラーは *body*の中で起こるいかなる事項にたいしても警告を発しない。

わたしたちはかわりに `with-suppressed-warnings`の使用を推奨するが、この構文を使用する場合には、あなたが抑制したいと意図する警告以外の警告を失わないようにするために、可能な限り小さいコード断片にたいしてこの構文を使用すること。

変数 `byte-compile-warnings`をセットすることにより、コンパイラーの警告をより詳細に制御できます。詳細は変数のドキュメント文字列を参照してください。

`error`を使用してバイトコンパイラーの警告が報告されることを望む場合があるかもしれません。そのような場合には `byte-compile-error-on-warn`を非 `nil`値にセットしてください。

17.7 バイトコード関数オブジェクト

バイトコンパイルされた関数は、バイトコード関数オブジェクト *(byte-code function objects)* という特別なデータ型をもちます。関数呼び出しとしてそのようなオブジェクトが出現したとき、Emacsはそのバイトコードを実行するために、常にバイトコードインタープリターを使用します。

内部的にはバイトコード関数オブジェクトはベクターとよく似ています。バイトコード関数オブジェクトの要素には `aref`を通じてアクセスできます。バイトコード関数オブジェクトのプリント表現

(printed representation) はベクターと似ていて、開き '['の前に '#'が追加されます。バイト関数オブジェクトは少なくとも 4 つの要素をもたねばならず、その要素数に上限はありません。しかし通常使用されるのは最初の 6 要素です。これらは:

argdesc 引数の記述子 (descriptor)。これは Section 13.2.3 [Argument List], page 212 で説明されるような引数のリスト、または要求される引数の個数をエンコードする整数のいずれかである。後者の場合、その記述子の値は 0 ビットから 6 ビットで引数の最小個数、8 ビットから 14 ビットで引数の最大個数を指定する。引数リストが`&rest`を使用するなら 7 ビットがセットされて、それい以外ならクリアーされる。

*argdesc*がリストなら、そのバイトコード実行前に引数はダイナミックにバインドされる。*argdesc*が整数なら、引数リストはそのバイトコード実行前にバイトコーピンタープリンターのスタックに push される。

byte-code バイトコード命令を含む文字列。

constants バイトコードにより参照される Lisp オブジェクトのベクター。関数名と変数名に使用されるシンボルが含まれる。

stacksize この関数が要するスタックの最大サイズ。

docstring (もしあれば) ドキュメント文字列。それ以外は`nil`。ドキュメント文字列がファイルに格納されている場合、値は数字かリストかもしれない。本当のドキュメント文字列の取得には、関数`documentation`を使用する (Section 24.2 [Accessing Documentation], page 520 を参照)。

interactive
(もしあれば) インタラクティブ仕様。文字列か Lisp 式。インタラクティブでない関数では`nil`。

以下はバイトコード関数オブジェクトのプリント表現の例です。これはコマンド`backward-sexp`の定義です。

```
#[256
  "\211\204^G^@\300\262^A\301^A[!\207"
  [1 forward-sexp]
  3
  1793299
  "^p"]
```

バイトコードオブジェクトを作成するプリミティブな方法は`make-byte-code`です:

`make-byte-code` **&rest** *elements* [Function]
この関数は *elements*を要素とするバイトコードオブジェクトを構築してリターンする。

あなた自身で要素を収集してバイトコード関数を構築しないでください。それらが矛盾する場合、その関数の呼び出しにより Emacs がクラッシュするかもしれません。これらのオブジェクトの作成は常にバイトコンパイラーにまかせてください。(願わくば) バイトコンパイラーは要素を矛盾なく構築します。

17.8 逆アセンブルされたバイトコード

人はバイトコードを記述しません。それはバイトコンパイラーの仕事です。しかし好奇心を満たすために、わたしたちはディスアセンブラを提供しています。ディスアセンブラはバイトコードを人間が読めるフォームに変換します。

バイトコードインタープリターは、シンプルなスタックマシンとして実装されています。これは値を自身のスタックに push して、計算で使用するためにそれらを pop して取り出し、その結果を再びそのスタックに push して戻します。バイトコード関数がリターンするときは、スタックから値を pop して取り出し、その関数の値としてリターンします。

それに加えてスタックとバイトコード関数は、値を変数とスタック間で転送することにより、普通の Lisp 変数を使用したり、バインドやセットを行うことができます。

`disassemble` *object* **&optional** *buffer-or-name* [Command]

このコマンドは *object*にたいするディスアセンブルされたコードを表示する。インタラクティブに使用した場合、または *buffer-or-name*が `nil`か省略された場合は、`*Disassemble*`という名前のバッファーに出力します。*buffer-or-name*が非 `nil`なら、それはバッファーもしくは既存のバッファーの名前でなければならない。その場合は、そのバッファーのポイント位置に出力され、ポイントは出力の前に残りされる。

引数 *object*には関数名、ラムダ式 (Section 13.2 [Lambda Expressions], page 211 を参照)、またはバイトコードオブジェクト (Section 17.7 [Byte-Code Objects], page 291 を参照) を指定できる。ラムダ式なら `disassemble`はそれをコンパイルしてから、そのコンパイル済みコードをディスアセンブルする。

以下に `disassemble`関数を使用した例を 2 つ示します。バイトコードと Lisp ソースを関連付ける助けとなるように、説明的なコメントを追加してあります。これらのコメントは `disassemble`の出力にはありません。

```
(defun factorial (integer)
  "Compute factorial of an integer."
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
     ⇒ factorial

(factorial 4)
     ⇒ 24

(disassemble 'factorial)
     ⊣ byte-code for factorial:
 doc: Compute factorial of an integer.
 args: (integer)

0   varref   integer      ; integerの値を取得して
                          ;    それをスタック上に push する
1   constant 1            ; スタック上に 1 を push する
2   eqlsign               ; 2 つの値をスタックから pop して取り出し、
                          ;    それらを比較して結果をスタック上に push する
3   goto-if-nil 1         ; スタックのトップを pop してテストする
                          ;    nilなら 1 へ、それ以外は continue
6   constant 1            ; スタックのトップに 1 を push する
7   return                ; スタックのトップの要素をリターンする
```

```
8:1 varref   integer      ; integerの値をスタック上に push する
9   constant factorial    ; factorialをスタック上に push する
10  varref   integer      ; integerの値をスタック上に push する
11  sub1                  ; integerを pop して値をデクリメントする
                          ;   スタック上に新しい値を push する
12  call     1            ; スタックの最初 (トップ) の要素を引数として
                          ;   関数 factorialを呼び出す
                          ;   リターン値をスタック上に push する
13 mult                   ; スタックのトップ 2 要素を pop して取り出し乗じ
                          ;   結果をスタック上に push する
14 return                 ; スタックのトップ要素をリターンする
```

silly-loopは幾分複雑です:

```
(defun silly-loop (n)
  "Return time before and after N iterations of a loop."
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
     ⇒ silly-loop

(disassemble 'silly-loop)
     ⊣ byte-code for silly-loop:
 doc: Return time before and after N iterations of a loop.
 args: (n)

0   constant current-time-string  ; current-time-stringを
                                  ;    スタック上のトップに push する
1   call     0            ; 引数なしで current-time-stringを呼び出し
                          ;   結果をスタック上に push する
2   varbind  t1           ; スタックを pop して t1に pop された値をバインドする
3:1 varref   n            ; 環境から nの値を取得して
                          ;   その値をスタック上に push する
4   sub1                  ; スタックのトップから 1 を減ずる
5   dup                   ; スタックのトップを複製する
                          ;   たとえばスタックのトップをコピーしてスタック上に
push する
6   varset   n            ; スタックのトップを pop して
                          ;   nをその値にバインドする

;; (要はシーケンス dup varsetは pop せずに
;;  スタックのトップを nの値にコピーする)

7   constant 0            ; スタック上に 0 を push する
8   gtr                   ; スタックのトップ 2 値を pop して取り出し
                          ;   nが 0 より大かテストし
                          ;   結果をスタック上に push する
```

```
9   goto-if-not-nil 1      ; n > 0 なら 1 へ
                           ;    (これは while-loop を継続する)
                           ;    それ以外は continue
12  varref   t1            ; t1の値をスタック上に push する
13  constant current-time-string  ; current-time-stringを
                                  ;    スタックのトップに push する
14  call     0             ; 再度 current-time-stringを呼び出す
15  unbind   1             ; ローカル環境の t1をアンバインドする
16  list2                  ; スタックのトップ 2 要素を pop して取り出し
                           ;    それらのリストを作りスタック上に push する
17  return                 ; スタックのトップの値をリターンする
```

18 Lisp プログラムのデバッグ

Emacs Lisp プログラム内の問題を見つけて詳細に調べる方法がいくつかあります。

- プログラム実行中に問題が発生した場合には、Lisp 評価機能をサスペンドするためにビルトインの Emacs Lisp デバッガを使用して評価機能の内部状態の調査および/または変更を行なうことができる。
- Emacs Lisp にたいするソースレベルデバッガの Edebug を使用できる。
- `trace.el`パッケージが提供するトレース機能を使用して問題に関する関数の実行をトレースできます。このパッケージは関数呼び出しのトレース用に `trace-function-foreground`と`trace-function-background`、値をトレースする変数の追加用に `trace-values`という関数を提供する。詳細は `trace.el`にあるこれらの機能のドキュメントを参照のと。
- 文法的な問題により Lisp がプログラムを読むことさえできない場合には、Lisp 編集コマンドを使用して該当箇所を見つけることができる。
- バイトコンパイラーがプログラムをコンパイルするとき、コンパイラーにより生成されるエラーメッセージと警告メッセージを調べることができる。Section 17.6 [Compiler Errors], page 290 を参照のこと。
- Testcover パッケージを使用してプログラムのテストカバレッジを行なうことができる。
- ERT パッケージを使用してプログラムにたいするリグレッションテストを記述できる。*ERT: Emacs Lisp Regression Testing* を参照のこと。
- プログラムをプロファイルしてプログラムをより効果的にするためのヒントを取得できる。

入出力の問題をデバックする便利なその他のツールとして、ドリブルファイル (dribble file: Section 40.13 [Terminal Input], page 1118 を参照) と、`open-termscript`関数 (Section 40.14 [Terminal Output], page 1119) があります。

18.1 Lisp デバッガ

普通の *Lisp* デバッガは、フォーム評価のサスペンド機能を提供します。評価がサスペンド (一般的には *break* の状態として知られる) されている間、実行時スタックを調べたり、ローカル変数やグローバル変数の値を調べたり変更することができます。break は再帰編集 (recursive edit) なので、Emacs の通常の編集機能が利用可能です。デバッガにエンターするようにプログラムを実行することさえ可能です。Section 21.13 [Recursive Editing], page 417 を参照してください。

18.1.1 エラーによるデバッガへのエンター

デバッガに入るタイミングとして一番重要なのは、Lisp エラーが発生したときです。デバッガではエラーの直接原因を調査できます。

しかしデバッガへのエンターは、エラーによる通常の結末ではありません。多くのコマンドは不適切に呼び出されたときに Lisp エラーをシグナルするので、通常の編集の間にこれが発生するたびデバッガにエンターするのは、とても不便でしょう。したがってエラーの際にデバッガにエンターしたいなら、変数 `debug-on-error`に非 `nil`をセットします (コマンド `toggle-debug-on-error`はこれを簡単に行う方法を提供する)。

`debug-on-error` [User Option]

この変数はエラーがシグナルされて、それがハンドルされていないときにデバッガを呼び出すかどうかを決定する。`debug-on-error`が `t`なら、`debug-ignored-errors`(以下参照) にリ

ストされているエラー以外の、すべての種類のエラーがデバッガを呼び出す。nilならデバッガを呼び出さない。

値にはエラー条件 (Section 11.7.3.1 [Signaling Errors], page 165 を参照) のリストも指定できる。その場合はこのリスト内のエラー条件だけによってデバッガが呼び出される (debug-ignored-errorsにもリストされているエラー条件は除外される)。たとえばdebug-on-errorをリスト (void-variable)にセットすると、値をもたない変数に関するエラーにたいしてのみデバッガが呼び出される。

eval-expression-debug-on-errorがこの変数をオーバーライドするケースがいくつかあることに注意 (以下参照)。

この変数が非 nilのとき、Emacs はプロセスフィルター関数と番兵 (sentinel) の周囲にエラーハンドラーを作成しない。したがってこれらの関数内でのエラーは、デバッガを呼び出す。Chapter 38 [Processes], page 934 を参照のこと。

debug-ignored-errors [User Option]

この変数はdebug-on-errorの値に関わらず、デバッガにエンターすべきでないエラーを指定する。変数の値はエラー条件のシンボルおよび/または正規表現のリスト。エラーがこれら条件シンボルのいずれか、またはエラーメッセージが正規表現のいずれかにマッチすれば、そのエラーはデバッガにエンターしない。

この変数の通常の値には user-error、および編集中に頻繁に発生するが Lisp プログラムのバグに起因することは稀であるような、いくつかのエラーが含まれる。しかし “稀である” ことは “絶対ない” ということではない。あなたのプログラムがこのリストにマッチするエラーによって機能しないなら、そのエラーをデバッグするためにこのリストの変更を試みるのもよいだろう。通常は debug-ignored-errorsを nilにセットしておくのが、もっとも簡単な方法である。

eval-expression-debug-on-error [User Option]

この変数が非 nil値 (デフォルト) なら、コマンド eval-expressionの実行により一時的にdebug-on-errorが tがバインドされる。Section “Evaluating Emacs-Lisp Expressions” in *The GNU Emacs Manual* を参照のこと。

eval-expression-debug-on-errorが nilなら、eval-expressionの間もdebug-on-errorの値は変更されない。

debug-on-signal [User Option]

condition-caseでキャッチされたエラー、は通常は決してデバッガを呼び出さない。condition-caseはデバッガがそのエラーをハンドルする前にエラーをハンドルする機会を得る。

debug-on-signalを非 nil値に変更すると、condition-caseの存在如何に関わらずすべてのエラーにおいてデバッガが最初に機会を得る (デバッガを呼び出すためには依然としてそのエラーが debug-on-errorと debug-ignored-errorsで指定された条件を満たさなければならない)。

たとえば emacsclient の--evalオプションによるコードの評価からバックトレースを取得するためにはこの変数をセットすると便利。この変数が非 nilのときに emacsclient で評価された Lisp コードがエラーをシグナルすると、バックトレースは実行中の Emacs にポップアップされる。

警告: この変数を非 nilにセットすると、芳しくない効果があるかもしれない。Emacs のさまざまな部分で処理の通常の過程としてエラーがキャッチされており、そのエラーが発生したこ

とに気づかないことさえあるかもしれない。`condition-case`でラップされたコードをデバッグする必要があるなら、`condition-case-unless-debug`(see Section 11.7.3.3 [Handling Errors], page 167 を参照) の使用を考慮されたい。

`debug-on-event` [User Option]

`debug-on-event`をスペシャルイベント (Section 21.9 [Special Events], page 412 を参照) にセットすると、Emacs は `special-event-map`をバイパスしてこのイベントを受け取ると即座にデバッガへのエンターを試みる。現在のところサポートされる値は、シグナル `SIGUSR1` と `SIGUSR2`に対応する値のみ (これがデフォルト)。これは `inhibit-quit`がセットされていて、それ以外は Emacs が応答しない場合に有用かもしれない。

`debug-on-message` [Variable]

`debug-on-message`に正規表現をセットした場合は、それにマッチするメッセージがエコーエリアに表示されると、Emacs はデバッガにエンターする。たとえばこれは特定のメッセージの原因を探すのに有用かもしれない。

init ファイルロード中に発生したエラーをデバッグするには、オプション '`--debug-init`'を使用する。これは init ファイルロードの間に `debug-on-error`を `t`にバインドして、通常は init ファイル内のエラーをキャッチする `condition-case`をバイパスする。

18.1.2 無限ループのデバッグ

プログラムが無限にループしてリターンできないとき、最初の問題はそのループをいかに停止するかです。ほとんどのオペレーティングシステムでは、(*quit* させる)*C-g*でこれを行うことができます。Section 21.11 [Quitting], page 413 を参照してください。

普通の quit では、なぜそのプログラムがループしたかについての情報は与えられません。変数 `debug-on-quit`に非 `nil`をセットすることにより、より多くの情報を得ることができます。無限ループの途中でデバッガを実行すれば、デバッガからステップコマンドで先へ進むことができます。ループ全体をステップで追えば、問題を解決するために十分な情報が得られるでしょう。

*C-g*による quit はエラーとは判断されないので、*C-g*のハンドルに `debug-on-error`は効果がありません。同じように `debug-on-quit`はエラーにたいして効果がありません。

`debug-on-quit` [User Option]

この変数は `quit`がシグナルされて、それがハンドルされていないときにデバッガを呼び出すかどうかを決定する。`debug-on-quit`が非 `nil`なら、quit(つまり *C-g*をタイプ) したときは常にデバッガが呼び出される。`debug-on-quit`が `nil`(デフォルト) なら、quit してもデバッガは呼び出されない。

18.1.3 関数呼び出しによるデバッガへのエンター

プログラムの途中で発生する問題を調べるための有用なテクニックの 1 つは、特定の関数が呼び出されたときデバッガにエンターする方法です。問題が発生した関数にこれを行ってその関数をステップで追ったり、問題箇所の少し手前の関数呼び出しでこれを行って、その関数をステップオーバーしてその後をステップで追うことができます。

`debug-on-entry` *function-name* [Command]

この関数は *function-name*が呼び出されるたびにデバッガの呼び出しを要求する。

Lisp コードで定義された任意の関数とマクロは、インタープリターに解釈されたコードかコンパイル済みのコードかに関わらず、エントリーに break をセットできる。その関数がコマンドなら Lisp から呼び出されたときと、インタラクティブに呼び出されたときにデバッガにエ

ンターする。(たとえば C で記述された) プリミティブ関数にもこの方法で `debug-on-entry` をセットできるが、そのプリミティブが Lisp コードから呼び出されたときだけ効果がある。`debug-on-entry`はスペシャルフォームにはセットできない。

`debug-on-entry`がインタラクティブに呼び出されたときは、ミニバッファーで *function-name* の入力を求める。その関数がすでにエントリーでデバッガを呼び出すようにセットアップされていたら、`debug-on-entry`は何も行わない。`debug-on-entry`は常に *function-name*をリターンする。

以下はこの関数の使い方を説明するための例である:

```
(defun fact (n)
  (if (zerop n) 1
      (* n (fact (1- n)))))
     ⇒ fact
(debug-on-entry 'fact)
     ⇒ fact
(fact 3)

------ Buffer: *Backtrace* ------
Debugger entered--entering a function:
* fact(3)
  eval((fact 3))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
------ Buffer: *Backtrace* ------
```

`cancel-debug-on-entry` **&optional** *function-name* [Command]

この関数は *function-name*にたいする `debug-on-entry`の効果をアンドゥする。インタラクティブに呼び出されたときは、ミニバッファーで *function-name*の入力を求める。*function-name*が省略または `nil`なら、すべての関数にたいする break-on-entry をキャンセルする。エントリー時に break するようセットアップされていない関数に `cancel-debug-on-entry`を呼び出したときは何も行わない。

18.1.4 変数の変更時にデバッガにエンターする。

不正な変数のセッティングが関数に問題をもたらすときがあります。元のセッティングを調べるためには、変数の変更時に常にデバッガがトリガーされるようにセットアップするのが手軽な方法です。

`debug-on-variable-change` *variable* [Command]

この関数は *variable*の変更時に常にデバッガが呼び出されるようにアレンジする。

これは watchpoint メカニズムを使用して実装されているので同じような特徴と制限を継承する。つまり *variable*のすべてのエイリアスは一緒に watch されて、watch 対象はダイナミック変数のみであり変数から参照されるオブジェクトの変更は検出されない。詳細は Section 12.9 [Watching Variables], page 184 を参照のこと。

`cancel-debug-on-variable-change` **&optional** *variable* [Command]

この関数は *variable*にたいする `debug-on-variable-change`の効果をアンドゥする。インタラクティブに呼び出されたときはミニバッファーで *variable*の入力をもとめる。*variable*が省略

か nilならすべての変数にたいする変更時のブレークを取り消す。カレントで変更時にブレークするようにセットアップされていない変数にたいして cancel-debug-on-variable-change の呼び出しは何も行わない。

18.1.5 明示的なデバッガへのエントリー

プログラム内の特定箇所に式 (debug)を記述することによって、その箇所でデバッガが呼び出されるようにできます。これを行うにはソースファイルを visit して、適切な箇所にテキスト '(debug)'を挿入し、*C-M-x*(Lisp モードでの eval-defunにたいするキーバインド) をタイプします。警告: 一時的なデバッグ目的のためにこれを行なう場合には、ファイルを保存する前に確実にアンドゥしてください!

'(debug)'を挿入する箇所は追加フォームが評価されることができ、かつその値を無視することができる箇所でなければなりません ('(debug)'の値を無視しないとプログラムの実行が変更されてしまうだろう!)。一般的にもっとも適した箇所は、prognまたは暗黙的な progn(Section 11.1 [Sequencing], page 145 を参照) の内部です。

デバッグ命令を配置したいソースコード中の正確な箇所がわからないが、特定のメッセージが表示されたときにバックトレースを表示したい場合には、意図するメッセージにマッチする正規表現を debug-on-messageにセットできます。

18.1.6 デバッガの使用

デバッガにエンターすると、その前に選択されていたウィンドウを 1 つのウィンドウに表示して、他のウィンドウに*Backtrace*という名前のバッファーを表示します。backtrace バッファーには、現在実行されている Lisp 関数の各レベルが 1 行ずつ含まれます。このバッファーの先頭は、デバッガが呼び出された理由を説明するメッセージ (デバッガがエラーにより呼び出された場合はエラーメッセージや関連するデータなど) です。

backtrace バッファーは読み取り専用で、文字キーにデバッガコマンドが定義された Debugger モードという特別なメジャーモードを使用します。通常の Emacs 編集コマンドが利用できます。したがってエラー時に編集されていたバッファーを調べるためにウィンドウを切り替えたり、バッファーの切り替えやファイルの visit、その他一連の編集処理を行なうことができます。しかしデバッガは再帰編集レベル (Section 21.13 [Recursive Editing], page 417 を参照) にあり、編集が終わったらそれは backtrace バッファーに戻って、(*q*コマンドで) デバッガを exit できます。デバッガを exit することによって再帰編集を抜け出し、backtrace バッファーはバリー (bury: 覆い隠す) されます (変数 debugger-bury-or-killwをセットすることによって backtrace バッファーで *q*コマンドが何を行うかをカスタマイズできる。たとえばバッファーをバリーせずに kill したいなら、この変数を killにセットする。他の値については変数のドキュメントを調べてほしい)。

デバッガにエンターしたとき、eval-expression-debug-on-errorに一致するように変数 debug-on-errorが一時的にセットされます。変数 eval-expression-debug-on-errorが非 nilなら、debug-on-errorは一時的に tにセットされます。これはデバッグセッション行っている間にさらにエラーが発生すると、(デフォルトでは) 他の backtrace がトリガーされることを意味します。これが望ましくなければ、debugger-mode-hook内で eval-expression-debug-on-error を nilにセットするか、debug-on-errorを nilにセットすることができます。

デバッガは Lisp インタープリターの状態について想定を行うので、バイトコンパイルされて実行されなければなりません。デバッガがインタープリターに解釈されて実行されているときは、これらの想定は正しくなくなります。

18.1.7 バックトレース

Debugger モードは Edebug と ERT(Section 18.2 [Edebug], page 306 と *ERT: Emacs Lisp Regression Testing* を参照) が backtrace 表示にも使用する Backtrace モードから派生したモードです。

backtrace バッファーは実行されている関数と、その関数の引数の値を示します。backtrace バッファー作成時には 1 つのスタックフレームにたいして 1 行 (非常に長い可能性がある) を表示します (スタックフレームとは Lisp インタープリターが特定の関数呼び出しに関する情報を記録する場所のこと)。もっとも最近の呼び出しが最上行になります。

backtrace ではフレームを記述する行にポイントを移動してフタックフレームを指定できます。ポイントのある行のフレームがカレントフレーム *(current frame)* とみなされます。

関数名に下線が引かれている場合には、Emacs がソースコードの場所を知っていることを意味します。その関数名をマウスでクリックするか、そこに移動して `RET`をタイプすることにより、ソースコードを visit できます。下線のない関数名や変数名にポイントがあるときに `RET`をタイプした場合には、もしあればそのシンボルにたいするヘルプ情報を help バッファーで確認することもできます。`M-.`にバインドされている `xref-find-definitions`コマンドは backtrace 内の任意の識別子にも使用できます (Section “Looking Up Identifiers” in *The GNU Emacs Manual* を参照)。

backtrace では長いリストの末尾およびベクターや構造、同じように深くネストされたオブジェクトの終端は下線つきの “...” でプリントされます。“...” 上でマウスをクリックするか、その上にポイントがある状態で `RET`をタイプすることにより、オブジェクトの隠蔽されている部分を表示できます。省略を行う量の制御は `backtrace-line-length`をカスタマイズしてください。

以下は backtrace の操作と閲覧を行うコマンドのリストです:

v カレントスタックフレームのローカル変数の表示を切り替える。

p フレームまたは前のフレームの先頭に移動する。

n 次のフレームの先頭に移動する。

`+` 可読性向上のためにポイント位置に行ブレークを追加してトップレベル Lisp フォームにインデントする。

`-` ポイント位置のトップレベル Lisp フォームを 1 行に折り畳む。

`#` ポイント位置のフレームの `print-circle`を切り替える。

: ポイント位置のフレームの `print-gensym`を切り替える。

`.` ポイント位置のフレームの “...” で省略されたすべてのフォームを展開する。

18.1.8 デバッガのコマンド

(Debugger モードの)debugger バッファーでは通常の Emacs コマンドに加えて特別なコマンド、および前セクションで説明した Backtrace モードのコマンドが提供されます。デバッガでもっとも重要な使い方をするのは、制御フローを見ることができるコードをステップ実行するコマンドです。デバッガはインタープリターによって解釈された制御構造のステップ実行はできますが、バイトコンパイル済みの関数ではできません。バイトコンパイル済み関数をステップ実行したいなら、同じ関数の解釈された定義に置き換えてください (これを行なうにはその関数のソースを visit して、関数の定義で *C-M-x*とタイプする)。プリミティブ関数のステップ実行に Lisp デバッガは使用できません。

デバッガのコマンドのいくつかはカレントフレームを処理します。フレームが星印で開始される場合には、そのフレームを exit することにより再びデバッガが呼び出されることを意味します。これは関数のリターン値の検証に有用です。

以下は Debugger モードのコマンドのリストです:

c デバッガを exit して実行を継続する。これはあたかもデバッガにエンターしなかったかのようにプログラムの実行を再開する (デバッガ内で行った変数値やデータ構造の変更などの副作用は除外)。

d 実行を継続するが、次に Lisp 関数が何か呼び出されたときはデバッガにエンターする。これによりある式の下位の式をステップ実行して、下位の式が計算する値や行うことを確認できる。

デバッガにエンターした関数呼び出しにたいして、この方法で作成されたスタックフレームには自動的にフラグがつくため、そのフレームを exit すると再びデバッガが呼び出される。このフラグは *u*コマンドを使用してキャンセルできる。

b カレントフレームにフラグをつけるので、そのフレームを exit するときデバッガにエンターする。この方法でフラグがつけられたフレームは、backtrace バッファーでスターのマークがつく。

u カレントフレームを exit したときデバッガにエンターしない。これはそのフレームの *b* コマンドをキャンセルする。目に見える効果としては backtrace バッファーの行からスターが削除される。

j *b*と同じようにカレントフレームにフラグをつける。その後に *c*のように実行を継続するが、`debug-on-entry`によりセットアップされたすべての関数にたいする break-on-entry を一時的に無効にする。

e ミニバッファーの Lisp 式を読み取り、(関連する lexical 環境が適切なら) それを評価してエコーエリアに値をプリントする。デバッガは特定の重要な変数とバッファーを処理の一部として変更する。*e*は一時的にデバッガの外部からそれらの値をリストアするので、それらを調べて変更できる。これによりデバッガはより透過的になる。対照的にデバッガ内で *M-:*は特別なことを行わず、デバッガ内での変数の値を表示する。

R *e*と同様だがバッファー`*Debugger-record*`内の評価結果も保存する。

q デバッグされているプログラムを終了して、Emacs コマンド実行のトップレベルにリターンする。

*C-g*によりデバッガにエンターしたが、実際はデバッグではなく quit したいときは *q*コマンドを使用する。

r デバッガから値をリターンする。ミニバッファーで式を読み取ってそれを評価することにより値が計算される。

*d*コマンドは、(*b*によるリクエストや *d*によるそのフレームへのエンターによる)Lisp 呼び出しフレームからの exit でデバッガが呼び出されたときに有用である。*r*コマンドで指定された値は、そのフレームの値として使用される。これは `debug`を呼び出して、そのリターン値を使用するときにも有用。それ以外は *r*は *c*と同じ効果をもち、指定されたリターン値は問題とはならない。

エラーによりデバッガにエンターしたときは *r*コマンドは使用できない。

l 呼び出されたときにデバッガを呼び出す関数をリストする。これは `debug-on-entry`によりエントリー時に break するようセットされた関数のリストである。

18.1.9 デバッガの呼び出し

以下ではデバッガを呼び出すために使用される関数 debugの完全な詳細を説明します。

debug **&rest** *debugger-args* [Command]

この関数はデバッガにエンターする。この関数は*Backtrace*(デバッガへの 2 回目以降の再帰エントリーでは*Backtrace*<2>、...) という名前のバッファーにバッファーを切り替えて、Lisp 関数呼び出しについての情報を書き込む。それから再帰編集にエンターして、Debugger モードで backtrace バッファーを表示する。

Debugger モードのコマンド *c*、*d*、*j*、*r*は再帰編集を exit する。その後、debugは以前のバッファーに戻って、debugを呼び出したものが何であれそこにリターンする。これは関数 debug が呼び出し元にリターンできる唯一の方法である。

*debugger-args*を使用すると、debugは*Backtrace*の最上部に残りの引数を表示するしてユーザーがそれらを確認できる。以下で説明する場合を除いて、これはこれらの引数を使用する唯一の方法である。

しかし debugへの 1 つ目の引数にたいする値は、特別な意味をもつ (これらの値は通常は debug を呼び出すプログラマーではなく、Emacs 内部でのみ使用される)。以下はこれら特別な値のテーブルである:

`lambda` 1 つ目の引数が lambdaのなら、それは debug-on-next-callが非 nilのときに関数にエントリーしたことによって debugが呼び出されたことを意味する。デバッガはバッファーのトップのテキスト行に 'Debugger entered--entering a function:'と表示する。

`debug` 1 つ目の引数がdebugなら、それはエントリー時にデバッグされるようにセットされた関数にエントリーしたことにって debugが呼び出されたことを意味する。デバッガは lambdaのときと同様、'Debugger entered--entering a function:'を表示する。これはその関数のスタックフレームもマークするので、exit 時にデバッガが呼び出される。

`t` 1 つ目の引数が tなら、それは debug-on-next-callが非 nilのときに関数呼び出しの評価によって debugが呼び出されたことを示す。デバッガはバッファーのトップの行に 'Debugger entered--beginning evaluation of function call form:'と表示する。

`exit` 1 つ目の引数が exitのときは、exit 時にデバッガを呼び出すよう以前にマークされたスタックフレームを exit したことを示す。この場合は debugに与えられた 2 つ目の引数がそのフレームからリターンされた値になる。デバッガはバッファーのトップの行に 'Debugger entered--returning value:'とリターンされた値を表示する。

`error` 1 つ目の引数が errorのときは、ハンドルされていないエラーまたは quitがシグナルされてデバッガにエンターした場合であり、デバッガは 'Debugger entered--Lisp error:'とその後にシグナルされたエラーと signalへの引数を表示してそれを示す。たとえば、

```
(let ((debug-on-error t))
  (/ 1 0))
```

```
------ Buffer: *Backtrace* ------
Debugger entered--Lisp error: (arith-error)
  /(1 0)
...
------ Buffer: *Backtrace* ------
```

エラーがシグナルされた場合はおそらく変数debug-on-errorは非nilで、quitがシグナルされた場合はおそらく変数debug-on-quitは非nilである。

nil 明示的にデバッガにエンターしたいときは、*debugger-args*の 1 つ目の引数にnilを使用する。残りの*debugger-args*はバッファーのトップの行にプリントされる。メッセージ— たとえばdebugが呼び出された条件を思い出すためのリマインダーとして — の表示にこの機能を使用できる。

18.1.10 デバッガの内部

このセクションではデバッガ内部で使用される関数と変数について説明します。

debugger [Variable]

この関数の値はデバッガを呼び出す関数呼び出しである。値には任意個数の引数をとる関数、より具体的には関数の名前でなければならない。この関数は何らかのデバッガを呼び出すこと。この変数のデフォルト値はdebug。

関数にたいして Lisp が渡す 1 つ目の引数は、その関数がなぜ呼び出されたかを示す。引数の慣習についてはdebug(Section 18.1.9 [Invoking the Debugger], page 303) に詳解がある。

backtrace [Function]

この関数は現在アクティブな Lisp 関数呼び出しのトレースをプリントする。このトレースはdebugが*Backtrace*バッファーで表示するものと等しい。リターン値は常にnil。

以下の例では Lisp 式で明示的にbacktraceを呼び出している。これはストリームstandard-output(この場合はバッファー ‘backtrace-output’) に backtrace をプリントする。

backtrace の各行は 1 つの関数呼び出しを表す。関数の引数が既知なら行に関数とその後に値が表示される。まだ計算中なら関数と未評価の引数を含むリストから行が構成される。長いリストや深くネストされた構造は省略されるかもしれない。

```
(with-output-to-temp-buffer "backtrace-output"
  (let ((var 1))
    (save-excursion
      (setq var (eval '(progn
                        (1+ var)
                        (list 'testing (backtrace)))))))

    ⇒ (testing nil)

----------- Buffer: backtrace-output ------------
  backtrace()
  (list 'testing (backtrace))
  (progn ...)
  eval((progn (1+ var) (list 'testing (backtrace))))
  (setq ...)
  (save-excursion ...)
  (let ...)
  (with-output-to-temp-buffer ...)
  eval((with-output-to-temp-buffer ...))
```

```
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: backtrace-output ------------
```

`debugger-stack-frame-as-list` [User Option]

この変数が非`nil`ならバックトレースのすべてのスタックフレームはリストとして表示される。これは通常の関数呼び出しとバックトレースの特殊形式の視覚的な違いによるコストをなくして、バックトレースの可読性を向上することが目的。

`debugger-stack-frame-as-list`が非`nil`なら上記の例は以下のようになる:

```
---------- Buffer: backtrace-output ------------
  (backtrace)
  (list 'testing (backtrace))
  (progn ...)
  (eval (progn (1+ var) (list 'testing (backtrace))))
  (setq ...)
  (save-excursion ...)
  (let ...)
  (with-output-to-temp-buffer ...)
  (eval (with-output-to-temp-buffer ...))
  (eval-last-sexp-1 nil)
  (eval-last-sexp nil)
  (call-interactively eval-last-sexp)
---------- Buffer: backtrace-output ------------
```

`debug-on-next-call` [Variable]

この変数が非`nil`なら、それは次の`eval`、`apply`、`funcall`の前にデバッガを呼び出すよう指定する。デバッガへのエンターによって`debug-on-next-call`は`nil`にセットされる。

デバッガの *d*コマンドは、この変数をセットすることにより機能します。

`backtrace-debug` *level flag* [Function]

この関数はそのスタックフレームの *level*下位のスタックフレームの debug-on-exit フラグに *flag*に応じた値をセットする。*flag*が非`nil`なら、後でそのフレームを exit するときデバッガにエンターする。そのフレームを通じた非ローカル exit でも、デバッガにエンターする。

この関数はデバッガだけに使用される。

`command-debug-status` [Variable]

この変数はカレントのインタラクティブコマンドのデバッグ状態を記録する。コマンドがインタラクティブに呼び出されるたびに、この変数は`nil`にバインドされる。デバッガは同じコマンドが呼び出されたときのデバッガ呼び出しに情報を残すために、この変数をセットできる。

普通のグローバル変数ではなくこの変数を使用する利点は、そのデータが後続のコマンド呼び出しに決して引き継がれないことである。

この変数は時代遅れであり将来のバージョンで削除されるだろう。

`backtrace-frame` *frame-number* **&optional** *base* [Function]

関数`backtrace-frame`は Lisp デバッガ内での使用を意図している。これは *frame-number* レベル下位のスタックフレームで何の評価が行われているかに関する情報をリターンする。

そのフレームがまだ引数を評価していない、またはそのフレームがスペシャルフォームなら値は`(nil` *function arg-forms...*`)`。

そのフレームが引数を評価して関数をすでに呼び出していたらリターン値は`(t` *function arg-values...*`)`。

リターン値の *function*は何であれ評価されたリストの CAR として提供される。マクロ呼び出しなら `lambda`式。その関数に`&rest`引数があればリスト *arg-values*の末尾に示される。

*base*を指定すると *frame-number*は *function*が *base*であるようなフレームの上端から相対的に数えられる。

*frame-number*が範囲外なら `backtrace-frame`は `nil`をリターンする。

`mapbacktrace` *function* **&optional** *base* [Function]

関数 `mapbacktrace`はバックトレース上の関数が *base*であるようなフレーム (*base*が省略か `nil`なら先頭) から順に各フレームにたいして一度 *function*を呼び出す。

*function*は *evald*、*func*、*args*、*flags*という 4 つの引数で呼び出される。

そのフレームがまだ引数を評価していない、またはそのフレームがスペシャルフォームなら *evald*は `nil`、*args*はフォームのリスト。

フレームが引数を評価して関数を呼び出し済みなら *evald*は `t`、*args*は値リスト。*flags*はカレントフレームのプロパティの plist。サポートされるプロパティは現在のところ`:debug-on-exit`のみであり、そのスタックフレームの `debug-on-exit`フラグがセットされていれば `t`。

18.2 Edebug

Edebug は Emacs Lisp プログラムにたいするソースレベルデバッガです。これにより以下のことができます:

- 式の前後でストップして評価をステップで実行する。
- 条件付きまたは無条件の breakpoint のセット。
- 指定された条件が true ならストップする (グローバル breakpoint)。
- ストップポイントごとに停止したり、breakpoint ごとに簡単に停止して低速または高速にトレースを行う。
- Edebug 外部であるかのように式の結果を表示して、式を評価する。
- 式のリストを自動的に再評価して、Edebug がディスプレイを更新するたびにそれらの結果を表示する。
- 関数呼び出しとリターンのトレース情報を出力する。
- エラー発生時にストップする。
- Edebug 自身のフレームを除外して backtrace を表示する。
- マクロとフォームの定義で引数の評価を指定する。
- 初歩的なカバレッジテストと頻度数の取得。

以下の初めの 3 つのセクションは、Edebug の使用を開始するために十分な説明を行います。

18.2.1 Edebug の使用

Edebug で Lisp プログラムをデバッグするには、最初にデバッグしたい Lisp コードをインストルメント *(instrument:* 計装*)* しなければなりません。これを行なうもっともシンプルな方法は、関数またはマクロの定義に移動して *C-u C-M-x*(プレフィクス引数を指定した `eval-defun`) を行います。コードをインストルメントする他の手段については、Section 18.2.2 [Instrumenting], page 307 を参照してください。

一度関数をインストルメントすると、その関数にたいする任意の呼び出しによって Edebug がアクティブになります。Edebug がアクティブになると、どの Edebug 実行モードを選択したかに依存し

て、その関数をステップ実行できるように実行がストップされるか、ディスプレイを更新してデバッグコマンドにたいするチェックの間、実行が継続されます。デフォルトの実行モード step で、これは実行をストップします。Section 18.2.3 [Edebug Execution Modes], page 308 を参照してください。

Edebug では通常は、デバッグしている Lisp コードを Emacs バッファーで閲覧します。これをソースコードバッファー *(source code buffer)* と呼び、バッファーは一時的に読み取り専用になります。

左フリンジの矢印は、その関数で実行されている行を示します。ポイントは最初はその関数の実行されている行にありますが、ポイントを移動するとこれは真ではなくなります。

以下は `fac`の定義 (以下を参照) をインストルメントして `(fac 3)`を実行した場合に通常目にするものです。ポイントは `if`の前の開きカッコにあります。

```
(defun fac (n)
=>*(if (< 0 n)
       (* n (fac (1- n)))
     1))
```

関数内で Edebug が実行をストップできる位置のことを、ストップポイント *(stop points)* と呼びます。ストップポイントはリストであるような部分式の前後、および変数参照の後でも発生します。以下は関数 `fac`内のストップポイントをピリオドで示したものです:

```
(defun fac (n)
  .(if .(< 0 n.).
       .(* n. .(fac .(1- n.).).).
     1).)
```

Emacs Lisp モードのコマンドに加えて、ソースコードバッファーでは Edebug のスペシャルコマンドが利用できます。たとえば Edebug コマンド `SPC`で次のストップポイントまで実行することができます。`fac`にエントリーした後に一度 `SPC`とタイプした場合は、以下のように表示されるでしょう:

```
(defun fac (n)
=>(if *(< 0 n)
       (* n (fac (1- n)))
     1))
```

式の後で Edebug が実行をストップしたときは、エコーエリアにその式の値が表示されます。

他にも頻繁に使用されるコマンドとして、ストップポイントに breakpoint をセットする *b*、breakpoint に達するまで実行する *g*、Edebug を exit してトップレベルのコマンドループにリターンする *q*があります。また *?*とタイプするとすべての Edebug コマンドがリストされます。

18.2.2 Edebug のためのインストルメント

Lisp コードのデバッグに Edebug を使用するためには、最初にそのコードをインストルメント *(instrument:* 計装*)* しなければなりません。コードをインストルメントすると、適切な位置で Edebug を呼び出すために追加コードが挿入されます。

関数定義でプレフィクス引数とともにコマンド *C-M-x* (`eval-defun`) を呼び出すと、それを評価する前にその定義をインストルメントします (ソースコード自体は変更しない)。変数 `edebug-all-defs`が非 `nil`ならプレフィクス引数の意味を反転します。この場合は、*C-M-x*はプレフィクス引数がなければその定義をインストルメントします。`edebug-all-defs`のデフォルト値は `nil`です。コマンド *M-x edebug-all-defs*は変数 `edebug-all-defs`の値を切り替えます。

`edebug-all-defs`が非 `nil`なら `eval-region`、`eval-current-buffer`、`eval-buffer`もそれらが評価する定義をインストルメントします。同様に `edebug-all-forms`は、`eval-region`が (非

定義フォームさえ含むあらゆるフォームをインストルメントするべきかを制御します。これはミニバッファー内でのロードや評価には適用されません。コマンド *M-x edebug-all-forms*はこのオプションを切り替えます。

他にもコマンド *M-x edebug-eval-top-level-form*が利用でき、これは edebug-all-defs や edebug-all-formsの値に関わらずトップレベルのすべてのフォームをインストルメントします。edebug-defunは edebug-eval-top-level-formのエイリアスです。

Edebug がアクティブのの間、コマンド *I*(edebug-instrument-callee) はポイント後のリストフォームに呼び出される関数およびマクロ定義がまだインストルメントされていなければ、それらをインストルメントします。これはそのファイルのソースの場所を Edebug が知っている場合だけ可能です。この理由により Edebug ロード後は、たとえ評価する定義をインストルメントしない場合でも、eval-regionは評価するすべての定義の位置を記録します。インストルメント済み関数呼び出しにステップインする *i*コマンドも参照してください (Section 18.2.4 [Jumping], page 310 を参照)。

Edebug はすべての標準スペシャルフォーム、式引数をもつ interactiveフォーム、無名ラムダ式、およびその他の定義フォームのインストルメント方法を知っています。しかし Edebug はユーザー定義マクロが引数にたいして何を行うかを判断できないので、Edebug 仕様を使用してその情報を与えなければなりません。詳細は Section 18.2.15 [Edebug and Macros], page 320 を参照してください。

Edebug がセッション内で最初にコードをインストルメントしようとするときは、フック edebug-setup-hookを実行してからそれに nilをセットします。使おうとしているパッケージに結びつけて Edebug 仕様をロードするためにこれを使用できますが、それは Edebug を使用するときだけ機能します。

Edebug がインストルメント中にシンタックスエラー (syntax error: 構文エラー) を検知した場合には、以下のように間違ったコードの箇所にポイントを残して invalid-read-syntaxエラーをシグナルします:

```
error  Invalid read syntax: "Expected lambda expression"
```

このようにインストルメントに失敗する可能性の 1 つとして、いくつかのマクロ定義を Emacs が把握していない場合があります。これに対処するためには、インストルメントしようとしている関数を定義しているファイルをロードしてください。

定義からインストルメントを削除するには、単にインストルメントを行わない方法でその定義を再評価するだけです。フォームを絶対にインストルメントせずに評価するには 2 つの方法があります。それはファイルからの loadによる評価と、ミニバッファーからの eval-expression(*M-:*) による評価です。

定義からインストルメントを削除する別の方法は edebug-remove-instrumentationコマンドの使用です。これはインストルメントしたすべてからインストルメントを削除することもできます。

Edebug 内で利用可能な他の評価関数については、Section 18.2.9 [Edebug Eval], page 314 を参照してください。

18.2.3 Edebug の実行モード

Edebug はデバッグするプログラムの実行にたいして、いくつかの実行モードをサポートします。これらの実行モードを *Edebug* 実行モード *(Edebug execution modes)* と呼びます。これらをメジャーモードやマイナーモードと混同しないでください。カレントの Edebug 実行モードは、プログラムをストップする前に Edebug がどれだけ実行を継続するか— たとえばストップポイントごとにストップ、あるいは次の breakpoint まで継続など — 、およびストップする前に Edebug がどれだけ進捗を表示するかを決定します。

Edebug 実行モードは、通常はある特定のモードでプログラムを継続させるコマンドをタイプすることによって指定します。以下はそれらのコマンドのテーブルです。*S*以外のコマンドはプログラムの実行を再開して、少なくともある長さの間だけは実行を継続します。

S Stop(ストップ): これ以上プログラムを実行しないで Edebug のコマンドを待つ (`edebug-stop`)。

SPC Step(ステップ): 次のストップポイントでストップする (`edebug-step-mode`)。

n Next(次へ): 式の後にある次のストップポイントでストップする (`edebug-next-mode`)。 Section 18.2.4 [Jumping], page 310 の `edebug-forward-sexp`も参照のこと。

t Trace(トレース): Edebug のストップポイントごとに pause(通常は 1 秒) する (`edebug-trace-mode`)。

T Rapid trace(高速でトレース):ストップポイントごとに表示を更新するが、実際に pause はしない (`edebug-Trace-fast-mode`)。

g Go(進む): 次の breakpoint まで実行する (`edebug-go-mode`)。Section 18.2.6.1 [Breakpoints], page 312 を参照のこと。

c Continue(継続): breakpoint ごとに pause してから継続する (`edebug-continue-mode`)。

C Rapid continue(高速で継続): ポイントを各 breakpoint へ移動するが pause しない (`edebug-Continue-fast-mode`)。

G Go non-stop(ストップせず進む): breakpoint を無視する (`edebug-Go-nonstop-mode`)。まだ *S*やその他の編集コマンドでプログラムをストップするのは可能。

一般的に上記リストの最初のほうにある実行モードは後のほうの実行モードに比べて、プログラムをより低速に実行するか、すぐにストップさせます。

新たな Edebug レベルにエンターしたとき、Edebug は通常は最初に遭遇したインストルメント済みの関数でストップするでしょう。breakpoint でのみストップするか、(たとえばカバレッジデータ収集時など) ストップさせないようにするには、`edebug-initial-mode`の値をデフォルトの `step`から `go`か `Go-nonstop`、あるいはその他の値に変更してください (Section 18.2.16 [Edebug Options], page 325 を参照)。*C-x C-a C-m* (`edebug-set-initial-mode`) でこれを容易に行うことができます:

`edebug-set-initial-mode` [Command]
*C-x C-a C-m*にバインドされるこのコマンドは `edebug-initial-mode`をセットする。これはモードを示すキーの入力を求める。対応するモードをセットする上述 8 つのキーのいずれかを入力すること。

たとえば 1 つのコマンドからインストルメント済みの関数が複数回呼び出されたら、同じ Edebug レベルに再エンターするかもしれないことに注意してください。

実行中とトレース中は、任意の Edebug コマンドをタイプすることによって実行をインタラプト (interrupt: 中断、割り込み) できます。Edebug は次のストップポイントでプログラムをストップしてからタイプされたコマンドを実行します。たとえば実行中に *t*をタイプすると、次のストップポイントでトレースモードに切り替えます。*S*を使用すれば他に何も行わずに実行をストップできます。

関数でたまたま読み取り入力が発生した場合には、実行のインタラプトを意図してタイプされた文字は、かわりにその関数により読み取られます。そのプログラムが入力を欲するタイミングに注意を払うことで、そのような意図せぬ結果を避けることができます。

このセクションのコマンドを含むキーボードマクロは、完全には機能しません。プログラムを再開するために Edebug から exit すると、キーボードマクロの追跡記録は失われます。これに対処するのは簡単ではありません。また Edebug 外部でキーボードマクロを定義または実行しても、Edebug 内部のコマンドに影響しません。通常これは利点です。Section 18.2.16 [Edebug Options], page 325 内の `edebug-continue-kbd-macro`オプションも参照してください。

`edebug-sit-for-seconds` [User Option]

このオプションは trace モードと continue モードで実行ステップの間を何秒待つか指定する。デフォルトは 1 秒。

18.2.4 ジャンプ

このセクションで説明するコマンドは、指定された場所に達するまで実行を続けます。`i`を除くすべてのコマンドは、ストップ場所を確立するために一時的な breakpoint を作成してから go モードにスイッチします。意図されたストップポイントの前にある他のストップポイントに達した場合にも実行はストップします。breakpoint の詳細は、Section 18.2.6.1 [Breakpoints], page 312 を参照してください。

以下のコマンドでは、非ローカル exit はプログラムのストップを望む一時的な breakpoint をバイパスできるので、期待どおり機能しないかもしれません。

`h` ポイントがある場所の近くのストップポイントへ実行を進める (`edebug-goto-here`)。

`f` プログラムの式を 1 つ実行する (`edebug-forward-sexp`)。

`o` sexp を含む終端までプログラムを実行する (`edebug-step-out`)。

`i` ポイントの後のフォームから呼び出された関数かマクロにステップインする (`edebug-step-in`)。

`h`コマンドは一時的な breakpoint を使用してポイントのカレント位置、またはその後のストップポイントまで処理を進めます。

`f`コマンドは式を 1 つ飛び越してプログラムを実行します。より正確には`forward-sexp`により到達できる位置に一時的な breakpoint をセットしてから go モードで実行するので、プログラムはその breakpoint でストップすることになります。

プレフィクス引数 *n*とともに使用すると、ポイントから *n*個の sexp(s-expression: S 式) を超えた場所に一時的な breakpoint をセットします。ポイントを含むリストが *n*より少ない要素で終わるような場合には、ストップ箇所はポイントが含まれる式の後になります。

`forward-sexp`が見つける位置が、プログラムを実際にストップさせたい位置なのかチェックしなければなりません。たとえば `cond`内ではこれは正しくないかもしれません。

`f`コマンドは柔軟性を与えるために、`forward-sexp`をストップポイントではなくポイント位置から開始します。カレントのストップポイントから 1 つの式を実行したい場合には、まずそこにポイントを移動するために `w`(`edebug-where`) をタイプして、それから `f`をタイプしてください。

`o`コマンドは、式の外側で実行を継続します。これはポイントを含む式の最後に一時的な breakpoint を配置します。ポイントを含む sexp が関数定義なら `o`はその定義内の最後の sexp の直前まで実行を継続します。もし定義内の最後の sexp の直前にポイントがある場合は、その関数からリターンしてからストップします。言い換えるとこのコマンドは最後の sexp の後にポイントがなければ、カレントで実行中の関数から exit しません。

コマンド `h`、`f`、`o`は通常は “Break” を表示して、正に評価した結果を表示する前に `edebug-sit-for-seconds`の間、一時停止します。`edebug-sit-on-break`を `nil`にセットすることによりこの一時停止を回避できます。Section 18.2.16 [Edebug Options], page 325 を参照してください。

*i*コマンドは、ポイントの後のリストフォームに呼び出された関数やマクロにステップインします。そのフォームは評価されようとしているものの 1 つである必要はないことに注意してください。しかしそのフォームが評価されようとしている関数呼び出しなら、引数が何も評価されないうちにこのコマンドを使用しないと、遅すぎることを覚えておいてください。

*i*コマンドはステップインしようとしている関数やマクロがまだインストルメントされていなければ、それらをインストルメントします。これは便利かもしれませんが、それらを明示的に非インストルメントしなければ、その関数やマクロはインストルメントされたままになることを覚えておいてください。

18.2.5 その他の Edebug コマンド

ここではその他の Edebug コマンドを説明します。

? Edebug のヘルプメッセージを表示する (`edebug-help`)。

C-] 1 レベルを中断して以前のコマンドレベルへ戻る (`abort-recursive-edit`)。

q エディターのトップレベルのコマンドループにリターンする (`top-level`)。これはすべてのレベルの Edebug アクティビティを含むすべての再帰編集レベルを exit する。しかしフォーム `unwind-protect`か `condition-case`で保護されたインストルメント済みのコードはデバッグを再開するかもしれない。

Q *q*と同様だが、保護されたコードでもストップしない (`edebug-top-level-nonstop`)。

r エコーエリアにもっとも最近の既知のコマンドを再表示する (`edebug-previous-result`)。

d backtrace を表示するが、明確であるように Edebug 自身の関数は除外される (`edebug-backtrace`)。

backtrace と backtrace に作用するコマンドの説明は Section 18.1.7 [Backtraces], page 301 を参照のこと。

backtrace 内で Edebug の関数を確認したければ *M-x edebug-backtrace-show-instrumentation*、それらを再び隠すには *M-x edebug-backtrace-hide-instrumentation*を使用する。

backtrace のフレームの先頭が '`>`'なら、そのフレームにたいするソースコードの場所を Edebug が知っていることを意味する。カレントフレームのソースコードにジャンプするには *s*を使用する。

実行を継続したときに backtrace バッファーは自動的に kill される。

Edebug から再帰的に Edebug をアクティブにするコマンドを呼び出すことができます。Edebug がアクティブなときは常に *q*によトップレベルの終了、または *C-]*による再帰編集 1 レベルの中断ができます。*d*によってすべての未解決な評価の backtrace を表示できます。

18.2.6 ブレーク

Edebug の step モードは、次のストップポイントに達したときに実行をストップします。一度開始された Edebug の実行をストップするには、他に 3 つの方法があります。それは breakpoint、グローバル break 条件、およびソース breakpoint です。

18.2.6.1 Edebug のブレークポイント

Edebug を使用しているときは、テスト中のプログラム内に *breakpoint* を指定できます。breakpoint とは実行がストップされる場所のことです。Section 18.2.1 [Using Edebug], page 306 で定義されている任意のストップポイントに breakpoint をセットできます。breakpoint のセットと解除で影響を受けるストップポイントは、ソースコードバッファー内でポイント位置、またはポイント位置の後の最初のストップポイントです。以下は Edebug の breakpoint 用のコマンドです:

`b` ポイント位置、またはポイント位置の後のストップポイントに breakpoint をセットする (`edebug-set-breakpoint`)。プレフィクス引数を使用すると、それは一時的な breakpoint となり、プログラムが最初にそこで停止したときに解除される。breakpoint 位置には `edebug-enabled-breakpoint`または `edebug-disabled-breakpoint`フェイスのオーバーレイが配置される。

`u` (もしあれば) ポイント位置、またはポイント位置の後のストップポイントにある breakpoint を解除 (unset) する (`edebug-unset-breakpoint`)。

`U` カレントフォーム内のすべての breakpoint にポイントのセットを解除する (`edebug-unset-breakpoints`)。

`D` ポイント付近の breakpoint の有効と無効を切り替える (`edebug-toggle-disable-breakpoint`)。このコマンドは主に breakpoint が条件つきであり、そのコンディションの再作成に幾分かの作業を要する場合に有用。

x condition RET
*condition*を評価して非`nil`値になる場合だけプログラムをストップする条件付き breakpoint をセットする (`edebug-set-conditional-breakpoint`)。プレフィクス引数を指定すると一時的な breakpoint になる。

`B` カレント定義内の次の breakpoint にポイントを移動する (`edebug-next-breakpoint`)。

Edebug 内では `b`で breakpoint をセットして、`u`でそれを解除できます。最初に望ましいストップポイントにポイントを移動してから、そこに breakpoint をセットまたは解除するために `b`または `u`をタイプします。breakpoint がない場所で breakpoint を解除しても影響はありません。

ある定義の再評価や再インストルメントを行うと、以前の breakpoint はすべて削除されます。

条件付き *breakpoint(conditional breakpoint)* は、プログラムがそこに達するたびに条件をテストします。条件を評価した結果エラーが発生した場合、エラーは無視されて結果は `nil`になります。条件付き breakpoint をセットするには `x`を使用して、ミニバッファーで条件式を指定します。以前にセットされた条件付き breakpoint があるストップポイントに条件付き breakpoint をセットすると、以前の条件式がミニバッファーに配置されるのでそれを編集できます。

プレフィクス引数を指定して breakpoint をセットするコマンドを使用することによって、一時的な条件付き breakpoint、および無条件の breakpoint を作成できます。一時的な breakpoint によりプログラムがストップしたとき、その breakpoint は自動的に解除されます。

Go-nonstop モードを除き、Edebug は常に breakpoint でストップ、または pause します。Go-nonstop モードでは breakpoint は完全に無視されます。

breakpoint がどこにあるか探すには `B`コマンドを使用します。このコマンドは同じ関数内からポイント以降にある次の breakpoint(ポイント以降に breakpoint が存在しなければ最初の breakpoint) にポイントを移動します。このコマンドは実行を継続せずに、単にバッファー内のポイントを移動します。

18.2.6.2 グローバルなブレーク条件

グローバル *break* 条件 *(global break condition)* は指定された条件が満たされたとき、それがどこで発生したかによらず、実行をストップします。Edebug は、すべてのストップポイントでグローバル break 条件を評価します。これが非 `nil`値に評価された場合は、あたかもそのストップポイントに breakpoint があったかのように、実行をストップまたは pause します (実行モードによる)。条件の評価でエラーを取得した場合は、実行をストップしません。

条件式は `edebug-global-break-condition`に格納されます。Edebug がアクティブなときにソースバッファーから *X*コマンドを使用するか、Edebug がロードされている間は任意のバッファーから任意のタイミングで *C-x X X*(`edebug-set-global-break-condition`) を使用して新たな式を指定できます。

グローバル break 条件は、コード内のどこでイベントが発生したかを見つけるもっともシンプルな方法ですが、コードの実行は遅くなります。そのため使用しないときは条件を `nil`にリセットするべきです。

18.2.6.3 ソースブレークポイント

定義内のすべての breakpoint は、それをインストルメントするたびに失われます。breakpoint が失われないようにしたければソースコード内で単に関数 `edebug`を呼び出すソース *breakpoint(source breakpoint)* を記述できます。もちろんそのような呼び出しを条件付きすることにもできます。たとえば `fac`関数内に以下のような行を 1 行目に挿入して、引数が 0 になったときストップさせることができます:

```
(defun fac (n)
  (if (= n 0) (edebug))
  (if (< 0 n)
      (* n (fac (1- n)))
    1))
```

`fac`の定義がインストルメントされて呼び出されたとき、`edebug`呼び出しは breakpoint として振る舞います。実行モードに応じて Edebug はそこでストップまたは pause します。

`edebug`が呼び出されたときにインストルメント済みのコードが実行されていなければ、この関数は `debug`を呼び出します。

18.2.7 エラーのトラップ

エラーがシグナルされて、それが `condition-case`でハンドルされていないとき、Emacs は通常はエラーメッセージを表示します。Edebug がアクティブでインストルメント済みコードの実行中は、ハンドルされていないエラーには通常は Edebug が対応します。オプション `edebug-on-error`と `edebug-on-quit`でこれをカスタマイズできます。Section 18.2.16 [Edebug Options], page 325 を参照してください。

Edebug がエラーに対応するときは、エラー発生箇所の前にある最後のストップポイントを表示します。この場所はインストルメントされていない関数の呼び出しであったり、その関数内で実際にエラーが発生したのかもしれません。バインドされていない変数に関するエラーの場合は、最後の既知のストップポイントは、その不正な変数参照から遠く離れた場所にあるかもしれません。そのような場合には完全な backtrace を表示したいと思うでしょう (Section 18.2.5 [Edebug Misc], page 311 を参照)。

Edebug がアクティブの間に `debug-on-error`か `debug-on-quit`を変更すると、それらの変更は Edebug が非アクティブになったとき失われます。さらに Edebug の再帰編集の間、これらの変数は Edebug の外部でもっていた値にバインドされます。

18.2.8 Edebug のビュー

これらの Edebug コマンドは、Edebug にエントリーする前のバッファーの外観とウィンドウの状態を調べるコマンドです。外部のウィンドウ構成はウィンドウのコレクションとその内容であり、それらは実際には Edebug の外部にあります。

v 外部のウィンドウ構成ビューに切り替える (`edebug-view-outside`)。Edebug にリターンするには *C-x X w*をタイプする。

p 一時的に外部のカレントバッファーを表示して、ポイントもその外部の位置になる (`edebug-bounce-point`)。Edebug にリターンする前に 1 秒 pause する。プレフィクス引数 *n*を指定すると、かわりに *n*秒 pause する。

w ソースコードバッファー内のカレントストップポイントにポイントを戻す (`edebug-where`)。

このコマンドを同じバッファーを表示する異なるウィンドウで使用すると、そのウィンドウは将来カレント定義を表示するために代用される。

W Edebug が外部のウィンドウ構成の保存とリストアを行うかどうかを切り替える (`edebug-toggle-save-windows`)。

プレフィクス引数を指定すると、*W*は選択されたウィンドウの保存とリストアだけを切り替える。ソースコードバッファーを表示していないウィンドウを指定するには、グローバルキーマップから *C-x X W*を使用しなければならない。

v、または単に *p*でカレントバッファーにポイントを反跳させれば、たとえ通常は表示されないウィンドウでも外部のウィンドウ構成を調べることができます。

ポイントを移動した後にストップポイントに戻りたいときがあるかもしれません。これはソースコードバッファーから *w*で行うことができます。どのバッファーにいても *C-x X w*を使用すれば、ソースコードバッファー内のストップポイントに戻ることができます。

保存をオフにするために *W*を使用するたびに、Edebug は外部のウィンドウ構成を忘れます。そのためたとえ保存をオンに戻しても、(プログラムを実行することによって) 次に Edebug を exit したとき、カレントウィンドウ構成は変更されないまま残ります。しかし十分な数のウィンドウをオープンしていない場合には、`*edebug*`と`*edebug-trace*`の再表示があなたが見たいバッファーと競合するかもしれません。

18.2.9 評価

Edebug 内では、まるで Edebug が実行されていないかのように式を評価できます。式の評価とプリントに際して、Edebug は不可視になるよう試みます。副作用をもつ式の評価は、Edebug が明示的に保存とリストアを行うデータへの変更を除いて期待したとおり機能するでしょう。このプロセスの詳細は、Section 18.2.14 [The Outside Context], page 318 を参照してください。

e exp RET Edebug のコンテキスト外で式 *exp*を評価する (`edebug-eval-expression`)。つまり、Edebug はその式への干渉を最小限にしようと努める。

M-: exp RET

Edebug 自身のコンテキスト内で式 *exp*を評価する (`eval-expression`)。

C-x C-e Edebug のコンテキスト外でポイントの前の式を評価する (`edebug-eval-last-sexp`)。プレフィクス引数が 0(*C-u 0 C-x C-e*) なら、(文字列やリストのような) 長いアイテムを短縮しない。

Edebug は `cl.el`内の構文 (`lexical-let`、`macrolet`、`symbol-macrolet`) によって作成された、レキシカル (lexical) にバインドされたシンボルへの参照を含む式の評価をサポートします。

18.2.10 評価 List Buffer

式をインタラクティブに評価するために、*edebug*と呼ばれる評価リストバッファー *(evaluation list buffer)* を使用できます。Edebug がディスプレイを更新するたびに自動的に評価される、式の評価リスト *(evaluation list)* もセットアップできます。

E 評価リストバッファー*edebug*に切り替える (edebug-visit-eval-list)。

*edebug*バッファーでは、以下の特別なコマンドと同様に Lisp Interaction モード (Section "Lisp Interaction" in *The GNU Emacs Manual* を参照) のコマンドも使用できます。

C-j ポイントの前の式をコンテキスト外で評価して、その値をバッファーに挿入する (edebug-eval-print-last-sexp)。プレフィクス引数が 0(*C-u 0 C-j*) なら、(文字列やリストのような) 長いアイテムを短縮しない。

C-x C-e Edebug のコンテキスト外でポイントの前の式を評価する (edebug-eval-last-sexp)。

C-c C-u バッファー内のコンテンツから新たに評価リストを構築する (edebug-update-eval-list)。

C-c C-d ポイントのある評価リストグループを削除する (edebug-delete-eval-item)。

C-c C-w ソースコードバッファーに切り替えてカレントストップポイントに戻る (edebug-where)。

評価リストウィンドウ内では、*scratch*にいるときと同様に *C-j*や *C-x C-e*で式を評価できますが、それらは Edebug のコンテキスト外で評価されます。

インタラクティブに入力した式 (と結果) は、実行を継続すると失われます。しかし実行がストップされるたびに評価されるように、式から構成される評価リストをセットアップできます。

これを行なうには、評価リストバッファー内で 1 つ以上の評価リストグループ *(evaluation list group)* を記述します。評価リストグループは 1 つ以上の Lisp 式から構成されます。グループはコメント行で区切られます。

コマンド *C-c C-u*(edebug-update-eval-list) はバッファーをスキャンして、各グループの最初の式を使用して評価リストを再構築します (これはグループの 2 つ目の式は以前に計算、表示されている値だという発想からである)。

Edebug にエントリーするたびに、評価リストの各式 (および式の後に式のカレント値) をバッファーに挿入して再表示します。これはコメント行も挿入するので、各式はそのグループの一員となります。したがってバッファーのテキストを変更せずに *C-c C-u*とタイプすると、評価リストは実際には変更されません。

評価リストからの評価の間にエラーが発生すると、それが式の結果であるかのようにエラーメッセージが文字列で表示されます。したがってカレントで無効な変数を使用する式によって、デバッグが中断されることはありません。

以下はいくつかの式を評価リストウィンドウに追加したとき、どのように見えるかの例です:

```
(current-buffer)
#<buffer *scratch*>
;----------------------------------------------------------------
(selected-window)
#<window 16 on *scratch*>
;----------------------------------------------------------------
(point)
196
;----------------------------------------------------------------
```

```
bad-var
"Symbol's value as variable is void: bad-var"
;;-------------------------------------------------------------
(recursion-depth)
0
;;-------------------------------------------------------------
this-command
eval-last-sexp
;;-------------------------------------------------------------
```

グループを削除するにはグループ内にポイントを移動して *C-c C-d*をタイプするか、単にグループのテキストを削除して *C-c C-u*で評価リストを更新します。評価リストに新たな式を追加するには、適切な箇所にその式を挿入して新たなコメント行を挿入してから *C-c C-u*をタイプします。コメント行にダッシュを挿入する必要はありません — 内容は関係ないのです。

`*edebug*`を選択した後に *C-c C-w*でソースコードバッファーにリターンできます。`*edebug*`は実行を継続したときに kill されて、次回必要となったときに再作成されます。

18.2.11 Edebug でのプリント

プログラム内の式が循環リスト構造 (circular list structure) を含む値を生成する場合は、Edebug がそれをプリントしようとしたときエラーとなるかもしれません。

循環構造への対処の 1 つとして、`print-length`と `print-level`にプリントの切り詰めをセットする方法があります。Edebug は変数 `edebug-print-length`と `edebug-print-level`の値 (非 `nil`値なら) を、これらの変数にバインドします。Section 19.6 [Output Variables], page 339 を参照してください。

`edebug-print-length` [User Option]
: 非 `nil`なら結果をプリントするとき Edebug は `print-length`をこの値にバインドする。デフォルト値は 50。

`edebug-print-level` [User Option]
: 非 `nil`なら結果をプリントするとき Edebug は `print-level`をこの値にバインドする。デフォルト値は 50。

`print-circle`を非 `nil`値にバインドして、循環構造や要素を共有する構造にたいして、より参考になる情報をプリントするよういにすることもできます。

以下は循環構造を作成するコードの例です:

```
(setq a (list 'x 'y))
(setcar a a)
```

カスタムプリントはこれを '`Result: #1=(#1# y)`'のようにプリントします。'`#1=`'という表記はその後の構造をラベル '`1`'とラベル付けして、'`#1#`'表記はその前にラベル付けされた構造を参照しています。この表記はリストとベクターの任意の共有要素に使用されます。

`edebug-print-circle` [User Option]
: 非 `nil`なら結果をプリントするとき Edebug は `print-circle`をこの値にバインドする。デフォルト値は `t`。

他のプログラムもカスタムプリントを使用できます。詳細は `cust-print.el`を参照してください。

18.2.12 トレースバッファー

Edebug は実行トレースを`*edebug-trace*`という名前のバッファーに格納して記録できます。実行トレースとは関数呼び出しとリターンのログのことで関数名と引数、および値を確認できます。トレースレコードを有効にするには、`edebug-trace`を非`nil`値にセットしてください。

トレースバッファーの作成は実行モードのトレースの使用 (Section 18.2.3 [Edebug Execution Modes], page 308 を参照) と同じではありません。

トレースレコードが有効なときは、関数へのエントリーと exit のたびにトレースバッファーに行が追加されます。関数エントリーレコードは‘`::::{`’、および関数名と引数の値によって構成されます。関数の exit レコードは‘`::::}`’、および関数名と関数の結果によって構成されます。

‘`:`’の数は関数エントリーの再帰レベルを表します。トレースバッファーでは関数呼び出しの開始と終了の検索に‘`{`’と‘`}`’を使用できます。

関数`edebug-print-trace-before`と`edebug-print-trace-after`を再定義することによって、関数エントリーと関数 exit のトレースレコードをカスタマイズできます。

`edebug-tracing` *string body...* [Macro]
このマクロは *body*フォームの実行活動にたいして追加のトレース情報をリクエストする。引数 *string*はトレースバッファーに配置する‘`{`’と‘`}`’の後のテキストを指定する。すべての引数は評価されて、`edebug-tracing`は *body*内の最後のフォームの値をリターンする。

`edebug-trace` *format-string* **&rest** *format-args* [Function]
この関数はトレースバッファーにテキストを挿入する。テキストは`(apply 'format` *format-string format-args*`)`によって計算される。エントリー間の区切りとして改行も付け加える。

`edebug-tracing`と`edebug-trace`は、たとえ Edebug が非アクティブでも、呼び出されたときは常にトレースバッファーに行を挿入します。トレースバッファーへのテキストの追加により、挿入された最後の行が見えるようにウィンドウもスクロールします。

18.2.13 カバレッジテスト

Edebug は基本的なカバレッジテスト (coverage test) と実行頻度 (execution frequency) の表示を提供します。

カバレッジテストは、すべての式の結果と以前の結果を比較することによって機能します。プログラム内の各フォームがカレント Emacs セッション内でカバレッジテストを開始して以降に、2 つの異なる値をリターンしたら、それらのフォームはカバーされたと判断されます。したがってプログラムにカバレッジテストを行なうには、そのプログラムをさまざまなコンディション下で実行して、プログラムが正しく振る舞うかに注目します。異なるコンディション下で十分にテストして、すべてのフォームが異なる 2 つの値をリターンしたとき、Edebug はそのことを告げるでしょう。

カバレッジテストにより実行速度が低下するので、`edebug-test-coverage`が非`nil`のときだけカバレッジテストが行なわれます。頻度計数 (frequency count) はたとえ実行モードが Go-nonstop でも、カバレッジテストが有効か無効かに関わらずすべての式にたいして行われます。

定義にたいするカバレッジ情報と頻度数の両方を表示するには *C-x X =* (`edebug-display-freq-count`) を使用します。単に`=` (`edebug-temp-display-freq-count`) とすると、他のキーをタイプするまでの間だけ一時的に同様の情報を表示します。

`edebug-display-freq-count` [Command]
このコマンドはカレント定義の各行の頻度数を表示する。

このコマンドはコードの各行の下にコメント行として頻度数を挿入する。1 回の undoコマンドですべての挿入をアンドゥできる。頻度数は式の前の '('か式の後の ')'、または変数の最後の文字の下に表示される。表示をシンプルにするために同一行にたいして式の以前頻度数と頻度数が同じ場合は表示しない。

ある式にたいする頻度数の後に文字 '='がある場合は、その式が評価されるたびに同じ値を毎回リターンしていることを表す。言い換えるとカバレッジテストの目的からは、その式はまだカバーされていないということである。

ある定義にたいして頻度数とカバレッジデータを明確にするには、単に eval-defunで再インストルメントすればよい。

たとえばソースの breakpoint で (fac 5)を評価した後に edebug-test-coverageを tにセットすると、breakpoint に達したときの頻度データは以下のようになります:

```
(defun fac (n)
  (if (= n 0) (edebug))
;#6           1      = =5
  (if (< 0 n)
;#5         =
      (* n (fac (1- n)))
;#    5               0
    1))
;#   0
```

コメント行は facが 6 回呼び出されたことを表しています。最初の if命令は毎回同じ結果を 5 回リターンしています。同じ結果という意味では 2 つ目の ifの条件にも当てはまります。facの再帰呼び出しは結局リターンしません。

18.2.14 コンテキスト外部

Edebug はデバッグ中のプログラムにたいして透過的であろうと努めますが完全には達成されません。Edebug は eや評価リストバッファーで式を評価するときにも、一時的に外部のコンテキストをリストアして透明化を試みます。このセクションでは Edebug がリストアするコンテキストと、Edebug が完全に透過的になるのに失敗する理由を正確に説明します。

18.2.14.1 停止するかどうかのチェック

Edebug にエンターするときは常に特定のデータの保存とリストアを行なう必要があり、それはトレース情報を作成するか、あるいはプログラムを停止するかを決定する前に行なう必要があります。

- max-lisp-eval-depth (Section 10.4 [Eval], page 140 を参照)、および max-specpdl-size (Section 12.3 [Local Variables], page 175 を参照) は Edebug がスタックに与える影響の低減効果を高める。しかしそれでも Edebug 使用時にスタック空間を使い切ってしまうことがあり得る。非常に大きいクォートされたリストを含むコードをインストルメントすることによって Edebug が再帰深さの制限に達してしまうようなら、edebug-max-depthの値を大きくすることもできる。
- キーボードマクロの実行状態の保存とリストアが行われる。Edebug がアクティブの間、edebug-continue-kbd-macroが nilなら executing-kbd-macroが nilにバインドされる。

18.2.14.2 Edebug の表示の更新

(たとえば trace モードなどで)Edebug が何かを表示する必要があるときは、Edebug の外部からカレントウィンドウ構成 (Section 28.26 [Window Configurations], page 686 を参照) を保存します。Edebug を exit するときに、以前のウィンドウ構成がリストアされます。

Emacs は pause 時だけ再表示を行います。通常は実行を継続すると、そのプログラムは breakpoint かステップ実行後に Edebug に再エンターして、その間に pause や入力の読み取りはありません。そのような場合、Emacs が外部の構成を再表示する機会は決してありません。結果としてユーザーが目にするウィンドウ構成は、前回 Edebug が中断なしでアクティブだったときのウィンドウ構成と同じになります。

何かを表示するために Edebug にエントリーすることにより、(たとえこれらのうちのいくつかは、エラーや quit がシグナルされたときは故意にリストアしないデータだとしても) 以下のデータも保存とリストアが行われます。

- カレントバッファー、およびカレントバッファー内のポイントとマークの位置が保存およびリストアされる。
- `edebug-save-windows`が非`nil`なら、外部のウィンドウ構成の保存とリストアが行われる (Section 18.2.16 [Edebug Options], page 325 を参照)。

  エラーや quit ではウィンドウ構成はリストアされないが、`save-excursion`がアクティブなら、たとえエラーや quit のときでも外部の選択されたウィンドウが再選択される。`edebug-save-windows`の値がリストなら、それにリストされたウィンドウだけが保存およびリストアされる。

  ただしソースコードバッファーのウィンドウの開始位置と水平スクロールはリストアされないので、表示は Edebug 内で整合性が保たれたままとなる。
- `edebug-save-displayed-buffer-points`が非`nil`なら、表示されているそれぞれのバッファー内のポイント値は保存およびリストアされる。
- 変数 `overlay-arrow-position`と `overlay-arrow-string`は保存とリストアが行われるので、同じバッファー内の他の場所の再帰編集から安全に Edebug を呼び出せる。
- `cursor-in-echo-area`は `nil`にローカルにバインドされるのでカーソルはそのウィンドウ内に現れる。

18.2.14.3 Edebug の再帰編集

Edebug にエンターしてユーザーのコマンドが実際に読み取られるとき、Edebug は以下の追加データを保存 (および後でリストア) します:

- カレントマッチデータ。Section 34.6 [Match Data], page 897 を参照のこと。
- 変数 `last-command`、`this-command`、`last-command-event`、`last-input-event`、`last-event-frame`、`last-nonmenu-event`、`track-mouse`。Edebug 内のコマンドは Edebug 外部のこれらの変数に影響をあたえない。

  Edebug 内でのコマンド実行は `this-command-keys`によりリターンされるキーシーケンスを変更でき、Lisp からそのキーシーケンスをリセットする方法はない。

  Edebug は `unread-command-events`の値の保存とリストアができない。この変数が重要な値をもつときに Edebug にエンターすると、デバッグ中のプログラムの実行に干渉する可能性がある。
- Edebug 内で実行された複雑なコマンドは変数 `command-history`に追加される。これは稀に実行に影響を与える。
- Edebug 内では再帰の深さが Edebug 外部の再帰の深さより 1 つ深くなる。これは自動的に更新される評価リストウィンドウでは異なる。

- `standard-output`と `standard-input`は、`recursive-edit`によって `nil`にバインドされるが Edebug は評価の間それらを一時的にリストアする。
- キーボードマクロ定義の状態は保存およびリストアされる。Edebug がアクティブの間、`defining-kbd-macro`は `edebug-continue-kbd-macro`にバインドされる。

18.2.15 Edebug とマクロ

Edebug が正しくマクロを呼び出す式をインストルメントするには、いくつかの特定な配慮が必要になります。このサブセクションでは、その詳細を説明します。

18.2.15.1 マクロ呼び出しのインストルメント

Edebug が Lisp マクロを呼び出す式をインストルメントするときは、正しくインストルメントを行なうために、そのマクロに関して追加の情報が必要になります。これはマクロ呼び出しのどの部分式 (subexpression) が評価されるフォームなのか推測する方法がないからです (評価はマクロの body で明示的に発生するかもしれないし、展開結果が評価されるとき、または任意のタイミングで行われるかもしれない)。

したがって Edebug が処理するかもしれないすべてのマクロにたいして、そのマクロの呼び出しフォーマットを説明するための Edebug 仕様 (Edebug specification) を定義しなければなりません。これを行なうにはマクロ定義に `debug`宣言を追加します。以下はマクロ例 `for`(Section 14.5.2 [Argument Evaluation], page 245 を参照) にたいする簡単な仕様の例です。

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  (declare (debug (symbolp "from" form "to" form "do" &rest form)))
  ...)
```

この Edebug 仕様はマクロ呼び出しのどの部分が評価されるフォームなのかを示しています。単純なマクロにたいする Edebug 仕様は、そのマクロ定義の正式な引数リストに酷似している場合がありますが、Edebug 仕様はマクロ引数に比べてより汎的です。`declare`フォームの詳細は Section 14.4 [Defining Macros], page 244 を参照してください。

コードをインストルメントするとき Edebug に仕様が確実に解るように留意してください。別ファイルで定義されたマクロを使用する関数をインストルメントする場合には、関数を含むファイル内の `require`フォームを評価するか、あるいはマクロを含むファイルを明示的にロードする必要があるかもしれません。マクロの定義が `eval-when-compile`でラップされていれば、それを評価する必要があるでしょう。

`def-edebug-spec`によりマクロ定義から個々のマクロにたいして Edebug 仕様を定義することもできます。Lisp で記述されたマクロ定義にたいしては `debug`宣言を追加するほうが好ましく便利でもありますが、`def-edebug-spec`では C で実装されたスペシャルフォームにたいして Edebug 仕様を定義することが可能になります。

`def-edebug-spec` *macro specification* [Macro]

マクロ *macro*呼び出しのどの式が評価される式かを指定する。*specification*は Edebug 仕様である。どちらの引数も評価されない。

引数 *macro*には単なるマクロ名ではない、任意の実シンボルを指定できる。

以下は *specification*に指定できるシンボルと、引数を処理する方法のテーブルです。

`t` すべての引数は評価のためにインストルメントされる。

`0` 引数はインストルメントされない。

シンボル　そのシンボルはかわりに使用される Edebug 仕様をもたなければならない。このインダイレクションは他の種類の仕様が見つかるまで繰り返される。これによって他のマクロの仕様を継承できる。

リスト　リストの要素はフォーム呼び出しの引数の型を記述する。仕様リストに指定できる要素については以降のセクションを参照のこと。

マクロが Edebug 仕様をもたなければ、`debug`宣言および `def-edebug-spec`呼び出しのどちらを介しても、変数 `edebug-eval-macro-args`が効果を発揮します。

`edebug-eval-macro-args` [User Option]
これは Edebug が明示的な Edebug 仕様をもたないマクロ引数を扱う方法を制御する。`nil`(デフォルト) なら引数は評価のためにインストルメントされない。それ以外ばらすべての引数がインストルメントされる。

18.2.15.2 仕様リスト

あるマクロ呼び出しにおいて、いくつかの引数は評価されても、それ以外の引数は評価されないような場合には、Edebug 仕様のために仕様リスト *(specification list)* が必要となります。仕様リスト内のいくつかの要素は 1 つ以上の引数にマッチしますが、それ以外の要素は以降に続くすべての引数の処理を変更します。後者は仕様キーワード *(specification keywords)* と呼ばれ、(`&optional`のように)'`&`'で始まるシンボルです。

仕様リストはそれ自身がリストであるような引数にマッチする部分リスト (sublist)、あるいはグループ化に使用されるベクターを含むかもしれません。したがって部分式とグループは仕様リストをレベル階層に細分化します。仕様キーワードは部分式やグループを含むものの残りに適用されます。

仕様リストに選択肢や繰り返しが含まれる場合は、実際のマクロの呼び出しのマッチでバックトラックが要求されるかもしれません。詳細は Section 18.2.15.3 [Backtracking], page 323 を参照してください。

Edebug 仕様はバランスのとれたカッコで括られた部分式へのマッチ、フォームの再帰処理、インダイレクト仕様を通じた再帰等の、正規表現によるマッチングとコンテキストに依存しない文法構成を提供します。

以下は仕様リストに使用できる要素と、その意味についてのテーブルです (使用例は Section 18.2.15.4 [Specification Examples], page 324 を参照):

`sexp`　評価されない単一の Lisp オブジェクト。インストルメントされない。

`form`　評価される単一の式。インストルメントされる。評価前にマクロが式のを `lambda`でラップしていれば、かわりに `def-form`を使用すること (以下の `def-form`を参照)。

`place`　汎変数 (generalized variable)。Section 12.17 [Generalized Variables], page 205 を参照のこと。

`body`　`&rest form`の短縮形 (以下の`&rest`を参照)。評価前にマクロが式のを `lambda`でラップしていれば、かわりに `def-form`を使用すること (以下の `def-form`を参照)。

`function-form`
関数フォーム。クォートされた関数シンボル、クォートされたラムダ式、または (関数シンボルかラムダ式に評価される) フォームのいずれか。これはラムダ式の body をいずれかの方法でインストルメントするので、`function`よりも `quote`でクォートされたラムダ式の引数にたいして有用。

`lambda-expr`
: クォートされないラムダ式。

`&optional`
: 仕様リスト内の後続の要素はオプション。マッチしない要素が出現すると Edebug はこのレベルのマッチングを停止する。

後続が非オプションの要素であるような数個の要素をオプションにするだけなら、`[&optional` *`specs...`*`]`を使用する。複数の要素すべてのマッチや非マッチを指定するには、`&optional [`*`specs...`*`]`を使用する。`defun`の例を参照のこと。

`&rest`
: 仕様リスト内の後続のすべての要素は 0 回以上繰り返される。しかし最後の繰り返しでは、仕様リスト内のすべての要素にたいするマッチングの前に式が終了しても問題はない。

数個の要素を繰り返すには `[&rest` *`specs...`*`]`を使用する。各繰り返しにおいてすべてマッチしなければならない複数要素を指定するには、`&rest [`*`specs...`*`]`を使用する。

`&or`
: 仕様リスト内の後続の各要素は選択肢である。選択肢の 1 つがマッチしなければならず、マッチしなければ`&or`仕様は失敗する。

`&or`に続く各リスト要素は単一の選択肢である。複数のリスト要素を単一の選択肢にグループ化するには、それらを `[...]`で括る。

`&not`
: 後続の各要素は`&or`が使用されたときのように選択肢にマッチするが、要素がマッチしたら失敗となる。マッチする要素がなければ何もマッチされないが`&not`仕様は成功となる。

`&define`
: フォームを定義する仕様であることを示す。フォームを定義する Edebug の定義は、後刻 (フォーム定義の実行後に) 保存および実行される 1 つ以上のコードを含んだフォーム。

フォーム定義自体はインストルメントされない (つまり Edebug はフォーム定義の前後でストップしない) が、フォーム内部は通常はインストルメントされるであろう。`&define` キーワードはリスト仕様の最初の要素であること。

`nil`
: カレント引数レベルでマッチさせる引数が存在しなければ成功し、それ以外は失敗する。部分リスト仕様とバッククォートの例を参照のこと。

`gate`
: 引数はマッチされないが gate を通じたバックトラックは、このレベルの仕様の残りをマッチングする間は無効にされる。これは主に特定の構文エラーメッセージを一般化するために使用される。詳細は Section 18.2.15.3 [Backtracking], page 323、および `let` の例も参照のこと。

`other-symbol`
: 仕様リスト内のその他の要素は、述語 (predicate) かインダイレクト仕様 (indirect specification) である。

シンボルが Edebug 仕様をもつなら、インダイレクト仕様 *(indirect specification)* はシンボル位置に使用されるリスト仕様か、引数を処理するための関数のいずれかである。この仕様はマクロにたいする `def-edebug-spec`のように定義される。`defun`の例を参照のこと。

それ以外ならシンボルは述語 (predicate) である。述語は引数とともに呼び出されて `nil` をリターンしたら、その仕様は失敗して引数はインストルメントされない。

適切な述語としては `symbolp`、`integerp`、`stringp`、`vectorp`、`atom`が含まれる。

`[`*`elements...`*`]`
: 要素のベクターは要素を単一のグループ仕様 *(group specification)* にグループ化する。このグループ仕様はベクター自体には何も行わない。

`"`*`string`*`"` 引数は *string* という名前のシンボルである。この仕様は *symbol* の名前が *string* であるようなクォートされたシンボル`'`*`symbol`* と等価だが、文字列形式のほうが好ましい。

`(vector` *`elements...`*`)`
引数は要素が仕様内の *elements* にマッチするようなベクターである。バッククォートの例を参照のこと。

`(`*`elements...`*`)`
他のリストは部分リスト仕様 *(sublist specification)* であり、引数は要素が仕様の *elements* にマッチするリストでなければならない。

部分リスト仕様はドットリスト (dotted list) かもしれず、その場合対応するリスト引数はドットリストである。かわりにドットリスト仕様の最後の CDR が、(グループ化やインダイレクト仕様による) 他の部分リスト仕様かもしれない (たとえば要素が非ドットリストにマッチする `(spec . [(more specs...)])`)。これはバッククォートの例のような再帰仕様に有用。このような再帰を終了させるには上述の `nil` 仕様も参照のこと。

`(specs . nil)` のように記述された部分リスト仕様は `(specs)`、`(specs . (sublist-elements...))` は `(specs sublist-elements...)` と等価であることに注意。

以下は `&define` の後だけに出現する追加仕様のリストです。`defun` の例を参照してください。

`name` 引数 (シンボル) は定義フォームの名前。

定義フォームは名前フィールドをもつ必要はなく、複数の名前フィールドをもつかもしれない。

`:name` この構文は引数に実際のマッチは行わない。`:name` の後の要素はシンボルであり、その定義の追加の名前要素として使用される。定義名に一意で静的な要素を加えるためにこれを使用できる。複数回使用できる。

`arg` 引数 (シンボル) は定義フォームの引数の名前である。しかし lambda-list キーワード (‘`&`’で始まるシンボル) は許されない。

`lambda-list`
これはラムダリスト (ラムダ式の引数リスト) にマッチする。

`def-body` 引数は定義内のコードの body である。これは上述の `body` と似ているが、定義の body はその定義に関連する情報を照会する別の Edebug 呼び出しでインストルメントされていなければならない。定義内のより高位レベルのフォームリストには `def-body` を使用する。

`def-form` 引数は定義内のもっとも高位レベルの単一フォームである。これは `def-body` と似ているが、フォームリストではなく単一フォームのマッチに使用される。特別なケースとして `def-form` はフォームが実行されるときトレース情報を出力しないことも意味する。`interactive` の例を参照のこと。

18.2.15.3 仕様でのバックトレース

あるポイント位置で仕様がマッチに失敗しても、構文エラーがシグナルされるとは限りません。そのかわりバックトラッキング *(backtracking)* が開始されます。バックトラックはすべての選択肢をマッチングするまで行なわれます。最終的に引数リストのすべての要素は仕様内の要素のいずれかとマッチしなければならず、仕様内の必須要素は引数のいずれかとマッチしなければなりません。

構文エラーが検出されてもその時点では報告されず、より高位レベルの選択肢のマッチングが終わった後、実際のエラー箇所から離れたポイント位置でエラーが報告されるかもしれません。しかしエラー発生時にバックトラックが無効ならエラーは即座に報告されるでしょう。ある状況ではバックトラックも自動的に再有効化されることに注意してください。`&optional`、`&rest`、`&or`により新たな選択肢が設定されたとき、または部分リスト、グループ、インダイレクト仕様が開始されたときはバックトラックが自動的に有効になります。バックトラックを有効、または無効にした場合の影響は、現在処理中のレベルの残り要素と低位レベルに限定されます。

何らかのフォーム仕様 (すなわち `form`、`body`、`def-form`、`def-body`) をマッチングする間、バックトラックは無効になっています。これらの仕様は任意のフォームにマッチするので、何らかのエラーが発生するとしたらそれは高位レベルではなく、そのフォーム自体の内部でなければなりません。

バックトラックはクォートされたシンボルや文字列仕様とのマッチに成功した後にも無効になります。なぜなら通常これは構文成が認識されたことを示すからです。しかし同じシンボルで始まる一連の選択肢構文がある場合には、たとえば `["foo" &or [first case] [second case] ...]`のように、通常は選択肢の外部にそのシンボルをファクタリングすることによりこの制約に対処できます。

ほとんどのニーズは、バックトラックを自動的に無効にする、これら 2 つの方法で満足させることができますが、`gate`仕様を使用して明示的にバックトラックを無効にするほうが便利なときもあります。これは高位に適用可能な選択肢が存在しないことが分かっている場合に有用です。`let`仕様の例を参照してください。

18.2.15.4 仕様の例

以下で提供する例から学ぶことにより、Edebug 仕様の理解が容易になるでしょう。

スペシャルフォーム `let`は、バインディングと body のシーケンスをもちます。各バインディングはそシンボル、またはシンボルとオプションの部分リストです。以下の仕様では部分リストを見つけたらバックトラックを抑止するために、部分リスト内の `gate`があることに注目してください。

```
(def-edebug-spec let
  ((&rest
    &or symbolp (gate symbolp &optional form))
   body))
```

Edebug は `defun`および関連する引数リスト、`interactive`仕様にたいして以下の仕様を使用します。式の引数はその関数 body の外部で実際に評価されるので、interactive フォームは特別に処理する必要があります。(`defmacro`にたいする仕様は `defun`にたいする仕様と酷似するが `declare`命令文が許される)

```
(def-edebug-spec defun
  (&define name lambda-list
           [&optional stringp]   ; ドキュメント文字列が与えられた場合はマッチする。
           [&optional ("interactive" interactive)]
           def-body))

(def-edebug-spec lambda-list
  (([&rest arg]
    [&optional ["&optional" arg &rest arg]]
    &optional ["&rest" arg]
    )))

(def-edebug-spec interactive
  (&optional &or stringp def-form))    ; def-formに注目
```

以下のバッククォートにたいする仕様はドットリストにマッチさせる方法と、`nil`を使用して再帰を終了させる方法を説明するための例です。またベクターのコンポーネントをマッチさせる方法も示

しています (Edebug により定義される実際の仕様は少し異なり、失敗するかもしれない非常に深い再帰を引き起こすためドットリストについてはサポートしない)。

```
(def-edebug-spec \` (backquote-form))   ; 単なる明確化用エイリアス

(def-edebug-spec backquote-form
  (&or ([&or "," ",@"] &or ("quote" backquote-form) form)
       (backquote-form . [&or nil backquote-form])
       (vector &rest backquote-form)
       sexp))
```

18.2.16 Edebug のオプション

以下のオプションは Edebug の動作に影響を与えます:

`edebug-setup-hook` [User Option]
Edebug が使用される前に呼び出される関数。この関数は毎回新たな値をセットする。Edebug はこれらの関数を一度呼び出したら、その後に `edebug-setup-hook`を `nil`にリセットする。使用するパッケージに関係する Edebug 仕様をロードするために使用で d きるがそれは Edebug を使用するときだけである。Section 18.2.2 [Instrumenting], page 307 を参照のこと。

`edebug-all-defs` [User Option]
これが非 `nil`の場合に `defun`や `defmacro`のような定義フォームの普通に評価すると、Edebug 用にインストルメントされる。これは `eval-defun`、`eval-region`、`eval-buffer`、and `eval-current-buffer`に適用される。

このオプションの切り替えにはコマンド *`M-x edebug-all-defs`*を使用する。Section 18.2.2 [Instrumenting], page 307 を参照のこと。

`edebug-all-forms` [User Option]
これが非 `nil`の場合には `eval-defun`、`eval-region`、`eval-buffer`、`eval-current-buffer`はたとえフォームが何も定義していなくても、すべてのフォームをインストルメントする。これはロードとミニバッファー内の評価には適用されない。

このオプションの切り替えにはコマンド *`M-x edebug-all-forms`*を使用する。Section 18.2.2 [Instrumenting], page 307 を参照のこと。

`edebug-eval-macro-args` [User Option]
これが非 `nil`なら、生成されるコード内のすべてのマクロ引数がインストルメントされる。`edebug-form-spec`はすべてのマクロにたいして、このオプションをオーバーライドする。ある引数を評価して他の引数を評価しないマクロにたいする例外を指定するためには、`edebug-form-spec`を指定するために `def-edebug-spec`を使用すること。

`edebug-save-windows` [User Option]
これが非 `nil`なら、Edebug はウィンドウ構成の保存とリストアを行なう。これにはある程度の時間を要するので、ウィンドウ構成に何が起こってもプログラムに関係なければ、この変数を `nil`にセットしたほうがよい。

値がリストならリストされたウィンドウだけが保存およびリストアされる。

Edebug 内ではこの変数をインタラクティブに変更するために *`W`*コマンドを使用できる。Section 18.2.14.2 [Edebug Display Update], page 319 を参照のこと。

`edebug-save-displayed-buffer-points` [User Option]
これが非 `nil`なら Edebug は表示されているすべてのバッファー内のポイントを保存およびリストアする。

選択されていないウィンドウ内に表示されているバッファーのポイントを変更するコードをデバッグしている場合は、他のバッファーのポイントを保存およびリストアする必要がある。その後に Edebug またはユーザーがそのウィンドウを選択した場合は、そのバッファー内のポイントはそのウィンドウのポイント値に移動される。

すべてのバッファー内のポイントの保存とリストアは、それぞれのウィンドウを 2 回選択する必要があり高価な処理なので、必要なときだけ有効にする。Section 18.2.14.2 [Edebug Display Update], page 319 を参照のこと。

`edebug-initial-mode` [User Option]
この変数が非 `nil`なら、Edebug が最初にアクティブになったときの Edebug の最初の実行モードを指定する。指定できる値は `step`、`next`、`go`、`Go-nonstop`、`trace`、`Trace-fast`、`continue`、`Continue-fast`。

デフォルト値は `step`。この変数は *C-x C-a C-m*でインタラクティブにセットできる。Section 18.2.3 [Edebug Execution Modes], page 308 を参照のこと。

`edebug-trace` [User Option]
これが非 `nil`なら各関数のエントリーと exit をトレースする。トレース出力は関数のエントリーと exit を行ごとに、再帰レベルにしたがって`*edebug-trace*`という名前のバッファーに表示される。

Section 18.2.12 [Trace Buffer], page 317 の `edebug-tracing`も参照されたい。

`edebug-test-coverage` [User Option]
非 `nil`なら Edebug はデバッグされるすべての式のカバレッジをテストする。Section 18.2.13 [Coverage Testing], page 317 を参照のこと。

`edebug-continue-kbd-macro` [User Option]
非 `nil`なら Edebug 外部で実行されている任意のキーボードマクロの定義または実行を継続する。これはデバッグされないので慎重に使用すること。Section 18.2.3 [Edebug Execution Modes], page 308 を参照されたい。

`edebug-print-length` [User Option]
非 `nil`なら、それは Edebug での結果プリントにおける `print-length`のデフォルト値。Section 19.6 [Output Variables], page 339 を参照のこと。

`edebug-print-level` [User Option]
非 `nil`なら、それは Edebug での結果プリントにおける `print-level`のデフォルト値。Section 19.6 [Output Variables], page 339 を参照のこと。

`edebug-print-circle` [User Option]
非 `nil`なら、それは Edebug での結果プリントにおける `print-circle`のデフォルト値。Section 19.6 [Output Variables], page 339 を参照のこと。

`edebug-unwrap-results` [User Option]
非 `nil`なら Edebug は式の結果を表示するときに、その式自体のインストルメント結果の削除を試みる。マクロをデバッグするときは、式の結果自体がインストルメントされた式になるということに関連するオプションである。実際的な例ではないが、サンプル例の関数 `fac`がインストルメントされたとき、そのフォームのマクロを考えてみるとよい。

```
(defmacro test () "Edebug example."
  (if (symbol-function 'fac)
```

```
...))
```

testマクロをインストルメントしてステップ実行すると、デフォルトでは symbol-function 呼び出しは多数の edebug-after フォームと edebug-before フォームをもつことになり、それにより実際の結果の確認が難しくなり得る。edebug-unwrap-results が非 nil なら Edebug は結果からこれらのフォームの削除を試みる。

edebug-on-error [User Option]
debug-on-error が以前 nil だったら、Edebug は debug-on-error をこの値にバインドする。Section 18.2.7 [Trapping Errors], page 313 を参照のこと。

edebug-on-quit [User Option]
debug-on-quit の以前の値が nil なら、Edebug は debug-on-quit にこの値をバインドする。Section 18.2.7 [Trapping Errors], page 313 を参照のこと。

Edebug がアクティブな間に edebug-on-error か edebug-on-quit の値を変更したら、次回に新たなコマンドを通じて Edebug が呼び出されるまでこれらの値は使用されない。

edebug-global-break-condition [User Option]
非 nil なら、値はすべてのステップポイントでテストされる式である。式の結果が nil なら break する。エラーは無視される。Section 18.2.6.2 [Global Break Condition], page 313 を参照のこと。

edebug-sit-for-seconds [User Option]
実行モードが trace か continue で breakpoint に達した際に一時停止する秒数。Section 18.2.3 [Edebug Execution Modes], page 308 を参照のこと。

edebug-sit-on-break [User Option]
breakpoint に達したときに edebug-sit-for-seconds の間、一時停止するかどうか。nil で一時停止の抑止、非 nil なら一時停止を許可。

edebug-behavior-alist [User Option]
デフォルトでは、この alist にはキーが edebug で 3 つの関数 edebug-enter、edebug-before、edebug-after からなるリストという 1 つのエントリーが含まれる。これらの関数はインストルメントされるコードに挿入される関数のデフォルト実装である。Edebug の全般的な挙動を変更するためには、このデフォルトエントリーを変更する。

Edebug の挙動はこの alist にエントリーにユーザーが選択したキーと 3 つの関数を追加することにより、定義ごとにもとづいて変更もできる。それからインストルメントされた定義のシンボルプロパティ edebug-behavior に新たなエントリーのキーをセットすれば、Edebug はその定義にたいして自身の関数を呼び出す箇所で新たな関数を呼び出す。

edebug-new-definition-function [User Option]
定義やクロージャの body をラップした後に Edebug が実行する関数。Edebug が自身のデータを初期化後に、この関数は定義に関連付けられたシンボル (Edebug が定義または生成した実際のシンボルかもしれない) を単一の引数として呼び出される。この関数は Edebug によりインストルメントされる各定義のシンボルプロパティ edebug-behavior をセットするために使用されるかもしれない。

edebug-after-instrumentation-function [User Option]
使用前に Edebug のインストルメントの検査や修正を行うには、インストルメントするトップレベルのフォームを単一の引数として受け取り、その後に Edebug がインストルメントの最終

結果として使用することになる同一フォームあるいは置換フォームをリターンする関数をこの変数にセットする。

18.3 無効な Lisp 構文のデバッグ

Lisp リーダーは無効な構文 (invalid syntax) について報告はしますが実際の問題箇所は報告しません。たとえばある式を評価中のエラー '`End of file during parsing`'は、開カッコまたは開角カッコ (open parenthese or open square bracket) が多いことを示しています。Lisp リーダーはこの不一致をファイル終端で検出しましたが、本来閉カッコがあるべき箇所を解決することはできません。同様に '`Invalid read syntax: \")\"`'は開カッコの欠落を示していますが、欠落しているカッコが属すべき場所は告げません。ならばどうやって変更すべき箇所を探せばよいのでしょうか?

問題が単なるカッコの不一致でない場合の便利なテクニックは、各 defun の先頭で *C-M-e* (`end-of-defun`。Section "Moving by Defuns" in *The GNU Emacs Manual* を参照) とタイプして、その defun の最後と思われる箇所に移動するか確認する方法です。もし移動しなければ、問題はその defun の内部にあります。

マッチしないカッコが Lisp においてもっとも一般的な構文エラーなので、これらのケースにたいしてさらにアドバイスすることができます (Show Paren モードを有効にしてコードにポイントを移動するだけでカッコの不一致を探しやすくなるだろう)。

18.3.1 過剰な開カッコ

カッコがマッチしない defun を探すのが、最初のステップです。過剰な開カッコが存在する場合は、ファイルの終端に移動して *C-u C-M-u* (`backward-up-list`。Section "Moving by Parens" in *The GNU Emacs Manual* を参照) とタイプします。これにより、カッコがマッチしない最初の defun の先頭に移動するでしょう。

何が間違っているのか正確に判断するのが次のステップです。これを確実に行なうにはプログラムを詳しく調べる以外に方法はありませんが、カッコがあるべき箇所を探すのに既存のインデントが手掛かりになることが多々あります。*C-M-q* (`indent-pp-sexp`。Section "Multi-line Indent" in *The GNU Emacs Manual* を参照) で再インデントして何が移動されるか確認するのが、この手掛かりを使用するもっとも簡単な方法です。しかし、行うのはちょっと待ってください! まず続きを読んでからにしましょう。

これを行なう前に defun に十分な閉カッコがあるか確認します。十分な閉カッコがなければ *C-M-q* がエラーとなるか、その defun からファイル終端までの残りすべてが再インデントされます。その場合には defun の最後に移動して、そこに閉カッコを挿入します。その defun のカッコの釣り合いがとれるまでは、defun の最後に移動するのに *C-M-e* (`end-of-defun`) は使用できません (失敗する)。

これで defun の先頭に移動して *C-M-q*とタイプすることができます。通常は一定のポイントからその関数の最後までのすべての行が、右へとシフトされるでしょう。これはおそらくそのポイント付近で閉カッコが欠落していたり不要な開カッコがあります (しかしこれを真実と仮定せずコードを詳しく調べること)。不一致箇所を見つけたら、元のインデントはおそらく意図されたカッコに適しているはずなので、*C-_* (`undo`) で *C-M-q*をアンドゥしてください。

問題を fix できたと思った後に、再度 *C-M-q*を使用します。実際に元のインデントが意図したカッコのネストに適合していて、足りないカッコを追加していたら、*C-M-q*は何も変更しないはずです。

18.3.2 過剰な閉カッコ

過剰な閉カッコへの対処は、まずファイルの先頭に移動してからカッコのマッチしない defun を探すために *C-u -1 C-M-u* (引数 −1 で `backward-up-list`) をタイプします。

それから defun の先頭で *C-M-f* (`forward-sexp`。Section “Expressions” in *The GNU Emacs Manual* を参照) をタイプして、実際にマッチする閉カッコを探します。これにより defun の終端より幾分手前の箇所に移動するでしょう。その付近に間違った閉カッコが見つかるはずです。

そのポイントに問題が見つからない場合には、その defun の先頭で *C-M-q* (`indent-pp-sexp`) をタイプするのが次のステップです。ある行範囲はおそらく左へシフトするでしょう。その場合には欠落している開カッコや間違った閉カッコは、おそらくそれらの行の 1 行目の近くにあるでしょう (しかしこれを真実と仮定せずコードを詳しく調べること)。不一致箇所が見つけたら、元のインデントはおそらく意図されたカッコに適しているはずなので *C-_* (`undo`) で *C-M-q*をアンドゥしてください。

問題を fix できたと思った後に再度 *C-M-q*を使用します。実際に元のインデントが意図したカッコのネストに適合していて、足りないカッコを追加していたら、*C-M-q*は何も変更しないはずです。

18.4 カバレッジテスト

`testcover`ライブラリーをロードしてコマンド *M-x testcover-start RET file RET*でコードをインストルメントすることにより、Lisp コードのファイルにたいしてカバレッジテストを行なうことができます。コードを 1 回以上呼び出すことによってテストが行なわれます。コマンド *M-x testcover-mark-all*を使用すれば、カバレッジが不十分な箇所が色付きでハイライト表示されます。コマンド *M-x testcover-next-mark*は次のハイライトされた箇所へポイントを前方に移動します。

赤くハイライトされた箇所は通常はそのフォームが完全に評価されたことが一度もないことを示し、茶色でハイライトされた箇所は常に同じ値に評価された (その結果にたいして少ししかテストされていない) ことを意味します。しかし `error`のように完全に評価するのが不可能なフォームにたいしては、赤いハイライトはスキップされます。`(setq x 14)`のように常に同じ値に評価されることが期待されるフォームにたいしては、茶色のハイライトはスキップされます。

難しいケースではテストカバレッジツールにアドバイスを与えるために、コードに do-nothing マクロを追加することができます。

`1value` *form* [Macro]
: *form*を評価してその値をリターンするが、テストカバレッジにたいして *form*が常に同じ値だという情報を与える。

`noreturn` *form* [Macro]
: *form*を評価して *form*が決してリターンしないという情報をカバレッジテストに与える。もしリターンしたら run-time エラーとなる。

Edebug にもカバレッジテスト機能があります (Section 18.2.13 [Coverage Testing], page 317 を参照)。これらの機能は部分的に重複しており、組み合わせることで明確になるでしょう。

18.5 プロファイリング

プログラムは正常に機能しているものの、十分に高速ではないのでより高速かつ効率的に実行させたい場合には、そのプログラムが実行時間の大半をどこで消費しているか知るために、コードをプロファイル *(profile)* することが最初に行うべきことです。ある特定の関数の実行が実行時間のうちの無視できない割り合いを占めるようなら、その部分を最適化する方法を探すことを開始できます。

Emacs にはこのためのビルトインサポートがあります。プロファイリングを開始するには *M-x profiler-start*をタイプします。プロファイルはプロセッサー使用 (processor usage) かメモリー使用 (memory usage)、またはその両方を選択できます。それから高速化したいコードを実行します。その後に *M-x profiler-report*とタイプするとプロファイルに選択した各リソース (CPU とメモ

リー) が summary バッファーに表示されます。report バッファーの名前にはレポートが生成された時刻が含まれるので、前の結果を消去せずに後で他のレポートを生成できます。プロファイリングが終了したら *M-x profiler-stop*とタイプしてください (プロファイリングに関連したオーバーヘッドが若干あるので実際に調査したいコードの実行中以外にアクティブのままに放置することは推奨しない)。

profiler report バッファーでは、各行に呼び出された関数、その後にプロファイリングが開始されてから使用したリソース (CPU またはメモリー) の絶対時間とパーセンテージが表示されます。左側にシンボル '+'のある行では RETをタイプして行を展開して高位レベルの関数に呼び出された関数を確認できます。関数ツリー配下全体の呼び出しを確認するにはプレフィクス引数を使用します (*C-u RET*)。もう一度 RETをタイプすれば元の状態へと行が折り畳まれます。

*j*か *mouse-2*を押下するとポイント位置の関数の定義にジャンプします。*d*を押下すると関数のドキュメントを閲覧できます。*C-x C-w*を使用してプロファイルをファイルに保存できます。=を使用すれば 2 つのプロファイルを比較することができます。

elpライブラリーはプロファイルしたい Lisp 関数が事前に解っているときに有用な別のアプローチを選択肢として提供します。このライブラリーの使用するには、まず elp-function-listに関数シンボルのリスト (プロファイルしたい関数) をセットします。それから関数をプロファイル用にアレンジするために *M-x elp-instrument-list RET nil RET* とタイプします。プロファイルしたいコードの実行後に *M-x elp-results* を呼び出してカレント結果を表示します。処理手順の詳細については elp.elファイルを参照してください。このアプローチは Lisp で記述された関数のプロファイリングに限定されており、Emacs プリミティブのプロファイルはできません。

benchmarkライブラリーを使用して個々の Emacs Lisp フォームの評価に消費される時間を計測できます。benchmark.el内のマクロ benchmark-runと benchmark-run-compiled、および benchmark-prognを参照してください。フォームをインタラクティブに時間計測するために benchmarkコマンドも使用できます。

19 Lisp オブジェクトの読み取りとプリント

プリント *(print)* と読み取り *(read)* は Lisp オブジェクトからテキスト形式への変換、またはその逆の変換を行なう操作です。これらは Chapter 2 [Lisp Data Types], page 8 で説明したプリント表現 (printed representation) と入力構文 (read syntax) を使用します。

このチャプターでは読み取りとプリントのための Lisp 関数について説明します。このチャプターではさらにストリーム *(stream)* についても説明します。ストリームとは、(読み取りでは) テキストがどこから取得されるか、(プリントでは) テキストをどこに出力するかを指定します。

19.1 読み取りとプリントの概念

Lisp オブジェクトの読み取りとは、テキスト形式の Lisp 式をパース (parse: 解析) して、対応する Lisp オブジェクトを生成することを意味します。これは LLisp プログラムが Lisp コードファイルから Lisp に取得される方法でもあります。わたしたちはそのテキストのことを、そのオブジェクトの入力構文 *(read syntax)* と呼んでいます。たとえばテキスト '`(a . 5)`'は、CAR が `a`で CDR が数字の 5 であるようなコンスセルにたいする入力構文です。

Lisp オブジェクトのプリントとは、あるオブジェクトをそのオブジェクトのプリント表現 *(printed representation)* に変換することによって、そのオブジェクトを表すテキストを生成することを意味します (Section 2.1 [Printed Representation], page 8 を参照)。上述のコンスセルをプリントするとテキスト '`(a . 5)`'が生成されます。

読み取りとプリントは概ね逆の処理といえます。あるテキスト断片を読み取った結果として生成されたオブジェクトをプリントすると、多くの場合は同じテキストが生成され、あるオブジェクトをプリントした結果のテキストを読み取ると、通常は同じようなオブジェクトが生成されます。たとえばシンボル `foo`をプリントするとテキスト '`foo`'が生成されて、そのテキストを読み取るとシンボル `foo` がリターンされます。要素が `a`と `b`のリストをプリントするとテキスト '`(a b)`'が生成されて、そのテキストを読み取ると、(同じリストではないが) 要素が `a`と `b`のリストが生成されます。

しかし、これら 2 つの処理は互いにまったく逆の処理というわけではありません。3 つの例外があります:

- プリントは読み取ることが不可能なテキストを生成できる。たとえばバッファー、フレーム、サブプロセス、マーカーは '`#`'で始まるテキストとしてプリントされる。このテキストの読み取りを試みるとエラーとなる。これらのデータ型を読み取る方法は存在しない。
- 1 つのオブジェクトが複数のテキスト的な表現をもつことができる。たとえば '`1`'と '`01`'は同じ整数を表し、'`(a b)`'と '`(a . (b))`'は同じリストを表す。読み取りは複数の候補を受容するかもしれないが、プリントはそのうちのただ 1 つを選択しなければならない。
- あるオブジェクトの読み取りシーケンスの中間の特定ポイントに、読み取り結果に影響を与えないコメントを置くことができる。

19.2 入力ストリーム

テキストを読み取る Lisp 関数の大部分は、引数として入力ストリーム *(input stream)* を受け取ります。入力ストリームは読み取られるテキストの文字をどこから、どのように取得するかを指定します。以下は利用できる入力ストリーム型です:

buffer 入力文字は *buffer*のポイントの後の文字から直接読み取られる。文字の読み取りとともにポイントが進む。

marker 入力文字は *marker*があるバッファーの、マーカーの後の文字から直接読み取られる。文字の読み取りとともにマーカーが進む。ストリームがマーカーならバッファー内のポイント値に影響はない。

string 入力文字は *string*の最初の文字から必要な文字数分が取得される。

function 入力文字は *function*から生成され、その関数は 2 種類の呼び出しをサポートしなければならない:

- 引数なしで呼び出されたときは次の文字をリターンする。
- 1 つの引数 (常に文字) で呼び出されたとき、*function*は引数を保存して次の呼び出しでリターンするように準備する。これは文字の読み戻し *(unreading)* と呼ばれ、Lisp リーダーが 1 文字多く読みとったとき、それを ‘読みとった場所に戻したいときに発生する。この場合には *function*のリターン値と違いはない。

`t` `t`はその入力がミニバッファーから読み取られるストリームであることを意味する。実際にはミニバッファーが 1 回呼び出されて、ユーザーから与えられたテキストが、その後に入力ストリームとして使用される文字列となる。Emacs が batch モードで実行されている場合には、ミニバッファーのかわりに標準入力が使用される。たとえば、

```
(message "%s" (read t))
```

このような場合には標準入力から Lisp 式が読み取られて、結果は標準出力にプリントされるだろう。

`nil` 入力ストリームとして `nil`が与えられた場合は、かわりに `standard-input`の値が使用されることを意味する。この値はデフォルトの入力ストリーム *(default input stream)* であり、非 `nil`の入力ストリームでなければならない。

symbol 入力ストリームとしてのシンボルは、(もしあれば) そのシンボルの関数定義と等価である。

以下の例ではバッファーストリームから読み込んで、読み取りの前後におけるポイント位置を示しています:

```
---------- Buffer: foo ----------
This⋆ is the contents of foo.
---------- Buffer: foo ----------

(read (get-buffer "foo"))
     ⇒ is
(read (get-buffer "foo"))
     ⇒ the

---------- Buffer: foo ----------
This is the⋆ contents of foo.
---------- Buffer: foo ----------
```

最初の読み取りではスペースがスキップされていることに注意してください。読み取りでは意味のあるテキストに先行する、任意のサイズの空白文字がスキップされます。

以下はマーカーストリームからの読み取りの例で、最初は表示されているバッファーの先頭にマーカーを配置されています。読み取られた値はシンボル `This`です。

```
---------- Buffer: foo ----------
This is the contents of foo.
---------- Buffer: foo ----------

(setq m (set-marker (make-marker) 1 (get-buffer "foo")))
     ⇒ #<marker at 1 in foo>
(read m)
     ⇒ This
m
     ⇒ #<marker at 5 in foo>   ;; 最初のスペースの前
```

以下では文字列のコンテンツから読み取っています:

```
(read "(When in) the course")
     ⇒ (When in)
```

以下はミニバッファーから読み取る例です。プロンプトは‘Lisp expression: ’です (このプロンプトはストリーム tから読み取る際は常に使用される)。ユーザーの入力はプロンプトの後に表示されます。

```
(read t)
     ⇒ 23
---------- Buffer: Minibuffer ----------
Lisp expression: 23 RET
---------- Buffer: Minibuffer ----------
```

最後は useless-streamという名前の関数ストリームから読み取る例です。ストリームを使用する前に変数 useless-listを文字のリストで初期化しています。その後はリスト内の次の文字を取得するため、または文字をリストの先頭に追加することにより読み戻すために関数 useless-streamを呼び出します。

```
(setq useless-list (append "XY()" nil))
     ⇒ (88 89 40 41)

(defun useless-stream (&optional unread)
  (if unread
      (setq useless-list (cons unread useless-list))
    (prog1 (car useless-list)
           (setq useless-list (cdr useless-list)))))
     ⇒ useless-stream
```

このストリームを使って以下のように読み取ります:

```
(read 'useless-stream)
     ⇒ XY

useless-list
     ⇒ (40 41)
```

開カッコと閉カッコがリスト内に残されることに注意してください。Lisp リーダーは開カッコに出会うと、それを入力の終わりと判断して読み戻します。次にこのポイント位置からこのストリームを読み取ると、‘()’が読み取られて nilがリターンされます。

19.3 入力関数

このセクションでは、読み取りに関係のある Lisp 関数と変数について説明します。

以下の関数では *stream*は入力ストリーム (前のセクションを参照) を意味します。*stream*が `nil` または省略された場合のデフォルト値は `standard-input`です。

読み取りにおいて終端されていないリスト、ベクター、文字列に遭遇したら `end-of-file`がシグナルされます。

`read` **&optional** *stream* [Function]
この関数は *stream*からテキスト表現された Lisp 式を 1 つ読み取って Lisp オブジェクトとしてリターンする。これは基本的な Lisp 入力関数である。

`read-from-string` *string* **&optional** *start end* [Function]
この関数は *string*内のテキストからテキスト表現された最初の Lisp 式を読み取る。リターン値は CAR がその式で、CDR が次に読み取られるその文字列内の残りの文字 (読み取られていない最初の文字) の位置を与える整数であるようなコンスセルである。

*start*が与えられると、文字列内のインデックス *start*(最初の文字はインデックス 0) から読み取りが開始される。*end*を指定すると、残りの文字列が存在しないかのごとくそのインデックスの直前で読み取りがストップされる。

たとえば:

```
(read-from-string "(setq x 55) (setq y 5)")
     ⇒ ((setq x 55) . 11)
(read-from-string "\"A short string\"")
     ⇒ ("A short string" . 16)

;; 最初の文字から読み取りを開始
(read-from-string "(list 112)" 0)
     ⇒ ((list 112) . 10)
;; 2 つ目の文字から読み取りを開始
(read-from-string "(list 112)" 1)
     ⇒ (list . 5)
;; 7 番目の文字から読み取りを開始
;;    して 9 番目の文字で停止
(read-from-string "(list 112)" 6 8)
     ⇒ (11 . 8)
```

`standard-input` [Variable]
この変数はデフォルト入力ストリーム (引数 *stream*が `nil`のときに `read`が使用するストリーム) を保持する。デフォルトは `t`で、これはミニバッファーを使用することを意味する。

`read-circle` [Variable]
非 `nil`なら、この変数は循環構造 (circular structure) と共有構造 (shared structures) の読み取りを有効にする。Section 2.6 [Circular Objects], page 29 を参照のこと。デフォルト値は `t`。

batch モードの Emacs プロセスの標準入力や標準出力のストリームにたいして読み取りや書き込みを行う際には、任意のバイナリーデータにたいしてそのまま読み取りや書き込みを行うことや、改行と CR-LF の変換を何も行わないことを保証することが要求される場合があります。この問題は

MS-Windows と MS-DOS だけに存在する問題であり、POSIX ではそのような問題はありません。以下の関数により Emacs プロセスのすべての標準ストリームの I/O モードを制御することができます。

`set-binary-mode` *stream mode* [Function]
: *stream*の I/O モードのバイナリーとテキストを切り替える。*mode*が非 `nil`ならバイナリーモード、それ以外ならテキストモードに切り替える。*stream*の値は `stdin`、`stdout`、`stderr` のいずれか。この関数は副作用として *stream*の保留中の出力データをすべてフラッシュして、*stream*の以前の I/O モードの値をリターンする。POSIX では常に非 `nil`値をリターンして、保留中の出力のフラッシュ以外は何も行わない。

19.4 出力ストリーム

出力ストリームはプリントによって生成された文字に何を行うかを指定します。ほとんどのプリント関数は引数としてオプションで出力ストリームを受け取ります。以下は利用できる出力ストリーム型です:

buffer
: 出力文字は *buffer*のポイント位置に挿入される。文字が挿入された分だけポイントが進む。

marker
: 出力文字は *marker*があるバッファーのマーカー位置に挿入される。文字が挿入された分だけマーカー位置が進む。ストリームがマーカーのときは、そのバッファー内のポイント位置にプリントは影響せず、この種のプリントでポイントは移動しない (マーカー位置がポイント位置かポイント位置より前の場合は除く。通常はテキストの周囲にポイントが進む)。

function
: 出力文字は文字を格納する役目をもつ *function*に渡される。この関数は 1 つの文字を引数に出力される文字の回数呼び出され、格納したい場所にその文字を格納する役目をもつ。

`t`
: 出力文字はエコーエリアに表示される。

`nil`
: 出力ストリームに `nil`が指定された場合は、かわりに `standard-output`の値が使用されることを意味する。この値はデフォルトの出力ストリーム *(default output stream)* であり、非 `nil`でなければならない。

symbol
: 出力ストリームとしてのシンボルは、(もしあれば) そのシンボルの関数定義と等価である。

有効な出力ストリームの多くは、入力ストリームとしても有効です。したがって入力ストリームと出力ストリームの違いは、Lisp オブジェクトの型ではなく、どのように Lisp オブジェクトを使うかという点です。

以下はバッファーを出力ストリームとして使用する例です。ポイントは最初は '`the`'の中の '`h`'の直前にあります。そして最後も同じ '`h`'の直前に配置されます。

```
---------- Buffer: foo ----------
This is t*he contents of foo.
---------- Buffer: foo ----------

(print "This is the output" (get-buffer "foo"))
     ⇒ "This is the output"
```

```
---------- Buffer: foo ----------
This is t
"This is the output"
⋆he contents of foo.
---------- Buffer: foo ----------
```

次はマーカーを出力ストリームとして使用する例です。マーカーは最初はバッファー foo内の単語 'the'の中の 't'と 'h'の間にあります。最後には挿入されたテキストによってマーカーが進んで、同じ 'h'の前に留まります。通常の方法で見られるようなポイント位置への影響がないことに注意してください。

```
---------- Buffer: foo ----------
This is the ⋆output
---------- Buffer: foo ----------

(setq m (copy-marker 10))
     ⇒ #<marker at 10 in foo>

(print "More output for foo." m)
     ⇒ "More output for foo."

---------- Buffer: foo ----------
This is t
"More output for foo."
he ⋆output
---------- Buffer: foo ----------

m
     ⇒ #<marker at 34 in foo>
```

以下はエコーエリアに出力を表示する例です:

```
(print "Echo Area output" t)
     ⇒ "Echo Area output"
---------- Echo Area ----------
"Echo Area output"
---------- Echo Area ----------
```

最後は関数を出力ストリームとして使用する例です。関数 eat-outputは与えられたそれぞれの文字を last-outputの先頭に cons します (Section 5.4 [Building Lists], page 75 を参照)。最後にはリストには出力されたすべての文字が逆順で含まれます。

```
(setq last-output nil)
     ⇒ nil

(defun eat-output (c)
  (setq last-output (cons c last-output)))
     ⇒ eat-output

(print "This is the output" #'eat-output)
     ⇒ "This is the output"
```

```
last-output
     ⇒ (10 34 116 117 112 116 117 111 32 101 104
    116 32 115 105 32 115 105 104 84 34 10)
```

このリストを逆転すれば正しい順序で出力することができます:

```
(concat (nreverse last-output))
     ⇒ "
\"This is the output\"
"
```

concatを呼び出してリストを文字列に変換すれば、内容をより明解に確認できます。

`external-debugging-output` *character* [Function]

この関数はデバッグ時の出力ストリームとして有用。これは標準エラーストリームに *character* を書き込む。

たとえば

```
(print "This is the output" #'external-debugging-output)
⊣ This is the output
⇒ "This is the output"
```

19.5 出力関数

このセクションではオブジェクトをオブジェクトのプリント表現に変換して、Lisp オブジェクトをプリントする Lisp 関数を説明します。

Emacs プリント関数には、正しく読み取れるように必要なとき出力にクォート文字を追加するものがあります。使用されるクォート文字は '"'と '\'です。これらは文字列をシンボルと区別するとともに、文字列とシンボル内の区切り文字が読み取りの際に区切り文字として扱われることを防ぎます。完全な詳細は Section 2.1 [Printed Representation], page 8 を参照してください。クォートするかしないかはプリント関数の選択によって指定できます。

そのテキストが Lisp に読み戻す場合、または Lisp プログラマーに Lisp オブジェクトを明解に説明するのが目的の場合は、曖昧さを避けるためにクォート文字をプリントするべきです。しかしプログラマー以外の人間にたいして出力の見栄えを良くするのが目的なら、通常はクォートなしでプリントしたほうがよいでしょう。

Lisp オブジェクトは自己参照ができます。通常の方法で自己参照オブジェクトをプリントするにはテキストが無限に必要であり、その試みにより無限再帰が発生する恐れがあります。Emacs はそのような再帰を検知して、すでにプリントされたオブジェクトを再帰的にプリントするかわりに、'`#level`' をプリントします。たとえば以下はカレントのプリント処理において、レベル 0 のオブジェクトを再帰的に参照することを示しています:

```
(setq foo (list nil))
     ⇒ (nil)
(setcar foo foo)
     ⇒ (#0)
```

以下の関数では *stream*は出力ストリームを意味します (出力ストリームの説明は前のセクションを参照。デバッグではストリーム値として [external-debugging-output], page 337 も有用) *stream* が `nil`か省略された場合のデフォルトは `standard-output`の値です。

`print` *object* **&optional** *stream* [Function]

`print`関数はプリントを行うための便利な手段である。この関数は *object*の前後に改行を付与して *object*のプリント表現を *stream*にプリントする。クォート文字が使用される。`print`は *object*をリターンする。たとえば:

```
(progn (print 'The\ cat\ in)
       (print "the hat")
       (print " came back"))
     ⊣
     ⊣ The\ cat\ in
     ⊣
     ⊣ "the hat"
     ⊣
     ⊣ " came back"
     ⇒ " came back"
```

`prin1` *object* **&optional** *stream* [Function]

この関数は *object*のプリント表現を *stream*に出力する。この関数は`print`のように出力を分割するための改行をプリントしないが、`print`のようにクォート文字を使用する。*object*をリターンする。

```
(progn (prin1 'The\ cat\ in)
       (prin1 "the hat")
       (prin1 " came back"))
     ⊣ The\ cat\ in"the hat"" came back"
     ⇒ " came back"
```

`princ` *object* **&optional** *stream* [Function]

この関数は *object*のプリント表現を *stream*に出力する。*object*をリターンする。

この関数は`read`ではなく人間が読める出力を生成することを意図しているので、クォート文字を挿入せず文字列のコンテンツの前後にダブルクォート文字を配置しない。各呼び出しの間にスペースを何も出力しない。

```
(progn
  (princ 'The\ cat)
  (princ " in the \"hat\""))
     ⊣ The cat in the "hat"
     ⇒ " in the \"hat\""
```

`terpri` **&optional** *stream ensure* [Function]

この関数は *stream*に改行を出力する。名前の由来は “terminate print(プリントを終端する)”。*ensure*が非`nil`の場合には、*stream*がすでに行頭にあれば改行をプリントしない。この場合には *stream*に関数は指定できず、指定するとエラーがシグナルされる。この関数は改行をプリントしたら`t`をリターンする。

`write-char` *character* **&optional** *stream* [Function]

この関数は *character*を *stream*に出力する。*character*をリターンする。

`prin1-to-string` *object* **&optional** *noescape* [Function]

この関数は同じ引数で`prin1`がプリントするテキストを含む文字列をリターンする。

```
(prin1-to-string 'foo)
     ⇒ "foo"
```

```
(prin1-to-string (mark-marker))
     ⇒ "#<marker at 2773 in strings-ja.texi>"
```

*noescape*が非 `nil`なら出力中のクォート文字の使用を抑制する (この引数は Emacs バージョン 19 以降でサポートされた)。

```
(prin1-to-string "foo")
     ⇒ "\"foo\""
(prin1-to-string "foo" t)
     ⇒ "foo"
```

Lisp オブジェクトのプリント表現を文字列として取得する別の手段については、Section 4.7 [Formatting Strings], page 61 の `format`を参照のこと。

`with-output-to-string` *body...* [Macro]

このマクロは出力を文字列に送るよう `standard-output`をセットアップしてフォーム *body* を実行する。その文字列をリターンする。

たとえばカレントバッファー名が‘`foo`’なら、

```
(with-output-to-string
  (princ "The buffer is ")
  (princ (buffer-name)))
```

は`"The buffer is foo"`をリターンする。

`pp` *object* **&optional** *stream* [Function]

この関数は `prin1`と同じように *object*を *stream*に出力するが、より優雅 (pretty) な方法でこれを行う。すなわちこの関数は人間がより読みやすいようにオブジェクトのインデントとパディングを行う。

任意のバイナリーデータを書き込んだり非 POSIX ホストで改行変換を回避するためにこのセクションで説明した関数を使用して batch モードでバイナリー I/O モードを使用する必要がある場合には Section 19.3 [Input Functions], page 334 を参照してください。

19.6 出力に影響する変数

`standard-output` [Variable]

この変数の値はデフォルト出力ストリーム (*stream*引数が `nil`のときプリント関数が使用するストリーム) である。デフォルトは `t`で、これはエコーエリアに表示することを意味する。

`print-quoted` [Variable]

これが非 `nil`なら、省略されたリーダー構文 (たとえば `(quote foo)`を`'foo`、`(function foo)`を`#'foo`のように) を使用してクォートされたフォームをプリントすることを意味する。デフォルトは `t`。

`print-escape-newlines` [Variable]

この変数が非 `nil`なら、文字列内の改行は‘`\n`’、改ページは‘`\f`’でプリントされる。これらの文字は通常は実際の改行と改ページとしてプリントされる。

この変数はクォートつきのプリントを行うプリント関数 `prin1`と `print`に影響を与える。`princ`に影響はない。以下は `prin1`を使用した場合の例である:

```
(prin1 "a\nb")
     ⊣ "a
     ⊣ b"
     ⇒ "a
b"

(let ((print-escape-newlines t))
  (prin1 "a\nb"))
     ⊣ "a\nb"
     ⇒ "a
b"
```

2 つ目の式では prin1を呼び出す間は print-escape-newlinesのローカルバインドが効果をもつが、結果をプリントするときには効果がない。

print-escape-control-characters [Variable]

この変数が非 nilなら、クォートつきでプリントするプリント関数 prin1と printは、文字列内のコントロール文字をバックスラッシュシーケンスとしてプリントする。この変数と print-escape-newlinesがいずれも非 nilなら改行と改頁には後者が優先される。

print-escape-nonascii [Variable]

この変数が非 nilなら、クォートつきでプリントするプリント関数 prin1と printは文字列内のユニバイトの非 ASCII文字を無条件でバックスラッシュシーケンスとしてプリントする。

これらの関数は出力ストリームがマルチバイトバッファー、あるいはマーカーがマルチバイトバッファーをポイントするときは、この変数の値に関わらずユニバイト非 ASCII文字にたいしてバックスラッシュシーケンスを使用する。

print-escape-multibyte [Variable]

この変数が非 nilなら、クォートつきでプリントするプリント関数 prin1と printは、文字列内のマルチバイトの非 ASCII文字を無条件でバックスラッシュシーケンスとしてプリントする。

これらの関数は出力ストリームがユニバイトバッファー、あるいはマーカーがユニバイトバッファーをポイントするときは、この変数の値に関わらずマルチバイト非 ASCII文字にたいしてバックスラッシュシーケンスを使用する。

print-charset-text-property [Variable]

この変数は文字列のプリントにおいてテキストプロパティ‘charset’ のプリントを制御する。値は nil、t、または defaultのいずれか。

値が nilなら charsetテキストプロパティを決してプリントせず、tなら常にプリントする。

値が defaultなら “予期せぬ (unexpected)” charsetプロパティがある場合だけ charsetテキストプロパティをプリントする。ASCII 文字ではすべての charset が “期待された (expected)” ものとみなされる。それ以外なら文字の期待される charsetプロパティは char-charsetにより与えられる。

print-length [Variable]

この変数の値は任意のリスト、ベクター、ブールベクターをプリントする際の最大要素数である。プリントされるオブジェクトがこれより多くの要素をもつ場合は、省略記号 (“...”) で省略される。

値が nil(デフォルト) の場合は無制限。

```
(setq print-length 2)
     ⇒ 2
(print '(1 2 3 4 5))
     ⊣ (1 2 ...)
     ⇒ (1 2 ...)
```

print-level [Variable]
この変数の値はプリント時の丸カッコ (parentheses: “()”) と角カッコ (brackets: “[]"’) のネスト最大深さである。この制限を超える任意のリストとベクターは省略記号 (“...”) で省略される。値 nil(デフォルト) は無制限を意味する。

eval-expression-print-length [User Option]
eval-expression-print-level [User Option]
これらは eval-expressionによって使用される print-lengthと print-levelの値であり、したがって間接的に多くのインタラクティブな評価コマンドにより使用される (Section “Evaluating Emacs-Lisp Expressions” in *The GNU Emacs Manual* を参照)。

以下の変数は循環構造および共有構造の検出と報告に使用されます:

print-circle [Variable]
非 nilなら、この変数はプリント時の循環構造と共有構造の検出を有効にする。Section 2.6 [Circular Objects], page 29 を参照のこと。

print-gensym [Variable]
非 nilなら、この変数はプリント時のインターンされていないシンボル (Section 9.3 [Creating Symbols], page 126 を参照) の検出を有効にする。これが有効なら、インターンされていないシンボルはプレフィックス ‘#:’とともにプリントされる。このプレフィックスは、Lisp リーダーにたいしてインターンされていないシンボルを生成するよう告げる。

print-continuous-numbering [Variable]
非 nilなら、複数のプリント呼び出しを通じて通番が振られることを意味する。これは ‘#*n*=’ラベルと ‘#*m*#’参照にたいしてプリントされる数字に影響する。この変数を setqでセットしてはならない。letを使用して一時的に tにバインドすること。これを行う場合は print-number-tableも nilにバインドすること。

print-number-table [Variable]
この変数は print-circle機能を実装するために、プリント処理で内部的に使用されるベクターを保持する。print-continuous-numberingをバインドするときにこの変数を nilにバインドする以外は、この変数を使用するべきではない。

float-output-format [Variable]
この変数は浮動小数点数をプリントする方法を指定する。デフォルトは nilで、これは情報を失わずにその数値を表せるもっとも短い出力を使用することを意味する。

出力フォーマットをより精密に制御するために、この変数に文字列をセットできる。この文字列には C の sprintf関数で使用される ‘%’指定子をセットする。この変数で使用することのできる制限についての詳細は、この変数のドキュメント文字列を参照のこと。

20 ミニバッファー

ミニバッファー *(minibuffer)* とは、単一の数プレフィックス引数 (numeric prefix argument) より複雑な引数を読み取るために Emacs コマンドが使用する特別なバッファーのことです。これらの引数にはファイル名、バッファー名、(`M-x`での) コマンド名が含まれます。ミニバッファーはフレームの最下行、エコーエリア (Section 39.4 [The Echo Area], page 983 を参照) と同じ場所に表示されますが、引数を読み取るときだけ使用されます。

20.1 ミニバッファーの概要

ほとんどの点においてミニバッファーは普通の Emacs バッファーです。編集コマンドのようなバッファーにたいする操作のほとんどはミニバッファーでも機能します。しかしバッファーを管理する操作の多くはミニバッファーに適用できません。ミニバッファーは常に ‘ `*Minibuf-`*number*`*` ’ という形式の名前をもち変更はできません。ミニバッファーはミニバッファー用の特殊なウィンドウだけに表示されます。これらのウィンドウは常にフレーム最下に表示されます (フレームにミニバッファーウィンドウがないときやミニバッファーウィンドウだけをもつ特殊なフレームもある)。Section 29.9 [Minibuffers and Frames], page 732 を参照してください。

ミニバッファー内のテキストは常にプロンプト文字列 *(prompt string)* で開始されます。これはミニバッファーを使用しているプログラムが、ユーザーにたいしてどのような種類の入力が求められているか告げるために指定するテキストです。このテキストは意図せずに変更してしまわないように、読み取り専用としてマークされます。このテキストは `beginning-of-line`、`forward-word`、`forward-sentence`、`forward-paragraph`を含む特定の移動用関数が、プロンプトと実際のテキストの境界でストップするようにフィールド (Section 32.19.9 [Fields], page 831 を参照) としてもマークされています。

ミニバッファーのウィンドウは通常は 1 行です。ミニバッファーのコンテンツがより多くのスペースを要求する場合には自動的に拡張されます。ミニバッファーのウィンドウがアクティブな間はウィンドウのサイズ変更コマンドで一時的にウィンドウのサイズを変更できます。サイズの変更はミニバッファーを exit したときに通常のサイズにリバートされます。ミニバッファーがアクティブでないときはフレーム内の他のウィンドウでウィンドウのサイズ変更コマンドを使用するか、マウスでモードラインをドラッグして、ミニバッファーのウィンドウのサイズを永続的に変更できます (現実装ではこれが機能するには `resize-mini-windows`が `nil`でなければならない)。フレームがミニバッファーウィンドウだけを含む場合にはフレームのサイズを変更してミニバッファーのサイズを変更できます。

ミニバッファーの使用によって入力イベントが読み取られて、`this-command`や `last-command` のような変数の値が変更されます (Section 21.5 [Command Loop Info], page 385 を参照)。プログラムにそれらを変更させたくない場合は、ミニバッファーを使用するコードの前後でそれらをバインドするべきです。

ある状況下では、アクティブなミニバッファーが存在するときでもコマンドがミニバッファーを使用できます。そのようなミニバッファーは再帰ミニバッファー *(recursive minibuffer)* と呼ばれます。この場合は最初のミニバッファーは ‘ `*Minibuf-1*` ’ という名前になります。再帰ミニバッファーはミニバッファー名の最後の数字を増加することにより命名されます (名前はスペースで始まるので通常のバッファーリストには表示されない)。再帰ミニバッファーが複数ある場合は、最内の (もっとも最近にエンターされた) ミニバッファーがアクティブなミニバッファーになります。このバッファーが、いわゆるミニバッファーと通常は呼ばれるバッファーです。変数 `enable-recursive-minibuffers`、またはコマンドシンボルのその名前のプロパティをセットすることにより再帰ミニバッファーを許可したり禁止できます (Section 20.13 [Recursive Mini], page 373 を参照)。

他のバッファーと同様、ミニバッファーは特別なキーバインドを指定するためにローカルキーマップ (Chapter 22 [Keymaps], page 422 を参照) を使用します。ミニバッファーを呼び出す関数も、処理を行うためにローカルマップをセットアップします。補完なしのミニバッファーローカルマップについては Section 20.2 [Text from Minibuffer], page 343 を参照してください。補完つきのミニバッファーローカルマップについては Section 20.6.3 [Completion Commands], page 355 を参照してください。

ミニバッファーが非アクティブのときのメジャーモードは `minibuffer-inactive-mode`、キーマップは `minibuffer-inactive-mode-map`です。これらは実際にはミニバッファーが別フレームにある場合のみ有用です。Section 29.9 [Minibuffers and Frames], page 732 を参照してください。

Emacs がバッチモードで実行されている場合には、ミニバッファーからの読み取りリクエストは、実装には Emacs 開始時に提供された標準入力記述子から行を読み取ります。これは基本的な入力だけをサポートします。特別なミニバッファーの機能 (ヒストリー、補完など) はバッチモードでは利用できません。

20.2 ミニバッファーでのテキスト文字列の読み取り

ミニバッファー入力にたいする基本的なプリミティブは `read-from-minibuffer`で、これは文字列と Lisp オブジェクトの両方からテキスト表現されたフォームを読み取ることができます。関数 `read-regexp`は特別な種類の文字列である正規表現式 (Section 34.3 [Regular Expressions], page 883 を参照) の読み取りに使用されます。コマンドや変数、ファイル名などの読み取りに特化した関数もあります (Section 20.6 [Completion], page 350 を参照)。

ほとんどの場合でには Lisp 関数の途中でミニバッファー入力関数を呼び出すべきではありません。かわりに `interactive`指定されたコマンドの引数の読み取りの一環として、すべてのミニバッファー入力を行います。Section 21.2 [Defining Commands], page 376 を参照してください。

`read-from-minibuffer` *prompt* **&optional** *initial keymap read history default inherit-input-method* [Function]

この関数はミニバッファーから入力を取得するもっとも一般的な手段である。デフォルトでは任意のテキストを受け入れて、それを文字列としてリターンする。しかし *read*が非 `nil`なら、テキストを Lisp オブジェクトに変換するために `read`を使用する (Section 19.3 [Input Functions], page 334 を参照)。

この関数が最初に行うのはミニバッファーをアクティブにして、プロンプトに *prompt*(文字列でなければならない) を用いてミニバッファーを表示することである。その後にユーザーはミニバッファーでテキストを編集できる。

ミニバッファーを exit するためにユーザーがコマンドをタイプするとき、`read-from-minibuffer`はミニバッファー内のテキストからリターン値を構築する。通常はそのテキストを含む文字列がリターンされる。しかし *read*が非 `nil`なら、`read-from-minibuffer`はテキストを読み込んで結果を未評価の Lisp オブジェクトでリターンする (読み取りについての詳細は See Section 19.3 [Input Functions], page 334 を参照)。

引数 *default*はヒストリーコマンドを通じて利用できるデフォルト値を指定する。値には文字列、文字列リスト、または `nil`を指定する。文字列と文字列リストは、ユーザーが `M-n`で利用可能な “未来のヒストリー (future history)” になる。

*read*が非 `nil`なら、ユーザーの入力が空のときの `read`の入力としても *default*が使用される。*default*が文字列リストの場合には最初の文字列が入力として使用される。*default*が `nil`なら、空の入力は `end-of-file`エラーとなる。しかし通常 (*read*が `nil`) の場合には、ユーザーの入

力が空のとき `read-from-minibuffer`は *default*を無視して空文字列`""`をリターンする。この点ではこの関数はこのチャプターの他のどのミニバッファー入力関数とも異なる。

*keymap*が非`nil`なら、そのキーマップはミニバッファー内で使用されるローカルキーマップとなる。*keymap*が省略または`nil`なら、`minibuffer-local-map`の値がキーマップとして使用される。キーマップの指定は補完のようなさまざまなアプリケーションにたいしてミニバッファーをカスタマイズする、もっとも重要な方法である。

引数 *history*は入力の保存やミニバッファー内で使用されるヒストリーコマンドが使用するヒストリーリスト変数を指定する。デフォルトは`minibuffer-history`。同様にオプションでヒストリーリスト内の開始位置を指定できる。Section 20.4 [Minibuffer History], page 348 を参照のこと。

変数`minibuffer-allow-text-properties`が非`nil`なら、リターンされる文字列にはミニバッファーでのすべてのテキストプロパティが含まれる。それ以外なら、値がリターンされるときすべてのテキストプロパティが取り除かれる。

`minibuffer-prompt-properties`内のテキストプロパティはプロンプトに適用される。このプロパティリストはデフォルトではプロンプトに使用するフェイスを定義する。このフェイスが与えられるとフェイスリストの最後に適用されて表示前にマージされる。

ユーザーがプロンプトの外観を完全に制御したければすべてのフェイスリストの最後に`default`フェイスを指定するのがもっとも簡便な方法である。たとえば:

```
(read-from-minibuffer
 (concat
  (propertize "Bold" 'face '(bold default))
  (propertize " and normal: " 'face '(default))))
```

引数 *inherit-input-method*が非`nil`なら、ミニバッファーにエンターする前にカレントだったバッファーが何であれ、カレントの入力メソッド (Section 33.11 [Input Methods], page 878 を参照)、および`enable-multibyte-characters`のセッティング (Section 33.1 [Text Representations], page 852 を参照) が継承される。

ほとんどの場合、*initial*の使用は推奨されない。非`nil`値の使用は、*history*にたいするコンスセル指定と組み合わせる場合のみ推奨する。Section 20.5 [Initial Input], page 350 を参照のこと。

`read-string` *prompt* **&optional** *initial history default inherit-input-method* [Function]

この関数はミニバッファーから文字列を読み取ってそれをリターンする。引数 *prompt*、*initial*、*history*、*inherit-input-method*は`read-from-minibuffer`で使用する場合と同様。使用されるキーマップは`minibuffer-local-map`。

オプション引数 *default*は`read-from-minibuffer`の場合と同様に使用されるが、ユーザーの入力が空の場合にリターンするデフォルト値も指定する。`read-from-minibuffer`の場合と同様に値は文字列、文字列リスト、または`nil`(空文字列と等価) である。*default*が文字列のときは、その文字列がデフォルト値になる。文字列リストのときは、最初の文字列がデフォルト値になる (これらの文字列はすべて “未来のミニバッファーヒストリー (future minibuffer history)” としてユーザーが利用できる)。

この関数は`read-from-minibuffer`を呼び出すことによって機能する。

```
(read-string prompt initial history default inherit)
≡
(let ((value
       (read-from-minibuffer prompt initial nil nil
                             history default inherit)))
  (if (and (equal value "") default)
      (if (consp default) (car default) default)
    value))
```

`read-regexp` *prompt* **&optional** *defaults history* [Function]

この関数はミニバッファーから文字列として正規表現を読み取ってそれをリターンする。ミニバッファーのプロンプト文字列 *prompt*が‘:’(とその後にオプションの空白文字) で終端されていなければ、この関数はデフォルトのリターン値 (空文字列でない場合。以下参照) の前に‘: ’を付加する。

オプション引数 *defaults*は、入力が空の場合にリターンするデフォルト値を制御する。値は文字列、`nil`(空文字列と等価)、文字列リスト、シンボルのうちのいずれか。

*defaults*がシンボルの場合、`read-regexp`は変数`read-regexp-defaults-function`(以下参照) の値を調べて非`nil`のときは *defaults*よりそちらを優先的に使用する。この場合は値は以下のいずれか:

- `regexp-history-last`。これは適切なミニバッファーヒストリーリスト (以下参照) の最初の要素を使用することを意味する。
- 引数なしの関数。リターン値 (`nil`、文字列、文字列リストのいずれか) が *defaults*の値となる。

これで`read-regexp`が *defaults*を処理した結果はリストに確定する (値が`nil`または文字列の場合は 1 要素のリストに変換する)。このリストにたいして`read-regexp`は以下のような入力として有用な候補をいくつか追加する:

- ポイント位置の単語かシンボル。
- インクリメンタル検索で最後に使用された regexp。
- インクリメンタル検索で最後に使用された文字列。
- 問い合わせつき置換コマンドで最後に使用された文字列またはパターン。

これで関数はユーザー入力を取得するために`read-from-minibuffer`に渡す正規表現のリストを得た。リストの最初の要素は入力が空の場合のデフォルト値である。リストのすべての要素は“未来のミニバッファーヒストリー (future minibuffer history)”となるリスト (see Section “Minibuffer History” in *The GNU Emacs Manual* を参照) としてユーザーが利用可能になる。

オプション引数 *history*が非`nil`なら、それは使用するミニバッファーヒストリーリストを指定するシンボルである (Section 20.4 [Minibuffer History], page 348 を参照)。これが省略または`nil`なら、ヒストリーリストのデフォルトは`regexp-history`となる。

`read-regexp-defaults-function` [User Option]

関数`read-regexp`は、デフォルトの正規表現リストを決定するためにこの変数の値を使用するかもしれない。非`nil`なら、この変数は以下のいずれかである:

- シンボル`regexp-history-last`。
- `nil`、文字列、文字列リストのいずれかをリターンする引数なしの関数。

これらの変数の使い方についての詳細は、上述の`read-regexp`を参照のこと。

`minibuffer-allow-text-properties` [Variable]

この変数が `nil`なら、`read-from-minibuffer`と `read-string`はミニバッファー入力をリターンする前にすべてのテキストプロパティを取り除く。しかし `read-no-blanks-input`(以下参照)、同様に補完つきでミニバッファー入力を行う `read-minibuffer`とそれに関連する関数 (Section 20.3 [Reading Lisp Objects With the Minibuffer], page 347 を参照) は、この変数の値に関わらず無条件でテキストプロパティを破棄する。

`minibuffer-local-map` [Variable]

これはミニバッファーからの読み取りにたいするデフォルトローカルキーマップである。デフォルトでは以下のバインディングをもつ:

C-j	`exit-minibuffer`
RET	`exit-minibuffer`
M-<	`minibuffer-beginning-of-buffer`
C-g	`abort-recursive-edit`
M-n DOWN	`next-history-element`
M-p UP	`previous-history-element`
M-s	`next-matching-history-element`
M-r	`previous-matching-history-element`

`read-no-blanks-input` *prompt* **&optional** *initial* *inherit-input-method* [Function]

この関数はミニバッファーから文字列を読み取るが、入力の一部として空白文字を認めず、そのかわりに空白文字は入力を終端させる。引数 *prompt*、*initial*、*inherit-input-method*は `read-from-minibuffer`で使用するときと同様。

これは関数 `read-from-minibuffer`の簡略化されたインターフェイスであり、キーマップ `minibuffer-local-ns-map`の値を *keymap*引数として `read-from-minibuffer`関数に渡す。キーマップ `minibuffer-local-ns-map`は *C-q*をリバインドしないので、クォートすることによって文字列内にスペースを挿入することが可能である。

`minibuffer-allow-text-properties`の値に関わらず、この関数はテキストプロパティを破棄する。

```
(read-no-blanks-input prompt initial)
≡
(let (minibuffer-allow-text-properties)
  (read-from-minibuffer prompt initial minibuffer-local-ns-map))
```

`minibuffer-local-ns-map` [Variable]

このビルトイン変数は関数 `read-no-blanks-input`内でミニバッファーローカルキーマップとして使用されるキーマップである。デフォルトでは `minibuffer-local-map`のバインディングに加えて、以下のバインディングが有効になる:

SPC	`exit-minibuffer`
TAB	`exit-minibuffer`
?	`self-insert-and-exit`

20.3 ミニバッファーでの Lisp オブジェクトの読み取り

このセクションではミニバッファーで Lisp オブジェクトを読み取る関数を説明します。

`read-minibuffer` *prompt* **&optional** *initial* [Function]

この関数はミニバッファーを使用して Lisp オブジェクトを読み取って、それを評価せずにリターンする。引数 *prompt*と *initial*は `read-from-minibuffer`のときと同様に使用する。

これは `read-from-minibuffer`関数にたいする簡略化されたインターフェイスである。

```
(read-minibuffer prompt initial)
≡
(let (minibuffer-allow-text-properties)
  (read-from-minibuffer prompt initial nil t))
```

以下の例では初期入力として文字列`"(testing)"`を与えている:

```
(read-minibuffer
 "Enter an expression: " (format "%s" '(testing)))

;; 以下はミニバッファーでの表示:

---------- Buffer: Minibuffer ----------
Enter an expression: (testing)⋆
---------- Buffer: Minibuffer ----------
```

ユーザーは `RET`をタイプして初期入力をデフォルトとして利用したり入力を編集することができる。

`eval-minibuffer` *prompt* **&optional** *initial* [Function]

この関数はミニバッファーを使用して Lisp 式を読み取り、それを評価して結果をリターンする。引数 *prompt*と *initial*の使い方は `read-from-minibuffer`と同様。

この関数は `read-minibuffer`の呼び出し結果を単に評価する:

```
(eval-minibuffer prompt initial)
≡
(eval (read-minibuffer prompt initial))
```

`edit-and-eval-command` *prompt form* [Function]

この関数はミニバッファーで Lisp 式を読み取り、それを評価して結果をリターンする。このコマンドと `eval-minibuffer`の違いは、このコマンドでは初期値としての *form*はオプションではなく、テキストの文字列ではないプリント表現に変換された Lisp オブジェクトとして扱われることである。これは `prin1`でプリントされるので、文字列の場合はテキスト初期値内にダブルクォート文字 (‘`"`’) が含まれる。Section 19.5 [Output Functions], page 337 を参照のこと。

以下の例では、すでに有効なフォームであるようなテキスト初期値として式をユーザーに提案している:

```
(edit-and-eval-command "Please edit: " '(forward-word 1))

;; 前の式を評価した後に、
;;    ミニバッファーに以下が表示される:

---------- Buffer: Minibuffer ----------
Please edit: (forward-word 1)⋆
---------- Buffer: Minibuffer ----------
```

すぐに `RET` をタイプするとミニバッファーを exit して式を評価するので、1 単語分ポイントは前進する。

20.4 ミニバッファーのヒストリー

ミニバッファーヒストリーリスト *(minibuffer history list)* は手軽に再利用できるように以前のミニバッファー入力を記録します。ミニバッファーヒストリーリストは、(以前に入力された) 文字列のリストであり、もっとも最近の文字列が先頭になります。

多数のミニバッファーが個別に存在し、異なる入力の種類に使用されます。それぞれのミニバッファー使用にたいして正しいヒストリーリストを指定するのは Lisp プログラマーの役目です。

ミニバッファーヒストリーリストは、`read-from-minibuffer`と`completing-read`のオプション引数 *history*に指定します。以下が利用できる値です:

variable ヒストリーリストとして *variable*(シンボル) を使用する。

(*variable* . *startpos*)
ヒストリーリストとして *variable*(シンボル) を使用して、ヒストリー位置の初期値を *startpos*(負の整数) とみなす。

*startpos*に 0 を指定するのは、単にシンボル *variable*だけを指定するのと等価である。`previous-history-element`はミニバッファー内のヒストリーリストの最新の要素を表示するだろう。正の *startpos*を指定すると、ミニバッファーヒストリー関数は `(elt variable(1- startpos))`がミニバッファー内でカレントで表示されているヒストリー要素であるかのように振る舞う。

一貫性を保つためにミニバッファー入力関数の *initial*引数 (Section 20.5 [Initial Input], page 350 を参照) を使用して、ミニバッファーの初期内容となるヒストリー要素も指定すべきである。

*history*を指定しない場合には、デフォルトのヒストリーリスト`minibuffer-history`が使用されます。他の標準的なヒストリーリストについては以下を参照してください。最初に使用する前に`nil`に初期化するだけで、独自のヒストリーリストを作成することもできます。変数がバッファーローカルなら各バッファーが独自に入力ヒストリーリストを所有することになります。

`read-from-minibuffer`と`completing-read`は、どちらも新たな要素を自動的にヒストリーリストに追加して、ユーザーがそのリストのアイテムを再使用するためのコマンドを提供します。ヒストリーリストを使用するためにプログラムが行う必要があるのはリストの初期化と、使用するときに入力関数にリストの名前を渡すだけです。しかしミニバッファー入力関数がリストを使用していないときに手動でリストを変更しても問題はありません。

新たな要素をヒストリーリストに追加する Emacs 関数は、リストが長くなりすぎたときに古い要素の削除を行うこともできます。変数`history-length`は、ほとんどのヒストリーリストの最大長を指定する変数です。特定のヒストリーリストにたいして異なる最大長を指定するには、そのヒストリーリストのシンボルの`history-length`プロパティにその最大長をセットします。変数`history-delete-duplicates`にはヒストリー内の重複を削除するかどうかを指定します。

`add-to-history` *history-var newelt* **&optional** *maxelt keep-all* [Function]
この関数は *newelt*が空文字列でなければ、それを新たな要素として変数 *history-var*に格納されたヒストリーリストに追加して、更新されたヒストリーリストをリターンする。これは *maxelt*か`history-length`がが非`nil`なら、リストの長さをその変数の値に制限する (以下参照)。*maxelt*に指定できる値の意味は`history-length`の値と同様。*history-var*はレキシカル変数を参照できない。

`add-to-history`は通常は`history-delete-duplicates`が非`nil`ならば、ヒストリーリスト内の重複メンバーを削除する。しかし *keep-all*が非`nil`なら、それは重複を削除しないことを意味し、たとえ *newelt*が空でもリストに追加する。

`history-add-new-input` [Variable]
この変数の値が `nil`なら、ミニバッファーから読み取りを行う標準的な関数はヒストリーリストに新たな要素を追加しない。これにより Lisp プログラムが `add-to-history`を使用して明示的に入力ヒストリーを管理することになる。デフォルト値は `t`。

`history-length` [User Option]
この変数の値は、最大長を独自に指定しないすべてのヒストリーリストの最大長を指定する。値が `t`なら最大長がない (古い要素を削除しない) ことを意味する。ヒストリーリスト変数のシンボルの `history-length`プロパティが非 `nil`なら、その特定のヒストリーリストにたいする最大長として、そのプロパティ値がこの変数をオーバーライドする。

`history-delete-duplicates` [User Option]
この変数の値が `t`なら、それは新たなヒストリー要素の追加時に以前からある等しい要素が削除されることを意味する。

以下は標準的なミニバッファーヒストリーリスト変数です:

`minibuffer-history` [Variable]
ミニバッファーヒストリー入力にたいするデフォルトのヒストリーリスト。

`query-replace-history` [Variable]
`query-replace`の引数 (と他のコマンドの同様の引数) にたいするヒストリーリスト。

`file-name-history` [Variable]
ファイル名引数にたいするヒストリーリスト。

`buffer-name-history` [Variable]
バッファー名引数にたいするヒストリーリスト。

`regexp-history` [Variable]
正規表現引数にたいするヒストリーリスト。

`extended-command-history` [Variable]
拡張コマンド名引数にたいするヒストリーリスト。

`shell-command-history` [Variable]
シェルコマンド引数にたいするヒストリーリスト。

`read-expression-history` [Variable]
評価されるための Lisp 式引数にたいするヒストリーリスト。

`face-name-history` [Variable]
フェイス引数にたいするヒストリーリスト。

`custom-variable-history` [Variable]
`read-variable`が読み取る変数名引数にたいするヒストリーリスト。

20.5 入力の初期値

ミニバッファー入力にたいする関数のいくつかには、*initial*と呼ばれる引数があります。これは通常のように空の状態で開始されるのではなく、特定のテキストとともにミニバッファーが開始されることを指定しますが、ほとんどの場合においては推奨されない機能です。

*initial*が文字列なら、ミニバッファーはその文字列のテキストを含む状態で開始され、ユーザーがそのテキストの編集を開始するとき、ポイントはテキストの終端にあります。ユーザーがミニバッファーを exit するために単に `RET`をタイプした場合には、この入力文字列の初期値をリターン値だと判断します。

*initial*にたいして非 `nil`値の使用には反対します。なぜなら初期入力は強要的なインターフェイスだからです。ユーザーにたいして有用なデフォルト入力を提案するためには、ヒストリーリストやデフォルト値の提供のほうがより有用です。

しかし *initial*引数にたいして文字列を指定すべき状況が 1 つだけあります。それは *history*引数にコンスセルを指定したときです。Section 20.4 [Minibuffer History], page 348 を参照してください。

*initial*は `(`*`string . position`*`)`という形式をとることもできます。これは *string*をミニバッファーに挿入するが、その文字列のテキスト中の *position*にポイントを配置するという意味です。

歴史的な経緯により、*position*は異なる関数の間で実装が統一されていません。`completing-read`では *position*の値は 0 基準です。つまり値 0 は文字列の先頭、1 は最初の文字の次、... を意味します。しかし `read-minibuffer`、およびこの引数をサポートする補完を行わない他のミニバッファー入力関数では、1 は文字列の先頭、2 は最初の文字の次、... を意味します。

*initial*の値としてのコンスセルの使用は推奨されません。

20.6 補完

補完 *(complete, ompletion)* は省略された形式から始まる名前の残りを充填する機能です。補完はユーザー入力と有効な名前リストを比較して、ユーザーが何をタイプしたかで名前をどの程度一意に判定できるか判断することによって機能します。たとえば *C-x b* (`switch-to-buffer`) とタイプしてからスイッチしたいバッファー名の最初の数文字をタイプして、その後に `TAB` (`minibuffer-complete`) をタイプすると、Emacs はその名前を可能な限り展開します。

標準的な Emacs コマンドはシンボル、ファイル、バッファー、プロセスの名前にたいする補完を提案します。このセクションの関数により、他の種類の名前にたいしても補完を実装できます。

`try-completion`関数は補完にたいする基本的なプリミティブです。これは初期文字列にたいして文字列セットをマッチして、最長と判定された補完をリターンします。

関数 `completing-read`は補完にたいする高レベルなインターフェイスを提供します。`completing-read`の呼び出しによって有効な名前リストの判定方法が指定されます。その後にこの関数は補完にたいして有用ないくつかのコマンドにキーバインドするローカルキーマップとともに、ミニバッファーをアクティブ化します。その他の関数は特定の種類の名前を補完つきで読み取る、簡便なインターフェイスを提供します。

20.6.1 基本的な補完関数

以下の補完関数は、その関数自身ではミニバッファーで何も行いません。ここではミニバッファーを使用する高レベルの補完機能とともに、これらの関数について説明します。

`try-completion` *string collection* **&optional** *predicate* [Function]

この関数は *collection*内の *string*に可能なすべての補完の共通する最長部分文字列をリターンする。

*collection*は補完テーブル *(completion table)* と呼ばれる。値は文字列リスト、コンスセル、obarray、ハッシュテーブル、または補完関数でなければならない。

`try-completion`は補完テーブルにより指定された許容できる補完それぞれにたいして、*string* と比較を行う。許容できる補完マッチが存在しなければ`nil`をリターンする。マッチする補完が 1 つだけで、それが完全一致ならば`t`をリターンする。それ以外は、すべてのマッチ可能な補完に共通する最長の初期シーケンスをリターンする。

*collection*がリストなら、許容できる補完 (permissible completions) はそのリストの要素によって指定される。リストの要素は文字列、または CAR が文字列、または (`symbol-name`によって文字列に変換される) シンボルであるようなコンスセルである。リストに他の型の要素が含まれる場合は無視される。

*collection*が obarray(Section 9.3 [Creating Symbols], page 126 を参照) なら、その obarray 内のすべてのシンボル名が許容できる補完セットを形成する。

*collection*がハッシュテーブルの場合には、文字列かシンボルのキーが利用可能な補完となる。他のキーは無視される。

*collection*として関数を使用することもできる。この場合にはその関数だけが補完を処理する役目を担う。つまり`try-completion`は、この関数が何をリターンしようともそれをリターンする。この関数は *string*、*predicate*、`nil`の 3 つの引数で呼び出される (3 つ目の引数は同じ関数を`all-completions`でも使用して、どちらの場合でも適切なことを行うため)。Section 20.6.7 [Programmed Completion], page 363 を参照のこと。

引数 *predicate*が非`nil`の場合には、*collection*がハッシュテーブルなら 1 引数、それ以外は 2 引数の関数でなければならない。これは利用可能なマッチのテストに使用され、マッチは *predicate*が非`nil`をリターンしたときだけ受け入れられる。*predicate*に与えられる引数は文字列、alist のコンスセル (CAR が文字列)、または obarray のシンボル (シンボル名ではない) のいずれか。*collection*がハッシュテーブルなら、*predicate*は文字列キー (string key) と連想値 (associated value) の 2 引数で呼び出される。

これらに加えて許容され得るためには、補完は `completion-regexp-list`内のすべての正規表現にもマッチしなければならない。(*collection*が関数なら、その関数自身が`completion-regexp-list`を処理する必要がある)。

以下の 1 つ目の例では、文字列 ‘`foo`’が alist のうち 3 つの CAR とマッチされている。すべてのマッチは文字 ‘`fooba`’で始まるので、それが結果となる。2 つ目の例では可能なマッチは 1 つだけで、しかも完全一致なのでリターン値は`t`になる。

```
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4)))
     ⇒ "fooba"

(try-completion "foo" '(("barfoo" 2) ("foo" 3)))
     ⇒ t
```

以下の例では文字 ‘`forw`’で始まるシンボルが多数あり、それらはすべて単語 ‘`forward`’で始まる。ほとんどのシンボルはその後に ‘`-`’が続くが、すべてではないので ‘`forward`’までしか補完できない。

```
(try-completion "forw" obarray)
     ⇒ "forward"
```

最後に以下の例では述語`test`に渡される利用可能なマッチは 3 つのうち 2 つだけである (文字列 ‘`foobaz`’は短すぎる)。これらは両方とも文字列 ‘`foobar`’で始まる。

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ "foobar"
```

`all-completions` *string collection* **&optional** *predicate* [Function]

この関数は *string*の利用可能な補完すべてのリストをリターンする。この関数の引数は `try-completion`の引数と同じであり、`try-completion`が行うのと同じ方法で `completion-regexp-list`を使用する。

*collection*か関数なら *string*、*predicate*、`t`の 3 つの引数で呼び出される。この場合はその関数がリターンするのが何であれ、`all-completions`はそれをリターンする。Section 20.6.7 [Programmed Completion], page 363 を参照のこと。

以下の例は `try-completion`の例の関数 `test`を使用している。

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test

(all-completions
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ ("foobar1" "foobar2")
```

`test-completion` *string collection* **&optional** *predicate* [Function]

この関数は *string*が *collection*と *predicate*で指定された有効な補完候補なら `nil`をリターンする。引数は `try-completion`の引数と同じ。たとえば *collection*が文字列リストなら、*string* がリスト内に存在して、かつ *predicate*を満足すれば true となる。

この関数は `try-completion`が行うのと同じ方法で `completion-regexp-list`を使用する。

*predicate*が非 `nil`で *collection*が同じ文字列を複数含む場合には、`completion-ignore-case`にしたがって `compare-strings`で判定してそれらすべてをリターンするか、もしくは何もリターンしない。それ以外では `test-completion`のリターン値は基本的に予測できない。

*collection*が関数の場合は *string*、*predicate*、`lambda`の 3 つの引数で呼び出される。それが何をリターンするにせよ `test-completion`はそれをリターンする。

`completion-boundaries` *string collection predicate suffix* [Function]

この関数はポイントの前のテキストが *string*、ポイントの後が *suffix*と仮定して、*collection* が扱うフィールドの境界 (boundary) をリターンする。

補完は通常は文字列 (string) 全体に作用するので、すべての普通のコレクション (collection) にたいして、この関数は常に `(0 . (length` *`suffix`*`))`をリターンするだろう。しかしファイルにたいする補完などの、より複雑な補完は 1 回に 1 フィールド行われる。たとえばたとえ`"/usr/share/doc"`が存在しても、`"/usr/sh"`の補完に`"/usr/share/"`は含まれるが、`"/usr/share/doc"`は含まれないだろう。また`"/usr/sh"`にたいする `all-completions`に`"/usr/share/"`は含まれず、`"share/"`だけが含まれるだろう。*string*が`"/usr/sh"`、*suffix* が`"e/doc"`なら、`completion-boundaries`は `(5 . 1)`をリターンするだろう。これは *collection*が`"/usr/"`の後ろにあり`"/doc"`の前にある領域に関する補完情報だけをリターンするであろうことを告げている。

補完 alist を変数に格納した場合は、変数の `risky-local-variable` プロパティに非 `nil` をセットして、その変数が risky(危険) だとマークすること。Section 12.12 [File Local Variables], page 197 を参照のこと。

`completion-ignore-case` [Variable]
この変数の値が非 `nil` なら、補完での case(大文字小文字) の違いは意味をもたない。`read-file-name` では、この変数は `read-file-name-completion-ignore-case` (Section 20.6.5 [Reading File Names], page 359 を参照) にオーバーライドされる。`read-buffer` では、この変数は `read-buffer-completion-ignore-case` (Section 20.6.4 [High-Level Completion], page 357 を参照) にオーバーライドされる。

`completion-regexp-list` [Variable]
これは正規表現のリストである。補完関数はこのリスト内のすべての正規表現にマッチした場合のみ許容できる補完と判断する。`case-fold-search` (Section 34.2 [Searching and Case], page 883 を参照) では `completion-ignore-case` の値にバインドされる。

`lazy-completion-table` *var fun* [Macro]
この変数は変数 *var* を補完のための collection として lazy(lazy: 力のない、だらけさせる、のろのろした、怠惰な、不精な、眠気を誘う) な方法で初期化する。ここで lazy とは、collection 内の実際のコンテンツを必要になるまで計算しないという意味。このマクロは *var* に格納する値の生成に使用する。*var* を使用して最初に補完を行ったとき、真の値が実際に計算される。これは引数なしで *fun* を呼び出すことにより行われる。*fun* がリターンする値は *var* の永続的な値となる。

以下は例:

```
(defvar foo (lazy-completion-table foo make-my-alist))
```

既存の補完テーブルを受け取って変更したバージョンをリターンする関数がいくつかあります。`completion-table-case-fold` は大文字小文字を区別しない、case-insensitive なテーブルをリターンします。`completion-table-in-turn` と `completion-table-merge` は、複数の入力テーブルを異なる方法で組み合わせます。`completion-table-subvert` はテーブルを異なる初期プレフィックス (initial prefix) で変更します。`completion-table-with-quoting` はクォートされたテキストの処理に適したテーブルをリターンします。`completion-table-with-predicate` は述語関数 (predicate function) によるフィルタリングを行います。`completion-table-with-terminator` は終端文字列 (terminating string) を追加します。

20.6.2 補完とミニバッファー

このセクションでは補完つきでミニバッファーから読み取るための、基本的なインターフェイスを説明します。

`completing-read` *prompt collection* **&optional** *predicate require-match initial history default inherit-input-method* [Function]
この関数は補完の提供によりユーザーを支援して、ミニバッファーから文字列を読み取る。*prompt* (文字列でなければならない) のプロンプトとともにミニバッファーをアクティブ化する。

実際の補完は補完テーブル *collection* と補完述語 *predicate* を関数 `try-completion` (Section 20.6.1 [Basic Completion], page 350 を参照) に渡すことにより行われる。これは補完の使用されるローカルキーマップに特定のコマンドをバインドしたとき発生する。これらのコマンドのいくつかは `test-completion` も呼び出す。したがって *predicate* が非 `nil` なら、

*collection*と `completion-ignore-case`が矛盾しないようにすること。[Definition of test-completion], page 352 を参照されたい。

*collection*が関数のときの詳細な要件は Section 20.6.7 [Programmed Completion], page 363 を参照のこと。

オプション引数 *require-match*の値はユーザーがミニバッファーを exit する方法を決定する。

- `nil`なら、通常のミニバッファー exit コマンドはミニバッファーの入力と無関係に機能する。
- `t`なら、入力が *collection*の要素に補完されるまで通常のミニバッファー exit コマンドは機能しない。
- `confirm`なら、どのような入力でもユーザーは exit できるが、入力が `confirm`の要素に補完されていなければ確認を求められる。
- `confirm-after-completion`なら、どのような入力でもユーザーは exit できるが、前のコマンドが補完コマンド (たとえば `minibuffer-confirm-exit-commands`の中のコマンドのいずれか) で、入力の結果が *collection*の要素でなければ確認を求められる。Section 20.6.3 [Completion Commands], page 355 を参照のこと。
- *require-match*にたいする他の値は `t`と同じだが、exit コマンドは補完処理中は exit しない。

しかし *require-match*の値に関わらず、空の入力は常に許容される。この場合 `completing-read`は *default*がリストなら最初の要素、*default*が `nil`なら`""`、または *default*をリターンする。文字列と *default*内の文字列はヒストリーコマンドを通じてユーザーが利用できる。

関数 `completing-read`は *require-match*が`nil`ならキーマップとして `minibuffer-local-completion-map`を、*require-match*が非 `nil`なら `minibuffer-local-must-match-map`を使用する。Section 20.6.3 [Completion Commands], page 355 を参照のこと。

引数 *history*は入力の保存とミニバッファーヒストリーコマンドに、どのヒストリーリスト変数を使用するか指定する。デフォルトは `minibuffer-history`。Section 20.4 [Minibuffer History], page 348 を参照のこと。

*initial*はほとんどの場合は推奨されない。*history*にたいするコンスセル指定と組み合わせた場合のみ非 `nil`値の使用を推奨する。Section 20.5 [Initial Input], page 350 を参照のこと。デフォルト入力にたいしてはかわりに *default*を使用すること。

引数 *inherit-input-method*が非 `nil`なら、ミニバッファーにエンターする前にカレントだったバッファーが何であれ、カレントの入力メソッド (Section 33.11 [Input Methods], page 878 を参照)、および `enable-multibyte-characters`のセッティング (Section 33.1 [Text Representations], page 852 を参照) が継承される。

変数 `completion-ignore-case`が非 `nil`なら、利用可能なマッチにたいして入力を比較するときの補完は case を区別しない。Section 20.6.1 [Basic Completion], page 350 を参照のこと。このモードでの操作では、*predicate*も case を区別してはならない (さもないと驚くべき結果となるであろう)。

以下は `completing-read`を使用した例:

```
(completing-read
 "Complete a foo: "
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 nil t "fo")
```

```
;; 前の式を評価後に、
;;    ミニバッファーに以下が表示される:

---------- Buffer: Minibuffer ----------
Complete a foo: fo*
---------- Buffer: Minibuffer ----------
```

その後ユーザーが *DEL DEL b RET*をタイプすると、`completing-read`は `barfoo`をリターンする。

`completing-read`関数は、実際に補完を行うコマンドの情報を渡すために変数をバインドする。これらの変数は以降のセクションで説明する。

`completing-read-function` [Variable]
この変数の値は関数でなければならず、補完つきの読み取りを実際に行うために `completing-read`から呼び出される。この関数は `completing-read`と同じ引数を受け入れる。他の関数のバインドして通常の `completing-read`の振る舞いを完全にオーバーライドすることができる。

20.6.3 補完を行うミニバッファーコマンド

このセクションでは補完のためにミニバッファーで使用されるキーマップ、コマンド、ユーザーオプションを説明します。

`minibuffer-completion-table` [Variable]
この変数の値はミニバッファー内の補完に使用される補完テーブル (Section 20.6.1 [Basic Completion], page 350 を参照)。これは `completing-read`が `try-completion`に渡す補完テーブルを含むグローバル変数。`minibuffer-complete-word`のようなミニバッファー補完コマンドにより使用される。

`minibuffer-completion-predicate` [Variable]
この変数の値は `completing-read`が `try-completion`に渡す述語 (predicate) である。この変数は他のミニバッファー補完関数にも使用される。

`minibuffer-completion-confirm` [Variable]
この変数はミニバッファーを exit する前に Emacs が確認を求めるかどうかを決定する。`completing-read`はこの変数をバインドして、exit する前に関数 `minibuffer-complete-and-exit`がこの値をチェックする。値が `nil`なら確認は求められない。値が `confirm`の場合は、入力が有効な補完候補でなくてもユーザーは exit するかもしれないが Emacs は確認を求めない。値が `confirm-after-completion`の場合、入力が有効な補完候補でなくてもユーザーは exit するかもしれないが、ユーザーが `minibuffer-confirm-exit-commands`内の任意の補完コマンドの直後に入力を確定した場合には Emacs は確認を求める。

`minibuffer-confirm-exit-commands` [Variable]
この変数には、`completing-read`の引数 *require-match*が `confirm-after-completion`のときにミニバッファー exit 前に Emacs に確認を求めさせるコマンドのリストが保持されている。このリスト内のコマンドを呼び出した直後にユーザーがミニバッファーの exit を試みると Emacs は確認を求める。

`minibuffer-complete-word` [Command]
この関数はせいぜい 1 つの単語からミニバッファーを補完する。たとえミニバッファーのコンテンツが 1 つの補完しかもたない場合でも、`minibuffer-complete-word`はその単語に属さない最初の文字を超えた追加はしない。Chapter 35 [Syntax Tables], page 906 を参照のこと。

`minibuffer-complete` [Command]
この関数は可能な限りミニバッファーのコンテンツを補完する。

`minibuffer-complete-and-exit` [Command]
この関数はミニバッファーのコンテンツを補完して確認が要求されない場合 (たとえば `minibuffer-completion-confirm`が `nil`のとき) は exit する。確認が要求される場合には、このコマンドを即座に繰り返すことによって確認が行われないようにする。このコマンドは 2 回連続で実行された場合は確認なしで機能するようにプログラムされている。

`minibuffer-completion-help` [Command]
この関数はカレントのミニバッファーのコンテンツで利用可能な補完のリストを作成する。これは `all-completions`の引数 *collection*に変数 `minibuffer-completion-table`の値、引数 *predicate*に `minibuffer-completion-predicate`の値を使用して呼び出すことによって機能する。補完リストは`*Completions*`と呼ばれるバッファーのテキストとして表示される。

`display-completion-list` *completions* [Function]
この関数は `standard-output`内のストリーム (通常はバッファー) に *completions*を表示する (ストリームについての詳細は Chapter 19 [Read and Print], page 331 を参照)。引数 *completions*は通常は `all-completions`がリターンする補完リストそのものだが、そうである必要はない。要素はシンボルか文字列で、どちらも単にプリントされる。文字列 2 つのリストでもよく、2 つの文字列が結合されたかのようにプリントされる。この場合、1 つ目の文字列は実際の補完で、2 つ目の文字列は注釈の役目を負う。

この関数は `minibuffer-completion-help`より呼び出される。一般的には以下のように `with-output-to-temp-buffer`とともに使用される。

```
(with-output-to-temp-buffer "*Completions*"
  (display-completion-list
    (all-completions (buffer-string) my-alist)))
```

`completion-auto-help` [User Option]
この変数が非 `nil`なら、次の文字が一意でなく決定できないために補完が完了しないときは常に、補完コマンドは利用可能な補完リストを自動的に表示する。

`minibuffer-local-completion-map` [Variable]
`completing-read`の値は、補完の 1 つが完全に一致することを要求されないときにローカルキーマップとして使用される。デフォルトではこのキーマップは以下のバインディングを作成する:

?	`minibuffer-completion-help`
SPC	`minibuffer-complete-word`
TAB	`minibuffer-complete`

親キーマップとして `minibuffer-local-map`を使用する ([Definition of minibuffer-local-map], page 346 を参照)。

`minibuffer-local-must-match-map` [Variable]
`completing-read`は、1 つの補完の完全な一致が要求されないときのローカルキーマップとしてこの値を使用する。したがって `exit-minibuffer`にキーがバインドされていなければ、無条件にミニバッファーを exit する。デフォルトでは、このキーマップは以下のバインディングを作成する:

C-j	`minibuffer-complete-and-exit`

```
RET       minibuffer-complete-and-exit
```

親キーマップは`minibuffer-local-completion-map`を使用する。

`minibuffer-local-filename-completion-map` [Variable]
これは単に SPCを非バインドする sparse キーマップ (sparse: 疎、希薄、まばら) を作成する。これはファイル名にスペースを含めることができるからである。関数`read-file-name`は、このキーマップと`minibuffer-local-completion-map`か`minibuffer-local-must-match-map`のいずれかを組み合わせる。

`minibuffer-beginning-of-buffer-movement` [Variable]
非`nil`の場合には、*M-<*コマンドはポイントがプロンプト終端の後ならポイントをプロンプト終端に移動する。ポイントがプロンプト終端またはプロンプト終端より前ならバッファーの先頭に移動する。この変数が`nil`なら *M-<*は`beginning-of-buffer`のように振る舞う。

20.6.4 高レベルの補完関数

このセクションでは特定の種類の名前を補完つきで読み取る便利な高レベル関数を説明します。

ほとんどの場合は、Lisp 関数の中盤でこれらの関数を呼び出すべきではありません。可能なときは`interactive`指定の内部で呼び出して、ミニバッファーのすべての入力をコマンドの引数読み取りの一部にします。Section 21.2 [Defining Commands], page 376 を参照してください。

`read-buffer` *prompt* **&optional** *default require-match predicate* [Function]
この関数はバッファーの名前を読み取ってそれを文字列でリターンする。プロンプトは *prompt*。引数 *default*はミニバッファーが空の状態でユーザーが exit した場合にリターンされるデフォルト名として使用される。非`nil`なら文字列、文字列リスト、またはバッファーを指定する。リストならリストの先頭の要素がデフォルト値になる。デフォルト値はプロンプトに示されるが、初期入力としてミニバッファーには挿入されない。

引数 *prompt*はコロンかスペースで終わる文字列である。*default*が非`nil`なら、この関数はデフォルト値つきでミニバッファーから読み取る際の慣習にしたがってコロンの前の *prompt*の中にこれを挿入する。

オプション引数 *require-match*は`completing-read`のときと同じ。Section 20.6.2 [Minibuffer Completion], page 353 を参照のこと。

オプション引数 *predicate*が非`nil`なら、それは考慮すべきバッファーをフィルターする関数を指定する。この関数は可能性のある候補を引数として呼び出されて、候補を拒絶するなら`nil`、許容するなら非`nil`をリターンすること。

以下の例ではユーザーが‘`minibuffer.t`’とエンターしてから、`RET`をタイプしている。引数 *require-match*は`t`であり、与えられた入力で始まるバッファー名は‘`minibuffer.texi`’だけなので、その名前が値となる。

```
(read-buffer "Buffer name: " "foo" t)
;; 前の式を評価した後、
;;    空のミニバッファーに
;;    以下のプロンプトが表示される:

---------- Buffer: Minibuffer ----------
Buffer name (default foo): ⋆
---------- Buffer: Minibuffer ----------
```

```
;; ユーザーが minibuffer.t RETとタイプする
     ⇒ "minibuffer.texi"
```

`read-buffer-function` [User Option]
この変数が非`nil`なら、それはバッファー名を読み取る関数を指定する。`read-buffer`は通常行うことを行うかわりに、`read-buffer`と同じ引数でその関数を呼び出す。

`read-buffer-completion-ignore-case` [User Option]
この変数が非 non-`nil`なら、バッファー名の読み取りの補完処理において`read-buffer`は case を無視する。

`read-command` *prompt* **&optional** *default* [Function]
この関数はコマンドの名前を読み取って、Lisp シンボルとしてそれをリターンする。引数 *prompt* は`read-from-minibuffer`で使用される場合と同じ。それが何であれ`commandp`が`t`をリターンすればコマンドであり、コマンド名とは`commandp`が`t`をリターンするシンボルだということを思い出してほしい。Section 21.3 [Interactive Call], page 382 を参照のこと。

引数 *default*はユーザーが null 入力をエンターした場合に何をリターンするか指定する。シンボル、文字列、文字列リストを指定できる。文字列なら`read-command`はリターンする前にそれを intern する。リストなら`read-command`はリストの最初の要素を intern する。*default*が`nil`ならデフォルトが指定されなかったことを意味する。その場合には、もしユーザーが null 入力をエンターするとリターン値は`(intern "")`、つまり名前が空文字列でプリント表現が`##`であるようなシンボル (Section 2.4.4 [Symbol Type], page 14 を参照)。

```
(read-command "Command name? ")

;; 前の式を評価した後に、
;;    空のミニバッファーに以下のプロンプトが表示される:

---------- Buffer: Minibuffer ----------
Command name?
---------- Buffer: Minibuffer ----------
```

ユーザーが *forward-c RET*とタイプすると、この関数は`forward-char`をリターンする。

`read-command`関数は`completing-read`の簡略化されたインターフェイスである。実在する Lisp 変数のセットを補完するために変数`obarray`、コマンド名だけを受け入れるために述語`commandp`を使用する。

```
(read-command prompt)
≡
(intern (completing-read prompt obarray
                         'commandp t nil))
```

`read-variable` *prompt* **&optional** *default* [Function]
この変数はカスタマイズ可能な変数の名前を読み取って、それをシンボルとしてリターンする。引数の形式は `read-command`の引数と同じ。この関数は `commandp`のかわりに`custom-variable-p`を述語に使用する点を除いて`read-command`と同様に振る舞う。

`read-color` **&optional** *prompt convert allow-empty display* [Command]
この関数はカラー指定 (カラー名、または`#RRRGGGBBB`のような形式の RGB16 進値) の文字列を読み取る。これはプロンプトに *prompt*(デフォルトは`"Color (name or #RGB triplet):"`)

を表示して、カラー名にたいする補完を提供する (16 進 RGB 値は補完しない)。標準的なカラー名に加えて、補完候補にはポイント位置のフォアグラウンドカラーとバックグラウンドカラーが含まれる。

Valid RGB values are described in Section 29.22 [Color Names], page 748.

この関数のリターン値はミニバッファー内でユーザーがタイプした文字列である。しかしインタラクティブに呼び出されたとき、またはオプション引数 *convert*が非 `nil`なら、入力されたカラー名のかわりにそれに対応する RGB 値文字列をリターンする。この関数は入力として有効なカラー指定を求める。*allow-empty*が非 `nil`でユーザーが null 入力をエンターした場合は空のカラー名が許容される。

インタラクティブに呼び出されたとき、または *display*が非 `nil`なら、エコーエリアにもリターン値が表示される。

Section 33.10.4 [User-Chosen Coding Systems], page 870 の関数 `read-coding-system` と `read-non-nil-coding-system`、および Section 33.11 [Input Methods], page 878 の `read-input-method-name`も参照されたい。

20.6.5 ファイル名の読み取り

高レベル補完関数`read-file-name`、`read-directory-name`、`read-shell-command`はそれぞれファイル名、ディレクトリー名、シェルコマンドを読み取るようにデザインされています。これらはデフォルトディレクトリーの自動挿入を含む特別な機能を提供します。

`read-file-name` *prompt* **&optional** *directory default require-match initial predicate* [Function]

この関数はプロンプト *prompt*とともに補完つきでファイル名を読み取る。

例外として以下のすべてが真ならば、この関数はミニバッファーのかわりにグラフィカルなファイルダイアログを使用してファイル名を読み取る:

1. マウスコマンドを通じて呼び出された。
2. グラフィカルなディスプレイ上の選択されたフレームがこの種のダイアログをサポートしている。
3. 変数`use-dialog-box`が非`nil`の場合。Section “Dialog Boxes” in *The GNU Emacs Manual* を参照のこと。
4. *directory*引数 (以下参照) がリモートファイルを指定しない場合。Section “Remote Files” in *The GNU Emacs Manual* を参照のこと。

グラフィカルなファイルダイアログを使用したときの正確な振る舞いはプラットホームに依存する。ここでは単にミニバッファーを使用したときの振る舞いを示す。

`read-file-name`はリターンするファイル名を自動的に展開しない。絶対ファイル名が必要ならば自分で`expand-file-name`を呼び出すことができる。

オプション引数 *require-match*は `completing-read`のときと同じ。Section 20.6.2 [Minibuffer Completion], page 353 を参照のこと。

引数 *directory*は、相対ファイル名の補完に使用するディレクトリーを指定する。値は絶対ディレクトリー名。変数 `insert-default-directory`が非 `nil`なら、初期入力としてミニバッファーに *directory*も挿入される。デフォルトはカレントバッファーの `default-directory`の値。

*initial*を指定すると、それはミニバッファーに挿入する初期ファイル名になる (*directory*が挿入された場合はその後に挿入される)。この場合、ポイントは *initial*の先頭に配置される。*initial*

のデフォルト値は nil(ファイル名を挿入しない)。*initial*が何を行うか確認するには、ファイルを visit しているバッファーで *C-x C-v*を試すとよい。注意**:** ほとんどの場合は *initial*よりも *default*の使用を推奨する。

*default*が非 nilなら、最初に read-file-nameが挿入したものと等しい空以外のコンテンツを残してユーザーがミニバッファーを exit すると、この関数は *default*をリターンする。insert-default-directoryが非 nilならそれがデフォルトとなるので、ミニバッファーの初期コンテンツは常に空以外になる。*require-match*の値に関わらず *default*の有効性はチェックされない。とはいえ *require-match*が非 nilなら、ミニバッファーの初期コンテンツは有効なファイル名 (またはディレクトリー名) であるべきだろう。それが有効でなければ、ユーザーがそれを編集せずに exit すると read-file-nameは補完を試みて、*default*はリターンされない。*default*はヒストリーコマンドからも利用できる。

*default*が nilなら、read-file-nameはその場所に代用するデフォルトを探そうと試みる。この代用デフォルトは明示的に *default*にそれが指定されたかのように、*default*とまったく同じ方法で扱われる。*default*が nilでも *initial*が非 nilなら、デフォルトは *directory*と *initial*から得られる絶対ファイル名になる。*default*と *initial*の両方が nilで、そのバッファーがファイルを visit しているバッファーなら、read-file-nameはそのファイルの絶対ファイル名をデフォルトとして使用する。バッファーがファイルを visit していなければデフォルトは存在しない。この場合はユーザーが編集せずに RETをタイプすると、read-file-nameは前にミニバッファーに挿入されたコンテンツを単にリターンする。

空のミニバッファー内でユーザーが RETをタイプすると、この関数は *require-match*の値に関わらず空文字列をリターンする。たとえばユーザーが *M-x set-visited-file-name*を使用して、カレントバッファーをファイルを visit していないことにするために、この方法を使用している。

*predicate*が非 nilなら、それは補完候補として許容できるファイル名を決定する 1 引数の関数である。*predicate*が関数名にたいして非 nilをリターンすれば、それはファイル名として許容できる値である。

以下は read-file-nameを使用した例:

```
(read-file-name "The file is ")

;; 前の式を評価した後に、
;;    ミニバッファーに以下が表示される:

---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/⋆
---------- Buffer: Minibuffer ----------
```

*manual TAB*をタイプすると以下がリターンされる:

```
---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/manual.texi⋆
---------- Buffer: Minibuffer ----------
```

ここでユーザーが RETをタイプすると、read-file-nameは文字列"/gp/gnu/elisp/manual.texi"をファイル名としてリターンする。

read-file-name-function [Variable]

非 nilなら、read-file-nameと同じ引数を受け取る関数である。read-file-nameが呼び出されたとき、read-file-nameは通常の処理を行なうかわりに与えられた引数でこの関数を呼び出す。

`read-file-name-completion-ignore-case` [User Option]
この変数が非 `nil`なら、`read-file-name`は補完を行なう際に case を無視する。

`read-directory-name` *prompt* **&optional** *directory default require-match initial* [Function]
この関数は `read-file-name`と似ているが補完候補としてディレクトリーだけを許す。

*default*が `nil`で *initial*が非 `nil`なら、`read-directory-name`は *directory* (*directory*が `nil`ならカレントバッファーのデフォルトディレクトリー) と *initial*を組み合わせて代用のデフォルトを構築する。この関数は *default*と *initial*の両方が `nil`なら *directory*、*directory*も `nil`ならカレントバッファーのデフォルトディレクトリーを代用のデフォルトとして使用する。

`insert-default-directory` [User Option]
この変数は `read-file-name`により使用されるため、ファイル名を読み取るほとんどのコマンドにより間接的に使用される (これらのコマンドにはコマンドのインタラクティブフォームに'`f`'や'`F`'のコードレター (code letter)) をふくむすべてのコマンドが含まれる。Section 21.2.2 [Code Characters for interactive], page 378 を参照されたい)。この変数の値は、(もしあれば) デフォルトディレクトリー名をミニバッファー内に配置して `read-file-name`を開始するかどうかを制御する。変数の値が `nil`なら、`read-file-name`はミニバッファーに初期入力を何も配置しない (ただし *initial*引数で初期入力を指定しない場合)。この場合には依然としてデフォルトディレクトリーが相対ファイル名の補完に使用されるが表示はされない。

この変数が `nil`でミニバッファーの初期コンテンツが空なら、ユーザーはデフォルト値にアクセスするために次のヒストリー要素を明示的にフェッチする必要があるだろう。この変数が非 `nil`ならミニバッファーの初期コンテンツは常に空以外となり、ミニバッファーで編集をおこなわず即座に `RET`をタイプすることによって、常にデフォルト値を要求できる (上記参照)。

たとえば:

```
;; デフォルトディレクトリーとともにミニバッファーが開始
(let ((insert-default-directory t))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is ~lewis/manual/⋆
---------- Buffer: Minibuffer ----------

;; ミニバッファーはプロンプトだけで空
;;    appears on its line.
(let ((insert-default-directory nil))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is ⋆
---------- Buffer: Minibuffer ----------
```

`read-shell-command` *prompt* **&optional** *initial history* **&rest** *args* [Function]
この関数はプロンプト *prompt*とインテリジェントな補完を提供して、ミニバッファーからシェルコマンドを読み取る。これはコマンド名にたいして適切な候補を使用してコマンドの最初の単語を補完する。コマンドの残りの単語はファイル名として補完する。

この関数はミニバッファー入力にたいするキーマップとして `minibuffer-local-shell-command-map`を使用する。*history*引数は使用するヒストリーリストを指定する。省略または `nil`の場合のデフォルトは `shell-command-history` (Section 20.4 [Minibuffer History], page 348 を参照)。オプション引数 *initial*はミニバッファーの初期コンテンツを指定する (Section 20.5 [Initial Input], page 350 を参照)。もしあれば残りの *args*は `read-from-minibuffer`内の *default*と *inherit-input-method*として使用される (Section 20.2 [Text from Minibuffer], page 343 を参照)。

`minibuffer-local-shell-command-map` [Variable]

このキーマップは `read-shell-command`により、コマンドとシェルコマンドの一部となるファイル名の補完のために使用される。これは親キーマップとして `minibuffer-local-map`を使用して、`TAB`を `completion-at-point`にバインドする。

20.6.6 補完変数

補完のデフォルト動作を変更するために使用される変数がいくつかあります。

`completion-styles` [User Option]

この変数の値は補完を行うために使用される補完スタイル (シンボル) である。補完スタイル *(completion style)* とは、補完を生成するためのルールセットのこと。このリストにあるシンボルはそれぞれ、`completion-styles-alist`内に対応するエントリーをもたなければならない。

`completion-styles-alist` [Variable]

この変数には補完スタイルのリストが格納される。リスト内の各要素は以下の形式をもつ

```
(style try-completion all-completions doc)
```

ここで *style*は補完スタイルの名前 (シンボル) であり、そのスタイルを参照するために変数 `completion-styles`内で使用されるかもしれない。*try-completion*は補完を行なう関数で、*all-completions*補完をリストする関数、*doc*は補完スタイルを説明する文字列である。

関数 *try-completion*と *all-completions*は *string*、*collection*、*predicate*、*point*の 4 つの引数をとる。引数 *string*、*collection*、*predicate*の意味は `try-completion` (Section 20.6.1 [Basic Completion], page 350 を参照) のときと同様。引数 *point*は *string*内のポイント位置。各関数は自身の処理を行ったら非 `nil`、行わなかった場合 (たとえば補完スタイルに一致するように *string*を行う方法がない場合) は `nil`をリターンする。

ユーザーが `minibuffer-complete` (Section 20.6.3 [Completion Commands], page 355 を参照) のような補完コマンドを呼び出すと、Emacs は `completion-styles`に最初にリストされたスタイルを探して、そのスタイルの *try-completion*関数を呼び出す。この関数が `nil`をリターンしたら、Emacs は次にリストされた補完スタイルに移動してそのスタイルの *try-completion*関数を呼び出すといったように、*try-completion*関数の 1 つが補完の処理に成功して非 `nil`値をリターンするまで順次これを行なう。同様の手順は *all-completions*関数を通じて補完のリストにも行われる。

利用できる補完スタイルについては Section “Completion Styles” in *The GNU Emacs Manual* を参照のこと。

`completion-category-overrides` [User Option]

この変数は特別な補完スタイルと、特定の種類のテキスト補完時に使用するその他の補完動作を指定する。この変数の値は `(category . alist)`という形式の要素をもつような alist である。*category*は何が補完されるかを記述するシンボルで、現在のところカテゴリーに `buffer`、

`file`、`unicode-name`が定義されているが、これに特化した補完関数 (Section 20.6.7 [Programmed Completion], page 363 を参照) を通じて他のカテゴリーを定義できる。*alist*はそのカテゴリーにたいして補完がどのように振る舞うべきかを記述する連想リスト。alist のキーとして以下がサポートされる:

`styles` 値は補完スタイル (シンボル) のリスト。

`cycle` 値はそのカテゴリーにたいする `completion-cycle-threshold` (Section "Completion Options" in *The GNU Emacs Manual* を参照) の値。

将来、さらに alist エントリーが定義されるかもしれない。

`completion-extra-properties` [Variable]

この変数はカレント補完コマンドの特別なプロパティの指定に使用される。この変数は補完に特化したコマンドにより let バインドされることを意図している。値はプロパティ/値ペアーのリスト。以下のプロパティがサポートされる:

`:annotation-function`
値は補完バッファー内に注釈 (annotation) を加える関数。この関数は引数 completion を 1 つ受け取り `nil`、または補完の隣に表示する文字列をリターンしなければならない。

`:exit-function`
値は補完を行った後に実行する関数。この関数は 2 つの引数 *string*と *status*を受け取る。*string*は補完されたフィールドのテキストで、*status*は行われた操作の種類を示す。操作の種類はテキストの補完が完了したなら `finished`、それ以上補完できないが補完が完了していなければ `sole`、有効な補完だがさらに補完できるときは `exact`となる。

20.6.7 プログラムされた補完

意図した利用可能な補完のすべてを含む alist か obarray を事前に作成するのが不可能または不便なことがあります。このような場合は与えられた文字列にたいする補完を計算するために独自の関数を提供できます。これはプログラム補完 *(programmed completion)* と呼ばれます。Emacs は数あるケースの中でも特にファイル名の補完 (Section 25.9.6 [File Name Completion], page 563 を参照) でプログラム補完を使用しています。

この機能を使用するためには、関数を `completing-read`の *collection*引数として渡します。関数 `completing-read`はその補完関数が `try-completion`、`all-completions`などの基本的な補完関数に渡されて、その関数がすべてを行えるよう取り計らいます。

補完関数は 3 つの引数を受け取ります:

- 補完される文字列。
- 利用可能なマッチをフィルターする述語関数。もしなければ `nil`。関数は利用可能なマッチにたいしてこの述語 (predicate) を呼び出して、述語が `nil`をリターンしたらそのマッチを無視する。
- 実行する補完操作のタイプを指定するフラグ。Section 20.6.1 [Basic Completion], page 350 を参照のこと。以下の値のうちいずれか 1 つを指定する:

  `nil` これは `try-completion`を指定する。マッチがなければ関数は `nil`をリターンすること。指定された文字列が一意でかつ正確にマッチしたら `t`をリターンすること。それ以外ならすべてのマッチに共通な最長の前置部分文字列をリターンすること。

`t` `all-completions`を指定する。関数は指定された文字列の利用可能なすべての補完のリストをリターンする。

`lambda` `test-completion`を指定する。関数は指定された文字列がいくつかの補完候補に完全一致するなら`t`、それ以外は`nil`をリターンする。

`(boundaries . `*suffix*`)`
`completion-boundaries`を指定する。関数は`(boundaries `*start* ` . `*end*`)`をリターンする。ここで*start*は指定された文字列内の境界の開始位置、*end*は*suffix*内の境界の終了位置。

`metadata` カレント補完の状態に関する情報の要求を指定する。リターン値は`(metadata . `*alist*`)`の形式をもち、*alist*は以下で説明する要素をもつ連想配列。

フラグに他の値が指定されたら、補完関数は`nil`をリターンする。

以下は`metadata`フラグ引数への応答として補完関数がリターンするかもしれない metadata エントリーのリストです:

`category` 値は補完関数が補完を試みているテキストの種類を説明するシンボル。シンボルが`completion-category-overrides`内のキーの 1 つにマッチする場合、通常の補完動作はオーバーライドされる。Section 20.6.6 [Completion Variables], page 362 を参照のこと。

`annotation-function`
値は補完に注釈 *(annotation)* を付ける関数。この関数は 1 つの引数 *string*を受け取り、これは利用可能な補完である。リターン値は文字列で、`*Completions*`バッファー内の補完 *string*の後に表示される。

`display-sort-function`
値は補完をソートする関数。関数は 1 つの引数をとる。これは補完文字列のリストで、ソートされた補完文字列リストがリターンされる。その入力のリストは破壊的に変更することが許容される。

`cycle-sort-function`
値は`completion-cycle-threshold`が非`nil`、かつユーザーが補完候補を巡回するときに補完をソートする関数。引数のリストとリターン値は`display-sort-function`と同様。

`completion-table-dynamic` *function* **&optional** *switch-buffer* [Function]
この関数はプログラムされた補完関数として動作可能な関数を記述する便利な方法である。引数 *function*は 1 つの引数 (文字列) をとる関数であり、利用可能な補完すべてを含む補完テーブル (Section 20.6.1 [Basic Completion], page 350 を参照) をリターンする。*function*がリターンするテーブルには文字列引数にマッチしない要素を含めることもできる。これらは`completion-table-dynamic`によって自動的にフィルターされる。特に *function*は引数を無視して利用可能なすべての補完の完全なリストをリターンできる。`completion-table-dynamic`を *function*とプログラムされた補完関数との間の変換器として考えることができる。

オプション引数 *switch-buffer*が非`nil`、かつ補完がミニバッファーで行われた場合、*function*はそのミニバッファーにエンターしたときのバッファーをカレントバッファーにセットして呼び出される。

`completion-table-dynamic`のリターン値は`try-completion`および`all-completions`の 2 つ目の引数として使用できる。この関数は常に空のメタデータと無意味な境界をリターンすることに注意 (Section 20.6.7 [Programmed Completion], page 363 を参照)。

`completion-table-with-cache` *function* **&optional** *ignore-case* [Function]
これは前回の引数/結果ペアーを保存する `completion-table-dynamic`にたいするラッパーである。これは同じ引数にたいする複数回の検査に必要なのが、1 回の *function*呼び出しだけであることを意味する。これは外部プロセス呼び出しなど、処理が低速のとき有用かもしれない。

20.6.8 通常バッファーでの補完

補完は通常はミニバッファー内で行われますが、補完機能は通常の Emacs バッファー内のテキストにも使用できます。多くのメジャーモードで、コマンド *C-M-i*または *M-TAB*によってバッファー内補完が行われ、それらは `completion-at-point`にバインドされています。Section “Symbol Completion” in *The GNU Emacs Manual* を参照してください。このコマンドはアブノーマルフック変数 `completion-at-point-functions`を使用します:

`completion-at-point-functions` [Variable]
このアブノーマルフックの値は関数のリスト。これらの関数はポイント位置のテキストの補完にたいする補完テーブルの計算に使用される (Section 20.6.1 [Basic Completion], page 350 を参照)。これはメジャーモードによるモード固有の補完テーブル (Section 23.2.1 [Major Mode Conventions], page 464 を参照) の提供に使用できる。

コマンド `completion-at-point`が実行されると引数なしでリスト内の関数が 1 つずつ呼び出される。それぞれの関数はポイント位置のテキストにたいして補完テーブルを生成でき、かつそれに責任を負いたいのでなければ `nil`をリターンすること。それ以外なら以下の形式のリストをリターンすること:

```
(start end collection . props)
```

ここで *start*と *end*は補完する (ポイントを取り囲む) テキストの区切りである。*collection* はそのテキストを補完する補完テーブルであり、`try-completion` (Section 20.6.1 [Basic Completion], page 350 を参照) の 2 つ目の引数として渡すのに適した形式である。補完候補は `completion-styles` (Section 20.6.6 [Completion Variables], page 362 を参照) で定義された補完スタイルを通じて、この補完テーブルを通常の方法で使用して生成されるだろう。*props*は追加の情報のためのプロパティリストである。`completion-extra-properties`内のすべてのプロパティ(Section 20.6.6 [Completion Variables], page 362 を参照) と、以下の追加のプロパティが認識される:

`:predicate`
値は補完候補が満足する必要がある述語。

`:exclusive`
値が `no`の場合は、もし補完テーブルがポイント位置のテキストのマッチに失敗したなら、補完の失敗を報告するかわりに `completion-at-point`は `completion-at-point-functions`内の次の関数へ移動する。

このフック上の関数は (たとえば `post-command-hook`から) 頻繁に呼び出され得るので一般的には素早くリターンすること。補完リストの生成が高価な処理なら *collection*にたいする関数の提供を強く推奨する。Emacs は `completion-at-point-functions`内の関数を頻繁に呼び出すかもしれないが、それらの呼び出しのいくつかにたいしてのみ *collection*の値を考慮する。*collection*にたいして関数を提供することにより Emacs は必要になるまで補完の生成を遅延できる。ラッパー関数を作成するために `completion-table-dynamic`を使用できる:

```
;; このパターンは避けて
(let ((beg ...) (end ...) (my-completions (my-make-completions)))
  (list beg end my-completions))
```

```
;; かわりに以下を使用する
(let ((beg ...) (end ...))
  (list beg
        end
        (completion-table-dynamic
          (lambda (_)
            (my-make-completions)))))
```

さらに一般的には *collection*は *start*と *end*の間のカレントのテキストにもとづいて事前にフィルターされるべきではない。なぜなら使用を判断した補完スタイルに応じてこれを行うのは `completion-at-point-functions`の呼び出し側の責任だからである。

`completion-at-point-functions`内の関数も上述のリストのかわりに関数をリターンするかもしれない。その場合には引数なしでリターンされた関数が呼び出されて、その関数が補完処理の全責任を負う。この方法は推奨されない。これは `completion-at-point`を使用する古いコードの救済だけを意図したものだからである。

非 `nil`値を最初にリターンした `completion-at-point-functions`内の関数が、`completion-at-point`によって使用される。残りの関数は呼び出されない。例外は上述の `:exclusive`指定があるとき。

以下の関数は Emacs バッファー内の任意に拡張されたテキストにたいして便利な補完方法を提供します:

`completion-in-region` *start end collection* **&optional** *predicate* [Function]

この関数は *collection*を使用してカレントバッファー内の位置 *start*と *end*の間のテキストを補完する。引数 *collection*は `try-completion` (Section 20.6.1 [Basic Completion], page 350 を参照) のときと同じ意味をもつ。

この関数は補完テキストを直接カレントバッファーに挿入する。`completing-read` (Section 20.6.2 [Minibuffer Completion], page 353 を参照) とは異なり、ミニバッファーをアクティブにしない。

この関数が機能するためには、ポイントが *start*と *end*の間になければならない。

20.7 Yes-or-No による問い合わせ

このセクションではユーザーに yes-or-no の確認を求める関数を説明します。関数 `y-or-n-p`は 1 文字での応答に使用できます。この関数は不注意による誤った答えが深刻な結果を招かない場合に有用です。`yes-or-no-p`は 3 文字から 4 文字の答えを要求するので、より重大な問いに適しています。

3 つの関数はいずれもマウスを使用して呼び出されたコマンドの場合、より正確には `last-nonmenu-event` (Section 21.5 [Command Loop Info], page 385 を参照) が `nil`かリストの場合は、問いに答えるためにダイアログボックスまたはポップアップメニューを使用します。それ以外の場合はキーボード入力を使用します。呼び出しの周囲で `last-nonmenu-event`に適切な値をバインドすることにより、マウスあるいはキーボードの使用を強制できます。

`yes-or-no-p`と `y-or-n-p`はどちらもミニバッファーを使用します。

`y-or-n-p` *prompt* [Function]

この関数はユーザーに答えを尋ねてミニバッファーに入力を求める。ユーザーが `y`をタイプしたら `t`、`n`をタイプしたら `nil`をリターンする。この関数は yes の意味で `SPC`、no の意味で `DEL`も受け入れる。quit として *C-g*と *C-]*も受け入れる。これは問いがミニバッファーを使用して、かつミニバッファーを抜けるためにユーザーが *C-]*の使用を試みるかもしれないということが

理由。応答は 1 文字であり、問いを終了させるための RETは必要ない。大文字と小文字は等価である。

“答えを尋ねる” とはミニバッファーに *prompt*、その後に文字列 ‘(y or n) ’ をプリントすることを意味する。期待される答え (*y*、*n*、SPC、DEL、もしくは質問を終了するその他のキー) 以外が入力されると、この関数は ‘Please answer y or n.’と応答して繰り返し答えの入力を要求する。

この関数は実際にはミニバッファーを使用するが答えの編集を許容しない。答えを求めている間、カーソルはモイニバッファーに移動される。

答えとその意味は、たとえ ‘y’と ‘n’であっても固定されたものではなく、キーマップ query-replace-mapによって指定される (Section 34.7 [Search and Replace], page 902 を参照)。特にユーザーが recenter、scroll-up、scroll-down、scroll-other-window、scroll-other-window-down(それぞれ query-replace-map内で *C-l*、*C-v*、*M-v*、*C-M-v*、*C-M-S-v*にバインドされている) のような特殊な応答をエンターした場合、この関数はは指定されたウィンドウの再センタリングやスクロール操作を処理してから再度答えを求める。

y-or-n-p-with-timeout *prompt seconds default* [Function]

y-or-n-pと同様だがユーザーが *seconds*秒以内に答えないと、この関数は待つのをやめて *default*をリターンする。これはタイマーをセットアップすることによって機能する。引数 *seconds* は数字である。

yes-or-no-p *prompt* [Function]

この関数は質問してミニバッファーに答えの入力を求める。これはユーザーが ‘yes’をエンターすると t、‘no’をエンターすると nilをリターンする。ユーザーは応答を終えるために RETをタイプしなければならない。大文字と小文字は等価。

yes-or-no-pはミニバッファーに *prompt*とその後に ‘(yes or no) ’ を表示することによって開始される。ユーザーは期待される応答の 1 つをタイプしなければならない。それ以外の答えなら、この関数は ‘Please answer yes or no.’と応答して約 2 秒待った後に要求を繰り返す。

yes-or-no-pは y-or-n-pより多くの作業をユーザーに要求するので、より重大な決定に適している。

以下は例:

```
(yes-or-no-p "Do you really want to remove everything? ")

;; 前の式を評価した後、
;;    空のミニバッファーに
;;    以下のプロンプトが表示される:

---------- Buffer: minibuffer ----------
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

ユーザーが最初に *y RET*とタイプしたら無効になる。なぜならこの関数は ‘yes’という単語全体を要求しているので、一時停止して以下のプロンプトを説明のために表示する。

```
---------- Buffer: minibuffer ----------
Please answer yes or no.
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

20.8 複数の問いを尋ねる

このセクションではより複雑な質問や複数の似かよった質問をユーザーに尋ねる機能を説明します。

同じような連続する質問と答えがある場合、たとえば各バッファーにたいして順に “Do you want to save this buffer?” と確認を求めるような場合には、個別に質問するより `map-y-or-n-p`を使用して質問のコレクションを尋ねるべきです。これはユーザーにたいして、質問全体にたいして 1 回で答えられるような便利な機能を提供します。

`map-y-or-n-p` *prompter actor list* **&optional** *help action-alist no-cursor-in-echo-area* [Function]

この関数はユーザーに一連の質問をし、それぞれの質問にたいしてエコーエリア内の 1 文字の答えを読み取る。

値 *list*は質問をするオブジェクトを指定する。これはリスト、オブジェクト、または生成関数 (generator function) のいずれかである。関数の場合は引数なしで次に質問するオブジェクト、または質問の中止を意味する `nil`のいずれかをリターンする。

引数 *prompter*は各質問について問い合わせ方法を指定する。*prompter*が文字列なら質問テキストは以下のようになる:

```
(format prompter object)
```

ここで *object*は、(*list*から得られる) 質問する次のオブジェクトである。

文字列でないければ、*prompter*は 1 つの引数 (質問する次のオブジェクト) をとる関数で、質問テキストをリターンする。値が文字列ならユーザーに問う質問であること。関数は `t`(ユーザーに尋ねずこのオブジェクトを処理する)、または `nil`(ユーザーに尋ねずこのオブジェクトを無視する) をリターンすることもできる。

引数 *actor*はユーザーが与えた答えにたいして、どのように処理するかを指定する。これは引数が 1 つの関数で、ユーザーが yes と答えたオブジェクトを引数として呼び出される。引数は常に *list*から取得したオブジェクトである。

引数 *help*が与えられたら、それは以下の形式のリストである:

```
(singular plural action)
```

*singular*はそのオブジェクトが概念的に何に作用するかを説明する単数形の名詞を含む文字列、*plural*はそれに対応する複数形の名詞、*action*は *actor*が何を行うかを説明する他動詞である。

*help*を指定しない場合のデフォルトは `("object" "objects" "act on")`。

質問のたびに、ユーザーはそのオブジェクトを処理するなら *y*、*Y*または SPC、そのオブジェクトをスキップするなら *n*、*N*、または DEL、以降のすべてのオブジェクトを処理するなら *!*、exit(以降のすべてのオブジェクトをスキップ) するなら ESCか *q*、カレントオブジェクトを処理した後に exit するなら `.`(ピリオド)、ヘルプを入手する場合は *C-h*をエンターする。これらは `query-replace`が受け入れるのと同じ答えである。キーマップ `query-replace-map`は `map-y-or-n-p`にたいするそれらの意味を定義して、`query-replace`にたいしても同様に定義する。Section 34.7 [Search and Replace], page 902 を参照のこと。

*action-alist*を使用して、利用できる追加の答えとそれらが何を意味するかを指定できる。これは要素が `(char function help)`という形式の alist で、それぞれの要素が追加の答えを 1 つ定義する。要素の内容は *char*が文字 (答え)、*function*が引数が 1 つ (*list*から取得するオブジェクト) の関数、*help*が文字列である。

ユーザーの応答が *char*なら、`map-y-or-n-p`は *function*を呼び出す。これが非 `nil`をリターンしたらそのオブジェクトが処理されたと判断して、`map-y-or-n-p`は *list*内の次のオブジェクトに進む。`nil`をリターンしたら同じオブジェクトにたいして質問を繰り返す。

確認を求める間、map-y-or-n-pは通常は cursor-in-echo-areaをバインドする。しかし *no-cursor-in-echo-area*が非 nilならバインドしない。

マウスを使用して呼び出されたコマンドから map-y-or-n-pが呼び出された場合 (より正確には last-nonmenu-eventは非 nilかリストの場合。Section 21.5 [Command Loop Info], page 385 を参照) には、確認を求めるためにダイアログボックスかポップアップメニューが使用される。この場合にはキーボード入力やエコーエリアは使用されない。呼び出しの前後で last-nonmenu-eventを適切な値にバインドすることによって、マウスあるいはキーボードの入力を強制できる。

map-y-or-n-pのリターン値は処理したオブジェクトの個数である。

3 つ以上の答えをもつかもしれない質問をユーザーに尋ねる必要がある場合には read-answerを使用してください。

read-answer *question answers* [Function]

この関数は *question*のテキスト ('SPC'文字で終端されていること) とともにユーザーに入力を求める。この関数は *question*に *answers*を追加することにより、プロンプト内に可能な応答を含めることができる。この可能な応答は以下の形式の要素をもつ alist として *answers*内に提供される。

```
(long-answer short-answer help-message)
```

*long-answer*はユーザーの応答の完全なテキスト (文字列)、*short-answer*は同じ応答の短い形式 (単一文字かファンクションキー)、*help-message*はその応答の意味を説明するテキスト。変数 read-answer-shortが非 nilなら可能な応答の短いバージョンをプロンプトに表示して、ユーザーがプロンプトに表示された 1 文字をタイプすることを期待する。それ以外なら可能な応答の長いバージョンをプロンプトに表示して、ユーザーにはプロンプトに表示された完全なテキストのいずれかを入力してから、入力完了で RETを押下することが期待される。RETが非 nilかつこの関数がマウスイベントから呼び出された場合には問いと答えは GUI のダイアログボックス内に表示される。

この関数はプロンプトに表示された応答の長短やユーザーがタイプした応答とは無関係に、ユーザーがセンタクした *long-answer*のテキストをリターンする。

以下はこの関数の使用例:

```
(let ((read-answer-short t))
  (read-answer "Foo "
     '(("yes"  ?y "perform the action")
       ("no"   ?n "skip to the next")
       ("all"  ?! "perform for the rest without more questions")
       ("help" ?h "show help")
       ("quit" ?q "exit"))))
```

read-char-from-minibuffer *prompt* **&optional** *chars history* [Function]

この関数はミニバッファーを使用して単一文字を読み取りリターンする。オプションで許容する文字のリスト *chars*以外のメンバーはすべて無視する。*history*引数は使用するヒストリーリストシンボルを指定する。これが省略か nilなら、この関数はヒストリーを使用しない。

20.9 パスワードの読み取り

他のプログラムに渡すためのパスワードを読み取るために関数 read-passwdを使用できます。

`read-passwd` *prompt* **&optional** *confirm default* [Function]
この関数はプロンプト *prompt*を表示してパスワードを読み取る。これはユーザーがタイプしたパスワードのかわりに、パスワード内の各文字を '*'に変更してエコーする。パスワードを隠すために別の文字を適用したければ、その文字を `read-hide-char`に let バインドすること。
オプション引数 *confirm*が非 `nil`なら、パスワードを 2 回読み取ることでそれらが同じものであることを強制する。同じでなければ、2 回の入力が同じになるまで、ユーザーはパスワードを繰り返しタイプする必要がある。
オプション引数 *default*は、ユーザーが空入力をエンターした場合のデフォルトパスワードである。*default*が `nil`なら、`read-passwd`は null 文字列をリターンする。

20.10 ミニバッファーのコマンド

このセクションではミニバッファー内で使用するコマンドを説明します。

`exit-minibuffer` [Command]
このコマンドはアクティブなミニバッファーを exit する。これは通常はミニバッファー内のローカルキーマップのキーにバインドされる。

`self-insert-and-exit` [Command]
このコマンドはキーボードでタイプされた最後の文字を挿入した後にアクティブなミニバッファーを exit する。Section 21.5 [Command Loop Info], page 385) を参照のこと。

`previous-history-element` *n* [Command]
このコマンドは *n*個前 (古い) のヒストリー要素の値でミニバッファー内のコンテンツを置換する。

`next-history-element` *n* [Command]
このコマンドはミニバッファー内のポイントの前のカレントコンテンツを、*n*個分のより最近のヒストリー要素の値で置換する。現在位置を超えたヒストリー内の位置への移動も可能であり、それは "将来のヒストリー (future history)" を呼び出す (Section 20.2 [Text from Minibuffer], page 343 を参照)。

`previous-matching-history-element` *pattern n* [Command]
このコマンドは *pattern*(正規表現) にマッチする *n*個前 (古い) のヒストリー要素でミニバッファー内のコンテンツを置換する。

`next-matching-history-element` *pattern n* [Command]
このコマンドは *pattern*(正規表現) にマッチする *n*個先 (新しい) のヒストリー要素でミニバッファー内のコンテンツを置換する。

`previous-complete-history-element` *n* [Command]
このコマンドはミニバッファー内のポイントの前のカレントコンテンツを、*n*個前 (古い) ヒストリー要素の値で置換する。

`next-complete-history-element` *n* [Command]
このコマンドはミニバッファー内のポイントの前のカレントコンテンツを、*n*個先 (新しい) ヒストリー要素の値で置換する。

`goto-history-element` *nabs* [Command]
この関数はミニバッファーにミニバッファーヒストリーの要素を配置する。引数 *nabs*は絶対ヒストリー位置 (absolute history position) を降順で指定する。0 ならカレント要素、正の数 *n* は *n*個前の要素を意味する。負の数-*n*は *n*番目の "将来のヒストリー" を意味する。

20.11 ミニバッファーのウィンドウ

以下の関数はミニバッファーウィンドウにアクセスと選択を行い、それがアクティブかどうかテストしてリサイズされる方法を制御します。

`minibuffer-window` **&optional** *frame* [Function]
この関数はフレーム *frame*にたいして使用されるミニバッファーウィンドウをリターンする。*frame*が `nil`ならそれは選択されたフレームを意味する。

フレームが使用するミニバッファーウィンドウがフレームの一部である必要はないことに注意(ミニバッファーをもたないフレームは必要に応じて別のフレームのミニバッファーウィンドウを使用する)。ミニバッファーのないフレームのミニバッファーウィンドウは、そのフレームの `minibuffer`フレームパラメーターをセットすることにより変更できる (Section 29.4.3.5 [Buffer Parameters], page 721 を参照)。

`set-minibuffer-window` *window* [Function]
この関数はミニバッファーウィンドウとして *window*を使用するよう指定する。これは通常のミニバッファーコマンドを呼び出さずにミニバッファーにテキストを入力する場合には、ミニバッファーをどこに表示するかに影響を及ぼす。通常のミニバッファー入力関数はすべて選択されたフレームに対応するミニバッファーを選択して開始されるので影響はない。

`window-minibuffer-p` **&optional** *window* [Function]
この関数は *window*がミニバッファーウィンドウなら `t`をリターンする。*window*のデフォルトは選択されたウィンドウ。

以下の関数はカレントでアクティブなミニバッファーを表示しているウィンドウをリターンします。

`active-minibuffer-window` [Function]
この関数はカレントでアクティブなミニバッファーのウィンドウ、アクティブなミニバッファーがなければ `nil`をリターンする。

`(minibuffer-window)`の結果の比較だけでは、与えられたウィンドウがカレントでアクティブなミニバッファーを表示するかどうかを判断するのは不十分である。なぜなら複数のフレームがある場合にはミニバッファーウィンドウも複数あり得る。

`minibuffer-window-active-p` *window* [Function]
この関数は *window*がカレントでアクティブなミニバッファーを表示するウィンドウなら非 `nil`をリターンする。

以下の 2 つのオプションはミニバッファーウィンドウが自動的にリサイズされるか否か、およびその処理でミニバッファーが大きくなり得るサイズを制御します。

`resize-mini-windows` [User Option]
このオプションはミニバッファーウィンドウが自動的にリサイズされるかどうかを指定する。デフォルト値は `grow-only`で、これはミニバッファーが表示するテキストを収容するために、ミニバッファーウィンドウがデフォルトにより自動的に拡張されて、ミニバッファーが空になれば即座に 1 行に縮小されることを意味する。値が `t`なら Emacs は常にミニバッファーが表示するテキストにミニバッファーウィンドウの高さをフィットさせるように試みる。値が `nil`ならミニバッファーウィンドウのサイズが自動的に変更されることは決してない。この場合には、ミニバッファーウィンドウの高さの調節のために、ウィンドウリサイズコマンドを使用できる (Section 28.4 [Resizing Windows], page 617 を参照)。

`max-mini-window-height` [User Option]
: このオプションはミニバッファーウィンドウの自動的なリサイズにたいする最大高さを提供する。浮動小数点数はフレーム高さにたいする割合、整数は最大行数を指定する。デフォルト値は 0.25。

上記の 2 つの変数の値は表示時に効果を発揮するので、エコーエリアのメッセージを生成するコードの前後でこれらの変数を let バインドしても機能しないことに注意してください。長いメッセージを表示する際にミニバッファーウィンドウのリサイズを抑止したければ、かわりに `message-truncate-lines`変数をバインドしてください (Section 39.4.4 [Echo Area Customization], page 988 を参照)。

オプション `resize-mini-windows`はミニバッファーのみのフレームの振る舞いには影響を与えません (Section 29.3.1 [Frame Layout], page 702 を参照)。以下のオプションによりそのようなフレームも同じように自動的にリサイズすることが可能になります。

`resize-mini-frames` [User Option]
: これが `nil`ならミニバッファーのみのフレームは決して自動的にリサイズされない。

  これが関数なら、その関数はリサイズするミニバッファーのみフレームを単一の引数として呼び出される。この関数が呼び出されるときは、そのフレームのミニバッファーウィンドウのバッファーは、そのウィンドウの次回再表示時にコンテンツが表示されるバッファーである。この関数には何らかの適切な方法によりフレームをバッファーにフィットさせることが期待される。

  それ以外の非 `nil`値は `fit-frame-to-buffer`を呼び出すことによりミニバッファーのみのフレームをリサイズすることを意味する (Section 28.4 [Resizing Windows], page 617 を参照)。

20.12 ミニバッファーのコンテンツ

以下の関数はミニバッファーのプロンプトとコンテンツにアクセスします。

`minibuffer-prompt` [Function]
: この関数はカレントでアクティブなミニバッファーのプロンプト文字列をリターンする。アクティブなミニバッファーがなければ `nil`をリターンする。

`minibuffer-prompt-end` [Function]
: この関数はミニバッファーがカレントならミニバッファープロンプトの終端のカレント位置をリターンする。それ以外はバッファーの有効な最小位置をリターンする。

`minibuffer-prompt-width` [Function]
: この関数はミニバッファーがカレントならミニバッファープロンプトのカレントの表示幅をリターンする。それ以外は 0 をリターンする。

`minibuffer-contents` [Function]
: この関数はミニバッファーがカレントなら、ミニバッファーの編集可能なコンテンツ (つまりプロンプト以外のすべて) を文字列でリターンする。それ以外はカレントバッファーのコンテンツ全体をリターンする。

`minibuffer-contents-no-properties` [Function]
: これは `minibuffer-contents`と同様だが、テキストプロパティをコピーせずに文字自身だけをリターンする。Section 32.19 [Text Properties], page 813 を参照のこと。

`delete-minibuffer-contents` [Command]
このコマンドはミニバッファーがカレントなら、ミニバッファーの編集可能なコンテンツ (つまりプロンプト以外のすべて) を削除する。それ以外はカレントバッファー全体を削除する。

20.13 再帰的なミニバッファー

以下の関数と変数は再帰ミニバッファーを処理します (Section 21.13 [Recursive Editing], page 417 を参照):

`minibuffer-depth` [Function]
この関数はアクティブなミニバッファーのカレント再帰深さを正の整数でリターンする。アクティブなミニバッファーが存在しなければ 0 をリターンする。

`enable-recursive-minibuffers` [User Option]
この変数が非`nil`ならミニバッファーがアクティブでも、(`find-file`のような) ミニバッファーを使用するコマンドを呼び出すことができる。このような呼び出しは新たなミニバッファーにたいして再帰編集レベル (recursive editing level) を生成する。内側レベルの編集の間は外側レベルのミニバッファーは非表示になる。

この変数が`nil`ならミニバッファーがアクティブなときは、たとえ他のウィンドウに切り替えてもミニバッファーコマンドの呼び出しはできない。

コマンド名が非`nil`のプロパティ`enable-recursive-minibuffers`をもつ場合には、たとえミニバッファーから呼び出された場合でも、そのコマンドは引数の読み取りにミニバッファーを使用できる。コマンドの interactive 宣言内で`enable-recursive-minibuffers`を`t`にしても、これを行うことができる (Section 21.2.1 [Using Interactive], page 376 を参照)。ミニバッファーコマンド`next-matching-history-element` (ミニバッファー内では通常 *`M-s`*) は後者を行う。

20.14 ミニバッファー、その他の事項

`minibufferp` **&optional** *buffer-or-name* [Function]
この関数は *buffer-or-name*がミニバッファーなら非`nil`をリターンする。*buffer-or-name*が省略されるとカレントバッファーをテストする。

`minibuffer-setup-hook` [Variable]
これはミニバッファーがエンターされたときは常に実行されるノーマルフックである。Section 23.1 [Hooks], page 461 を参照のこと。

`minibuffer-with-setup-hook` *function* **&rest** *body* [Macro]
このマクロは指定された *function*が`minibuffer-setup-hook`を通じて呼び出されるように計らった後に *body*を実行する。デフォルトでは *function*は`minibuffer-setup-hook`リストの他の関数の前に呼び出されるが、*function*が (`:append` *`func`*) というフォームなら、*func* は他のフック関数の後に呼び出される。

*body*はミニバッファーを複数回使用しないこと。ミニバッファーが再帰的に再エンターされると、*function*は最外のミニバッファーの使用にたいして 1 回だけ呼び出されるだろう。

`minibuffer-exit-hook` [Variable]
これはミニバッファーが exit されたときは常に実行されるノーマルフックである。

`minibuffer-help-form` [Variable]
この変数のカレント値はミニバッファー内で`help-form`をローカルにリバインドするために使用される (Section 24.6 [Help Functions], page 526 を参照)。

`minibuffer-scroll-window` [Variable]
この変数の値が非`nil`なら、それはウィンドウオブジェクトである。ミニバッファー内で関数`scroll-other-window`が呼び出されたときは、このウィンドウをスクロールする (Section 28.21 [Textual Scrolling], page 675 を参照)。

`minibuffer-selected-window` [Function]
この関数はミニバッファーウィンドウが選択される直前に選択されていたウィンドウをリターンする。選択されたウィンドウがミニバッファーウィンドウ以外なら`nil`をリターンする。

`minibuffer-message` *string* **&rest** *args* [Function]
この関数は数秒、あるいは次の入力イベントが到着するまでミニバッファーテキストの最後に一時的に *string* を表示する。変数`minibuffer-message-timeout`は入力がない場合に待機する秒数を指定する (デフォルトは 2)。*args* が非`nil`なら、実際のメッセージは`format-message`に *string* と *args* を渡して作成される。See Section 4.7 [Formatting Strings], page 61 を参照のこと。

`minibuffer-inactive-mode` [Command]
これはインタラクティブなミニバッファー内で使用されるメジャーモードである。キーマップ`minibuffer-inactive-mode-map`を使用する。ミニバッファーが別のフレームにある場合には有用かもしれない。Section 29.9 [Minibuffers and Frames], page 732 を参照のこと。

21 コマンドループ

Emacs を実行すると、ほぼ即座にエディターコマンドループ *(editor command loop)* に移行します。このループはキーシーケンスを読み取り、それらの定義を実行して結果を表示します。このチャプターではこれらが行われる方法と、Lisp プログラムがこれらを行えるようにするサブルーチンを説明します。

21.1 コマンドループの概要

コマンドループが最初に行わなければならないのはキーシーケンスの読み取りです。キーシーケンスはコマンドに変換される入力イベントのシーケンスです。これは関数 `read-key-sequence`を呼び出すことによって行われます。Lisp プログラムもこの関数を呼び出すことができます (Section 21.8.1 [Key Sequence Input], page 404 を参照)。これらはより低レベルの `read-key`や `read-event` (Section 21.8.2 [Reading One Event], page 405) で入力を読み取ったり、`discard-input` (Section 21.8.6 [Event Input Misc], page 410 を参照) で保留中の入力を無視することもできます。

キーシーケンスはカレントでアクティブなキーマップを通じてコマンドに変換されます。これが行われる方法については Section 22.10 [Key Lookup], page 434 を参照してください。結果はキーボードマクロかインタラクティブに呼び出し可能な関数になります。キーが `M-x`なら他のコマンドの名前を読み取って、それを呼び出します。これはコマンド `execute-extended-command` (Section 21.3 [Interactive Call], page 382 を参照) により行われます。

コマンドの実行に先立ち、Emacs はアンドゥ境界 (undo boundary) を作成するために `undo-boundary`を実行します。Section 32.10 [Maintaining Undo], page 796 を参照してください。

コマンドを実行するために、Emacs はまず `command-execute`を呼び出してコマンドの引数を読み取ります (Section 21.3 [Interactive Call], page 382 を参照)。Lisp で記述されたコマンドについては、`interactive`指定で引数を読み取る方法を指定します。これはプレフィクス引数 (Section 21.12 [Prefix Command Arguments], page 415 を参照) を使用したり、ミニバッファー内 (Chapter 20 [Minibuffers], page 342 を参照) で確認を求めて読み取りを行うかもしれません。たとえばコマンド `find-file`には `interactive`指定があり、これはミニバッファーを使用してファイル名を読み取ることを指定します。`find-file`の関数 body はミニバッファーを使用しないので、Lisp コードから関数として `find-file`を呼び出す場合には、通常の Lisp 関数引数としてファイル名を文字列で与えなければなりません。

コマンドがキーボードマクロ (文字列やベクター) なら、Emacs は `execute-kbd-macro`を使用してそれを実行します (Section 21.16 [Keyboard Macros], page 420 を参照)。

`pre-command-hook` [Variable]

このノーマルフックはコマンドを実行する前に、エディターコマンドループにより実行される。その際、`this-command`には実行しようとするコマンドが含まれ、`last-command`には前のコマンドが記述される。Section 21.5 [Command Loop Info], page 385 を参照のこと。

`post-command-hook` [Variable]

このノーマルフックはコマンドを実行した後 (quit やエラーにより早期に終了させられたコマンドを含む) に、エディターコマンドループにより実行される。その際、`this-command`は正に実行されたコマンド、`last-command`は前に実行されたコマンドを参照する。

このフックは Emacs が最初にコマンドループにエンターしたときにも実行される (その時点では `this-command`と `last-command`はいずれも `nil`)。

`pre-command-hook`と`post-command-hook`の実行中は、quit は抑制されます。これらのフックのいずれかを実行中にエラーが発生しても、そのエラーはフックの実行を終了させません。そのかわりにエラーは黙殺されて、エラーが発生した関数はそのフックから取り除かれます。

Emacs サーバー (Section “Emacs Server” in *The GNU Emacs Manual* を参照) に届くリクエストは、キーボードコマンドが行うのと同じように、これらの 2 つのフックを実行します。

21.2 コマンドの定義

スペシャルフォーム `interactive`は Lisp 関数をコマンドに変更します。`interactive`フォームは関数 body のトップレベルに置かなければならず、通常は body 内の最初のフォームとして記述されます。これはラムダ式 (Section 13.2 [Lambda Expressions], page 211 を参照) と `defun` (Section 13.4 [Defining Functions], page 215 を参照) の両方を受け入れます。このフォームはその関数が実際に実行される間は何も行いません。このフォームの存在はフラグとしての役割りをもち、Emacs コマンドループにたいしてその関数がインタラクティブに呼び出せることを告げます。`interactive`フォームの引数はインタラクティブな呼び出しが引数を読み取る方法を指定します。

`interactive`フォームのかわりに、関数シンボルの `interactive-form`プロパティで指定されることもあります。このプロパティが非 `nil`値なら、関数 body 内の `interactive`フォームより優先されます。この機能はほとんど使用されません。

インタラクティブに呼び出されることだけを意図していて、決して Lisp から直接呼び出されない関数が時折あります。この場合には、直接あるいは `declare` (Section 13.14 [Declare Form], page 237 を参照) を通じて、その関数の `interactive-only`プロパティに非`nil`を与えます。これにより、そのコマンドが Lisp から呼び出されるとバイトコンパイラーが警告を発します。`describe-function`の出力にはこれに類似する情報が含まれます。このプロパティの値には文字列、`t`、または任意のシンボルを指定できます。文字列なら、それはバイトコンパイラーによる警告内で直接使用されます (最初は大文字でなくピリオドで終端される文字列であること。たとえば`\"use (system-name) instead.\"`)。シンボルなら、それは Lisp コード内で使用されるかわりの関数です。

ジェネリック関数 (Section 13.8 [Generic Functions], page 222 を参照) に `interactive`フォームを追加してコマンドにすることはできません。

21.2.1 `interactive`の使用

このセクションでは、Lisp 関数をインタラクティブに呼び出し可能なコマンドにする `interactive` フォームの記述方法と、コマンドの `interactive`フォームの検証方法について説明します。

`interactive` *arg-descriptor* [Special Form]

このスペシャルフォームは関数がコマンドであり、したがって (`M-x`を通じて、またはそのコマンドにバインドされたキーシーケンスをエンターすることにより) インタラクティブに呼び出すことができることを宣言する。引数 *arg-descriptor*は、そのコマンドがインタラクティブに呼び出されたときに引数を計算する方法を宣言する。

コマンドは他の関数と同じように Lisp 関数から呼び出されるかもしれないが、その場合には呼び出し側は引数を提供して、*arg-descriptor*は効果をもたない。

`interactive`フォームは関数 body 内のトップレベルに置くか、関数シンボルの `interactive-form`プロパティ((Section 9.4 [Symbol Properties], page 129) を参照) になければならない。これはコマンドループが関数を呼び出す前に interactive フォームを調べることにより効果をもつ (Section 21.3 [Interactive Call], page 382 を参照)。一度関数が呼び出されると関数 body 内のすべてのフォームが実行される。このとき body 内に

`interactive`フォームが出現しても、そのフォームは引数の評価さえされず単に `nil`をリターンする。

慣例により`interactive`フォームは関数 body 内の最初のトップレベルフォームとするべきである。`interactive`フォームがシンボルの `interactive-form`プロパティと関数 body の両方に存在する場合には前者が優先される。`interactive-form`フォームは既存の関数に interactive フォームを追加したり、その関数を再定義することなく引数をインタラクティブに処理する方法を変更するために使用できる。

引数 *arg-descriptor*は以下の 3 つの可能性があります:

- 省略または`nil`ならコマンドは引数なしで呼び出される。コマンドが 1 つ以上の引数を要求する場合は即座にエラーとなる。
- 文字列なら、その文字列の内容は改行で区切られた要素シーケンスであり、1 つの要素が 1 つの引数に対応する [1]。各要素はコード文字 (Section 21.2.2 [Interactive Codes], page 378 を参照) と、オプションでその後のプロンプト (コード文字として使用される文字やコード文字としては無視されるものもある) により構成される。以下は例である:

  ```
  (interactive "P\nbFrobnicate buffer: ")
  ```

  コード文字 '`P`'はそのコマンドの 1 つ目の引数を raw コマンドプレフィクス (Section 21.12 [Prefix Command Arguments], page 415 を参照) にセットする。'`bFrobnicate buffer: `' は、ユーザーに '`Frobnicate buffer: `'のプロンプトを示して既存のバッファーの名前の入力を促し、これは 2 つ目かつ最後の引数になる。

  プロンプト文字列には、プロンプト内の前の引数 (1 つ目の引数から始まる) の値を含めるために '`%`'を使用できる。これは `format-message` (Section 4.7 [Formatting Strings], page 61 を参照) を使用して行われる。たとえば以下は既存のバッファーの名前を読み取って、その後にそのバッファーに与える新たな名前を読み取る例である:

  ```
  (interactive "bBuffer to rename: \nsRename buffer %s to: ")
  ```

  文字列の先頭に '`*`'がある場合、そのバッファーが読み取り専用ならエラーがシグナルされる。

  文字列の先頭が '`@`'で、そのコマンドの呼び出しに使用されたキーシーケンスに何らかのマウスイベントが含まれる場合は、そのコマンドを実行する前に、それらのうち最初のイベントに結びつくウィンドウが選択される。

  文字列の先頭が '`^`'で、そのコマンドがシフト転換 *(shift-translation)* を通じて呼び出された場合は、そのコマンドを実行する前にマークをセットして一時的にリージョンをアクティブにするか、すでにアクティブなリージョンを拡張する。コマンドがシフト転換なしで呼び出されて、リージョンが一時的にアクティブな場合は、コマンドを実行する前にそのリージョンを非アクティブにする。シフト転換は `shift-select-mode`によりユーザーレベルで制御される。Section "Shift Selection" in *The GNU Emacs Manual* を参照のこと。

  '`*`'、'`@`'、`^`は一緒に使用でき、その場合は順序に意味はない。実際の引数の読み取りは残りのプロンプト文字列 ('`*`'、'`@`'、`^`以外の最初の文字以降) により制御される。
- 文字列以外の Lisp 式なら、そのコマンドに渡す引数リストを取得するために評価されるフォームである。このフォームは通常はユーザーから入力を読み取るためにさまざまな関数を呼び出し、そのためにほとんどの場合はミニバッファー (Chapter 20 [Minibuffers], page 342 を参照) を通じてか、キーボードから直接読み取りを行う (Section 21.8 [Reading Input], page 403 を参照)。

[1] いくつかの要素は実際に 2 つの引数を提供します。

引数値としてポイントやマークを提供するのも一般的だが、何かを行いかつ (ミニバッファー使用の有無に関わらず) 入力を読み取る場合には、読み取りの前にポイント値またはマーク値の整数を確実に取得しておくこと。カレントバッファーはサブプロセスの出力を受信するかもしれず、コマンドが入力を待つ間にサブプロセス出力が到着すると、ポイントやマークの再配置が起こり得る。

以下は行ってはいけない例である:

```
(interactive
 (list (region-beginning) (region-end)
       (read-string "Foo: " nil 'my-history)))
```

これにたいして以下はキーボード入力を読み取った後にポイントとマークを調べることにより、上記の問題を避ける例である:

```
(interactive
 (let ((string (read-string "Foo: " nil 'my-history)))
   (list (region-beginning) (region-end) string)))
```

警告: 引数値にはプリントや読み取りが不可能なデータ型を含めないこと。いくつかの機能は後続のセッションに読み込ませるために `command-history`をファイルに保存する。コマンドの引数に '`#<...>`'構文を使用してプリントされるデータ型が含まれていると、それらの機能は動作しなくなるだろう。

しかしこれには少数の例外がある。`(point)`、`(mark)`、`(region-beginning)`、`(region-end)`などの一連の式に限定して使用することに問題はない。なぜなら Emacs はこれらを特別に認識して、コマンドヒストリー内に (値ではなく) その式を配置すからである。記述した式がこれらの例外に含まれるかどうか確認するには、コマンドを実行した後に `(car command-history)`を調べればよい。

`interactive-form` *function* [Function]

この関数は *function*の `interactive`フォームをリターンする。*function*がインタラクティブに呼び出し可能な関数 (Section 21.3 [Interactive Call], page 382 を参照) なら、値はそのコマンドの引数を計算する方法を指定する `interactive`フォーム (`(interactive` *spec*`)`) である。それ以外では値は `nil`である。*function*がシンボルなら、そのシンボルの関数定義が使用される。

21.2.2 `interactive`にたいするコード文字

ここで説明されているコード文字には、以下で定義されるいくつかのキーワードが含まれています:

Completion
: 補完を提供する。`TAB`、`SPC`、`RET`は `completing-read` (Section 20.6 [Completion], page 350 を参照) を使用して引数を読み取って名前の補完を行う。`?`で利用可能な補完リストを表示する。

Existing
: 既存オブジェクトの名前を要求する。無効な名前は受け付けられない。カレント入力が有効でなければ、ミニバッファーを exit するコマンドは exit しない。

Default
: ユーザーがテキストを何もエンターしなければ、ある種のデフォルト値が使用される。デフォルトはコード文字に依存する。

No I/O
: このコード文字は入力を読み取らずに引数を計算する。したがってプロンプト文字列を使用せず、与えられたプロンプト文字列は無視される。

  たとえそのコード文字がプロンプト文字列を使用しなくても、それが文字列内で最後のコード文字でなければ、その後に改行を付加しなければならない。

Prompt コード文字の直後にプロンプトが続く。プロンプトの終端は文字列の終端、または改行。

Special このコード文字はインタラクティブ文字列の先頭にあるときのみ意味があり、プロンプトと改行を要求しない。単一の独立した文字。

以下は `interactive`で使用されるコード文字です:

‘`*`’ カレントバッファーが読み取り専用ならエラーをシグナルする。[Special]

‘`@`’ このコマンドを呼び出したキーシーケンス内の最初のマウスイベントに関連するウィンドウを選択する。[Special]

‘`^`’ シフト転換を通じてコマンドが呼び出された場合はコマンドを実行する前に、マークをセットして一時的にリージョンをアクティブにするか、すでにリージョンがアクティブならリージョンを拡張する。シフト転換を通じずにコマンドが呼び出されて、リージョンが一時的にアクティブならコマンドを実行する前にそのリージョンを非アクティブにする。[Special]

‘`a`’ 関数名 (`fboundp`を満足するシンボル)。[Existing]、[Completion]、[Prompt]

‘`b`’ 既存バッファーの名前。デフォルトではカレントバッファー (Chapter 27 [Buffers], page 590 を参照) の名前を使用する。[Existing]、[Completion]、[Default]、[Prompt]

‘`B`’ バッファー名。そのバッファーが存在する必要はない。デフォルトではカレントバッファーではなくもっとも最近使用されたバッファーの名前を使用する。[Completion]、[Default]、[Prompt]

‘`c`’ 文字。カーソルはエコーエリアに移動しない。[Prompt]

‘`C`’ コマンド名 (`commandp`を満足するシンボル)。[Existing]、[Completion]、[Prompt]

‘`d`’ ポイント位置の整数 (Section 30.1 [Point], page 755 を参照)。[No I/O]

‘`D`’ ディレクトリー。デフォルトはカレントバッファーのカレントのデフォルトディレクトリー `default-directory` (Section 25.9.4 [File Name Expansion], page 559 を参照)。[Existing]、[Completion]、[Default]、[Prompt]

‘`e`’ そのコマンドを呼び出したキーシーケンス内の 1 つ目か 2 つ目の非キーボードイベント。より正確には、‘`e`’はリストとしてイベントを取得するので、リスト内のデータを調べることができる。Section 21.7 [Input Events], page 387 を参照のこと。[No I/O]

‘`e`’はマウスイベント、および特別なシステムイベント (Section 21.7.10 [Misc Events], page 395 を参照) にたいして使用する。コマンドが受け取るイベントリストは、そのイベントに依存する。Section 21.7 [Input Events], page 387 ではそれぞれのイベントのリスト形式を、対応するサブセクションでそれぞれ説明しているので参されたい。

1 つのコマンドの interactive 仕様の中で ‘`e`’を複数回使用できる。そのコマンドを呼び出したキーシーケンスがイベント *n*(リスト) をもつなら、‘`e`’の *n*番目がそのイベントを提供する。フンクションキーや ASCII文字のようなリスト以外のイベントは、‘`e`’に関連するイベントとしてカウントされない。

‘`f`’ 既存ファイルのファイル名 (Section 25.9 [File Names], page 555 を参照)。デフォルトのディレクトリーは `default-directory`。[Existing]、[Completion]、[Default]、[Prompt]

‘`F`’ ファイル名。ファイルが存在している必要はない。[Completion]、[Default]、[Prompt]

‘G’ ファイル名。ファイルが存在している必要はない。ユーザーがディレクトリー名だけをエンターしたら値はそのディレクトリー名となり、そのディレクトリー名にファイル名は追加されない。[Completion]、[Default]、[Prompt]

‘i’ 無関係な引数。このコード文字は引数値として常に `nil`を与える。[No I/O]

‘k’ キーシーケンス (Section 22.1 [Key Sequences], page 422 を参照)。これはカレントキーマップ内でコマンド (または未定義のコマンド) が見つかるまで、イベントを読み取り続ける。キーシーケンス引数は文字列かベクターで表される。カーソルはエコーエリアに移動しない。[Prompt]

‘k’が (マウスの)down-event で終わるキーシーケンスを読み取ると、後続の (マウスの)up-event も読み取ってそれを廃棄する。コード文字 ‘U’により up-event へのアクセスを得られる。

この種の入力は `describe-key`や `global-set-key`のようなコマンドにより使用される。

‘K’ キーシーケンス。その定義は変更されることを意図している。これは ‘k’と同じように機能するが、キーシーケンス内の最後の入力イベントにたいして、通常は (必要なら) 使用される未定義キーから定義済みキーへの変換を抑制する。

‘m’ マーク位置の整数。[No I/O]

‘M’ 任意のテキスト。ミニバッファー内でカレントバッファーの入力メソッド (Section “Input Methods” in *The GNU Emacs Manual* を参照) を使用して読み取りを行い、それを文字列でリターンする。[Prompt]

‘n’ 数字。ミニバッファーで読み取られる。入力が数字でなければユーザーは再試行する必要がある。‘n’は決してプレフィクス引数を使用しない。[Prompt]

‘N’ 数引数 (numeric prefix argument)。ただしプレフィクス引数がなければ *n*のように数字を読み取る。値は常に数字。Section 21.12 [Prefix Command Arguments], page 415 を参照のこと。[Prompt]

‘p’ 数引数 (小文字の ‘p’であることに注意)。[No I/O]

‘P’ raw プレフィクス引数 (大文字の ‘P’であることに注意)。[No I/O]

‘r’ 2 つの数引数 (ポイントとマーク)。小さいほうが先。これは 1 つではなく連続する 2 つの引数を指定する唯一のコード文字である。コマンド呼び出し時にカレントなバッファーにマークがセットされていなければエラーをシグナルする。[No I/O]

‘s’ 任意のテキスト。ミニバッファー内で読み取りを行って文字列としてリターンする (Section 20.2 [Text from Minibuffer], page 343 を参照)。*C-j*か `RET`で入力を終端する (これらの文字を入力に含めるために *C-q*を使用できる)。[Prompt]

‘S’ intern 済みのシンボル。名前はミニバッファー内で読み取られる。*C-j*か `RET`で入力を終端する。ここでは通常はシンボルを終端するその他の文字 (たとえば空白文字、丸カッコ、角カッコ) では終端されない。[Prompt]

‘U’ キーシーケンスか `nil`。‘k’(または ‘K’) が読み取った後に、(もしあれば) 捨てられる (マウスの)up-event を取得するために、引数 ‘k’(または ‘K’) の後で使用され得る。捨てられた up-event が存在しなければ、‘U’は引数として `nil`を提供する。[No I/O]

‘v’ ユーザーオプションとして宣言された変数 (述語 custom-variable-pを満足する)。これは read-variableを使用して変数を読み取る。[Definition of read-variable], page 358 を参照のこと。[Existing]、[Completion]、[Prompt]

‘x’ Lisp オブジェクト。そのオブジェクトの入力構文により指定され、*C-j*か RETで終端される。オブジェクトは評価されない。Section 20.3 [Object from Minibuffer], page 347 を参照のこと。[Prompt]

‘X’ Lisp フォームの値。‘X’は ‘x’のように読み取りを行いフォームを評価して、その値がコマンドの引数になる。[Prompt]

‘z’ コーディングシステム名 (シンボル)。ユーザーが null 入力をエンターすると、引数値は nilになる。Section 33.10 [Coding Systems], page 864 を参照のこと。[Completion]、[Existing]、[Prompt]

‘Z’ コマンドにプレフィクス引数があればコーディングシステム名。プレフィクス引数がなければ ‘Z’は引数値として nilを提供する。[Completion]、[Existing]、[Prompt]

21.2.3 interactiveの使用例

以下に interactiveの例をいくつか示します:

```
(defun foo1 ()              ; foo1は 1 つの引数をとり
    (interactive)           ;    単に 2 単語分前に移動する
    (forward-word 2))
     ⇒ foo1

(defun foo2 (n)             ; foo2は引数を 1 つとる
    (interactive "^p")      ;    引数は数引数
                            ; shift-select-modeでは、
                            ;    リージョンをアクティブにするか、拡張する
    (forward-word (* 2 n)))
     ⇒ foo2

(defun foo3 (n)             ; foo3は引数を 1 つとる
    (interactive "nCount:") ;    引数はミニバッファーで読み取られる
    (forward-word (* 2 n)))
     ⇒ foo3

(defun three-b (b1 b2 b3)
  "Select three existing buffers.
Put them into three windows, selecting the last one."
    (interactive "bBuffer1:\nbBuffer2:\nbBuffer3:")
    (delete-other-windows)
    (split-window (selected-window) 8)
    (switch-to-buffer b1)
    (other-window 1)
    (split-window (selected-window) 8)
    (switch-to-buffer b2)
    (other-window 1)
    (switch-to-buffer b3))
```

```
     ⇒ three-b
(three-b "*scratch*" "declarations.texi" "*mail*")
     ⇒ nil
```

21.2.4 コマンド候補からの選択

マクロ `define-alternatives`はジェネリックコマンド *(generic command)* を定義するために使用できます。これらはユーザーの選択により複数の候補から選択可能な interactive 関数の実装です。

`define-alternatives` *command* **&rest** *customizations* [Macro]

新たなコマンド *command*(シンボル) を定義する。

最初にユーザーが *M-x command RET*を実行したとき、Emacs はコマンドが使用する実際のフォームにたいして確認を求めて、その選択をカスタム変数として記録する。プレフィクス引数を使用すると選択肢の選択のプロセスを繰り返す。

変数 *command*`-alternatives`には、*command*の実装候補が alist で含まれる。この変数がセットされるまで `define-alternatives`は効果をもたない。

*customizations*が非 `nil`なら、`defcustom`キーワード (典型的には`:group`と`:version`) と、*command*`-alternatives`の宣言に追加する値により構成される選択肢。

21.3 インタラクティブな呼び出し

コマンドループはキーシーケンスをコマンドに変換した後、関数 `command-execute`を使用してその関数を呼び出します。そのコマンドが関数なら、`command-execute`は引数を読み取りコマンドを呼び出す `call-interactively`を呼び出します。自分でこれらの関数を呼び出すこともできます。

このコンテキストにおいて用語 “command” はインタラクティブにコール可能な関数 (または関数 like なオブジェクト) やキーボードマクロを指すことに注意してください。つまりコマンドを呼び出すキーシーケンスのことではありません (Chapter 22 [Keymaps], page 422 を参照)。

`commandp` *object* **&optional** *for-call-interactively* [Function]

この関数は *object*がコマンドなら `t`、それ以外は `nil`をリターンする。

コマンドには文字列とベクター (キーボードマクロとして扱われる)、トップレベルの `interactive`フォーム (Section 21.2.1 [Using Interactive], page 376 を参照) を含むラムダ式、そのようなラムダ式から作成されたバイトコンパイル関数オブジェクト、interactive として宣言 (`autoload`の 4 つ目の引数が非 `nil`) された autoload オブジェクト、およびいくつかのプリミティブ関数が含まれる。`interactive-form`プロパティが非 `nil`のシンボル、および関数定義が `commandp`を満足するシンボルもコマンドとされる。

*for-call-interactively*が非 `nil`なら、`call-interactively`が呼び出すことができるオブジェクトにたいしてのみ `commandp`は `t`をリターンする。したがってキーボードマクロは該当しなくなる。

`commandp`を使用する現実的な例については、Section 24.2 [Accessing Documentation], page 520 内の `documentation`を参照のこと。

`call-interactively` *command* **&optional** *record-flag keys* [Function]

この関数は interactive 呼び出し仕様にしたがって引数を取得し、インタラクティブに呼び出し可能な関数 *command*を呼び出す。これは *command*がリターンするものが何であれ、それをリターンする。

たとえばもし以下の署名をもつ関数があり:

```
(defun foo (begin end)
```

```
  (interactive "r")
  ...)
```

以下を行うと

```
(call-interactively 'foo)
```

これはリージョン (pointと mark) を引数として fooを呼び出すだろう。

*command*が関数でない、またはインタラクティブに呼び出せない (コマンドでない) 場合にはエラーをシグナルする。たとえコマンドだとしても、キーボードマクロ (文字列かベクター) は関数ではないので許容されないことに注意。*command*がシンボルなら `call-interactively` はそれの関数定義を使用する。

*record-flag*が非 `nil`なら、このコマンドとコマンドの引数は無条件にリスト `command-history`に追加される。それ以外なら引数の読み取りにミニバッファーを使用した場合のみコマンドが追加される。Section 21.15 [Command History], page 419 を参照のこと。

引数 *keys*が与えらたら、それはコマンドを呼び出すためにどのイベントを使用するかコマンドが問い合わせた場合に与えるべきイベントシーケンスを指定するベクターである。*keys*が `nil` または省略された場合のデフォルトは、`this-command-keys-vector`のリターン値である。[Definition of this-command-keys-vector], page 386 を参照のこと。

`funcall-interactively` *function* **&rest** *arguments* [Function]

この関数は `funcall` (Section 13.5 [Calling Functions], page 217 を参照) と同様に機能するが、インタラクティブな呼び出しのように見える呼び出しを生成する。*function*内部での `called-interactively-p`の呼び出しは `t`をリターンするだろう。*function*がコマンドでなければ、エラーをシグラルすることなくそれを呼び出す。

`command-execute` *command* **&optional** *record-flag keys special* [Function]

この関数は *command*を実行する。引数 *command*は述語 `commandp`を満足しなければならない。つまりインタラクティブに呼び出し可能な関数かキーボードマクロでなければならない。

*command*が文字列かベクターなら、`execute-kbd-macro`により実行される。関数は *record-flag*および *keys*引数とともに `call-interactively`に渡される (上記参照)。

*command*がシンボルなら、その位置にシンボルの関数定義が使用される。`autoload`定義のあるシンボルは、インタラクティブに呼び出し可能な関数を意味するよう宣言されていればコマンドとして判断される。そのような宣言は指定されたライブラリーのロードと、シンボル定義の再チェックにより処理される。

引数 *special*が与えられたら、それはプレフィクス引数を無視して、それをクリアーしないという意味である。これはスペシャルイベント (Section 21.9 [Special Events], page 412 を参照) を実行する場合に使用される。

`execute-extended-command` *prefix-argument* [Command]

この関数は `completing-read`(Section 20.6 [Completion], page 350 を参照) を使用して、ミニバッファーからコマンド名を読み取る。その後で指定されたコマンドを呼び出すために `command-execute`を使用する。そのコマンドがリターンするのが何であれ、それが `execute-extended-command`の値となる。

そのコマンドがプレフィクス引数を求める場合には、*prefix-argument*の値を受け取る。`execute-extended-command`がインタラクティブに呼び出されたら、カレントの raw プレフィクス引数が *prefix-argument*に使用され、それが何であれ実行するコマンドに渡される。

通常は execute-extended-commandは *M-x*の定義なので、プロンプトとして文字列 'M-x ' を使用する (execute-extended-commandを呼び出したイベントからプロンプトを受け取るほうが良いのだろうが実装は苦痛を併なう)。プレフィクス引数の値の説明がもしあれば、それもプロンプトの一部となる。

```
(execute-extended-command 3)
---------- Buffer: Minibuffer ----------
3 M-x forward-word RET
---------- Buffer: Minibuffer ----------
     ⇒ t
```

21.4 インタラクティブな呼び出しの区別

interactive 呼び出しの際に、コマンドが (エコーエリア内の情報メッセージなどのような) 視覚的な追加フィードバックを表示すべきときがあります。これを行うためには 3 つの方法があります。その関数が call-interactivelyを使用して呼び出されたかどうかテストするには、オプション引数 print-messageを与えるとともに、interactive 呼び出しで非 nilとなるように interactive仕様を使うのが推奨される方法です。以下は例です:

```
(defun foo (&optional print-message)
  (interactive "p")
  (when print-message
    (message "foo")))
```

数プレフィクス引数は決して nilにならないので、わたしたちは"p"を使用します。この方法で定義された関数はキーボードマクロから呼び出されたときにメッセージを表示します。

追加引数による上記の手法は、呼び出し側に "この呼び出しを interactive として扱うように" 伝えることができるので通常は最善です。しかし called-interactively-pをテストすることによってこれを行うこともできます。

called-interactively-p *kind* [Function]

この関数は呼び出された関数が call-interactivelyを使用して呼び出されえいたら tをリターンする。

引数 *kind*はシンボル interactiveかシンボル anyのいずれかである。これが interactiveなら、called-interactively-pはユーザーから直接呼び出しが行われたとき— たとえば関数呼び出しにバインドされたキーシーケンスをユーザーがタイプした場合がそれに該当するが、ユーザーがその関数を呼び出すキーボードマクロ (Section 21.16 [Keyboard Macros], page 420 を参照) を実行中した場合は該当しない — だけ tをリターンする。*kind*が anyなら、called-interactively-pはキーボードマクロを含む任意の種類の interactive 呼び出しにたいして tをリターンする。

疑わしい場合には anyを使用すること。interactiveの使用が正しいと解っているのは、関数が実行中に役に立つメッセージを表示するかどうか判断が必要な場合だけである。

Lisp 評価 (または applyや funcall)) を通じて呼び出された場合には、関数は決してインタラクティブに呼び出されたとは判断されない。

以下は called-interactively-pを使用する例:

```
(defun foo ()
  (interactive)
  (when (called-interactively-p 'any)
    (message "Interactive!")
    'foo-called-interactively))

;; M-x fooとタイプする
     ⊣ Interactive!

(foo)
     ⇒ nil
```

以下は called-interactively-pの直接呼び出しと間接呼び出しを比較した例。

```
(defun bar ()
  (interactive)
  (message "%s" (list (foo) (called-interactively-p 'any))))

;; M-x barとタイプする
     ⊣ (nil t)
```

21.5 コマンドループからの情報

エディターコマンドループは自分自身と実行するコマンドのために、いくつかの Lisp 変数にステータス記録を保持します。一般的に this-commandと last-command以外は、Lisp プログラム内でこれらの変数を変更するのは良いアイデアではありません。

last-command [Variable]
: この変数はコマンドループによって実行された以前のコマンド (前にカレントだったコマンド) の名前を記録する。値は通常は関数定義をもつシンボルだが、その保証はない。

  コマンドがコマンドループからリターンするとき、this-commandから値がコピーされる。ただしそのコマンドが後続のコマンドにたいしてプレフィクス引数を指定されたときを除く。

  この変数は常にカレント端末にたいしてローカルであり、バッファーローカルにできない。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

real-last-command [Variable]
: この変数は Emacs により last-commandと同様にセットアップされるが、Lisp プログラムから決して変更されない。

last-repeatable-command [Variable]
: この変数は入力イベントの一部ではない、もっとも最近実行されたコマンドを格納する。これはコマンド repeatが再実行を試みるコマンドである。Section “Repeating” in *The GNU Emacs Manual* を参照のこと。

this-command [Variable]
: この変数はコマンドループにより現在実行中のコマンドの名前を記録する。last-commandと同様、通常は関数定義をもつシンボルである。

  コマンドループはコマンドを実行する直前にこの変数をセットして、(そのコマンドが後続のコマンドのプレフィクス引数を指定しなければ) そのコマンドが終了したときにその値を last-commandにコピーする。

いくつかのコマンドは次に実行されるコマンドが何であれ、それにたいするフラグとして実行中の間この変数をセットする。特にテキストを kill する関数は `this-command`を `kill-region` にセットするので、直後に実行された任意の kill コマンドは、kill したテキストを前に kill されたテキストに追加するべきことが解かるだろう。

特定のコマンドでエラー発生時に前のコマンドとして認識されたくなければ、それを防ぐようにそのコマンドをコーディングしなければなりません。これを行う 1 つの方法は、以下のようにコマンドの最初で `this-command`に `t`をセットして、最後に `this-command`に正しい値をセットする方法です:

```
(defun foo (args...)
  (interactive ...)
  (let ((old-this-command this-command))
    (setq this-command t)
    ... 処理を行う ...
    (setq this-command old-this-command)))
```

エラーなら `let`は古い値をリストアするので、わたしたちは `let`で `this-command`をバインドしません。この場合における `let`の機能は、わたしたちが正に避けたいと思っていることを行ってしまうでしょう。

`this-original-command` [Variable]

コマンドのリマップ (Section 22.13 [Remapping Commands], page 441 を参照) が発生したときを除き、これは `this-command`と同じ値をもつ。リマップが発生すると `this-command` は実際に実行されたコマンド、`this-original-command`は実行を指定されたが他のコマンドにリマップされたコマンドを与える。

`this-command-keys` [Function]

この関数は現在のコマンドを呼び出したキーシーケンスと、加えてそのコマンドにたいするプレフィクス引数を生成した前のコマンドを含む文字列かベクターをリターンする。`read-event` を使用するコマンドにより、タイムアウトせずに読み取られたすべてのイベントが最後に加えられる。

しかしそのコマンドが`read-key-sequence`を呼び出していたら、最後に読み取られたキーシーケンスをリターンする。Section 21.8.1 [Key Sequence Input], page 404 を参照のこと。シーケンス内のすべてのイベントが文字列として適当な文字なら文字列が値になる。Section 21.7 [Input Events], page 387 を参照のこと。

```
(this-command-keys)
;; これを評価するために C-u C-x C-eを使用すると
     ⇒ "^U^X^E"
```

`this-command-keys-vector` [Function]

`this-command-keys`と同様だが常にベクターでイベントをリターンするので、入力イベントを文字列内に格納する複雑さを処理する必要がない (Section 21.7.15 [Strings of Events], page 402 を参照)。

`clear-this-command-keys` **&optional** *keep-record* [Function]

この関数は `this-command-keys`がリターンするイベントテーブルを空にする。*keep-record* が `nil`なら、その後に関数 `recent-keys`(Section 40.13.2 [Recording Input], page 1118 を参照) がリターンするレコードも空にする。これは特定のケースにおいてパスワードを読み取った後、次のコマンドの一部として不用意にパスワードがエコーされるのを防ぐために有用である。

`last-nonmenu-event` [Variable]
この変数はキーシーケンス (マウスメニューからのイベントは勘定しない) の一部として読み取られた最後の入力イベントを保持する。

この変数の 1 つの使い方は、`x-popup-menu`にたいしてどこにメニューをポップアップすべきか告げる場合である。これは内部的に `y-or-n-p`(Section 20.7 [Yes-or-No Queries], page 366 を参照) にも使用されている。

`last-command-event` [Variable]
この変数にはコマンドの一部としてコマンドループに読み取られた最後の入力イベントがセットされる。この変数は主に `self-insert-command`内でどの文字が挿入されたか判断するために使用されている。

```
last-command-event
;; これを評価するために C-u C-x C-eを使用すると
     ⇒ 5
```

`C-e`の ASCIIコードの 5 が値になる。

`last-event-frame` [Variable]
この変数は最後の入力イベントが送られたフレームを記録する。これは通常はそのイベントが生成されたときに選択されていたフレームだが、そのフレームの入力が他のフレームにリダイレクトされていたら、そのリダイレクトされていたフレームが値となる。Section 29.10 [Input Focus], page 732 を参照のこと。

最後のイベントがキーボードマクロに由来する場合、値は `macro`になる。

21.6 コマンド後のポイントの調整

プロパティ`display`や `composition`をもつテキストや非表示のテキストシーケンスの中間では、Emacs はポイント値を表示できません。したがってコマンドが終了した後にコマンドループにリターンした後にそのようなシーケンス中にポイントがある場合には、そのシーケンスを効果的に不可触にするために、コマンドループは通常ポイントをそのようなシーケンスの端へと移動します。

変数`disable-point-adjustment`をセットすることにより、コマンドはこの機能を抑制できます:

`disable-point-adjustment` [Variable]
この変数が非 `nil`ならコマンドがコマンドループにリターンするとき、コマンドループはこれらのテキストプロパティをチェックせず、これらのプロパティをもつシーケンスの外にポイントを移動しない。

コマンドループは各コマンドを実行する前にこの変数を `nil`にセットするので、あるコマンドがこれをセットしても効果が適用されるのはそのコマンドにたいしてだけである。

`global-disable-point-adjustment` [Variable]
この変数を非 `nil`にセットするとシーケンス外にポイントを移動する、これらの機能は完全にオフになる。

21.7 入力イベント

Emacs コマンドループは入力イベント *(input events)* のシーケンスを読み取ります。入力イベントとはキーボードやマウスのアクティビティ、または Emacs に送られるシステムイベントを表します。キーボードアクティビティにたいするイベントは文字かシンボルです。それ以外のイベントは常にリストになります。このセクションでは入力イベントの表現と意味について詳細を説明します。

`eventp` *object* [Function]

この関数は *object*が入力イベントかイベント型なら非`nil`をリターンする。

イベントまたはイベント型として非`nil`の任意のシンボルが使用されるかもしれないことに注意。`eventp`は Lisp コードによりイベントとして使用されることを意図したシンボルかどうかは区別できない。

21.7.1 キーボードイベント

キーボードから取得できる入力には 2 つの種類があります。それは通常のキーとファンクションキーです。通常のキーは文字に対応しており (修飾されているかもしれない)、それらが生成するイベントは Lisp 内では文字で表現されます。文字イベントのイベント型は文字自身 (整数) であり、何らかの修飾ビットがセットされているかもしれません。Section 21.7.12 [Classifying Events], page 398 を参照してください。

入力文字イベントは 0 から 524287 までの基本コード *(basic code)* に加えて、以下の修飾ビット *(modifier bits)* の一部、またはすべてによって構成されます:

meta 文字イベントコードのビット 2^{27} はメタキーが押下された状態で文字がタイプされたことを示す。

control 文字イベントコードのビット 2^{26} は非 ASCIIコントロール文字を示す。

`C-a`のような非 ASCII コントロール文字は、自身が特別な基本コードをもつため、それらを示すために Emacs は特別なビットを必要としない。つまり `C-a`のコードは単なる 1 である。

しかし`%`のような非 ASCIIとコントロールを組み合わせてタイプすると取得される数値は`%`に 2^{26} を加えた値となる (端末が非 ASCIIコントロール文字、すなわち 27 番目のビットがセットされた文字をサポートすると仮定する)。

shift 文字イベントコードのビット 2^{25} (26 番目のビット) はシフトキーが押下された状態でASCIIコントロール文字がタイプされたことを示す。

アルファベット文字にたいしては、基本コード自身が大文字か小文字かを示す。数字と句読点文字にたいしてシフトキーは、異なる基本コードをもつ完全に違う文字を選択する。可能な限り ASCII文字として保つために、Emacs はこれらの文字にたいしてビット 2^{25} を使用しない。

しかし ASCIIは `C-A`と `C-a`を区別する方法を提供しないので、Emacs は `C-A`にたいしてビット 2^{25} を使用し、`C-a`には使用しない。

hyper 文字イベントコードのビット 2^{24} はハイパーキーが押下された状態で文字がタイプされたことを示す。

super 文字イベントコードのビット 2^{23} はスーパーキーが押下された状態で文字がタイプされたことを示す。

alt 文字イベントコードのビット 2^{22} はアルトキーが押下された状態で文字がタイプされたことを示す (ほとんどのキーボードで `Alt`とラベルされたキーは、実際にはアルトキーではなくメタキーとして扱われる)。

プログラム内での特定のビット数値の記述は避けるのが最善の方法です。文字の修飾ビットをテストするためには、関数 `event-modifiers` (Section 21.7.12 [Classifying Events], page 398 を参照) を使用してください。キーバインディングを作成するときは、修飾ビットつきの文字にたいする読み取り構文を使用できます (‘`\C-`’、‘`\M-`’、... など)。`define-key`でのキーバインディング作成

では、文字を指定するために `(control hyper ?x)`のようなリストを使用できます (Section 22.12 [Changing Key Bindings], page 438 を参照)。関数 `event-convert-list`はそのようなリストをイベント型に変換します (Section 21.7.12 [Classifying Events], page 398 を参照)。

21.7.2 ファンクションキー

ほとんどのキーボードにはファンクションキー *(function keys)* があります。これは名前や文字以外のシンボルをもつキーです。Emacs Lisp ではファンクションキーはシンボルとして表現されます。そのシンボル名はファンクションキーのラベルの小文字です。たとえば`F1`とラベルされたキーを押下すると、シンボル`f1`で表される入力イベントが生成されます。

ファンクションキーのイベント型はイベントシンボル自身です。Section 21.7.12 [Classifying Events], page 398 を参照してください。

ファンクションキーにたいするシンボルの命名規約には、以下のような特別なケースがいくつかあります:

`backspace`、`tab`、`newline`、`return`、`delete`
: これらのキーは、ほとんどのキーボードにおいて特別にキーをもつ、一般的な ASCII コントロール文字に対応する。

  ASCIIでは *C-i* と `TAB`は同じ文字である。端末がこれらを区別できるなら Emacs は前者を整数の 9、後者をシンボル`tab`で表現することによって Lisp プログラムにこれらの違いを伝える。

  ほとんどの場合はこれらの 2 つを区別するのは役に立たない。そのため`local-function-key-map` (Section 22.14 [Translation Keymaps], page 442 を参照) は`tab`を 9 にマップするようセットアップされている。したがって文字コード 9(文字 *C-i*) へのキーバインディングは`tab`にも適用される。このグループ内の他のシンボルも同様である。関数`read-char`がこれらのイベントを文字に変換する場合も同様である。

  ASCIIでは`BS`は実際は *C-h* である。しかし`backspace`は文字コード 8(`BS`) ではなく、文字コード 127(`DEL`) に変換される。ほとんどのユーザーにとってこれは好ましいだろう。

`left`、`up`、`right`、`down`
: 矢印カーソルキー

`kp-add`、`kp-decimal`、`kp-divide`、...
: キーパッドのキー (標準的なキーボードにおいては右側にある)。

`kp-0`、`kp-1`、...
: キーパッドの数字キー。

`kp-f1`、`kp-f2`、`kp-f3`、`kp-f4`
: キーパッドの PF キー。

`kp-home`、`kp-left`、`kp-up`、`kp-right`、`kp-down`
: キーパッドの矢印キー。Emacs は通常これらを非キーパッドのキー`home`、`left`、...に変換する。

`kp-prior`、`kp-next`、`kp-end`、`kp-begin`、`kp-insert`、`kp-delete`
: 通常は他の箇所にあるキーと重複するキーパッド追加キー。Emacs は通常これらを同じような名前の非キーパッドキーに変換する。

ファンクションキーにたいしても修飾キー ALT、CTRL、HYPER、META、SHIFT、SUPERを使用できます。シンボル名のプレフィクスとしてこれらを表します:

‘A-’ アルト修飾。

‘C-’ コントロール修飾。

‘H-’ ハイパー修飾。

‘M-’ メタ修飾。

‘S-’ シフト修飾。

‘s-’ スーパー修飾。

したがって METAを押下した場合の F3キーにたいするシンボルは M-f3になります。複雑のプレフィクスを使用する場合には、アルファベット順の記述を推奨します。とはいえキーバインディングが修飾されたファンクションキーを探す際に引数の順序は関係ありません。

21.7.3 マウスイベント

Emacs は 4 つの種類のマウスイベントをサポートします。それはクリックイベント、ドラッグイベント、ボタンダウンイベント、モーションイベントです。すべてのマウスイベントはリストで表現されます。このリストの CAR はイベント型です。イベント型はどのマウスボタンが関与するのか、それにたいしてどの修飾キーが使用されたかを示します。イベント型によりダブル、あるいはトリプルでボタンが押されたかを区別することもできます (Section 21.7.7 [Repeat Events], page 393 を参照)。残りのリスト要素は位置と時間の情報を提供します。

キーの照合ではイベント型だけが問題になります。2 つのイベントが同じコマンドを実行するには同じイベント型が必要です。実行されるコマンドは interactive のコード ‘e’を使用して、これらのイベントの完全な値にアクセスできます。Section 21.2.2 [Interactive Codes], page 378 を参照してください。

マウスイベントで開始されたキーシーケンスはカレントバッファーではなく、マウスのあったウィンドウ内のバッファーのキーマップを使用して読み取られます。これはウィンドウ内でクリックすることによりそのウィンドウやそのウィンドウのバッファーが選択されることを意味しません。つまりそれは完全にそのキーシーケンスのコマンドバインディングの制御下にあるのです。

21.7.4 クリックイベント

ユーザーが同じ場所でマウスボタンを押してからリリース (release: 離す) すると、*click* イベントが生成されます。すべてのマウスクリックイベントは同じフォーマットを共有します:

```
(event-type position click-count)
```

event-type これはマウスボタンが使用されたことを示す。これはシンボル mouse-1、mouse-2、... のうちのいずれかで、マウスボタンは左から右に番号が付される。

ファンクションキーにたいして行うのと同様にアルト、コントロール、ハイパー、メタ、シフト、スーパーの修飾にたいしてプレフィクス ‘A-’、‘C-’、‘H-’、‘M-’、‘S-’、‘s-’も使用できる。

このシンボルはイベントのイベント型としての役割りももつ。イベントのキーバインディングはこれらの型により示される。したがって mouse-1にたいするキーバインディングが存在すれば、そのバインディングは *event-type*が mouse-1であるようなすべてのイベントに適用されるだろう。

position これはマウスクリックがどこで発生したかを表すマウス位置リスト *(mouse position list)* である。詳細は以下を参照のこと。

click-count
これは同じマウスボタンを素早く繰り返し押下したときの回数である。Section 21.7.7 [Repeat Events], page 393 を参照のこと。

クリックイベントの *position*スロット内にあるマウス位置リストの内容にアクセスするためには、一般的には Section 21.7.13 [Accessing Mouse], page 400 に記述された関数を使用するべきです。このリストの明示的なフォーマットはどこでクリックが発生したかに依存します。テキストエリア、モードライン、ヘッダーライン、タブライン、フリンジ、マージンエリアでのクリックにたいしてマウス位置リストは以下のフォーマットをもちます

```
(window pos-or-area (x . y) timestamp
 object text-pos (col . row)
 image (dx . dy) (width . height))
```

以下はこれらのリスト要素がもつ意味です:

window クリックが発生したウィンドウ。

pos-or-area
テキストエリア内でクリックされた文字のバッファー位置。またはテキストエリア外がクリックされたなら、クリックが発生したウィンドウエリア。これはシンボル `mode-line`、`header-line`、`tab-line`、`vertical-line`、`left-margin`、`right-margin`、`left-fringe`、`right-fringe`のいずれか。

特別なケースの 1 つとして *pos-or-area*が単なるシンボルではなく、(上記シンボルのいずれか 1 つの) シンボルを含むリストのような場合がある。これは Emacs により登録されたイベントにたいする、イマジナリープレフィクスキー (imaginary prefix key) の後に発生する。Section 21.8.1 [Key Sequence Input], page 404 を参照のこと。

x, *y* クリックの相対ピクセル座標 (relative pixel coordinates)。あるウィンドウのテキストエリア内でのクリックにたいする座標原点 `(0 . 0)`は、テキストエリアの左上隅となる。Section 28.3 [Window Sizes], page 613 を参照のこと。モードライン、ヘッダーラインやタブライン内でのクリックにたいする座標原点は、そのウィンドウ自身の左上隅となる。フリンジ、マージン、垂直ボーダー (vertical border) では *x*は有意なデータをもたない。フリンジ、マージンでは *y*はヘッダーラインの最下端からの相対位置である。すべてのケースにおいて *x*と *y*の座標はそれぞれ右方向と下方向で増加する。

timestamp
そのイベントが発生した時刻をシステム依存の初期時刻 (initial time) からの経過ミリ秒で表す整数。

object `nil` (バッファーテキスト上でクリックが発生したことを意味する)、クリック箇所にテキストプロパティやオーバーレイがあれば (*string* . *string-pos*) という形式のコンスセル。

string クリックされた文字列。すべてのテキストプロパティを含む。

string-pos クリックが発生した文字列内の位置。

text-pos マージンエリアやフリンジにたいするクリックでは、そのウィンドウ内の対応する行内の最初の可視な文字のバッファー位置となる。モードライン、ヘッダーラインやタブラインにたいするクリックでは `nil`。他のイベントにたいしてはクリックされたバッファーのクリックされた最寄りの位置となる。

col, *row* これらは *x*、*y*の位置にあるグリフ (gliph) の実際の行と列の座標数値である。行 *x*がその行の実際のテキストの最後の列を超えるなら、*col*はデフォルトの文字幅をもつ仮想的な追加列数を加えた値が報告される。そのウィンドウがヘッダーラインをもてば行 0 はヘッダーライン、タブラインももてば行 1 はタブラインとなり、それ以外ならテキストエリアの上端ラインが行 0 となる。ウィンドウのテキストエリアのクリックにたいしては、テキストエリアの左端列が列 0 となり、モードラインまたはヘッダーラインのクリックにたいしてはそのラインの左端が列 0 となる。フリンジまたは垂直ボーダーのクリックにたいしては、これらは有意なデータをもたない。マージンのクリックにたいしては、*col*はマージンエリアの左端、*row*はマージンエリアの上端から測られる。

image クリック箇所にイメージがあれば `find-image`がリターンするようなイメージオブジェクト (Section 39.17.8 [Defining Images], page 1061 を参照)、それ以外は `nil`。

dx, *dy* これらは *object*の左上隅 `(0 . 0)`からの相対的ピクセル座標。*object*が `nil` (バッファーを意味する) の場合には、クリックされた文字グリフの左上隅からの相対座標。

width, *height*
これらは *object*のピクセル幅とピクセル高さであり、*object*が `nil`ならクリックされた文字グリフのピクセル幅とピクセル高さ。

スクロールバーへのクリックにたいして、*position*は以下の形式をもちます:

```
(window area (portion . whole) timestamp part)
```

window スクロールバーがクリックされたウィンドウ。

area これはシンボル `vertical-scroll-bar`である。

portion スクロールバーの上端からクリック位置までのピクセル数。GTK+を含むいくつかのツールキットでは、Emacs がこれらのデータを抽出できないので値は常に `0`。

whole スクロールバーの全長のピクセル数。GTK+を含むいくつかのツールキットでは、Emacs がこれらのデータを抽出できないので値は常に `0`。

timestamp
イベントが発生したミリ秒時刻。GTK+を含むいくつかのツールキットでは、Emacs がこれらのデータを抽出できないので値は常に `0`。

part クリックが発生したスクロールバー部分。これはシンボル `handle`(スクロールバーのハンドル)、`above-handle`(ハンドルの上側エリア)、`below-handle`(ハンドルの下側エリア)、`up`(スクロールバー端の上矢印)、`down`(スクロールバー端の下矢印) のいずれか。

21.7.5 ドラッグイベント

Emacs では特別なことをしなくてもドラッグイベントを取得できます。ドラッグイベント *(drag event)* はユーザーがマウスボタンを押下して、ボタンをリリースする前にマウスを異なる文字位置に移動すると毎回発生します。すべてのマウスイベントと同じように、ドラッグイベントは Lisp ではリストで表現されます。このリストは以下のように開始マウス位置と最終位置ぼ両方を記録します:

```
(event-type
 (window1 START-POSITION)
 (window2 END-POSITION))
```

ドラッグイベントにたいしては、シンボル *event-type*の名前にプレフィクス '`drag-`'が含まれます。たとえばボタン 2 を押下したままマウスをドラッグすると `drag-mouse-2`イベントが生成されます。このイベントの 2 つ目と 3 つ目の要素は、マウス位置リスト (Section 21.7.4 [Click Events],

page 390 を参照) としてドラッグの開始と終了の位置を与えます。任意のマウスイベントの 2 つ目の要素に同じ方法でアクセスできます。しかしドラッグイベントは最初に選択されていたフレームの境界外で終了するかもしれません。この場合のには 3 つ目の要素の位置リストに、ウィンドウのかわりにそのフレームが含まれます。

'drag-'プレフィクスは、その後に 'C-'や 'M-'のような修飾キープレフィクスが続きます。

read-key-sequenceがキーバインディングをもたず、対応するクリックイベントにキーバインディングがあるようなドラッグイベントを受け取ると、この関数はそのドラッグイベントをドラッグ開始位置でのクリックイベントに変更します。これはもし望まなければクリックイベントとドラッグイベントを区別する必要がないことを意味します。

21.7.6 ボタンダウンイベント

クリックイベントとドラッグイベントは、ユーザーがマウスボタンをリリースしたときに発生します。ボタンがリリースされるまでクリックとドラッグを区別することはできないので、リリース前にイベントが発生することはありません。

ボタンが押下されたらすぐに何か処理したいなら、ボタンダウン *(button-down)* イベントを処理する必要があります [2]。これらは *event-type*のシンボル名に 'down-'が含まれることを除き、クリックイベントとまったく同じようなリストにより表現されます。'down-'プレフィクスの後には 'C-'や 'M-'のような修飾キープレフィクスが続きます。

関数 read-key-sequenceはコマンドバインディングをもたないボタンダウンイベントを無視します。したがって Emacs コマンドループもこれらを無視します。これはボタンダウンイベントで何かしたい場合以外は、ボタンダウンイベントの定義について配慮する必要がないことを意味します。ボタンダウンイベントを定義する通常の理由は、ボタンがリリースされるまで (モーションイベントを読み取ることにより) マウスモーションを追跡できるからです。Section 21.7.8 [Motion Events], page 395 を参照してください。

21.7.7 リピートイベント

マウスを移動せずに同じマウスボタンを素早く 2 回以上連続して押下すると、Emacs は 2 回目とそれ以降の押下にたいして特別なリピート *(repeat)* マウスイベントを生成します。

もっとも一般的なリピートイベントはダブルクリック *(double-click)* イベントです。Emacs はボタンを 2 回クリックしたときにダブルクリックイベントを生成します。このイベントは、(すべてのクリックイベントが通常そうであるように) ボタンをリリースしたときに発生します。

ダブルクリックイベントのイベント型にはプレフィクス 'double-'が含まれます。したがって meta を押しながら 2 つ目のマウスボタンをダブルクリックすると、Lisp プログラムには M-double-mouse-2が渡されます。ダブルクリックイベントがバインディングをもたなければ、対応する通常のクリックイベントのバインディングが実行に使用されます。したがって実際に望んだ場合でなければダブルクリック機能に注意を払う必要はありません。

ユーザーがダブルクリックを行うとき、Emacs はまず通常のクリックイベントを生成して、その後ダブルクリックイベントを生成します。したがってダブルクリックイベントのコマンドバインディングは、すでにシングルクリックイベントが実行された想定でデザインしなければなりません。つま

[2] ボタンダウンはドラッグの保守的なアンチテーゼです。
訳注: 原文は "Button-down is the conservative antithesis of drag."。
ちなみに IT 用語で使用される前は"button-down"はボタンダウンシャツを表すとともに「保守的、堅苦しい」という意味もあり、一方の"drag"は IT 用語として使用される前から「引っ張る、引きずる」という意味で用いられてきましたが「本来は異性が着る洋服」という意味もあります。

りシングルクリックの結果から開始して、ダブルクリックの望むべき結果を生成しなければならないのです。

これはダブルクリックの意味合いが、シングルクリックの意味合いの何らかにもとづいて構築される場合は便利です。これはダブルクリックにたいするユーザーインターフェイスにおける推奨されるデザインプラクティスです。

ボタンをクリックした後にもう一度ボタンを押下して、そのままマウスの移動を開始すると、最終的にボタンをリリースしたときダブルドラッグ *(double-drag)* イベントが取得されます。このイベント型には単なる '`drag`'のかわりに '`double-drag`'が含まれます。ダブルドラッグイベントがバインディングをもたなければ、それがあたかも通常のドラッグイベントだったかのように Emacs はかわりのバインディングを探します。

ダブルクリックやダブルドラッグイベントの前に、Emacs はユーザーが 2 回目にボタンを押したタイミングでダブルダウン *(double-down)* イベントを生成します。このイベント型には単なる '`down`' のかわりに '`double-down`'が含まれます。ダブルダウンイベントがバインディングをもたなければ、それがあたかも通常のボタンダウンイベントだったかのように Emacs はかわりのバインディングを探します。どちらの方法でもバインディングが見つからなければダブルダウンイベントは無視されます。

要約するとボタンをクリックしてすぐにまた押したとき、Emacs は 1 回目のクリックにたいしてダウンイベントとクリックイベントを生成して、2 回目に再度ボタンを押したときにダブルダウンイベント、そして最後にダブルクリックまたはダブルドラッグイベントを生成します。

ボタンを 2 回クリックした後にもう一度押したとき、それらすべてが素早く連続で行われたら、Emacs はトリプルダウン *(triple-down)* イベントと、その後続のトリプルクリック *(triple-click)* かトリプルドラッグ *(triple-drag)* イベントを生成します。これらイベントのイベント型には '`double`' のかわりに '`triple`'が含まれます。トリプルイベントがバインディングをもたなければ Emacs は対応するダブルイベントに使用されるであろうバインディングを使用します。

ボタンを 3 回以上クリックした後に再度ボタンを押すと、3 回を超えた押下にたいするイベントはすべてトリプルイベントになります。Emacs はクワドループル (quadruple: 4 連)、クインティプル (quintuple: 5 連)、... 等のイベントにたいして個別のイベント型をもちません。しかしボタンが何回押下されたかを正確に調べるためにイベントリストを調べることができます。

`event-click-count` *event* [Function]
この関数は *event*を誘因した連続するボタン押下の回数をリターンする。*event*がダブルダウン、ダブルクリック、ダブルドラッグなら値は 2 である。*event*がトリプルイベントなら値は 3 以上になる。*event*が (リピートイベントではない) 通常のマウスイベントなら値は 1。

`double-click-fuzz` [User Option]
リピートイベントを生成するためには、ほぼ同じスクリーン位置で連続でマウスボタンを押下しなければならない。`double-click-fuzz`の値はダブルクリックを生成するために連続する 2 回のクリック間で、マウスが移動 (水平と垂直) するかもしれない最大ピクセル数を指定する。

この変数はドラッグとみなされるマウスモーションの閾値でもある。

`double-click-time` [User Option]
リピートイベントを生成するためには、連続するボタン押下のミリ秒間隔が `double-click-time`の値より小さくなければならない。`double-click-time`を `nil`にセットすると複数回クリック検知が完全に無効になる。`t`にセットすると時間制限が取り除かれる。その場合は Emacs は位置だけで複数回のクリックを検知する。

21.7.8 モーションイベント

Emacs は、ボタンアクティビティが何もないマウスのモーション (motion: 動き) を記述するマウスモーション *(mouse motion)* イベントを生成するときがあります。マウスモーションイベントは以下のようなリストによって表現されます:

```
(mouse-movement POSITION)
```

*position*はマウスカーソルのカレント位置を指定するマウス位置リスト (Section 21.7.4 [Click Events], page 390 を参照) です。ドラッグイベントの終了位置のように、この位置リストは最初に選択されていた境界外の位置を表すかもしれず、その場合にはそのフレーム内のその位置のウィンドウが含まれます。

スペシャルフォーム `track-mouse`は、ボタン内でのモーションイベントの生成を有効にします。`track-mouse`フォームの外側では、Emacs はマウスの単なるモーションにたいするイベントは生成せず、これらのイベントは発生しません。Section 29.15 [Mouse Tracking], page 742 を参照してください。

`mouse-fine-grained-tracking` [Variable]

非 `nil`ならたとえ非常に小さい移動でもマウスモーションイベントを生成する。それ以外ならテキスト内の同一グリフをマウスカーソルがポイントし続けるかぎりモーションイベントは生成されない。

21.7.9 フォーカスイベント

ウィンドウシステムはユーザーにたいしてどのウィンドウがキーボード入力を受け取るか制御するための一般的な方法を提供します。このウィンドウ選択はフォーカス *(focus)* と呼ばれます。Emacs のフレームを切り替えるためにユーザーが何かを行うと、それはフォーカスイベント *(focus event)* を生成します。フォーカスイベントの通常の定義はグローバルキーマップ内にあり、ユーザーが期待するように Emacs で新たなフレームを選択するためのものです。Section 29.10 [Input Focus], page 732 ではフォーカスイベントに関連するフックも説明しています。

フォーカスイベントは以下のように Lisp のリストで表現されます:

```
(switch-frame new-frame)
```

ここで *new-frame*は切り替え先のフレームです。

X ウィンドウマネージャーには、あるウィンドウにマウスを移動するだけで、そこにフォーカスされるようにセットアップするものがいくつかあります。通常は他の種類の入力が到着するまで、Lisp プログラムがフォーカスの変更を知る必要はありません。Emacs はユーザーが新たなフレーム内で実際にキーボードのキーをタイプするかマウスボタンを押下したときしか、フォーカスイベントを生成しません。つまりフレーム間でマウスを移動させても、フォーカスイベントは生成されません。

キーシーケンスの途中におけるフォーカスイベントは、そのシーケンスを誤ったものにするかもしれません。そのため Emacs は決してキーシーケンスの途中でフォーカスイベントを生成しません。ユーザーがキーシーケンスの途中 (つまりプレフィクス引数の後) でフォーカスを変更すると、複数イベントキーシーケンスの前か後にフォーカスイベントが到着するように、Emacs はフォーカスイベントを記録しておきます。

21.7.10 その他のシステムイベント

他にもシステム内での出来事を表現するイベント型がいくつかあります。

`(delete-frame (frame))`

このイベントの種類はユーザーがウィンドウマネージャーに特定のウィンドウを削除するコマンドを与えたことを示し、Emacs のフレームにたいして発生する。

フレーム削除 (`delete-frame`)イベントの標準的な定義では *frame*が削除される。

`(iconify-frame (`*frame*`))`
: このイベントの種類はウィンドウマネージャーを使用してユーザーが *frame*をアイコン化したことを示す。標準的な定義は `ignore`。これはそのフレームがすでにアイコン化されているので、Emacs が行う必要のことは何もないからである。このイベント型の目的は、望むならこのようなイベントの追跡を可能にしておくためである。

`(make-frame-visible (`*frame*`))`
: このイベントの種類はウィンドウマネージャーを使用してユーザーが *frame*を非アイコン化したことを示す。標準的な定義は `ignore`。これは、そのフレームがすでに可視化されているので、Emacs が行う必要のことは何もないからである。

`(wheel-up` *position*`)`
`(wheel-down` *position*`)`
: この種類のイベントはマウスホイールを移動したことにより発生する。*position*要素はそのイベント発生時のマウスカーソル位置を指定するマウス位置リスト (Section 21.7.4 [Click Events], page 390 を参照)。

この種類のイベントはある種のシステムでのみ発生する。いくつかのシステムでは、かわりに `mouse-4`と `mouse-5`が使用される。可搬性のあるコードとするためには、マウスホイールにたいしてどのイベント型が期待されるかを決定するために `mwheel.el`内で定義されている変数 `mouse-wheel-up-event`と `mouse-wheel-down-event`を使用すること。

`(drag-n-drop` *position files*`)`
: この種類のイベントは Emacs 外部アプリケーション内でファイルグループが選択されて、それが Emacs フレーム内にドラッグアンドドロップされたときに発生する。

要素 *position*は、そのイベント位置を記述しマウスクリックイベントで使用されるフォーマット (Section 21.7.4 [Click Events], page 390 を参照) と同じ。要素 *files*はドラッグアンドドロップされたファイル名のリスト。通常はそれらのファイルを visit することによってこのイベントは処理される。

この種類のイベントは現在のところある種のシステムでのみ生成される。

`help-echo`
: この種類のイベントは、テキストプロパティ`help-echo`をもつバッファーテキスト部分上にマウスポインターが移動したときに生成される。生成されるイベントは以下の形式をもつ:

```
(help-echo frame help window object pos)
```

イベントパラメーターの正確な意味とヘルプテキストを表示するためにこれらのパラメーターを使用する方法は、[Text help-echo], page 821 で説明されている。

`sigusr1`
`sigusr2`
: これらのイベントは Emacs プロセスがシグナル `SIGUSR1`や `SIGUSR2`を受け取ったときに生成される。シグナルは追加情報を運搬しないので追加データは含まれない。これらのシグナルはデバッグに有用 (Section 18.1.1 [Error Debugging], page 296 を参照)。

ユーザーシグナルを catch するためには、`special-event-map` (Section 22.9 [Controlling Active Maps], page 432 を参照) 内で対応するイベントにバインドする。そのコマンドは引数なしで呼び出され、`last-input-event`内の特定のシグナルイベントが利用できる (Section 21.8.6 [Event Input Misc], page 410 を参照)。たとえば:

```
(defun sigusr-handler ()
```

```
  (interactive)
  (message "Caught signal %S" last-input-event))

(define-key special-event-map [sigusr1] 'sigusr-handler)
```

シグナルハンドラーをテストするために、自身で Emacs にシグナルを送信できる:

```
(signal-process (emacs-pid) 'sigusr1)
```

`language-change`

この種類のイベントは MS-Windows 上で入力言語が変更されたときに生成される。これは通常はキーボードキーが異なる言語の文字で Emacs に送られることを意味する。生成されるイベントは以下の形式をもつ:

```
(language-change frame codepage language-id)
```

ここで *frame*は言語が変更されたときカレントだったフレーム、*codepage*は新たなコードページ番号 (codepage number)、*language-id*は新たな入力言語の数値 ID である。*codepage*に対応するコーディングシステム (Section 33.10 [Coding Systems], page 864 を参照) は、`cp`*codepage*か `windows-`*codepage*。*language-id*を文字列に変更する (たとえば `set-language-environment`のようなさまざまな言語依存機能にたいしこれを使用する) には、以下のように `w32-get-locale-info`関数を使用する:

```
;; 英語にたいする"ENU"のような言語の省略形を取得する
(w32-get-locale-info language-id)
;; "English (United States)"のような
;; その言語の完全な英語名を取得する
(w32-get-locale-info language-id 4097)
;; その言語の完全なローカライズ名を取得する
(w32-get-locale-info language-id t)
```

キーシーケンスの途中、つまりプレフィクスキーの後にこれらのイベントの 1 つが到着すると、複数イベントキー内ではなくその前か後にそのイベントが到着するように Emacs はそのイベントを記録する。

21.7.11 イベントの例

ユーザーが同じ場所でマウス左ボタンを押して離すと、それは以下のようなイベントシーケンスを生成します:

```
(down-mouse-1 (#<window 18 on NEWS> 2613 (0 . 38) -864320))
(mouse-1      (#<window 18 on NEWS> 2613 (0 . 38) -864180))
```

コントロールキーを押したままユーザーがマウス第 2 ボタンを押してマウスをある行から次の行へドラッグすると、以下のような 2 つのイベントが生成されます:

```
(C-down-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219))
(C-drag-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219)
                (#<window 18 on NEWS> 3510 (0 . 28) -729648))
```

メタキーとシフトキーを押したままユーザーがそのウィンドウのモードライン上でマウス第 2 ボタンを押して他ウィンドウへマウスをドラッグすると、以下のようなイベントのペアが生成されます:

```
(M-S-down-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844))
(M-S-drag-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844)
                  (#<window 20 on carlton-sanskrit.tex> 161 (33 . 3)
                   -453816))
```

全画面表示されていないフレームに入力フォーカスがあってユーザーがマウスをそのフレームのスコープ外へマウスを移動すると、スペシャルフォーム `track-mouse`内では以下のようなイベントが生成されます:

```
(mouse-movement (#<frame *ielm* 0x102849a30> nil (563 . 205) 532301936))
```

SIGUSR1 シグナルを処理するためにはインタラクティブ関数を定義して、それを signal usr1 イベントシーケンスにバインドします:

```
(defun usr1-handler ()
  (interactive)
  (message "Got USR1 signal"))
(global-set-key [signal usr1] 'usr1-handler)
```

21.7.12 イベントの分類

すべてのイベントはイベント型 *(event type)* をもっています。イベント型はキーバインディング目的でイベントをクラス分けします。キーボードイベントにたいするイベント型はイベント値と等しく、したがって文字のイベント型は文字、ファンクションキーシンボルのイベント型はそのシンボル自身になります。リストであるようなイベントのイベント型は、そのリストの CAR 内のシンボルです。したがってイベント型は常にシンボルか文字です。

同じ型の 2 つのイベントはキーバインディングに関する限りは同じものです。したがってそれらは常に同じコマンドを実行します。これらが同じことを行う必要があるという意味ではありませんが、イベント全体を調べてから何を行うか決定するコマンドもいくつかあります。たとえばバッファー内でどこに作用するか決定するためにマウスイベントの場所を使用するコマンドもいくつかあります。

広範なイベントのクラス分けが役に立つときもあります。たとえば他の修飾キーやマウスボタンが使用されたかとは無関係に、METAキーとともに呼び出されたイベントを尋ねたいと思うかもしれません。

関数 event-modifiersや event-basic-typeは、そのような情報を手軽に取得するために提供されています。

`event-modifiers` *event* [Function]

この関数は *event*がもつ修飾子のリストをリターンする。この修飾子はシンボルであり shift、control、meta、alt、hyper、superが含まれる。さらにマウスイベントシンボルの修飾子リストには常に click、drag、downのいずれか 1 つが含まれる。ダブルイベントとトリプルイベントには、doubleや tripleも含まれる。

引数 *event*はイベントオブジェクト全体、または単なるイベント型かもしれない。*event*がカレント Emacs セッション内で入力として読み取られたイベント内で決して使用されないシンボルなら、実際に *event*が変更されたときでも event-modifiersは nilをリターンできる。

いくつか例を挙げる:

```
(event-modifiers ?a)
     ⇒ nil
(event-modifiers ?A)
     ⇒ (shift)
(event-modifiers ?\C-a)
     ⇒ (control)
(event-modifiers ?\C-%)
     ⇒ (control)
(event-modifiers ?\C-\S-a)
     ⇒ (control shift)
(event-modifiers 'f5)
     ⇒ nil
(event-modifiers 's-f5)
     ⇒ (super)
```

```
(event-modifiers 'M-S-f5)
     ⇒ (meta shift)
(event-modifiers 'mouse-1)
     ⇒ (click)
(event-modifiers 'down-mouse-1)
     ⇒ (down)
```

クリックイベントにたいする修飾子リストは明示的に `click`を含むが、イベントシンボル名自体には‘`click`’が含まれない。同じように‘`C-a`’のようなASCIIコントロール文字にたいする修飾子リストでは、たとえ‘`C-a`’の修飾ビットを取り除いて値1をリターンする`read-char`を通じて読み取られたイベントであっても`control`が含まれる。

`event-basic-type` *event* [Function]

この関数は *event*を記述するキー、またはマウスボタンをリターンする。*event*引数は`event-modifiers`の場合と同様。たとえば:

```
(event-basic-type ?a)
     ⇒ 97
(event-basic-type ?A)
     ⇒ 97
(event-basic-type ?\C-a)
     ⇒ 97
(event-basic-type ?\C-\S-a)
     ⇒ 97
(event-basic-type 'f5)
     ⇒ f5
(event-basic-type 's-f5)
     ⇒ f5
(event-basic-type 'M-S-f5)
     ⇒ f5
(event-basic-type 'down-mouse-1)
     ⇒ mouse-1
```

`mouse-movement-p` *object* [Function]

*object*がマウス移動イベントなら、この関数は非`nil`をリターンする。Section 21.7.8 [Motion Events], page 395 を参照のこと。

`event-convert-list` *list* [Function]

この関数は修飾子名リストと基本イベント型 (basic event type) を、それらすべてを指定するイベント型に変換する。基本イベント型はそのリストの最後の要素でなければならない。たとえば、

```
(event-convert-list '(control ?a))
     ⇒ 1
(event-convert-list '(control meta ?a))
     ⇒ -134217727
(event-convert-list '(control super f1))
     ⇒ C-s-f1
```

21.7.13 マウスイベントへのアクセス

このセクションではマウスボタンやモーションイベント内のデータアクセスに役立つ関数を説明します。同じ関数を使用してキーボードイベントデータにもアクセスできますが、キーボードイベントに不適切なデータ要素は 0 か `nil`になります。

以下の 2 つの関数は、マウスイベントの位置を指定するマウス位置リスト (Section 21.7.4 [Click Events], page 390 を参照) をリターンします。

`event-start` *event* [Function]

これは *event*の開始位置をリターンする。

*event*がクリックイベントかボタンダウンイベントなら、この関数はそのイベントの位置をリターンする。*event*がドラッグイベントなら、そのドラッグの開始位置をリターンする。

`event-end` *event* [Function]

これは *event*の終了位置をリターンする。

*event*がドラッグイベントなら、この関数はユーザーがマウスボタンをリリースした位置をリターンする。*event*がクリックイベントかボタンダウンイベントなら、値はそのイベント固有の開始位置となる。

`posnp` *object* [Function]

この関数は *object*が (Section 21.7.4 [Click Events], page 390 に記述されたフォーマットの) マウス位置リストなら非 `nil`、それ以外では `nil`をリターンする。

以下の関数は引数にマウス位置リストを受け取り、そのリストのさまざまな部分をリターンします:

`posn-window` *position* [Function]

*position*があったウィンドウをリターンする。*position*が最初にイベントがあったフレームの外部の位置を表す場合には、かわりにそのフレームをリターンする。

`posn-area` *position* [Function]

*position*内に記録されたウィンドウエリアをリターンする。そのウィンドウのテキストエリアでイベントが発生したときは `nil`、それ以外ではイベントがどこで発生したかを識別するシンボルをリターンする。

`posn-point` *position* [Function]

*position*内のバッファー位置をリターンする。ウィンドウのテキストエリア、マージンエリア、フリンジでイベントが発生したときはバッファー位置を識別する整数値、それ以外では値は未定義。

`posn-x-y` *position* [Function]

*position*内のピクセル単位の xy 座標を、コンスセル `(x . y)` でリターンする。これらは `posn-window`により与えられるウィンドウにたいする相対座標である。

以下はあるウィンドウのテキストエリア内のウィンドウ相対座標をフレーム相対座標に変換する方法を示す例:

```
(defun frame-relative-coordinates (position)
  "POSITION のフレーム相対座標をリターンする。
POSITION はウィンドウのテキストエリアにあるものとする。"
  (let* ((x-y (posn-x-y position))
         (window (posn-window position))
```

```
      (edges (window-inside-pixel-edges window)))
  (cons (+ (car x-y) (car edges))
        (+ (cdr x-y) (cadr edges)))))
```

`posn-col-row` *position* [Function]
この関数は *position*で記述されるのバッファー位置で推定される列と行を含むコンスセル (`col . row`) をリターンする。リターン値は *position*にたいする *x*と *y*の値より計算され、そのフレームのデフォルト文字幅とデフォルト行高 (行間スペースを含む) の単位で与えられる (そのため実際の文字サイズが非デフォルト値の場合には、実際の行と列は計算された値とは異なるかもしれない)。

*row*はそのテキストエリアの上端から数えられることに注意。*position*により与えられるウィンドウがヘッダーライン (Section 23.4.7 [Header Lines], page 493 を参照) やタブラインをもつなら、それらは *row*の数に含まれない。

`posn-actual-col-row` *position* [Function]
*position*内の実際の行と列をコンスセル (`col . row`) でリターンする。値は *position*与えられるウィンドウの実際の行と列。Section 21.7.4 [Click Events], page 390 を参照のこと。*position*が実際のポジション値を含まなければ、この関数は `nil`をリターンする。この場合にはおおよその値を取得するために `posn-col-row`を使用できる。

この関数はタブ文字やイメージによるビジュアル列数のように、ディスプレイ上の文字のビジュアル幅を意味しない。標準的な文字単位の座標が必要なら、かわりに `posn-col-row`を使用すること。

`posn-string` *position* [Function]
*position*に記述された文字列オブジェクトをリターンする。`nil` (*position*がバッファーテキストを記述することを意味する)、またはコンスセル (`string . string-pos`) のいずれか。

`posn-image` *position* [Function]
*position*にあるイメージオブジェクトをリターンする。`nil` (*position*にイメージがない)、またはイメージ spec (`image ...`) のいずれか。

`posn-object` *position* [Function]
*position*により記述されるイメージオブジェクトか文字列オブジェクトをリターンする。`nil` (*position*がバッファーテキストを記述することを意味する)、イメージ (`image ...`)、またはコンスセル (`string . string-pos`) のいずれか。

`posn-object-x-y` *position* [Function]
*position*で記述されるオブジェクトの左上隅からのピクセル単位の xy 座標を、コンスセル (`dx . dy`) でリターンする。*position*がバッファーテキストを記述する場合には、その位置にもっとも近いバッファーテキストの相対位置をリターンする。

`posn-object-width-height` *position* [Function]
*position*で記述されるオブジェクトのピクセル幅とピクセル高さを、コンスセル (`width . height`)でリターンする。*position*がバッファー位置を記述する場合には、その位置の文字のサイズをリターンする。

`posn-timestamp` *position* [Function]
*position*内のタイムスタンプをリターンする。これはミリ秒で表されたイベント発生時刻である。

以下の関数は与えられた特定のバッファー、またはスクリーン位置によって位置リストを計算します。上述の関数でこの位置リスト内のデータにアクセスできます。

`posn-at-point` **&optional** *pos window* [Function]
この関数は *window*内の位置 *pos*にたいする位置リストをリターンする。*pos*のデフォルトは *window*内のポイント、*window*のデフォルトは選択されたウィンドウ。

*window*内で *pos*が不可視なら、`posn-at-point`は `nil`をリターンする。

`posn-at-x-y` *x y* **&optional** *frame-or-window whole* [Function]
この関数は指定されたフレームかウィンドウ *frame-or-window*(デフォルトは選択されたウィンドウ) 内のピクセル座標 *x*と *y*に対応する位置情報をリターンする。*x*と *y*は、使用されたフレームかウィンドウにたいする相対座標である。*whole*が `nil`なら、座標はウィンドウのテキストエリアにたいする相対座標、それ以外ではスクロールバー、マージン、フリンジを含むウィンドウエリア全体にたいする相対座標。

21.7.14 スクロールバーイベントへのアクセス

以下の関数はスクロールバーイベントの解析に役立ちます。

`scroll-bar-event-ratio` *event* [Function]
この関数はスクロールバーで発生したスクロールバーイベントの位置の垂直位置の割り合いをリターンする。値は位置の割り合いを表す 2 つの整数を含むコンスセル (*portion* . *whole*)。

`scroll-bar-scale` *ratio total* [Function]
この関数は、(実質的には)*ratio*に *total*を乗じて、結果を整数に丸める。引数 *ratio*は数字ではなく、`scroll-bar-event-ratio`によってリターンされる典型的な値ペア (*num* . *denom*)である。

この関数はスクロールバー位置をバッファー位置にスケーリングするのに有用。以下のように行う:

```
(+ (point-min)
   (scroll-bar-scale
      (posn-x-y (event-start event))
      (- (point-max) (point-min))))
```

スクロールバーイベントは、xy 座標ペアのかわりに割り合いを構成する 2 つの整数をもつことを思い出してほしい。

21.7.15 文字列内へのキーボードイベントの配置

文字列が使用される場所のほとんどにおいて、わたしたちはテキスト文字を含むもの、つまりバッファーやファイル内で見出すのと同種のものとして文字列を概念化します。Lisp プログラムはときおりキーボード文字、たとえばキーシーケンスやキーボードマクロ定義かもしれないキーボード文字を概念的に含んだ文字列を使用します。しかし文字列内へのキーボード文字の格納は、歴史的な互換性の理由から複雑な問題であり、常に可能なわけではありません。

新たに記述するプログラムでは文字列内にキーボードイベントを格納しないことによって、これらの複雑さを扱うことを避けるよう推奨します。以下はこれを行う方法です:

- `lookup-key`と `define-key`の引数として使用するのでなければ、キーシーケンスにたいして文字列のかわりにベクターを使用する。たとえば `read-key-sequence`のかわりに `read-key-sequence-vector`、`this-command-keys`のかわりに `this-command-keys-vector`を使用できる。

- メタ文字を含むキーシーケンス定数を記述する際には、たとえそれを直接 `define-key`に渡す場合でもベクターを使用する。
- 文字列かもしれないキーシーケンスの内容を調べる必要があるときは、それをリストに変換するために最初に `listify-key-sequence` (Section 21.8.6 [Event Input Misc], page 410 を参照) を使用する。

複雑さはキーボード入力に含まれるかもしれない修飾ビットに起因します。メタ修飾以外の修飾ビットは文字列に含めることができず、メタ文字も特別な場合だけ許容されます。

GNU Emacs の初期のバージョンでは、メタ文字を 128 から 255 のコードで表していました。その頃は基本的な文字コードの範囲は 0 から 127 だったので、すべてのキーボード文字を文字列内に適合させることができました。Lisp プログラムの多くは、特に `define-key`やその種の関数の引数として文字列定数内にメタ文字を意味する '`\M-`'を使用していて、キーシーケンスとイベントシーケンスは常に文字列として表現されていました。

127 超のより大きい基本文字コードと追加の修飾ビットにたいするサポートを加えたとき、わたしたちはメタ文字の表現を変更する必要がありました。現在では文字のメタ修飾を表すフラグは 2^{27} であり、そのような値は文字列内に含めることができません。

プログラムで文字列定数内の '`\M-`'をサポートするために、文字列内に特定のメタ文字を含めるための特別なルールがあります。以下は入力文字シーケンスとして文字列を解釈するためのルールです:

- キーボード文字の値の範囲が 0 から 127 なら、文字列を変更せずに含めることができる。
- これらの 2^{27} から $2^{27} + 127$, までの文字のコード範囲にあるメタ修飾された変種も文字列に含めることができるが、それらの数値を変更しなければならない。値が 128 から 255 の範囲となるように、ビット 2^7 のかわりにビット 2^{27} をセットしなければならない。ユニバイト文字列だけがこれらの文字を含むことができる。
- 265 を超える非 ASCII文字はマルチバイト文字に含めることができる。
- その他のキーボード文字イベントは文字列に適合させられない。これには 128 から 255 の範囲のキーボードイベントが含まれる。

キーボード入力文字の文字列定数を構築する `read-key-sequence`のような関数は、イベントが文字列内に適合しないときは文字列のかわりにベクターを構築するというルールにしたがいます。

文字列内で入力構文 '`\M-`'を使用すると、それは 128 から 255 の範囲のコード、つまり対応するキーボードイベントを文字列内に配すために変更するとき取得されるのと同じコードが生成されます。したがって文字列内のメタイベントは、それが文字列内にどのように配置されたかと無関係に一貫して機能します。

しかしほとんどのプログラムはこのセクションの冒頭の推奨にしたがって、これらの問題を避けたほうがよいでしょう。

21.8 入力の読み取り

エディターコマンドループはキーシーケンスの読み取りに関数`read-key-sequence`を使用して、この関数は`read-event`を使用します。イベント入力にたいしてこれらの関数、およびその他の関数が Lisp 関数から利用できます。Section 39.8 [Temporary Displays], page 996 の `momentary-string-display`、および Section 21.10 [Waiting], page 412 の `sit-for`も参照してください。端末の入力モードの制御、および端末入力のデバッグに関する関数と変数については、Section 40.13 [Terminal Input], page 1118 を参照してください。

高レベル入力機能については Chapter 20 [Minibuffers], page 342 を参照してください。

21.8.1 キーシーケンス入力

コマンドループは `read-key-sequence`を呼び出すことによって、キーシーケンスの入力を一度に読み取ります。Lisp 関数もこの関数を呼び出すことができます。たとえば `describe-key`はキーを記述するためにこの関数を使用します。

`read-key-sequence` *prompt* **&optional** *continue-echo* [Function]
dont-downcase-last switch-frame-ok command-loop

この関数はキーシーケンスを読み取って、それを文字列かベクターでリターンする。この関数は完全なキーシーケンスに蓄積されるまで、つまりカレントでアクティブなキーマップを使用してプレフィクスなしでコマンドを指定するのに十分なキーシーケンスとなるまでイベントの読み取りを継続する (マウスイベントで始まるキーシーケンスは、カレントバッファーではなくマウスのあったウィンドウ内のバッファーのキーマップを使用して読み取られることを思い出してほしい)。

イベントがすべて文字で、それらがすべて文字列に適合すれば、`read-key-sequence`は文字列をリターンする (Section 21.7.15 [Strings of Events], page 402 を参照)。それ以外なら文字、シンボル、リストなどすべての種類のイベントを保持できるベクターをリターンする。文字列やベクターの要素は、キーシーケンス内のイベント。

キーシーケンスの読み取りには、そのイベントを変換するさまざまな方法が含まれる。Section 22.14 [Translation Keymaps], page 442 を参照のこと。

引数 *prompt*はプロンプトとしてエコーエリアに表示される文字列、プロンプトを表示しない場合は `nil`。引数 *continue-echo*が非 `nil`なら、それは前のキーの継続としてそのキーをエコーすることを意味する。

元となる大文字のイベントが未定義で、それと等価な小文字イベントが定義されていれば、通常は大文字のイベントが小文字のイベントに変換される。引数 *dont-downcase-last*が非 `nil`なら、それは最後のイベントを小文字に変換しないことを意味する。これはキーシーケンスを定義するときに適している。

引数 *switch-frame-ok*が非 `nil`なら、たとえ何かをタイプする前にユーザーがフレームを切り替えたとしても、この関数が `switch-frame`を処理すべきではないことを意味する。キーシーケンスの途中でユーザーがフレームを切り替えた場合、またはシーケンスの最初だが *switch-frame-ok*が `nil`のときにフレームを切り替えた場合、そのイベントはカレントキーシーケンスの後に延期される。

引数 *command-loop*が非 `nil`なら、そのキーシーケンスがコマンドを逐次読み取る何かによって読み取られることを意味する。呼び出し側が 1 つのキーシーケンスだけを読み取る場合には、`nil`を指定すること。

以下の例では Emacs はエコーエリアにプロンプト '`?`'を表示して、その後ユーザーが *`C-x C-f`* をタイプする。

```
(read-key-sequence "?")

---------- Echo Area ----------
?C-x C-f
---------- Echo Area ----------

     ⇒ "^X^F"
```

関数 read-key-sequenceは quit を抑制する。この関数による読み取りの間にタイプされた C-gは他の文字と同じように機能し、quit-flagをセットしない。Section 21.11 [Quitting], page 413 を参照のこと。

read-key-sequence-vector *prompt* **&optional** *continue-echo dont-downcase-last switch-frame-ok command-loop* [Function]

これは read-key-sequenceと同様だが、キーシーケンスを常にベクターでリターンして、文字列では決してリターンしない点が異なる。Section 21.7.15 [Strings of Events], page 402 を参照のこと。

入力文字が大文字 (またはシフト修飾をもつ) で、キーバインディングをもたないものの、等価な小文字はキーバインディングをもつ場合、read-key-sequenceはその文字を小文字に変換します。lookup-keyはこの方法による case 変換を行わないことに注意してください。

入力を読み取った結果がシフト変換 *(shift-translation)* されていたら、Emacs は変数this-command-keys-shift-translatedに非 nil値をセットします。シフト変換されたキーにより呼びだされたときに挙動を変更する必要がある Lisp プログラムは、この変数を調べることができます。たとえば関数handle-shift-selectionはリージョンをアクティブ、または非アクティブにするかを判断するためにこの変数の値を調べます (Section 31.7 [The Mark], page 772 を参照)。

関数 read-key-sequenceもマウスイベントのいくつかを変換します。これはバインドされていないドラッグイベントをクリックイベントに変換して、バインドされていないボタンダウンイベントを完全に破棄します。さらにフォーカスイベントとさまざまなウィンドウイベントの再配置も行うため、これらのイベントはキーシーケンス中に他のイベントとも出現することは決してありません。

モードラインやスクロールバーのようなウィンドウの特別な箇所でマウスイベントが発生したとき、そのイベント型は特別なことは何も示さずにマウスボタンと修飾キーの組み合わせを通常表すのと同じシンボルになります。ウィンドウの箇所についての情報はイベント内の別のどこか、すなわち座標に保持されています。しかし read-key-sequenceはこの情報を仮想的なプレフィクスキーに変換します。これらはすべてシンボルであり tab-line、header-line、horizontal-scroll-bar、menu-bar、tab-bar、mode-line、vertical-line、vertical-scroll-barです。これらの仮想的なプレフィクスキーを使用してキーシーケンスを定義することにより、ウィンドウの特別な部分でのカウスクリックにたいして意味を定義できます。

たとえばread-key-sequenceを呼び出した後にそのウィンドウのモードラインをマウスでクリックすると、以下のように 2 つのマウスイベントが取得されます:

```
(read-key-sequence "Click on the mode line: ")
     ⇒ [mode-line
         (mouse-1
          (#<window 6 on NEWS> mode-line
           (40 . 63) 5959987))]
```

num-input-keys [Variable]

この変数の値は、その Emacs セッション内で処理されたキーシーケンスの数である。これには端末からのキーシーケンスと、実行されるキーボードマクロによって読み取られたキーシーケンスが含まれる。

21.8.2 単一イベントの読み取り

read-event、read-char、read-char-exclusiveはコマンド入力にたいするもっとも低レベルの関数です。

ミニバッファーを使用して 1 文字を読み取る関数が必要なら `read-char-from-minibuffer`を使用してください (Section 20.8 [Multiple Queries], page 368 を参照)。

`read-event` **&optional** *prompt inherit-input-method seconds* [Function]

この関数はコマンド入力の次のイベントを読み取ってリターンする。必要ならイベントが利用可能になるまで待機する。

リターンされるイベントはユーザーからの直接のイベントかもしれないし、キーボードマクロからのイベントかもしれない。イベントはキーボードの入力コーディングシステム (Section 33.10.8 [Terminal I/O Encoding], page 877 を参照) によりデコードされていない。

オプション引数 *prompt*が非 `nil`なら、それはエコーエリアにプロンプトとして表示される文字列。*prompt*が `nil`か文字列 '`""`'なら、`read-event`は入力待ちを示すメッセージを何も表示せず、エコーを行うことによってプロンプトの代用とする。エコーに表示される記述はカレントコマンドに至ったイベントや読み取られたイベント。Section 39.4 [The Echo Area], page 983 を参照のこと。

*inherit-input-method*が非 `nil`なら、(もしあれば) 非 ASCII文字の入力を可能にするためにカレントの入力メソッドが採用される。それ以外では、このイベントの読み取りにたいして入力メソッドの処理が無効になる。

`cursor-in-echo-area`が非 `nil`の場合、`read-event`はカーソルを一時的にエコーエリアの、そこに表示されているメッセージの終端に移動する。それ以外では、`read-event`はカーソルを移動しない。

*seconds*が非 `nil`なら、それは入力を待つ最大秒数を指定する数値である。その時間内に入力が何も到着しなければ、`read-event`は待機を終えて `nil`をリターンする。浮動小数点数 *seconds* は待機する秒の分数を意味する。いくつかのシステムではサポートされるのは整数の秒数だけであり、そのようなシステムでは *seconds*は切り捨てられる。*seconds*が `nil`なら、`read-event` は入力が到着するのに必要なだけ待機する。

*seconds*が `nil`ならユーザー入力が到着するのを待つ間、Emacs はアイドル状態にあるとみなされる。この期間中にアイドルタイマー— `run-with-idle-timer` (Section 40.12 [Idle Timers], page 1116 を参照) — を実行できる。しかし *seconds*が非 `nil`なら、非アイドル状態は変更されずに残る。`read-event`が呼び出されたとき Emacs が非アイドルだったら、`read-event`の処理を通じて非アイドルのままとなる。Emacs がアイドルだった場合 (これはアイドルタイマー内部からその呼び出しが行われた場合に起こり得る) は、アイドルのままとまる。

`read-event`がヘルプ文字として定義されたイベントを取得すると、ある状況においては `read-event`がリターンせずに直接イベントを処理することがある。Section 24.6 [Help Functions], page 526 を参照のこと。その他のスペシャルイベント *(special events)*(Section 21.9 [Special Events], page 412 を参照) と呼ばれる特定のイベントも `read-event`で直接処理される。

以下は `read-event`を呼び出してから右矢印キーを押下したとき何が起こるかの例:

```
(read-event)
     ⇒ right
```

`read-char` **&optional** *prompt inherit-input-method seconds* [Function]

この関数は文字入力イベントを読み取ってリターンする。ユーザーが文字以外 (たとえばマウスクリックやファンクションキー) のイベントを生成した場合には、`read-char`はエラーをシグナルする。引数は `read-event`と同じように機能する。

イベントが修飾子をもつ場合には、Emacs はそれらの解決を試みて対応する文字のコードをリターンする。たとえばユーザーが *C-*a をタイプすると、関数は文字 'C-a' の ASCII コードである 1 をリターンする。いくつかの修飾子を文字コードに反映できない場合には、read-char は未解決の修飾子ビットをセットしたままイベントをリターンする。たとえばユーザーが *C-M-*a をタイプすると、関数は 134217729(16 進の 8000001 であり、これはメタ修飾子がセットされた 'C-a') をリターンする。この値は有効な文字コードではないので、characterp のテストに失敗する (Section 33.5 [Character Codes], page 856 を参照)。修飾子ビットが削除された文字の復元には event-basic-type (Section 21.7.12 [Classifying Events], page 398 を参照)、read-char がリターンした文字イベント内の修飾子をテストするには event-modifiers を使用すること。

以下の 1 つ目の例ではユーザーは文字 *1*(ASCII コード 49) をタイプしている。2 つ目の例では eval-expression を使用してミニバッファーから read-char を呼び出すキーボード定義を示している。read-char はキーボードマクロの直後の文字 *1* を読み取る。その後に eval-expression はリターン値をエコーエリアに表示する。

```
(read-char)
     ⇒ 49

;; M-: を使用して以下を評価するものとする
(symbol-function 'foo)
     ⇒ "^[:(read-char)^M1"
(execute-kbd-macro 'foo)
     ⊣ 49
     ⇒ nil
```

read-char-exclusive **&optional** *prompt inherit-input-method seconds* [Function]

この関数は文字入力イベントを読み取ってリターンする。ユーザーが文字イベント以外を生成した場合には、read-char-exclusive はそれを無視して文字を取得するまで他のイベントを読み取る。引数は read-event と同じように機能する。リターン値には read-char のように修飾ビットが含まれるかもしれない。

上記の関数で quit を抑制するものはありません。

num-nonmacro-input-events [Variable]

この変数は端末から受信した入力イベント (キーボードマクロにより生成されたイベントは勘定しない) の総数を保持する。

read-key-sequence と異なり関数 read-event、read-char、read-char-exclusive は Section 22.14 [Translation Keymaps], page 442 で説明した変換を行わないことを強調しておきます。単一キー読み取りでこれらの変換を行いたければ関数 read-key を使用してください。

read-key **&optional** *prompt* [Function]

この関数は 1 つのキーを読み取る。これは read-key-sequence と read-event の間の中間的な関数である。read-key-sequence と異なるのは、キーシーケンスではなく単一キーを読み取ることである。read-event と異なるのは、raw イベントをリターンせずに input-decode-map、local-function-key-map、key-translation-map (Section 22.14 [Translation Keymaps], page 442 を参照) に合わせてデコードと変換を行うことである。

引数 *prompt* はプロンプトとしてエコーエリアに表示する文字列で、nil はプロンプトを表示しないことを意味する。

`read-char-choice` *prompt chars* **&optional** *inhibit-quit* [Function]
この関数は 1 つの文字を読み取ってリターンするために `read-key`を使用する。これは *chars*(許容される文字のリスト) のメンバー以外の入力を無視する。オプションで有効な入力を待つ間の quit イベントも無視する。`read-char-choice`呼び出しの間に `help-form` (Section 24.6 [Help Functions], page 526 を参照) を非 `nil`値にバインドすると、`help-char`の押下により `help-form`が評価され結果が表示される。その後で有効な入力文字、またはキーボード quit の待機を継続する。

`read-multiple-choice` *prompt choices* [Function]
複数の選択肢のある問いをユーザーに尋ねる。*prompt*はプロンプトとして表示する文字列であること。

*choices*は各エントリーの 1 つ目の要素が入力される文字、2 つ目の要素がプロンプトを表示する際にそのエントリーにたいして表示する短い名前であるような alist(スペースがあれば短縮され得る) であり、3 つ目のオプションのエントリーはユーザーがより多くのヘルプを要求した際にヘルプバッファーに表示する長い説明。

リターン値は *choices*のマッチする値。

```
(read-multiple-choice
 "Continue connecting?"
 '((?a "always" "Accept certificate for this and future sessions.")
   (?s "session only" "Accept certificate this session only.")
   (?n "no" "Refuse to use certificate, close connection.")))
```

グラフィカル端末で名前文字列にマッチする文字をハイライトするために `read-multiple-choice-face`フェイスが使用される。

21.8.3 入力イベントの変更と変換

Emacs は `extra-keyboard-modifiers`に合わせて読み取ったすべてのイベントを変更して `read-event`からリターンする前に、(もし適切なら)`keyboard-translate-table`を通じてそれを変換します。

`extra-keyboard-modifiers` [Variable]
この変数は Lisp プログラムにキーボード上の修飾キーを “押下” させる。値は文字。文字の修飾子だけが対象となる。ユーザーがキーボードのキーを押下するたびに、その修飾キーがすでに押下されたかのように処理される。たとえば `extra-keyboard-modifiers`を`?\C-\M-a`にバインドすると、このバインディングのスコープ内にある間、すべてのキーボード入力文字はコントロール修飾とメタ修飾を適用されるだろう。文字`?\C-@`は 0 と等価なので、この目的にたいしてはコントロール文字として勘定されないが、修飾無しの文字として扱われる。したがって `extra-keyboard-modifiers`を 0 にセットすることによって、すべての修飾をキャンセルできる。

ウィンドウシステムを利用していれば、この方法によってプログラムが任意の修飾キーを押下できる。それ以外では `CTL`と `META`のキーだけを仮想的に押下できる。

この変数は実際にキーボードに由来するイベントだけに適用され、マウスイベントやその他のイベントには効果がないことに注意。

`keyboard-translate-table` [Variable]
この端末ローカルな変数はキーボード文字にたいする変換テーブルである。これによりコマンドバインディングを変更することなく、キーボード上のキーを再配置できる。値は通常は文字テーブル、または `nil`(文字列かベクターも指定できるが時代遅れとされている)。

keyboard-translate-tableが文字テーブル (Section 6.6 [Char-Tables], page 110 を参照) なら、キーボードから読み取られた各文字はその文字テーブルを調べる。非 nilの値が見つかったら実際の入力文字のかわりにそれを使用する。

この変換は文字が端末から読み取られた後、最初に発生することに注意。recent-keysのような記録保持機能や文字を記録する dribble ファイルは、この変換の後に処理される。

さらにこの変換は入力メソッド (Section 33.11 [Input Methods], page 878 を参照) に文字を提供する前に行われることにも注意。入力メソッド処理の後に文字を変換したいなら translation-table-for-input (Section 33.9 [Translation of Characters], page 863 を参照) を使用すること。

keyboard-translate *from to* [Function]
この関数は文字コード *from*を文字コード *to*に変換するために keyboard-translate-table を変更する。必要ならキーボード変換テーブルを作成する。

以下は *C-x*でカット、*C-*でコピー、*C-v*でペーストを処理するように keyboard-translate-tableを使用する例:

```
(keyboard-translate ?\C-x 'control-x)
(keyboard-translate ?\C-c 'control-c)
(keyboard-translate ?\C-v 'control-v)
(global-set-key [control-x] 'kill-region)
(global-set-key [control-c] 'kill-ring-save)
(global-set-key [control-v] 'yank)
```

拡張 ASCII入力をサポートするグラフィカルな端末上では、シフトキーとともにタイプすることによって、標準的な Emacs における意味をこれらの文字から依然として取得することが可能です。これはキーボード変換が関与する文字とは異なりますが、それらは通常と同じ意味をもちます。

read-key-sequenceのレベルでイベントシーケンスを変換するメカニズムについては、Section 22.14 [Translation Keymaps], page 442 を参照してください。

21.8.4 入力メソッドの呼び出し

イベント読み取り関数は、もしあればカレント入力メソッドを呼び出します (Section 33.11 [Input Methods], page 878 を参照)。input-method-functionの値が非 nilなら関数を指定します。read-eventが修飾ビットのないプリント文字 (SPCを含む) を読み取ったときは、その文字を引数としてその関数を呼び出します。

input-method-function [Variable]
これが非 nilなら、その値はカレントの入力メソッド関数を指定する。

警告: この変数を letでバインドしてはならない。この変数はバッファーローカルであることが多く、入力の前後 (これは正にあなたがバインドするであろうタイミングである) でバインドすると、Emacs が待機中に非同期にバッファーを切り替えた場合に、誤ったバッファーに値がリストアされてしまう。

入力メソッド関数は入力として使用されるイベントのリストをリターンするべきです (このリストが nilなら、それは入力がないことを意味するので read-eventは他のイベントを待機する)。これらのイベントは unread-command-events (Section 21.8.6 [Event Input Misc], page 410 を参照) 内のイベントの前に処理されます。入力メソッドによってリターンされるイベントは、たとえそれらが修飾ビットのないプリント文字であっても再度入力メソッドに渡されることはありません。

入力メソッド関数が read-eventや read-key-sequenceを呼び出したら、再帰を防ぐために最初に input-method-functionを nilにバインドするべきです。

キーシーケンスの 2 つ目および後続のイベントを読み取るときは、入力メソッド関数は呼び出されません。したがってそれらの文字は入力メソッドの処理対象外です。入力メソッド関数は overriding-local-mapと overriding-terminal-local-mapの値をテストするべきです。これらの変数のいずれかが非 nilなら入力メソッドは引数をリストに put して、それ以上の処理を行わずにそのリストをリターンするべきです。

21.8.5 クォートされた文字の入力

ユーザーが手軽にコントロール文字やメタ文字、リテラルや 8 進文字コードを指定できるように文字の指定をもとめることができます。コマンド quoted-insertはこの関数を使用しています。

read-quoted-char **&optional** *prompt* [Function]

この関数は read-charと同様だが、最初に読み取った文字が 8 進数 (0–7) なら任意の個数の 8 進数 (8 進数以外の文字を見つけた時点でストップする) を読み取って、その文字コードにより表される文字をリターンする。8 進シーケンスを終端させた文字が RETならそれは無視される。他の終端文字はこの関数がリターンした後の入力として使用される。

最初の文字の読み取り時には quit は抑制されるので、ユーザーはは *C-g*を入力できる。Section 21.11 [Quitting], page 413 を参照のこと。

*prompt*が与えられたら、それはユーザーへのプロンプトに使用する文字列を指定する。プロンプト文字列はその後に 1 つの '-'とともに常にエコーエリアに表示される。

以下の例ではユーザーは 8 進数の 177(10 進数の 127) をタイプしている。

```
(read-quoted-char "What character")

---------- Echo Area ----------
What character 1 7 7-
---------- Echo Area ----------

     ⇒ 127
```

21.8.6 その他のイベント入力の機能

このセクションではイベントを使い切らずに先読みする方法と、入力の保留や保留の破棄の方法について説明します。Section 20.9 [Reading a Password], page 369 の関数 read-passwdも参照してください。

unread-command-events [Variable]

この変数はコマンド入力として読み取り待機中のイベントのリストを保持する。イベントはこのリスト内の出現順に使用され、使用されるごとにリストから取り除かれる。

ある関数がイベントを読み取ってそれを使用するかどうか決定する場合がいくつかあるためにこの変数が必要になる。この変数にイベントを格納するとコマンドループやコマンド入力を読み取る関数によってイベントは通常のように処理される。

たとえば数引数を実装する関数は、任意の個数の数字を読み取る。数字イベントが見つからないとき、関数はそのイベントを読み戻す (unread) ので、そのイベントはコマンドループによって通常通り読み取られることができる。同様にインクリメンタル検索は、検索において特別な意味をもたないイベントを読み戻すためにこの機能を使用する。なぜならそれらのイベントは検索を exit して、通常どおり実行されるべきだからである。

unread-command-eventsにイベントを置くためにキーシーケンスからイベントを抽出するには、listify-key-sequence (以下参照) を使用するのが簡単で信頼のおける方法である。

もっとも最近読み戻したイベントが最初に再読み取りされるように、通常はこのリストの先頭にイベントを追加する。

このリストから読み取ったイベントは、通常はそのイベントが最初に読み取られたときにすでに一度追加されたときのように、カレントコマンドのキーシーケンスに (たとえばthis-command-keysにリターンされたときのように) 追加される。フォーム (t . *event*) の要素はカレントコマンドのキーシーケンスに *event*を強制的に追加する。

このリストから読み取った要素は通常は記録保持機能 (Section 40.13.2 [Recording Input], page 1118 を参照) により記録されるとともに、キーボードマクロ定義の間 (Section 21.16 [Keyboard Macros], page 420 を参照) は記録される。しかし (no-record . *event*) という形式の要素は、通常は記録されることなく *event*が処理される。

listify-key-sequence *key* [Function]

この関数は文字列かベクターの *key*を unread-command-eventsに put することができる個別のイベントのリストに変換する。

input-pending-p **&optional** *check-timers* [Function]

この関数はコマンド入力がカレントで読み取り可能かどうか判断する。入力が利用可能なら t、それ以外は nilを即座にリターンする。非常に稀だが入力が利用できないときは tをリターンする。

オプション引数 *check-timers*が非 nilなら、Emacs は準備ができるとすべてのタイマーを実行する。Section 40.11 [Timers], page 1114 を参照のこと。

last-input-event [Variable]

この変数は最後に読み取られた端末入力イベントがコマンドの一部なのか、それとも Lisp プログラムによる明示的なものなのかを記録する。

以下の例では文字 *1*(ASCIIコード 49) を Lisp プログラムが読み取っている。*C-e* (*C-x C-e*は式を評価するコマンドとする) が last-command-eventに値として残っている間は、それが last-input-eventの値となる。

```
(progn (print (read-char))
       (print last-command-event)
       last-input-event)
⊣ 49
⊣ 5
⇒ 49
```

while-no-input *body...* [Macro]

この構文は *body*フォームを実行して、入力が何も到着しない場合だけ最後のフォームの値をリターンする。*body*フォームを実行する間に何らかの入力が到着したら、それらの入力を abort する (quit のように機能する)。while-no-inputフォームは実際の quit により abort したら nil、入力の到着によって abort したら tをリターンする。

*body*の一部で inhibit-quitを非 nilにバインドすると、その部分の間に到着した入力はその部分が終わるまで abort しない。

両方の abort 条件を *body*により計算された可能なすべての値で区別できるようにしたければ、以下のようにコードを記述する:

```
(while-no-input
```

```
        (list
          (progn . body)))
```

`while-no-input-ignore-events` [Variable]
この変数は `while-no-input`が無視するべきスペシャルイベントのセッティングを可能にする。これはイベントシンボルのリスト (Section 21.7.11 [Event Examples], page 397 を参照)。

`discard-input` [Function]
この関数は端末入力バッファーの内容を破棄して定義処理中かもしれないキーボードマクロをキャンセルする。この関数は `nil`をリターンする。

以下の例ではフォームの評価開始直後にユーザーが数字か文字をタイプするかもしれない。`sleep-for`がスリープを終えた後に `discard-input`はスリープ中にタイプされた文字を破棄する。

```
(progn (sleep-for 2)
       (discard-input))
     ⇒ nil
```

21.9 スペシャルイベント

特定のスペシャルイベント *(special event)* は、読み取られると即座に非常に低レベルで処理されます。`read-event`関数はそれらのイベントを自身で処理してそれらを決してリターンしません。かわりにスペシャルイベント以外の最初のイベントを待ってそれをリターンします。

スペシャルイベントはエコーされず、決してキーシーケンスにグループ化されず、`last-command-event`や `(this-command-keys)`の値として出現することもありません。スペシャルイベントは数引数を破棄して、`unread-command-events`による読み戻しができず、キーボードマクロ内に出現することもなく、キーボードマクロ定義中にキーボードマクロに記録されることもありません。

しかしスペシャルイベントは読み取られた直後に `last-input-event`内に出現するので、これがイベント定義にたいして実際のイベントを探す方法になります。

イベント型 `iconify-frame`、`make-frame-visible`、`delete-frame`、`drag-n-drop`、`language-change`、および `sigusr1`ようなユーザーシグナルは通常はこの方法によって処理されます。何がスペシャルイベントで、スペシャルイベントをどのように処理するかを定義するキーマップは変数 `special-event-map` (Section 22.9 [Controlling Active Maps], page 432 を参照) の中にあります。

21.10 時間の経過や入力の待機

待機関数 (wait function) は特定の時間が経過するか、入力があるまで待機するようにデザインされています。たとえば計算の途中でユーザーがディスプレイを閲覧できるように一時停止したいときがあるかもしれません。`sit-for`は一時停止して画面を更新して、`sleep-for`は画面を更新せずに一時停止して入力が到着したら即座にリターンします。

`sit-for` *seconds* **&optional** *nodisp* [Function]
この関数は、(ユーザーからの保留中入力がければ) 再描画を行ってから *seconds*秒、または入力が利用可能になるまで待機する。`sit-for`の通常の目的は、表示したテキストをユーザーが読み取る時間を与えるためである。入力が何も到着せず (Section 21.8.6 [Event Input Misc], page 410 を参照)、時間をフルに待機したら `t`、それ以外は `nil`が値となる。

引数 *seconds*は整数である必要はない。浮動小数点数なら `sit-for`は少数点数の秒を待機する。整数の秒だけをサポートするいくつかのシステムでは *seconds*は切り捨てられる。

保留中の入力が存在しなければ、式 `(sit-for 0)`は遅延なしで再描画をリクエストする `(redisplay)`と等価である。Section 39.2 [Forcing Redisplay], page 981 を参照のこと。

*nodisp*が非 `nil`なら `sit-for`は再描画を行わないが、それでも入力が利用可能になると (またはタイムアウト時間が経過すると) 即座にリターンする。

batch モード (Section 40.17 [Batch Mode], page 1121 を参照) では、たとえ標準入力ディスクリプタからの入力でも割り込みできない。これは以下で説明する `sleep-for`でも同様。

`(sit-for` *seconds millisec nodisp*`)`のように 3 つの引数で `sit-for`を呼び出すことも可能だが、時代遅れだと考えられている。

`sleep-for` *seconds* **&optional** *millisec* [Function]

この関数は表示を更新せず単に *seconds*秒の間一時停止する。これは利用可能な入力に注意を払わない。この関数は `nil`をリターンする。

引数 *seconds*は整数である必要はない。浮動小数点数なら `sleep-for`は少数点数の秒を待機する。整数の秒だけをサポートするいくつかのシステムでは *seconds*は切り捨てられる。

オプション引数 *millisec*はミリ秒単位で追加の待機時間を指定する。これは *seconds*で指定された時間に追加される。システムが小数点数の秒数をサポートしなければ、非 0 の *millisec*を指定するとエラーとなる。

遅延を保証したければ `sleep-for`を使用すること。

現在時刻を取得する関数については Section 40.5 [Time of Day], page 1104 を参照してください。

21.11 quit

Lisp 関数を実行中に *C-g*をタイプすると、Emacs が何を行っていても Emacs を *quit(*中止、終了*)* させます。これはアクティブなコマンドループの最内に制御がリターンすることを意味します。

コマンドループがキーボード入力の待機中に *C-g*をタイプしても quit はしません。これは通常の入力文字として機能します。もっともシンプルなケースでは、通常 *C-g*は quit の効果をもつ `keyboard-quit`を実行するので区別はできません。しかしプレフィクスキーの後の *C-g*は、未定義のキー組み合わせになります。これはプレフィクスキーやプレフィクス引数も同様にキャンセルする効果をもちます。

ミニバッファー内では *C-g*は異なる定義をもち、それはミニバッファーを abort(失敗、中止、中断) します。これは実際にはミニバッファーを exit して quit します (単に quit するのはミニバッファー内のコマンドループにリターンするだろう)。*C-g*がなぜコマンドリーダーが入力読み取り時に直接 quit しないかという理由は、ミニバッファー内での *C-g*の意味をこの方法によって再定義可能にするためです。プレフィクスキーの後の *C-g*はミニバッファー内で再定義されておらず、プレフィクスキーおよびプレフィクス引数のキャンセルという通常の効果をもちます。もし *C-g*が常に直接 quit するならこれは不可能でしょう。

*C-g*が直接 quit を行うときは、変数 `quit-flag`を `t`にセットすることによってそれを行います。Emacs は適切なタイミングでこの変数をチェックして、`nil`でなじぇれば quit します。どのような方法でも `quit-flag`を非 `nil`にセットすると quit が発生します。

C コードのレベルでは任意の場所で quit を発生させることはできず、`quit-flag`をチェックする特別な場所でのみ quit が発生します。この理由は他の場所で quit すると、Emacs の内部状態で矛盾が生じるかもしれないからです。安全な場所まで quit が遅延されるので、quit が Emacs をクラッシュさせることがなくなります。

read-key-sequenceや read-quoted-charのような特定の関数は、たとえ入力を待機中でも quit を抑制します。quit するかわりに C-gは要求された入力として処理されます。read-key-sequenceの場合、これはコマンドループ内での C-gの特別な振る舞いを引き起こすのに役立ちます。read-quoted-charの場合、これは C-gをクォートするのに C-qを使用できるようにします。

変数 inhibit-quitを非 nil値にバインドすることにより、Lisp 関数の一部で quit を抑止できます。その場合は quit-flagが tにセットされていても、C-gの通常の結果である quit は抑止されます。letフォームの最後でこのバインディングが unwind されるなどして、結果として inhibit-quitは再び nilになります。このとき quit-flagが nilなら、即座に要求された quit が発生します。この挙動はプログラム中のクリティカルセクション内で quit が発生しないことを確実にしたいときに理想的です。

(read-quoted-charのような) いくつかの関数では、quit を起こさない特別な方法で C-gが処理されます。これは inhibit-quitを tにバインドして入力を読み取り、再び inhibit-quitが nilになる前に quit-flagを nilにセットすることにより行われます。以下はこれを行う方法を示すためのread-quoted-charの抜粋です。この例は入力の最初の文字の後で通常の quit を許す方法も示しています。

```
(defun read-quoted-char (&optional prompt)
  "...documentation..."
  (let ((message-log-max nil) done (first t) (code 0) char)
    (while (not done)
      (let ((inhibit-quit first)
            ...)
        (and prompt (message "%s-" prompt))
        (setq char (read-event))
        (if inhibit-quit (setq quit-flag nil)))
      ... 変数 codeをセット ...)
    code))
```

quit-flag [Variable]

この変数が非 nilで inhibit-quitが nilなら、Emacs は即座に quit する。C-gをタイプすると通常は inhibit-quitとは無関係に quit-flagを非 nilにセットする。

inhibit-quit [Variable]

この変数は quit-flagが非 nilにセットされているとき Emacs が quit するかどうかを決定する。inhibit-quitが非 nilなら quit-flagに特に効果はない。

with-local-quit *body...* [Macro]

このマクロは *body*を順番に実行するが、たとえこの構文の外部で inhibit-quitが非 nilでも、少なくともローカルに *body*内での quit を許容する。このマクロは quit により exit したら nil、それ以外は *body*内の最後のフォームの値をリターンする。

inhibit-quitが nilなら with-local-quitへのエントリーで *body*だけが実行され、quit-flagをセットすることにより通常の quit が発生する。しかし通常の quit が遅延されるように inhibit-quitが非 nilにセットされていれば、非 nilの quit-flagは特別な種類のローカル quit を引き起こす。これは *body*の実行を終了して、quit-flagを非 nilのままにして with-local-quitの body を exit するので、許され次第 (通常の) 別の quit が発生する。*body*の先頭ですでに quit-flagが非 nilなら即座にローカル quit が発生して結局 body は実行されない。

このマクロは主にタイマー、プロセスフィルター、プロセスセンチネル、`pre-command-hook`、`post-command-hook`、および`inhibit-quit`が通常のように`t`にバイドされている場所で役に立つ。

`keyboard-quit` [Command]
この関数は`(signal 'quit nil)`によって`quit`条件をシグナルする。これは quit が行うことと同じ (Section 11.7.3 [Errors], page 164 の`signal`を参照)。

quit に使用する *C-g*以外の文字を指定できます。Section 40.13.1 [Input Modes], page 1118 内の関数`set-input-mode`を参照してください。

21.12 プレフィクスコマンド引数

ほとんどの Emacs コマンドはプレフィクス引数 *(prefix argument)* を使用できます。プレフィクス引数はコマンド自身の前に数字を指定するものです (プレフィクス引数とプレフィクスキーを混同しないこと)。プレフィクス引数は常に値により表され、`nil`のときはカレントでプレフィクス引数が存在しないことを意味します。すべてのコマンドはプレフィクス引数を使用するか、あるいは無視します。

プレフィクス引数には 2 つの表現があります。それは *raw(*生の、加工していない、原料のままの、未加工の*)* と数字 *(numeric)* です。エディターコマンドループは内部的に raw 表現を使用し、Lisp 変数もその情報を格納するのにこれを使用しますが、コマンドはいずれかの表現を要求できます。

以下は利用できる raw プレフィクス引数の値です:

- `nil`はプレフィクス引数がないことを意味する。これの数値的な値は 1 だが多くのコマンドは`nil`と整数 1 を区別する。
- 整数はそれ自身を意味する。
- 整数の要素を 1 つもつリスト。プレフィクス引数のこの形式は、1 つまたは数字無しの連続する *C-u*の結果である。数値的な値はリスト内の整数だが、そのようなリストと単独の整数を区別するコマンドがいくつかある。
- シンボル-。これは後に数字をともなわない *M--*か *C-u -*がタイプされたことを示す。数値的に等価な値は -1 だが、整数の -1 をシンボルの-を区別するコマンドがいくつかある。

以下の関数をさまざまなプレフィクスで呼び出して、これらの可能なプレフィクスを説明しましょう:

```
(defun display-prefix (arg)
  "raw プレフィクス引数の値を表示する"
  (interactive "P")
  (message "%s" arg))
```

以下はさまざまな raw プレフィクス引数で`display-prefix`を呼び出した結果です:

```
          M-x display-prefix  ⊣ nil

C-u       M-x display-prefix  ⊣ (4)

C-u C-u M-x display-prefix  ⊣ (16)

C-u 3   M-x display-prefix  ⊣ 3

M-3     M-x display-prefix  ⊣ 3      ; (C-u 3と同じ)
```

```
C-u -   M-x display-prefix  ⊣ -

M--     M-x display-prefix  ⊣ -       ; (C-u -と同じ)

C-u - 7 M-x display-prefix  ⊣ -7

M-- 7   M-x display-prefix  ⊣ -7      ; (C-u -7と同じ)
```

Emacs にはプレフィクス引数を格納するために 2 つの変数prefix-argと current-prefix-arg があります。他のコマンドにたいしてプレフィクス引数をセットアップする universal-argumentのようなコマンドは、プレフィクス引数を prefix-arg内に格納します。対照的に current-prefix-argはカレントコマンドにプレフィクス引数を引き渡すので、これらの変数をセットしても将来のコマンドにたいするプレフィクス引数に効果はありません。

コマンドは通常は interactive内で、プレフィクス引数にたいして raw と数値のどちらの表現を使用するかを指定します (Section 21.2.1 [Using Interactive], page 376 を参照)。そのかわりに関数は変数 current-prefix-arg内のプレフィクス引数の値を直接調べるかもしれませんが、これは明確さで劣っています。

prefix-numeric-value *arg* [Function]
: この関数は *arg*の有効な raw プレフィクス引数の数値的な意味をリターンする。引数はシンボル、数字、またはリストかもしれない。これが nilなら値 1、-なら −1 がリターンされる。これが数字なら、その数字がリターンされる。リスト (数字であること) なら、そのリストの CAR がリターンされる。

current-prefix-arg [Variable]
: この変数はカレントのコマンドにたいする raw プレフィクス引数を保持する。コマンドはこの変数を直接調べるかもしれないが、この変数にたいするアクセスには通常は (interactive "P")を使用する。

prefix-arg [Variable]
: この変数の値は次の編集コマンドにたいする raw プレフィクス引数である。後続のコマンドにたいしてプレフィクス引数を指定する universal-argumentのようなコマンドは、この変数をセットすることによって機能する。

last-prefix-arg [Variable]
: この raw プレフィクス引数の値は、前のコマンドにより使用された値である。

以下のコマンドは、後続のコマンドにたいしてプレフィクス引数をセットアップするために存在します。これらを他の用途で呼び出さないでください。

universal-argument [Command]
: このコマンドは入力を読み取って、後続のコマンドにたいするプレフィクス引数を指定する。何をしているかわかっているのでなければ、このコマンドを自分で呼び出してはならない。

digit-argument *arg* [Command]
: このコマンドは、後続のコマンドにたいしてプレフィクス引数を追加する。引数 *arg*はこのコマンドの前の raw プレフィクス引数であり、これはプレフィクス引数を更新するために使用される。何をしているかわかっているのでなければ、このコマンドを自分で呼び出してはならない。

`negative-argument` *arg* [Command]
: このコマンドは、次のコマンドにたいして数引数を追加する。引数 *arg*はこのコマンドの前の raw プレフィクス引数であり、この値に負の符号が付されて新しいプレフィクス引数を構築する。何をしているかわかっているのでなければ、このコマンドを自分で呼び出してはならない。

21.13 再帰編集

Emacs はスタートアップ時に、自動的に Emacs コマンドループに移行します。このトップレベルのコマンドループ呼び出しは決して exit することなく、Emacs 実行中は実行を継続します。Lisp プログラムもコマンドループを呼び出せます。これは複数のコマンドループを活性化するので、再帰編集 *(recursive editing)* と呼ばれています。再帰編集レベルは呼び出したコマンドが何であれそれをサスペンドして、そのコマンドを再開する前にユーザーが任意の編集を行うことを可能にする効果をもちます。

再帰編集の間に利用可能なコマンドは、トップレベルの編集ループ内で利用できるコマンドと同じでありキーマップ内で定義されます。数少ない特別なコマンドだけが再帰編集レベルを exit して、他のコマンドは再帰編集レベルが終了したときに再帰編集レベルからリターンします (exit するための特別なコマンドは常に利用できるが再帰編集が行われていないときは何も行わない)。

再帰コマンドループを含むすべてのコマンドループは、コマンドループから実行されたコマンド内のエラーによってそのループを exit しないように、汎用エラーハンドラーをセットアップします。

ミニバッファー入力は特殊な再帰編集です。これはミニバッファーとミニバッファーウィンドウの表示を有効にするなどの欠点をもちますが、それはあなたが思うより少ないでしょう。ミニバッファー内では特定のキーの振る舞いが異なりますが、これははミニバッファーのローカルマップによるものです。ウィンドウを切り替えれば通常の Emacs コマンドを使用できます。

再帰編集レベルを呼び出すには関数 `recursive-edit`を呼び出します。この関数はコマンドループを含んでいます。さらに `exit`を throw することにより再帰編集レベルの exit を可能にする、タグ `exit`をともなった `catch`呼び出しも含んでいます (Section 11.7.1 [Catch and Throw], page 162 を参照)。`t`以外の値を throw すると、`recursive-edit`は通常はそれを呼び出した関数にリターンします。コマンド *C-M-c* (`exit-recursive-edit`) がこれを行います。値 `t`を throw することによって `recursive-edit`が quit されるので、1 レベル上位のコマンドループに制御がリターンされます。これは *abort* と呼ばれ、*C-]*(`abort-recursive-edit`) がこれを行います。

ほとんどのアプリケーションはミニバッファー使用の一部として使用する場合を除き、再帰編集を使用するべきではありません。カレントバッファーのメジャーモードから特殊なメジャーモードに一時的に変更する場合に、そのモードに戻るコマンドをもつ必要があるときは、通常は再帰編集のほうが便利です (Rmail の *e*コマンドはこのテクニックを使用)。またはユーザーが新たなバッファーの特殊なモードで、異なるテキストを再帰的に編集・作成・選択できるようにしたい場合が該当します。このモードでは処理を完了させるコマンドを定義して前のバッファーに戻ります (Rmail の *m*コマンドはこれを使用)。

再帰編集はデバッグに便利です。一種のブレークポイントとして関数定義内に `debug`を挿入して、関数がそこに達したときにその箇所を調べることができます。`debug`は再帰編集を呼び出しますが、デバッガのその他の機能も提供します。

`query-replace`内で *C-r*をタイプしたときや *C-x q* (`kbd-macro-query`) を使用したときにも再帰編集レベルが使用されます。

`recursive-edit` [Command]

この関数はエディターコマンドループを呼び出す。これはユーザーに編集を開始させるために、Emacs の初期化により自動的に呼び出される。Lisp プログラムから呼び出されたときは再帰編集レベルにエンターする。

カレントバッファーが選択されたウィンドウのバッファーと異なる場合、`recursive-edit` はカレントバッファーの保存とリストアを行う。それ以外ではバッファーを切り替えると、`recursive-edit`がリターンした後にその切り替えたバッファーがカレントになる。

以下の例では関数 `simple-rec`が最初にポイントを 1 単語分進めてからメッセージをエコーエリアにプリントして再帰編集にエンターする。その後ユーザーは望む編集を行い、*`C-M-c`*をタイプすれば再帰編集を exit して `simple-rec`の実行を継続できる。

```
(defun simple-rec ()
  (forward-word 1)
  (message "Recursive edit in progress")
  (recursive-edit)
  (forward-word 1))
     ⇒ simple-rec
(simple-rec)
     ⇒ nil
```

`exit-recursive-edit` [Command]

この関数は最内の再帰編集 (ミニバッファー入力を含む) から exit する。関数の実質的な定義は `(throw 'exit nil)`。

`abort-recursive-edit` [Command]

この関数は再帰編集を exit した後に `quit`をシグナルすることにより、最内の再帰編集 (ミニバッファー入力を含む) を要求したコマンドを abort する。関数の実質的な定義は `(throw 'exit t)`。Section 21.11 [Quitting], page 413 を参照のこと。

`top-level` [Command]

この関数はすべての再帰編集レベルを exit する。これはすべての計算を直接抜け出してメインのコマンドループに戻って値をリターンしない。

`recursion-depth` [Function]

この関数は再帰編集のカレントの深さをリターンする。アクティブな再帰編集が存在しなければ 0 をリターンする。

21.14 コマンドの無効化

コマンドを無効化 *(disabling a command)* とは、それを実行可能にする前にユーザーによる確認を要求するようにコマンドをマークすることです。無効化は初めてのユーザーを混乱させるかもしれないコマンドにたいして、意図せずそのコマンドが使用されるのを防ぐために使用されます。

コマンド無効化の低レベルにおけるメカニズムは、そのコマンドにたいする Lisp シンボルの `disabled`プロパティに非 `nil`を put することです。これらのプロパティは、通常はユーザーの init ファイル (Section 40.1.2 [Init File], page 1093 を参照) で以下のような Lisp 式によりセットアップされます:

```
(put 'upcase-region 'disabled t)
```

いくつかのコマンドにたいしては、これらのプロパティがデフォルトで与えられています (これらを削除したければ init ファイルで削除できる)。

`disabled`プロパティの値が文字列なら、そのコマンドが無効化されていることを告げるメッセージにその文字列が含まれます。たとえば:

```
(put 'delete-region 'disabled
     "この方法で削除されたテキストは yank で戻せない!\n")
```

無効化されたコマンドをインタラクティブに呼び出したときに何が起こるかの詳細は、Section "Disabling" in *The GNU Emacs Manual* を参照してください。コマンドの無効化は、それを Lisp プログラムから関数として呼び出したときは効果がありません。

`enable-command` *command* [Command]
: その時点から特別な確認なしで *command*(シンボル) が実行されることを許す。さらにユーザーの init ファイル (Section 40.1.2 [Init File], page 1093 を参照) も修正するので将来のセッションにもこれが適用される。

`disable-command` *command* [Command]
: その時点から *command*(シンボル) の実行に特別な確認を要求する。さらにユーザーの init ファイル (Section 40.1.2 [Init File], page 1093 を参照) も修正するので将来のセッションにもこれが適用される。

`disabled-command-function` [Variable]
: この変数の値は関数であること。ユーザーが無効化されたコマンドを呼び出したときは無効化されたコマンドのかわりにその関数が呼び出される。そのコマンドを実行するためにユーザーが何のキーをタイプしたかを判断するために `this-command-keys`を使用して、そのコマンド自体を探すことができる。

  値は `nil`もあり得る。その場合にはたとえ無効化されたコマンドでも、すべてのコマンドが通常のように機能する。

  デフォルトでは値はユーザーに処理を行うかどうかを尋ねる関数。

21.15 コマンドのヒストリー

コマンドループは複雑なコマンドを手軽に繰り返せるように、すでに実行された複雑なコマンドのヒストリー (history: 履歴) を保持します。複雑なコマンド *(complex command)* とは、ミニバッファーを使用して interactive 引数を読み取るコマンドです。これには *`M-x`*コマンド、*`M-:`*コマンド、および`interactive`指定によりミニバッファーから引数を読み取るすべてのコマンドが含まれます。コマンド自身の実行の間に明示的にミニバッファーを使用するものは、複雑なコマンドとは判断されません。

`command-history` [Variable]
: この変数の値は最近実行された複雑なコマンドのリストであり、それぞれが評価されるべきフォームとして表現される。このリストは編集セッションの間、すべての複雑なコマンドを蓄積するが、最大サイズ (Section 20.4 [Minibuffer History], page 348 を参照) に達したときは、もっとも古い要素が削除されて新たな要素が追加される。

  ```
  command-history
  ⇒ ((switch-to-buffer "chistory.texi")
      (describe-key "^X^[")
      (visit-tags-table "~/emacs/src/")
      (find-tag "repeat-complex-command"))
  ```

このヒストリーリストは実際にはミニバッファーヒストリーの特殊ケースであり、それは要素が文字列ではなく式であることです。

以前のコマンドを編集したり再呼び出しするためのコマンドがいくつかあります。コマンド`repeat-complex-command`と`list-command-history`はユーザーマニュアルに説明されています (Section “Repetition” in *The GNU Emacs Manual* を参照)。ミニバッファー内では通常のミニバッファーヒストリーコマンドが利用できます。

21.16 キーボードマクロ

キーボードマクロ *(keyboard macro)* はコマンドとして考えることが可能な入力イベントの記録されたシーケンスであり、キー定義によって作成されます。キーボードマクロの Lisp 表現はイベントを含む文字列かベクターです。キーボードマクロと Lisp マクロ (Chapter 14 [Macros], page 242 を参照) を混同しないでください。

`execute-kbd-macro` *kbdmacro* **&optional** *count loopfunc* [Function]

この関数はイベントシーケンスとして *kbdmacro*を実行する。*kbdmacro*が文字列かベクターなら、たとえそれがユーザーによる入力であっても、その中のイベントは忠実に実行される。シーケンスは単一のキーシーケンスであることを要求されない。キーボードマクロ定義は、通常は複数のキーシーケンスを結合して構成される。

*kbdmacro*がシンボルなら、そのシンボルの関数定義は *kbdmacro*の箇所に使用される。それが別のシンボルならこのプロセスを繰り返す。最終的に結果は文字列かベクターになる。結果がシンボル、文字列、ベクターでなければエラーがシグナルされる。

引数 *count*は繰り返すカウントであり、*kbdmacro*がその回数実行される。*count*が省略または`nil`なら 1 回実行される。0 なら *kbdmacro*はエラーに遭遇するか検索が失敗するまで何度も実行される。

*loopfunc*が非`nil`なら、それはマクロの繰り返しごとに呼び出される引数なしの関数である。*loopfunc*が`nil`をリターンするとマクロの実行が停止する。

`execute-kbd-macro`の使用例は Section 21.8.2 [Reading One Event], page 405 を参照のこと。

`executing-kbd-macro` [Variable]

この変数はカレントで実行中のキーボードマクロを定義する文字列かベクター。`nil`ならカレントで実行中のマクロは存在しない。マクロの実行により実行されたときに異なる振る舞いをするように、コマンドはこの変数をテストできる。この変数を自分でセットしてはならない。

`defining-kbd-macro` [Variable]

この変数はキーボードマクロの定義中のときだけ非`nil`である。マクロ定義中の間は異なる振る舞いをするように、コマンドはこの変数をテストできる。既存のマクロ定義に追加する間、値は`append`になる。コマンド`start-kbd-macro`、`kmacro-start-macro`、`end-kbd-macro`はこの変数をセットする。この変数を自分でセットしてはならない。

この変数は常にカレント端末にたいしてローカルであり、バッファーローカルにできない。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`last-kbd-macro` [Variable]

この変数はもっとも最近定義されたキーボードマクロの定義である。値は文字列、ベクター、または`nil`。

この変数は常にカレント端末にたいしてローカルであり、バッファーローカルにできない。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`kbd-macro-termination-hook` [Variable]

これはキーボードマクロが終了したときに実行されるノーマルフックであり、何がキーボードマクロを終了させたか (マクロの最後に到達したのか、あるいはエラーにより最後到達する前に終了したのか) は問わない。

22 キーマップ

入力イベントのコマンドバインディングはキーマップ *(keymap)* と呼ばれるデータ構造に記録されます。キーマップ内の各エントリーは個別のイベント型 (他のキーマップ、またはコマンド) に関連づけ (またはバインド) されます。イベント型がキーマップにバインドされていれば、そのキーマップは次の入力イベントを調べるために使用されます。これはコマンドが見つかるまで継続されます。このプロセス全体をキールックアップ *(key lookup:* キーの照合*)* と呼びます。

22.1 キーシーケンス

キーシーケンス *(key sequence)*、短くはキー *(key)* とは 1 つの単位を形成する 1 つ以上の入力イベントのシーケンスです。入力イベントには文字、ファンクションキー、マウスアクション、または `iconify-frame`のような Emacs 外部のシステムイベントが含まれます (Section 21.7 [Input Events], page 387 を参照)。キーシーケンスにたいする Emacs Lisp の表現は文字列かベクターです。特に明記しない限り、引数としてキーシーケンスを受け取る Emacs Lisp 関数は両方の表現を処理することができます。

文字列表現ではたとえば`"a"`は `a`、`"2"`は *2*を表すといったように、英数字はその文字自身を意味します。コントロール文字イベントは部分文字列`"\C-"`、メタ文字は`"\M-"`によりプレフィクスされます。たとえば`"\C-x"`はキー *C-x*を表します。それらに加えて `TAB`、`RET`、`ESC`、`DEL`などのイベントはそれぞれ`"\t"`、`"\r"`、`"\e"`、`"\d"`で表されます。複雑なキーシーケンスの文字列表現はイベント成分の文字列表現を結合したものです。したがって`"\C-xl"`はキーシーケンス *C-x l*を表します。

キーシーケンスにはファンクションキー、マウスボタンイベント、システムイベント、または *C-=* や *H-a*のような文字列で表現できない非 ASCII文字が含まれます。これらはベクターとして表現する必要があります。

ベクター表現ではベクターの各要素は 1 つの入力イベントをイベントの Lisp 形式で表します。Section 21.7 [Input Events], page 387 を参照してください。たとえばベクター `[?\C-x ?l]`はキーシーケンス *C-x l*を表します。

キーシーケンスを文字列やベクターによる表現で記述する例は、Section “Init Rebinding” in *The GNU Emacs Manual* を参照してください。

`kbd` *keyseq-text* [Function]

この関数はテキスト *keyseq-text*(文字列定数) をキーシーケンス (文字列かベクターの定数) に変換する。*keyseq-text*の内容は *C-x C-k RET* (`kmacro-edit-macro`) コマンドにより呼び出されたバッファー内と同じ構文を使用するべきである。特にファンクションキーの名前は ‘`<...>`’で囲まなければならない。Section “Edit Keyboard Macro” in *The GNU Emacs Manual* を参照のこと。

```
(kbd "C-x") ⇒ "\C-x"
(kbd "C-x C-f") ⇒ "\C-x\C-f"
(kbd "C-x 4 C-f") ⇒ "\C-x4\C-f"
(kbd "X") ⇒ "X"
(kbd "RET") ⇒ "\^M"
(kbd "C-c SPC") ⇒ "\C-c "
(kbd "<f1> SPC") ⇒ [f1 32]
(kbd "C-M-<down>") ⇒ [C-M-down]
```

22.2 キーマップの基礎

キーマップはさまざまなキーシーケンスにたいしてキーバインディング *(key binding)* を指定する Lisp データ構造です。

1 つのキーマップが、個々のイベントにたいする定義を直接指定します。単一のイベントでキーシーケンスが構成されるとき、そのキーシーケンスのキーマップ内でのバインディングは、そのイベントにたいするそのキーマップの定義です。それより長いキーシーケンスのバインディングは対話的プロセスによって見つけ出されます。まず最初にイベント (それ自身がキーマップでなければならない) の定義を探します。そして次にそのキーマップ内で 2 つ目のイベントを探すといったように、そのキーシーケンス内のすべてのイベントが処理されるまで、これを続けます。

あるキーシーケンスのバインディングがキーマップであるような場合、わたしたちはそのキーシーケンスをプレフィクスキー *(prefix key)* と呼び、それ以外の場合には (それ以上イベントを追加できないので) コンプリートキー *(complete keyl* と呼んでいます。バインディングが `nil`の場合、わたしたちはそのキーを未定義 *(undefined)* と呼びます。*C-c*、*C-x*、*C-x 4*などはプレフィクスキーの例です。*X*、RET、*C-x 4 C-f*などは定義されたコンプリートキーの例です。*C-x C-g*や *C-c 3*などは未定義なコンプリートキーの例です。詳細は Section 22.6 [Prefix Keys], page 428 を参照してください。

キーシーケンスのバインディングを見つけ出すルールは、(最後のイベントの前までに見つかる) 中間的なバインディングがすべてキーマップであると仮定します。もしそうでなければ、そのイベントシーケンスは単位を形成せず、実際の単一キーシーケンスではありません。言い換えると任意の有効なキーシーケンスから 1 つ以上のイベントを取り除くと、常にプレフィクスキーにならなければなりません。たとえば *C-f C-n*はキーシーケンスではありません。*C-f*はプレフィクスキーではないので、*C-f*で始まるこれより長いシーケンスは、キーシーケンスではあり得ないからです。

利用可能な複数イベントキーシーケンスのセットは、プレフィクスキーにたいするバインディングに依存します。したがってこれはキーマップが異なれば異なるかもしれず、バインディングが変更されたときに変更されるかもしれません。しかし単一イベントキーシーケンスは整合性において任意のプレフィクスキーに依存しないので、常に単一のキーシーケンスです。

常に複数のプライマリーキーマップ (primary keymap: 主キーマップ) がアクティブであり、これらはキーバインディングを見つけるために使用されます。すべてのバッファーで共有されるグローバルキーマップ *(global map)* というキーマップが存在します。ローカルキーマップ *(local keymap)* は通常は特定のメジャーモードに関連します。そして 0 個以上のマイナーモードキーマップ *(minor mode keymap)* はカレントで有効なマイナーモードに属します (すべてのマイナーモードがキーマップをもつわけでなない)。ローカルキーマップは対応するグローバルバインディングを shadow(訳注: 隠すという意味) します。マイナーモードキーマップは、ローカルキーマップとグローバルキーマップの両方を shadow します。詳細は Section 22.7 [Active Keymaps], page 429 を参照してください。

22.3 キーマップのフォーマット

キーマップはそれぞれ CAR がシンボル `keymap`であるようなリストです。このリストの残りの要素はそのキーマップのキーバインディングを定義します。関数定義がキーマップであるようなシンボルもキーマップです。あるオブジェクトがキーマップかどうかテストするには、関数 `keymapp`(以下参照) を使用してください。

キーマップを開始するシンボル `keymap`の後には、いくつかの種類の要素が出現します:

`(`*type* `.` *binding*`)`
: これは型 *type*のイベントにたいする 1 つのバインディングを指定する。通常のバインディングはそれぞれ、常に文字かシンボルであるような特定のイベント型 *(event type)*

のイベントに適用される。Section 21.7.12 [Classifying Events], page 398 を参照のこと。この種のバインディングでは、*binding*はコマンドである。

(*type item-name . binding*)
: これはメニュー内で *item-name*として表示されるシンプルなメニューアイテムでもあるようなバインディングを指定する。Section 22.17.1.1 [Simple Menu Items], page 448 を参照のこと。

(*type item-name help-string . binding*)
: これはヘルプ文字列 *help-string*のシンプルなメニューアイテムである。

(*type* menu-item . *details*)
: これは拡張されたメニューアイテムでもあるようなバインディングを指定する。これは他の機能も使用できる。Section 22.17.1.2 [Extended Menu Items], page 448 を参照のこと。

(t . *binding*)
: これはデフォルトキーバインディング *(default key binding)* を指定する。キーマップの他の要素でバインドされないイベントは、バインディングとして *binding*が与えられる。デフォルトバインディングにより、利用可能なすべてのイベント型を列挙することなくバインドできる。デフォルトバインディングをもつキーマップは、明示的に `nil`にバインドされるイベント (以下参照) を除いて、より低い優先度にあるすべてのキーマップをマスクする。

char-table
: キーマップのある要素が文字テーブル (char-table) なら、それは修飾ビットなしのすべての文字イベントにたいするバインディングを保持するとみなされる ([modifier bits], page 14 を参照)。インデックス *c*の要素は文字 *c*にたいしてバインドされる。これは多量のバインディングを記録するためのコンパクトな方法である。そのような文字テーブルのキーマップは、*full* キーマップ *(full keymap:* 完全なキーマップ*)* と呼ばれる。それにたいして他のキーマップは *sparse* キーマップ *(sparse keymaps:* 疎なキーマップ*)* と呼ばれる。

vector
: この種の要素は文字テーブルと類似する。インデックス *c*の要素は文字 *c*にバインドされる。この方法でバインド可能な文字の範囲はそのベクターのサイズに制限され、かつベクターの作成により 0 からすべての文字コードまでスペースが割り当てられるので、バインディング自身が問題とならないメニューキーマップ (Section 22.17 [Menu Keymaps], page 447 を参照) の作成以外では、このフォーマットを使用しないこと。

string
: キーにたいするバインディングを指定する要素は別として、キーマップは要素として文字列ももつことができる。これは *overall* プロンプト文字列 *(overall prompt string:* 全般的なプロンプト文字列*)* と呼ばれ、メニューとしてキーマップを使用することを可能にする。Section 22.17.1 [Defining Menus], page 447 を参照のこと。

(keymap ...)
: キーマップのある要素それ自身がキーマップなら、外側のキーマップ内でこれが内側のキーマップとして inline 指定されているかのようにみなされる。これは `make-composed-keymap`内で行なわれるような多重継承にたいして使用される。

バインディングが`nil`なら、それは定義の構成要素ではありませんが、デフォルトバインディングや親キーマップ内のバインディングに優先されます。一方 `nil`のバインディングは、より低い優先度

のキーマップをオーバーライドしません。したがってローカルマップで nilのバインディングが与えられると、Emacs はグローバルマップのバインディングを使用します。

キーマップはメタ文字にたいするバインディングを直接記録しません。かわりにメタ文字は 1 文字目が ESC(または何であれ meta-prefix-charのカレント値) であるような、2 文字のキーシーケンスをルックアップするものとみなされます。したがってキー *M-a*は内部的には *ESC* aで表され、そのグローバルバインディングは esc-map内の aにたいするスロットで見つけることができます (Section 22.6 [Prefix Keys], page 428 を参照)。

この変換は文字にたいしてのみ適用され、ファンクションキーや他の入力イベントには適用されないので *M-end*は *ESC end*と何も関係ありません。

以下に例として Lisp モードにたいするローカルキーマップ (sparse キーマップ) を挙げます。以下では DEL、*C-c C-z*、*C-M-q*、*C-M-x*にたいするバインディングを定義しています (実際の値はメニューバインディングも含まれるが簡潔にするためここでは省略)。

```
lisp-mode-map
⇒
(keymap
 (3 keymap
    ;; C-c C-z
    (26 . run-lisp))
 (27 keymap
     ;; C-M-xは ESC C-xとして扱われる
     (24 . lisp-send-defun))
 ;; この部分は lisp-mode-shared-mapから継承
 keymap
 ;; DEL
 (127 . backward-delete-char-untabify)
 (27 keymap
     ;; C-M-qは ESC C-qとして扱われる
     (17 . indent-sexp)))
```

keymapp *object* [Function]

この関数は *object*がキーマップなら t、それ以外は nilをリターンする。より正確にはこの関数はリストにたいしてその CAR が keymapか、あるいはシンボルにたいしてその関数定義が keymappかどうかをテストする。

```
(keymapp '(keymap))
    ⇒ t
(fset 'foo '(keymap))
(keymapp 'foo)
    ⇒ t
(keymapp (current-global-map))
    ⇒ t
```

22.4 キーマップの作成

以下はキーマップを作成する関数です。

`make-sparse-keymap` **&optional** *prompt* [Function]

この関数はエントリーをもたない新たな sparse キーマップを作成してそれをリターンする (sparse キーマップはあなたが通常望む類のキーマップのこと)。`make-keymap`と異なり新たなキーマップは文字テーブルを含まず、何のイベントもバインドしない。

```
(make-sparse-keymap)
    ⇒ (keymap)
```

*prompt*を指定すると、それはキーマップにたいする overall プロンプト文字列になる。これはメニューキーマップ (Section 22.17.1 [Defining Menus], page 447 を参照) にたいしてのみ指定すべきである。overall プロンプト文字列をともなうキーマップがアクティブなら、次の入力イベントのルックアップにたいしてマウスメニューとキーボードメニューを常に提示する。これはコマンドループにたいして毎回キーボードメニューを提示するので、overall プロンプト文字列をメインマップ、メジャーモードマップ、マイナーモードマップに指定しないこと。

`make-keymap` **&optional** *prompt* [Function]

この関数は新たな full キーマップを作成してそれをリターンする。このキーマップは修飾されないすべての文字にたいするスロットをもつ文字テーブル (Section 6.6 [Char-Tables], page 110 を参照) を含む。この新たなキーマップは初期状態ではすべての文字、およびその他の種類のイベントが `nil`にバインドされている。引数 *prompt*は `make-sparse-keymap`のようにプロンプト文字列を指定する。

```
(make-keymap)
    ⇒ (keymap #^[nil nil keymap nil nil nil ...])
```

full キーマップは多くのスロットを保持するときは sparse キーマップより効果的であり、少ししかスロットを保持しないときは sparse キーマップのほうが適している。

`copy-keymap` *keymap* [Function]

この関数は *keymap*のコピーをリターンする。これはほとんど必要ないだろう。ほとんど差のないキーマップが必要なら、コピーより以下のように継承を使用するべきである:

```
(let ((map (make-sparse-keymap)))
  (set-keymap-parent map <theirmap>)
  (define-key map ...)
  ...)
```

`copy-keymap`を処理する際には、*keymap*内でバインディングとして直接出現するすべてのキーマップも、すべてのレベルまで再帰的にコピーされる。しかしある文字の定義が関数定義にキーマップをもつ関数のときには再帰的なコピーは行われず、新たにコピーされたキーマップには同じシンボルがコピーされる。

```
(setq map (copy-keymap (current-local-map)))
⇒ (keymap
     ;; (これはメタ文字を実装する)
     (27 keymap
         (83 . center-paragraph)
         (115 . center-line))
     (9 . tab-to-tab-stop))

(eq map (current-local-map))
    ⇒ nil
(equal map (current-local-map))
    ⇒ t
```

22.5 継承とキーマップ

キーマップは他のキーマップを継承することができ、この継承元のキーマップを親キーマップ *(parent keymap)* と呼びます。そのようなキーマップは以下のようなキーマップです:

```
(keymap elements... . parent-keymap)
```

これにはそのキーマップのキールックアップ時に *parent-keymap*のすべてのバインディングを継承するものの、それらにバインディングを追加したり *elements*でオーバーライドできるという効果があります。

`define-key`や他のキーバインディング関数を使用して *parent-keymap*内のバインディングを変更すると、変更されたバインディングは *elements*で作られたバインディングに shadow されない限り継承されたキーマップ内で可視になります。逆は真ではありません。`define-key`を使用して継承されたキーマップ内のバインディングを変更すると、これらの変更は *elements*内に記録されますが *parent-keymap*に影響はありません。

親キーマップからキーマップを構築するには`set-keymap-parent`を使用するのが正しい方法です。親キーマップから直接キーマップを構築するコードがあるなら、かわりに`set-keymap-parent`を使用するようにプログラムを変更してください。

`keymap-parent` *keymap* [Function]
: これは *keymap*の親キーマップをリターンする。*keymap*に親キーマップがなければ`keymap-parent`は`nil`をリターンする。

`set-keymap-parent` *keymap parent* [Function]
: これは *keymap*の親キーマップを *parent*にセットして *parent*をリターンする。*parent*が`nil`ならこの関数は *keymap*に親キーマップを与えない。

  *keymap*がサブマップ (プレフィクスキーにたいするバインディング) をもつ場合は、それらも新たな親キーマップを受け取ってそれらのプレフィクスキーにたいして *parent*が何を指定するかが反映される。

以下は`text-mode-map`から継承してキーマップを作成する方法を示す例です:

```
(let ((map (make-sparse-keymap)))
  (set-keymap-parent map text-mode-map)
  map)
```

非 sparse キーマップも親キーマップをもつことができますが便利とは言えません。非 sparse キーマップは修飾ビットをもたないすべての数値文字コードにたいするバインディングとして、たとえそれが`nil`であっても常に何かを指定するので、これらの文字のバインディングが親キーマップから継承されることは決してないのです。

複数のマップからキーマップを継承したいときがあるかもしれません。これにたいして関数`make-composed-keymap`が使用できます。

`make-composed-keymap` *maps* **&optional** *parent* [Function]
: この関数は既存のキーマップから構成される新たなキーマップをリターンする。またオプションで親キーマップ *parent*から継承を行う。*maps*には単一のキーマップ、または複数のキーマップのリストを指定できる。リターンされた新たなマップ内でキーをルックアップするとき、Emacs は *maps*内のキーマップを順に検索してから *parent*内を検索する。この検索は最初のマッチで停止する。*maps*のいずれか 1 つのキーマップ内の`nil`バインディングは、*parent*内のすべてのバインディングをオーバーライドするが、*maps*にないキーマップの非`nil`なバインディングはオーバーライドしない。

For example, here is how Emacs sets the parent of たとえば以下は button-buffer-mapと special-mode-mapの両方を継承する help-mode-mapのようなキーマップの親キーマップを Emacs がセットする方法です:

```
(defvar help-mode-map
  (let ((map (make-sparse-keymap)))
    (set-keymap-parent map
      (make-composed-keymap button-buffer-map special-mode-map))
    ... map) ... )
```

22.6 プレフィクスキー

プレフィクスキー *(prefix key)* とは、バインディングがキーマップであるようなキーシーケンスです。このキーマップはプレフィクスキーを拡張するキーシーケンスが何を行うかを定義します。たとえば *C-x*はプレフィクスキーであり、これはキーマップを使用してそのキーマップは変数 ctl-x-mapにも格納されています。このキーマップは *C-x*で始まるキーシーケンスにたいするバインディングを定義します。

標準的な Emacs のプレフィクスキーのいくつかは、Lisp 変数でも見い出すことができるキーマップを使用しています:

- esc-mapはプレフィクスキー ESCにたいするグローバルキーマップである。したがってすべてのメタ文字にたいする定義は、このキーマップで見い出すことができる。このマップは ESC-prefix の関数定義でもある。
- help-mapはプレフィクスキー *C-h*にたいするグローバルキーマップである。
- mode-specific-mapはプレフィクスキー *C-c*にたいするグローバルキーマップである。このマップは実際にはモード特有 (mode-specific) ではなくグローバルであるが、このプレフィクスキーは主にモード特有なバインディングに使用されるので、*C-h b* (display-bindings) の出力内の *C-c*に関する情報で、この名前は有意義な情報を提供する。
- ctl-x-mapはプレフィクスキー *C-x*にたいして使用されるグローバルキーマップである。このマップはシンボル Control-X-prefixの関数セルを通して見つけることができる。
- mule-keymapはプレフィクスキー *C-x RET* にたいして使用されるグローバルキーマップである。
- ctl-x-4-mapはプレフィクスキー *C-x 4*にたいして使用されるグローバルキーマップである。
- ctl-x-5-mapはプレフィクスキー *C-x 5*にたいして使用されるグローバルキーマップである。
- 2C-mode-mapはプレフィクスキー *C-x 6*にたいして使用されるグローバルキーマップである。
- tab-prefix-mapはプレフィクスキー *C-x t*にたいして使用されるグローバルキーマップである。
- vc-prefix-mapはプレフィクスキー *C-x v*にたいして使用されるグローバルキーマップである。
- goto-mapはプレフィクスキー *M-g*にたいして使用されるグローバルキーマップである。
- search-mapはプレフィクスキー *M-s*にたいして使用されるグローバルキーマップである。
- facemenu-keymapはプレフィクスキー *M-o*にたいして使用されるグローバルキーマップである。
- Emacs の他のプレフィクスキーには *C-x @*、*C-x a i*、*C-x ESC*、*ESC ESC*がある。これらは特別な名前をもたないキーマップを使用する。

プレフィクスキーのキーマップバインディングは、プレフィクスキーに続くイベントをルックアップするために使用されます (これは関数定義がキーマップであるようなシンボルかもしれない。効果は同じだがシンボルはプレフィクスキーにたいする名前の役割を果たす)。したがって *C-x*のバインディングはシンボル Control-X-prefixであり、このシンボルの関数セルが *C-x*コマンドにたいするキーマップを保持します (ctl-x-mapの値も同じキーマップ)。

プレフィクスキー定義は任意のアクティブなキーマップ内に置くことができます。プレフィクスキーとしての *C-c*、*C-x*、*C-h*、ESCの定義はグローバルマップ内にもあるので、これらのプレフィクスキーは常に使用できます。メジャーモードとマイナーモードは、ローカルマップやマイナーモードのマップ内にプレフィクスキー定義を置くことによってキーをプレフィクスキーとして再定義できます。Section 22.7 [Active Keymaps], page 429 を参照してください。

あるキーが複数のアクティブなマップ内でプレフィクスキーとして定義されていると、それぞれの定義がマージされて効果をもちます。まずマイナーモードキーマップ内で定義されたコマンド、次にローカルマップのプレフィクス定義されたコマンド、そしてグローバルマップのコマンドが続きます。

以下の例ではローカルキーマップ内で *C-p*を *C-x*と等価なプレフィクスキーにしています。すると *C-p C-f*にたいするバインディングは *C-x C-f*と同様に関数 `find-file`になります。キーシーケンス *C-p 6*はすべてのアクティブなキーマップで見つけることができません。

```
(use-local-map (make-sparse-keymap))
    ⇒ nil
(local-set-key "\C-p" ctl-x-map)
    ⇒ nil
(key-binding "\C-p\C-f")
    ⇒ find-file

(key-binding "\C-p6")
    ⇒ nil
```

`define-prefix-command` *symbol* **&optional** *mapvar prompt* [Function]

この関数はプレフィクスキーのバインディングとして使用するために *symbol*を用意する。これは sparse キーマップを作成してそれを *symbol*の関数定義として格納する。その後は *symbol* にキーシーケンスをバインディングすると、そのキーシーケンスはプレフィクスキーになるだろう。リターン値は `symbol`。

この関数は値がそのキーマップであるような変数としても *symbol*をセットする。しかし *mapvar* が非 `nil`なら、かわりに *mapvar*を変数としてセットする。

*prompt*が非 `nil`なら、これはそのキーマップにたいする overall プロンプト文字列になる。プロンプト文字列はメニューキーマップにたいして与えらること (Section 22.17.1 [Defining Menus], page 447 を参照)。

22.7 アクティブなキーマップ

Emacs には多くのキーマップを含まれていますが、常にいくつかのキーマップだけがアクティブです。Emacs がユーザー入力を受け取ったとき、それは入力イベントに変換されて (Section 22.14 [Translation Keymaps], page 442 を参照)、アクティブなキーマップ内でキーバインディングがルックアップされます。

アクティブなキーマップは通常は、(1) `keymap`プロパティにより指定されるキーマップ、(2) 有効なマイナーモードのキーマップ、(3) カレントバッファーのローカルキーマップ、(4) グローバルキーマップの順です。Emacs は入力キーシーケンスそれぞれにたいして、これらすべてのキーマップ内を検索します。

これらの通常のキーマップのうち最優先されるのは、もしあればポイント位置の `keymap`テキストにより指定されるキーマップか overall プロパティです (マウス入力イベントにたいしては Emacs はポイント位置のかわりにイベント位置を使用する。詳細は次のセクションを参照されたい)。

次に優先されるのは有効なマイナーモードにより指定されるキーマップです。もしあればこれらのキーマップは変数 `emulation-mode-map-alists`、`minor-mode-overriding-map-alist`、`minor-mode-map-alist`により指定されます。Section 22.9 [Controlling Active Maps], page 432 を参照してください。

次に優先されるのはバッファーのローカルキーマップ *(local keymap)* で、これにはそのバッファー特有なキーバインディングが含まれます。ミニバッファーもローカルキーマップをもちます (Section 20.1 [Intro to Minibuffers], page 342 を参照)。ポイント位置に `local-map`テキスト、または overlay プロパティがあるなら、それはバッファーのデフォルトローカルキーマップのかわりに使用するローカルキーマップを指定します。

ローカルキーマップは通常はそのバッファーのメジャーモードによってセットされます。同じメジャーモードをもつすべてのバッファーは、同じローカルキーマップを共有します。したがってあるバッファーでローカルキーマップを変更するために `local-set-key` (Section 22.15 [Key Binding Commands], page 444 を参照) を呼び出すと、それは同じメジャーモードをもつ他のバッファーのローカルキーマップにも影響を与えます。

最後は `C-f`のようなカレントバッファーとは関係なく定義されるキーバインディングを含んだグローバルキーマップ *(global keymap)* です。このキーマップは常にアクティブであり変数 `global-map` にバインドされています。

これら通常のキーマップとは別に、Emacs はプログラムが他のキーマップをアクティブにするための特別な手段を提供します。1 つ目はグローバルキーマップ以外の通常アクティブなキーマップを置き換えるキーマップを指定する変数 `overriding-local-map`です。2 つ目は他のすべてのキーマップより優先されるキーマップを指定する端末ローカル変数 `overriding-terminal-local-map`です。この端末ローカル変数は通常は modal(訳注: 他のキーマップを選択できない状態) かつ一時的なキーバインディングに使用されます (ここの変数にたいして関数 `set-transient-map`は便利なインターフェイスを提供する)。詳細は Section 22.9 [Controlling Active Maps], page 432 を参照してください。

これらを使用するのがキーマップをアクティブにする唯一の方法ではありません。キーマップは `read-key-sequence`によるイベントの変換のような他の用途にも使用されます。Section 22.14 [Translation Keymaps], page 442 を参照してください。

いくつかの標準的なキーマップのリストは Appendix G [Standard Keymaps], page 1220 を参照してください。

`current-active-maps` **&optional** *olp position* [Function]

これはカレント状況下でコマンドループによりキーシーケンスをルックアップするために使用される、アクティブなキーマップのリストをリターンする。これは通常は `overriding-local-map`と `overriding-terminal-local-map`を無視するが、*olp*が非 `nil`なら、それらのキーマップにも注意を払う。オプションで *position*に `event-start`によってリターンされるイベント位置、またはバッファー位置を指定でき、`key-binding`で説明されているようにキーマップを変更するかもしれない。

`key-binding` *key* **&optional** *accept-defaults no-remap position* [Function]

この関数はカレントのアクティブキーマップで *key*にたいするバインディングをリターンする。そのキーマップ内で *key*が未定義なら結果は `nil`。

引数 *accept-defaults*は `lookup-key` (Section 22.11 [Functions for Key Lookup], page 436 を参照) のようにデフォルトバインディングをチェックするかどうかを制御する。

コマンドがリマップ (remap: 再マップ。Section 22.13 [Remapping Commands], page 441 を参照) されたとき、`key-binding`が実際に実行されるであろうリマップされたコマンドをリ

ターンするように、通常のようにコマンドのリマップを行う。しかし *no-remap*が非 `nil`なら、`key-binding`はリマップを無視して *key*にたいして直接指定されたバインディングをリターンする。

*key*がマウスイベント (もしかしたらプレフィクスイベントが先行するかもしれない) で始まるなら、照合されるマップはそのイベントの位置を元に決定される。それ以外では、それらのマップはポイント値にもとづき決定される。しかし *position*を指定することによってこれらをオーバーライドできる。*position*が非 `nil`なら、それはバッファー位置か `event-start`の値のようなイベント位置のいずれかである。その場合には照合されるマップは *position*にもとづいて決定される。

*key*が文字列とベクターのいずれでもなければ Emacs はエラーをシグナルする。

```
(key-binding "\C-x\C-f")
    ⇒ find-file
```

22.8 アクティブなキーマップの検索

以下は Emacs がアクティブなキーマップを検索する方法を示す Lisp 処理の概要です:

```
(or (if overriding-terminal-local-map
        (find-in overriding-terminal-local-map))
    (if overriding-local-map
        (find-in overriding-local-map)
      (or (find-in (get-char-property (point) 'keymap))
          (find-in-any emulation-mode-map-alists)
          (find-in-any minor-mode-overriding-map-alist)
          (find-in-any minor-mode-map-alist)
          (if (get-text-property (point) 'local-map)
              (find-in (get-char-property (point) 'local-map))
            (find-in (current-local-map)))))
    (find-in (current-global-map)))
```

ここで *find-in*と *find-in-any*はそれぞれ、1 つのキーマップとキーマップの alist を検索する仮の関数です。関数 `set-transient-map`が `overriding-terminal-local-map` (Section 22.9 [Controlling Active Maps], page 432 を参照) をセットすることによって機能する点に注意してください。

上記の処理概要ではキーシーケンスがマウスイベント (Section 21.7.3 [Mouse Events], page 390 を参照) で始まる場合には、ポイント位置のかわりにそのイベント位置、カレントバッファーのかわりにそのイベントのバッファーが使用されます。これは特にプロパティ`keymap`と `local-map`をルックアップする方法に影響を与えます。`display`、`before-string`、`after-string`プロパティ(Section 32.19.4 [Special Properties], page 820 を参照) が埋め込まれていて `keymap`か `local-map`プロパティが非 `nil`の文字列上でマウスイベントが発生すると、それは基調となるバッファーテキストの対応するプロパティをオーバーライドします (バッファーテキストにより指定されたプロパティは無視される)。

アクティブなキーマップの 1 つでキーバインディングが見つかって、そのバインディングがコマンドなら検索は終了してそのコマンドが実行されます。しかしそのバインディングが値をもつ変数か文字列なら、Emacs は入力キーシーケンスをその変数の値か文字列で置き換えて、アクティブなキーマップの検索を再開します。Section 22.10 [Key Lookup], page 434 を参照してください。

最終的に見つかったコマンドもリマップされるかもしれません。Section 22.13 [Remapping Commands], page 441 を参照してください。

22.9 アクティブなキーマップの制御

global-map [Variable]
この変数はEmacsキーボード入力をコマンドにマップするデフォルトのグローバルキーマップを含む。通常はこのキーマップがグローバルキーマップである。デフォルトグローバルキーマップは`self-insert-command`をすべてのプリント文字にバインドするfullキーマップである。

これはグローバルキーマップ内のバインディングを変更する通常の手段だが、この変数に開始時のキーマップ以外の値を割り当てるべきではない。

current-global-map [Function]
この関数はカレントのグローバルキーマップをリターンする。デフォルトグローバルキーマップとカレントグローバルキーマップのいずれも変更していなければ`global-map`と同じ値。リターン値はコピーではなく参照である。これに`define-key`などの関数を使用すると、グローバルバインディングが変更されるだろう。

```
(current-global-map)
⇒ (keymap [set-mark-command beginning-of-line ...
           delete-backward-char])
```

current-local-map [Function]
この関数はカレントバッファーのローカルキーマップをリターンする。ローカルキーマップがなければ`nil`をリターンする。以下の例では、(Lisp Interactionモードを使用する)`*scratch*`バッファーにたいするキーマップは、ESC(ASCIIコード27)にたいするエントリーが別のsparseキーマップであるようなsparseキーマップである。

```
(current-local-map)
⇒ (keymap
    (10 . eval-print-last-sexp)
    (9 . lisp-indent-line)
    (127 . backward-delete-char-untabify)
    (27 keymap
        (24 . eval-defun)
        (17 . indent-sexp)))
```

`current-local-map`はローカルキーマップのコピーではなく参照をリターンします。これに`define-key`などの関数を使用するとローカルバインディングが変更されるでしょう。

current-minor-mode-maps [Function]
この関数はカレントで有効なメジャーモードのキーマップリストをリターンする。

use-global-map *keymap* [Function]
この関数は*keymap*を新たなカレントグローバルキーマップにする。これは`nil`をリターンする。

グローバルキーマップの変更は異例である。

use-local-map *keymap* [Function]
この関数は*keymap*をカレントバッファーの新たなローカルキーマップにする。*keymap*が`nil`なら、そのバッファーはローカルキーマップをもたない。`use-local-map`は`nil`をリターンする。ほとんどのメジャーモードコマンドはこの関数を使用する。

`minor-mode-map-alist` [Variable]

この変数はアクティブかどうかに関わらず、特定の変数の値にたいするキーマップを示す alist である。要素は以下のようになる:

```
(variable . keymap)
```

キーマップ *keymap* は *variable* が非 `nil` 値をもつときはアクティブである。*variable* は通常はメジャーモードを有効か無効にする変数である。Section 23.3.2 [Keymaps and Minor Modes], page 482 を参照のこと。

`minor-mode-map-alist` の要素が `minor-mode-alist` の要素と異なる構造をもつことに注意。マップは要素の CDR でなければならず、そうでなければ 2 つ目の要素にマップリストは用いられないだろう。CDR はキーマップ (リスト)、または関数定義がキーマップであるようなシンボルである。

1 つ以上のマイナーモードキーマップがアクティブなとき、`minor-mode-map-alist` 内で前のキーマップが優先される。しかし互いが干渉しないようにマイナーモードをデザインすること。これを正しく行えば順序は問題にならない。

マイナーモードについての詳細な情報は、Section 23.3.2 [Keymaps and Minor Modes], page 482 を参照のこと。`minor-mode-key-binding` (Section 22.11 [Functions for Key Lookup], page 436 を参照) も確認されたい。

`minor-mode-overriding-map-alist` [Variable]

この変数はメジャーモードによる特定のマイナーモードにたいするキーバインディングのオーバーライドを可能にする。この alist の要素は `minor-mode-map-alist` の要素のような (*variable* . *keymap*) という形式である。

ある変数が `minor-mode-overriding-map-alist` の要素として出現するなら、その要素によって指定されるマップは `minor-mode-map-alist` 内の同じ変数にたいして指定されるすべてのマップを完全に置き換える。

すべてのバッファーにおいて `minor-mode-overriding-map-alist` は自動的にバッファーローカルである。

`overriding-local-map` [Variable]

この変数が非 `nil` ならバッファーのローカルキーマップ、テキストプロパティまたは overlay によるキーマップ、マイナーモードキーマップのかわりに使用されるするキーマップを保持する。このキーマップが指定されると、カレントグローバルキーマップ以外のアクティブだった他のすべてのマップがオーバーライドされる。

`overriding-terminal-local-map` [Variable]

この変数が非 `nil` なら `overriding-local-map`、バッファーのローカルキーマップ、テキストプロパティまたは overlay によるキーマップ、およびすべてのマイナーモードキーマップのかわりに使用されるキーマップを保持する。

この変数はカレント端末にたいして常にローカルでありバッファーローカルにできない。Section 29.2 [Multiple Terminals], page 698 を参照のこと。これはインクリメンタル検索モードの実装に使用される。

`overriding-local-map-menu-flag` [Variable]

この変数が非 `nil` なら、`overriding-local-map` と `overriding-terminal-local-map` の値がメニューバーの表示に影響し得る。デフォルト値は `nil` なので、これらのマップ変数なメニューバーに影響をもたない。

これら 2 つのマップ変数は、たとえこれらの変数がメニューバー表示に影響し得るを与えない場合でも、メニューバーを使用してエンターされたキーシーケンスの実行には影響を与えることに注意。したがってもしメニューバーキーシーケンスが到着したら、そのキーシーケンスをルックアップして実行する前に変数をクリアーすること。この変数を使用するモードは通常は何らかの手段でこれを行っている。これらのモードは通常は “読み戻し (unread)” と exit によって処理されないイベントに応答する。

`special-event-map` [Variable]
この変数はスペシャルイベントにたいするキーマップを保持する。あるイベント型がこのキーマップ内でバインディングをもつなら、それはスペシャルイベントであり、そのイベントにたいするバインディングは `read-event`によって直接実行される。Section 21.9 [Special Events], page 412 を参照のこと。

`emulation-mode-map-alists` [Variable]
この変数はエミュレーションモードにたいして使用するキーマップの alist のリストを保持する。この変数は複数マイナーモードキーマップを使用するモードとパッケージを意図している。リストの各要素は `minor-mode-map-alist`と同じフォーマットと意味をもつキーマップの alist か、そのような alist 形式の変数バインディングをもつシンボルである。それぞれの alist 内のアクティブなキーマップは `minor-mode-map-alist`と `minor-mode-overriding-map-alist` の前に使用される。

`set-transient-map` *keymap* **&optional** *keep-pred on-exit* [Function]
この関数は一時的 *(transient)* なキーマップとして *keymap*を追加する。一時的なキーマップは 1 つ以上の後続するキーにたいして、他のキーマップより優先される。

*keymap*は通常は直後のキーをルックアップするために 1 回だけ使用される。しかし、オプション引数 *keep-pred*が `t`なら、そのマップはユーザーが *keymap*内で定義されたキーをタイプするまでアクティブのままとなる。*keymap*内にないキーをユーザーがタイプしたとき一時的キーマップは非アクティブとなり、そのキーにたいして通常のキールックアップが継続される。

*keep-pred*には関数も指定できる。この場合には *keymap*がアクティブの間は、各コマンドの実行に優先してその関数が引数なしで呼び出される。*keymap*がアクティブの間、関数は非 `nil` をリターンすること。

オプション引数 *on-exit*が非 `nil`なら、それは *keymap*が非アクティブになった後に引数なしで呼び出される関数を指定する。

この関数は他のすべてのアクティブなキーマップに優先される変数 `overriding-terminal-local-map`にたいして、*keymap*を追加または削除することによって機能する (Section 22.8 [Searching Keymaps], page 431 を参照)。

22.10 キーの照合

キールックアップ *(key lookup:* キー照合*)* とは与えられたキーマップからキーシーケンスのバインディングを見つけ出すことです。そのバインディングの使用や実行はキールックアップの一部ではありません。

キールックアップはキーシーケンス内の各イベントのイベント型だけを使用して、そのイベントの残りは無視します。実際のところキールックアップに使用されるキーシーケンスは、マウスイベントをイベント全体 (リスト) のかわりにイベント型のみ (シンボル) を用いるでしょう。Section 21.7 [Input Events], page 387 を参照してください。そのようなキーシーケンスは `command-execute` による実行には不十分ですが、キーのルックアップやリバインドには十分です。

キーシーケンスが複数イベントから構成されるとき、キールックアップはイベントを順に処理します。最初のイベントのバインディングが見つかったとき、それはキーマップでなければなりません。そのキーマップ内で 2 つ目のイベントを見つけ出して、そのキーシーケンス内のすべてのイベントが消費されるまで、このプロセスを続けます (故に最後のイベントにたいして見つかったイベントはキーマップかどうかはわからない)。したがってキールックアッププロセスはキーマップ内で単一イベントを見つけ出す、よりシンプルなプロセスで定義されます。これが行なわれる方法はキーマップ内でそのイベントに関連するオブジェクトの型に依存します。

キーマップ内のイベント型ルックアップによる値の発見を説明するために、キーマップエントリー *(keymap entry)* という用語を導入しましょう (これにはメニューアイテムにたいするキーマップ内のアイテム文字列や他の余計な要素は含まれない。なぜなら `lookup-key`や他のキーマップルックアップ関数がリターン値にそれらを含まないから)。任意の Lisp オブジェクトがキーマップエントリーとしてキーマップに格納されるかもしれませんが、すべてがキールックアップに意味をもつわけではありません。以下のテーブルはキーマップエントリーで重要な型です:

`nil` `nil`はそれまでにルックアップに使用されたイベントが未定義キーを形成することを意味する。最終的にキーマップがイベント型を調べるのに失敗してデフォルトバインディングも存在しないときは、そのイベント型のバインディングが `nil`であるのと同じである。

command それまでにルックアップに使用されたイベントがコンプリートキーを形成して、*command* がそのバインディングである。Section 13.1 [What Is a Function], page 209 を参照のこと。

array array(文字列かベクター) はキーボードマクロである。それまでにルックアップに使用されたイベントはコンプリートキーを形成して、array がそのバインディングである。詳細は Section 21.16 [Keyboard Macros], page 420 を参照のこと。

keymap それまでにルックアップに使用されたイベントはプレフィクスキーを形成する。そのキーシーケンスの次のイベントは *keymap*内でルックアップされる。

list list の意味はそのリストが何を含んでいるかに依存する:

- *list*の CAR がシンボル `keymap`なら、そのリストはキーマップでありキーマップとして扱われる (上記参照)。
- *list*の CAR が `lambda`なら、そのリストはラムダ式である。これは関数とみなされてそのように扱われる (上記参照)。キーバインディングとして正しく実行されるために、この関数はコマンドでなければならず `interactive`指定をもたなければならない。Section 21.2 [Defining Commands], page 376 を参照のこと。

symbol *symbol*の関数定義が *symbol*のかわりに使用される。もし関数定義もシンボルなら、任意の回数このプロセスが繰り返される。これは最終的にキーマップであるようなオブジェクト、コマンド、またはキーボードマクロに行き着くはずである。

キーマップとキーボードマクロ (文字列かベクター) は有効な関数ではないので関数定義にキーマップ、文字列、ベクターをもつシンボルは関数としては無効であることに注意。しかしキーバインディングとしては有効である。その定義がキーボードマクロなら、そのシンボルは `command-execute`(Section 21.3 [Interactive Call], page 382 を参照) の引数としても有効である。

シンボル `undefined`は特記するに値する。これはそのキーを未定義として扱うことを意味する。厳密に言うとそのキーは定義されているが、そのバインディングがコマンド `undefined`なのである。しかしこのコマンドは未定義キーにたいして自動的に行われるのと同じことを行う。これは (`ding`を呼び出して)bell を鳴らすがエラーはシグナルしない。

`undefined`はグローバルキーバインディングをオーバーライドして、そのキーをローカルに未定義とするために使用される。`nil`にローカルにバインドしてもグローバルバインディングをオーバーライドしないであろうから、これを行うのに失敗するだろう。

anything else
: オブジェクトの他の型が見つかったら、それまでにルックアップで使用されたイベントはコンプリートキーを形成してそのオブジェクトがバインディングになるが、そのバインディングはコマンドとして実行不可能である。

要約するとキーマップエントリーはキーマップ、コマンド、キーボードマクロ、あるいはそれらに導出されるシンボル、あるいは`nil`のいずれかです。

22.11 キー照合のための関数

以下はキールックアップに関連する関数および変数です。

`lookup-key` *keymap key* **&optional** *accept-defaults* [Function]

この関数は *keymap*内の *key*の定義をリターンする。このチャプターで説明されているキーをルックアップする他のすべての関数が`lookup-key`を使用する。以下は例:

```
(lookup-key (current-global-map) "\C-x\C-f")
    ⇒ find-file
(lookup-key (current-global-map) (kbd "C-x C-f"))
    ⇒ find-file
(lookup-key (current-global-map) "\C-x\C-f12345")
    ⇒ 2
```

文字列かベクターの *key*が *keymap*内で指定されるプレフィクスキーとして有効なキーシーケンスでなければ、それは最後に余計なイベントをもった、単一のキーシーケンスに適合しない長過ぎるキーのはずである。その場合のリターン値は数となり、この数はコンプリートキーを構成する *key*の前にあるイベントの数である。

*accept-defaults*が非`nil`なら、`lookup-key`は *key*内の特定のイベントにたいするバインディングと同様にデフォルトバインディングも考慮する。それ以外では`lookup-key`は特定の *key*のシーケンスにたいするバインディングだけを報告して、明示的に指定したとき以外はデフォルトバインディングを無視する (これを行うには *key*の要素として`t`を与える。Section 22.3 [Format of Keymaps], page 423 を参照されたい)。

*key*がメタ文字 (ファンクションキーではない) を含むなら、その文字は暗黙に`meta-prefix-char`の値と対応する非メタ文字からなる 2 文字シーケンスに置き換えられる。したがって以下の 1 つ目の例は 2 つ目の例に変換されて処理される。

```
(lookup-key (current-global-map) "\M-f")
    ⇒ forward-word
(lookup-key (current-global-map) "\ef")
    ⇒ forward-word
```

*keymap*引数はキーマップのリストでもよい。

`read-key-sequence`とは異なり、この関数は指定されたイベントの情報を破棄する変更 (Section 21.8.1 [Key Sequence Input], page 404 を参照) を行わない。特にこの関数はアルファベット文字を小文字に変更せず、ドラッグイベントをクリックイベントに変更しない。

`undefined` [Command]
キーを未定義にするためにキーマップ内で使用される。これは`ding`を呼び出すがエラーを発生ささない。

`local-key-binding` *key* **&optional** *accept-defaults* [Function]
この関数はカレントのローカルキーマップ内の *key*にたいするバインディングをリターンする。カレントのローカルキーマップ内で未定義なら`nil`をリターンする。

引数 *accept-defaults*は`lookup-key`(上記) と同じようにデフォルトバインディングのチェックを制御する。

`global-key-binding` *key* **&optional** *accept-defaults* [Function]
この関数はカレントのグローバルキーマップ内でコマンド *key*にたいするバインディングをリターンする。カレントのグローバルキーマップ内で未定義なら`nil`をリターンする。

引数 *accept-defaults*は`lookup-key`(上記) と同じようにデフォルトバインディングのチェックを制御する。

`minor-mode-key-binding` *key* **&optional** *accept-defaults* [Function]
この関数はアクティブなマイナーモードの *key*のバインディングをリストでリターンする。より正確にはこの関数は`(`*`modename . binding`*`)`のようなペアの alist をリターンする。ここで *modename*なそのマイナーモードを有効にする変数、*binding*はそのモードでの *key*のバインディングである。*key*がマイナーモードバインディングをもたなければ値は`nil`。

最初に見つかったバインディングがプレフィクス定義 (キーマップ、またはキーマップとして定義されたシンボル) でなければ、他のマイナーモードに由来するすべての後続するバインディングは完全に shadow されて省略される。同様にこのリストはプレフィクスバインディングに後続する非プレフィクスバインディングは省略される。

引数 *accept-defaults*は`lookup-key`(上記) と同じようにデフォルトバインディングのチェックを制御する。

`meta-prefix-char` [User Option]
この変数はメタ/プレフィクス文字コードである。これはメタ文字をキーマップ内でルックアップできるように 2 文字シーケンスに変換する。有用な結果を得るために値はプレフィクスイベント (Section 22.6 [Prefix Keys], page 428 を参照) であること。デフォルト値は 27 で、これは`ESC`にたいする ASCIIコード。

`meta-prefix-char`の値が 27 であるような限り、キールックアップは通常は`backward-word`コマンドとして定義される *`M-b`*を *`ESC b`*に変換する。しかし`meta-prefix-char`を 24(*`C-x`*のコード) にセットすると、Emacs は *`M-b`*を *`C-x b`*に変換するだろうが、これの標準のバインディングは`switch-to-buffer`コマンドである。以下に何が起こるかを示す (実際にこれを行ってはならない!):

```
meta-prefix-char                    ; デフォルト値
     ⇒ 27
(key-binding "\M-b")
     ⇒ backward-word
?\C-x                               ; 文字. の
     ⇒ 24                           ; プリント表現
(setq meta-prefix-char 24)
     ⇒ 24
```

```
(key-binding "\M-b")
     ⇒ switch-to-buffer            ; 今や M-bをタイプすると
                                   ;   C-x bをタイプしたようになる

(setq meta-prefix-char 27)         ; 混乱を避けよう!
     ⇒ 27                          ; デフォルト値をリストア!
```

この単一イベントから 2 イベントへの変換は文字にたいしてのみ発生し、他の種類の入力イベントには発生しない。したがってファンクションキー *M-F1*は *ESC F1*に変換されない。

22.12 キーバインディングの変更

キーのリバインド (rebind: 再バインド、再束縛) は、キーマップ内でそのキーのバインディングエントリーを変更することによって行われます。グローバルキーマップ内のバインディングを変更すると、その変更は (たとえローカルバインディングによりグローバルバインディングを shadow しているバッファーでは直接影響しないとしても) すべてのバッファーに影響します。カレントバッファーのローカルマップを変更すると、通常は同じメジャーモードを使用するすべてのバッファーに影響します。関数 `global-set-key`と `local-set-key`は、これらの操作のための使いやすいインターフェイスです (Section 22.15 [Key Binding Commands], page 444 を参照)。より汎用的な関数 `define-key`を使用することもできます。その場合には変更するマップを明示的に指定しなければなりません。

Lisp プログラムでリバインドするキーシーケンスを選択するときは、さまざまなキーの使用についての Emacs の慣習にしたがってください (Section D.2 [Key Binding Conventions], page 1164 を参照)。

リバインドするキーシーケンスの記述では、コントロール文字とメタ文字にたいして特別なエスケープシーケンスを使用すると良いでしょう (Section 2.4.8 [String Type], page 19 を参照)。構文 '`\C-`' は後続する文字がコントロール文字でること、'`\M-`'は後続する文字がメタ文字であることを意味します。したがって文字列`"\M-x"`は 1 つの *M-x*、`"\C-f"`は 1 つの *C-f*、`"\M-\C-x"`と`"\C-\M-x"`は 1 つの *C-M-x*として読み取られます。ベクター内でもこのエスケープシーケンス、および文字列では使用できない他のエスケープシーケンスを使用できます。一例は '`[?\C-\H-x home]`'です。Section 2.4.3 [Character Type], page 11 を参照してください。

キー定義とルックアップ関数は、ベクターであるようなキーシーケンス内のイベント型にたいして別の構文を受け入れます。修飾名に基本イベント (文字かファンクションキー名) を付加したものを含んだリストを使用できます。たとえば`(control ?a)`は`?\C-a`、`(hyper control left)`は`C-H-left`と等価です。このようなリストの利点の 1 つは、コンパイル済みファイル内に修飾ビットの正確な数値コードが出現しないことです。

以下の関数は *keymap*がキーマップでない場合、および *key*がキーシーケンスを表す文字列やベクターでない場合にはエラーをシグナルします。リストであるようなイベントにたいする略記として、イベント型 (シンボル) を使用できます。`kbd`関数 (Section 22.1 [Key Sequences], page 422 を参照) はキーシーケンスを指定するための便利な方法です。

`define-key` *keymap key binding* [Function]

この関数は *keymap*内で *key*にたいするバインディングをセットする (*key*が長さ 2 以上のイベントなら、その変更は実際は *keymap*から辿られる他のキーマップで行なわれる)。引数 *binding* には任意の Lisp オブジェクトを指定できるが、意味があるのは特定のオブジェクトだけである (意味のある型のリストは Section 22.10 [Key Lookup], page 434 を参照)。`define-key`のリターン値は *binding*である。

*key*が `[t]`なら、それは *keymap*内でデフォルトバインディングをセットする。イベントが自身のバインディングをもたないとき、そのキーマップ内にデフォルトバインディングが存在すれば Emacs コマンドループはそれを使用する。

*key*のすべてのプレフィクスは、プレフィクスキー (キーマップにバインドされる) か、あるいは未定義でなけらばならず、それ以外ならエラーがシグナルされる。*key*のいくつかのプレフィクスが未定義なら、`define-key`はそれをプレフィクスキーとして定義するので、残りの *key* は指定されたように定義できる。

前に *keymap*内で *key*にたいするバインディングが存在しなければ、新たなバインディングが *keymap*の先頭に追加される。キーマップ内のバインディングの順序はキーボード入力にたいし影響を与えないが、メニューキーマップにたいしては問題となる (Section 22.17 [Menu Keymaps], page 447 を参照)。

以下は sparse キーマップを作成してその中にバインディングをいくつか作成する例:

```
(setq map (make-sparse-keymap))
    ⇒ (keymap)
(define-key map "\C-f" 'forward-char)
    ⇒ forward-char
map
    ⇒ (keymap (6 . forward-char))

;; C-xにたいし sparse サブマップを作成して
;; その中で fをバインドする
(define-key map (kbd "C-x f") 'forward-word)
    ⇒ forward-word
map
⇒ (keymap
    (24 keymap                ; C-x
        (102 . forward-word)) ;      f
    (6 . forward-char))       ; C-f

;; C-pを ctl-x-mapにバインド
(define-key map (kbd "C-p") ctl-x-map)
;; ctl-x-map
⇒ [nil ... find-file ... backward-kill-sentence]

;; ctl-x-map内で C-fを fooにバインド
(define-key map (kbd "C-p C-f") 'foo)
⇒ 'foo
map
⇒ (keymap     ; ctl-x-map内の fooに注目
    (16 keymap [nil ... foo ... backward-kill-sentence])
    (24 keymap
        (102 . forward-word))
    (6 . forward-char))
```

*C-p C-f*にたいする新たなバインディングの格納は、実際には `ctl-x-map`内のエントリーを変更することによって機能し、これはデフォルトグローバルマップ内の *C-p C-f*と *C-x C-f*の両方のバインディングを変更する効果をもつことに注意。

関数`substitute-key-definition`はキーマップから特定のバインディングをもつキーをスキャンして、それらを異なるバインディングにリバインドする。より明快かつ多くの場合には同じ結果を生成できる他の機能として、あるコマンドから別のコマンドへのリマップがある (Section 22.13 [Remapping Commands], page 441 を参照)。

`substitute-key-definition` *olddef newdef keymap* **&optional** *oldmap* [Function]

この関数は *keymap*内で *olddef*にバインドされるすべてのキーについて *olddef*を *newdef*に置き換える。言い換えると *olddef*が出現する箇所のすべてを *newdef*に置き換える。この関数は `nil`をリターンする。

たとえば以下を Emacs の標準バインディングで行うと `C-x C-f`を再定義する:

```
(substitute-key-definition
 'find-file 'find-file-read-only (current-global-map))
```

*oldmap*が非 `nil`なら、どのキーをリバインドするかを *oldmap*内のバインディングが決定するように `substitute-key-definition`の動作を変更する。リバインディングは依然として *oldmap*ではなく *keymap*で発生する。したがって他のマップ内のバインディングの制御下でマップを変更することができる。たとえば、

```
(substitute-key-definition
  'delete-backward-char 'my-funny-delete
  my-map global-map)
```

これは標準的な削除コマンドにグローバルにバインドされたキーにたいして `my-map`内の特別な削除コマンドを設定する。

以下はキーマップの置き換え (substitution) の前後を示した例:

```
(setq map (list 'keymap
                (cons ?1 olddef-1)
                (cons ?2 olddef-2)
                (cons ?3 olddef-1)))
⇒ (keymap (49 . olddef-1) (50 . olddef-2) (51 . olddef-1))

(substitute-key-definition 'olddef-1 'newdef map)
⇒ nil
map
⇒ (keymap (49 . newdef) (50 . olddef-2) (51 . newdef))
```

`suppress-keymap` *keymap* **&optional** *nodigits* [Function]

この関数は `self-insert-command`をコマンド `undefined`にリマップ (Section 22.13 [Remapping Commands], page 441 を参照) することによって full キーマップのコンテンツを変更する。これはすべてのプリント文字を未定義にする効果をもつので、通常のテキスト挿入は不可能になる。`suppress-keymap`は `nil`をリターンする。

*nodigits*が `nil`なら、`suppress-keymap`は数字が `digit-argument`、`-`が `negative-argument`を実行するように定義する。それ以外は残りのプリント文字と同じように、それらの文字も未定義にする。

`suppress-keymap`関数は `yank`や `quoted-insert`のようなコマンドを抑制 (suppress) しないのでバッファーの変更は可能。バッファーの変更を防ぐには、バッファーを読み取り専用 (read-only) にすること (Section 27.7 [Read Only Buffers], page 599 を参照)。

この関数は *keymap*を変更するので、通常は新たに作成したキーマップにたいして使用するだろう。他の目的のために使用されている既存のキーマップに操作を行うと恐らくトラブルの原因となる。たとえば `global-map`の抑制は Emacs をほとんど使用不可能にするだろう。

この関数はテキストの挿入が望ましくないメジャーモードの、ローカルキーマップ初期化に使用され得る。しかしそのようなモードは通常は `special-mode` (Section 23.2.5 [Basic Major Modes], page 472 を参照) から継承される。この場合にはそのモードのキーマップは既に抑制

済みの special-mode-mapから自動的に受け継がれる。以下に special-mode-mapが定義される方法を示す:

```
(defvar special-mode-map
  (let ((map (make-sparse-keymap)))
    (suppress-keymap map)
    (define-key map "q" 'quit-window)
    ...
    map))
```

22.13 コマンドのリマップ

あるコマンドから他のコマンドへのリマップ *(remap)* には、特別な種類のキーバインディングが使用できます。この機能を使用するためには、ダミーイベント remapで始まり、その後にリマップしたいコマンド名が続くようなキーシーケンスにたいするキーバインディングを作成します。そしてそのバインディングにたいしては、新たな定義 (通常はコマンド名だがキーバインディングにたいして有効な他の任意の定義を指定可能) を指定します。

たとえばMy モードというモードが、kill-lineのかわりに呼び出される my-kill-lineという特別なコマンドを提供するとします。これを設定するには、このモードのキーマップに以下のようなリマッピングが含まれるはずです:

```
(define-key my-mode-map [remap kill-line] 'my-kill-line)
```

その後は my-mode-mapがアクティブなときは常に、ユーザーが *C-k* (kill-lineにたいするデフォルトのグローバルキーシーケンス) をタイプするとEmacs はかわりに my-kill-lineを実行するでしょう。

リマップはアクティブなキーマップでのみ行なわれることに注意してください。たとえば ctl-x-mapのようなプレフィクスキーマップ内にリマッピングを置いても、そのようなキーマップはそれ自体がアクティブでないので通常は効果がありません。それに加えてリマップは 1 レベルを通じてのみ機能します。以下の例では、

```
(define-key my-mode-map [remap kill-line] 'my-kill-line)
(define-key my-mode-map [remap my-kill-line] 'my-other-kill-line)
```

これは kill-lineを my-other-kill-lineにリマップしません。かわりに通常のキーバインディングが kill-lineを指定する場合には、それが my-kill-lineにリマップされます。通常のバインディングが my-kill-lineを指定すると、my-other-kill-lineにリマップされます。

コマンドのリマップをアンドゥするには、以下のようにそれを nilにリマップします :

```
(define-key my-mode-map [remap kill-line] nil)
```

`command-remapping` *command* **&optional** *position keymaps* [Function]

この関数はカレントアクティブキーマップによって与えられる *command*(シンボル) にたいするリマッピングをリターンする。*command*がリマップされていない (これは普通の状況である)、あるいはシンボル以外なら、この関数は nilをリターンする。positionは key-bindingの場合と同様、使用するキーマップを決定するためにバッファー位置かイベント位置をオプションで指定できる。

オプション引数 keymapsが非 nilなら、それは検索するキーマップのリストを指定する。この引数は positionが非 nilなら無視される。

22.14 イベントシーケンス変換のためのキーマップ

read-key-sequence関数がキーシーケンス (Section 21.8.1 [Key Sequence Input], page 404 を参照) を読み取るときには、特定のイベントシーケンスを他のものに変換 (translate) するために変換キーマップ *(translation keymaps)* を使用します。input-decode-map、local-function-key-map、key-translation-map(優先順) は変換キーマップです。

変換キーマップは他のキーマップと同じ構造をもちますが使い方は異なります。変換キーマップはキーシーケンスを読み取るときに、コンプリートキーシーケンスにたいするバインディングではなくキーシーケンスに行う変換を指定します。キーシーケンスが読み取られると、それらのキーシーケンスは変換キーマップにたいしてチェックされます。ある変換キーマップが kをベクター vにバインドするなら、キーシーケンス内のどこかにサブシーケンスとして kが出現すると、それは v内のイベントに置き換えられます。

たとえばキーパッドキー PF1が押下されたとき、VT100 端末は *ESC O P*を送信します。そのような端末では Emacs はそのイベントシーケンスを単一イベント pf1に変換しなければなりません。これは input-decode-map内で *ESC O P*を [pf1]にバインドすることにより行われます。したがってその端末上で *C-c PF1*をタイプしたとき、端末は文字シーケンス *C-c ESC O P*を発行して、read-key-sequenceがそれを *C-c PF1*に変換、ベクター [?\C-c pf1]としてリターンします。

変換キーマップは、(keyboard-coding-systemで指定された入力コーディングシステムを通じて)Emacs がキーボード入力をデコードした直後だけ効果をもちます。Section 33.10.8 [Terminal I/O Encoding], page 877 を参照してください。

input-decode-map [Variable]

この変数は通常の文字端末上のファンクションキーから送信された文字シーケンスを記述するキーマップを保持する。

input-decode-mapの値は、通常はその端末の Terminfo か Termcap のエントリーに応じて自動的にセットアップされるが、Lisp の端末仕様ファイルの助けが必要なときもある。Emacs には一般的な多くの端末の端末仕様ファイルが同梱されている。これらのファイルの主な目的は Termcap や Terminfo から推定できないエントリーを input-decode-map内に作成することである。Section 40.1.3 [Terminal-Specific], page 1095 を参照のこと。

local-function-key-map [Variable]

この変数は input-decode-mapと同じようにキーマップを保持するが、通常は優先される解釈選択肢 (alternative interpretation) に変換されるべきキーシーケンスを記述するキーマップを保持する。このキーマップは input-decode-mapの後、key-translation-mapの前に適用される。

local-function-key-map内のエントリーはマイナーモード、ローカルキーマップ、グローバルキーマップによるバインディングと衝突する場合には無視される。つまり元のキーシーケンスが他にバインディングをもたない場合だけリマッピングが適用される。

local-function-key-mapは function-key-mapを継承する。後者はすべての端末にバインディングを適用したい場合のみ修正するべきなので、ほとんど常に前者の使用が望ましい。

key-translation-map [Variable]

この変数は入力イベントを他のイベントに変換するために、input-decode-mapと同じように使用される別のキーマップを保持する。input-decode-mapとの違いは、local-function-key-mapの前ではなく後に機能する点である。このキーマップは local-function-key-mapによる変換結果を受け取る。

input-decode-mapと同様だが local-function-key-mapとは異なり、このキーマップは入力キーシーケンスが通常のバインディングをもつかどうかかに関わらず適用される。しかしこのキーマップによりキーバインディングがオーバーライドされても、key-translation-mapでは実際のキーバインディングが効果をもち得ることに注意。確かに実際のキーバインディングは local-function-key-mapをオーバーライドし、したがって key-translation-mapが受け取るキーシーケンスは変更されるだろう。明確にするためにはこのような類の状況は避けたほうがよい。

key-translation-mapは通常は self-insert-commandにバインディングされるような通常文字を含めて、ユーザーがある文字を他の文字にマップすることを意図している。

キーシーケンスのかわりにキーの変換として関数を使用することにより、シンプルなエイリアスより多くのことに input-decode-map、local-function-key-map、key-translation-mapを使用できます。その場合にはこの関数はそのキーの変換を計算するために呼び出されます。

キー変換関数は引数を 1 つ受け取ります。この引数は read-key-sequence内で指定されるプロンプトです。キーシーケンスがエディターコマンドループに読み取られる場合は nilです。ほとんどの場合にはプロンプト値は無視できます。

関数が自身で入力を読み取る場合、その関数は後続のイベントを変更する効果をもつことができます。たとえば以下は *C-c h*をハイパー文字に後続する文字とするために定義する方法の例です:

```
(defun hyperify (prompt)
  (let ((e (read-event)))
    (vector (if (numberp e)
                (logior (ash 1 24) e)
              (if (memq 'hyper (event-modifiers e))
                  e
                (add-event-modifier "H-" e))))))

(defun add-event-modifier (string e)
  (let ((symbol (if (symbolp e) e (car e))))
    (setq symbol (intern (concat string
                                 (symbol-name symbol))))
    (if (symbolp e)
        symbol
      (cons symbol (cdr e)))))

(define-key local-function-key-map "\C-ch" 'hyperify)
```

22.14.1 通常のキーマップとの対話

そのキーシーケンスがコマンドにバインドされたとき、またはさらにイベントを追加してもコマンドにバインドされるシーケンスにすることができないと Emacs が判断したときにキーシーケンスの終わりが検出されます。

これは元のキーシーケンスがバインディングをもつかどうかに関わらず、input-decode-mapや key-translation-mapを適用するときに、そのようなバインディングが変換の開始を妨げることを意味します。たとえば前述の VT100 の例に戻って、グローバルマップに *C-c ESC*を追加してみましょう。するとユーザーが *C-c PF1*をタイプしたとき、Emacs は *C-c ESC O P*を *C-c PF1*に変換するのに失敗するでしょう。これは Emacs が *C-x ESC*の直後に読み取りを停止して、*O P*が読み取られずに

残るからです。この場合にはユーザーが実際に *C-c ESC*をタイプすると、ユーザーが実際に ESCを押下したのか、あるいは PF1を押下したのか判断するために Emacs が待つべきではないのです。

この理由によりキーシーケンスの終わりがキー変換のプレフィクスであるようなキーシーケンスをコマンドにバインドするのは、避けたほうがよいでしょう。そのような問題を起こす主なサフィックス、およびプレフィクスは ESC、*M-O* (実際は *ESC O*)、*M-[* (実際は *ESC [*) です。

22.15 キーのバインドのためのコマンド

このセクションではキーバインディングを変更するための便利な対話的インターフェイスを説明します。これらは define-keyを呼び出すことにより機能します。

ユーザーは init ファイルにたいしてシンプルなカスタマイズを行うとき、しばしば global-set-keyを使用します。たとえば、

```
(global-set-key (kbd "C-x C-\\") 'next-line)
```

または

```
(global-set-key [?\C-x ?\C-\\] 'next-line)
```

または

```
(global-set-key [(control ?x) (control ?\\)] 'next-line)
```

は、次の行に移動するように *C-x C-*を再定義します。

```
(global-set-key [M-mouse-1] 'mouse-set-point)
```

は、メタキーを押してマウスの第一ボタン (左ボタン) をクリックすると、クリックした箇所にポイントをセットするように再定義します。

バインドするキーの Lisp 指定に非 ASCII文字のテキストを使用するときには注意してください。マルチバイトとして読み取られたテキストがあるなら、Lisp ファイル内でマルチバイトテキストが読み取られるときのように (Section 16.4 [Loading Non-ASCII], page 274 を参照)、マルチバイトとしてキーをタイプしなければなりません。たとえば、

```
(global-set-key "^^c3^^b6" 'my-function) ; bind o-umlaut
```

または

```
(global-set-key ?^^c3^^b6 'my-function) ; bind o-umlaut
```

を Latin-1 のマルチバイト環境で使用すると、これらのコマンドは Latin-1 端末から送信されたバイトコード 246(*M-v*) ではなく、コード 246 のマルチバイト文字に実際にはバインドされます。このバインディングを使用するためには適切な入力メソッド (Section “Input Methods” in *The GNU Emacs Manual* を参照) を使用して、キーボードをデコードする方法を Emacs に教える必要があります。

`global-set-key` *key binding* [Command]

この関数はカレントグローバルマップ内で *key*のバインディングを *binding*にセットする。

```
(global-set-key key binding)
≡
(define-key (current-global-map) key binding)
```

`global-unset-key` *key* [Command]

この関数はカレントグローバルマップから *key*のバインディングを削除する。

プレフィクスとして *key*を使用する長いキーの定義の準備に使用するのもこの関数の 1 つの用途である。*key*が非プレフィクスのようなバインディングをもつならこの使い方は許容されないだろう。たとえば、

```
(global-unset-key "\C-l")
    ⇒ nil
```

```
(global-set-key "\C-l\C-l" 'redraw-display)
    ⇒ nil
```

この関数は以下のように define-keyを使用するのと等しい:

```
(global-unset-key key)
≡
(define-key (current-global-map) key nil)
```

local-set-key *key binding* [Command]

この関数はカレントローカルキーマップ内の *key*のバインディングを *binding*にセットする。

```
(local-set-key key binding)
≡
(define-key (current-local-map) key binding)
```

local-unset-key *key* [Command]

この関数はカレントローカルキーマップから *key*のバインディングを削除する。

```
(local-unset-key key)
≡
(define-key (current-local-map) key nil)
```

22.16 キーマップのスキャン

このセクションではすべてのカレントキーマップをスキャンして、ヘルプ情報をプリントするために使用される関数を説明します。

accessible-keymaps *keymap* **&optional** *prefix* [Function]

この関数は、(0 個以上のプレフィクスキーを通じて)*keymap*から到達可能なすべてのキーマップのリストをリターンする。リターン値は (*key* . *map*)のような形式の要素をもつ連想配列 (alist) である。ここで *key*は *keymap*内での定義が *map*であるようなプレフィクスキーである。

alist の要素は *key*の長さにたいして昇順にソートされている。1 つ目の要素は常に (`[]` . *keymap*)。これは指定されたキーマップがイベントなしのプレフィクスによって、自分自身からアクセス可能だからである。

*prefix*が与えられたら、それはプレフィクスキーシーケンスである。その場合には *prefix*で始まるプレフィクスキーをもつサブマップだけが accessible-keymapsに含まれる。これらの要素の意味は (accessible-keymaps)の値の場合と同様であり、いくつかの要素が省略されている点だけが異なる。

以下の例ではリターンされる alist により '^['と表示されるキー ESCがプレフィクスキーであり、その定義が sparse キーマップ (keymap (83 . center-paragraph) (115 . foo))であることが示される。

```
(accessible-keymaps (current-local-map))
⇒(([] keymap
      (27 keymap   ; 以降 ESCにたいするこのキーマップが繰り返されることに注意
          (83 . center-paragraph)
          (115 . center-line))
      (9 . tab-to-tab-stop))

   ("^[" keymap
    (83 . center-paragraph)
    (115 . foo)))
```

また以下の例では *C-h*は (keymap (118 . describe-variable)...)で始まる sparse キーマップを使用するプレフィクスキーである。他のプレフィクス *C-x 4*は変数 ctl-x-4-mapの

値でもあるキーマップを使用する。イベント `mode-line`はウィンドウの特別な箇所でのマウスイベントにたいするプレフィクスとして使用される、いくつかのダミーイベントのうちの 1 つである。

```
(accessible-keymaps (current-global-map))
⇒ (([] keymap [set-mark-command beginning-of-line ...
                  delete-backward-char])
   ("^H" keymap (118 . describe-variable) ...
    (8 . help-for-help))
   ("^X" keymap [x-flush-mouse-queue ...
    backward-kill-sentence])
   ("^[" keymap [mark-sexp backward-sexp ...
    backward-kill-word])
   ("^X4" keymap (15 . display-buffer) ...)
   ([mode-line] keymap
    (S-mouse-2 . mouse-split-window-horizontally) ...))
```

これらが実際に目にするであろうキーマップのすべてではない。

`map-keymap` *function keymap* [Function]

関数 `map-keymap`は *keymap*内のバインディングそれぞれにたいして 1 回 *function*を呼び出す。呼び出す際の引数はイベント型と、そのバインディングの値の 2 つ。*keymap*に親キーマップがあれば、その親キーマップのバインディングも含まれる。これは再帰的に機能する。つまりその親キーマップ自身が親キーマップをもてば、それのバインディングも含まれる、といった具合である。

これはキーマップ内のすべてのバインディングを検証するもっとも明快な方法である。

`where-is-internal` *command* **&optional** *keymap firstonly noindirect no-remap* [Function]

この関数は `where-is`コマンド (Section “Help” in *The GNU Emacs Manual* を参照) により使用されるサブルーチンである。これはキーマップのセット内で *command*にバインドされる、(任意の長さの) キーシーケンスすべてのリストをリターンする。

引数 *command*には任意のオブジェクトを指定できる。このオブジェクトはすべてのキーマップエントリーにたいして、`eq`を使用して比較される。

*keymap*が `nil`なら、`overriding-local-map`の値とは無関係に (`overriding-local-map`の値が `nil`であると装って)、カレントアクティブキーマップをマップとして使用する。*keymap*がキーマップなら *keymap*とグローバルキーマップが検索されるマップとなる。*keymap*がキーマップのリストなら、それらのキーマップだけが検索される。

*keymap*にたいする式としては、通常は `overriding-local-map`を使用するのが最善である。その場合には `where-is-internal`は正にアクティブなキーマップを検索する。グローバルマップだけを検索するには *keymap*の値に `(keymap)`(空のキーマップ) を渡せばよい。

*firstonly*が `non-ascii`なら、値はすべての可能なキーシーケンスのリストではなく最初に見つかったキーシーケンスを表す単一のベクターとなる。*firstonly*が `t`なら、値は最初のキーシーケンスだが全体が ASCII文字 (またはメタ修飾された ASCII文字) で構成されるキーシーケンスが他のすべてのキーシーケンスに優先されて、リターン値がメニューバインディングになることは決してない。

*noindirect*が非 `nil`なら `where-is-internal`は自身のコマンドを探すためにメニューアイテムの内部を調べない。これによりメニューアイテム自体の検索が可能になる。

5 つ目の引数 *no-remap*はこの関数がコマンドリマッピング (Section 22.13 [Remapping Commands], page 441 を参照) を扱う方法を決定する。興味深いケースが 2 つある:

コマンド *other-command*が *command*にリマップされる場合:
*no-remap*が `nil`なら *other-command*にたいするバインディングを探して、*command*にたいするバインディングであるかのようにそれらを扱う。*no-remap*が非 `nil`ならそれらのバインディングを探すかわりに、利用可能なキーシーケンスリストにベクター `[remap other-command]` を含める。

*command*が *other-command*にリマップされる場合:
*no-remap*が `nil`なら、*command*ではなく *other-command*にたいするバインディングをリターンする。*no-remap*が非 `nil`なら、リマップされていることを無視して *command*にたいするバインディングをリターンする。

`describe-bindings` **&optional** *prefix buffer-or-name* [Command]
この関数はすべてのカレントキーバインディングのリストを作成して、`*Help*`という名前のバッファーにそれを表示する。テキストはモードごとにグループ化されて順番はマイナーモード、メジャーモード、グローバルバインディングの順である。

*prefix*が非 `nil`なら、それはプレフィクスキーである。その場合にはリストに含まれるのは *prefix*で始まるキーだけになる。

複数の連続する ASCIIコードが同じ定義をもつとき、それらは '*firstchar..lastchar*'のようにまとめて表示される。この場合にはそれがどの文字に該当するかを理解するには、そのASCIIコードを知っている必要がある。たとえばデフォルトグローバルマップでは文字 'SPC .. ~'は 1 行で記述される。SPCは ASCIIの 32, ~は ASCIIの 126 で、その間のすべての文字には通常のプリント文字 (アルファベット文字や数字、区切り文字等) が含まれる。これらの文字はすべて `self-insert-command`にバインドされる。

*buffer-or-name*が非 `nil`のならそれはバッファーかバッファー名である。その場合は `describe-bindings`はカレントバッファーのかわりに、そのバッファーのバインディングをリストする。

22.17 メニューキーアップ

キーマップはキーボードキーやマウスボタンにたいするバインディング定義と同様に、メニューとして操作することができます。メニューは通常はマウスにより操作されますが、キーボードでも機能させことができます。次の入力イベントにたいしてメニューキーマップがアクティブならキーボードメニュー機能がアクティブになります。

22.17.1 メニューの定義

キーマップが *overall* プロンプト文字列 *(overall prompt string)* をもつ場合には、そのキーマップはメニューとして動作します。overall プロンプト文字列はキーマップの要素として表される文字列です (Section 22.3 [Format of Keymaps], page 423 を参照)。この文字列にはメニューコマンドの目的を記述します。(もしあれば)Emacs はメニュー表示に使用されるツールキットに応じて、メニュータイトルに overall メニュー文字列を表示します [1]。キーボードメニューも overall プロンプト文字列を表示します。

プロンプト文字列をもつキーマップを構築するもっとも簡単な方法は `make-keymap`、`make-sparse-keymap` (Section 22.4 [Creating Keymaps], page 425 を参照)、`define-prefix-command` ([Definition of define-prefix-command], page 429 を参照) を呼び出すときに引数として文字列を指定する方法です。キーマップをメニューとして操作したくなければ、これらの関数にたいしてプロンプト文字列を指定しないでください。

[1] これはテキスト端末のようなツールキットを使用しないメニューにたいして要求されます。

`keymap-prompt` *keymap* [Function]
この関数は *keymap*の overall プロンプト文字列、もしなければ `nil`をリターンする。

メニューのアイテムは、そのキーマップ内のバインディングです。各バインディングはイベント型と定義を関連付けますが、イベント型はメニューの外見には何の意味ももっていません (通常はイベント型としてキーボードが生成できない擬似イベントのシンボルをメニューアイテムのバインディングに使用する)。メニュー全体はこれらのイベントにたいするキーマップ内のバインディングから生成されます。

メニュー内のアイテムの順序はキーマップ内のバインディングの順序と同じです。`define-key`は新たなバインディングを先頭に配置するので、メニューアイテムの順序が重要ならメニューの最後から先頭へメニューアイテムを定義する必要があります。既存のメニューにアイテムを追加するときには、`define-key-after`を使用してメニュー内の位置を指定できます (Section 22.17.7 [Modifying Menus], page 458 を参照)。

22.17.1.1 単純なメニューアイテム

メニューアイテムを定義するシンプル (かつ初歩的) な方法は、何らかのイベント型 (何のイベント型かは問題ではない) を以下のようにバインドすることです:

```
(item-string . real-binding)
```

CAR の *item-string*はメニュー内で表示される文字列です。これは短いほうが望ましく、1 個から 3 個の単語が望ましいでしょう。この文字列は対応するコマンドの動作を記述します。すべてのグラフィカルツールキットが非 ASCIIテキストを表示できる訳ではないことに注意してください (キーボードメニューと GTK+ツールキットの大部分では機能するだろう)。

以下のようにヘルプ文字列と呼ばれる 2 つ目の文字列を与えることもできます:

```
(item-string help . real-binding)
```

*help*はマウスがそのアイテム上にあるときに、`help-echo`テキストプロパティ([Help display], page 826 を参照) と同じ方法で表示される help-echo 文字列を指定します。

`define-key`に関する限り、*item-string*と *help-string*はそのイベントのバインディングの一部です。しかし `lookup-key`は単に *real-binding*だけをリターンし、そのキーの実行には *real-binding*だけが使用されます。

*real-binding*が `nil`なら *item-string*はメニューに表示されますが選択できません。

*real-binding*がシンボルで `menu-enable`プロパティが非 `nil`なら、そのプロパティはメニューアイテムが有効か無効かを制御する式です。メニュー表示にキーマップが使用されるたびに Emacs はその式を評価して、式の値が非 `nil`の場合のみそのメニューのメニューアイテムを有効にします。メニューアイテム無効なときには、そのアイテムは fuzzy 形式で表示されて選択できなくなります。

メニューバーはメニューを調べる際にどのアイテムが有効かを再計算しません。これは X ツールキットが事前にメニュー全体を要求するからです。メニューバーの再計算を強制するには `force-mode-line-update`を呼び出してください (Section 23.4 [Mode Line Format], page 485 を参照)。

22.17.1.2 拡張メニューアイテム

メニューアイテムの拡張フォーマットは、単純なフォーマットに比べてより柔軟かつ明快です。拡張フォーマットではシンボル `menu-item`で始まるリストでイベント型を定義します。選択できない文字列にたいしては以下のようなバインディングになります:

```
(menu-item item-name)
```

2 つ以上のダッシュで始まる文字列はリストのセパレーターを指定します。Section 22.17.1.3 [Menu Separators], page 450 を参照してください。

選択可能な実際のメニューアイテムを定義するには以下のような拡張フォーマットでバインドします:

```
(menu-item item-name real-binding
    . item-property-list)
```

ここで *item-name*はメニューアイテム文字列に評価される式。つまり文字列は定数である必要はない。

3 つ目の引数 *real-binding*は実行するコマンドでもよい (この場合には通常のメニューアイテムを取得する)。キーマップでもよく、この場合には結果はサブメニューとなる。最後に `nil`でもよく、この場合には選択不可なメニューアイテクを取得する。これは主に区切り線のようなものの作成時に有用。

リスト末尾 *item-property-list*はその他の情報を含むプロパティリストの形式をもつ。

以下はサポートされるプロパティのテーブルです:

`:enable` *form*
: *form*の評価結果はそのアイテムを有効にするかどうかを決定する (非 `nil`なら有効)。アイテムが無効なら▽まったくクリックできない。

`:visible` *form*
: *form*の評価結果はそのアイテムを実際にメニューに表示するかどうかを決定する (非 `nil`なら表示)。アイテムが非表示ならそのアイテムが定義されていないかのようにメニューが表示される。

`:help` *help*
: このプロパティ*help*の値はそのアイテム上にマウスがある間に表示する help-echo 文字列を指定する。この文字列は `help-echo`テキストプロパティ([Help display], page 826 を参照) と同じ方法で表示される。これはテキストや overlay にたいする `help-echo`プロパティとは異なり、文字列定数でなければならないことに注意。

`:button (`*type* `.` *selected*`)`
: このプロパティはラジオボタンとトグルボタンを定義する手段を提供する。CAR の *type* には、`:toggle`か`:radio`のいずれかを指定する。CDR の *selected*はフォームで、評価結果によってそのボタンがカレントで選択されているかどうかを指定する。

  トグル *(toggle)* は *selected*の値に応じて on か off のいずれかがラベルされるメニューアイテムである。コマンド自身は *selected*が `nil`なら `t`、`t`なら `nil`に *selected*を切り替える (toggle する) こと。以下は `debug-on-error`フラグが定義されているときにメニューアイテムをトグルする方法の例:

```
(menu-item "Debug on Error" toggle-debug-on-error
           :button (:toggle
                    . (and (boundp 'debug-on-error)
                           debug-on-error)))
```

  これは `toggle-debug-on-error`が変数 `debug-on-error`をトグルするコマンドとして定義されていることによって機能する。

  ラジオボタンとはメニューアイテムのグループであり、常にただ 1 つのメニューアイテムだけが選択される (selected)。そのためにはどのメニューアイテムが選択されているかを示す変数が存在する必要がある。グループ内の各ラジオボタンにたいする *selected* フォームは、そのボタンを選択するためにその変数が正しい値をもつかどうかをチェックする。そしてボタンのクリックにより変数をセットして、クリックされたボタンが選択される。

`:key-sequence` *key-sequence*
: このプロパティは等価なキーボード入力として表示するキーシーケンスを指定する。Emacs はメニューに *key-sequence*を表示する前に、真に *key-sequence*がそのメニューアイテムと等価か検証するので、これは正しいキーシーケンスを指定した場合のみ効果をもつ。*key-sequence*にたいする `nil`の指定は`:key-sequence`属性が存在しないことに等しい。

`:keys` *string*
: このプロパティはそのメニューにたいする等価なキーボード入力として表示される文字列 *string*を指定する。*string*内ではドキュメント構文 '`\\[...]`'を使用できる。

`:filter` *filter-fn*
: このプロパティはメニューアイテムを直接計算する手段を提供する。このプロパティの値 *filter-fn*は引数が 1 つの関数で、呼び出し時の引数は *real-binding*。この関数はかわりに使用するバインディングをリターンすること。

  Emacs、メニューデータ構造の再表示や操作を行うすべてのタイミングでこの関数を呼び出すかもしれないので、いつ呼び出されても安全なように関数を記述すること。

22.17.1.3 メニューセパレーター

メニューセパレーターはテキストを表示するかわりに、水平ラインでメニューをサブパーツに分割するメニューアイテムの一種です。メニューキーマップ内でセパレーターは以下のように見えるでしょう:

```
(menu-item separator-type)
```

ここで *separator-type*は 2 つ以上のダッシュで始まる文字列です。

もっとも単純なケースではダッシュだけで *separator-type*が構成されます。これはデフォルトのセパレーターを指定します (互換性のため`""`と`-`もセパレーターとみなされる)。

*separator-type*にたいする他の特定の値は、異なるスタイルのセパレーターを指定します。以下はそれらのテーブルです:

`"--no-line"`
`"--space"`
: 実際のラインではない余分な垂直スペース。

`"--single-line"`
: メニューの foreground カラーの一重ライン。

`"--double-line"`
: メニューの foreground カラーの二重ライン。

`"--single-dashed-line"`
: メニューの foreground カラーの一重ダッシュライン。

`"--double-dashed-line"`
: メニューの foreground カラーの二重ダッシュライン。

`"--shadow-etched-in"`
: 3D の窪んだ外観 (3D sunken appearance) をもつ一重ライン。これはダッシュだけで構成されるセパレーターに使用されるデフォルト。

`"--shadow-etched-out"`
: 3D の浮き上がった外観 (3D raised appearance) をもつ一重ライン。

`"--shadow-etched-in-dash"`
3D の窪んだ外観 (3D sunken appearance) をもつ一重ダッシュライン。

`"--shadow-etched-out-dash"`
3D の浮き上がった外観 (3D raised appearance) をもつ一重ダッシュライン。

`"--shadow-double-etched-in"`
3D の窪んだ外観をもつ二重ライン。

`"--shadow-double-etched-out"`
3D の浮き上がった外観をもつ二重ライン。

`"--shadow-double-etched-in-dash"`
3D の窪んだ外観をもつ二重ダッシュライン。

`"--shadow-double-etched-out-dash"`
3D の浮き上がった外観をもつ二重ダッシュライン。

2 連ダッシュの後にコロンを追加して 1 連ダッシュの後の単語の先頭の文字を大文字にすることによって、別のスタイルで名前を与えることもできます。つまり`"--:singleLine"`は`"--single-line"`と等価です。

メニューセパレーターにたいして`:enable`や`:visible`のようなキーワードを指定するために長い形式を使用できます。

`(menu-item` *`separator-type`* `nil .` *`item-property-list`*`)`

たとえば:

```
(menu-item "--" nil :visible (boundp 'foo))
```

いくつかのシステムとディスプレイツールキットは、これらすべてのセパレータータイプを実際に処理しません。サポートされていないタイプのセパレーターを使用すると、メニューはサポートされている似た種別のセパレーターを表示します。

22.17.1.4 メニューアイテムのエイリアス

同じコマンドを使用するものの有効条件が異なるメニューアイテムを作成できれば便利な場合が時折あります。Emacs でこれを行う最善の方法は拡張メニューアイテム (extended menu item) です。この機能が存在する以前にはエイリアスコマンドを定義して、それらをメニューアイテムで使用することによってこれを行っていました。以下は`read-only-mode`にたいして 2 つのエイリアスを作成して、それらに異なる有効条件を与える例です:

```
(defalias 'make-read-only 'read-only-mode)
(put 'make-read-only 'menu-enable '(not buffer-read-only))
(defalias 'make-writable 'read-only-mode)
(put 'make-writable 'menu-enable 'buffer-read-only)
```

メニュー内でエイリアスを使用するときには、エイリアスではなく実際のコマンド名にたいする等価なキーバインディングを表示する方が便利な場合が多々あります (エイリアスは通常はメニュー自身を除いてキーバインディングをもたない)。これを要求するにはエイリアスシンボルの`menu-alias`プロパティに非`nil`を与えます。したがって、

```
(put 'make-read-only 'menu-alias t)
(put 'make-writable 'menu-alias t)
```

は`make-read-only`と`make-writable`にたいするメニューアイテムに`read-only-mode`のキーバインディングを表示します。

22.17.2 メニューとマウス

メニューキーマップがメニューを生成する通常の方法は、それをプレフィクスキーの定義とすることです (Lisp プログラムは明示的にメニューをポップアップしてユーザーの選択を受け取ることができる。Section 29.17 [Pop-Up Menus], page 744 を参照)。

プレフィクスキーがマウスイベントで終わる場合には、Emacs はユーザーがマウスで選択できるように可視なメニューをポップアップすることによってメニューキーマップを処理します。ユーザーがメニューアイテムをクリックしたときは、そのメニューアイテムによりもたらされるバインディングの文字やシンボルが何であれイベントが生成されます (メニューが複数レベルをもつ場合やメニューバー由来ならメニューアイテムは 1 連のイベントを生成するかもしれない)。

メニューのトリガーに button-down イベントを使用するのが最善な場合もしばしばあります。その場合にはユーザーはマウスボタンをリリースすることによってメニューアイテムを選択できます。

メニューキーマップがネストされたキーマップにたいするバインディングを含む場合、そのネストされたキーマップはサブメニュー *(submenu)* を指定します。それはネストされたキーマップのアイテム文字列によってラベル付けされれメニューアイテムをもち、そのアイテムをクリックすることによって指定されたサブメニューが自動的にポップアップされます。特別な例外としてメニューキーマップが単一のネストされたキーマップを含み、それ以外のメニューアイテムを含まなければ、そのメニューはネストされたキーマップの内容をサブメニューとしてではなく直接メニューに表示します。

しかし X ツールキットのサポートなしで Emacs をコンパイルした場合、またはテキスト端末の場合にはサブメニューはサポートされません。ネストされたキーマップはメニューアイテムとして表示されますが、それをクリックしてもサブメニューは自動的にポップアップされません。サブメニューの効果を模倣したければ、ネストされたキーマップに '`@`'で始まるアイテム文字列を与えることによってこれを行うことができます。これにより Emacs は別個のメニューペイン *(menu pane)* を使用してネストされたキーマップを表示します。'`@`'の後の残りのアイテム文字列はそのペインのラベルです。X ツールキットのサポートなしで Emacs をコンパイルした場合、またはメニューがテキスト端末で表示されている場合にはメニューペインは使用されません。この場合はアイテム文字列の先頭の '`@`'は、メニューラベル表示時には省略されて他に効果はありません。

22.17.3 メニューとキーボード

キーボードイベント (文字かファンクションキー) で終わるプレフィクスキーがメニューキーマップであるような定義をもつときには、そのキーマップはキーボードメニューのように動作します。ユーザーはキーボードでメニューアイテムを選択して次のイベントを指定します。

Emacs はエコーエリアにキーボードメニュー、そのマップの overall プロンプト文字列、その後に選択肢 (そのマップのバインディングのアイテム文字列) を表示します。そのバインディングを一度に全部表示できない場合、ユーザーは `SPC`をタイプして候補の次の行を確認できます。連続して `SPC` を使用するとメニューの最後に達して、その後は先頭へ巡回します (変数 `menu-prompt-more-char` はこのために使用する文字を指定する。デフォルトは `SPC`)。

ユーザーがメニューから望ましい候補を見つけたら、バインディングがその候補であるような対応する文字をタイプする必要があります。

`menu-prompt-more-char` [Variable]
: この変数はメニューの次の行を確認するために使用する文字を指定する。初期値は 32 でこれは `SPC`のコード。

22.17.4 メニューの例

以下はメニューキーマップを定義する完全な例です。これはメニューバー内の '`Edit`'メニューにサブメニュー '`Replace`'を定義して、その定義内で拡張メニューフォーマット (Section 22.17.1.2 [Extended

Menu Items], page 448 を参照) を使用します。例ではまずキーマップを作成してそれに名前をつけています:

```
(defvar menu-bar-replace-menu (make-sparse-keymap "Replace"))
```

次にメニューアイテムを定義します:

```
(define-key menu-bar-replace-menu [tags-repl-continue]
  '(menu-item "Continue Replace" multifile-continue
              :help "Continue last tags replace operation"))
(define-key menu-bar-replace-menu [tags-repl]
  '(menu-item "Replace in tagged files" tags-query-replace
              :help "Interactively replace a regexp in all tagged files"))
(define-key menu-bar-replace-menu [separator-replace-tags]
  '(menu-item "--"))
;; ...
```

バインディングがそのシンボルのために作成されることに注意してください。これらのシンボルは定義されるキーシーケンス内の角カッコ内に記述されます。このシンボルはコマンド名と同じときもあれば異なることもあります。これらのシンボルはファンクションキーとして扱われますが、これらはキーボード上の実際のファンクションキーではありません。これらはメニュー自体の機能に影響しませんが、ユーザーがメニューから選択したときにエコーエリアにエコーされて、`where-is`と `apropos` の出力に現れます。

この例のメニューはマウスによる使用を意図しています。もしキーボードの使用を意図したメニュー、つまりキーボードイベントで終了するキーシーケンスにバインドされたメニューの場合には、メニューアイテムはキーボードでタイプできる文字、または本当のファンクションキーにバインドされるべきです。

定義が `("--")`のバインディングはセパレーターラインです。実際のメニューアイテムと同様にセパレーターはキーシンボルをもち、この例では `separator-replace-tags`です。1 つのメニューが 2 つのセパレーターをもつ場合には、それらは 2 つの異なるキーシンボルをもたなければなりません。

以下では親メニュー内のアイテムとしてこのメニューがどのように表示されるかを記述しています:

```
(define-key menu-bar-edit-menu [replace]
  (list 'menu-item "Replace" menu-bar-replace-menu))
```

これはシンボル `menu-bar-replace-menu`自体ではなく、変数 `menu-bar-replace-menu`の値であるサブメニューキーマップを組み込むことに注意してください。`menu-bar-replace-menu`はコマンドではないので親メニューアイテムにそのシンボルを使用するのは無意味です。

同じ replace メニューをマウスクリックに割り当てたければ以下のようにしてこれを行うことができます:

```
(define-key global-map [C-S-down-mouse-1]
   menu-bar-replace-menu)
```

22.17.5 メニューバー

Emacs は通常は各フレームの最上部にメニューバー *(menu bar)* を表示します。Section “Menu Bars” in *The GNU Emacs Manual* を参照してください。メニューバーのアイテムはアクティブキーマップ内で定義される偽りのファンクションキー `MENU-BAR`のサブコマンドです。

メニューバーにアイテムを追加するには、自分で偽りのファンクションキー (これを *key*と呼ぶこととする) を創作して、キーシーケンス `[menu-bar` *key*`]`にたいするキーバインディングを作成します。ほとんどの場合において、そのバインディングはメニューキーマップなので、メニューバーアイテム上でボタンを押下すると他のメニューに導かれます。

メニューバーにたいして同じファンクションキーを定義するアクティブなキーマップが 1 つ以上存在するとき、そのアイテムは一度だけ出現します。ユーザーがメニューバーのそのアイテムをクリックすると、そのアイテムのすべてのサブコマンド、すなわちグローバルサブコマンド、ローカルサブコマンド、マイナーモードサブコマンドが組み合わされた単一のメニューを表示します。

変数 `overriding-local-map`は通常はメニューバーのコンテンツを決定する際は無視されます。つまりメニューバーは `overriding-local-map`が `nil`の場合にアクティブになるであろうキーマップから計算されます。Section 22.7 [Active Keymaps], page 429 を参照してください。

以下はメニューバーのアイテムをセットアップする例です:

```
;; (プロンプト文字列とともに) メニューキーマップを作成して
;; それをメニューバーアイテムの定義にする
(define-key global-map [menu-bar words]
  (cons "Words" (make-sparse-keymap "Words")))

;; メニュー内に具体的なサブコマンドを定義する
(define-key global-map
  [menu-bar words forward]
  '("Forward word" . forward-word))
(define-key global-map
  [menu-bar words backward]
  '("Backward word" . backward-word))
```

ローカルキーマップはグローバルキーマップにより作成されたメニューバーアイテムにたいして、同じ偽ファンクションキーを `undefined`にリバインドしてキャンセルすることができます。たとえば以下は Dired が '`Edit`'メニューバーアイテムを抑制する方法です:

```
(define-key dired-mode-map [menu-bar edit] 'undefined)
```

ここで `edit`はメニューバーアイテム '`Edit`'にたいしてグローバルキーマップが使用する偽ファンクションキーが生成するシンボルです。グローバルメニューバーアイテムを抑制する主な理由は、モード特有のアイテムのためのスペースを確保するためです。

`menu-bar-final-items` [Variable]

メニューバーは通常はローカルマップで定義されるアイテムを終端にもつグローバルアイテムを表示する。

この変数は通常の順番による位置ではなく、メニューの最後に表示するアイテムのための偽ファンクションキーのリストを保持する。デフォルト値は `(help-menu)`。したがって '`Help`' メニューアイテムはメニューバーの最後、ローカルメニューアイテムの後に表示される。

`menu-bar-update-hook` [Variable]

このノーマルフックはメニューバーの再表示の前に、メニューバーのコンテンツ更新のための再表示によって実行される。コンテンツを変化させる必要があるメニューの更新に使用できる。このフックは頻繁に実行されるので、フックが呼び出す関数は通常は長い時間を要さないことを確実にするよう助言する。

Emacs はすべてのメニューバーアイテムの隣に、(もしそのようなキーバインディングが存在するなら) 同じコマンドを実行するキーバインディングを表示します。これはキーバインディングを知らないユーザーにたいして有用なヒントを与える役目をもちます。コマンドが複数のバインディングをもつ場合、Emacs は通常は最初に見つけたバインディングを表示します。コマンドのシンボルプロパティ `:advertised-binding`に割り当てることによって特定のキーバインディングを指定できます。Section 24.3 [Keys in Documentation], page 522 を参照してください。

22.17.6 ツールバー

ツールバー *(tool bar)* とはフレームの最上部、メニューバー直下にあるクリック可能なアイコンの行のことです。Section “Tool Bars” in *The GNU Emacs Manual* を参照してください。Emacs は通常はグラフィカルなディスプレイ上でツールバーを表示します。

各フレームではツールバーに何行分の高さを割り当てるかをフレームパラメーター `tool-bar-lines`で制御します。値 0 はツールバーを抑制します。値が非 0 で `auto-resize-tool-bars`が非`nil`なら、指定されたコンテンツを維持するのに必要な分、ツールバーは拡大縮小されます。値が`grow-only`ならツールバーは自動的に拡大されますが、自動的に縮小はされません。

ツールバーのコンテンツは、(メニューバーが制御されるのと似た方法により) `TOOL-BAR`と呼ばれる偽りのファンクションキーに割り当てられたメニューキーマップにより制御されます。したがって以下のように `define-key`を使用してツールバーアイテムを定義します。

```
(define-key global-map [tool-bar key] item)
```

ここで *key*はそのアイテムを他のアイテムと区別する偽ファンクションキー、*item*はそのアイテムを表示する方法とアイテムの振る舞いを示すメニューアイテムキーバインディングです (Section 22.17.1.2 [Extended Menu Items], page 448 を参照)。

メニューキーマップの通常のプロパティ`:visible`、`:enable`、`:button`、`:filter`はツールバーバインディングでも有用で、いずれのプロパティも通常通りの意味をもちます。アイテム内の *real-binding*はキーマップではなくコマンドでなければなりません。言い換えるとこれはツールバーアイコンをプレフィクスキーとして定義するようには機能しないということです。

`:help`プロパティは、そのアイテム上にマウスがある間表示する help-echo 文字列を指定します。これはテキストプロパティ`help-echo`と同じ方法で表示されます ([Help display], page 826 を参照)。

これらに加えて`:image`プロパティも使用するべきでしょう。ツールバー内にイメージを表示するにはこのプロパティを使用します。

`:image` *image*

*image*は単一イメージ様式 (single image specification) か 4 ベクターイメージ様式 (vector of four image specifications) で指定する (Section 39.17 [Images], page 1049 を参照)。4 ベクターを使用する場合には状況に応じて以下のいずれかが使用される:

item 0 アイテムが有効かつ選択されているときに使用。

item 1 アイテムが有効かつ未選択のときに使用。

item 2 アイテムが無効かつ選択されているときに使用。

item 3 アイテムが無効かつ未選択のときに使用。

GTK+バージョンと NS バージョンの Emacs は、無効および/または未選択のイメージを item0 から自動的に計算するので、item1 から item3 は無視されます。

*image*が単一イメージ様式なあ、Emacs はそのイメージにエッジ検出アルゴリズム (edge-detection algorithm) を適用することによってツールバーの無効な状態のボタンを描画します。

`:rtl`プロパティには右から左に記述する言語のためのイメージ候補を指定します。これをサポートするのは現在のところ GTK+バージョンの Emacs だけです。

メニューバーと同様、ツールバーはセパレーター (Section 22.17.1.3 [Menu Separators], page 450 を参照) を表示できます。ツールバーのセパレーターは水平ラインではなく垂直ラインであり、1 つのスタイルだけがサポートされます。これらはツールバーキーマップ内では `(menu-item`

"--")エントリーで表されます。ツールバーのセパレーターでは、:visibleのようなプロパティはサポートされません。GTK+と Nextstep のツールバーでは、セパレーターはネイティブに描画されます。それ以外ではセパレーターは垂直ラインイメージを使用して描画されます。

デフォルトツールバーはコマンドシンボルの mode-classプロパティに specialをもつメジャーモードにたいしては、編集に特化したアイテムは表示しないよう定義されています (Section 23.2.1 [Major Mode Conventions], page 464 を参照)。メジャーモードは、ローカルマップ内でバインディング [tool-bar *foo*]によって、グローバルバーにアイテムを追加するかもしれません。デフォルトツールバーの多くを適宜流用するのができないかもしれないので、デフォルトツールバーを完全に置き換えることは、いくつかのメジャーモードにとっては有意義です。デフォルトバインディングで tool-bar-mapを通じてインダイレクトすることにより、これを簡単に行うことができます。

tool-bar-map [Variable]

デフォルトではグローバルマップは [tool-bar]を以下のようにバインドする:

```
(global-set-key [tool-bar]
                `(menu-item ,(purecopy "tool bar") ignore
                            :filter tool-bar-make-keymap))
```

関数 tool-bar-make-keymapは、変数 tool-bar-mapの値より順番に実際のツールバーマップをダイナミックに継承する。したがって通常はそのマップを変更することにより、デフォルト (グローバル) ツールバーを調整すること。Info モードのようないくつかのメジャーモードは、tool-bar-mapをバッファーローカルにして、それに異なるキーマップをセットすることによりグローバルツールバーを完全に置き換える。

以下のようなツールバーアイテムを定義するのに便利な関数があります。

tool-bar-add-item *icon def key* **&rest** *props* [Function]

この関数は tool-bar-mapを変更することにより、ツールバーにアイテムを追加する。使用するイメージは *icon*により定義され、これは find-imageに配置された XPM、XBM、PBM のイメージファイルの拡張子を除いたファイル名 (basename) である。たとえばカラーディスプレイ上では、値に '"exit"'を与えると exit.xpm、exit.pbm、exit.xbmの順に検索されるだろう。モノクロディスプレイでは検索は '.pbm'、'.xbm'、'.xpm'の順になる。使用するバインディングはコマンド *def*で、*key*はプレフィクスキーマップ内の偽ファンクションキーである。残りの引数 *props*はメニューアイテム仕様に追加する追加のプロパティリスト要素である。

あるローカルマップ内にアイテムを定義するためには、この関数呼び出しの周囲の letで tool-bar-mapをバインドする:

```
(defvar foo-tool-bar-map
  (let ((tool-bar-map (make-sparse-keymap)))
    (tool-bar-add-item ...)
    ...
    tool-bar-map))
```

tool-bar-add-item-from-menu *command icon* **&optional** *map* **&rest** *props* [Function]

この関数は既存のメニューバインディングと矛盾しないツールバーアイテムの定義に有用。*command*のバインディングは *map*(デフォルトは global-map) 内よりルックアップ (lookup: 照合) され、*icon*にたいするイメージ仕様は tool-bar-add-itemと同じ方法で見つけ出される。結果のバインディングは tool-bar-mapに配置されるので、この関数の使用はグローバルツールバーアイテムに限定される。

*map*には `[menu-bar]` にバインドされた適切なキーマップが含まれていなければならない。残りの引数 *props*はメニューアイテム仕様に追加する追加のプロパティリスト要素。

`tool-bar-local-item-from-menu` *command icon in-map* **&optional** *from-map* **&rest** *props* [Function]

この関数は非グローバルツールバーアイテムの作成に使用される。*in-map*に定義を作成するローカルマップを指定する以外は `tool-bar-add-item-from-menu`と同じように使用する。引数 *from-map*は `tool-bar-add-item-from-menu`の *map*と同様。

`auto-resize-tool-bars` [Variable]

この変数が非 `nil`なら定義されたすべてのツールバーアイテムを表示するためにツールバーは自動的にリサイズされるが、そのフレーム高さの 1/4 を超えてリサイズされることはない。

値が `grow-only`ならツールバーは自動的に拡張されるが縮小はされない。ツールバーを縮小するためにユーザーは *C-l*をエンターしてフレームを再描画する必要がある。

GTK+や Nextstep とともに Emacs がビルドされた場合には、ツールバーが表示できるのは 1 行だけであり、この変数は効果がない。

`auto-raise-tool-bar-buttons` [Variable]

この変数が非 `nil`ならツールバーアイテム上をマウスが通過したとき、浮き上がった形式 (raised form) で表示される。

`tool-bar-button-margin` [Variable]

この変数はツールバーアイテムの周囲に追加する余白 (extra margin) を指定する。値はピクセル数を整数で指定する。デフォルトは 4。

`tool-bar-button-relief` [Variable]

この変数はツールバーアイテムの影 (shadow) を指定する。値はピクセル数を整数で指定する。デフォルトは 1。

`tool-bar-border` [Variable]

この変数はツールバーエリアの下に描画するボーダー高さを指定する。値が整数なら高さのピクセル数。値が `internal-border-width`(デフォルト) か `border-width`のいずれかなら、ツールバーのボーダー高さはそのフレームの対応するパラメーターとなる。

シフトやメタ等の修飾キーを押下した状態でのツールバーアイテムのクリックに特別な意味を付与できます。偽りのファンクションキーを通じて元のアイテムに関連する追加アイテムをセットアップすることによって、これを行うことができます。より具体的には追加アイテムは、元のアイテムの命名に使用されたのと同じ偽ファンクションキーの修飾されたバージョンを使用するべきです。

つまり元のアイテムが以下のように定義されていれば、

```
(define-key global-map [tool-bar shell]
  '(menu-item "Shell" shell
              :image (image :type xpm :file "shell.xpm")))
```

シフト修飾とともに同じツールバーイメージをクリックしたときを以下のような方法で定義することができます:

```
(define-key global-map [tool-bar S-shell] 'some-command)
```

ファンクションキーに修飾を追加する方法についての詳細な情報は、Section 21.7.2 [Function Keys], page 389 を参照してください。

22.17.7 メニューの変更

既存のメニューに新たなアイテムを挿入するときは、そのメニューの既存のアイテムの中の特定の位置にアイテムを追加したいと思うかもしれません。`define-key`を使用してアイテムを追加すると、そのアイテムは通常はメニューの先頭に追加されます。メニュー内の他の位置にアイテムを追加するには `define-key-after`を使用します:

`define-key-after` *map key binding* **&optional** *after* [Function]

`define-key`と同じように *map*内に *key*にたいする値 *binding*のバインディングを定義するが、*map*内でのバインディング位置はイベント *after*のバインディングの後になる。引数 *key* は長さ 1 — 1 要素だけのベクターか文字列にすること。しかし *after*は単一のイベント型 — シーケンスではないシンボルか文字にすること。新たなバインディングは *after*のバインディングの後に追加される。*after*が `t`または省略された場合には、新たなバインディングはそのキーマップの最後に追加される。しかし新たなバインディングは継承されたすべてのキーマップの前に追加される。

以下は例:

```
(define-key-after my-menu [drink]
  '("Drink" . drink-command) 'eat)
```

これは偽ファンクションキー `DRINK`のバインディングを作成して、`EAT`のバインディングの直後に追加する。

以下は Shell モードの '`Signals`'メニュー内のアイテム `break`の後に '`Work`'と呼ばれるアイテムを追加する方法:

```
(define-key-after
  (lookup-key shell-mode-map [menu-bar signals])
  [work] '("Work" . work-command) 'break)
```

22.17.8 easy-menu

以下のマクロはポップアップメニューおよび/またはメニューバーメニューを定義する便利な方法を提供します。

`easy-menu-define` *symbol maps doc menu* [Macro]

このマクロは *menu*により与えるコンテンツのポップアップメニューおよび/またはメニューバーサブメニューを定義する。

*symbol*が非 `nil`なら、それはシンボルである。その場合、このマクロはドキュメント文字列 *doc*をもつ、メニューをポップアップ (Section 29.17 [Pop-Up Menus], page 744 を参照) する関数として *symbol*を定義する。*symbol*はクォートしないこと。

*symbol*の値とは関係なく、*maps*がキーマップならメニューはメニューバーのトップレベルのメニュー (Section 22.17.5 [Menu Bar], page 453 を参照) として *maps*に追加される。これにはキーマップのリストも指定でき、その場合メニューはそれらのキーマップに個別に追加される。

*menu*の最初の要素は文字列でなければならず、それはメニューラベルの役割をもつ。値には以下のキーワード/引数ペアーが任意の個数続くかもしれない:

`:filter` *function*

*function*は 1 つの引数 (他のメニューアイテムのリスト) で呼び出される関数でなければならず、メニュー内に表示される実際のアイテムをリターンする。

`:visible` *include*
: *include*には式を指定する。その式が`nil`に評価されるとメニューは不可視になる。`:included`は`:visible`にたいするエイリアス。

`:active` *enable*
: *enable*は式を指定する。その式が`nil`に評価されるとメニューは選択不可になる。`:enable`は`:active`にたいするエイリアス。

*menu*内の残りの要素はメニューアイテム。

メニューアイテムには 3 要素のベクター `[`*name callback enable*`]`を指定できる。ここで*name*はメニューアイテム名 (文字列)、*callback*はアイテム選択時に実行するコマンドか評価される式。*enable*が式で`nil`に評価されると、そのアイテムの選択は無効になる。

かわりにメニューアイテムは以下の形式をもつことができる:

```
[ name callback [ keyword arg ]... ]
```

ここで *name*と *callback*は上記と同じ意味をもち、オプションの *keyword*と *arg*の各ペアーは以下のいずれかである:

`:keys` *keys*
: *keys*はメニューアイテムにたいする等価なキーボード入力として表示する文字列。等価なキーボード入力は自動的に計算されるので通常は必要ない。*keys*は表示前に`substitute-command-keys`により展開される (Section 24.3 [Keys in Documentation], page 522 を参照)。

`:key-sequence` *keys*
: *keys*はコナドが複数のキーシーケンスにバインドされている場合に、等価なキーボード入力としてどのキーシーケンスを表示するかを示すためのヒント。このメニューアイテムとして *keys*が同じコマンドがバインドされていなければ効果はない。

`:active` *enable*
: *enable*には式を指定する。その式が`nil`に評価されるとアイテムは選択不可になる。*enable*は`:active`にたいするエイリアス。

`:visible` *include*
: *include*には式を指定する。その式が`nil`に評価されるとアイテムは不可視になる。`:included`は`:visible`にたいするエイリアス。

`:label` *form*
: *form*はメニューアイテムのラベル (デフォルトは *name*) の役目をもつ値を取得するために表示される式である。

`:suffix` *form*
: *form*は動的に評価される式であり、値はメニューエントリーのラベルに結合される。

`:style` *style*
: *style*はメニューアイテムの型を記述するシンボルであり、`toggle`(チェックボックス)、`radio`(ラジオボタン)、またはそれ以外 (通常のメニューアイテムであることを意味する) のいずれかである。

`:selected` *selected*
: *selected*には式を指定し、その式の値が非`nil`のときはチェックボックスまたはラジオボタンが選択状態になる。

`:help` *help*
: *help*はメニューアイテムを説明する文字列。

かわりにメニューアイテムに文字列を指定できる。その場合には文字列は選択不可なテキストとしてメニューに表示される。ダッシュから構成される文字列はセパレーターとして表示される (Section 22.17.1.3 [Menu Separators], page 450 を参照)

かわりにメニューアイテムに *menu*と同じフォーマットのリストを指定できる。これはサブメニューとなる。

以下は`easy-menu-define`を使用して Section 22.17.5 [Menu Bar], page 453 内で定義したメニューと同等なメニューを定義する例:

```
(easy-menu-define words-menu global-map
  "単語単位コマンドにたいするメニュー"
  '("Words"
     ["Forward word" forward-word]
     ["Backward word" backward-word]))
```

23 メジャーモードとマイナーモード

モード *(mode)* とは Emacs の挙動を簡便な方法でカスタマイズする定義のセットです。モードは 2 種類あります。マイナーモード *(minor modes)* は編集時にユーザーがオンとオフを切り替えられる機能を提供します。メジャーモード *(major modes)* は特定の種類のテキストにたいする編集や相互作用に使用します。ある時点においてバッファーはそれぞれ正確に 1 つのメジャーモードをもちます。

このチャプターではメジャーモードとマイナーモードを記述する方法、それらをモードラインに示す方法、そしてそれらのモードがユーザーが提供するフックを実行する方法を説明します。キーマップ (keymaps) や構文テーブル (syntax tables) のような関連するトピックについては Chapter 22 [Keymaps], page 422 と Chapter 35 [Syntax Tables], page 906 を参照してください。

23.1 フック

フック *(hook)* とは既存のプログラムから特定のタイミングで呼び出される関数 (複数可) を格納できる変数のことです。Emacs はカスタマイズ用にフックを提供します。ほとんどの場合には init ファイル内 (Section 40.1.2 [Init File], page 1093 を参照) でフックをセットアップしますが、Lisp プログラムもフックをセットできます。標準的なフック変数のリストは Appendix H [Standard Hooks], page 1223 を参照してください。

Emacs のほとんどのフックはノーマルフック *(normal hooks)* です。これらの変数は、引数なしで呼び出される関数のリストを含んでいます。慣習により名前が '`-hook`'で終わるフックは、そのフックがノーマルフックであることを意味します。わたしたちは一貫した方法でフックを使用できるように、すべてのフックが可能な限りノーマルフックとなるよう努力しています。

すべてのメジャーモードコマンドは、初期化の最終ステップの 1 つとして、モードフック *(mode hook)* と呼ばれるノーマルフックを実行するとみなされます。これによってそのモードですでに作成されたバッファーローカル変数割り当てをオーバーライドすることにより、ユーザーがそのモードの動作をカスタマイズするのが簡単になります。ほとんどのマイナーモード関数も最後にモードフックを実行します。しかしフックは他のコンテキストでも使用されます。たとえばフック `suspend-hook` は、Emacs が自身をサスペンド (Section 40.2.2 [Suspending Emacs], page 1098 を参照) する直前に実行されます。

フックにフック関数を追加するには `add-hook`(Section 23.1.2 [Setting Hooks], page 462 を参照) を呼び出す方法が推奨されています。フック関数には `funcall`(Section 13.1 [What Is a Function], page 209 を参照) が受け入れる任意の種類の関数を指定できます。ほとんどのフック変数の初期値は void です。`add-hook`はこれを扱う方法を知っています。`add-hook`によりグローバルフックやバッファーローカルフックのいずれも追加することが可能です。

フック変数の名前が '`-hook`'で終わらなければ、それが恐らくアブノーマルフック *(abnormal hook)* であることを示しています。これはフック関数が引数とともに呼ぶ出されること、または何らかの方法によってそのリターン値が使用されることを意味します。その関数の呼び出し方はそのフックのドキュメントに記載されています。アブノーマルフックとして関数を追加するために `add-hook`を使用できますが、その関数はフック呼び出しの慣習にしたがって記述しななければなりません。慣習によりアブノーマルフックの名前の最後は '`-functions`'です。

変数の名前が '`-function`'で終われば、その値は関数のリストではなく単一の関数です。`add-hook` を単一関数フックのように修正して使用することはできないので、かわりに `add-function`を使用します (Section 13.11 [Advising Functions], page 227 を参照)。

23.1.1 フックの実行

このセクションではノーマルフックを実行するために使用される run-hooksについて説明します。またさまざまな種類のアブノーマルフックを実行する関数についても説明します。

`run-hooks` **&rest** *hookvars* [Function]

この関数は引数として 1 つ以上のノーマルフック変数名を受け取って、各フックを順に実行する。引数はそれぞれノーマルフック変数であるようなシンボルであること。これらの引数は指定された順に処理される。

フック変数の値が非 nilならその値は関数のリストであること。run-hooksはすべての関数を引数なしで 1 つずつ呼び出す。

フック変数の値には、単一の関数 (ラムダ式、またはシンボルの関数定義) も指定でき、その場合 run-hooksはそれを呼び出す。しかしこの使い方は時代遅れである。

フック変数がバッファーローカルならグローバル変数のかわりにそのバッファーローカル変数が使用される。しかしそのバッファーローカル変数が要素 tを含む場合には、そのグローバルフック変数も同様に実行されるだろう。

`run-hook-with-args` *hook* **&rest** *args* [Function]

この関数は、*hook*内のすべての関数に 1 つの引数 *args*を渡して呼び出すことによってアブノーマルフックを実行する。

`run-hook-with-args-until-failure` *hook* **&rest** *args* [Function]

この関数は各フック関数を順に呼び出すことによりアブノーマルフック関数を実行し、それらのうち 1 つが nilをリターンして失敗すると停止する。それぞれのフック関数は引数として *args* を渡される。この関数はフック関数の 1 つが失敗して停止したら nil、それ以外は非 nil値をリターンする。

`run-hook-with-args-until-success` *hook* **&rest** *args* [Function]

この関数は各フック関数を順に呼び出すことによりアブノーマルフック関数を実行して、それらのうち 1 つが非 nil値をリターンして成功したら停止する。それぞれのフック関数は引数として *args*を渡される。この関数はフック関数の 1 つが失敗して停止したらその値、それ以外は nilをリターンする。

23.1.2 フックのセット

以下は Lisp Interaction モードのときに Auto Fill モードをオンに切り替えるためにモードフックに関数を追加する例です:

```
(add-hook 'lisp-interaction-mode-hook 'auto-fill-mode)
```

フック変数の値は関数のリストにする必要があります。通常の Lisp 機能を使用してこのリストを操作できますが、モジュール方式では以下で説明する関数 add-hookと remove-hookを使用します。これらの関数はいくつかの異常な状況を処理して問題を回避します。

フックに lambda式を配置しても機能しますが、これは混乱を招くので避けることを推奨します。2 回目は記述を微妙に変えて同じ lambda式を追加すると、そのフックは等価な 2 つの別々の関数をもつことになります。それから一方を削除しても、もう一方は残り続けるでしょう。

`add-hook` *hook function* **&optional** *depth local* [Function]

この関数はフック変数に関数 *function*を追加する手軽な方法である。ノーマルフックと同じようにアブノーマルフックにたいしてもこの関数を使用できる。*function*には正しい数の引数を受け付ける任意の Lisp 関数を指定できる。たとえば、

```
(add-hook 'text-mode-hook 'my-text-hook-function)
```

は text-mode-hookと呼ばれるフックに my-text-hook-functionを追加する。

*hook*内に *function*がすでに存在する場合 (比較には equalを使用)、add-hookは 2 回目の追加を行わない。

*function*のプロパティpermanent-local-hookが非 nilなら kill-all-local-variables(またはメジャーモードを変更しても) はそのフック変数のローカル値から関数を削除しない。

ノーマルフックにたいしてフック関数は実行される順序に無関係であるようにデザインされるべきである。順序への依存はトラブルを招く。とはいえその順序は予測可能である。*function*は通常はフックリストの先頭に追加されるので、(他の add-hook呼び出しがなければ) それは最初に実行される。

いくつかのケースではフック上の相対順序の制御が重要になる。オプション引数によりリストのどこに関数を挿入するべきかを指定できる。値は-100 から 100 の数値であり、より大きい値では関数はリストの終端に近づく。*depth*のデフォルトは 0 であり、後方互換のために非 nil なら depth を 90 と解釈する。さらに *depth*が厳密に 0 より大なら、関数は同じ depth の関数の前ではなく後に追加される。あなたの関数の前 (や後) に他の関数を配置する必要が絶対ないとは限らないので、100(や-100) のを depth は決して使用しないこと。

add-hookは *hook*が void のとき、または値が単一の関数の場合には、値を関数リストにセットまたは変更してそれらを扱うことができる。

*local*が非 nilなら、グローバルフックリストではなくバッファーローカルフックリストに *function*を追加する。これはフックをバッファーローカルにして、そのバッファーローカルな値にtを追加する。バッファーローカルな値への tの追加は、ローカル値と同じようにデフォルト値でもフック関数を実行するためのフラグである。

`remove-hook` *hook function* **&optional** *local* [Function]

この関数はフック変数 *hook*から *function*を削除する。これは equalを使用して *function*と *hook*要素を比較するので、その比較はシンボルとラムダ式の両方で機能する。

*local*が非 nilなら、それはグローバルフックリストではなくバッファーローカルフックリストから *function*を削除する。

23.2 メジャーモード

メジャーモードは特定の種類のテキストの編集や相互作用にたいして Emacs を特化します。すべてのバッファーは一度に 1 つのメジャーモードをもちます。すべてのメジャーモードは、メジャーモードコマンド *(major mode command)* に関連付けられ、そのコマンド名は '-mode'で終わるべきです。このコマンドは、ローカルキーマップのようなさまざまなバッファーローカル変数をセットすることにより、カレントバッファーないでそのモードに切り替える配慮をします。Section 23.2.1 [Major Mode Conventions], page 464 を参照してください。マイナーモードとは異なりメジャーモードを "オフに切り替える" 手段は存在せず、かわりにバッファーは別のメジャーモードに切り替えられなければなりません。しかしメアーモードを一時的にサスペンドして、後でサスペンドしたモードをリストアできます。以下を参照してください。

Fundamental モードと呼ばれるモードはもっとも特化されていないメジャーモードであり、モード特有な定義や変数セッティングをもちません。

`fundamental-mode` [Command]

これは Fundamental モードにたいするメジャーモードコマンドである。他のモードコマンドと異なり、このモードはカスタマイズしてはならないことになっているので、モードフックは何も実行されない (Section 23.2.1 [Major Mode Conventions], page 464 を参照)。

`major-mode-suspend` [Function]
この関数はすべてのバッファーローカル変数を kill する点において `fundamental-mode`のように機能するが、これは後でリストアできるように効力をもつメジャーモードを記録する。この関数と `major-mode-restore` (以下参照) は、Emacs がそのバッファー用に自動的に選択したモード (Section 23.2.2 [Auto Major Mode], page 468 を参照) ではない何らかの特化したモードにバッファーを置く必要があり、なおかつ後で元のモードに戻れるようにしたい場合に有用。

`major-mode-restore` **&optional** *avoided-modes* [Function]
この関数は `major-mode-suspend`が記録したメジャーモードをリストアする。メジャーモードが何も記録されていなければ、この関数は `normal-mode` (Section 23.2.2 [Auto Major Mode], page 468 を参照) を呼び出すが、*avoided-modes*引数が非 `nil`ならこの引数内のモードを選択させないように試みる。

メジャーモードを記述するもっとも簡単な方法はマクロ `define-derived-mode`を使用する方法です。これは既存のメジャーモードを変形して新たなモードをセットアップします。Section 23.2.4 [Derived Modes], page 470 を参照してください。`define-derived-mode`は多くのコーディング規約を自動的に強要するので、たとえ新たなモードが他のモードから明示的に派生されない場合でも、わたしたちは `define-derived-mode`の使用を推奨します。派生元とするための一般的なモードについては Section 23.2.5 [Basic Major Modes], page 472 を参照してください。

標準的な GNU Emacs の Lisp ディレクトリーツリーには、いくつかのメジャーモードが `text-mode.el`、`texinfo.el`、`lisp-mode.el`、`rmail.el`のようなファイルとして含まれています。モードの記述方法を確認するために、これらのライブラリーを学ぶことができます。

`major-mode` [User Option]
この変数のバッファーローカル値はカレントのメジャーモードにたいするシンボルを保持する。この変数のデフォルト値は新たなバッファーにたいするデフォルトのメジャーモードを保持する。標準的なデフォルト値は `fundamental-mode`である。

デフォルト値が `nil`なら、*C-x b* (`switch-to-buffer`) のようなコマンドを通じて Emacs が新たなバッファーを作成したとき、新たなバッファーは以前カレントだったバッファーのメジャーモードになる。例外として以前のバッファーのメジャーモードのシンボルプロパティ `mode-class`が値 `special`をもつ場合には、新たなバッファーは Fundamental モードになる (Section 23.2.1 [Major Mode Conventions], page 464 を参照)。

23.2.1 メジャーモードの慣習

メジャーモードにたいするすべてのコードはさまざまなコーディング規約にしたがうべきであり、それらの規約にはローカルキーマップおよび構文テーブルの初期化、関数名や変数名、フックにたいする規約が含まれます。

`define-derived-mode`マクロを使用すれば、これらの規約を自動的に配慮します。Section 23.2.4 [Derived Modes], page 470 を参照してください。Fundamental モードは Emacs のデフォルト状態を表すモードなので、これらの規約が該当しないことに注意してください。

以下の規約リストはほんの一部です。一般的にすべてのメジャーモードは Emacs 全体が首尾一貫するよう、他の Emacs メジャーモードとの一貫性を目指すべきです。ここでこの問題を洗い出すすべての想定される要点をリストするのは不可能です。自身の開発するメジャーモードが通常の規約を逸脱する領域を示すような場合には、Emacs 開発者は互換性を保つようにしてください。

- 名前が '`-mode`'で終わるようにメジャーモードコマンドを定義する。引数なしで呼び出されたときこのコマンドはキーマップ、構文テーブル、既存バッファーのバッファーローカル変数をセッ

トアップして、カレントバッファーを新たなモードに切り替えること。そのバッファーのコンテンツを変更しないこと。

- そのモードで利用できる特別なコマンドを説明するドキュメント文字列を記述する。Section 23.2.3 [Mode Help], page 470 を参照のこと。

  そのユーザー自身のキーバインディングに自動的に適合してヘルプが表示されるように、ドキュメント文字列に特別なドキュメントサブストリング '`\[`*command*`]`'、'`\{`*keymap*`}`'、'`\<`*keymap*`>`' を含めるとよいかもしれない。Section 24.3 [Keys in Documentation], page 522 を参照のこと。
- メジャーモードコマンドは `kill-all-local-variables`を呼び出すことによって開始すること。これはノーマルフック `change-major-mode-hook`を実行してから、前のメジャーモードで効力のあったバッファーローカル変数を解放する。Section 12.11.2 [Creating Buffer-Local], page 192 を参照のこと。
- メジャーモードコマンドは変数 `major-mode`にメジャーモードコマンドのシンボルをセットすること。これは `describe-mode`がプリントするドキュメントを探す手掛かりとなる。
- メジャーモードコマンドは変数 `mode-name`にそのモードの "愛称 (pretty name)" をセットすること (これは通常は文字列だが他の利用可能な形式については Section 23.4.2 [Mode Line Data], page 486 を参照)。このモード名はモードラインに表示される。
- 連続してメジャーモードコマンドを直接呼び出しても失敗せずに、1 回だけ呼び出されたときと同じことを行うこと。言い換えるとメジャーモードコマンドはべき等であること。
- すべてのグローバル名は同じネームスペースにあるので、モードの一部であるようなすべてのグローバルな変数、定数、関数はメジャーモード名 (メジャーモード名が長いようなら短縮名) で始まる名前をもつこと。Section D.1 [Coding Conventions], page 1162 を参照されたい。
- プログラム言語のようなある種の構造型テキストを編集するためのメジャーモードでは、その構造に応じたテキストのインデントがおそらく有用であろう。したがってそのようなモードは `indent-line-function`に適切な関数をセットするとともに、インデント用のその他の変数をカスタマイズするべきだろう。Section 23.7 [Auto-Indentation], page 508 を参照のこと。
- メジャーモードは、通常はそのモードにあるすべてのバッファーのローカルキーマップとして使用されるモード自身のキーマップをもつこと。メジャーモードコマンドはそのローカルマップをインストールするために、`use-local-map`を呼び出すこと。詳細は Section 22.7 [Active Keymaps], page 429 を参照されたい。

  このキーマップは *modename*`-mode-map`という名前のグローバル変数に永続的に格納されること。そのモードを定義するライブラリーは、通常はこの変数をセットする。

  モード用のキーマップ変数をセットアップするコードの記述する方法に関するアドバイスは Section 12.6 [Tips for Defining], page 180 を参照されたい。
- メジャーモードのキーマップ内でバインドされるキーシーケンスは、通常は *C-c*で始まってその後にコントロール文字、数字、`{`、`}`、`<`、`>`、`:`、`;`が続くこと。その他の記号文字 (punctuation characters) はマイナーモード、通常のアルファベット文字はユーザー用に予約済みである。

  メジャーモードは *M-n*、*M-p*、*M-s*などのキーもリバインドできる。*M-n*と *M-p*にたいするバインディングは、通常は前方か後方への移動を意味するような類のものであるべきだが、これが必ずしもカーソル移動を意味する必要はない。

  そのモードにより適した方法でテキストに同じ処理を行うコマンドを提供する場合には、メジャーモードが標準的なキーシーケンスをリバインドするのは正当性がある。たとえばプログラム言語を編集するためのメジャーモードは、その言語にとって関数の先頭に移動するために、より良く機能する方法で *C-M-a*を再定義するかもしれない。メジャーモードにニーズに応じて *C-M-a*を構成する

ための推奨方法は、そのモード固有の関数を呼び出すために `beginning-of-defun-function` をセットすること (Section 30.2.6 [List Motion], page 762 を参照)。

ある標準的なキーシーケンスの標準的な意味がそのモードではほとんど役に立たないような場合にも、メジャーモードが標準的なキーシーケンスをリバインドする正当性がある。たとえば *M-r* の標準的な意味はミニバッファーではほとんど使用されないので、このキーシーケンスをリバインドする。テキストの自己挿入を許さない Dired や Rmail のようなメジャーモードがアルファベット文字や、その他のプリント文字を特別なコマンドに再定義することには正当性がある。

- テキストを編集するメジャーモードは改行の挿入以外の何かに `RET`を定義すべきではない。しかしユーザーが直接テキストを編集しない、Dired や Info のような特別なモードにたいしては完全に異なることを行うように `RET`を再定義しても構わない。
- メジャーモードは、たとえば Auto-Fill モードを有効にする等の、主にユーザーの好みに関するオプションを変更しないこと。それらのオプションはユーザーに選択に任せること。ただしもしユーザーが Auto-Fill モードを使用すると決定したら、それが便利に機能するように他の変数をカスタマイズすること。
- モードは自身の構文テーブルをもつことができ、他の関連するモードと構文テーブルを共有するかもすることもできる。モードが自身の構文テーブルをもつ場合には、*modename*`-mode-syntax-table`という名前の変数にそれを格納すること。Chapter 35 [Syntax Tables], page 906 を参照されたい。
- コメントにたいする構文をもつ言語を扱うモードは、コメント構文を定義する変数をセットすること。Section “Options Controlling Comments” in *The GNU Emacs Manual* を参照されたい。
- モードは自身の abbrev テーブルをもつことができ、他の関連するモードと構文テーブルを共有することもできる。モードが自身の abbrev テーブルをもつ場合には、*modename*`-mode-abbrev-table`という名前の変数にそれを格納すること。メジャーモードコマンドが自身で何らかの abbrev を定義する場合には、`define-abbrev`の *system-flag*引数に `t`を渡すこと。Section 36.2 [Defining Abbrevs], page 923 を参照されたい。
- モードは変数`font-lock-defaults`にバッファーローカルな値をセットすることによって、Font Lock モードにたいしてハイライトする方法を指定すること (Section 23.6 [Font Lock Mode], page 496 を参照)。
- モードが定義するすべてのフェイスは、もし可能なら既存の Emacs フェイスを継承すること。Section 39.12.8 [Basic Faces], page 1024 と Section 23.6.7 [Faces for Font Lock], page 504 を参照されたい。
- モードは変数 `imenu-generic-expression`、`imenu-prev-index-position-function`と `imenu-extract-index-name-function`の 2 つの変数、または変数 `imenu-create-index-function`にバッファーローカルな値をセットすることによって Imenu がバッファー内の定義やセクションを探す方法を指定すること (Section 23.5 [Imenu], page 494 を参照)。
- モードは `eldoc-documentation-function`にローカル値を指定して、ElDoc モードがそのモードを処理する方法を指定できる。
- モードはスペシャルフック `completion-at-point-functions`に 1 つ以上のバッファーローカルエントリーを追加することにより、さまざまなキーワードの補完方法を指定できる。Section 20.6.8 [Completion in Buffers], page 365 を参照のこと。
- Emacs のカスタマイズ変数にたいしてバッファーローカルなバインディングを作成するには、`make-variable-buffer-local`ではなくメジャーモードコマンド内で `make-local-variable`を使用すること。関数 `make-variable-buffer-local`はそれ以降にカスタマイズ

変数をセットするすべてのバッファーにたいしてその変数をローカルにして、そのモードを使用しないバッファーにたいしても影響があるだろう。そのようなグローバルな効果はモードにとって好ましくない。Section 12.11 [Buffer-Local Variables], page 190 を参照のこと。

稀な例外として Lisp パッケージ内で `make-variable-buffer-local`を使用する唯一の正当な方法は、そのパッケージ内でのみ使用される変数にたいして使用をする場合である。他のパッケージにより使用される変数にたいしてこの関数を使用すると競合が発生するだろう。

- すべてのメジャーモードは *`modename`*`-mode-hook`という名前のノーマルなモードフック *(mode hook)* をもつこと。メジャーモードコマンドが一番最後に `run-mode-hooks`を呼び出すこと。これはノーマルフック `change-major-mode-after-body-hook`、モードフック、(バッファーがファイルを visit していれば) 関数 `hack-local-variables`、その後にノーマルフック `after-change-major-mode-hook`を実行する。Section 23.2.6 [Mode Hooks], page 473 を参照のこと。
- メジャーモードコマンドは親モード *(parent mode)* と呼ばれる他のいくつかのメジャーモードを呼び出すことにより開始されるかもしれず、それらのセッティングのいくつかを変更するかもしれない。これを行うモードは派生モード *(derived mode)* と呼ばれる。派生モードを定義する推奨方法は `define-derived-mode`マクロの使用だが必須ではない。そのようなモードは `delay-mode-hooks`フォーム内で親のモードコマンドを呼び出すこと (`define-derived-mode`は自動的にこれを行う)。Section 23.2.4 [Derived Modes], page 470 と Section 23.2.6 [Mode Hooks], page 473 を参照されたい。
- ユーザーがそのモードのバッファーから他のモードのバッファーに切り替える際に特別な何かを行う必要がある場合、モードは `change-major-mode-hook`にたいしてバッファーローカル値をセットアップできる (Section 12.11.2 [Creating Buffer-Local], page 192 を参照)。
- そのモードが、(ユーザーがキーボードでタイプしたテキストや外部ファイルのテキストではなく) モード自身が生成する特別に用意されたテキストにたいしてのみ適している場合、メジャーモードコマンドのシンボルは以下のように `mode-class`という名前のプロパティに値 `special`を put すること:

  ```
  (put 'funny-mode 'mode-class 'special)
  ```

  これは Emacs にたいしてカレントバッファーが Funny モードのときに新たなバッファーを作成したとき、たとえ `major-mode`のデフォルト値が `nil`であってもそのバッファーを Funny モードにしないよう指示する。デフォルトでは `major-mode`にたいする値 `nil`は新たなバッファー作成時にカレントバッファーのメジャーモードを使用することを意味するが (Section 23.2.2 [Auto Major Mode], page 468 を参照)、`special`なモードにたいしてはかわりに Fundamental モードが使用される。Dired、Rmail、Buffer List のようなモードはこの機能を使用する。

  関数 `view-buffer`は mode-class が special であるようなバッファーでは View モードを有効にしない。そのようなモードは通常は自身で View に相当するバインディングを提供するからである。

  `define-derived-mode`マクロは親モードが special なら、自動的に派生モードを special にマークする。親モードで special モードが有用ならそれを継承したモードでも有用だろう。Section 23.2.5 [Basic Major Modes], page 472 を参照のこと。
- 新たなモードを識別可能な特定のファイルにたいするデフォルトとしたければ、そのようなファイル名にたいしてそのモードを選択するために `auto-mode-alist`に要素を追加する。autoload 用にモードコマンドを定義する場合には、`autoload`を呼び出すのと同じファイル内にその要素を追加すること。モードコマンドにたいして autoload cookie を使用する場合はに、その要素を追加するフォームにたいしても autoload cookie を使用できる ([autoload cookie], page 276

を参照)。モードコマンドを autoload しない場合には、モード定義を含むファイル内で要素を追加すれば十分である。

- 悪影響を与えることなく 1 回以上評価されるように、モード定義はファイル内のトップレベルのフォームとして記述すべきである。たとえばすでに値をもつ変数が再初期化されないように、モードに関連した変数をセットするときは`defvar`か`defcustom`を使用する (Section 12.5 [Defining Variables], page 178 を参照)。

23.2.2 Emacs がメジャーモードを選択する方法

Emacs はファイルを visit するとき、ファイル名やファイル自体の内容などの情報を元にそのバッファーにたいするメジャーモードを選択します。またファイルのテキスト内で指定されたローカル変数も処理します。

`normal-mode` **&optional** *find-file* [Command]

この関数はカレントバッファーにたいして適切なメジャーモード、およびバッファーローカル変数のバインディングを設定する。これは`set-auto-mode` (以下参照) を呼び出す。Emacs 26.1 ではもはや`hack-local-variables`を呼び出さずに、メジャーモードの初期化時の`run-mode-hooks`でこれが行われる (Section 23.2.6 [Mode Hooks], page 473 を参照)。

`normal-mode`の *find-file*引数が非`nil`なら、`normal-mode`は`find-file`関数が自身を呼び出したとみなす。この場合、`normal-mode`はそのファイル内の '`-*-`'行、またはファイルの最後にあるローカル変数を処理できる。これを行うかどうかは変数`enable-local-variables`が制御する。ファイルのローカル変数セクションの構文は Section "Local Variables in Files" in *The GNU Emacs Manual* を参照のこと。

インタラクティブに`normal-mode`を実行すると、引数 *find-file*は通常は`nil`である。この場合、`normal-mode`は無条件に任意のファイルローカル変数を処理する。

この関数はメジャーモードを選択してセットするために`set-auto-mode`を呼び出す。この関数がモードを特定しなければバッファーの`major-mode` (以下参照) のデフォルト値により決定されるメジャーモードに留まる。

`normal-mode`はメジャーモードコマンド呼び出しの周囲に`condition-case`を使用するのでエラーは catch されて、'`File mode specification error`'とともに元のエラーメッセージがその後に報告される。

`set-auto-mode` **&optional** *keep-mode-if-same* [Function]

この関数はカレントバッファーにたいして適切なメジャーモードを選択してセットする。この選択は関数自身の (優先順位による) 決定にもとづく。優先順位は '`-*-`' 行、ファイル終端近傍の任意の '`mode:`'ローカル変数、'`#!`' 行 (`interpreter-mode-alist`を使用)、バッファーの先頭のテキスト (`magic-mode-alist`を使用)、最後が visit されるファイル名 (`auto-mode-alist`を使用) の順。Section "How Major Modes are Chosen" in *The GNU Emacs Manual* を参照のこと。`enable-local-variables`が`nil`なら`set-auto-mode`は '`-*-`' 行とファイル終端近傍にたいして mode タグのチェックを何も行わない。

モード特定のためにファイル内容をスキャンするのがふさわしくないファイルタイプがいくつかある。たとえば tar アーカイブファイルの終端付近に特定のファイルにたいしてモードを指定するローカル変数セクションをもつアーカイブメンバーファイルがたまたま含まれているかもしれない。こがそのファイルを含んだ tar ファイルに適用されるべきではないだろう。同様に tiff イメージファイルが '`-*-`' パターンにマッチするように見える行を最初の行に偶然含むかもしれない。これらの理由により、これらのファイル拡張子はいずれも`inhibit-local-variables-regexps`リストのメンバーになっている。Emacs が、(モード指定に限らず) ファ

イルから任意の種類のローカル変数を検索することを防ぐには、このリストにパターンを追加する。

*keep-mode-if-same*が非 `nil`なら、すでにそのバッファーが適切なメジャーモードをもつときにこの関数はモードコマンドを呼び出さない。たとえば`set-visited-file-name`はユーザーがセットしたかもしれないバッファーローカル変数を kill することを防ぐために、これを `t`にセットする。

`set-buffer-major-mode` *buffer* [Function]

この関数は *buffer*のメジャーモードを`major-mode`のデフォルト値にセットする。`major-mode`が`nil`なら、(それが適切なら) カレントバッファーのメジャーモードを使用する。例外として *buffer*の名前が`*scratch*`なら、モードを `initial-major-mode`にセットする。

バッファーを作成する低レベルのプリミティブはこの関数を使用しないが、`switch-to-buffer`や `find-file-noselect`のような中位レベルのコマンドは、バッファー作成時は常にこの関数を使用する。

`initial-major-mode` [User Option]

この変数の値は`*scratch*`バッファーの初期のメジャーモードを決定する。値はメジャーモードコマンドであるようなシンボルであること。デフォルト値は `lisp-interaction-mode`。

`interpreter-mode-alist` [Variable]

この変数は‘`#!`’行内のコマンドインタープリターを指定するスクリプトにたいして使用するメジャーモードを指定する。変数の値は `(`*`regexp . mode`*`)`という形式の要素をもつ alist である。これはそのファイルが`\\``*regexp*`\\'`にマッチするインタープリターを指定する場合には *mode*を使用することを意味する。たとえばデフォルト要素の 1 つは `("python[0-9.]*" . python-mode)`である。

`magic-mode-alist` [Variable]

この変数の値は `(`*`regexp function`*`)`という形式の要素をもつ alist である。ここで *regexp*は正規表現、*function*は関数、または `nil`である。ファイルを visit した後にバッファーの先頭のテキストが *regexp*にマッチした場合、*function*が非 `nil`なら `set-auto-mode`は *function*を呼び出す。*function*が `nil`なら `auto-mode-alist`がモードを決定する。

`magic-fallback-mode-alist` [Variable]

これは`magic-mode-alist`と同様に機能するが、そのファイルにたいして `auto-mode-alist`がモードを指定しない場合だけ処理される点が異なる。

`auto-mode-alist` [Variable]

この変数はファイル名パターン (正規表現) と対応するメジャーモードコマンドの連想配列を含む。ファイル名パターンは通常は‘`.el`’や‘`.c`’のようなサフィックスをテストするが必須ではない。この alist の通常の要素は `(`*`regexp . mode-function`*`)`のようになる。

たとえば、

```
((\\`/tmp/fol/" . text-mode)
 ("\\.texinfo\\'" . texinfo-mode)
 ("\\.texi\\'" . texinfo-mode)
 ("\\.el\\'" . emacs-lisp-mode)
 ("\\.c\\'" . c-mode)
 ("\\.h\\'" . c-mode)
 ...)
```

バージョン番号とバックアップ用サフィックスをもつファイルを visit したとき、それらのサフィックスは `file-name-sans-versions` (Section 25.9.1 [File Name Components], page 555

を参照) を使用して展開されたファイル名 (Section 25.9.4 [File Name Expansion], page 559 を参照) から取り除かれて *regexp*とマッチされて、`set-auto-mode`はそれに対応する *mode-function*を呼び出す。この機能によりほとんどのファイルにたいして Emacs が適切なメジャーモードを選択することが可能になる。

`auto-mode-alist`の要素が `(`*regexp function* `t)`という形式なら、*function*を呼び出した後に Emacs は前回マッチしなかったファイル名部分にたいしてマッチするために再度`auto-mode-alist`を検索する。この機能は圧縮されたパッケージにたいして有用である。`("\\.gz\\'"` *function* `t)`という形式のエントリーは、ファイルを解凍してから '`.gz`'抜きのファイル名の解凍されたファイルを適切なモードに置く。

以下は`auto-mode-alist`の先頭に複数のパターンペアーを追加する方法の例である (あなたは init ファイル内でこの種の式を使ったことがあるかもしれない)。

```
(setq auto-mode-alist
  (append
   ;; ドットで始まる (ディレクトリー名付きの) ファイル名
   '(("/\\.[^/]*\\'" . fundamental-mode)
     ;; ドットのないファイル名
     ("/[^\\./]*\\'" . fundamental-mode)
     ;; '.c'で終わるファイル名
     ("\\.C\\'" . c++-mode))
   auto-mode-alist))
```

23.2.3 メジャーモードでのヘルプ入手

`describe-mode`関数はメジャーモードに関する情報を提供します。これは通常は *C-h m*にバインドされています。この関数は変数 `major-mode` (Section 23.2 [Major Modes], page 463 を参照) の値を使用します。すべてのメジャーモードがこの変数をセットする必要があるのはこれが理由です。

`describe-mode` **&optional** *buffer* [Command]

このコマンドはカレントバッファーのメジャーモードとマイナーモードのドキュメントを表示する。この関数はメジャーモードおよびマイナーモードのコマンドのドキュメント文字列を取得するために`documentation`関数を使用する (Section 24.2 [Accessing Documentation], page 520 を参照)。

*buffer*引数に非`nil`を指定して Lisp から呼び出されると、この関数はカレントバッファーではなくそのバッファーのメジャーモードとマイナーモードのドキュメントを表示する。

23.2.4 派生モードの定義

新しいメジャーモードを定義する推奨方法は、`define-derived-mode`を使用して既存のメジャーモードから派生させる方法です。それほど近いモードが存在しない場合は`text-mode`、`special-mode`、または`prog-mode`から継承するべきです。Section 23.2.5 [Basic Major Modes], page 472 を参照してください。これらがいずれも適切でなければ、`fundamental-mode`から継承することができます (Section 23.2 [Major Modes], page 463 を参照)。

`define-derived-mode` *variant parent name docstring keyword-args. . . body. . .* [Macro]

このマクロは *variant*をメジャーモードコマンドとして定義して、*name*をモード名の文字列形式とする。*variant*と *parent*はクォートされていないシンボルであること。

新たなコマンド *variant*は関数 *parent*を呼び出すよう定義されて、その後その親モードの特定の性質をオーバーライドする。

- 新たなモードは *variant*-mapという名前の、自身の sparse キーマップ (疎キーマップ) をもつ。define-derived-modeは *variant*-mapがすでにセットされていて、かつすでに親をもつ場合を除いて親モードのキーマップを新たなマップの親キーマップにする。
- 新たなモードは自身の構文テーブル (syntax table) をもち、それは変数 *variant*-syntax-tableに保持される。ただし:syntax-tableキーワード (以下参照) を使用してこれをオーバーライドした場合は異なる。define-derived-modeは *variant*-syntax-tableがすでにセットされていて、かつ標準的な構文テーブルよ異なる親をもつ場合を除いて、親モードの構文テーブルを *variant*-syntax-tableの親とする。
- 新たなモードは自身の abbrev テーブル (略語テーブル) をもち、それは変数 *variant*-abbrev-tableに保持される。ただし:abbrev-tableキーワード (以下参照) を使用してこれをオーバーライドした場合は異なる。
- 新たなモードは自身のモードフック *variant*-hookをもつ。これはフックを実行した後に:after-hookがあればそれを実行して、それとは別に最後に run-mode-hooksによって自身の祖先のモードのフックを実行する。

これらに加えて *body*で *parent*のその他の性質をオーバーライドする方法を指定できます。コマンド *variant*は通常のオーバーライドをセットアップした後、そのモードのフックを実行する直前に *body*内のフォームを評価します。

*parent*が非 nilの mode-classシンボルプロパティをもつ場合、define-derived-modeは *variant*の mode-classプロパティに同じ値をセットします。これはたとえば *parent*が special モードなら *variant*も special モードになることを保証します (Section 23.2.1 [Major Mode Conventions], page 464 を参照)。

*parent*にたいして nilを指定することもできます。これにより新たなモードは親をもたなくなります。その後に define-derived-modeは上述のように振る舞いますが、当然 *parent*につながるすべてのアクションは省略されます。

引数 *docstring*は新たなモードにたいするドキュメント文字列を指定します。define-derived-modeはこのドキュメント文字列の最後にそのモードフックに関する一般的な情報と、その後にそのモードのキーマップを追加します。*docstring*を省略すると define-derived-modeがドキュメント文字列を生成します。

*keyword-args*はキーワードと値のペアー。:after-hookのものを除いて値は評価される。現在のところ以下のキーワードがサポートされる:

`:syntax-table`
: 新たなモードにたいする構文テーブルを明示的に指定するためにこれを使用できる。nil値を指定すると新たなモードは *parent*と同じ構文テーブル、*parent*も nilなら標準的な構文テーブルを使用する (これは nil値の非キーワード引数は引数を指定しないのと同じという通常の慣習にはしたがわないことに注意)。

`:abbrev-table`
: 新たなモードにたいする abbrev テーブルを明示的に指定するためにこれを使用できる。nil値を指定すると新たなモードは *parent*と同じ abbrev テーブル、*parent*も nilなら fundamental-mode-abbrev-tableを使用する (繰り返すが nil値はこのキーワードを指定しないことではない)。

`:group`
: これが指定する場合、値はそのモードにたいするカスタマイズグループ (customization group) であること (すべてのメジャーモードがカスタマイズグループをもつ訳ではない)。customize-modeコマンドはこれを使用する。

`define-derived-mode`は指定されたカスタマイズグループを自動的に定義しない。

`:after-hook`
: このオプションのｋｅｙはモードフック実行後にモード関数の最後の活動として評価される単一の Lisp フォームを指定する。クォートしないこと。モードにが終了した後にフォーが評価されるので、モード関数のローカル状態のすべての要素にアクセスするべきではない。`:after-hook`フォームはモードフックで変更されているかもしれないユーザーのセッティングに依存するモードの様相をセットアップするために有用。

以下は架空の例:

```
(defvar hypertext-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map [down-mouse-3] 'do-hyper-link)
    map))

(define-derived-mode hypertext-mode
  text-mode "Hypertext"
  "ハイパーテキスト用のメジャーモード"
  (setq-local case-fold-search nil))
```

`define-derived-mode`が自動的に行うので、この定義内に`interactive`指定を記述してはならない。

`derived-mode-p` **&rest** *modes* [Function]
: この関数はカレントメジャーモードがシンボル *modes*で与えられたメジャーモードのいずれかから派生されていたら非`nil`をリターンする。

23.2.5 基本的なメジャーモード

Fundamental モードは別として他のメジャーモードの一般的な派生元となるメジャーモードが 3 つあります。それは Text モード、Prog モード、および Special モードです。Text モードはその本来もつ機能から有用なモードです (たとえば`.txt`ファイルの編集など)。一方、Prog モードと Special モードは主にそのようなモード以外のモードの派生元とするために存在します。

新たなモードは直接と間接を問わず、可能な限りそれら 3 つのモードから派生させるべきです。その理由の 1 つは関連のあるモードファミリー全体 (たとえばすべてのプログラミング言語のモード) にたいして、ユーザーが単一のモードフックをカスタマイズできるからからです。

`text-mode` [Command]
: Text モードは人間の言語を編集するためのメジャーモードである。このモードは文字 '`"`'と '`\`' を区切り文字構文 (punctuation syntax: Section 35.2.1 [Syntax Class Table], page 907 を参照) としてもち、*`M-TAB`*を`ispell-complete-word`にバインドする (Section “Spelling” in *The GNU Emacs Manual* を参照)。

Text モードから派生されたメジャーモードの例として HTML モードがある。Section “SGML and HTML Modes” in *The GNU Emacs Manual* を参照のこと。

`prog-mode` [Command]
: Prog モードはプログラミング言語のソースコードを含むバッファーにたいする基本的なメジャーモードである。Emacs ビルトインのプログラミング言語用メジャーモードはこのモードから派生されている。

Prog モードは `parse-sexp-ignore-comments`を `t` (Section 35.6.1 [Motion via Parsing], page 914 を参照)、`bidi-paragraph-direction`を `left-to-right` (Section 39.26 [Bidirectional Display], page 1085 を参照) にバインドする。

`special-mode` [Command]

Special モードはファイルから直接ではなく、Emacs により特別 (specially) に生成されたテキストを含むバッファーにたいする基本的なメジャーモードである。Special モードから派生されたメジャーモードは`mode-class`プロパティに `special`が与えられる (Section 23.2.1 [Major Mode Conventions], page 464 を参照)。

Special モードはバッファーを読み取り専用にセットする。このモードのキーマップはいくつかの一般的なバインディングを定義して、それには`quit-window`にたいする *q*、`revert-buffer` (Section 26.3 [Reverting], page 586 を参照) にたいする *g*が含まれる。

Special から派生されたメジャーモードの例としては Buffer Menu モードがあり、これは `*Buffer List*`バッファーにより使用される。Section “Listing Existing Buffers” in *The GNU Emacs Manual* を参照のこと。

これらに加えて表形式データのバッファーにたいするモードを Tabulated List モードから継承できます。このモードは Special モードから順に派生されているモードです。Section 23.2.7 [Tabulated List Mode], page 474 を参照してください。

23.2.6 モードフック

すべてのメジャーモードコマンドはモード独自のノーマルフック `change-major-mode-after-body-hook`、そのモードのモードフック、ノーマルフック `after-change-major-mode-hook`を実行することによって終了すべきです。これは `run-mode-hooks`を呼び出すことにより行われます。もしそのモードが派生モードなら自身の body 内で他のメジャーモード (親モード) を呼び出す場合には、親モードが自身でこれらのフックを実行しないように `delay-mode-hooks`の中でこれを行うべきです。かわりに派生モードは親のモードフックも実行する `run-mode-hooks`を呼び出します。Section 23.2.1 [Major Mode Conventions], page 464 を参照してください。

Emacs 22 より前のバージョンの Emacs には `delay-mode-hooks`がありません。また Emacs 24 より前のバージョンには `change-major-mode-after-body-hook`がありません。ユーザー実装のメジャーモードが `run-mode-hooks`を使用せず、これらの新しい機能を使用するようにアップデートされていないときは、これらのメジャーモードは以下の慣習に完全にしたがわないでしょう。それらのモードは親のモードフックをあまりに早く実行したり、`after-change-major-mode-hook`の実行に失敗するかもしれません。そのようなメジャーモードに遭遇したら以下の慣習にしたがって修正をお願いします。

`define-derived-mode`を使用してメジャーモードを定義したときは、自動的にこれらの慣習にしたがうことが保証されます。`define-derived-mode`を使用せずにメジャーモードを “手動” で定義したら、これらの慣習を自動的に処理するように以下の関数を使用してください。

`run-mode-hooks` **&rest** *hookvars* [Function]

メジャーモードはこの関数を使用してモードフックを実行すること。これは `run-hooks` (Section 23.1 [Hooks], page 461 を参照) と似ているが `change-major-mode-after-body-hook`、(バッファーがファイルを visit していれば)`hack-local-variables` (Section 12.12 [File Local Variables], page 197 を参照)、`after-change-major-mode-hook`も実行する。これは最後に親モード (Section 23.2.4 [Derived Modes], page 470 を参照) で宣言されている`:after-hook`フォームをすべて評価する。

この関数が delay-mode-hooksフォーム実行中に呼び出されたときはフックや hack-local-variablesの実行、およびフォームの評価を即座には行わない。かわりに次の run-mode-hooks呼び出しでそれらを実行するようにアレンジする。

delay-mode-hooks *body...* [Macro]
あるメジャーモードコマンドが他のメジャーモードコマンドを呼び出すときは delay-mode-hooksの内部で行うこと。

このマクロは *body*を実行するが、*body*実行中はすべての run-mode-hooks呼び出しにたいしてそれらのフックの実行を遅延するよう指示する。それらのフックは実際には delay-mode-hooks構造の最後の後、次の run-mode-hooks呼び出しの間に実行されるだろう。

change-major-mode-after-body-hook [Variable]
これは run-mode-hooksにより実行されるノーマルフックである。これはそのモードのフックの前に実行される。

after-change-major-mode-hook [Variable]
これは run-mode-hooksにより実行されるノーマルフックである。これはすべての適切に記述されたメジャーモードコマンドの一番最後に実行される。

23.2.7 Tabulated List モード

Tabulated List モードとは、表形式データ (エントリーから構成されるデータで各エントリーはそれぞれテキストの 1 行を占め、エントリーの内容は列に分割されるようなデータ) を表示するためのメジャーモードです。Tabulated List モードは行列の見栄えよくプリントする機能、および各列の値に応じて行をソートする機能を提供します。これは Special モードから派生されたモードです (Section 23.2.5 [Basic Major Modes], page 472 を参照)。

Tabulated List モードは、より特化したメジャーモードの親モードとして使用されることを意図しています。例としては Process Menu モード (Section 38.6 [Process Information], page 947 を参照)、Package Menu モード (Section “Package Menu” in *The GNU Emacs Manual* を参照) が含まれます。

このような派生されたモードは tabulated-list-modeを 2 つ目の引数に指定して、通常の方法で define-derived-modeを使用するべきです (Section 23.2.4 [Derived Modes], page 470 を参照)。define-derived-modeフォームの body は以下にドキュメントされている変数に値を割り当てることにより、表形式データのフォーマットを指定するべきです。その後にオプションで列名のヘッダーを挿入する関数 tabulated-list-init-headerを呼び出すことができます。

派生されたモードはリスティングコマンド *(listing command)* も定義するべきです。これはモードコマンドではなく、(*M-x list-processes*のように) ユーザーが呼び出すコマンドです。リスティングコマンドはバッファーを作成または切り替えて、派生モードをオンにして表形式データを指定し、最後にそのバッファーを事前設定 (populate) するために tabulated-list-printを呼び出すべきです。

tabulated-list-gui-sort-indicator-asc [User Option]
この変数は GUI フレームにおいて列が昇順でソートされていることを示すために使用する文字を指定する。

Tabulated List バッファーでソート方向を変更するたびに、このインジケーターの昇順 (“asc”) と降順 (“desc”) が切り替わる。

`tabulated-list-gui-sort-indicator-desc` [User Option]
`tabulated-list-gui-sort-indicator-asc`と同様だが列が降順でソートされている際に使用される。

`tabulated-list-tty-sort-indicator-asc` [User Option]
`tabulated-list-gui-sort-indicator-asc`と同様だがテキストモードのフレームに使用される。

`tabulated-list-tty-sort-indicator-desc` [User Option]
`tabulated-list-tty-sort-indicator-asc`と同様だが列が降順でソートされている際に使用される。

`tabulated-list-format` [Variable]
このバッファーローカル変数は表形式データのフォーマットを指定する。値はベクターであり、ベクターの各要素はデータ列を表すリスト`(`*`name width sort`*`)`である。ここで

- *name*は列の名前 (文字列)。
- *width*は列にたいして予約される文字数幅 (整数)。最終列は各行の終端までなので意味がない。
- *sort*は列によりエントリーをソートする方法を指定する。`nil`ならその列はソートに使用できない。`t`なら列の文字列値を比較することによりソートされる。それ以外なら`tabulated-list-entries`の要素と同じ形式の 2 つの引数をとる、`sort`にたいする述語関数 (predicate function) であること。

`tabulated-list-entries` [Variable]
このバッファーローカル変数は Tabulated List バッファー内に表示されるエントリーを指定する。値はリストか関数のいずれかであること。

値がリストなら各リスト要素は 1 つのエントリーに対応し、`(`*`id contents`*`)` という形式であること。ここで

- *id*は`nil`、またはエントリーを識別する Lisp オブジェクト。Lisp オブジェクトならエントリーを再ソートした際、カーソルは同じエントリー上に留まる。比較は`equal`で行われる。
- *contents*は`tabulated-list-format`と要素数が同じベクター。ベクター要素は文字列かリスト。文字列ならバッファーにそのまま挿入される。リスト`(`*`label . properties`*`)`なら、*label*と*properties*を引数として`insert-text-button`を呼び出すことによってテキストボタンを挿入することを意味する (Section 39.19.3 [Making Buttons], page 1069 を参照)。

  これらの文字列には改行を含めないこと。

それ以外なら、それは値は引数なしで呼び出されて上記形式のリストをリターンする関数であること。

`tabulated-list-revert-hook` [Variable]
このノーマルフックは Tabulated List バッファーのリバートに先立ち実行される。派生モードは`tabulated-list-entries`を再計算するためにこのフックに関数を追加できる。

`tabulated-list-printer` [Variable]
この変数の値はポイント位置にエントリー (エントリーを終端する改行を含む) を挿入するために呼び出される関数である。この関数は`tabulated-list-entries`と同じ意味をもつ 2 つの

引数 *id*と *contents*を受け取る。デフォルト値はエントリーをそのまま挿入する関数である。より複雑な方法で Tabulated List モードを使用するモードは別の関数を指定できる。

`tabulated-list-sort-key` [Variable]
この変数の値は Tabulated List バッファーにたいするカレントのソートキーを指定する。`nil` ならソートは行われない。それ以外なら (*name* . *flip*)という形式の値をもつ。ここで *name* は `tabulated-list-format`内の列目の 1 つとマッチする文字列、*flip*が非 `nil`なら逆順でのソートを意味する。

`tabulated-list-init-header` [Function]
この関数は Tabulated List バッファーにたいする `header-line-format`を計算してセットし、列ヘッダー上でのクリックでソートを可能にするキーマップをヘッダー行に割り当てる。

Tabulated List から派生したモードは、上記の変数 (特に `tabulated-list-format`をセットした後のみ) をセットした後にこれを呼び出すこと。

`tabulated-list-print` **&optional** *remember-pos update* [Function]
この関数はカレントバッファーにエントリーを挿入する。これをリスティングコマンドとして呼び出すこと。この関数はバッファーを消去して `tabulated-list-entries`で指定されるエントリーを `tabulated-list-sort-key`にしたがってソートした後、各エントリーを挿入するために `tabulated-list-printer`で指定される関数を呼び出す。

オプション引数 *remember-pos*が非 `nil`なら、この関数はカレント行で *id*要素を探して、もしあればすべてのエントリーを (再) 挿入して、その後にそのエントリーの移動を試みる。

オプション引数 *update*が非 `nil`なら、この関数は最後のプリント以降に変更されたエントリーの削除か追加だけを行う。この関数が最後に呼び出されて以降、ほとんどのエントリーが変更されていなければ、この関数は数倍高速になる。結果の違いは `tabulated-list-put-tag`を通じて配置されたタグが変更されていないエントリーから削除されないことだけである (通常はすべてのタグが削除される)。

`tabulated-list-delete-entry` [Function]
この関数はポイント位置のエントリーを削除する。

リスト (*id cols*)をリターンする。ここで *id*は削除したエントリーの ID、*cols*は列修飾子 (column descriptors) のベクター。カレント行の先頭にポイントを移動する。ポイント位置にエントリーがなければ `nil`をリターンする。

この関数はバッファーのコンテンツだけを変更することに注意。`tabulated-list-entries` は変更しない。

`tabulated-list-get-id` **&optional** *pos* [Function]
この `defsubst`は `tabulated-list-entries`がリストなら ID オブジェクト、関数なら `tabulated-list-entries`がリターンするリストから ID オブジェクトをリターンする。*pos*が省略か `nil`の場合のデフォルトはポイント位置。

`tabulated-list-get-entry` **&optional** *pos* [Function]
この `defsubst`は `tabulated-list-entries`がリストならエントリーオブジェクト、関数なら `tabulated-list-entries`がリターンするリストからエントリーオブジェクトをリターンする。これは *pos*にある ID にたいするベクターになるだろう。*pos*にエントリーがなければ、この関数は `nil`をリターンする。

`tabulated-list-header-overlay-p` **&optional** *POS* [Function]
この `defsubst`は *pos*に偽ヘッダーがあれば非 `nil`をリターンする。偽ヘッダー (fake header) はバッファー先頭に列名を配置するために `tabulated-list-use-header-line`が `nil`にセットされている場合に使用される。*pos*が省略か `nil`の場合のデフォルトは `point-min`。

`tabulated-list-put-tag` *tag* **&optional** *advance* [Function]
この関数はレント行のパディングエリアに *tag*を配置する。パディングエリアはその行の先頭にある空スペースであり、幅は `tabulated-list-padding`により制御される。*tag*は長さが `tabulated-list-padding`以下の文字列であること。*advance*が非 `nil`なら、この関数は 1 行分ポイントを前方に移動する。

`tabulated-list-clear-all-tags` [Function]
この関数はカレントバッファーのパディングエリアからすべてのタグをクリアーする。

`tabulated-list-set-col` *col desc* **&optional** *change-entry-data* [Function]
この関数は *col*を *desc*にセットしてポイント位置にある Tabulated List のエントリーを変更する。*col*は変更する列番号か列名、*desc*は新たな列記述子であり、`tabulated-list-print-col`を通じて挿入される。

*change-entry-data*が非 `nil`なら、この関数は列記述子のベクターを `desc`にセットすることにより、背後のデータ (通常はリスト `tabulated-list-entries`内の列記述子) を変更する。

23.2.8 ジェネリックモード

generic モード *(*汎用モード*)* とは、コメント構文にたいする基本的なサポートと Font Lock モードをもつシンプルなメジャーモードです。generic モードを定義するにはマクロ `define-generic-mode`を使用します。`define-generic-mode`の使い方の例は、ファイル `generic-x.el`を参照してください。

`define-generic-mode` *mode comment-list keyword-list font-lock-list* [Macro]
auto-mode-list function-list **&optional** *docstring*
このマクロは *mode* (クォートされていないシンボル) という名前の generic モードコマンドを定義する。オプション引数 *docstring*は、そのモードコマンドにたいするドキュメント文字列。これを与えなければ `define-generic-mode`がデフォルトのドキュメント文字列を生成する。

引数 *comment-list*は要素が文字、2 文字以下の文字列、またはコンスセルである。文字か文字列ならそのモードの構文テーブル内でコメント開始識別子としてセットアップされる。エントリーがコンスセルなら CAR はコメント開始識別子、CDR はコメント終了識別子としてセットアップされる (行末によりコメントを終端させたければ後者に `nil`を使用する)。構文テーブルのメカニズムには実際にコメントの開始および終了識別子に関する制限があることに注意。Chapter 35 [Syntax Tables], page 906 を参照のこと。

引数 *keyword-list*は `font-lock-keyword-face`でハイライトするキーワードのリストである。キーワードは文字列であること。一方、*font-lock-list*はハイライトするための追加の式リストである。このリストの各要素は `font-lock-keywords`の要素と同じ形式をもつこと。Section 23.6.2 [Search-based Fontification], page 498 を参照されたい。

引数 *auto-mode-list*は変数 `auto-mode-alist`に追加する正規表現のリストである。これらのは、マクロ呼び出しの展開時ではなく、`define-generic-mode`の実行時に追加される。

最後に *function-list*は追加セットアップのためにモードコマンドに呼び出される関数のリストである。これらの関数はモードフック変数 *mode*`-hook`の実行の直前に呼び出される。

23.2.9 メジャーモードの例

おそらく Text モードは、Fundamental を除いてもっともシンプルなモードです。上述した慣習の多くを説明するために以下に `text-mode.el`の抜粋を示します:

```
;; Create the syntax table for this mode.
(defvar text-mode-syntax-table
  (let ((st (make-syntax-table)))
    (modify-syntax-entry ?\" ".   " st)
    (modify-syntax-entry ?\\ ".   " st)
    ;; Add 'p' so M-c on 'hello' leads to 'Hello', not 'hello'.
    (modify-syntax-entry ?' "w p" st)
    ...
    st)
  "Syntax table used while in `text-mode'.")

;; このモード用にキーマップを作成
(defvar text-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\e\t" 'ispell-complete-word)
    ...
    map)
  "Keymap for `text-mode'.
Many other modes, such as `mail-mode', `outline-mode' and
`indented-text-mode', inherit all the commands defined in this map.")
```

そして実際にモードコマンドが定義される方法が以下になります:

```
(define-derived-mode text-mode nil "Text"
  "人間が読むために記述されたテキストを編集するためのメジャーモード
このモードではパラグラフを区切るのはブランク行か空白行だけである
したがって適応型フィル (adaptive filling) の全恩恵を受けられる
 (変数`adaptive-fill-mode'を参照のこと)
\\{text-mode-map}
Text モードのオンによりノーマルフック`text-mode-hook'が実行される"
  (setq-local text-mode-variant t)
  (setq-local require-final-newline mode-require-final-newline))
```

3 つの Lisp 用モード (Lisp モード、Emacs Lisp モード、Lisp Interaction モード) は Text モードより多くの機能をもち、それにふさわしくコードもより複雑です。そのようなモードの記述方法を説明するために `lisp-mode.el`の抜粋を示します。

以下は Lisp モードの構文テーブルと abbrev テーブルを定義する方法です:

```
;; モード固有のテーブル変数の作成
(define-abbrev-table 'lisp-mode-abbrev-table ()
  "Abbrev table for Lisp mode.")

(defvar lisp-mode-syntax-table
  (let ((table (make-syntax-table lisp--mode-syntax-table)))
    (modify-syntax-entry ?\[ "_   " table)
    (modify-syntax-entry ?\] "_   " table)
    (modify-syntax-entry ?# "' 14" table)
    (modify-syntax-entry ?| "\" 23bn" table)
    table)
  "`lisp-mode'で使用される構文テーブル")
```

Lisp 用の 3 つのモードは多くのコードを共有します。たとえば以下の関数呼び出しによってさまざまな変数がセットされます:

```
(defun lisp-mode-variables (&optional syntax keywords-case-insensitive elisp)
  (when syntax
    (set-syntax-table lisp-mode-syntax-table))
  ...
```

その中でも特に以下の関数はLispコメントを処理するために変数`comment-start`をセットアップします:

```
  (setq-local comment-start ";")
  ...
```

これらの異なるLisp用モードは、微妙に異なるキーマップをもちます。たとえばLispモードは*C-c C-z*を`run-lisp`にバインドしますが、他のLisp用モードはこれを行いません。とはいえすべてのLisp用モードに共通なコマンドがいくつかあります。以下のコードはそれらの共通コマンドをセットアップします:

```
(defvar lisp-mode-shared-map
  (let ((map (make-sparse-keymap)))
    (set-keymap-parent map prog-mode-map)
    (define-key map "\e\C-q" 'indent-sexp)
    (define-key map "\177" 'backward-delete-char-untabify)
    map)
  "すべてのLisp用モードでコマンドを共有するためのキーマップ")
```

そして以下がLispモードのためのキーマップをセットアップするコードです:

```
(defvar lisp-mode-map
  (let ((map (make-sparse-keymap))
        (menu-map (make-sparse-keymap "Lisp")))
    (set-keymap-parent map lisp-mode-shared-map)
    (define-key map "\e\C-x" 'lisp-eval-defun)
    (define-key map "\C-c\C-z" 'run-lisp)
    ...
    map)
  "Keymap for ordinary Lisp mode.
All commands in `lisp-mode-shared-map' are inherited by this map.")
```

最後はLispモードのためのメジャーモードコマンドです:

```
(define-derived-mode lisp-mode prog-mode "Lisp"
  "GNU Emacs Lisp以外のLispコードを編集するためのメジャーモード
コマンド:
あたかも後方に移動するようにタブをスペースに削除変換する
パラグラフ区切りはブランク行でコメント開始はセミコロン

\\{lisp-mode-map}
`run-lisp'はinferior Lispジョブの開始と既存ジョブ
から戻るための両方に使われるかもしれないことに注意"
  (lisp-mode-variables nil t)
  (setq-local find-tag-default-function 'lisp-find-tag-default)
  (setq-local comment-start-skip
              "\\(\\(^\\|[^\\\\\n]\\)\\(\\\\\\\\)*\\)\\(;+\\|#|\\) *")
  (setq imenu-case-fold-search t))
```

23.3 マイナーモード

マイナーモード *(minor mode)* はメジャーモードの選択とは無関係にユーザーが有効や無効にできるオプション機能の使用を提供します。マイナーモードは個別、あるいは組み合わせて有効にできます。

ほとんどのマイナーモードはメジャーモードとは独立した機能を実装するので、ほとんどのメジャーモードと一緒に使用することができます。たとえばAuto Fillモードはテキスト挿入を許容する任意のメジャーモードとともに機能します。しかし少数ながら特定のメジャーモードに特化したマイナー

モードもあります。たとえばDiff Auto Refineモードは Diffモードとの使用だけを意図したマイナーモードです。

理想的にはマイナーモードは他のマイナーモードの効果と無関係に期待した効果をもつべきです。任意の順序でマイナーモードをアクティブや非アクティブにすることも可能であるべきです。

`minor-mode-list` [Variable]
この変数の値はすべてのマイナーモードコマンドのリスト。

23.3.1 マイナーモード記述の規約

メジャーモードにも監修があるように (Section 23.2 [Major Modes], page 463 を参照)、マイナーモードの記述にも慣習があります。その慣習について以下で説明します。これらの慣習にしたがうにはマクロ `define-minor-mode`を使用するのがもっとも簡単な方法です。Section 23.3.3 [Defining Minor Modes], page 482 を参照してください。

- 名前が '`-mode`'で終わる変数を定義する。これをモード変数 *(mode variable)* と呼ぶ。マイナーモードコマンドはこの変数をセットすること。値はそのモードが無効なら `nil`、有効なら非`nil`になる。そのマイナーモードがバッファーローカルならこの変数もバッファーローカルであること。

  この変数はモードラインにマイナーモードの名前を表示するために `minor-mode-alist`と結合して使用される。これは `minor-mode-map-alist`を通じて、そのマイナーモードのキーマップがアクティブかどうかも判定する (Section 22.9 [Controlling Active Maps], page 432 を参照)。個々のコマンドやフックもこの変数の値をチェックできる。

- モード変数と同じ名前をもつモードコマンド *(mode command)* と呼ばれるコマンドを定義する。このコマンドの役目はモード変数の値のセットに加えて、そのモードの機能を使用を実際に有効や無効にするために必要な他のすべてを行うことである。

  モードコマンドは 1 つのオプション引数を受け入れること。プレフィクス引数なしで interactive に呼び出されたらモードをトグルする (toggle: 切り替える。たとえば無効なら有効に、有効なら無効にする) こと。プレフィクス引数とともに interactive に呼び出された場合にはその引数が正であればモードを有効にして、それ以外なら無効にすること。

  モードコマンドが Lisp から (つまり非 interactive に) 呼び出された場合は、引数が省略または`nil`ならモードを有効にすること。引数がシンボル `toggle`ならモードをトグルして、それ以外なら上述の数引数とともに interactive に呼び出されたときと同じ方法によってその引数を扱うこと。

  以下はこの挙動の実装方法を示す例である (`define-minor-mode`マクロが生成するコードもこれに類似する)。

```
(interactive (list (or current-prefix-arg 'toggle)))
(let ((enable
       (if (eq arg 'toggle)
           (not foo-mode) ; そのモードのモード変数
         (> (prefix-numeric-value arg) 0))))
  (if enable
      do-enable
    do-disable))
```

  やや複雑なこの挙動の理由は、ユーザーが簡単かつ interactive にマイナーモードをトグルできることと、以下のようにモードフック内で簡単にマイナーモードを有効にできるからである:

```
(add-hook 'text-mode-hook 'foo-mode)
```

`foo-mode`モードコマンドは引数なしで Lisp から呼び出されたときは無条件にそのマイナーモードを有効にするので、これは `foo-mode`がすでに有効でもそうでなくても正しく振る舞う。モードフック内でマイナーモードを無効にする場合は少々醜くなる:

```
(add-hook 'text-mode-hook (lambda () (foo-mode -1)))
```

しかしこれは頻繁には行われない。

マイナーモードを 2 回連続で直接有効 (や無効) にしても失敗せずに、1 回だけ有効 (や無効) にしたときと同じことを行うこと。言い換えるとマイナーモードコマンドはべき等であること。

- モードラインにマイナーモードを表示したければ、それぞれのマイナーモードにたいして要素を`minor-mode-alist`に追加する ([Definition of minor-mode-alist], page 490 を参照)。この要素は以下の形式のリストであること:

  ```
  (mode-variable string)
  ```

  ここで *mode-variable*はマイナーモードの有効化を制御する変数、*string*はモードラインに表示するためのスペースで始まる短い文字列である。一度に複数モードの文字列がスペースを占有するので、これらの文字列は短くなければならない。

  `minor-mode-alist`に要素を追加する際は、重複を避けるために既存要素のチェックに `assq`を使用すること。たとえば:

  ```
  (unless (assq 'leif-mode minor-mode-alist)
    (push '(leif-mode " Leif") minor-mode-alist))
  ```

  または以下のように `add-to-list`(Section 5.5 [List Variables], page 79 を参照) を使用すること:

  ```
  (add-to-list 'minor-mode-alist '(leif-mode " Leif"))
  ```

これらに加えてメジャーモードにたいする慣習 (Section 23.2.1 [Major Mode Conventions], page 464 を参照) のいくつかは、マイナーモードにたいしても同様に適用されます。それらの慣習はグローバルシンボルの名前、初期化関数の最後でのフックの使用、キーマップおよびその他のテーブルの使用です。

マイナーモードは、可能なら Custom(Chapter 15 [Customization], page 250 を参照) を通じた有効化と無効化をサポートするべきです。これを行うには、モード変数は`:type 'boolean`とともに `defcustom`で通常は定義されるべきです。その変数をセットするだけではモードの有効化に不足なら、モードコマンドを呼び出すことによりモードを有効にする`:set`メソッドも指定するべきです。そしてその変数のドキュメント文字列に Custom を通じて変数をセットしなければ効果がないことを注記してください。さらにその定義を autoload cookie([autoload cookie], page 276 を参照) でマークして、その変数のカスタマイズによりモードを定義するライブラリーがロードされるように`:require`を指定します。たとえば:

```
;;;###autoload
(defcustom msb-mode nil
  "msb-mode をトグルする
この変数を直接セットしても効果がない
\\[customize] か関数`msb-mode'を使用すること"
  :set 'custom-set-minor-mode
  :initialize 'custom-initialize-default
  :version "20.4"
  :type    'boolean
  :group   'msb
  :require 'msb)
```

23.3.2 キーマップとマイナーモード

マイナーモードはそれぞれ自身のキーマップをもつことができ、そのモードが有効になるとそのキーマップがアクティブになります。マイナーモード用のキーマップをセットアップするには`minor-mode-map-alist`という alist に要素を追加します。[Definition of minor-mode-map-alist], page 433 を参照してください。

特定の自己挿入文字にたいして自己挿入と同様に他の何かを行うように振る舞いを変更するのは、マイナーモードキーマップの 1 つの使い方です。(`self-insert-command`をカスタマイズする別の方法は `post-self-insert-hook`を通じて行う方法。Section 32.5 [Commands for Insertion], page 783 を参照のこと。これ以外の `self-insert-command`カスタマイズ用の機能は特別なケースに限定されており abbrev モードと Auto Fill モードのためにデザインされている。`self-insert-command`の標準定義から独自の定義への置き換えを試みてはならない。エディターコマンドループはこの関数を特別に処理する。)

マイナーモードはコマンドを *`C-c`*とその後の区切り文字よって構成されるキーシーケンスにバインドできます。しかし *`C-c`*とその後の *`{}<>:;`*のいずれかの文字、またはコントロール文字、数字より構成されるシーケンスはメジャーモード用に予約済みです。また *`C-c letter`*はユーザー用に予約済みです。Section D.2 [Key Binding Conventions], page 1164 を参照してください。

23.3.3 マイナーモードの定義

マクロ `define-minor-mode`は、自己完結した単一定義内にモードを実装する便利な方法を提供します。

`define-minor-mode` *mode doc* [*init-value* [*lighter* [*keymap*]]] *keyword-args... body...* [Macro]

このマクロは名前が *mode*(シンボル) の新たなマイナーモードを定義する。これはドキュメント文字列として *doc*をもつマイナーモードをトグルするために *mode*という名前のコマンドを定義する。

トグルコマンドは 1 つのオプション (プレフィクス) 引数を受け取る。引数なしで interactive に呼び出されると、そのモードのオンとオフをトグルする。正のプレフィクス引数はモードを有効にして、それ以外のプレフィクス引数はモードを無効にする。Lisp から呼び出すと引数が `toggle`ならモードをトグルして、引数が省略か `nil`ならモードを有効にする。これはたとえばメジャーモードフック内でマイナーモードを有効にするのを簡便にする。*doc*が `nil`なら、このマクロは上記を記述したデフォルトのドキュメント文字列を提供する。

デフォルトではこれはモードを有効にすると `t`、無効にすると `nil`にセットされる、*mode*という名前の変数も定義する。この変数は *init-value*に初期化される。通常 (以下参照) はこの値は `nil`でなければならない。

文字列 *lighter*はモード有効時にモードライン内に何を表示するか指定する。これが `nil`ならこのモードはモードライン内に表示されない。

オプション引数 *keymap*はそのマイナーモードにたいするキーマップを指定する。非 `nil`なら、それは (値がキーマップであるような) 変数の名前かキーマップ、または以下の形式の alist であること

```
(key-sequence . definition)
```

ここで *key-sequence*と *definition*は `define-key`に渡すのに適した引数である (Section 22.12 [Changing Key Bindings], page 438 を参照)。*keymap*はキーマップか alist であり、これは変数 *`mode`*`-map`も定義する。

上記の 3 つの引数 *init-value*、*lighter*、*keymap*は *keyword-args*が使用されたときは (部分的に) 省略できる。*keyword-args*はキーワードとその後の対応する値により構成され、いくつかのキーワードは特別な意味をもつ:

`:group` *group*
: 生成されるすべての `defcustom`フォームで使用されるカスタムグループ名。*mode* (後に '`-mode`'がある場合はそれを除く) にたいするデフォルトである。警告: そのグループを定義するため `defgroup`を正しく記述していなければ、このデフォルトグループ名を使用してはならない。Section 15.2 [Group Definitions], page 252 を参照のこと。

`:global` *global*
: 非 `nil`ならそのマイナーモードがバッファーローカルでなくグローバルであることを指定する。デフォルトは `nil`。

  マイナーモードをグローバルにしたときの効果の 1 つは、*mode*変数がカスタマイズ変数になることである。Customize インターフェイスを通じてこの変数をトグルするとモードがオンやオフになり、変数の値は将来の Emacs セッション用に保存できるようになる (Section "Saving Customizations" in *The GNU Emacs Manual* を参照)。保存された変数が機能するためには Emacs が開始されるたびに `define-minor-mode`フォームが確実に評価されるようにすること。Emacs の一部ではないパッケージにたいしては、`:require`キーワードを指定するのがこれを行う一番簡単な方法である。

`:init-value` *init-value*
: これは *init-value*引数を指定するのと等しい。

`:lighter` *lighter*
: これは *lighter*引数を指定するのと等しい。

`:keymap` *keymap*
: これは *keymap*引数を指定するのと等しい。

`:variable` *place*
: これはそのモードの状態を格納するために使用されるデフォルトの変数 *mode*を置き換える。これを指定すると *mode*変数は定義されず、すべての *init-value*引数は使用されない。*place*は異なる名前の変数 (あなた自身が定義しなければならない)、または `setf`関数とともに使用され得るすべてのもの (Section 12.17 [Generalized Variables], page 205 を参照)。*place*にはコンス (`(`*get* `.` *set*`)`) も指定できる。ここで *get*はカレント状態をリターンする式であり、*set*はそれをセットする 1 つの引数 (状態) をとる関数である。

`:after-hook` *after-hook*
: これはモードフック実行後に評価される単一の Lisp フォームを定義する。これをクォートしないこと。

その他のすべてのキーワード引数は変数 *mode*にたいして生成された `defcustom`に直接渡される。

*mode*という名前のコマンドは最初に *mode*という名前の変数をセットする等の標準的な動作を処理した後に、もしあれば *body*フォームを実行する。それからモードフック変数 *mode*`-hook` を実行してから`:after-hook`内のフォームを評価して終了する。

*init-value*の値は`nil`でなければなりません。ただし、(1)Emacs によりそのモードが事前ロードされている、または (2) たとえユーザーが要求しなくともモードを有効にするためにロードするのが容易な場合を除きます。たとえば他の何かが有効でなければそのモードの効果がなく、常にそのタイミングでロードされるような場合には、デフォルトでそのモードを有効にすることに害はありません。しかしこの状況は通常はあり得ません。通常は *init-value*の値は`nil`でなければなりません。

`easy-mmode-define-minor-mode`という名前はこのマクロにたいするエイリアスです。

以下は`define-minor-mode`の使い方の例です:

```
(define-minor-mode hungry-mode
  "Hungry モードをトグルする
引数なしで interactive に呼び出すとモードをトグルする
正のプレフィクス引数でモードを有効に、その他のプレフィクス引数で
無効にする。Lisp から呼び出す場合、引数を省略、または nil なら
モードを有効に、`toggle'なら状態をトグルする

Hungry モードが有効なときは、C-DEL キーは、
最後を除く先行するすべての空白を飲み込む
コマンド \\[hungry-electric-delete] を参照"
 ;; 初期値
 nil
 ;; モードラインの標示
 " Hungry"
 ;; マイナーモードのバインディング
 '(([C-backspace] . hungry-electric-delete)))
```

これは “Hungry mode” という名前のマイナーモード、モードをトグルする`hungry-mode`という名前のコマンド、モードが有効かどうかを示す`hungry-mode`という名前の変数、モードが有効なときそのキーマップを保持する`hungry-mode-map`という名前の変数を定義します。これは *C-DEL*にたいするキーバインディングでキーマップを初期化します。*body*フォームはありません— 多くのマイナーモードはそれを必要としません。

以下はこれを記述する等価な方法です:

```
(define-minor-mode hungry-mode
  "Hungry モードをトグルする
... 省略..."
 ;; 初期値
 :init-value nil
 ;; モードラインへのインジケーター
 :lighter " Hungry"
 ;; マイナーモードのバインディング
 :keymap
 '(([C-backspace] . hungry-electric-delete)
   ([C-M-backspace]
    . (lambda ()
        (interactive)
        (hungry-electric-delete t)))))
```

`define-globalized-minor-mode` *global-mode mode turn-on keyword-args... body...* [Macro]

これは *global-mode*という名前をグローバルにトグルする。これは *mode*という名前のバッファーローカルなマイナーモードをすべてのバッファーで有効か無効にするということを意味する。*body*フォームの実行も行う。あるバッファー内でそのマイナーモードをオンにするには関数 *turn-on*を使用する。マイナーモードをオフにするには −1 を引数として *mode*を呼び出す。

モードをグローバルに有効にすると、それ以降ファイルを visit することによって作成されるバッファーや Fundamental 以外のメジャーモードを使用するバッファーにも影響がある。しかし Fundamental で作成される新たなバッファーは検知しない。

これは Customize インターフェイス内でそのマイナーモードのオン/オフを切り替えるカスタムオプション *global-mode* (Chapter 15 [Customization], page 250) を定義する。`define-minor-mode`と同様に、たとえば:`require`を与える等によって Emacs 開始時に毎回確実に `define-globalized-minor-mode`フォームが評価されるようにすること。

グローバルマイナーモードのモード変数にたいしてカスタムグループを指定するには *keyword-args*内で:`group` *group*を使用する。

一般的にはグローバル化されたマイナーモードを定義するときは、ユーザーがバッファーごとにモードを使用 (または無効に) できるように非グローバル版も定義すること。これにより特定のメジャーモード内でそのモードのフックを使用すればグローバルに有効化されたマイナーモードを無効にすることができるようになる。

23.4 モードラインのフォーマット

Emacs の各ウィンドウ (ミニバッファーウィンドウを除く) には、通常は最下部にモードラインがあってそのウィンドウ内に表示されたバッファーに関するステータス情報がモードラインに表示されます。モードラインにはバッファー名、関連するファイル、再帰編集の深さ、およびメジャーモードやマイナーモードなどのようなそのバッファーに関する情報が含まれています。ウィンドウはヘッダーライン *(header line)* をもつこともでき、これはモードラインによく似ていますがウィンドウの最上部に表示されます。

このセクションではモードラインおよびヘッダーラインのコンテンツの制御の仕方について説明します。このチャプターにモードラインを含めた理由は、モードラインに表示される情報の多くが有効化されたメジャーモードとマイナーモードに関連があるからです。

23.4.1 モードラインの基礎

各モードラインのコンテンツはバッファーローカル変数 `mode-line-format`により指定されます (Section 23.4.3 [Mode Line Top], page 487 を参照)。この変数はモードライン構文 *(mode line construct)* を保持します。これはバッファーのモードラインに何を表示するかを制御するテンプレートです。`header-line-format`の値は同じ方法によりバッファーのヘッダーラインを指定します。同一のバッファーにたいするすべてのウィンドウは、`mode-line-format`や `header-line-format`のパラメーター (Section 28.27 [Window Parameters], page 688 を参照) がそのウィンドウに指定されていなければ、同じ `mode-line-format`と `header-line-format`を使用します。

効率的な理由により Emacs は各ウィンドウのモードラインとヘッダーラインを連続で再評価しません。たとえばウィンドウ設定 (window configuration) の変更やバッファーの切り替え、バッファーのナローイング (narrowing) やワイドニング (widening)、スクロールやバッファーの変更等、それを呼び出す状況が出現したときに Emacs は再評価を行います。`mode-line-format`や `header-line-format`(Section 23.4.4 [Mode Line Variables], page 488 を参照) により参照されるすべての変数、またはテキストが表示される方法に影響を与えるデータ構造 (Chapter 39 [Display], page 981 を参照) を変更する場合には、表示を更新するために関数 `force-mode-line-update`を使用するべきです。

`force-mode-line-update` **&optional** *all* [Function]

この関数は次の再表示サイクルの間にすべての関連する変数の最新の値にもとづいて、カレントバッファーのモードラインとヘッダーラインの更新を Emacs に強制する。オプション引数 *all*が非 `nil`なら、すべてのモードラインとヘッダーラインの更新を強制する。

この関数はメニューバーとフレームタイトルの更新も強制する。

選択されたウィンドウのモードラインは、通常はフェイス `mode-line`を使用して異なるカラーで表示されます。かわりに他のウィンドウのモードラインはフェイス `mode-line-inactive`で表示されます。Section 39.12 [Faces], page 1011 を参照してください。

23.4.2 モードラインのデータ構造

モードラインのコンテンツはモードライン構文 *(mode line construct)* と呼ばれるデータ構造によって制御されます。モードライン構文はリストやシンボル、数字を保持するバッファーローカル変数により構成されます。それぞれのデータ型は以下で説明するようにモードラインの外見にたいして特別な意味をもちます。フレームタイトル (Section 29.6 [Frame Titles], page 730 を参照) とヘッダーライン (Section 23.4.7 [Header Lines], page 493 を参照) にも同じデータ構造が使用されます。

固定文字列のようなシンプルなモードライン構文の場合もありますが、通常はモードライン構文のテキストを構築するために固定文字列と変数の値を組み合わせる方法を指定します。これらの変数の多くはその変数自体がその値によりモードライン構文を定義する変数です。

以下はモードライン構文における、さまざまなデータ型の意味です:

`string`
: モードライン構文における文字列は、文字列内に`%`構文 *(`%`-constructs)* を含む以外はそのまま表現される。これらは他のデータによる置換を意味する。Section 23.4.5 [%-Constructs], page 491 を参照のこと。

  文字列の一部が `face`プロパティをもつ場合には、バッファー内でそれらが表示されるときと同じようにテキスト表示を制御する。`face`プロパティをもたない文字はデフォルトのフェイス `mode-line`、または `mode-line-inactive`で表示される (Section “Standard Faces” in *The GNU Emacs Manual* を参照)。*string*内の `help-echo`プロパティと `keymap`プロパティは特別な意味をもつ。Section 23.4.6 [Properties in Mode], page 492 を参照のこと。

`symbol`
: モードライン構文におけるシンボルはその値を意味する。モードライン構文としては、*symbol*の値は *symbol*の位置に使用される。しかしシンボル `t`と `nil`は値が void であるようなシンボルとして無視される。

  例外が 1 つある。*symbol*の値が文字列なら、それはそのまま表示されて`%`構文は認識されない。

  *symbol*が risky(危険) とマークされていない (非 `nil`の `risky-local-variable`プロパティをもつ) 場合には、*symbol*の値中で指定されたテキストプロパティはすべて無視される。これには *symbol*の値中の文字列のテキストプロパティ、同様に文字列内の `:eval`フォームと `:propertize`フォームすべてが含まれる (これはセキュリティー上の理由による。危険とマークされていない変数は、ユーザーへの問い合わせなしでファイル変数から自動的にセットされ得る)。

`(`*`string rest...`*`)`
`(`*`list rest...`*`)`
: 最初の要素が文字列かリストであるようなリストは、すべての要素を再帰的に処理してその結果を結合することを意味する。これはモードライン構文においてもっとも一般的なフォームである。

`(:eval` *`form`*`)`
: 最初の要素がシンボル `:eval`であるようなリストは、*form*を評価してその結果を表示する文字列として使用するよう指示する。この評価が任意のファイルをロードできないことを確認すること。ファイルをロードすると無限再帰が発生するかもしれない。

`(:propertize` *elt props...*`)`
: 最初の要素がシンボル`:propertize`であるようなリストはモードライン構文 *elt*を再帰的に処理して、*props*により指定されるテキストプロパティに結果を加えるよう指示する。引数 *props*は 0 個以上の *text-property*と *value*のペアーで構成されること。*elt*がテキストプロパティをもつ文字列、またはテキストプロパティをもつ文字列を生成する場合には、その文字列内のすべての文字は同一のプロパティをもつこと。さもなければ`:propertize`によっていくつかのプロパティは削除されるかもしれない。

`(`*symbol then else*`)`
: 最初の要素がキーワード以外のシンボルであるようなリストは条件文を指定する。その意味は *symbol*の値に依存する。*symbol*が非`nil`値をもつ場合は、モードライン構文として 2 つ目の要素 *then*が再帰的に処理され、それ以外は 3 つ目の要素 *else*が再帰的に処理される。*else*は省略でき、その場合には *symbol*の値が`nil`か void ならモードライン構文は何も表示しない。

`(`*width rest...*`)`
: 最初の要素が整数であるようなリストは *rest*の結果の切り詰め、またはパディングを指定する。残りの要素 *rest*はモードライン構文として再帰的に処理されて互いに結合される。*width*が正で結果の幅が *width*より少なければ右側にスペースがパディングされる。*width*が負で結果の幅が $-$*width*より大きければ右側が切り詰められる。

  たとえばウィンドウ最上部からのバッファー位置をパーセント表示するには`(-3 "%p")`のようなリストを使用すればよい。

23.4.3 モードライン制御のトップレベル

変数`mode-line-format`はモードラインの全体的な制御を行います。

`mode-line-format` [User Option]
: この変数の値はモードラインのコンテンツを制御するモードライン構文である。これはすべてのバッファーにおいて常にバッファーローカルである。

  あるバッファー内でこの変数に`nil`をセットすると、そのバッファーはモードラインをもたない (高さが 1 行しかないウィンドウもモードラインを表示しない)。

`mode-line-format`のデフォルト値は`mode-line-position`や`mode-line-modes` (これは`mode-name`と`minor-mode-alist`の値を組み込む) のような、他の変数の値を使用するようデザインされています。`mode-line-format`自体を変更する必要があるモードはほとんどありません。ほとんどの用途にたいしては、`mode-line-format`が直接または間接的に参照するいくつかの変数を修正すれば十分です。

`mode-line-formatl`自体の変更を行う場合には、コンテンツを複製したり異なる様式で情報を表示するのではなく、新たな値にはデフォルト値 (Section 23.4.4 [Mode Line Variables], page 488 を参照) に出現する同じ変数を使用するべきです。この方法を使用すればユーザーや (`display-time`やメジャーモードのような)Lisp プログラムにより行われたカスタマイズは、それらの変数への変更を通じて効力を保ちます。

以下は Shell モードにたいして有用かもしれない架空の`mode-line-format`の例です (実際には Shell モードは`mode-line-format`をセットしない):

```
(setq mode-line-format
  (list "-"
   'mode-line-mule-info
   'mode-line-modified
   'mode-line-frame-identification
   "%b--"
   ;; これはリスト作成中に評価されることに注意
   ;; これは単なる文字列のモードライン構文を作成する
   (getenv "HOST")
   ":"
   'default-directory
   "   "
   'global-mode-string
   "   %[("
   '(:eval (format-time-string "%F"))
   'mode-line-process
   'minor-mode-alist
   "%n"
   ")%]--"
   '(which-func-mode ("" which-func-format "--"))
   '(line-number-mode "L%l--")
   '(column-number-mode "C%c--")
   '(-3 "%p")))
```

(変数 `line-number-mode`、`column-number-mode`、`which-func-mode`は特定のマイナーモードを有効にする。これらの変数名は通常のようにマイナーモードコマンド名でもある。)

23.4.4 モードラインで使用される変数

このセクションでは `mode-line-format`の標準的な値としてモードラインテキストに組み込まれる変数を説明します。これらの変数は本質的には特別なものではありません。`mode-line-format`が使用する変数を他の変数に変更すれば、それらはモードライン上で同様の効果をもちます。しかしEmacs のさまざまな部分は、それらの変数がモードラインを制御するという認識の元でそれらの変数をセットします。したがって事実上モードラインでそれらの変数を使用するのは必須なのです。Section "Optional Mode Line" in *The GNU Emacs Manual* も参照してください。

`mode-line-mule-info` [Variable]

この変数は言語環境 (language environment)、バッファーコーディングシステム、カレント入力メソッド (current input method) に関する情報のモードライン構文の値を保持する。Chapter 33 [Non-ASCII Characters], page 852 を参照のこと。

`mode-line-modified` [Variable]

この変数はカレントバッファーが変更されたかどうかを表示するモードライン構文の値を保持する。デフォルト値ではバッファーが変更されていれば '`**`'、バッファーが変更されていなければ '`--`'、バッファーが読み取り専用なら '`%%`'、読み取り専用だが変更されているときは '`%*`' を表示する。

この変数を変更してもモードラインは強制的に更新されない。

`mode-line-frame-identification` [Variable]
この変数はカレントフレームを識別する。デフォルト値では複製フレームを表示可能なウィンドウシステムを使用している場合は`" "`、一度に 1 つのフレームだけを表示する通常の端末では`"-%F "`を表示する。

`mode-line-buffer-identification` [Variable]
この変数はそのウィンドウ内で表示されているバッファーを識別する。デフォルト値では少なくとも 12 列になるようスペースパディングされたバッファー名を表示する。

`mode-line-position` [Variable]
この変数はバッファー内での位置を表示する。デフォルト値ではバッファーのパーセント位置、オプションでバッファーサイズ、行番号、列番号を表示する。

`mode-line-percent-position` [User Option]
このオプションは`mode-line-position`の中で使用される。これの値はバッファーのパーセンテージ (`nil`、`"%o"`、`"%p"`、`"%P"`、`"%q"`のいずれか。Section 23.4.5 [%-Constructs], page 491 を参照) とフィルするスペースか切り詰めの幅の両方を指定する。このオプションは`customize-variable`機能でセットすることを推奨する。

`vc-mode` [Variable]
変数`vc-mode`は各バッファーにたいしてバッファーローカルであり、そのバッファーが visit しているファイルがバージョンコントロールで保守されているかどうか、保守されている場合はバージョンコントロールシステムの種別を表示する。値はモードラインに表示される文字列、またはバージョンコントロールされていなければ`nil`。

`mode-line-modes` [Variable]
この変数はそのバッファーのメジャーモードとマイナーモードを表示する。デフォルト値では再帰編集レベル (recursive editing level)、プロセス状態の情報、ナローイング (narrowing) 効果の有無を表示する。

`mode-line-remote` [Variable]
この変数はカレントバッファーの`default-directory`がリモートかどうかを表示するために使用される。

`mode-line-client` [Variable]
この変数は`emacsclient`フレームを識別するために使用される。

以下の 3 つの変数は`mode-line-modes`内で使用されます:

`mode-name` [Variable]
このバッファーローカル変数はカレントバッファーのメジャーモードの “愛称 (pretty name)” を保持する。モードラインにモード名が表示されるように、すべてのメジャーモードはこの変数をセットすること。値は文字列である必要はなく、モードライン構文内で有効な任意のデータ型 (Section 23.4.2 [Mode Line Data], page 486 を参照) を使用できる。モードライン内でモード名を識別する文字列の計算には`format-mode-line`を使用する (Section 23.4.8 [Emulating Mode Line], page 493 を参照)。

`mode-line-process` [Variable]
このバッファーローカル変数には、そのモードにおいてサブプロセスとの通信にたいするプロセス状態のモードライン情報が含まれる。これはメジャーモード名の直後 (間にスペースはない) に表示される。たとえば`*shell*`バッファーでの値は`(":%s")`であり、これは‘`(Shell:run)`’のように、メジャーモードとともにその状態を表示する。この変数は通常は`nil`。

`mode-line-front-space` [Variable]
この変数はモードラインの一番前に表示される。memory-full メッセージがある場合を除き、デフォルトではこの構文はモードライン先頭の右側に表示される。

`mode-line-end-spaces` [Variable]
この変数はモードラインの終端に表示される。

`mode-line-misc-info` [Variable]
その他の情報にたいするモードライン構文。デフォルトでは `global-mode-string`で指定される情報を表示する。

`minor-mode-alist` [Variable]
この変数はアクティブなマイナーモードをモードラインに示す方法を指定する要素をもった連想リスト (association list) を保持する。`minor-mode-alist`の各要素は以下のような 2 要素のリストであること:

```
(minor-mode-variable mode-line-string)
```

より一般的には *mode-line-string*は任意のモードライン構文を指定できる。*minor-mode-variable*の値が非 `nil`ならモードラインに表示され、それ以外なら表示されない。混合しないようにこれらの文字列はスペースで始めること。慣例的に特定のモードにたいする *minor-mode-variable*は、そのマイナーモードがアクティブになった際に非 `nil`値にセットされる。

`minor-mode-alist`自体はバッファーローカルではない。この alist 内で参照される各変数は、そのマイナーモードをバッファーごとに個別に有効にできるならバッファーローカルであること。

`global-mode-string` [Variable]
この変数はモードライン内でマイナーモード `which-func-mode`がセットされていればその直後、セットされていなければ`mode-line-modes`の後に表示されるモードライン構文を保持する (デフォルト)。コマンド `display-time`は、」時間とロードの情報を含む文字列を保持する変数 `display-time-string`を参照する `global-mode-string`をセットする。

'`%M`'構文は `global-mode-string`の値を置き換えるが、この変数は `mode-line-format`からモードラインに include されるので時代遅れである。

以下は`mode-line-format`のデフォルト値の簡略化バージョンです。実際のデフォルト値には追加のテキストプロパティ指定も含まれます。

```
("-"
 mode-line-mule-info
 mode-line-modified
 mode-line-frame-identification
 mode-line-buffer-identification
 "   "
 mode-line-position
 (vc-mode vc-mode)
 "   "
 mode-line-modes
 (which-func-mode ("" which-func-format "--"))
 (global-mode-string ("--" global-mode-string))
 "-%-")
```

23.4.5 モードラインでの%構文

モードライン構文として使用される文字列では、さまざまな種類のデータを置き換えるために%構文を使用できます。以下は定義済みの%構文と意味のリストです。

‘%%’以外の構文では、フィールドの最小幅を指定するために‘%’の後に 10 進整数を追加できます。幅がそれより小さければそのフィールドは最小幅にパディングされます。純粋に数値的な構文 (‘c’、‘i’、‘I’、‘l’) は左側、それ以外は右側にスペースを追加してパディングされます。

`%b` `buffer-name`関数により取得されるカレントバッファー名。Section 27.3 [Buffer Names], page 592 を参照のこと。

`%c` ポイント位置のカレント列番号。そのウィンドウの左マージンより 0 からカウントされる。

`%C` ポイント位置のカレント列番号。そのウィンドウの左マージンより 1 からカウントされる。

`%e` Emacs が Lisp オブジェクトにたいしてメモリー不足になりそうなときは、それを伝える簡略なメッセージを示す。それ以外の場合は空。

`%f` `buffer-file-name`関数により取得される visit 中のファイル名。Section 27.4 [Buffer File Name], page 594 を参照のこと。

`%F` 選択されたフレームのタイトル (ウィンドウシステム上のみ) か名前。Section 29.4.3.1 [Basic Parameters], page 715 を参照のこと。

`%i` カレントバッファーのアクセス可能な範囲のサイズ。基本的には `(- (point-max) (point-min))`。

`%I` ‘`%i`’と同様だが 10^3 は‘`k`’、10^6 は‘`M`’、10^9 は‘`G`’を使用して略記することで、より読みやすい方法でサイズをプリントする。

`%l` ポイント位置のカレント行番号。そのバッファーのアクセス可能な範囲内でカウントされる。

`%n` ナローイングが有効なときは‘`Narrow`’、それ以外は何も表示しない (Section 30.4 [Narrowing], page 766 の `narrow-to-region`を参照)。

`%o` バッファー (の可視な範囲) を通じてウィンドウが *travel* した割合 (ウィンドウ外部にあるすべてのテキストにたいしてウィンドウ上端の上にあるテキストのサイズのパーセンテージまたは‘`Top`’、‘`Bottom`’、‘`All`’)。

`%p` ウィンドウの最上部より上にあるバッファーテキストのパーセント表示、または‘`Top`’、‘`Bottom`’、‘`All`’のいずれか。デフォルトのモードライン構文は、これを 3 文字に切り詰めることに注意。

`%P` ウィンドウの最下部より上にあるバッファーテキスト (ウィンドウ内の可視なテキストと最上部の上にあるテキスト) のパーセント表示、およびバッファーの最上部がスクリーン上で可視なら、それに加えて‘`Top`’。または‘`Bottom`’か‘`All`’。

`%q` ‘`-`’で区切ったウィンドウの上端および下端より上にあるテキストのパーセンテージ、または‘`All`’。

`%s` `process-status`により取得されるカレントバッファーに属するサブプロセスの状態。Section 38.6 [Process Information], page 947 を参照のこと。

`%z` キーボード、端末、およびバッファーコーディングシステムのニーモニック。

`%Z` '`%z`'と同様だが、EOL 形式 (end-of-line format: 改行形式) を含む。

`%*` バッファーが読み取り専用 (`buffer-read-only`を参照) なら '`%`'、
変更 (`buffer-modified-p`を参照) されていれば '`*`'、
それ以外は '`-`'。Section 27.5 [Buffer Modification], page 596 を参照のこと。

`%+` バッファーが変更 (`buffer-modified-p`を参照) されていれば '`*`'
バッファーが読み取り専用 (`buffer-read-only`を参照) なら '`%`'、
それ以外は '`-`'。これは読み取り専用バッファーの変更にたいしてのみ '`%*`'と異なる。
Section 27.5 [Buffer Modification], page 596 を参照のこと。

`%&` バッファーが変更されてれば '`*`'、それ以外は '`-`'。

`%[` 再帰編集レベルの深さを表示する (ミニバッファーレベルは勘定しない) 編集レベル 1 つが '`[`'。Section 21.13 [Recursive Editing], page 417 を参照のこと。

`%]` 編集レベル 1 つが '`]`'(ミニバッファーレベルは勘定しない)。

`%-` モードラインの残りを充填するのに十分なダッシュ。

`%%` 文字 '`%`'。`%`構文が許される文字列内にリテラル '`%`'を含めるにはこの方法を使用する。

以下の 2 つの`%`構文はまだサポートされていますが、同じ結果を変数`mode-name`と`global-mode-string`で取得できるので時代遅れです。

`%m` `mode-name`の値。

`%M` `global-mode-string`の値。

23.4.6 モードラインでのプロパティ

モードライン内では特定のテキストプロパティが意味をもちます。`face`プロパティはテキストの外見に影響します。`help-echo`プロパティはそのテキストのヘルプ文字列に関連し、`keymap`によりテキストをマウスに感応させることができます。

モードライン内のテキストにたいしてテキストプロパティを指定するには 4 つの方法があります:

1. モードラインデータ構造内にテキストプロパティをもつ文字列を直接配置する。
2. '`%12b`'のようなモードライン%構文にテキストプロパティを配置する。その場合には%構文を展開すると同じテキストプロパティをもつことになる。
3. *props*で指定されるテキストプロパティを *elt*に与えるために `(:propertize` *elt props*`...)` 構文を使用する。
4. *form*がテキストプロパティをもつ文字列に評価されるようにモードラインデータ構造内に`:eval` *form*を含むリストを使用する。

キーマップを指定するために`keymap`プロパティを使用できます。このキーマップはマウスクリックにたいしてのみ実際の効果をもちます。モードライン内にポイントを移動させるのは不可能なので、これに文字キーやファンクションキーをバインドしても効果はありません。

`risky-local-variable`が非`nil`であるようなプロパティをもつ変数をモードラインが参照する場合には、その変数の値から取得または指定されるテキストプロパティはすべて無視されます。そのようなプロパティは呼び出される関数を指定するかもしれず、その関数はファイルローカル変数に由来するかもしれないからです。

23.4.7 ウィンドウのヘッダーライン

最下部にモードラインをもつことができるのと同じように、ウィンドウは最上部にヘッダーライン *(header line)* をもつことができます。ヘッダーライン機能は、それが`header-line-format`によって制御されることを除けばモードラインと同じように機能します。

`header-line-format` [Variable]
すべてのバッファーにたいしてローカルなこの変数は、そのバッファーを表示するバッファーにたいしてヘッダーラインを表示する方法を指定する。この変数の値のフォーマットは`mode-line-format`にたいするフォーマットと同じ (Section 23.4.2 [Mode Line Data], page 486 を参照)。この変数は通常は`nil`なので、通常のバッファーはヘッダーラインをもたない。

`window-header-line-height` **&optional** *window* [Function]
この関数は *window*のヘッダーラインの高さをピクセルでリターンする。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。

高さが 1 行しかないウィンドウがヘッダーラインを表示することは決してありません。また高さが 2 行しかないウィンドウは、同時にモードラインとヘッダーラインを表示できません。そのようなウィンドウがモードラインをもつ場合にはヘッダーラインは表示されません。

23.4.8 モードラインのフォーマットのエミュレート

関数`format-mode-line`を使用して、特定のモードライン構文にもとづいてモードラインやヘッダーラインに表示されるテキストを計算できます。

`format-mode-line` *format* **&optional** *face window buffer* [Function]
この関数は、あたかも *window*にたいしてモードラインを生成するかのように *format*に応じてテキスト行をフォーマットするが、さらにそのテキストを文字列としてリターンする。引数 *window*のデフォルトは選択されたウィンドウ。*buffer*が非`nil`なら、使用されるすべての情報は *buffer*から取得される。デフォルトでは *window*のバッファーから取得される。

文字列の値は通常はモードラインがもつであろうフェイス、キーマップ等に対応したテキストプロパティをもつ。*format*により指定される`face`プロパティをもたないすべての文字は、*face*により決定されるデフォルト値を取得する。*face*が`t`の場合は *window*が選択されていれば`mode-line`、それ以外は`mode-line-inactive`であることを意味する。*face*が`nil`または省略された場合はデフォルトのフェイスを意味する。*face*が整数なら、この関数はテキストプロパティをもたない値をリターンするだろう。

*face*の値として他の有効なフェイスを指定することもできる。指定された場合、それは *format*でフェイスを指定されていない文字の`face`プロパティのフェイスを提供する。

*face*として`mode-line`、`mode-line-inactive`、`header-line`を使用することにより、フォーマットされた文字列のリターンに加えて、対応するフェイスのカレント定義を使用して実際にモードラインやヘッダーラインの再描画が行われることに注意 (他のフェイスでは再描画は行われない)。

たとえば`(format-mode-line header-line-format)`は選択されたウィンドウに表示されるテキスト (ヘッダーラインがない場合は`""`) をリターンするだろう。`(format-mode-line header-line-format 'header-line)`は、各文字がヘッダーライン内でもつであろうフェイスをもつ同じテキストをリターンするとともに、それに加えてヘッダーラインの再描画も行う。

23.5 Imenu

Imenu とはバッファー内の定義やセクションをすべてリストするメニューをユーザーが選択することによって、バッファー内の該当箇所に直接移動する機能です。Imenu は定義 (またはバッファーのその他の名前つき範囲) の名前とその定義のバッファー内での位置をリストするバッファーインデックスを構築して、ユーザーがそれを選択すればポイントをそこに移動できるようにして機能します。メジャーモードは `imenu-add-to-menubar`を使用して、メニューバーアイテムを追加することができます。

`imenu-add-to-menubar` *name* [Command]
この関数は Imenu を実行するための *name*という名前のローカルメニューバーを定義する。

Imenu を使用するためのユーザーレベルコマンドは Emacs マニュアルで説明されています (Section "Imenu" in *the Emacs Manual* を参照)。このセクションでは特定のメジャーモードにたいして定義や名前つき範囲を見つける Imenu メソッドのカスタマイズ方法を説明します。

変数 `imenu-generic-expression`をセットするのが通常、かつもっともシンプルな方法です:

`imenu-generic-expression` [Variable]
この変数が非 `nil`なら、それは Imenu にたいして定義を探すための正規表現を指定するリストである。シンプルな `imenu-generic-expression`の要素は以下のようになる:

```
(menu-title regexp index)
```

ここで *menu-title*が非 `nil`なら、それはこの要素にたいするマッチがバッファーインデックスのサブメニューとなることを指示する。*menu-title*自体はそのサブメニューにたいして名前を指定する。*menu-title*が `nil`なら、この要素にたいするマッチは直接トップレベルのバッファーインデックスとなる。

このリストの 2 つ目の要素 *regexp*は正規表現である (Section 34.3 [Regular Expressions], page 883 を参照)。これはバッファー内でこれにマッチするものは定義、あるいはバッファーインデックス内に記載すべき何かであると判断される。3 つ目の要素 *index*は 0 以上の整数なら、*regexp*内の部分式 (subexpression) が定義名にマッチすることを示す。

以下のような要素もある:

```
(menu-title regexp index function arguments...)
```

この要素にたいする各マッチはインデックスアイテムを作成して、ユーザーにがそのインデックスアイテムを選択したときアイテム名、バッファー位置、および *arguments*から構成される引数で *function*を呼び出す。

Emacs Lisp モードでは `imenu-generic-expression`は以下のようになるだろう:

```
((nil "^\\s-*(def\\(un\\|subst\\|macro\\|advice\\)\
\\s-+\\([-A-Za-z0-9+]+\\)" 2)
 ("*Vars*" "^\\s-*(def\\(var\\|const\\)\
\\s-+\\([-A-Za-z0-9+]+\\)" 2)
 ("*Types*"
  "^\\s-*\
(def\\(type\\|struct\\|class\\|ine-condition\\)\
\\s-+\\([-A-Za-z0-9+]+\\)" 2))
```

この変数はセットによりカレントバッファーにたいしてバッファーローカルになる。

`imenu-case-fold-search` [Variable]
この変数は `imenu-generic-expression`の値中の正規表現マッチが case(大文字小文字) を区別するかどうかを制御する。`t`(デフォルト) なら case の違いを無視することを意味する。

この変数はセットによりカレントバッファーにたいしてバッファーローカルになる。

`imenu-syntax-alist` [Variable]
この変数は `imenu-generic-expression`処理中に、カレントバッファーの構文テーブルをオーバーライドするために使用する構文テーブル変更用の alist。この alist の各要素は以下の形式をもつこと:

```
(characters . syntax-description)
```

CAR の *characters*には文字か文字列を指定できる。この要素はその文字か文字列が *syntax-description*により指定される構文であることを示し、`modify-syntax-entry`に渡される (Section 35.3 [Syntax Table Functions], page 910 を参照)。

典型的にはこの機能はシンボル構文 (symbol syntax) をもつ文字にたいして単語構文 (word syntax) を与えるために通常は使用され、それにより `imenu-generic-expression`が単純になってマッチングのスピードも向上する。たとえば Fortran モードでは以下のようにこれを使用する:

```
(setq imenu-syntax-alist '(("_$" . "w")))
```

`imenu-generic-expression`の正規表現は、'`\\(\\sw\\|\\s_\\)+`'のかわりに、'`\\sw+`'を使用できる。このテクニックは名前をモード名として許容されるより短い頭文字に制限する必要があるときは不便かもしれないことに注意。

この変数はセットによりカレントバッファーにたいしてバッファーローカルになる。

あるメジャーモードにたいして Imenu をカスタマイズする別の方法として `imenu-prev-index-position-function`と `imenu-extract-index-name-function`があります:

`imenu-prev-index-position-function` [Variable]
この変数が非 `nil`なら、その値はポイント位置からバッファーを後方にスキャンしてバッファーインデックスに配置すべき次の定義を探すための関数であること。そしてポイントより前に他の定義が見つからなければ `nil`をリターンすること。見つかった場合には定義を見つけた場所にポイントを残して任意の非 `nil`値をリターンすること。

この変数はセットによりカレントバッファーにたいしてバッファーローカルになる。

`imenu-extract-index-name-function` [Variable]
この変数が非 `nil`なら、その値はポイントが定義中にある (`imenu-prev-index-position-function`関数がポイントを残す場所) という想定にもとづき、その定義の名前をリターンする関数であること。

この変数はセットによりカレントバッファーにたいしてバッファーローカルになる。

メジャーモードにたいして Imenu をカスタマイズするための最後の方法は変数 `imenu-create-index-function`のセットです:

`imenu-create-index-function` [Variable]
この変数はバッファーインデックスを作成するために使用する関数を指定する。この関数は引数がを受け取らず、カレントバッファーにたいするインデックス alist(index alist) をリターンすること。この関数は `save-excursion`内で呼び出されるので、どこにポイントを残しても違いはない。

このインデックス alist は 3 つのタイプの要素をもつことができる。以下はシンプル要素 (simple element) の例:

```
(index-name . index-position)
```

シンプル要素の選択はそのバッファー内の位置 *index-position*に移動する効果をもつ。スペシャル要素 (special element) は以下のようなもの:

```
(index-name index-position function arguments...)
```

スペシャル要素の選択により以下が処理される:

```
(funcall function
         index-name index-position arguments...)
```

ネストされたサブ alist 要素 (nested sub-alist element) は以下のようなもの:

```
(menu-title . sub-alist)
```

これは *sub-alist*により指定されるサブメニュー *menu-title*を作成する。

`imenu-create-index-function`の デ フォル ト 値 は `imenu-default-create-index-function`。この関数はインデックス alist を生成するために `imenu-prev-index-position-function`の 値 と `imenu-extract-index-name-function`の 値 を呼び出す。しかしこれら 2 つ変数のいずれかが `nil`なら、デフォルト関数はかわりに `imenu-generic-expression`を使用する。

この変数はセットによりカレントバッファーにたいしてバッファーローカルになる。

23.6 Font Lock モード

Font Lock モードとはバッファーの特定の部分にたいして、それらの構文的役割 (syntactic role) にもとづき自動的に `face`プロパティをアタッチするバッファーローカルなマイナーモードです。このモードがバッファーをパースする方法はそのメジャーモードに依存します。ほとんどのメジャーモードは、どのコンテキストでどのフェイスを使用するかにたいして構文的条件 (syntactic criteria) を定義します。このセクションでは特定のメジャーモードにたいして Font Lock をカスタマイズする方法を説明します。

Font Lock モードは 2 つの方法によりハイライトするテキストを探します。それは構文テーブル (syntax table) にもとづく構文解析と、(通常は正規表現にたいする) 検索です。最初に構文的フォント表示 (syntactic fontification) が発生します。これはコメントと文字列定数を見つけてそれらをハイライトします。検索ベースのフォント表示が発生するのは 2 番目です。

23.6.1 Font Lock の基礎

Font Lock 機能はいくつかの基本的な関数にもとづきます。これらはそれぞれ対応する変数により指定される関数を呼び出します。このインダイレクションによりメジャーモードとマイナーモードはそのモードにあるバッファーのフォント表示が機能する方法を変更したり、フォント表示を何も行わない機能にたいしてさえ Font Lock メカニズムを使用することが可能になります (以下の記述で関数が何を行うか説明する際に “should(すること、するべき)” と表現しているのはこれが理由。モードは完全に異なる何かを行うように対応する変数をカスタマイズできる)。以下で言及される変数は Section 23.6.4 [Other Font Lock Variables], page 502 で説明されています。

`font-lock-fontify-buffer`
: この関数は `font-lock-fontify-buffer-function`で指定される関数の呼び出しにより、カレントバッファーのアクセス可能範囲をフォント表示すること。

`font-lock-unfontify-buffer`
: フォント表示削除のために Font Lock をオフに切り替える際に使用する。`font-lock-unfontify-buffer-function`で指定される関数を呼び出す。

`font-lock-fontify-region beg end &optional loudly`
: *beg*と *end*の間のリージョンをフォント表示すること。*loudly*が非`nil`なら、フォント表示中にステータスメッセージを表示すること。`font-lock-fontify-region-function`で指定される関数を呼び出す。

`font-lock-unfontify-region beg end`
: *beg*と *end*の間のリージョンのフォント表示を削除すること。`font-lock-unfontify-region-function`で指定される関数を呼び出す。

`font-lock-flush &optional beg end`
: この関数は *beg*と *end*の間のリージョンのフォント表示を期限切れ (outdated) とマークすること。*beg*と *end*が未指定または`nil`なら、デフォルトはそのバッファーのアクセス可能範囲の先頭と終端。`font-lock-flush-function`で指定される関数を呼び出す。

`font-lock-ensure &optional beg end`
: この関数は *beg*と *end*の間のリージョンのフォント表示を保証すること。オプション引数 *beg*と *end*のデフォルトは、そのバッファーのアクセス可能範囲の先頭と終端。`font-lock-ensure-function`で指定される関数を呼び出す。

`font-lock-debug-fontify`
: これはモード用フォントロック開発時の使用を意図した便利なコマンドであり、Lisp コードから呼び出すべきではありません。これは関連するすべての変数を再計算してから、バッファー全体にたいして`font-lock-fontify-region`を呼び出します。

Font Lock モードのテキストのハイライト方法を制御する変数がいくつかあります。しかしメジャーモードはこれらの変数を直接セットするべきではありません。かわりにメジャーモードはバッファーローカル変数として`font-lock-defaults`をセットするべきです。Font Lock モードが有効なときは、他のすべての変数をセットするためにこの変数に割り当てられた値が使用されます。

`font-lock-defaults` [Variable]
: この変数はそのモード内のテキストをフォント表示する方法を指定するためにモードによりセットされる。この変数はセットした際に自動的にバッファーローカルになる。変数の値が`nil`なら Font Lock モードはハイライトを行わず、バッファー内のテキストに明示的にフェイスを割り当てるために‘`Faces`’メニュー (メニューバーの‘`Edit`’の下の‘`Text Properties`’) を使用できる。

  非`nil`なら値は以下のようであること:

  ```
  (keywords [keywords-only [case-fold
   [syntax-alist other-vars...]]])
  ```

  1 つ目の要素 *keywords*は検索ベースのフォント表示を制御する`font-lock-keywords`の値を間接的に指定する。値にはシンボル、変数、または`font-lock-keywords`にたいして使用するリストが値であるような関数を指定できる。またそれぞれのシンボルがフォント表示の可能なレベルであるような、いくつかのシンボルからなるリストも指定できる。この場合には、1 つ目のシンボルはフォント表示の‘モードデフォルト`(mode default)`’レベル、次のシンボルはフォント表示のレベル 1、その次はレベル 2、... のようになる。‘モードデフォルト’レベルは通常はレベル 1 と等しい。これは`font-lock-maximum-decoration`が`nil`値をもつとき使用される。Section 23.6.5 [Levels of Font Lock], page 503 を参照のこと。

2 つ目の要素 *keywords-only*は変数 `font-lock-keywords-only`の値を指定する。これが省略または `nil`なら、(文字列とコメントの) 構文的フォント表示も行われる。非 `nil`なら構文的フォント表示は行われない。Section 23.6.8 [Syntactic Font Lock], page 505 を参照のこと。

3 つ目の要素 *case-fold*は `font-lock-keywords-case-fold-search`の値を指定する。非 `nil`なら検索ベースフォント表示の間、Font Lock モードは case の違いを無視する。

4 つ目の要素 *syntax-alist*が非 `nil`なら、それは `(`*`char-or-string . string`*`)`という形式のコンスセルのリストであること。これらは構文的フォント表示にたいする構文テーブルのセットアップに使用される。結果となる構文テーブルは `font-lock-syntax-table`に格納される。*syntax-alist*が省略または `nil`なら、構文的フォント表示は `syntax-table`関数によりリターンされる構文テーブルを使用する。Section 35.3 [Syntax Table Functions], page 910 を参照のこと。

(もしあれば) 残りすべての要素はまとめて *other-vars*と呼ばれる。これらの要素はすべて `(`*`variable . value`*`)`という形式をもつこと。これは *variable*をバッファーローカルにしてから、それに *value*をセットすることを意味する。これら *other-vars*を使用して、最初の 5 つの要素による制御とは別にフォント表示に影響する他の変数をセットできる。Section 23.6.4 [Other Font Lock Variables], page 502 を参照のこと。

モードが `font-lock-face`プロパティ追加により明示的にテキストをフォント表示する場合には、自動的なフォント表示すべてをオフにするために `font-lock-defaults`に `(nil t)`を指定できます。しかしこれは必須ではありません。`font-lock-face`を使用して何かをフォント表示して、それ以外の部分のテキストを自動的にフォント表示するようにセットアップすることが可能です。

23.6.2 検索ベースのフォント化

検索ベースのフォント表示を直接制御する変数は `font-lock-keywords`です。この変数は通常は `font-lock-defaults`内の要素 *keywords*を通じて指定されます。

`font-lock-keywords` [Variable]
この変数の値はハイライトするキーワードのリスト。Lisp プログラムはこの変数を直接セットしないこと。通常は `font-lock-defaults`内の要素 *keywords*を使用して Font Lock モードが自動的に値をセットする。この値は関数 `font-lock-add-keywords`と `font-lock-remove-keywords`を使用して変更することもできる (Section 23.6.3 [Customizing Keywords], page 501 を参照)。

`font-lock-keywords`の各要素は、特定の例に該当するテキストを見つける方法や、それらをハイライトする方法を指定します。Font Lock モードは `font-lock-keywords`の要素を逐次処理してマッチを探して、すべてのマッチを処理します。通常はテキストの一部はすでに一度はフォント表示されており、同じテキスト内で連続するマッチによりこれをオーバーライドすることははできません。しかし *subexp-highlighter*の要素 *override*を使用して異なる挙動を指定できます。

`font-lock-keywords`の各要素は以下の形式のいずれかをもつべきです:

regexp `font-lock-keyword-face`を使用して *regexp*にたいするすべてのマッチをハイライトする。たとえば、

```
;; font-lock-keyword-faceを使用して
;; 単語 'foo'をハイライトする
"\\<foo\\>"
```

これらの正規表現を作成するときは慎重に行うこと。下手に記述されたパターンによりスピードが劇的に低下し得る! 関数 `regexp-opt` (Section 34.3.3 [Regexp Functions],

page 892 を参照) は、いくつかのキーワードとマッチするために最適な正規表現の計算に有用である。

function *function*を呼び出すことによりテキストを探し、`font-lock-keyword-face`を使用して見つかったマッチをハイライトする。

*function*は呼び出される際に 1 つの引数 (検索のリミット) を受け取る。検索はポイント位置から開始しリミットを超えた検索は行わないこと。これは検索が成功したら非 `nil` をリターンして見つかったマッチを表すマッチデータをセットすること。`nil`のリターンは検索の失敗を示す。

フォント表示は前の呼び出しでポイントが残された位置から同じリミットを用いて *function*を呼び出し、*function*が失敗するまで *function*を繰り返し呼び出すだろう。検索が失敗しても何らかの特別な方法で *function*がポイントをリセットする必要はない。

(*matcher* . *subexp*)
この種の要素では *matcher*は上述の regexp か function のいずれかである。CDR の *subexp*は、(*matcher*がマッチするテキスト全体のかわりに)*matcher*のどの部分式 (subexpression) がハイライトされるべきかを指定する。

```
;; font-lock-keyword-faceを使用して
;; 'bar'が 'fubar'の一部のときに
;; ハイライトする
("fu\\(bar\\)" . 1)
```

正規表現 *matcher*の生成に `regexp-opt`を使用する場合には、*subexp*にたいする値の計算に `regexp-opt-depth` (Section 34.3.3 [Regexp Functions], page 892 を参照) を使用できる。

(*matcher* . *facespec*)
この種の要素では *facespec*の値がハイライトに使用するフェイスを指定する。もっともシンプルな例では *facespec*は値がフェイス名であるようなは Lisp 変数 (シンボル)。

```
;; fubar-faceの値のフェイスを使用して
;; 'fubar'をハイライトする
("fubar" . fubar-face)
```

しかし *facespec*は以下のような形式のリストに評価されてもよい:

```
(face face prop1 val1 prop2 val2...)
```

これはマッチしたテキストにフェイス *face*を指定し、さまざまなテキストプロパティを put する。これを行う場合には、この方法によって `font-lock-extra-managed-props`に値をセットする、他テキストプロパティ名を確実に追加すること。そうすればそれらのプロパティが妥当性を失ったとき、それらのプロパティもクリアーされるだろう。これらのプロパティをクリアーする関数を変数 `font-lock-unfontify-region-function`にセットすることもできる。Section 23.6.4 [Other Font Lock Variables], page 502 を参照のこと。

(*matcher* . *subexp-highlighter*)
この種の要素では *subexp-highlighter*は *matcher*により見つかったマッチをハイライトする方法を指定するリストである。これは以下の形式をもつ。

```
(subexp facespec [override [laxmatch]])
```

CAR の *subexp*はマッチのどの部分式をフォント表示するかを指定する整数 (0 はマッチしたテキスト全体を意味する)。これの 2 つ目の要素 *facespec*は上述したような、値がフェイスを指定する式である。

*subexp-highlighter*内の残りの値 *override*と *laxmatch*はオプションのフラグである。*override*が `t`なら、この要素は前の `font-lock-keywords`の要素により作成された既存のフォント表示をオーバーライドできる。値が `keep`なら、すでに他の要素によりフォント表示されていない文字がフォント表示される。値が `prepend`なら、*facespec*により指定されたフェイスが `font-lock-face`プロパティの先頭に追加される。値が `append`なら、そのフェイスが `font-lock-face`プロパティの最後に追加される。

*laxmatch*が非 `nil`なら、それは *matcher*内で番号付けされた部分式 *subexp*が存在しなくてもエラーにならないことを意味する。番号付けされた部分式 *subexp*のフォント表示は当然発生しない。しかし他の部分式 (と他の regexp) のフォント表示は継続されるだろう。*laxmatch*が `nil`、かつ指定された部分式が存在しなければ、エラーがシグナルされて検索ベースのフォント表示は終了する。

以下はこのタイプの要素とそれが何を行うかの例:

```
;; foo-bar-faceを使用して、たとえハイライト済みでも
;; 'foo'と 'bar'をハイライトする
;; foo-bar-faceは値がフェイスであるような変数であること
("foo\\|bar" 0 foo-bar-face t)

;; fubar-faceの値のフェイスを使用して
;; 関数 fubar-matchが見つけた各マッチの
;; 最初の部分式をハイライトする
(fubar-match 1 fubar-face)
```

(*matcher* . *anchored-highlighter*)

この種の要素では *anchored-highlighter*は *matcher*が見つけたマッチに後続するテキストをハイライトする方法を指定する。つまり *matcher*が見つけたマッチは、*anchored-highlighter*により指定されるその先の検索にたいするアンカー (anchor) として機能する。*anchored-highlighter*は以下の形式のリストである:

```
(anchored-matcher pre-form post-form
                  subexp-highlighters...)
```

ここで *anchored-matcher*は *matcher*と同様、正規表現か関数である。*matcher*にたいするマッチを見つけた後に、ポイントはそのマッチの終端に移動する。そこで Font Lock はフォーム *pre-form*を評価する。それから *anchored-matcher*にたいするマッチを検索し、*subexp-highlighters*を使用してそれらのマッチをハイライトする。*subexp-highlighter*については上記を参照のこと。最後に Font Lock は *post-form*を評価する。

フォーム *pre-form*と *post-form*は、*anchored-matcher*使用時の事前の初期化と事後のクリーンアップに使用できる。*pre-form*は通常は *anchored-matcher*の開始前に、*matcher*のマッチに関連する何らかの位置にポイントを移動するために使用される。*post-form*は、*matcher*の再開前にポイントを戻すために使用できる。

*pre-form*を評価した後、Font Lock はその行の終端の先にたいして *anchored-matcher*の検索を行わない。しかし *pre-form*が *pre-form*評価後のポイント位置より大きいバッファー位置をリターンした場合には、かわりに *pre-form*によりリターンされた位置が検索リミットとして使用される。その行の終端より大きい位置をリターンするのは、一般的にはよいアイデアではない。言い換えると *anchored-matcher*検索は複数行にわたる (span lines) べきではない。

たとえば、

```
;; item-faceの値を使用して
;; 単語 'anchor'に (同一行内で)
;; 後続する単語 'item'をハイライトする
```

```
("\\<anchor\\>" "\\<item\\>" nil nil (0 item-face))
```

ここでは *pre-form*と *post-form*は `nil`である。したがって '`item`'にたいする検索は '`anchor`'にたいするマッチの終端から開始されて、後続する '`anchor`'インスタンスにたいする検索は '`item`'にたいする検索が終了した位置から再開される。

(*matcher highlighters...*)
: この種の要素は単一の *matcher*にたいして複数の *highlighter*リストを指定する。*highlighter*リストには、上述した *subexp-highlighter*か *anchored-highlighter*のいずれかを指定できる。

  たとえば、

```
;; anchor-faceの値内に現れる単語 'anchor'、
;; および、(同じ行の) 後続の item-faceの
;;  値内に現れる単語 'item'をハイライトする
("\\<anchor\\>" (0 anchor-face)
                ("\\<item\\>" nil nil (0 item-face)))
```

(eval . *form*)
: ここで *form*はバッファー内でこの `font-lock-keywords`の値が最初に使用されるときに評価される式である。この値は上述のテーブルで説明したいずれかの形式をもつこと。

警告**:** 複数行にわたるテキストにたいするマッチさせるために、`font-lock-keywords`の要素をデザインしてはならない。これは確実に機能するとは言えない。詳細は Section 23.6.9 [Multiline Font Lock], page 506 を参照のこと。

検索ベースのフォント表示が case を区別すべきかどうかを告げる `font-lock-keywords-case-fold-search`の値を指定するために `font-lock-defaults`内で *case-fold*を使用できる。

`font-lock-keywords-case-fold-search` [Variable]
: 非 `nil`は `font-lock-keywords`のための正規表現マッチングが case を区別すべきではないことを意味する。

23.6.3 検索ベースのフォント化のカスタマイズ

メジャーモードにたいして検索ベースフォント表示ルールを追加するために `font-lock-add-keywords`、削除には `font-lock-remove-keywords`を使用することができます。

`font-lock-add-keywords` *mode keywords* **&optional** *how* [Function]
: この関数はカレントバッファー、またはメジャーモード *mode*にたいしてハイライトする *keywords*を追加する。引数 *keywords*は変数 `font-lock-keywords`と同じ形式のリストであること。

  *mode*が、`c-mode`のようにメジャーモードのコマンド名であるようなシンボルなら、その *mode*内で Font Lock モードを有効にすることによって *keywords*が `font-lock-keywords`に追加される効果がある。非 `nil`値の *mode*による呼び出しは`~/.emacs`ファイル内でのみ正しい。

  *mode*が `nil`なら、この関数はカレントバッファーの `font-lock-keywords`に *keywords*を追加する。この方法での `font-lock-add-keywords`呼び出しは通常はモードフック関数内で使用される。

  デフォルトでは *keywords*は `font-lock-keywords`の先頭に追加される。オプション引数 *how*が `set`なら、それらは `font-lock-keywords`の値の置換に使用される。*how*がそれ以外の非 `nil`値なら、これらは `font-lock-keywords`の最後に追加される。

追加のハイライトパターンの使用を可能にする、特別なサポートを提供するモードがいくつかある。それらの例については変数 `c-font-lock-extra-types`、`c++-font-lock-extra-types`、`java-font-lock-extra-types`を参照のこと。

警告: メジャーモードコマンドはモードフックを除き、いかなる状況においても直接間接を問わず`font-lock-add-keywords`を呼び出してはならない (これを行うといくつかのマイナーモードは不正な振る舞いを起こしかねない)。メジャーモードコマンドは`font-lock-keywords`をセットすることにより、検索ベースフォント表示のルールをセットアップすること。

`font-lock-remove-keywords` *mode keywords* [Function]

この関数はカレントバッファーやメジャーモード *mode*にたいして、`font-lock-keywords`から *keywords*を削除する。`font-lock-add-keywords`の場合と同様に *mode*はメジャーモードコマンド名か`nil`であること。この関数にも`font-lock-add-keywords`にたいするすべての制約と条件が適用される。引数 *keywords*は対応する`font-lock-add-keywords`が使用するキーワードと正確に一致しなければならない。

たとえば以下は C モードに 2 つのフォント表示パターンを追加するコードの例である。フォント表示の 1 つはたとえコメント内であろうとも単語 'FIXME'をフォント表示し、もう 1 つは 'and'、'or'、'not'をキーワードとしてフォント表示する。

```
(font-lock-add-keywords 'c-mode
 '(("\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
   ("\\<\\(and\\|or\\|not\\)\\>" . font-lock-keyword-face)))
```

この例は厳密に C モードだけに効果がある。C モード、およびその派生モードにたいして同じパターンを追加するには、かわりに以下を行う:

```
(add-hook 'c-mode-hook
 (lambda ()
  (font-lock-add-keywords nil
   '(("\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
     ("\\<\\(and\\|or\\|not\\)\\>" .
      font-lock-keyword-face)))))
```

23.6.4 Font Lock のその他の変数

このセクションでは`font-lock-defaults`内の *other-vars*を用いて、メジャーモードがセットできる追加の変数について説明します (Section 23.6.1 [Font Lock Basics], page 496 を参照)。

`font-lock-mark-block-function` [Variable]

この変数が非`nil`なら、それはコマンド *M-o M-o* (`font-lock-fontify-block`) で再フォント表示するテキスト範囲を選択するために引数なしで呼び出される関数であること。

この関数は結果を報告するために選択されたテキスト範囲にリージョンを配置すること。正しい結果を与えるのに十分、かつ再フォント表示が低速にならない程度のテキスト範囲がよい選択である。典型的な値はプログラミングのモードにたいしては`mark-defun`、テキストを扱うモードにたいしては`mark-paragraph`。

`font-lock-extra-managed-props` [Variable]

この変数は、(`font-lock-face`以外の)Font Lock により管理される追加プロパティを指定する。これらの追加プロパティは通常は`font-lock-face`プロパティだけを管理する、`font-lock-default-unfontify-region`により使用される。他のプロパティも同様に Font Lock に管理させたければ、このリストに追加するのと同じように`font-lock-keywords`内の *facespec*内でもこれらを指定しなければならない。Section 23.6.2 [Search-based Fontification], page 498 を参照のこと。

`font-lock-fontify-buffer-function` [Variable]
そのバッファーをフォント表示するために使用する関数。デフォルト値は`font-lock-default-fontify-buffer`。

`font-lock-unfontify-buffer-function` [Variable]
そのバッファーを非フォント表示するために使用する関数。デフォルト値は`font-lock-default-unfontify-buffer`。

`font-lock-fontify-region-function` [Variable]
リージョンをフォント表示するための関数。この関数はリージョンの開始と終了の 2 つを引数に受け取り、オプションで 3 つ目の引数 *verbose*を受け取ること。*verbose*が非`nil`なら、その関数はステータスメッセージをプリントすべきである。デフォルト値は`font-lock-default-fontify-region`。

`font-lock-unfontify-region-function` [Variable]
リージョンを非フォント表示するための関数。この関数はリージョンの開始と終了の 2 つを引数に受け取ること。デフォルト値は`font-lock-default-unfontify-region`。

`font-lock-flush-function` [Variable]
リージョンのフォント表示の期限切れの宣言に使用する関数。そのリージョンの開始と終了という 2 つの引数を受け取る。この変数のデフォルト値は`font-lock-after-change-function`。

`font-lock-ensure-function` [Variable]
カレントバッファーのリージョンのフォント表示の保証に使用する関数。そのリージョンの開始と終了という 2 つの引数を受け取る。この変数のデフォルト値は、バッファーがフォント表示されていないときに`font-lock-default-fontify-buffer`を呼び出す関数。効果はそのバッファーのアクセス可能範囲全体がフォント表示されることの保証。

`jit-lock-register` *function* **&optional** *contextual* [Function]
この関数はカレントバッファーの一部をフォント表示/非表示する必要がある任意のタイミングで、Font Lock モードが Lisp 関数 *function*を実行することを宣言する。これはデフォルトのフォント表示関数が呼び出される前に、フォント表示/非表示するリージョンを指定する 2 つの引数 *start*と *end*で *function*を呼び出す。

オプション引数 *contextual*が非`nil`なら、行が更新されたときに限らずそのバッファーの構文的に関連する部分を常にフォント表示するよう Font Lock モードに強制する。この引数は通常は省略できる。

バッファーで Font Lock がアクティブのときには、もし`font-lock-keywords-only` (Section 23.6.8 [Syntactic Font Lock], page 505 を参照) の値が`nil`なら、非`nil`値の *contextual*でこの関数を呼び出す。

`jit-lock-unregister` *function* [Function]
以前に`jit-lock-register`を使用してフォント表示関数として *function*を登録した場合は、その関数を未登録にする。

23.6.5 Font Lock のレベル

フォント表示にたいして 3 つの異なるレベルを提供するモードがいくつかあります。`font-lock-defaults`内の *keywords*にたいしてシンボルのリストを使用することにより複数のレベルを定義できます。このリストのシンボルはそれぞれフォント表示の 1 レベルを指定します。これらのレベルの選択は、通常は`font-lock-maximum-decoration`をセットすることによりユーザーの責任で行わ

れます (Section “Font Lock” in *the GNU Emacs Manual* を参照)。選択されたレベルのシンボルの値は `font-lock-keywords`の初期化に使用されます。

フォント表示レベルの定義方法に関する慣習を以下に挙げます:

- レベル 1: 関数宣言、(include や import のような) ファイルディレクティブ、文字列、コメントをハイライトする。これは、もっとも重要かつトップレベルのコンポーネントだけをフォント表示すれば高速になるという発想である。
- レベル 2: レベル 1 に加えて、すべての言語のキーワード (キーワードと同様に作用する型名を含む)、および名前付き定数値をハイライトする。これは、(構文的、または意味的な) すべてのキーワードは適切にフォント表示されるべきという発想である。
- レベル 3: レベル 2 に加えて、関数内で定義されるシンボル、変数宣言、およびすべてのビルトイン関数名にたいして、それがどこに出現しようとハイライトする。

23.6.6 事前計算されたフォント化

`list-buffers`や `occur`のようないくつかのメジャーモードは、バッファーのテキストをプログラム的に構築します。これらにたいして Font Lock モードをサポートするためには、そのバッファーにテキストを挿入するタイミングでテキストのフェイスを指定するのがもっとも簡単な方法です。

これはスペシャルテキストプロパティ`font-lock-face` (Section 32.19.4 [Special Properties], page 820 を参照) により、テキスト内にフェイスを指定することによって行われます。Font Lock モードが有効になったとき、このプロパティは `face`と同じように表示を制御します。Font Lock モードが無効になると `font-lock-face`は表示に効果をもちません。

何らかのテキストにたいして `font-lock-face`を使用するモードや、通常の Font Lock 機構を使用するモードでも問題はありません。しかし通常の Font Lock 機構を使用しないモードでは、変数 `font-lock-defaults`をセットするべきではありません。この場合には `face`プロパティはオーバーライドされないので、`face`プロパティの使用も機能します。しかし `font-lock-mode`の切り替えによりユーザーがフォント化を制御でき、かつモードの Font Lock 機構の使用の有無に関わらずコードが機能するので、一般的には `font-lock-face`の使用の方が優っています。

23.6.7 Font Lock のためのフェイス

Font Lock モードはハイライトに任意のフェイスを使用できますが、Emacs は特に FontLock がテキストのハイライトに使用するいくつかのフェイスを定義しています。これらの *Font Lock* フェイス *(Font Lock faces)* を以下にリストします。これらのフェイスは FontLock モードの外部における構文的なハイライトでメジャーモードが使用することもできます (Section 23.2.1 [Major Mode Conventions], page 464 を参照)。

以下の各シンボルはフェイス名であり、かつデフォルト値がシンボル自身であるような変数でもあります。つまり `font-lock-comment-face`のデフォルト値は `font-lock-comment-face`です。

リストはそのフェイスの典型的な使い方の説明とともに、重要度が高い順にソートされています。あるモードの構文的カテゴリーが以下の使い方の記述にうまく適合しない場合には、この並び順をガイドとして使用することによってフェイスを割り当てることができるでしょう。

`font-lock-warning-face`
: 特有な構文 (たとえば‘ ‘`foo`’のように Emacs Lisp シンボルにおけるエスケープされていない判りにくいクォート) や、Emacs Lisp の‘`;;;###autoload`’、C の‘`#error`’のような他のテキストの意味を大きく変更する構文にたいして使用される。

`font-lock-function-name-face`
: 定義、または宣言される関数の名前にたいして使用される。

`font-lock-variable-name-face`
: 定義、または宣言される変数の名前にたいして使用される。

`font-lock-keyword-face`
: C の‘`for`’や‘`if`’のように、構文的に特別な意味をもつキーワードにたいして使用される。

`font-lock-comment-face`
: コメントにたいして使用される。

`font-lock-comment-delimiter-face`
: C の‘`/*`’と‘`*/`’のようなコメント区切りにたいして使用される。ほとんどの端末ではこのフェイスは`font-lock-comment-face`を継承する。

`font-lock-type-face`
: ユーザー定義データ型にたいして使用される。

`font-lock-constant-face`
: C の‘`NULL`’のような定数の名前にたいして使用される。

`font-lock-builtin-face`
: ビルトイン関数の名前にたいして使用される。

`font-lock-preprocessor-face`
: プロセッサーコマンドにたいして使用される。デフォルトでは、`font-lock-builtin-face`を継承する。

`font-lock-string-face`
: 文字列定数にたいして使用される。

`font-lock-doc-face`
: コード内のドキュメント文字列にたいして使用される。デフォルトでは`font-lock-string-face`を継承する。

`font-lock-negation-char-face`
: 見逃しやすい否定文字にたいして使用される。

23.6.8 構文的な Font Lock

構文的フォント表示 (syntactic fontification) は、構文的に関連性のあるテキストを探してハイライトするために構文テーブル (syntax table: Chapter 35 [Syntax Tables], page 906 を参照) を使用します。有効な場合には検索ベースのフォント表示に先立って実行されます。以下で説明する変数`font-lock-syntactic-face-function`はどの構文的構造をハイライトするかを決定します。構文的フォント表示に影響を与える変数がいくつかあります。`font-lock-defaults`のためにそれらをセットするべきです (Section 23.6.1 [Font Lock Basics], page 496 を参照)。

Font Lock モードが一連のテキストにたいして構文的フォント表示を処理するときは、常に`syntax-propertize-function`で指定される関数を最初に呼び出します。メジャーモードは特別なケースでは`syntax-table`テキストプロパティを適用してバッファーの構文テーブルをオーバーライドするために、これを使用することができます。Section 35.4 [Syntax Properties], page 912 を参照してください。

`font-lock-keywords-only` [Variable]
: この変数の値が非`nil`なら、Font Lock は構文的フォント表示を行わずに`font-lock-keywords`にもとづく検索ベースのフォント表示だけを行う。これは通常は`font-lock-defaults`内の *keywords-only* 要素にもとづいて Font Lock モードによりセットされる。

値が nilなら Font Lock は jit-lock-register (Section 23.6.4 [Other Font Lock Variables], page 502 を参照) を呼び出して、変更行以降のバッファーテキストに変更による新たな構文コンテキストを反映するために、自動的な再フォント表示をセットアップする。

font-lock-syntax-table [Variable]
この変数はコメントと文字列のフォント表示に使用するための構文テーブルを保持する。これは通常は font-lock-defaults内の *syntax-alist*要素にもとづいて Font Lock モードによりセットされる。この値が nilなら、構文的フォント表示はバッファーの構文テーブル (関数 syntax-tableがリターンする構文テーブル。Section 35.3 [Syntax Table Functions], page 910 を参照) を使用する。

font-lock-syntactic-face-function [Variable]
この変数が非 nilなら、それは与えられた構文的要素 (文字列かコメント) にどのフェイスを使用するかを決定する関数であること。

この関数は 1 つの引数で呼び出され、parse-partial-sexpがリターンするポイントの状態をパースしてフェイスをリターンすること。リターンされるデフォルト値はコメントにたいしては font-lock-comment-face、文字列にたいしては font-lock-string-face (Section 23.6.7 [Faces for Font Lock], page 504 を参照)。

この変数は通常は font-lock-defaults内の “他” の要素を通じてセットされる:

```
(setq-local font-lock-defaults
            `(,python-font-lock-keywords
              nil nil nil nil
              (font-lock-syntactic-face-function
               . python-font-lock-syntactic-face-function)))
```

23.6.9 複数行の Font Lock 構造

font-lock-keywordsの要素は、通常は複数行にわたるマッチを行うべきではありません。それらの動作に信頼性はありません。なぜなら Font Lock は通常はバッファーのごく一部をスキャンするので、そのスキャンが開始される行境界をまたがる複数行構造を見逃しかねないからです (スキャンは通常は行頭から開始される)。

ある要素にたいして複数行構造にたいするマッチを正しく機能させるために 2 つの観点があります。それは識別 *(identification)* の補正と、再ハイライト *(rehighlighting)* の補正です。1 つ目は Font Lock がすべての複数行構造を探すことを意味します。2 つ目は複数行構造が変更されたとき、たとえば以前は複数行構造の一部だったテキストが複数行構造から除外されたときに、関連するすべてのテキストを Font Lock に正しく再ハイライトさせることを意味します。これら 2 つの観点は密接に関連しており、一方を機能させることがもう一方を機能させるようなことが多々あります。しかし信頼性のある結果を得るためには、これら 2 つの観点双方にたいして明示的に注意しなければなりません。

複数行構造の識別を確実に補正するには 3 つの方法があります:

- スキャンされるテキストが複数行構造の途中で開始や終了することがないように識別を行ってスキャンを拡張する関数を font-lock-extend-region-functionsに追加する。
- 同様に、スキャンされるテキストが複数行構造の途中で開始や終了することがないようスキャンを拡張するために、font-lock-fontify-region-functionフックを使用する。
- 複数行構造がバッファーに挿入されたとき (または挿入後に Font Lock がハイライトを試みる前の任意のタイミングで)、何らかの方法によりそれを正しく認識して、Font Lock が複数行構造の途中で開始や終了しないように指示する font-lock-multilineでそれをマークする。

複数行構造の再ハイライトを行うには 3 つの方法があります:

- その構造にたいして正しく `font-lock-multiline`を配置する。これによりその構造の一部が変更されると構造全体が再ハイライトされるだろう。あるケースにおいてはそれを参照する`font-lock-multiline`変数をセットすることにより自動的にこれを行うことができる。
- `jit-lock-contextually`を確実にセットしてそれが行う処理に委ねる。これにより、実際の変更に続いて構造の一部だけが若干の遅延の後に再ハイライトされるだろう。これは複数行構造のさまざまな箇所のハイライトが後続行のテキストに依存しない場合のみ機能する。`jit-lock-contextually`はデフォルトでアクティブなので、これは魅力的な解決策になり得る。
- その構造上に正しく `jit-lock-defer-multiline`を配置する。これは `jit-lock-contextually`が使用された場合のみ機能し、再ハイライト前に同様の遅延を伴うが、`font-lock-multiline`のように後続行に依存する箇所のハイライトも処理する。

23.6.9.1 複数行の Font Lock

複数行構造の Font Lock を確実に再ハイライトする方法の 1 つは、それらをテキストプロパティ`font-lock-multiline`に put する方法です。複数行構造の一部であるようなテキストには値が非`nil`であるようなこのプロパティが存在するべきです。

Font Lock がテキスト範囲をハイライトしようとする際は、まずそれらが`font-lock-multiline`プロパティでマークされたテキストにならないように必要に応じて範囲の境界を拡張します。それからその範囲のすべての`font-lock-multiline`を削除してハイライトします。ハイライト指定 (大抵は`font-lock-keywords`) は、適宜このプロパティを毎回再インストールしなければなりません。

警告: ハイライトが低速になるので大きなテキスト範囲にたいして`font-lock-multiline`を使用してはならない。

`font-lock-multiline` [Variable]

`font-lock-multiline`変数が`t`にセットされていると Font Lock は自動的に複数行構造にたいして`font-lock-multiline`プロパティの追加を試みる。しかしこれにより Font Lock が幾分遅くなるので普遍的解決策ではない。これは何らかの複数行構造を見逃したり、必要なものより多く、または少なくプロパティをセットするかもしれない。

*matcher*が関数であるような要素は、たとえ少量のサブパート (subpart) だけがハイライトされるような場合でも、submatch 0(訳注:正規表現の後方参照において submatch 0 はマッチした文字列全体を指す) が関連する複数行構造全体を確実に網羅するようにすべきである。単に手動で`font-lock-multiline`を追加するのが容易な場合も多々ある。

`font-lock-multiline`プロパティは正しい再フォント表示を確実に行うことを意図しています。これは新たな複数行構造を自動的に認識しません。それらを認識するためには Font Lock が一度に十分な大きさの chunk を処理することを要求します。これは多くの場合にアクシデントにより発生し得るかもしれないので、複数行構造が不可解に機能するような印象を与えるかもしれません。変数`font-lock-multiline`を非`nil`にセットした場合には、発見されたこれらの構造にたいするハイライトは変数をセットした後は正しく更新されるので、さらにこの印象が強くなるでしょう。しかしこれは信頼性をもって機能しません。

信頼性を保ち複数行構造を見つけるためには、Font Lock が調べる前にテキストの`font-lock-multiline`プロパティを手動で配置するか、`font-lock-fontify-region-function`を使用しなければなりません。

23.6.9.2 バッファー変更後のリージョンのフォント化

バッファーが変更されたとき Font Lock が再フォント表示するリージョンは、デフォルトではその変更に関連する最小の行全体からなるシーケンスです。これはほとんどの場合は良好に機能しますが、うまく機能しないとき (たとえば変更がそれより前の行のテキストの構文的な意味を変更してしまうとき) もあります。

以下の変数をセットすることにより、再フォント表示するリージョンを拡張 (または縮小さえ) することができます:

`font-lock-extend-after-change-region-function` [Variable]

このバッファーローカル変数は `nil`、または Font Lock モードにたいしてスキャンしてフォント表示すべきリージョンを決定するために呼び出される関数である。

この関数には標準的な *beg*と *end*、および `after-change-functions`の *old-len* (Section 32.32 [Change Hooks], page 850 を参照) という 3 つのパラメーターが渡される。この関数はフォント表示するリージョンのバッファー位置の開始と終了 (この順) からなるコンスセル、または `nil` (標準的な方法でリージョンを選択することを意味する) のいずれかをリターンすること。この関数はポイント位置、match-data、カレントのナローイングを保つ必要がある。これがリターンするリージョンは、行の途中で開始や終了するかもしれない。

この関数はバッファーを変更するたびに呼び出されるので有意に高速であること。

23.7 コードの自動インデント

プログラミング言語のメジャーモードにとって、自動的なインデントの提供は重要な機能です。これには 2 つのパートがあります。1 つ目は正しい行のインデントが何か、そして 2 つ目はいつ行を再インデントするかの判断です。デフォルトでは `electric-indent-chars`に含まれる文字 (デフォルトでは改行のみ) をタイプしたとき、Emacs は常に行を再インデントします。メジャーモードはその言語の構文に合わせて `electric-indent-chars`に文字を追加できます。

正しいインデントの決定は `indent-line-function`により Emacs 内で制御されます (Section 32.17.2 [Mode-Specific Indent], page 807 を参照)。いくつかのモードでは右へのインデントは信頼性がないことが知られています。これは通常は複数のインデントが有効であり、それぞれが異なる意味をもつのでインデント自体が重要だからです。そのような場合には、そのモードは行が常にユーザーの意に反して行が毎回再インデントされないことを保証するために `electric-indent-inhibit`をセットするべきです。

よいインデント関数の記述は難しく、その広範な領域において未だ黒魔術の域を脱していません。メジャーモード作者の多くは、単純なケース (たとえば前のテキスト行のインデントとの比較) にたいして機能する、単純な関数の記述からスタートすることでしょう。実際には行ベースではないほとんどのプログラミング言語にたいして、これは貧弱なスケールになりがちです。そのような関数にたいして、より多様な状況を処理するような改良を行うと関数はより一層複雑になり、最終的な結果は誰にも触れようとする気を起こさせない、巨大で複雑な保守不可能のインデント関数になる傾向があります。

よいインデント関数は、通常はその言語の構文に応じて実際にテキストをパースする必要があるでしょう。幸運なことにこのテキストパースはコンパイラーが要するほど詳細である必要はないでしょうが、その一方でインデントコードに埋め込まれたパーサーは構文的に不正なコードにたいして、コンパイラーより幾分寛容な振る舞いを求められるでしょう。

保守可能なよいインデント関数は、通常は 2 つのカテゴリーに落ち着きます。どちらも何らかの安全な開始ポイントから、関心のある位置まで前方か後方へパースを行います。この 2 つの方法は、いずれも一方が他方に明快に優る選択ではありません。後方へのパースはプログラミング言語が前方に

パースされるようデザインされているため、前方へのパースに比べて難しいことが多々ありますが、インデントという目的においては安全な開始ポイントを推測する必要がないという利点があり、一般的にある行のインデントの判断のために分析を要するのは最小限のテキストだけという特性に恵まれているので、前の無関係なコード片内にある何らかの構文エラーの影響をインデントが受けにくくなる傾向があります。一方で前方へのパースは通常はより簡単であり、一度のパースでリージョン全体を効果的に再インデントすることが可能になるという利点があります。

インデント関数をスクラッチから記述するよりも、既存のインデント関数の使用と再利用、または一般的なインデントエンジンに委ねるほうが優る場合がしばしばあります。しかしそのようなエンジンは悲しむべきほど少数しかありません。(C、C++、Java、Awk、およびその類のモードに使用される)CC モードのインデントコードは年月を経てより一般化されてきているので、あなたの言語にこれらの言語と何らかの類似点があるなら、このエンジンの使用を試みるかもしれません。もう一方の SMIE は Lisp の sexp 精神によるアプローチを採用して、それを非 Lisp 言語に適応します。

23.7.1 SMIE: 無邪気なインデントエンジン

SMIE は一般的な操作とインデントを提供するエンジンです。これは演算子順位文法 (operator precedence grammar) を使用する非常にシンプルなパーサーにもとづいたエンジンであり、メジャーモードが Lisp の S 式ベースの操作を非 Lisp 言語に拡張するのを助けるとともにシンプルに使用できるにも関わらず、信頼できる自動インデントを提供します。

演算子順位文法はコンパイラー内で使用されるより一般的なパーサーと比較すると非常に原始的なパーステクノロジーです。このパーサーには次のような特徴があります。このパーサーのパース能力は非常に限定的で構文エラーを大概は検出できません。しかしアルゴリズム的に前方パースと同様に後方パースを効果的に行うことが可能です。実際にそれは SMIE が後方パースにもとづくインデントを使用でき、`forward-sexp`と `backward-sexp`の両方の機能を提供できるとともに、特別な努力を要さずに構文的に不正なコードにたいして自然に機能するであろうことを意味します。欠点はほとんどのプログラミング言語は、少なくとも何らかの特別なトリック (Section 23.7.1.5 [SMIE Tricks], page 513 を参照) で再分類しなければ SMIE を使用して正しくパースできないことをも意味することです。

23.7.1.1 SMIE のセットアップと機能

SMIE は構造的な操作とコードの構造的構造にもとづくその他さまざまな機能、特に自動インデントにたいするワンストップショップ (一カ所で必要な全ての買い物ができる店やそのような場所) であることを意図しています。メインのエントリーポイントは `smie-setup`で、これは通常はメジャーモードセットアップの間に呼び出される関数です。

`smie-setup` *grammar rules-function* **&rest** *keywords* [Function]
SMIE の操作とインデントをセットアップする。*grammar*は `smie-prec2->grammar`により生成される文法テーブル (grammar table)、*rules-function*は `smie-rules-function`で使用されるインデントルールのセット、*keywords*は追加の引数であり以下のキーワードを含むことができる:

- `:forward-token` *fun*: 使用する前方 lexer(lexer=lexical analyzer: 字句解析プログラム) を指定する。
- `:backward-token` *fun*: 使用する後方 lexer を指定する。

この関数を呼び出せば`forward-sexp`、`backward-sexp`、`transpose-sexps`のようなコマンドが、すでに構文テーブルにより処理されている単なるカッコのペア以外の、構造的な要素を正しく扱うことができるようになります。たとえば与えられた文法が十分に明快ならば、`transpose-sexps`はその言語の優先順位のルールを考慮して+演算子の 2 つの引数を正しく入れ替えることができます。

`smie-setup`の呼び出しは `begin...end`のような要素に適用するために `blink-matching-paren`を拡張して `TAB`によるインデントを期待通り機能させるとともに、メジャーモードのキーマップ内でバインドできるいくつかのコマンドの提供を満足します。

`smie-close-block` [Command]
このコマンドは、もっとも最近オープンされた (まだクローズされていない) ブロックをクローズする。

`smie-down-list` **&optional** *arg* [Command]
このコマンドは `down-list`と似ているが、`begin...end`のようなカッコ以外のネストされたトークンにも注意を払う。

23.7.1.2 演算子順位文法

SMIE の演算子順位文法は、各トークンにたいしてシンプルに左優先 (left-precedence) と右優先 (right-precedence) という順位ペアを与えます。トークン `T1`の右優先がトークン `T2`の左優先より小さければ `T1 < T2`であると言うことにしましょう。これを解読するには`<`をカッコの一種だとみなすのがよい方法です。`... T1 something T2 ...`を見つけたら、これは`... T1 something) T2 ...`ではなく`... T1 (something T2 ...`とパースされるべきです。`... T1 something) T2 ...`と解釈するのは `T1 > T2`を見つけた場合でしょう。`T1 = T2`を見つけた場合、それはトークン T2 とその後のトークン T1 が同じ構文にあり、通常は`"begin" = "end"`を得ます。このような優先順位のペアは 2 項演算子 (infix operator)、カッコのようなネストされたトークン、およびその他多くのケースにたいして左結合 (left-associativity) や右結合 (right-associativity) を表現するのに十分です。

`smie-prec2->grammar` *table* [Function]
この関数は *prec2* 文法 *table*を引数に受け取り、`smie-setup`で使用するのに適した alist をリターンする。*prec2* 文法 *table*は、それ自体が以下の関数のいずれかによりビルドされることを意図している。

`smie-merge-prec2s` **&rest** *tables* [Function]
この関数は複数の *prec2* 文法 *tables*を、新たな *prec2* テーブルにマージする。

`smie-precs->prec2` *precs* [Function]
この関数は順位テーブル *precs*から *prec2* テーブルをビルドする。*precs*は優先順 (たとえば `"+"`は`"*"`より前にくる) にソートされたリストであり、要素は `(assoc op ...)`の形式であること。ここで *op*は演算子として振る舞うトークン、*assoc*はそれらの結合法則であり `left`、`right`、`assoc`、`nonassoc`のいずれかである。与えられた要素内のすべての演算子は同じ優先レベルと結合法則を共有する。

`smie-bnf->prec2` *bnf* **&rest** *resolvers* [Function]
この関数により BNF 記法を使用した文法を指定することができる。これはその文法の *bnf* 表記と、同様に競合解決ルール *resolvers*を受け取って *prec2* テーブルをリターンする。

*bnf*は `(nonterm rhs1 rhs2 ...)`という形式の非終端定義、各 *rhs*は終端記号 (トークンとも呼ばれる)、または非終端記号の (空でない) リストである。

すべての文法が許容される訳ではない:

- *rhs*に空のリストは指定できない (いずれにせよ SMIE は空文字列にマッチさせるためにすべての非終端記号を許容するので空リストが必要になることは決してない)。
- *rhs*の後に連続する 2 つの非終端記号は指定できない。非終端記号の各ペアは終端記号 (かトークン) で区切られる必要がある。これは演算子順位文法の基本的な制約である。

さらに競合が発生し得る:

- リターンされる *prec2* テーブルはトークンのペア間の制約を保持し、与えられた任意のペアは T1 < T2、T1 = T2、T1 > T2 のいずれかのうち 1 つの制約をだけ与えることができる。
- トークンは `opener`(開カッコに似た何か)、`closer`(閉カッコのようなもの)、またはこれら 2 つのいずれでもない `neither`(2 項演算子や`"else"`のような inner トークン) である。

順位の競合は *resolvers*を通じて解決され得る。これは *precs* テーブル (`smie-precs->prec2` を参照) のリストである。それぞれの順位競合にたいして、これらの `precs`テーブルが特定の制約を指定している場合は、かわりにこの制約により競合が解決され、それ以外は競合する制約のうち任意の 1 つが報告されて他は単に無視される。

23.7.1.3 言語の文法の定義

ある言語にたいして SMIE 文法を定義する通常の方法は、順位のテーブルを保持する新たなグローバル変数を定義して BNF ルールのセットを与える方法です。たとえば小規模な Pascal 風言語の文法定義は以下のようになるでしょう:

```
(require 'smie)
(defvar sample-smie-grammar
  (smie-prec2->grammar
   (smie-bnf->prec2
    '((id)
      (inst ("begin" insts "end")
            ("if" exp "then" inst "else" inst)
            (id ":=" exp)
            (exp))
      (insts (insts ";" insts) (inst))
      (exp (exp "+" exp)
           (exp "*" exp)
           ("(" exps ")"))
      (exps (exps "," exps) (exp)))
    '((assoc ";"))
    '((assoc ","))
    '((assoc "+") (assoc "*")))))
```

注意すべき点がいくつかあります:

- 上記の文法は関数呼び出しの構文に明示的に言及していない。SMIE は識別子、対応がとれたカッコ (balanced parentheses)、または `begin ... end`ブロックのような sexp の任意のシーケンスがどこに、どのように出現しても自動的にそれを許容するだろう。
- 文法カテゴリー `id`は右側に何ももたない。これは `id`が空文字列だけにマッチ可能なことを意味しない。なぜなら上述のように任意の sexp シーケンスはどこに、どのような方法でも出現するからである。
- BNF 文法では非終端記号が連続して出現し得ないので、終端記号として作用するトークンを正しく扱うのが困難なため、上述の文法では SMIE が容易に扱える`";"`をセパレーター *(separator)* ステートメントのかわりとして扱っている。
- シーケンス内で使用される、(上記の`","`や`";"`のような) セパレーターは、BNF ルールでは `(foo (foo "separator" foo) ...)`のように定義するのが最善である。これは順位の競合を生成するが、明示的に `(assoc "separator")`を与えることにより解決される、

- SMIE は構文テーブル (syntax table) 内でカッコ構文 (paren syntax) をもつようにマークされた任意の文字をペアにするだろうから、`("(" exps ")")`ルールにカッコをペアにする必要はなかった。(`exps`の定義と併せて) これはかわりに`","`がカッコの外に出現すべきではないことを明確にするためのルール。
- 競合解決のための *precs* テーブルは単一のテーブルより複数のテーブルをもつほうが、可能な場合は文法の BNF 部分が関連する順位を指定できるので優れている。
- `left`や `right`を選択することが優るという明白な理由がなければ、通常は `assoc`を使用して演算子を結合演算子 (associative) とマークするほうが優れている。この理由により上述の`"+"`と`"*"`は、たとえその言語がそれらを形式上は左結合 (left associative) と定義していても `assoc`として定義されている。

23.7.1.4 トークンの定義

SMIE には事前定義された字句解析プログラムが付属しており、それは次の方法で構文テーブルを使用します: 文字の任意のシーケンスはトークンとみなせる単語構文 (word syntax) かシンボル構文 (symbol syntax) をもち、区切り文字構文 (punctuation syntax) をもつ任意の文字シーケンスもトークンとみなされます。このデフォルトの lexer は開始ポイントとして適している場合が多々ありますが、任意の与えられた言語にたいして実際に正しいことは稀です。たとえばこれは`"2,+3"`が 3 つのトークン`"2"`、`",+"`、`"3"`から構成されていると判断するでしょう。

あなたの言語の lexer ルールを SMIE にたいして説明するためには、次のトークンを fetch する関数と前のトークンを fetch する関数という 2 つの関数が必要になります。これらの関数は通常は最初に空白文字とコメントをスキップして、その後に次のテキスト chunk(塊) を調べてそれが特別なトークンか確認します。これは通常は単にバッファーから抽出された文字列ですが、あなたが望む他の何かでも構いません。たとえば:

```
(defvar sample-keywords-regexp
  (regexp-opt '("+" "*" "," ";" ">" ">=" "<" "<=" ":=" "=")))
(defun sample-smie-forward-token ()
  (forward-comment (point-max))
  (cond
   ((looking-at sample-keywords-regexp)
    (goto-char (match-end 0))
    (match-string-no-properties 0))
   (t (buffer-substring-no-properties
       (point)
       (progn (skip-syntax-forward "w_")
              (point))))))
(defun sample-smie-backward-token ()
  (forward-comment (- (point)))
  (cond
   ((looking-back sample-keywords-regexp (- (point) 2) t)
    (goto-char (match-beginning 0))
    (match-string-no-properties 0))
   (t (buffer-substring-no-properties
       (point)
       (progn (skip-syntax-backward "w_")
              (point))))))
```

これらの lexer がカッコの前にあるとき空文字列をリターンする方法に注目してください。これは SMIE が構文テーブル内で定義されているカッコにたいして自動的に配慮するからです。より厳密には lexer が `nil`、または空文字列をリターンしたら、SMIE は構文テーブルにしたがって対応するテキストを sexp として処理します。

23.7.1.5 非力なパーサーと歩む

SMIE が使用するパーステクニックは、異なるコンテキストでトークンが異なる振る舞いをすることを許容しません。ほとんどのプログラミング言語にたいして、これは順位の競合により BNF 文法を変換するとき明らかになります。

その文法を若干異なるように表現することにより、これらの競合を回避できる場合があります。たとえば Modula-2 にたいしては以下のような BNF 文法をもつことが自然に思えるかもしれません:

```
...
(inst ("IF" exp "THEN" insts "ELSE" insts "END")
      ("CASE" exp "OF" cases "END")
      ...)
(cases (cases "|" cases)
       (caselabel ":" insts)
       ("ELSE" insts))
...
```

しかしこれは`"ELSE"`にたいする競合を生み出すでしょう。その一方で IF ルールは、(他の多くのものの中でも特に)`"ELSE" = "END"`を暗示します。しかしその一方で`"ELSE"`は `cases`内に出現しますが、`cases`は`"END"`の左に出現するので、わたしたちは`"ELSE" > "END"`も得ることになります。これは以下を使用して解決できます:

```
...
(inst ("IF" exp "THEN" insts "ELSE" insts "END")
      ("CASE" exp "OF" cases "END")
      ("CASE" exp "OF" cases "ELSE" insts "END")
      ...)
(cases (cases "|" cases) (caselabel ":" insts))
...
```

または

```
...
(inst ("IF" exp "THEN" else "END")
      ("CASE" exp "OF" cases "END")
      ...)
(else (insts "ELSE" insts))
(cases (cases "|" cases) (caselabel ":" insts) (else))
...
```

文法書き換えによる競合の解決には欠点があります。なぜなら SMIE はその文法がコードの論理的構造を反映すると仮定するからです。そのため BNF と意図する抽象的構文木の関係を密接に保つことが望まれます。

注意深く考慮した結果、これらの競合が深刻ではなく、`smie-bnf->prec2`の *resolvers*引数を通じて解決する決心をする場合もあるでしょう。これは通常はその文法が単に不明瞭だからです。その文法により記述されるプログラムセットは競合の影響を受けませんが、それらのプログラムにたいする唯一の方法はパースだけです。`'((assoc "|"))`のようなリゾルバ (resolver: 解決するもの) を追

加したいと望むような場合、通常それはセパレーターと 2 項結合演算子にたいするケースです。これが発生し得る他のケースは`'((assoc "else" "then"))`を使用するような場合における、古典的なぶら下がり *else* 問題 *(dangling else problem)* です。これは実際に競合があり解決不能なものの、実際のところ問題が発生しそうにないケースにたいしても発生し得ます。

最後に多くのケースではすべての文法再構築努力にも関わらず、いくつかの競合が残るでしょう。しかし失望しないでください。パーサーをより賢くすることはできませんが、あなたの望むように lexer をスマートにすることは可能です。その方法は競合が発生したら競合を引き起こしたトークンを調べて、それらのうちの 1 つを 2 つ以上の異なるトークンに分割する方法です。たとえばトークン`"begin"`にたいする互換性のない 2 つの使用を文法が区別する必要があり、見つかった`"begin"`の種類によって lexer に異なるトークン (たとえば`"begin-fun"`と`"begin-plain"`) をリターンさせる場合です。これは lexer にたいして異なるケースを区別する処理を強制し、そのために lexer は特別な手がかりを見つけるために周囲のテキストを調べる必要があるでしょう。

23.7.1.6 インデントルールの指定

提供された文法にもとづき、他に特別なことを行わなくても SMIE は自動的なインデントを提供できるでしょう。しかし恐らく実際にはこのデフォルトのインデントスタイルでは十分ではありません。多くの異なる状況においてこれを微調整したいと思うかもしれません。

SMIE のインデントは、インデントルールは可能な限りローカルであるべきという考えにもとづきます。バーチャルインデント *(virtual indentation)* という考えによってこの目的を達成しています。これは特定のプログラムポイント (program point) は行頭にバーチャルインデントがあれば、それをもつだろう、という発想です。もちろんそのプログラムポイントが正に行頭にあれば、そのプログラムポイントのバーチャルインデントはプログラムポイントのカレントのインデントです。しかしそうでなければ SMIE がそのポイントのバーチャルインデントを計算するためにインデントアルゴリズムを使用します。ところで実際にはあるプログラムポイントのバーチャルインデントは、その前に改行を挿入した場合にプログラムポイントがもつであろうインデントと等しい必要はありません。これが機能する方法を確認するためには、C における`{`の後の SMIE のインデントルールは`{`がインデントする行自体にあるか、あるいは前の行の終端にあるかを配慮しないことが挙げられます。かわりにこれらの異なるケースは`{`の前のインデントを決定するインデントルール内で処理されます。

他の重要な考え方として *parent* の概念があります。あるトークン *parent* は周囲にある直近の構文構造の代表トークン (head token) です。たとえば`else`の parent はそれが属する`if`であり、`if`の parent は周囲を取り囲む構造の先導トークン (lead token) です。コマンド`backward-sexp`は、あるトークンからトークンの parent にジャンプしますが注意する点がいくつかあります。他のトークンではそのトークンの後のポイントから開始する必要があるのにたいして、*opener* (`if`のようなある構造を開始するトークン) ではそのトークンの前のポイントから開始する必要があります。`backward-sexp`は parent トークンがそのトークンの *opener* なら parent トークンの前のポイントで停止し、それ以外では parent トークンの後のポイントで停止します。

SMIE のインデントルールは、2 つの引数 *method* と *arg* を受け取る関数により指定されます。ここで *arg* の値と期待されるリターン値は *method* に依存します。

method には以下のいずれかを指定できます:

- `:after`: この場合、*arg* はトークンであり関数は *arg* の後に使用するインデントにたいする *offset* をリターンすること。
- `:before`: この場合、*arg* はトークンであり関数は *arg* 自体に使用するインデントの *offset* をリターンすること。

- `:elem`: この場合、関数は関数の引数に使用するインデントのオフセット (*arg*がシンボル `arg`の場合)、または基本的ナインデントステップ (*arg*がシンボル `basic`の場合) のいずれかをリターンすること。
- `:list-intro`: この場合、*arg*はトークンであり関数はそのトークンの後が単一の式ではなく、(任意のトークンにより区切られない) 式のリストが続くなら非 `nil`をリターンすること。

*arg*がトークンのとき関数はそのトークンの直前のポイントで呼び出されます。リターン値 `nil`は常にデフォルトの振る舞いへのフォールバックを意味するので、関数は期待した引数でないときは `nil`をリターンするべきです。

*offset*には以下のいずれかを指定できます:

- `nil`: デフォルトのインデントルールを使用する。
- `(column . column)`: 列 *column*にインデントする。
- *number*: 基本トークン (base token: `:after`にたいするカレントトークンであり、かつ`:before`にたいして parent であるようなトークン) にたいして相対的な *number*によるオフセット。

23.7.1.7 インデントルールにたいするヘルパー関数

SMIE はインデントを決定する関数内で使用するために特別にデザインされたさまざまな関数を提供します (これらの関数のうちのいくつかは異なるコンテキスト内で使用された場合に中断する)。これらの関数はすべてプレフィックス `smie-rule-`で始まります。

`smie-rule-bolp` [Function]
カレントトークンが行の先頭にあれば非 `nil`をリターンする。

`smie-rule-hanging-p` [Function]
カレントトークンが *hanging(*ぶら下がり*)* なら非 `nil`をリターンする。トークンがその行の最後のトークンであり、他のトークンが先行する場合、そのトークンは *hanging* である。行に単独のトークンは hanging ではない。

`smie-rule-next-p` **&rest** *tokens* [Function]
次のトークンが *tokens*内にあれば非 `nil`をリターンする。

`smie-rule-prev-p` **&rest** *tokens* [Function]
前のトークンが *tokens*内にあれば非 `nil`をリターンする。

`smie-rule-parent-p` **&rest** *parents* [Function]
カレントトークンの parent が *parents*内にあれば非 `nil`をリターンする。

`smie-rule-sibling-p` [Function]
カレントトークンの parent が実際は sibling(兄弟) なら非 `nil`をリターンする。たとえば`","`の parent が直前の`","`のような場合が該当。

`smie-rule-parent` **&optional** *offset* [Function]
カレントトークンを parent とアライン (align: 桁揃え) するための適切なオフセットをリターンする。*offset*が非 `nil`なら、それは追加オフセットとして適用される整数であること。

`smie-rule-separator` *method* [Function]
セパレーター *(separator)* としてカレントトークンをインデントする。

ここでのセパレーターとは周囲を取り囲む何らかの構文構造内でさまざまな要素を区切ることを唯一の目的とするトークンであり、それ自体は何も意味をもたないトークン (通常は抽象構文木内でノードとして存在しないこと) を意味する。

このようなトークンは結合構文をもち、その構文的 parent と密に結び付けられることが期待される。典型的な例としては引数リスト内の`","` (カッコで括られた内部)、または命令文シーケンス内の`";"` (`{...}`や `begin...end`で括られたブロックの内部) が挙げられる。

*method*は `smie-rules-function`に渡されるメソッド名であること。

23.7.1.8 インデントルールの例

以下はインデント関数の例です:

```
(defun sample-smie-rules (kind token)
  (pcase (cons kind token)
    (`(:elem . basic) sample-indent-basic)
    (`(,_ . ",") (smie-rule-separator kind))
    (`(:after . ":=") sample-indent-basic)
    (`(:before . ,(or `"begin" `"(" `"{"))
     (if (smie-rule-hanging-p) (smie-rule-parent)))
    (`(:before . "if")
     (and (not (smie-rule-bolp)) (smie-rule-prev-p "else")
          (smie-rule-parent)))))
```

注意すべき点がいくつかあります:

- 最初の case は使用する基本的なインデントの増分を示す。`sample-indent-basic`が `nil`なら、SMIE はグローバルセッティング `smie-indent-basic`を使用する。メジャーモードがかわりに `smie-indent-basic`をバッファーローカルにセットするかもしれないが推奨しない。
- トークン`","`にたいするルールによってカンマセパレーターが行頭にある場合に SMIE をより賢明に振る舞わせようとしている。これはセパレーターのインデントを解除 (outdent)、カンマの後のコードにアラインされるよう試みる。たとえば:

```
x = longfunctionname (
        arg1
      , arg2
    );
```

- そうしなければ SMIE が`":="`を 2 項演算子として扱い、左の引数に併せて右の引数をアラインするであろうから、`":="`の後のインデントのルールが存在する。
- `"begin"`の前のインデントのルールはバーチャルインデントの使用例である。このルールは`"begin"`が hanging のときだけ使用され、これは`"begin"`が行頭にないときのみ発生し得る。そのためこれは`"begin"`自体のインデントには使用されないが、この`"begin"`に関連する何かをインデントするときだけ使用される。このルールは具体的には以下のフォームを:

```
if x > 0 then begin
        dosomething(x);
    end
```

以下に変更する

```
if x > 0 then begin
    dosomething(x);
end
```

- "if"の前のインデントのルールは"begin"のインデントルールと似ているが、ここでの目的は"else if"を 1 単位として扱うことにあり、それにより各テストより右にインデントされずに一連のテストにアラインされる。この関数は smie-rule-bolpをテストして"if"が別の行にないときだけこれを行う。

  "else"がそれの属する"if"にたいして常にアラインされて、かつそれが常に行頭であるることが判っていれば、より効果的なルールを使用できる:

```
((equal token "if")
 (and (not (smie-rule-bolp))
      (smie-rule-prev-p "else")
      (save-excursion
        (sample-smie-backward-token)
        (cons 'column (current-column)))))
```

  この式の利点はこれがシーケンスの最初の"if"まで戻ってすべてをやり直すのではなく、前の"else"のインデントを再利用することである。

23.7.1.9 インデントのカスタマイズ

SMIE により提供されるインデントを使用するモードを使っている場合には、好みに合わせてインデントをカスタマイズできます。これはモードごと (オプション smie-configを使用)、またはファイルごと (ファイルローカル変数指定内で関数 smie-config-localを使用) に行うことができます。

smie-config [User Option]
: このオプションによりモードごとにインデントをカスタマイズできる。これは (*mode* . *rules*) という形式の要素をもつ alist。rules の正確な形式については変数のドキュメントを参照のこと。しかしコマンド smie-config-guessを使用したほうが、より簡単に見つけられるかもしれない。

smie-config-guess [Command]
: このコマンドは好みのスタイルのインデントを生成する適切セッティングの解決を試みる。あなたのスタイルでインデントされたファイルを visit しているときに単にこのコマンドを呼び出せばよい。

smie-config-save [Command]
: smie-config-guessを使用した後にこのコマンドを呼び出すと将来のセッション用にセッティングを保存する。

smie-config-show-indent **&optional** *move* [Command]
: このコマンドはカレント行のインデントに使用されているルールを表示する。

smie-config-set-indent [Command]
: このコマンドはカレント行のインデントに合わせてローカルルールを追加する。

smie-config-local *rules* [Function]
: この関数はカレントバッファーにたいするインデントルールとして *rules*を追加する。これらのルールは smie-configオプションにより定義された任意のモード固有ルールに追加される。特定のファイルにたいしてカスタムインデントルールを指定するには、eval: (smie-config-local '(*rules*))の形式のエントリーをそのファイルのローカル変数に追加する。

23.8 Desktop Save モード

Desktop Save モードとは、あるセッションから別のセッションへ Emacs 状態を保存する機能です。Desktop Save モードの使用に関するユーザーレベルのコマンドについては、GNU Emacs マニュアルに記載されています (Section “Saving Emacs Sessions” in *the GNU Emacs Manual* を参照)。バッファーでファイルを visit しているモードでは、この機能を使うために何も行う必要はありません。

ファイルを visit していないバッファーについて状態を保存するには、そのメジャーモードがバッファーローカル変数 `desktop-save-buffer`を非 `nil`値にバインドしなければなりません。

`desktop-save-buffer` [Variable]

このバッファーローカル変数が非 `nil`なら、デスクトップ保存時にそのバッファー状態が desktop ファイルに保存される。値が関数なら、その関数はデスクトップ保存時に引数 *desktop-dirname* で呼び出されて、関数が呼び出されたバッファーの状態とともに関数の値が desktop ファイルに保存される。補助的な情報の一部としてファイル名がリターンされたとき、それらは以下を呼び出してフォーマットされること

```
(desktop-file-name file-name desktop-dirname)
```

ファイルを visit していないバッファーがリストアされるようにするには、メジャーモードがその処理を行う関数を定義しなければならず、その関数は連想配列 `desktop-buffer-mode-handlers` にリストされなければならない。

`desktop-buffer-mode-handlers` [Variable]

以下を要素にもつ alist

```
(major-mode . restore-buffer-function)
```

関数 *restore-buffer-function*は以下の引数リストで呼び出される

```
(buffer-file-name buffer-name desktop-buffer-misc)
```

この関数はリストアされたバッファーをリターンすること。ここで *desktop-buffer-misc*は、オプションで `desktop-save-buffer`にバインドされる関数がリターンする値。

24 ドキュメント

GNU Emacs には便利なビルトインのヘルプ機能があり、それらのほとんどは関数や変数のドキュメント文字列に付属するドキュメント文字列の情報が由来のものです。このチャプターでは Lisp プログラムからドキュメント文字列にアクセスする方法について説明します。

ドキュメント文字列のコンテンツはある種の慣習にしたがう必要があります。特に最初の行はその関数や変数を簡単に説明する 1 つか 2 つの完全なセンテンスであるべきです。よいドキュメント文字列を記述する方法については Section D.6 [Documentation Tips], page 1168 を参照してください。

Emacs 向けのドキュメント文字列は Emacs マニュアルと同じものではないことに注意してください。マニュアルは Texinfo 言語で記述された独自のソースファイルをもちます。それにたいしドキュメント文字列はそれが適用される関数と変数の定義内で指定されたものです。ドキュメント文字列を収集してもそれはマニュアルとしては不十分です。なぜならよいマニュアルはそのやり方でまとめられたものではなく、議論にたいするトピックという観点でまとめられたものだからです。

ドキュメント文字列を表示するコマンドについては、Section “Help” in *The GNU Emacs Manual* を参照してください。

24.1 ドキュメントの基礎

ドキュメント文字列はテキストをダブルクォート文字で囲んだ文字列にたいする Lisp 構文を使用して記述されます。実はこれは実際の Lisp 文字列です。関数または変数の定義内の適切な箇所に文字列があると、それは関数や変数のドキュメントの役割を果たします。

関数定義 (`lambda`や `defun`フォーム) の中では、ドキュメント文字列は引数リストの後に指定され、通常は関数オブジェクト内に直接格納されます。Section 13.2.4 [Function Documentation], page 213 を参照してください。関数名の `function-documentation`プロパティに関数ドキュメントを put することもできます (Section 24.2 [Accessing Documentation], page 520 を参照)。

変数定義 (`defvar`フォーム) の中では、ドキュメント文字列は初期値の後に指定されます。Section 12.5 [Defining Variables], page 178 を参照してください。この文字列はその変数の `variable-documentation`プロパティに格納されます。

Emacs がメモリー内にドキュメント文字列を保持しないときがあります。それには、これには 2 つの状況があります。1 つ目はメモリーを節約するためで、事前ロードされた関数と変数 (プリミティブを含む) のドキュメントは、`doc-directory`で指定されたディレクトリー内の `DOC`という名前のファイルに保持されます (Section 24.2 [Accessing Documentation], page 520 を参照)。2 つ目は関数や変数がバイトコンパイルされたファイルからロードされたときで、Emacs はそれらのドキュメント文字列のロードを無効にします (Section 17.3 [Docs and Compilation], page 288 を参照)。どちらの場合も、ある関数にたいしてユーザーが *`C-h f`*(`describe-function`) を呼び出したとき等の必要なときだけ Emacs はファイルのドキュメント文字列を照会します。

ドキュメント文字列にはユーザーがドキュメントを閲覧するときのみルックアップされるキーバインディングを参照する、特別なキー置換シーケンス *(key substitution sequences)* を含めることができます。これにより、たとえユーザーがデフォルトのキーバインディングを変更していてもヘルプコマンドが正しいキーを表示できるようになります。

オートロードされたコマンド (Section 16.5 [Autoload], page 274 を参照) のドキュメント文字列ではこれらのキー置換シーケンスは特別な効果をもち、そのコマンドにたいする *`C-h f`*によってオートロードをトリガーします (これは`*Help*`バッファー内のハイパーリンクを正しくセットアップするために必要となる)。

24.2 ドキュメント文字列へのアクセス

`documentation-property` *symbol property* **&optional** *verbatim* [Function]

この関数はプロパティ*property*配下の*symbol*のプロパティリスト内に記録されたドキュメント文字列をリターンする。これはほとんどの場合 *property*を `variable-documentation`にして、変数のドキュメント文字列の照会に使用される。しかしカスタマイゼーショングループのような他の種類のドキュメント照会にも使用できる (が関数のドキュメントには以下の `documentation`関数を使用する)。

そのプロパティの値が `DOC`ファイルやバイトコンパイル済みファイルに格納されたドキュメント文字列を参照する場合、この関数はその文字列を照会してそれをリターンする。

プロパティの値が `nil`や文字列以外でファイル内のテキストも参照しなければ、文字列を取得する Lisp 式として評価される。

最終的にこの関数はキーバインディングを置換するために、文字列を `substitute-command-keys`に引き渡す (Section 24.3 [Keys in Documentation], page 522 を参照)。*verbatim*が非 `nil`ならこのステップはスキップされる。

```
(documentation-property 'command-line-processed
   'variable-documentation)
     ⇒ "Non-nil once command line has been processed"
(symbol-plist 'command-line-processed)
     ⇒ (variable-documentation 188902)
(documentation-property 'emacs 'group-documentation)
     ⇒ "Customization of the One True Editor."
```

`documentation` *function* **&optional** *verbatim* [Function]

この関数は *function*のドキュメント文字列をリターンする。この関数はマクロ、名前付きキーボードマクロ、およびスペシャルフォームも通常の関数と同様に処理する。

*function*がシンボルならそのシンボルの `function-documentation`プロパティを最初に調べる。それが非 `nil`値をもつなら、その値 (プロパティの値が文字列以外ならそれを評価した値) がドキュメントとなる。

*function*がシンボル以外、あるいは `function-documentation`プロパティをもたなければ、`documentation`は必要ならファイルを読み込んで実際の関数定義のドキュメント文字列を抽出する。

最後に *verbatim*が `nil`なら、この関数は `substitute-command-keys`を呼び出す。結果はリターンするための文字列。

`documentation`関数は *function*が関数定義をもたなければ `void-function`エラーをシグナルする。しかし関数定義がドキュメントをもたない場合は問題ない。その場合は `documentation`は `nil`をリターンする。

`face-documentation` *face* [Function]

この関数は *face*のドキュメント文字列をフェイスとしてリターンする。

以下は `documentation`と `documentation-property`を使用した例で、いくつかのシンボルのドキュメント文字列を`*Help*`バッファー内に表示します。

```
(defun describe-symbols (pattern)
  "Describe the Emacs Lisp symbols matching PATTERN.
All symbols that have PATTERN in their name are described
in the *Help* buffer."
  (interactive "sDescribe symbols matching: ")
  (let ((describe-func
         (lambda (s)
```

```
            ;; シンボルの説明をプリントする
            (if (fboundp s)             ; これは関数
                (princ
                 (format "%s\t%s\n%s\n\n" s
                   (if (commandp s)
                       (let ((keys (where-is-internal s)))
                         (if keys
                             (concat
                              "Keys: "
                              (mapconcat 'key-description
                                         keys " "))
                           "Keys: none"))
                     "Function")
                   (or (documentation s)
                       "not documented"))))

           (if (boundp s)               ; これは変数
               (princ
                (format "%s\t%s\n%s\n\n" s
                  (if (custom-variable-p s)
                      "Option " "Variable")
                  (or (documentation-property
                       s 'variable-documentation)
                      "not documented"))))))
       sym-list)

     ;; パターンにマッチするシンボルのリストを構築
     (mapatoms (lambda (sym)
                 (if (string-match pattern (symbol-name sym))
                     (setq sym-list (cons sym sym-list)))))

     ;; データを表示
     (help-setup-xref (list 'describe-symbols pattern) (interactive-p))
     (with-help-window (help-buffer)
       (mapcar describe-func (sort sym-list 'string<)))))
```

describe-symbols関数はaproposのように機能しますが、より多くの情報を提供します。

```
(describe-symbols "goal")

---------- Buffer: *Help* ----------
goal-column     Option
Semipermanent goal column for vertical motion, as set by ...

minibuffer-temporary-goal-position      Variable
not documented

set-goal-column Keys: C-x C-n
Set the current horizontal position as a goal for C-n and C-p.
```

```
Those commands will move to this position in the line moved to
rather than trying to keep the same horizontal position.
With a non-nil argument ARG, clears out the goal column
so that C-n and C-p resume vertical motion.
The goal column is stored in the variable ‘goal-column’.

msgid ""
"(defun describe-symbols (pattern)\n"
"  \"Describe the Emacs Lisp symbols matching PATTERN.\n"
"All symbols that have PATTERN in their name are described\n"
"in the `*Help*' buffer.\"\n"
"  (interactive \"sDescribe symbols matching: \")\n"
"  (let ((describe-func\n"
"         (function\n"
"          (lambda (s)\n"

temporary-goal-column   Variable
Current goal column for vertical motion.
It is the column where point was at the start of the current run
of vertical motion commands.

When moving by visual lines via the function ‘line-move-visual’, it is a cons
cell (COL . HSCROLL), where COL is the x-position, in pixels,
divided by the default column width, and HSCROLL is the number of
columns by which window is scrolled from left margin.

When the ‘track-eol’ feature is doing its job, the value is
‘most-positive-fixnum’.
---------- Buffer: *Help* ----------
```

`Snarf-documentation` *filename* [Function]

この関数は Emacs ビルド時の実行可能な Emacs ダンプ直前に使用される。これはファイル *filename*内に格納されたドキュメント文字列の位置を探して、メモリー上の関数定義および変数のプロパティリスト内にそれらの位置を記録する。Section E.1 [Building Emacs], page 1175 を参照のこと。

Emacs は `emacs/etc`ディレクトリーからファイル *filename*を読み込む。その後ダンプされた Emacs 実行時に、ディレクトリー `doc-directory`内の同じファイルを照会する。*filename*は通常は`"DOC"`。

`doc-directory` [Variable]

この変数はビルトインおよび事前ロードされた関数と変数のドキュメント文字列を含んだファイル`"DOC"`があるべきディレクトリーの名前を保持する。

これはほとんどの場合は `data-directory`と同一。実際にインストールした Emacs ではなく Emacs をビルドしたディレクトリーから Emacs を実行したときは異なるかもしれない。[Definition of data-directory], page 528 を参照のこと。

24.3 ドキュメント内でのキーバインディングの置き換え

ドキュメント文字列がキーシーケンスを参照する際、それらはカレントである実際のキーバインディングを使用するべきです。これらは以下で説明する特別なキーシーケンスを使用して行うことができます。通常の方法によるドキュメント文字列へのアクセスは、これらの特別なキーシーケンスをカレントキーバインディングに置き換えます。これは `substitute-command-keys`を呼び出すことにより行われます。あなた自身がこの関数を呼び出すこともできます。

以下はそれら特別なシーケンスと、その意味についてのリストです:

`\[`*command*`]`
: これは *command*を呼び出すキーシーケンス、または *command*がキーバインディングをもたなければ‘`M-x` *command*’。

`\{`*mapvar*`}`
: これは変数 *mapvar*の値であるようなキーマップの要約 (summary) を意味する。この要約は `describe-bindings`を用いて作成される。

`\<`*mapvar*`>`
: これ自体は何のテキストも意味せず副作用のためだけに使用される。これはこのドキュメント文字列内にある、後続のすべての‘`\[`*command*`]`’にたいするキーマップとして *mapvar*の値を指定する。

`` ` ``
: グレイブアクセント (grave accent) は左クォートを意味する。これは `text-quoting-style`の値に応じて左シングルクォーテーションマーク、アポストロフィ、グレイブアクセントのいずれかを生成する。Section 24.4 [Text Quoting Style], page 524 を参照のこと。

`'`
: アポストロフィ(apostrophe) は右クォートを意味する。これは `text-quoting-style`の値に応じて右シングルクォーテーションマーク、アポストロフィのいずれかを生成する。Section 24.4 [Text Quoting Style], page 524 を参照のこと。

`\=`
: これは後続の文字をクォートして無効にする。したがって‘`\=\[`’は‘`\[`’、‘`\=\=`’は‘`\=`’を出力する。

注意してください: Emacs Lisp 内の文字列として記述する際は‘`\`’を 2 つ記述しなければなりません。

`text-quoting-style` [User Option]
: この変数の値はヘルプやメッセージの文章内のシングルクォートにたいして Emacs が使用するべきスタイルを指定するシンボル。この変数の値が `curve`なら‘`like this`’のような curved single quotes スタイル。値が `straight`なら`'like this'`のような straight apostrophes スタイル。値が `grave`ならクォートは変換せずに、‘`like this`’のような Emacs バージョン 25 以前の標準の grave accent と apostrophe によるスタイル。デフォルト値の `nil`は curved single quotes が表示可能なようなら `curve`、それ以外なら `grave`のように機能する。

  このオプションは curved quotes に問題があるプラットフォームで有用。個人の好みに応じてこれを自由にカスタマイズできる。

`substitute-command-keys` *string* [Function]
: この関数は上述の特別なシーケンスを *string*からスキャンして、それらが意味するもので置き換えてその結果を文字列としてリターンする。これによりそのユーザー自身がカスタマイズした実際のキーシーケンスを参照するドキュメントが表示できる。

  あるコマンドが複数のバインディングをもつ場合、通常この関数は最初に見つかったバインディングを使用する。以下のようにしてコマンドのシンボルプロパティ`:advertised-binding`に割り当てることにより、特定のキーバインディングを指定できる:

```
(put 'undo :advertised-binding [?\C-/])
```

  `:advertised-binding`プロパティはメニューアイテム (Section 22.17.5 [Menu Bar], page 453 を参照) に表示されるバインディングにも影響する。コマンドが実際にもたないキーバインディングを指定するとこのプロパティは無視される。

以下は特別なキーシーケンスの例:

```
(substitute-command-keys
   "To abort recursive edit, type `\\[abort-recursive-edit]'.")
⇒ "To abort recursive edit, type ‘C-]’."

(substitute-command-keys
   "The keys that are defined for the minibuffer here are:
  \\{minibuffer-local-must-match-map}")
⇒ "The keys that are defined for the minibuffer here are:

?               minibuffer-completion-help
SPC             minibuffer-complete-word
TAB             minibuffer-complete
C-j             minibuffer-complete-and-exit
RET             minibuffer-complete-and-exit
C-g             abort-recursive-edit
"

(substitute-command-keys
   "To abort a recursive edit from the minibuffer, type \
`\\<minibuffer-local-must-match-map>\\[abort-recursive-edit]'.")
⇒ "To abort a recursive edit from the minibuffer, type ‘C-g’."
```

ドキュメント文字列内のテキストにたいしては他にも特別な慣習があります。それらはたとえばこのマニュアルの関数、変数、およびセクションで参照できます。詳細は Section D.6 [Documentation Tips], page 1168 を参照してください。

24.4 テキストのクォートスタイル

ドキュメント文字列や診断メッセージの中では、グレイブアクセント (grave accent) とアポストロフィ (apostrophe) は通常は特別に扱われてマッチするシングルクォーテーションマーク (“curved quotes” とも呼ばれる) に変換されます。たとえばドキュメント文字列`"Alias for ‘foo’."`と関数呼び出し`(message "Alias for `foo'.")`はどちらも`"Alias for ‘foo’."`に変換されます。あまり一般的ではありませんが、Emacs がグレイブアクセントとアポストロフィをそのまま表示したり、アポストロフィだけ (`"Alias for 'foo'."`) を表示する場合もあります。ドキュメント文字列やメッセージフォーマットは、これらのスタイルすべてで良好に表示されるように記述する必要があります。たとえば通常の英語スタイル`"Alias for 'foo'."`で表示できるのならば、ドキュメント文字列`"Alias for 'foo'."`はあなたが望むスタイルではないでしょう。

テキストのクォートスタイルとは無関係に変換なしでグレイブアクセントやアポストロフィの表示を要するときがあるかもしれません。ドキュメント文字列ではエスケープでこれを行うことができます。たとえばドキュメント文字列`"\\=‘(a ,(sin 0)) ==> (a 0.0)"`では、グレイブアクセントは Lisp コードの表現を意図しているので、グレイブアクセントエスケープしてクォートスタイルとは無関係にそれ自身を表示します。`message`や`error`の呼び出しでは、フォーマット`"%s"`と`format`呼び出しを引数として変換を回避できます。たとえば`(message "%s" (format "`(a ,(sin %S)) ==> (a %S)" x (sin x)))`はテキストのクォートスタイルに関わらず、グレイブアクセントで始まるメッセージを表示します。

`text-quoting-style` [User Option]

このユーザーオプションの値はヘルプやメッセージの文言中のシングルクォートにたいして、Emacs が使用するべきスタイルを指定するシンボル。このオプションの値が`curve`なら‘`like this`’のような curved single quotes スタイル。値が`straight`なら`'like this'`のような素のアポストロフィスタイル。値が`grave`ならクォートは変換せずに、`‘like this'`のよう

な Emacs バージョン 25 以前の標準スタイルであるグレイブアクセントとアポストロフィのスタイル。デフォルト値の `nil`は curved single quotes が表示可能なようなら `curve`、それ以外なら `grave`のように機能する。

このオプションは curved quotes に問題があるプラットフォームで有用。個人の好みに応じてこれを自由にカスタマイズできる。

24.5 ヘルプメッセージの文字記述

以下の関数はイベント、キーシーケンス、文字をテキスト表記 (textual descriptions) に変換します。これらの変換された表記は、メッセージ内に任意のテキスト文字やキーシーケンスを含める場合に有用です。なぜなら非プリント文字や空白文字はプリント文字シーケンスに変換されるからです。空白文字以外のプリント文字はその文字自身が表記になります。

`key-description` *sequence* **&optional** *prefix* [Function]

この関数は *sequence*内の入力イベントにたいして Emacs の標準表記を含んだ文字列をリターンする。*prefix*が非 `nil`なら、それは *sequence*に前置される入力イベントシーケンスであり、リターン値にも含まれる。引数には文字列、ベクター、またはリストを指定できる。有効なイベントに関する詳細は Section 21.7 [Input Events], page 387 を参照のこと。

```
(key-description [?\M-3 delete])
     ⇒ "M-3 <delete>"
(key-description [delete] "\M-3")
     ⇒ "M-3 <delete>"
```

以下の `single-key-description`の例も参照のこと。

`single-key-description` *event* **&optional** *no-angles* [Function]

この関数はキーボード入力にたいする Emacs の標準表記として *event*を表記する文字列をリターンする。通常のプリント文字はその文字自身で表れるが、コントロール文字は '`C-`'で始まる文字列、メタ文字は '`M-`'で始まる文字列、スペースやタブ等は '`SPC`'や '`TAB`'のように変換される。ファンクションキーのシンボルは '`<...>`'のように角カッコ (angle brackets) の内側に表れる。リストであるようなイベントは、そのリストの CAR 内のシンボル名が角カッコの内側に表れる。

オプション引数 *no-angles*が非 `nil`なら、ファンクションキーやイベントシンボルを括る角カッコは省略される。これは角カッコを使用しない古いバージョンの Emacs との互換性のため。

```
(single-key-description ?\C-x)
     ⇒ "C-x"
(key-description "\C-x \M-y \n \t \r \f123")
     ⇒ "C-x SPC M-y SPC C-j SPC TAB SPC RET SPC C-l 1 2 3"
(single-key-description 'delete)
     ⇒ "<delete>"
(single-key-description 'C-mouse-1)
     ⇒ "<C-mouse-1>"
(single-key-description 'C-mouse-1 t)
     ⇒ "C-mouse-1"
```

`text-char-description` *character* [Function]

この関数はテキスト内に出現し得る文字にたいする Emacs の標準表記として *character*を記述する文字列をリターンする。これは `single-key-description`と似ているが、引数が `characterp`によるテスト (Section 33.5 [Character Codes], page 856 を参照) をパスする有効な文字コードでなければならない点が異なる。この関数は先頭のカレットによりコントロール文字の記述を生成する (Emacs がバッファー内にコントロール文字を表示する通常の方法)。

修飾ビットをもつ文字にたいして、この関数はエラーをシグナルする (コントロール修飾された ASCII文字は例外であり、これらはコントロール文字として表現される)。

```
(text-char-description ?\C-c)
     ⇒ "^C"
(text-char-description ?\M-m)
     error  Wrong type argument: characterp, 134217837
```

`read-kbd-macro` *string* **&optional** *need-vector* [Command]

この関数は主にキーボードマクロを操作するために使用されるが、大雑把な意味で `key-description`の逆の処理にも使用できる。キー表記を含むスペース区切りの文字列でこれを呼び出すと、それに対応するイベントを含む文字列かベクターをリターンする (これは単一の有効なキーシーケンスであるか否かは問わず何のイベントを使用するかに依存する。Section 22.1 [Key Sequences], page 422 を参照のこと)。*need-vector*が非 `nil`ならリターン値は常にベクター。

24.6 ヘルプ関数

Emacs はさまざまなビルトインのヘルプ関数を提供しており、それらはすべてプレフィックス *C-h*のサブコマンドとしてユーザーがアクセスできます。それらについての詳細は Section “Help” in *The GNU Emacs Manual* を参照してください。ここでは同様な情報に関するプログラムレベルのインターフェイスを説明します。

`apropos` *pattern* **&optional** *do-all* [Command]

この関数は名前に apropos パターン (apropos pattern: 適切なパターン) *pattern*を含む重要なすべてのシンボルを探す。マッチに使用される apropos パターンは単語、最低 2 つはマッチしなければならないスペース区切りの単語、または (特別な正規表現文字があれば) 正規表現のいずれか。あるシンボルが関数、変数、フェイスとしての定義、あるいはプロパティをもつならそのシンボルは重要とみなされる。

この関数は以下のような要素のリストをリターンする:

```
(symbol score function-doc variable-doc
 plist-doc widget-doc face-doc group-doc)
```

ここで *score*はマッチの面からそのシンボルがどれだけ重要に見えるかを比較する整数である。残りの各要素は *symbol*にたいする関数、変数、... 等のドキュメント文字列 (または `nil`)。

これは`*Apropos*`という名前のバッファーにもシンボルを表示する。その際、各行にはドキュメント文字列の先頭から取得した 1 行説明とともに表示される。

*do-all*が非 `nil`、またはユーザーオプション `apropos-do-all`が非 `nil`なら、`apropos`は見つかった関数のキーバインディングも表示する。これは重要なものだけでなく、intern されたすべてのシンボルも表示する (同様にリターン値としてもそれらをリストする)。

`help-map` [Variable]

この変数の値は Help キー *C-h*に続く文字にたいするローカルキーマップである。

`help-command` [Prefix Command]

このシンボルは関数ではなく関数定義セルには `help-map`としてキーマップを保持する。これは `help.el`内で以下のように定義されている:

```
(define-key global-map (string help-char) 'help-command)
(fset 'help-command help-map)
```

`help-char` [User Option]
この変数の値はヘルプ文字 (help character: Help を意味する文字として Emacs が認識する文字)。デフォルトの値は *C-h*を意味する 8。この文字を読み取った際に `help-form`が非 `nil` の Lisp 式なら、Emacs はその式を評価して結果が文字列の場合はウィンドウ内にそれを表示する。

`help-form`の値は通常は `nil`。その場合にはヘルプ文字はコマンド入力のレベルにおいて特別な意味を有さず、通常の方法におけるキーシーケンスの一部となる。*C-h*の標準的なキーバインディングは、複数の汎用目的をもつヘルプ機能のプレフィックスキー。

ヘルプ文字はプレフィックスキーの後でも特別な意味をもつ。ヘルプ文字がプレフィックスキーのサブコマンドとしてバインディングをもたなければ、そのプレフィックスキーのすべてのサブコマンドのリストを表示する `describe-prefix-bindings`を実行する。

`help-event-list` [User Option]
この変数の値はヘルプ文字の選択肢の役割を果たすイベント型のリスト。これらのイベントは `help-char`で指定されるイベントと同様に処理される。

`help-form` [Variable]
この変数が非 `nil`なら、その値は文字 `help-char`が読み取られるたびに評価されるフォームであること。そのフォームの評価によって文字列が生成されたらその文字列が表示される。

`read-event`、`read-char-choice`、`read-char`を呼び出すコマンドは、それが入力を行う間は恐らく `help-form`を非 `nil`にバインドするべきだろう (*C-h*が他の意味をもつなら行わないこと)。この式を評価した結果は、それが何にたいする入力なのかと、それを正しくエンターする方法を説明する文字列であること。

ミニバッファーへのエントリーにより、この変数は `minibuffer-help-form`の値にバインドされる ([Definition of minibuffer-help-form], page 374 を参照)。

`prefix-help-command` [Variable]
この変数はプレフィックスキーにたいするヘルプをプリントする関数を保持する。その関数はユーザーが後にヘルプ文字を伴うプレフィックスキーをタイプして、そのヘルプ文字がプレフィックスの後のバインディングをもたないたときに呼び出される。この変数のデフォルト値は `describe-prefix-bindings`。

`describe-prefix-bindings` [Command]
この関数はもっとも最近のプレフィックスキーのサブコマンドすべてにたいするリストを表示する `describe-bindings`を呼び出す。記述されるプレフィックスは、そのキーシーケンスの最後のイベントを除くすべてから構成される (最後のイベントは恐らくヘルプ文字)。

以下の 2 つの関数は electric モードのように制御を放棄せずにヘルプを提供したいモードを意図しています。これらは通常のヘルプ関数と区別するために名前が '`Helper`'で始まります。

`Helper-describe-bindings` [Command]
このコマンドはローカルキーマップとグローバルキーマップの両方のキーバインディングすべてのリストを含むヘルプバッファーを表示するウィンドウをポップアップする。これは `describe-bindings`を呼び出すことによって機能する。

`Helper-help` [Command]
このコマンドはカレントモードにたいするヘルプを提供する。これはミニバッファー内でメッセージ '`Help (Type ? for further options)`'とともにユーザーに入力を求めて、その後

キーバインディングが何か、何を意図するモードなのかを探すための助けを提供する。これは `nil`をリターンする。

これはマップ `Helper-help-map`を変更することによってカスタマイズできる。

`data-directory` [Variable]
この変数は Emacs に付随する特定のドキュメントおよびテキストファイルを探すディレクトリーの名前を保持する。

`help-buffer` [Function]
この関数はヘルプバッファーの名前 (通常は`*Help*`) をリターンする。そのようなバッファーが存在しなければ最初にそれを作成する。

`with-help-window` *buffer-or-name body...* [Macro]
このマクロは `with-output-to-temp-buffer` (Section 39.8 [Temporary Displays], page 996 を参照) のように *body*を評価して、そのフォームが生成したすべての出力を *buffer-name*で指定されるバッファーに挿入する (*buffer-name*は関数 `help-buffer`によりリターンされる値であることが多い)。このマクロは指定されたバッファーを Help モードにして、ヘルプウィンドウの quit やスクロールする方法を告げるメッセージを表示する。これはユーザーオプション `help-window-select`のカレント値が適切にセットされていればヘルプウィンドウの選択も行う。これは *body*内の最後の値をリターンする。

`help-setup-xref` *item interactive-p* [Function]
この関数は`*Help*`バッファー内のクロスリファレンスデータを更新する。このクロスリファレンスはユーザーが '`Back`'ボタンか '`Forward`'ボタン上でクリックした際のヘルプ情報の再生成に使用される。`*Help*`バッファーを使用するほとんどのコマンドは、バッファーをクリアーする前にこの関数を呼び出すべきである。*item*引数は `(`*`function . args`*`)`という形式であること。ここで *function*は引数リスト *args*で呼び出されるヘルプバッファーを再生成する関数。コマンド呼び出しが interactive に行われた場合、*interactive-p*引数は非 `nil`。この場合には `*Help*`バッファーの '`Back`'ボタンにたいする item のスタックはクリアーされる。

`help-buffer`、`with-help-window`、`help-setup-xref`の使用例は [describe-symbols example], page 520 を参照してください。

`make-help-screen` *fname help-line help-text help-map* [Macro]
このマクロは提供するサブコマンドのリストを表示するプレフィックスキーのように振る舞う、*fname*という名前のヘルプコマンドを定義する。

呼び出された際、*fname*はウィンドウ内に *help-text*を表示してから *help-map*に応じてキーシーケンスの読み取りと実行を行う。文字列 *help-text*は *help-map*内で利用可能なバインディングを説明すること。

コマンド *fname*は *help-text*の表示をスクロールすることによる、自身のいくつかのイベントを処理するために定義される。*fname*がこれらのスペシャルイベントのいずれかを読み取った際には、スクロールを行った後で他のイベントを読み取る。自身が処理する以外のイベントを読み取りそのイベントが *help-map*内にバインディングを有す際は、そのキーのバインディングを実行した後にリターンする。

引数 *help-line*は *help-map*内の候補の 1 行要約であること。Emacs のカレントバージョンでは、オプション `three-step-help`を `t`にセットした場合のみこの引数が使用される。

このマクロは *`C-h C-h`*にバインドされるコマンド `help-for-help`内で使用される。

`three-step-help` [User Option]

この変数が非 `nil`なら、`make-help-screen`で定義されたコマンドは最初にエコーエリア内に自身の *help-line*文字列を表示して、ユーザーが再度ヘルプ文字をタイプした場合のみ長い *help-text*文字列を表示する。

25 ファイル

このチャプターでは検索、作成、閲覧、保存、その他ファイルとディレクトリーにたいして機能する Emacs Lisp の関数と変数について説明します。その他のいくつかのファイルに関する関数については Chapter 27 [Buffers], page 590、バックアップと auto-save(自動保存) に関する関数については Chapter 26 [Backups and Auto-Saving], page 578 で説明されています。

ファイル関数の多くはファイル名であるような引数を 1 つ以上受け取ります。このファイル名は文字列です。これらの関数のほとんどは関数 `expand-file-name`を使用してファイル名引数を展開するので、`~`は相対ファイル名 (`../`と空文字列を含む) として正しく処理されます。Section 25.9.4 [File Name Expansion], page 559 を参照してください。

加えて特定の *magic* ファイル名は特別に扱われます。たとえばリモートファイル名が指定された際、Emacs は適切なプロトコルを通じてネットワーク越しにファイルにアクセスします。Section "Remote Files" in *The GNU Emacs Manual* を参照してください。この処理は非常に低レベルで行われるので、特に注記されたものを除いて、このチャプターで説明するすべての関数がファイル名引数として magic ファイル名を受け入れると想定しても良いでしょう。詳細は See Section 25.12 [Magic File Names], page 569 を参照してください。

ファイル I/O 関数が Lisp エラーをシグナルする際、通常はコンディション `file-error`を使用します (Section 11.7.3.3 [Handling Errors], page 167 を参照)。ほとんどの場合にはオペレーティングシステムからロケール `system-messages-locale`に応じたエラーメッセージが取得されて、コーディングシステム `locale-coding-system`を使用してデコードされます (Section 33.12 [Locales], page 879 を参照)。

25.1 ファイルの visit

ファイルの visit とはファイルをバッファーに読み込むことを意味します。一度これを行うと、わたしたちはバッファーがファイルを *visit(訪問)* していると言い、ファイルのことをバッファーの visit されたファイルと呼んでいます。

ファイルとバッファーは 2 つの異なる事柄です。ファイルとは、(削除しない限り) コンピューター内に永続的に記録された情報です。一方バッファーとは編集セッションの終了 (またはバッファーの kill) とともに消滅する、Emacs 内部の情報です。あるバッファーがファイルを vist しているとき、バッファーにはファイルからコピーされた情報が含まれます。編集コマンドにより変更されるのはバッファー内のコピーです。バッファーへの変更によってファイルは変更されません。その変更を永続化するためにはバッファーを保存 *(save)* しなければなりません。これは変更されたバッファーのコンテンツをファイルにコピーして書き戻すことを意味します。

ファイルとバッファーは異なるにも関わらず、人はバッファーという意味でファイルを呼んだり、その逆を行うことが多々あります。実際のところ、"わたしはまもなく同じ名前のファイルに保存するためのバッファーを編集している" ではなく、"わたしはファイルを編集している" と言います。人間がこの違いを明確にする必要は通常はありません。しかしコンピュータープログラムで対処する際には、この違いを心に留めておくのが良いでしょう。

25.1.1 ファイルを visit する関数

このセクションではファイルの visit に通常使用される関数を説明します。歴史的な理由によりこれらの関数は '`visit-`'ではなく、'`find-`'で始まる名前をもちます。バッファーを visit しているファイルの名前へのアクセスや、visit されたファイル名から既存のバッファーを見つける関数および変数については Section 27.4 [Buffer File Name], page 594 を参照してください。

ファイル内容を見たいものの変更したくない場合にはテンポラリーバッファー (temporary buffer: 一時的なバッファー) で `insert-file-contents`を使用例するのが、Lisp プログラム内ではもっとも高速な方法です。時間を要するファイルの visit は必要ありません。Section 25.3 [Reading from Files], page 537 を参照してください。

`find-file` *filename* **&optional** *wildcards* [Command]

このコマンドはファイル *filename*を visit しているバッファーを選択する。visit している既存のバッファーがあればそのバッファー、なければバッファーを新たに作成してそのバッファーにファイルを読み込む。これはそのバッファーをリターンする。

技術的な詳細を除けば `find-file`関数の body は基本的には以下と等価:

```
(switch-to-buffer (find-file-noselect filename nil nil wildcards))
```

(Section 28.12 [Switching Buffers], page 637 の `switch-to-buffer`を参照されたい。)

*wildcards*が非 `nil` (interactive に呼び出された場合は常に真) の場合、`find-file`は *filename*内のワイルドカード文字を展開してマッチするすべてのファイルを visit する。

`find-file`が interactive に呼び出された際にはミニバッファー内で *filename*の入力を求める。

`find-file-literally` *filename* [Command]

このコマンドは `find-file`が行うように *filename*を visit するが、フォーマット変換 (Section 25.13 [Format Conversion], page 573 を参照)、文字コード変換 (Section 33.10 [Coding Systems], page 864 を参照)、EOL 変換 (Section 33.10.1 [Coding System Basics], page 865 を参照) を何も行わない。ファイルを visit しているバッファーは unibyte になり、ファイル名とは無関係にバッファーのメジャーモードは Fundamental モードになる。ファイル内で指定されたファイルローカル変数 (Section 12.12 [File Local Variables], page 197 を参照) は無視され、自動的な解凍と `require-final-newline`によるファイル終端への改行追加 (Section 25.2 [Saving Buffers], page 534 を参照) も無効になる。

Emacs がすでにリテラリー (literally: 文字通り、そのまま) でない方法で同じファイルを visit しているバッファーをもつ場合には、Emacs はその同じファイルをリテラリーに visit せず、単に既存のバッファーに切り替えることに注意。あるファイルのコンテンツにたいして確実にリテラリーにアクセスしたければテンポラリーバッファーを作成して、`insert-file-contents-literally`を使用してファイルのコンテンツを読み込むこと (Section 25.3 [Reading from Files], page 537 を参照)。

`find-file-noselect` *filename* **&optional** *nowarn rawfile wildcards* [Function]

これはファイルを visit するすべての関数の要となる関数である。これはファイル *filename*を visit しているバッファーをリターンする。望むならそのバッファーをカレントにしたり、あるウィンドウ内に表示することができるだろうがこの関数はそれを行わない。

関数は既存のバッファーがあればそれをリターンし、なければ新たにバッファーを作成してそれにファイルを読み込む。`find-file-noselect`が既存のバッファーを使用する際は、まずファイルがそのバッファーに最後に visit、または保存したときから変更されていないことを検証する。ファイルが変更されていれば、この関数は変更されたファイルを再読み込みするかどうかをユーザーに尋ねる。ユーザーが‘`yes`’と応えたら、以前に行われたそのバッファー内での編集は失われる。

ファイルの読み込みは EOL 変換、フォーマット変換 (Section 25.13 [Format Conversion], page 573 を参照) を含むファイルコンテンツのデコードを要する (Section 33.10 [Coding Systems], page 864 を参照)。*wildcards*が非 `nil`なら、`find-file-noselect`は *filename* 内のワイルドカード文字を展開してマッチするすべてのファイルを visit する。

この関数はオプション引数 *nowarn*が `nil`なら、さまざまな特殊ケースにおいて警告メッセージ (warning message)、および注意メッセージ (advisory message) を表示する。たとえば関数がバッファーの作成を必要とし、かつ *filename*という名前のファイルが存在しなければ、エコーエリア内にメッセージ‘`(New file)`’を表示してそのバッファーを空のままに留める。

`find-file-noselect`関数は、ファイルを読み込んだ後に通常は `after-find-file`を呼び出す (Section 25.1.2 [Subroutines of Visiting], page 533 を参照)。この関数はバッファーのメジャーモードのセット、ローカル変数のパース、正に visit したファイルより新しい auto-save ファイルが存在する場合にユーザーへの警告を行い、`find-file-hook`内の関数を実行することにより終了する。

オプション引数 *rawfile*が非 `nil`なら `after-find-file`は呼び出されず、失敗時に `find-file-not-found-functions`は呼び出されない。さらに非 `nil`値の *rawfile*は、コーディングシステム変換とフォーマット変換を抑制する。

`find-file-noselect`関数は、通常はファイル *filename*を visit しているバッファーをリターンする。しかしワイルドカードが実際に使用、展開された場合には、それらのファイルを visit しているバッファーのリストをリターンする。

```
(find-file-noselect "/etc/fstab")
     ⇒ #<buffer fstab>
```

`find-file-other-window` *filename* **&optional** *wildcards* [Command]

このコマンドはファイル *filename*を visit しているバッファーを選択するが、選択されたウィンドウではない他のウィンドウでこれを行う。これは別の既存ウィンドウを使用したり、ウィンドウを分割するかもしれない。Section 28.12 [Switching Buffers], page 637l を参照のこと。

このコマンドが interactive に呼び出された際は *filename*の入力を求める。

`find-file-read-only` *filename* **&optional** *wildcards* [Command]

このコマンドは `find-file`のようにファイル *filename*を visit しているバッファーを選択するが、そのバッファーを読み取り専用 (read-only) とマークする。関連する関数と変数については Section 27.7 [Read Only Buffers], page 599 を参照のこと。

このコマンドが interactive に呼び出された際は *filename*の入力を求める。

`find-file-wildcards` [User Option]

この変数が非 `nil`なら、各種 `find-file`コマンドはワイルドカード文字をチェックして、それらにマッチするすべてのファイルを visit する (interactive に呼び出されたときや *wildcards*引数が非 `nil`のとき)。このオプションが `nil`なら、`find-file`コマンドはそれらの *wildcards* 引数を無視してワイルドカード文字を特別に扱うことは決してない。

`find-file-hook` [User Option]

この変数の値はファイルが visit された後に呼び出される関数のリスト。ファイルのローカル変数指定は、(もしあれば) このフックが実行される前に処理されるだろう。フック関数実行時はそのファイルを visit しているバッファーがカレントになる。

この変数はノーマルフックである。Section 23.1 [Hooks], page 461 を参照のこと。

`find-file-not-found-functions` [Variable]

この変数の値は `find-file`や `find-file-noselect`が存在しないファイル名を受け取った際に呼び出される関数のリスト。存在しないファイルを検知すると `find-file-noselect`は直ちにこれらの関数を呼び出す。これらのいずれかが非 `nil`をリターンするまで、リスト順に関数を呼び出す。`buffer-file-name`はすでにセットアップ済みである。

関数の値が使用されること、および多くの場合いくつかの関数だけが呼び出されるので、これはノーマルフックではない。

`find-file-literally` [Variable]
このバッファーローカル変数が非`nil`値にセットされると、`save-buffer`はあたかもそのバッファーがリテラリー、つまり何の変換も行わずにファイルを visit していたかのように振る舞う。コマンド`find-file-literally`はこの変数のローカル値をセットするが、その他の等価な関数およびコマンドも、たとえばファイル終端への改行の自動追加を避けるためにこれを同様に行うことができる。この変数は恒久的にローカルなのでメジャーモードの変更による影響を受けない。

25.1.2 visit のためのサブルーチン

`find-file-noselect`関数は、2 つの重要なサブルーチン`create-file-buffer`と`after-find-file`を使用します。これらはユーザーの Lisp コードでも役に立つことがあります。このセクションではそれらの使い方について説明します。

`create-file-buffer` *filename* [Function]
この関数は *filename*の visit にたいして適切な名前のバッファーを作成してそれをリターンする。これは *filename* (ディレクトリーを含まず) の名前がフリーならバッファー名にそれを使用し、フリーでなければ未使用の名前を取得するために '`<2>`'のような文字列を付加する。Section 27.9 [Creating Buffers], page 602 も参照のこと。`uniquify`ライブラリーはこの関数の結果に影響を与えることに注意。Section "Uniquify" in *The GNU Emacs Manual* を参照のこと。

注意されたい: `create-file-buffer`はファイルに新たなバッファーを関連付けない。バッファーの選択もせず、さらにデフォルトのメジャーモードも使用しない。

```
(create-file-buffer "foo")
     ⇒ #<buffer foo>
(create-file-buffer "foo")
     ⇒ #<buffer foo<2>>
(create-file-buffer "foo")
     ⇒ #<buffer foo<3>>
```

この関数は `find-file-noselect`により使用される。この関数自身は `generate-new-buffer`を使用する (Section 27.9 [Creating Buffers], page 602 を参照)。

`after-find-file` **&optional** *error warn noauto* [Function]
after-find-file-from-revert-buffer nomodes
この関数はバッファーのメジャーモードをセットして、ローカル変数をパースする (Section 23.2.2 [Auto Major Mode], page 468 を参照)。これは`find-file-noselect`、およびデフォルトのリバート関数 (Section 26.3 [Reverting], page 586 を参照) により呼び出される。

ファイルが存在しないという理由によりファイルの読み込みがエラーを受け取るがディレクトリーは存在するなら、呼び出し側は *error*にたいして非`nil`値を渡すこと。この場合、`after-find-file`は警告 '`(New file)`'を発する。より深刻なエラーにたいしては、呼び出し側は通常は`after-find-file`を呼び出さないこと。

*warn*が非`nil`なら、もし auto-save ファイルが存在して、かつそれが visit されているファイルより新しければ、この関数は警告を発する。

*noauto*が非 `nil`なら、それは Auto-Save モードを有効や無効にしないことを告げる。以前に Auto-Save モードが有効なら有効のまま留まる。

*after-find-file-from-revert-buffer*が非 `nil`なら、それはこの関数が `revert-buffer`から呼び出されたことを意味する。これに直接的な効果はないが、モード関数とフック関数の中には、この変数の値をチェックするものがいくつかある。

*nomodes*が非 `nil`なら、それはバッファーのメジャーモードを変更せず、ファイル内のローカル変数指定を処理せず、`find-file-hook`を実行しないことを意味する。この機能はあるケースにおいて `revert-buffer`により使用される。

`after-find-file`はリスト `find-file-hook`内のすべての関数を最後に呼び出す。

25.2 バッファーの保存

Emacs 内でファイルを編集する際には、実際にはそのファイルを visit しているバッファーにたいして編集を行っています。つまりファイルのコンテンツをバッファーにコピーしてそのコピーを編集しています。そのバッファーを変更してもバッファーを保存 *(save)* するまでファイルは変更されません。保存とはバッファーのコンテンツをファイルにコピーすることを意味します。ファイルを visit していないバッファーでも、バッファーローカルん q `write-contents-functions`の関数を使用することにより “saved(保存済み)” にすることができます。

`save-buffer` **&optional** *backup-option* [Command]

この関数はバッファーが最後に visit されたときや保存されたときから変更されていれば、カレントバッファーのコンテンツをバッファーによって visit されているファイルに保存、変更されていなければ何も行わない。

`save-buffer`はバックアップファイルの作成に責任を負う。*backup-option*は通常は `nil`であり、`save-buffer`はファイルの visit 以降、それが最初の保存の場合のみバックアップファイルを作成する。*backup-option*にたいする他の値は、別の条件によるバックアップファイル作成を要求する:

- 引数 4 は 1 つの *C-u*、引数 64 は 3 つの *C-u*を意味するので、`save-buffer`はバッファーの次回保存時にこのバージョンのファイルがバックアップされるようマークする。
- 引数 16 は 2 つの *C-u*、引数 64 は 3 つの *C-u*を意味するので、`save-buffer`関数はそれを保存する前に前バージョンのファイルを無条件にバックアップする。
- 引数 0 は無条件にバックアップファイルを何も作成しない。

`save-some-buffers` **&optional** *save-silently-p pred* [Command]

このコマンドはファイルを visit している変更されたバッファーのいくつかを保存する。これは通常は各バッファーごとにユーザーに確認を求める。しかし *save-silently-p*が非 `nil`なら、ユーザーに質問せずにファイルを visit しているすべてのバッファーを保存する。

オプション引数 *pred*は、どのバッファーで確認を求めるか (または *save-silently-p*が非 `nil`ならどのバッファーで確認せずに保存するか) を制御する。

*pred*が `nil`なら、*pred*のかわりに `save-some-buffers-default-predicate`の値を使用することを意味する。その結果が `nil`ならファイルを visit しているバッファーにたいしてのみ確認を求めることを意味する。`t`なら `buffer-offer-save`のバッファーローカル値が `nil`であるような非ファイルバッファー以外の特定のバッファーの保存も提案することを意味する (Section 27.10 [Killing Buffers], page 603 を参照)。ユーザーが非ファイルバッファーの保存にたいして ‘`yes`’と応えると保存に使用するファイル名の指定を求める。`save-buffers-kill-emacs`関数は *pred*にたいして値 `t`を渡す。

述語が `t`と `nil`のいずれでもなければ引数なしの関数であること。その関数はバッファーの保存を提案するか否かを決定するためにバッファーごとに呼び出される。これが特定のバッファーで非 `nil`値をリターンすればバッファー保存の提案を行うことを意味する。

`write-file` *filename* **&optional** *confirm* [Command]

この関数はカレントバッファーをファイル *filename*に書き込んで、バッファーがそのファイルを visit していることにして未変更とマークする。次に *filename*にもとづいてバッファー名をリネームする。バッファー名を一意にするため、必要なら '`<2>`'のような文字列を付加する。処理のほとんどは `set-visited-file-name` (Section 27.4 [Buffer File Name], page 594 を参照)、および `save-buffer`を呼び出すことにより行われる。

*confirm*が非 `nil`なら、それは既存のファイルを上書きする前に確認を求めることを意味する。ユーザーがプレフィックス引数を与えなければ interactive に確認が求められる。

*filename*がディレクトリー名 (Section 25.9.3 [Directory Names], page 558 を参照)、または既存のディレクトリーへのシンボリックリンクなら、`write-file`はディレクトリー *filename* 内で visit されているファイルの名前を使用する。そのバッファーがファイルを visit していなければ、かわりにバッファーの名前を使用する。

バッファーの保存によりフックがいくつか実行されます。これによりフォーマット変換も処理されます (Section 25.13 [Format Conversion], page 573 を参照)。以下で説明するこれらのフックはバッファーのテキストをファイルに書き込む `save-buffer`以外の他のプリミティブや関数、とりわけ auto-saving (Section 26.2 [Auto-Saving], page 583 を参照) では実行されないことに注意してください。

`write-file-functions` [Variable]

この変数の値は visit されているファイルをバッファーに書き出す前に呼び出される関数のリスト。それらのうちのいずれかが非 `nil`をリターンしたら、そのファイルは書き込み済みだと判断されて残りの関数は呼び出されないし、ファイルを書き込むための通常のコードも実行されない。

`write-file-functions`内の関数が非 `nil`をリターンしたら、(それが適切なら) その関数はファイルをバックアップする責任を負う。これを行うには以下のコードを実行する:

```
(or buffer-backed-up (backup-buffer))
```

`backup-buffer`によりリターンされるファイルモードの値を保存して、(もし非 `nil`なら) 書き込むファイルのモードビットをセットしたいと思うかもしれない。これは正に `save-buffer`が通常行うことである。Section 26.1.1 [Making Backup Files], page 578 を参照のこと。

`write-file-functions`内のフック関数は、データのエンコード (が望ましければ) にも責任を負う。これらは適切なコーディングシステムと改行規則 (Section 33.10.3 [Lisp and Coding Systems], page 867 を参照) を選択してエンコード (Section 33.10.7 [Explicit Encoding], page 876 を参照) を処理して、使用されていたコーディングシステム (Section 33.10.2 [Encoding and I/O], page 866 を参照) を `last-coding-system-used`にセットしなければならない。

バッファー内でこのフックをローカルにセットすると、バッファーはそのファイル、またはバッファーのコンテンツを取得したファイルに類するものに関連付けられる。このようにして変数は恒久的にローカルとマークされるので、メジャーモードの変更がバッファーローカルな値を変更することはない。その一方で `set-visited-file-name`を呼び出すことによって変数はリセットされるだろう。これを望まなければ、かわりに `write-contents-functions`を使用したいと思うかもしれない。

たとえこれがノーマルフックでなくても、このリストを操作するために `add-hook`と`remove-hook`を使用することはできる。Section 23.1 [Hooks], page 461 を参照のこと。

`write-contents-functions` [Variable]

これは正に`write-file-functions`と同様に機能するが、こちらは visit している特定のファイルやファイルの場所ではなくバッファーのコンテンツに関連するフックを意図しており、実際にはファイルを visit していないバッファーにたいして任意の保存処理を作成するために使用できる。そのようなフックは、この変数にたいするバッファーローカルなバインディングとして、通常はメジャーモードにより作成される。この変数はセットされた際には常に自動的にバッファーローカルになる。新たなメジャーモードへの切り替えは常にこの変数をリセットするが、`set-visited-file-name`の呼び出しではリセットされない。

このフック内の関数のいずれかが非`nil`をリターンすると、そのファイルはすでに書き込み済みとみなされて、残りの関数は呼び出されず`write-file-functions`内の関数も呼び出されない。

(スペシャルモードのバッファーのような) ファイルを visit していないバッファーを保存するためにこのフックを利用する際には、その関数が正常に保存することに失敗して`nil`値をリターンすると、`save-buffer`がユーザーにたいしてバッファーを保存するファイルの入力を求めることに留意。これが望ましくなければエラーのレイズによる関数の失敗を考慮されたい。

`before-save-hook` [User Option]

このノーマルフックは visit しているファイルにバッファーが保存される前に実行される。保存が通常の方法で行われるか、あるいは上述のフックのいずれかで行われたかは問題ではない。たとえば`copyright.el`プログラムは、ファイルの保存においてそれの著作権表示が今年であることを確認するためにこのフックを使用する。

`after-save-hook` [User Option]

これはバッファーを visit するファイルに保存後に実行されるノーマルフック。

`file-precious-flag` [User Option]

この変数が非`nil`なら、`save-buffer`は保存ファイルがもつ名前のかわりに一時的な名前で新たなファイルに書き込み、エラーがないことが明確になった後にファイルを意図する名前にリネームすることによって保存中の I/O エラーから防御する。この手順は無効なファイルが原因となるディスク容量逼迫のような問題を防ぐ。

副作用としてバックアップ作成にコピーが必要になる。Section 26.1.2 [Rename or Copy], page 580 を参照のこと。しかし同時にこの高価なファイル保存によって保存したファイルと他のファイル名との間のすべてのハードリンクは切断される。

いくつかのモードは特定のバッファーにおいてこの変数に非`nil`のバッファーローカル値を与える。

`require-final-newline` [User Option]

この変数はファイルが改行で終わらないように書き込まれるかどうかを決定する。変数の値が`t`なら、`save-buffer`はバッファーの終端に改行がなければ暗黙理に改行を追加する。値が`visit`なら、Emacs はファイルを visit した直後に不足している改行を追加する。値が`visit-save`なら、Emacs は visit と保存の両方のタイミングで不足している改行を追加する。その他の非`nil`値にたいしては、そのようなケースが生じるたびに改行を追加するかどうか`save-buffer`がユーザーに尋ねる。

変数の値が`nil`なら`save-buffer`は改行を追加しない。デフォルト値は`nil`だが、特定のバッファーでこれを`t`にセットするメジャーモードも少数存在する。

Section 27.4 [Buffer File Name], page 594 の関数 `set-visited-file-name`も参照されたい。

25.3 ファイルからの読み込み

ファイルのコンテンツをバッファーにコピーするためには関数 `insert-file-contents`を使用します (マークをセットするので Lisp プログラム内でコマンド `insert-file`は使用してはならない)。

`insert-file-contents` *filename* **&optional** *visit beg end replace* [Function]

この関数はファイル *filename*のコンテンツをカレントバッファーのポイントの後に挿入する。これは絶対ファイル名と挿入だれたデータの長さからなるリストをリターンする。*filename*が読み取り可能なファイルの名前でなければエラーがシグナルされる。

この関数は定義されたファイルフォーマットに照らしてファイルのコンテンツをチェックして、適切ならそのコンテンツの変換、およびリスト `after-insert-file-functions`内の関数の呼び出しも行う。Section 25.13 [Format Conversion], page 573 を参照のこと。通常はリスト `after-insert-file-functions`内のいずれかの関数が EOL 変換を含むファイルコンテンツのデコードに使用されるコーディングシステム (Section 33.10 [Coding Systems], page 864 を参照) を判断する。しかしファイルに null バイトが含まれる場合には、デフォルトではコード変換なしで visit される。Section 33.10.3 [Lisp and Coding Systems], page 867 を参照のこと。

*visit*が非 `nil`なら、この関数は追加でそのバッファーを未変更とマークしてそのバッファーのさまざまなフィールドをセットアップして、バッファーがファイル *filename*を visit しているようにする。これらのフィールドにはバッファーが visit したファイルの名前、最終保存したファイルの modtime が含まれる。これらの機能は `find-file-noselect`により使用されるものであり、恐らくあなた自身が使用するべきではない。

*beg*と *end*が非 `nil`なら、それらはファイル挿入範囲を指定するバイトオフセット数値であること。この場合、*visit*は `nil`でなければならない。たとえば、

```
(insert-file-contents filename nil 0 500)
```

これはファイルの先頭 500 文字 (バイト) を挿入する。

引数 *replace*が非 `nil`なら、それはバッファーのコンテンツ (実際にはアクセス可能な範囲) をファイルのコンテンツで置き換えることを意味する。これは単にバッファーのコンテンツを削除してファイル全体を挿入するより優れている。なぜなら、(1) マーカー位置を維持して、(2)undo リストに配置するデータも少ないからである。

*replace*と *visit*が `nil`なら、`insert-file-contents`で (FIFO や I/O デバイスのような) スペシャルファイルの読み取りが可能。

`insert-file-contents-literally` *filename* **&optional** *visit beg end replace* [Function]

この関数は `insert-file-contents`のように機能するが、`after-insert-file-functions`を実行せずフォーマットのデコード、文字コード変換、自動解凍、... などを行わない点が異なる。

他のプログラムがファイルを読めるように他プロセスにファイル名を渡したければ関数 `file-local-copy`を使用します。Section 25.12 [Magic File Names], page 569 を参照してください。

25.4 ファイルへの書き込み

関数 `append-to-file`と `write-region`を使用することによってディスク上のファイルにバッファーのコンテンツやバッファーの一部を直接書き込むことができます。visit されているファイルに書き込むためにこれらの関数を使用しないでください。これによって visit にたいするメカニズムが混乱するかもしれません。

`append-to-file` *start end filename* [Command]

この関数はカレントバッファー内で *start*と *end*によるリージョンのコンテンツをファイル *filename*の終端に追加する。そのファイルが存在しなければ作成する。この関数は `nil`をリターンする。

*filename*の書き込みや作成ができなければエラーがシグナルされる。

Lisp から呼び出した場合、この関数は以下と完全に等価:

```
(write-region start end filename t)
```

`write-region` *start end filename* **&optional** *append visit lockname mustbenew* [Command]

この関数はカレントバッファー内の *start*と *end*で区切られたリージョンを *filename*で指定されたファイルに書き込む。

*start*が `nil`なら、このコマンドはバッファーのコンテンツ全体 (アクセス可能な範囲だけではない) をファイルに書き込んで *end*は無視する。

*start*が文字列なら、`write-region`はバッファーのテキストではなくその文字列を追加する。その場合には *end*は無視される。

*append*が非 `nil`なら、指定されたテキストが (もしあれば) 既存のファイルコンテンツに追加される。*append*が数字なら `write-region`はファイル開始位置からそのバイトオフセットを seek してデータをそこに書き込む。

*mustbenew*が非 `nil`の場合には、*filename*が既存ファイルの名前なら `write-region`は確認を求める。*mustbenew*がシンボル `excl`なら、ファイルがすでに存在する場合には `write-region`は確認を求めるかわりにエラー `file-already-exists`をシグナルする。たとえ `write-region`が通常はシンボリックリンクをフォローして、もしシンボリックリンクが壊れていれば pointed-to ファイル (訳注: ファイルへのハードリンクにたいするポインター) を作成するとしても、*mustbenew*が `excl`ならシンボリックリンクをフォローしない。

*mustbenew*が `excl`のときは、存在するファイルのテストに特別なシステム機能を使用する。少なくともローカルディスク上のファイルにたいしては、Emacs がファイルを作成する前に Emacs に通知せずに他のプログラムが同じ名前のファイルを作成することはありえない。

*visit*が `t`なら、Emacs はバッファーとファイルの関連付けを設定してそのバッファーがそのファイルを victi する。またカレントバッファーにたいする最終ファイル変更日時に *filename*をセットして、そのバッファーを未変更としてマークする。この機能は `save-buffer`により使用されるが、おそらくあなた自身が使用するべきではないだろう。

*visit*が文字列なら、それは visit するファイルの名前を指定する。この方法を使えば、そのバッファーが別のファイルを visit していると記録しつつ 1 つのファイル (*filename*) にデータを書き込むことができる。引数 *visit*はエコーエリアに使用される他にファイルのロックにも使用され、*visit*が `buffer-file-name`に格納される。この機能は `file-precious-flag`の実装に使用される。自分が何をしているか本当にわかっているのでなければこれを使用してはならない。

オプション引数 *lockname*が非`nil`なら、それはロックとアンロックの目的に使用する *filename* と *visit*をオーバーライドするファイル名を指定する。

関数 `write-region`は書き込むデータを `buffer-file-format`によって指定される適切なファイルフォーマットに変換しするとともに、リスト `write-region-annotate-functions` 内の関数の呼び出しも行う。Section 25.13 [Format Conversion], page 573 を参照のこと。

`write-region`は通常はエコーエリア内にメッセージ '`Wrote` *`filename`*'を表示する。*visit* が `t`、`nil`、文字列のいずれでもない場合、こまたは Emacs が batch モード (Section 40.17 [Batch Mode], page 1121 を参照) で処理中ならこのメッセージは抑制される。この機能は内部的な目的のためにユーザーが知る必要がないファイルを使用したり、Emacs が batch モードで処理中に有用である。

`write-region-inhibit-fsync` [Variable]

この変数の値が `nil`なら `write-region`はファイル書き込み後にシステムコール `fsync`を使用する。これはたとえ Emacs を低速化するとしても、電源喪失時のデータ損失リスクを軽減する。値が `t`なら Emacs は `fsync`を使用しない。デフォルト値は Emacs が対話的に実行されていれば `nil`、batch モードで実行時には `t`。Section 25.8 [Files and Storage], page 554 を参照のこと。

`with-temp-file` *file body...* [Macro]

`with-temp-file`マクロは一時バッファー (temporary buffer) をカレントバッファーとして *body*フォームを評価して、最後にそのバッファーのコンテンツを *file*に書き込む。これは終了時に一時バッファーを kill して、`with-temp-file`フォームの前にカレントだったバッファーをリストアする。その後に *body*内の最後のフォームの値をリターンする。

`throw`やエラーによる異常な exit(abnormal exit) でも、カレントバッファーはリストアされる (Section 11.7 [Nonlocal Exits], page 162 を参照)。

[The Current Buffer], page 592 の `with-temp-buffer`も参照のこと。

25.5 ファイルのロック

2 人のユーザーが同時に同じファイルを編集する際には、彼らはおそらく互いに干渉しあうでしょう。Emacs はファイルが変更される際にファイルロック *(file lock)* を記録することにより、このような状況の発生を防ぎます。そして Emacs は他の Emacs ジョブにロックされているファイルを visit しているバッファーへの変更の最初の試みを検知して、ユーザーに何を行うべきかを尋ねます。このファイルロックの実態は、編集中のファイルと同じディレクトリーに格納される特別な名前をもつシンボリックリンクです。この名前はバッファーのファイル名に`.#`を前置することにより構築されます。シンボリックリンクのターゲットは *`user@host.pid:boot`*という形式になります。ここで *user*はカレントのユーザー名 (`user-login-name`かより得)、*host*は Emacs を実行中のホスト (`system-name`より取得)、*pid*は Emacs のプロセス ID、*boot*は最後のブートからの経過時間です。ブート時刻が利用できなければ:*`boot`*は省略されます。(シンボリックリンクをサポートしないファイルシステムでは、*`user@host.pid:boot`*という形式を構築するか割りに通常のファイルが使用される)。

ファイルのアクセスに NFS を使用する際には、可能性は小さいものの他のユーザーと同じファイルを同時にロックするかもしれません。これが発生すると 2 人のユーザーが同時にファイルを変更することが可能になりますが、それでも Emacs は 2 番目に保存するユーザーにたいして警告を発するでしょう。たファイルを visit しているバッファーでディスク上でファイルの変更を検知することにより、ある種の同時編集を捕捉できます。Section 27.6 [Modification Time], page 597 を参照してください。

`file-locked-p` *filename* [Function]
この関数はファイル *filename*がロックされていなければ `nil`をリターンする。この Emacs プロセスによりロックされていれば `t`、他の Emacs ジョブによりロックされている場合はロックしたユーザーの名前をリターンする。

```
(file-locked-p "foo")
     ⇒ nil
```

`lock-buffer` **&optional** *filename* [Function]
この関数はカレントバッファーが変更されていればファイル *filename*をロックする。引数 *filename*のデフォルトはカレントバッファーが visit しているファイル。カレントバッファーがファイルを visit していない、バッファーが変更されていない、または `create-lockfiles`が `nil`なら何もしない。

`unlock-buffer` [Function]
この関数はカレントバッファーが変更されていればバッファーにより visit されているファイルをアンロックする。バッファーが変更されていなければ、そのファイルをロックしてはならないのでこの関数は何もしない。カレントバッファーがファイルを visit していない、またはファイルがロックされていなければこの関数は何もしない。

`create-lockfiles` [User Option]
この変数が `nil`なら Emacs はファイルをロックしない。

`ask-user-about-lock` *file other-user* [Function]
この関数はユーザーが *file*の変更を試みたが、それが名前 *other-user*のユーザーにロックされていたとき呼び出される。この関数のデフォルト定義は何を行うかユーザーに尋ねる関数。この関数がリターンする値は Emacs が次に何を行うかを決定する:

- 値 `t`はそのファイルのロックを奪うことを意味する。その場合には *other-user*はロックを失い、そのユーザーがファイルを編集することができる。
- 値 `nil`はロックを無視して、とにかくユーザーがファイルを編集できるようにすることを意味する。
- この関数はかわりにエラー `file-locked`をシグナルする。この場合には、ユーザーが行おうとしていた変更は行われない。

  このエラーにたいするエラーメッセージは以下のようになる:

  error `File is locked: file other-user`

  ここで `file`はファイル名、*other-user*はそのファイルのロックを所有するユーザーの名前。

望むなら他の方法で判定を行う独自バージョンで `ask-user-about-lock`関数を置き換えることができる。

25.6 ファイルの情報

このセクションではファイル (またはディレクトリーやシンボリックリンク) に関してファイルが読み込み可能か、書き込み可能か、あるいはファイルのサイズのようなさまざまなタイプの情報を取得する関数を説明します。これらの関数はすべて引数にファイルの名前を受け取ります。特に注記した場合を除きこれらの引数には既存のファイルを指定する必要があり、ファイルが存在しなければエラーをシグナルします。

スペースで終わるファイル名には気をつけてください。いくつかのファイルシステム (特に MS-Windows) では、ファイル名の末尾の空白文字は暗黙かつ自動的に無視されます。

25.6.1 アクセシビリティのテスト

以下の関数はファイルにたいする読み取り、書き込み、実行のパーミッションをテストします。これらの関数は明示しない限りボリックリンクをフォローします。Section 25.6.2 [Kinds of Files], page 542を参照してください。

いくつかのオペレーティングシステムではACL(Access Control Lists: アクセス制御リスト)のような機構を通じて、より複雑なアクセスパーミッションセットが指定できます。それらのパーミッションにたいする問い合わせやセットの方法についてはSection 25.6.5 [Extended Attributes], page 548を参照してください。

`file-exists-p` *filename* [Function]

この関数はファイル名 *filename*が存在しているようなら `t`をリターンする。これはファイルが読み取り可能である必要はなく、ファイルの属性を調べることが恐らく可能なこと意味する(Unix やその他の POSIX システム類ではファイルが存在してファイルを含むディレクトリーの実行パーミッションをもつ場合にはファイル自体のパーミッションに関わらず `t`)。

ファイルが存在しない、またはファイルが存在するかどうかの判断に問題があるようなら、この関数は `nil`をリターンする。

ディレクトリーはファイルなので、ディレクトリーが与えられると `file-exists-p`は `t`をリターンするかもしれない。しかし `file-exists-p`はシンボリックリンクをフォローするので、ターゲットファイルが存在する場合のみシンボリックリンク名にたいして `t`をリターンする。

`file-readable-p` *filename* [Function]

この関数は *filename*という名前のファイルが存在して、それを読み取ることが可能なら `t`、それ以外は `nil`をリターンする。

`file-executable-p` *filename* [Function]

この関数は、*filename*という名前のファイルが存在して、それを実行することが可能なら `t`をリターンする。それ以外は `nil`をリターンする。Unix やその他の POSIX システム類ではファイルがディレクトリーなら実行パーミッションはディレクトリー内のファイルの存在と属性をチェックできるので、ファイルのモードが許せばオープンできることを意味する。

`file-writable-p` *filename* [Function]

この関数は *filename*という名前のファイルが書き込み可能か作成可能可能なら `t`、それ以外は `nil`をリターンする。ファイルが存在してそれに書き込むことができるならファイルは書き込み可能。ファイルは存在しないが親ディレクトリーが存在して、そのディレクトリーに書き込むことができるなら書き込み可能。

以下の例では、`foo`は親ディレクトリーが存在しないので、たとえユーザーがそのディレクトリーを作成可能であってもファイルは書き込み可能ではない。

```
(file-writable-p "~/no-such-dir/foo")
     ⇒ nil
```

`file-accessible-directory-p` *dirname* [Function]

この関数はファイルとしての名前が *dirname*であるようなディレクトリー内にある既存のファイルをオープンするパーミッションをもつ場合は `t`、それ以外 (そのようなディレクトリーが存在しない場合) は `nil`をリターンする。*dirname*の値はディレクトリー名 (`/foo/`など)、または名前がディレクトリー (最後のスラッシュがない`/foo`など) であるようなファイル。

たとえば以下では`/foo/`内の任意のファイルを読み取る試みはエラーになると推測できる:

```
(file-accessible-directory-p "/foo")
     ⇒ nil
```

`access-file` *filename string* [Function]
この関数は *filename*が読み取り可能なら `nil`、それ以外なら *string*をエラーメッセージのテキストに使用してエラーをシグナルする。

`file-ownership-preserved-p` *filename* **&optional** *group* [Function]
この関数はファイル *filename*を削除後に新たに作成してもファイルの所有者が変更されずに維持されるようなら `t`をリターンする。これは存在しないファイルにたいしても `t`をリターンする。

オプション引数 *group*が非 `nil`なら、この関数はファイルのグループが変更されないこともチェックする。

この関数はシンボリックリンクをフォローしない。

`file-modes` *filename* [Function]
この関数は *filename*のモードビット *(mode bits)* をリターンする。これは読み取り、書き込み、実行パーミッションを要約する整数。この関数はシンボリックリンクを許容する。ファイルが存在しなければリターン値は `nil`。

モードビットの説明は Section “File permissions” in *The* GNU `Coreutils` *Manual* を参照のこと。たとえば最下位ビットが 1 ならそのファイルは実行可能、2 ビット目が 1 なら書き込み可能、... となる。設定できる最大の値は 4095(8 進の 7777) であり、これはすべてのユーザーが読み取り、書き込み、実行のパーミッションをもち、他のユーザーとグループにたいして SUIDビット、および sticky ビットがセットされる。

これらのパーミッションのセットに使用される `set-file-modes`関数については Section 25.7 [Changing Files], page 550 を参照のこと。

```
(file-modes "~/junk/diffs")
     ⇒ 492               ; 10 進整数
(format "%o" 492)
     ⇒ "754"             ; 8 進に変換した値

(set-file-modes "~/junk/diffs" #o666)
     ⇒ nil

$ ls -l diffs
-rw-rw-rw- 1 lewis lewis 3063 Oct 30 16:00 diffs
```

MS-DOS にたいする注意**:** MS-DOS では実行可能であることをを表すようなファイルのモードビットは存在しない。そのため `file-modes`はファイル名が`.com`、`.bat`、`.exe`などのような標準的な実行可能な拡張子のいずれかで終わる場合にはファイルを実行可能であると判断する。POSIX 標準の ‘`#!`’署名で始まる shell スクリプトや Perl スクリプトも実行可能と判断される。POSIX との互換性のためにディレクトリーも実行可能と報告される。`file-attributes` (Section 25.6.4 [File Attributes], page 545 を参照) もこれらの慣習にしたがう。

25.6.2 ファイル種別の区別

このセクションではディレクトリー、シンボリックリンク、および通常ファイルのようなさまざまな種類のファイルを区別する方法を説明します。

シンボリックリンクは通常はそれが出現した場合には常にフォローされます。たとえばファイル名 `a/b/c`を解釈するためにシンボリックリンクかもしれない `a`、`a/b`、`a/b/c`はすべてフォローされて、リンクターゲット自身がシンボリックリンクなら再帰的にフォローされるかもしれません。しかしい

くつかの関数は最後のファイル名 (この例では `a/b/c`) ではシンボリックリンクをフォローしません。そのような関数のことをシンボリックリンク非フォロー *(not follow symbolic links)* と呼びます。

`file-symlink-p` *filename* [Function]

ファイル *filename*がシンボリックリンクなら `file-symlink-p`関数はリンクをフォローせずに、かわりにリンクターゲットを文字列としてリターンする (リンクターゲット文字列はターゲットの完全な絶対ファイル名である必要はない。リンクが指すのが完全なファイル名かどうかを判断するのは簡単な処理ではない。以下参照)。

`file-symlink-p`はファイル *filename*がシンボリックリンクではないか、あるいは存在しなかったりシンボリックリンクかどうか判断するのに問題がある場合には `nil`をリターンする。

この関数の使用例をいくつか示す:

```
(file-symlink-p "not-a-symlink")
     ⇒ nil
(file-symlink-p "sym-link")
     ⇒ "not-a-symlink"
(file-symlink-p "sym-link2")
     ⇒ "sym-link"
(file-symlink-p "/bin")
     ⇒ "/pub/bin"
```

3 つ目の例では関数は `sym-link`をリターンするが、たとえそれ自体がシンボリックリンクであってもリンク先の解決を行わないことに注意。これは関数がシンボリックリンクをフォローしない、すなわちシンボリックリンクをフォローする処理はファイル名の最後のコンポーネントには適用されないからである。

この関数がリターンするのはそのシンボリックリンクに何が記録されているかを示す文字列であり、それにディレクトリー部分が含まれているかどうかは構わない。この関数は完全修飾されたファイル名を生成するためにリンクターゲットを展開しないし、リンクターゲットが絶対ファイル名でなければ、(もしあっても)*filename*引数のディレクトリー部分は使用しない。以下に例を示す:

```
(file-symlink-p "/foo/bar/baz")
     ⇒ "some-file"
```

ここでは、たとえ与えられた`/foo/bar/baz`が完全修飾されたファイル名であるにも関わらずその結果は異なり、実際には何のディレクトリー部分ももたない。`some-file`自体がシンボリックリンクかもしれないので、単にその前に先行ディレクトリーを追加することはできず、絶対ファイル名を生成するために単に `expand-file-name` (Section 25.9.4 [File Name Expansion], page 559 を参照) を使用することもできないからである。

この理由により、あるファイルがシンボリックリンクか否かという単一の事実よりも多くを判定する必要がある場合にこの関数が有用であることは稀である。実際にリンクターゲットのファイル名が必要なら、Section 25.6.3 [Truenames], page 544 で説明する `file-chase-links`や `file-truename`を使用すること。

`file-directory-p` *filename* [Function]

この関数は *filename*が既存のディレクトリー名なら `t`。*filename*がディレクトリー名ではない、あるいはディレクトリーかどうかの判断に問題がある場合には `nil`をリターンする。この関数はシンボリックリンクをフォローする。

```
(file-directory-p "~rms")
     ⇒ t
```

```
(file-directory-p "~rms/lewis/files-ja.texi")
     ⇒ nil
(file-directory-p "~rms/lewis/no-such-file")
     ⇒ nil
(file-directory-p "$HOME")
     ⇒ nil
(file-directory-p
 (substitute-in-file-name "$HOME"))
     ⇒ t
```

`file-regular-p` *filename* [Function]

この関数はファイル *filename*が存在して、それが通常ファイル (ディレクトリー、名前付きパイプ、端末、その他 I/O デバイス以外) なら `t`をリターンする。*filename*が存在しない、通常ファイルではない、あるいは通常ファイルかどうかの判断に問題がある場合には `nil`をリターンする。この関数はシンボリックリンクをフォローする。

25.6.3 本当の名前

ファイルの実名 *(truename)* とは、全階層においてシンボリックリンクを残らずフォローした後に名前コンポーネントに出現する '`.`'と '`..`'を除いて簡略化した名前のことです。これはそのファイルにたいする正規名 (canonical name) の一種です。ファイルが常に一意な実名をもつ訳ではありません。あるファイルにたいする異なる実名の個数は、そのファイルにたいするハードリンクの個数と同じです。しかし実名はシンボリックリンクによる名前の変動を解消するのに有用です。

`file-truename` *filename* [Function]

この関数はファイル *filename*の実名をリターンする。引数が絶対ファイル名でなければ、この関数は最初に `default-directory`にたいしてこれを展開する。

この関数は環境変数を展開しない。これを行うのは `substitute-in-file-name`のみ。[Definition of substitute-in-file-name], page 560 を参照のこと。

名前コンポーネントに出現する '`..`'に先行するシンボリックリンクリンクをフォローする必要がある場合には直接と間接を問わず、`expand-file-name`を呼び出す前に `file-truename`を呼び出すこと。そうしないと '`..`'の直前にある名前コンポーネントは、`file-truename`が呼び出される前の簡略化により取り除かれてしまう。`expand-file-name`呼び出しの必要を無くすために、`file-truename`は `expand-file-name`が行うのと同じ方法で '`~`'を扱う。Section 25.9.4 [Functions that Expand Filenames], page 559 を参照のこと。

シンボリックリンクのターゲットがリモートファイル名の構文をもつ場合には、`file-truename`はそれをクォートしてリターンする。Section 25.9.4 [Functions that Expand Filenames], page 559 を参照のこと。

`file-chase-links` *filename* **&optional** *limit* [Function]

この関数は *filename*で始まるシンボリックリンクを、シンボリックリンクではない名前のファイル名までフォローして、そのファイル名をリターンする。この関数は親ディレクトリーの階層にあるシンボリックリンクをフォローしない。

*limit*に数を指定するとその数のリンクを追跡した後、この関数はたとえそれが依然としてシンボリックリンクであってもそれをリターンする。

`file-chase-links`と `file-truename`の違いを説明するために、`/usr/foo`がディレクトリー `/home/foo`へのシンボリックリンク、`/home/foo/hello`が (少なくともシンボリックリンクではない) 通常ファイル、または存在しないファイルだとします。この場合には以下のようになります:

```
(file-chase-links "/usr/foo/hello")
     ;; 親ディレクトリーのリンクはフォローしない
     ⇒ "/usr/foo/hello"
(file-truename "/usr/foo/hello")
     ;; /homeはシンボリックリンクではないと仮定
     ⇒ "/home/foo/hello"
```

`file-equal-p` *file1 file2* [Function]

この関数はファイル *file1* と *file2* の名前が同じファイルなら `t` をリターンする。これはリモートファイル名も適切な方法で処理することを除いて実名の比較と似ている。*file1* か *file2* が存在しなければリターン値は不定。

`file-name-case-insensitive-p` *filename* [Function]

ファイル名やその一部にたいして文字列としての比較を要する場合には、背景にあるファイルシステムが非 case センシティブ (case-insensitive: 大文字小文字を区別しない) かどうかを知ることが重要になる。この関数はファイル *filename* が非 case センシティブなファイルシステムにあれば `t` をリターンする。MS-DOS と MS-Windows では常に `t` をリターンする。Cygwin と macOS では非 case センシティブかもしれないので、実行時テストにより case センシティブ性の判定を試みる。テストで決定されなければ Cygwin なら `t`、macOS なら `nil` をリターンする。

この関数は現在のところ MS-DOS、MS-Windows、Cygwin、macOS 以外のプラットフォームでは常に `nil` をリターンする。これは Samba 共有や NFS マウントされた Windows ボリュームのようにマウントされたファイルシステムの case センシティブ性は検知しない。リモートホストでは '`smb`' メソッドにたいしては `t` とみなす。

`file-in-directory-p` *file dir* [Function]

この関数は、*file* がディレクトリー *dir* 内のファイルかサブディレクトリーなら `t` をリターンする。また *file* と *dir* が同じディレクトリーの場合も `t` をリターンする。この関数は 2 つのディレクトリーの実名を比較する。*dir* が既存のディレクトリーの名前でなければリターン値は `nil`。

`vc-responsible-backend` *file* [Function]

この関数は与えられた *file* にたいして VC バックエンドが責任を負うかどうかを判断する。たとえば `emacs.c` が Git でトラック (track: 追跡) されていれば `(vc-responsible-backend "emacs.c")` は '`Git`' をリターンする。*file* がシンボリックリンクなら `vc-responsible-backend` はシンボリックリンクを解決せずに、シンボリックリンクにたいするバックエンドが報告されることに注意。*file* が参照するファイルのバックエンド VC を取得するには、`file-chase-links` のようなシンボリックリンク解決用の関数で *file* をラップすること。

```
(vc-responsible-backend (file-chase-links "emacs.c"))
```

25.6.4 ファイルの属性

このセクションではファイルの詳細な情報を取得する関数について説明します。それらの情報にはファイルの所有者やグループの番号、ファイル名の個数、inode 番号、サイズやアクセス日時、変更日時が含まれます。

`file-newer-than-file-p` *filename1 filename2* [Function]

この関数はファイル *filename1* がファイル *filename2* より新しければ `t` をリターンする。*filename1* が存在しなければ `nil`、*filename1* は存在するが *filename2* が存在しなければ `t` をリターンする。

以下の例では、aug-19の書き込みが 19 日、aug-20の書き込みが 20 日、ファイル no-file は存在しないものとする。

```
(file-newer-than-file-p "aug-19" "aug-20")
     ⇒ nil
(file-newer-than-file-p "aug-20" "aug-19")
     ⇒ t
(file-newer-than-file-p "aug-19" "no-file")
     ⇒ t
(file-newer-than-file-p "no-file" "aug-19")
     ⇒ nil
```

`file-attributes` *filename* **&optional** *id-format* [Function]

この関数はファイル *filename*の属性 (attributes) のリストをリターンする。オープンできないファイルが指定された場合は、nilをリターンする。指定されたファイルが存在しなければnilをリターンする。この関数はシンボリックリンクをフォローしない。オプション引数 *id-format*は属性 UIDと GID(以下参照) にたいして望ましいフォーマットを指定する。有効な値は'stringと'integer。デフォルトは'integerだが、わたしたちはこれの変更を計画しているので、リターンされる UIDや GIDを使用する場合には *id-format*にたいして非 nil値を指定すること。

GNU プラットフォームではこの関数はロックファイル処理時はアトミックである。別のプロセスによりファイルシステムが同時に変更された場合には、この関数は変更の前か後のいずれかのファイル属性をリターンする。それ以外ならこの関数はアトミックではなく競合状態を検知したら nil、または以前とカレントが混ざったファイル属性をリターンするかもしれない。

このリスト内の要素にアクセスするためにアクセッサ関数が提供される。このアクセッサ以下の要素の記述とともに示される。

リストの要素は順に:

0. ディレクトリーにたいしては t、シンボリックリンクにたいしては文字列 (リンクされる名前)、テキストファイル (file-attribute-type) にたいしては nil。
1. そのファイルがもつ名前の個数 (file-attribute-link-number)。ハードリンクとして知られる代替え名は関数 add-name-to-fileを使用して作成できる (Section 25.7 [Changing Files], page 550 を参照)。
2. ファイルの UIDであり通常は文字列 file-attribute-user-id。しかし名前をもつユーザーに対応しなければ値は整数。
3. 同様にファイルの GID (file-attribute-group-id)。
4. Lisp のタイムスタンプによる最終アクセス時刻 (file-attribute-access-time)。タイムスタンプは current-time (Section 40.5 [Time of Day], page 1104 を参照) の形式であり、ファイルシステムのタイムスタンプの精度に切り詰められる。たとえば FAT ベースのいくつかのファイルシステムでは最終アクセスの日付だけが記録されるので、この時刻には常に最終アクセス日の真夜中が保持されることに注意。
5. Lisp のタイムスタンプによる最終変更時刻 (file-attribute-modification-time)。これはファイルのコンテンツが変更された最終時刻。
6. Lisp のタイムスタンプによるステータスの最終変更時刻 (file-attribute-status-change-time。上記参照)。これはファイルのアクセスモードビット、所有者とグループ、およびファイルにたいしてファイルのコンテンツ以外にファイルシステムが記録するその他の情報にたいする最終変更時刻。
7. ファイルのバイトサイズ (file-attribute-size)。

8. ‘`ls -l`’で表示されるような 10 個の文字、またはダッシュからなる文字列で表されるファイルのモード (`file-attribute-modes`)。
9. 後方互換のために提供される不定値。
10. ファイルの inode 番号であり非負の整数 (`file-attribute-inode-number`)。
11. そのファイルがあるデバイスのファイルシステム番号 (`file-attribute-device-number`)。この要素とファイルの inode 番号を併せればシステム上の 2 つを区別するのに足る情報が得られる (2 つのファイルがこれら両方の番号で同じ値をもつことはできない)。

たとえば以下は`files-ja.texi`のファイル属性:

```
(file-attributes "files-ja.texi" 'string)
     ⇒  (nil 1 "lh" "users"
         (20614 64019 50040 152000)
         (20000 23 0 0)
         (20614 64555 902289 872000)
         122295 "-rw-rw-rw-"
         t 6473924464520138
         1014478468)
```

この結果を解釈すると:

`nil` ディレクトリーでもシンボリックリンクでもない。

`1` (カレントデフォルトディレクトリー内で名前`files-ja.texi`は) 単一の名前をもつ。

`"lh"` 名前"lh"のユーザーにより所有される。

`"users"` 名前"users"のグループ。

`(20614 64019 50040 152000)`
最終アクセスが October 23, 2012, at 20:12:03.050040152 UTC。

`(20000 23 0 0)`
最終更新が July 15, 2001, at 08:53:43 UTC。

`(20614 64555 902289 872000)`
最終ステータス変更が October 23, 2012, at 20:20:59.902289872 UTC。

`122295` バイト長は 122295 バイト (しかしマルチバイトシーケンスが含まれていたり、EOL フォーマットが CRLF なら 122295 文字は含まれないかもしれない)。

`"-rw-rw-rw-"`
所有者、グループ、その他にたいして読み取り、書き込みアクセスのモードをもつ。

`t` 単なるプレースホルダーであり何の情報ももたない。

`6473924464520138`
inode 番号は 6473924464520138。

`1014478468`
ファイルシステムのデバイス番号は 1014478468。

`file-nlinks` *filename* [Function]

この関数はファイル *filename*がもつ名前 (ハードリンク) の個数をリターンする。ファイルが存在しなければ、この関数は `nil`をリターンする。シンボリックリンクはリンク先のファイルの名前とは判断されないので、この関数に影響しないことに注意。この関数はシンボリックリンクをフォローしない。

```
$ ls -l foo*
-rw-rw-rw- 2 rms rms 4 Aug 19 01:27 foo
-rw-rw-rw- 2 rms rms 4 Aug 19 01:27 foo1

(file-nlinks "foo")
     ⇒ 2
(file-nlinks "doesnt-exist")
     ⇒ nil
```

25.6.5 拡張されたファイル属性

いくつかのオペレーティングシステムでは、それぞれのファイルを任意の拡張ファイル属性 *(extended file attributes)* に関連付けることができます。現在のところ Emacs は拡張ファイル属性のうち 2 つの特定セット (ACL: Access Control List、および SELinux コンテキスト) にたいする問い合わせと設定をサポートします。これらの拡張ファイル属性は、前のセクションで議論した Unix スタイルの基本的なパーミッションより洗練されたファイルアクセス制御を強制するために、いくつかのシステムで利用されます。

ACL と SELinux についての詳細な解説はこのマニュアルの範囲を超えています。わたしたちの目的のためには、それぞれのファイルは *ACL* (ACL ベースのファイル制御システムの元で ACL のプロパティを指定) および/または *SELinux* コンテキスト (SELinux システムの元で SELinux のプロパティを指定) に割り当てることができるという理解で問題ないでしょう。

`file-acl` *filename* [Function]

この関数はファイル *filename*にたいする ACL をリターンする。ACL にたいする正確な Lisp 表現は不確定 (かつ将来の Emacs バージョンで変更され得る) だが、これは `set-file-acl`が引数 *acl*にとる値と同じである (Section 25.7 [Changing Files], page 550 を参照)。

根底にある ACL 実装はプラットフォームに固有である。Emacs は GNU/Linux と BSD では POSIX ACL インターフェイスを使用して、MS-Windows ではネイティブのファイルセキュリティ API を POSIX ACL インターフェイスでエミュレートする。

ACL がサポートされていない、あるいはファイルが存在しなければリターン値は `nil`。

`file-selinux-context` *filename* [Function]

この関数はファイル *filename*の SELinux コンテキストを `(`*`user role type range`*`)`という形式のリストでリターンする。リストの要素はそのコンテキストのユーザー、ロール、タイプ、レンジを文字列として表す値である。これらの実際の意味についての詳細は SELinux のドキュメントを参照のこと。リターン値は `set-file-selinux-context`が *context*引数で受け取るのと同じ形式 (Section 25.7 [Changing Files], page 550 を参照)。

SELinux をサポートしない、あるいはファイルが存在しなければリターン値は `(nil nil nil nil)`。

`file-extended-attributes` *filename* [Function]

この関数は Emacs が認識するファイル *filename*の拡張属性を alist でリターンする。現在のところこの関数は ACL と SELinux の両方を取得するための便利な方法としての役目を果た

す。他のファイルに同じファイルアクセス属性を適用するためにリターンされた alist を 2 つ目の引数として `set-file-extended-attributes`を呼び出すことができる (Section 25.7 [Changing Files], page 550 を参照)。

要素のうちの 1 つは `(acl . acl)`で、*acl*は `file-acl`がリターンするのと同じ形式。

他の要素は `(selinux-context . context)`で、*context*は `file-selinux-context`がリターンするのと同じ形式。

25.6.6 標準的な場所へのファイルの配置

このセクションではディレクトリーのリスト (パス *(path)*) からファイルを検索したり、標準の実行可能ファイル用ディレクトリーから実行可能ファイルを検索する方法を説明します。

ユーザー固有の設定ファイル (configuration file) の検索については Section 25.9.7 [Standard File Names], page 565 の関数 `locate-user-emacs-file`を参照してください。

`locate-file` *filename path* **&optional** *suffixes predicate* [Function]

この関数は *path*で与えられるディレクトリーリスト内で *filename*という名前のファイルを検索して、*suffixes*内のサフィックスの検索を試みる。そのようなファイルが見つかったらファイルの絶対ファイル名 (Section 25.9.2 [Relative File Names], page 557 を参照)、それ以外は `nil`をリターンする。

オプション引数 *suffixes*は検索時に *filename*に追加するファイル名サフィックスのリストを与える。`locate-file`は検索するディレクトリーごとにそれらのサフィックスを試みる。*suffixes* が `nil`や `("")`なら、サフィックスなしで *filename*だけがそのまま使用される。*suffixes*の典型的な値は `exec-suffixes` (Section 38.1 [Subprocess Creation], page 934 を参照)、`load-suffixes`、`load-file-rep-suffixes`、および関数 `get-load-suffixes` (Section 16.2 [Load Suffixes], page 271 を参照)。

実行可能プログラムを探すときは `exec-path` (Section 38.1 [Subprocess Creation], page 934 を参照)、Lisp ファイルを探すときは `load-path` (Section 16.3 [Library Search], page 272 を参照) が *path*の典型的な値である。*filename*が絶対ファイル名なら *path*の効果はないが、サフィックスにたいする *suffixes*は依然として試行される。

オプション引数 *predicate*が非 `nil`なら、それは候補ファイルが適切かどうかテストする述語関数を指定する。述語関数には単一の引数として候補ファイル名が渡される。*predicate*が `nil`か省略なら述語として `file-readable-p`を使用する。`file-executable-p`や `file-directory-p`など、その他の有用な述語については Section 25.6.2 [Kinds of Files], page 542 を参照のこと。

この関数は通常はディレクトリーをスキップするので、ディレクトリーを探したければ、*predicate*関数がディレクトリーにたいして確実に `dir-ok`をリターンすること。たとえば:

```
(locate-file "html" '("/var/www" "/srv") nil
             (lambda (f) (if (file-directory-p f) 'dir-ok)))
```

互換性のために *predicate*には `executable`、`readable`、`writable`、`exists`、またはこれらシンボルの 1 つ以上のリストも指定できる。

`executable-find` *program* **&optional** *remote* [Function]

この関数は *program*という名前の実行可能ファイルを検索して、その実行可能ファイルの絶対ファイル名と、もしあればファイル名の拡張子も含めてリターンする。ファイルが見つからなければ `nil`をリターンする。この関数は `exec-path`内のすべてのディレクトリーを検索して、`exec-suffixes`内のすべてのファイル名拡張子の検索も試みる (Section 38.1 [Subprocess Creation], page 934 を参照)。

*remote*が `nil`、かつ非 `default-directory`がリモートディレクトリーなら、*program*は各リモートホスト上で検索される。

25.7 ファイルの名前と属性の変更

このセクションの関数はファイルのリネーム、コピー、削除やリンク、モードをセットします。これらの関数は処理に失敗すると、通常は失敗の理由を記述するシステム依存のエラーメッセージを報告する `file-error`エラーをシグナルします。ファイルが存在しないために失敗すると、かわりに `file-missing`エラーをシグナルします。

性能的な理由によりオペレーティングシステムはこれらの関数により行われた変更を 2 次記憶装置に書き込むかわりに、キャッシュしたりエイリアスするかもしれません。Section 25.8 [Files and Storage], page 554 を参照してください。

引数 *newname*をもつ関数では、それがディレクトリー名なら source 名の非ディレクトリー部分が追加されたかのように扱われます。ディレクトリー名は通常は '`/`'で終端されます (Section 25.9.3 [Directory Names], page 558 を参照)。たとえば *oldname*が `a/b/c`で *newname*が `d/e/f/`なら `d/e/f/c`であるかのように処理されます。*newname*がディレクトリー名ではなくディレクトリーであるような名前なら、この特別な扱いは適用されません。たとえば *newname*の `d/e/f`がディレクトリーでも、そのまま処理されます。

*newname*という引数をもつ関数では、*newname*という名前のファイルが既に存在する場合には、その挙動が引数 *ok-if-already-exists*の値に依存します。

- *ok-if-already-exists*が `nil`なら `file-already-exists`エラーがシグナルされる。
- *ok-if-already-exists*が数字なら確認を求める。
- *ok-if-already-exists*が他の値なら確認なしで古いファイルを置き換える。

`add-name-to-file` *oldname newname* **&optional** *ok-if-already-exists* [Command]

この関数は、*oldname*という名前のファイルに *newname*という名前を追加で与える。これは *newname*という名前が *oldname*にたいする新たなハードリンクになることを意味する。

*newname*がシンボリックリンクなら、リンクが指すディレクトリーエントリーではなく、シンボリックリンクリンクのディレクトリーエントリーが置き換えられる。*oldname*がシンボリックリンクなら、この関数はリンクをフォローする可能性があるが、GNU プラットフォームではリンクをフォローしない。*oldname*がディレクトリーなら、たとえ処理を正常に行ってツリー構造ではないファイルシステムを作成できる古い様式の非 GNU プラットフォームのスーパーユーザーでも、この関数は通常は失敗する。

以下の例の最初の部分では 2 つのファイル `foo`と `foo3`をリストする。

```
$ ls -li fo*
81908 -rw-rw-rw- 1 rms rms 29 Aug 18 20:32 foo
84302 -rw-rw-rw- 1 rms rms 24 Aug 18 20:31 foo3
```

ここで `add-name-to-file`を呼び出してハードリンクを作成して再度ファイルをリストする。このリストには 1 つのファイルにたいして 2 つの名前 `foo`と `foo2`が表示される。

```
(add-name-to-file "foo" "foo2")
     ⇒ nil

$ ls -li fo*
81908 -rw-rw-rw- 2 rms rms 29 Aug 18 20:32 foo
81908 -rw-rw-rw- 2 rms rms 29 Aug 18 20:32 foo2
84302 -rw-rw-rw- 1 rms rms 24 Aug 18 20:31 foo3
```

最後に以下を評価する:

```
(add-name-to-file "foo" "foo3" t)
```

そしてファイルを再度リストする。今度は 1 つのファイルにたいして 3 つの名前 `foo`、`foo2`、`foo3`がある。`foo3`の古いコンテンツは失われた。

```
(add-name-to-file "foo1" "foo3")
     ⇒ nil

$ ls -li fo*
81908 -rw-rw-rw- 3 rms rms 29 Aug 18 20:32 foo
81908 -rw-rw-rw- 3 rms rms 29 Aug 18 20:32 foo2
81908 -rw-rw-rw- 3 rms rms 29 Aug 18 20:32 foo3
```

この関数は 1 つのファイルにたいして複数の名前をもつことが許されないオペレーティングシステムでは無意味である。いくつかのシステムでは、かわりにファイルをコピーすることにより複数の名前を実装している。

Section 25.6.4 [File Attributes], page 545 の `file-nlinks`も参照のこと。

`rename-file` *filename newname* **&optional** *ok-if-already-exists* [Command]

このコマンドは *filename*を *newname*にリネームする。

*filename*が *filename*とは別に追加の名前をもつ場合には、それらは自身の名前をもち続ける。実際のところ `add-name-to-file`で名前 *newname*を追加してから *filename*を削除するのは、瞬間的な遷移状態とエラーの処理、ディレクトリーとシンボリックリンクを別とすればリネームと同じ効果がある。

このコマンドはシンボリックリンクをフォローしない。*filename*がシンボリックリンクなら、このコマンドはリンクが指すファイルではなくシンボリックリンクをリネームする。*newname*がシンボリックリンクなら、リンクが指すディレクトリーエントリーではなくリンクのディレクトリーエントリーを置き換える。

*filename*と *newname*が同じディレクトリーエントリー (親ディレクトリーが同じでありそのディレクトリー内で同じ名前を与える) なら、このコマンドは何もしない。それ以外なら *filename*と *newname*が同一のファイルを命名する場合には、このコマンドは POSIX 準拠システムでは何も行わなず、いくつかの非 POSIX システムでは *filename*を削除する。

*newname*がすでに存在する場合には *oldname*がディレクトリーなら空のディレクトリー、非ディレクトリーなら非ディレクトリーでなければならない。

`copy-file` *oldname newname* **&optional** *ok-if-already-exists time preserve-uid-gid preserve-extended-attributes* [Command]

このコマンドはファイル *oldname*を *newname*にコピーする。*oldname*が通常のファイルでなければエラーをシグナルする。*newname*がディレクトリーなら最後の名前コンポーネントを保持するようにディレクトリーの中に *oldname*をコピーする。

この関数は *newname*を作成するために壊れたシンボリックリンクをフォローしない点を除いて、シンボリックリンクをフォローする。

*time*が非 `nil`なら、この関数は新たなファイルにたいして古いファイルと同じ最終変更時刻を与える (これはいくつかの限られたオペレーティングシステムでのみ機能する)。時刻のセットでエラーが発生すると、`copy-file`は `file-date-error`エラーをシグナルする。インタラクティブに呼び出された場合には、プレフィックス引数は *time*にたいして非 `nil`値を指定する。

引数 *preserve-uid-gid*が `nil`なら、新たなファイルのユーザーとグループの所有権の決定をオペレーティングシステムに委ねる (通常は Emacs を実行中のユーザー)。*preserve-uid-gid*が非 `nil`なら、そのファイルのユーザーとグループの所有権のコピーを試みる。これはいくつかのオペレーティングシステムで、かつそれを行うための正しいパーミッションをもつ場合のみ機能する。

オプション引数 *preserve-permissions*が非 `nil`なら、この関数は *oldname*のファイルモード (または "パーミッション")、同様に ACL(Access Control List) と SELinux コンテキストを *newname*にコピーする。Section 25.6 [Information about Files], page 540 を参照のこと。

それ以外では、*newname*が既存ファイルならファイルモードは変更されず、新たに作成された場合はデフォルトのファイルパーミッション (以下の `set-default-file-modes`を参照) によりマスクされる。どちらの場合でも ACL や SELinux コンテキストはコピーされない。

`make-symbolic-link` *target linkname* **&optional** *ok-if-already-exists* [Command]

このコマンドは *linkname*という名前で *target*にたいするシンボリックリンクを作成する。これはシェルコマンドと '`ln -s target linkname`'と似ている。*target*は文字列としてのみ扱われる。既存ファイルの名前である必要はない。*ok-if-already-exists*が整数ならインタラクティブな使用を意味しており、*target*文字列内の先頭の '`~`'は展開されて、先頭の '`/:`'は取り除かれる。

*target*が相対ファイル名なら結果となるシンボリックリンクはシンボリックリンクを含むディレクトリーにたいして相対的に解釈される。Section 25.9.2 [Relative File Names], page 557 を参照のこと。

*target*と *linkname*の両方がリモートファイル構文をもち、かつ両者のリモート識別が等価なら、シンボリックリンクは *target*のローカルファイル名部分を指す。

この関数はシンボリックリンクをサポートしないシステムでは利用できない。

`delete-file` *filename* **&optional** *trash* [Command]

このコマンドはファイル *filename*を削除する。ファイルが複数の名前をもつ場合には他の名前で存在し続ける。*filename*がシンボリックリンクなら `delete-file`はシンボリックリンクだけを削除してターゲットは削除しない。

ファイルが存在しない、または削除できない場合には適切な種類の `file-error`エラーがシグナルされる (GNU/およびその他の POSIX 準拠システムではファイルのディレクトリーが書き込み可能ならファイルは削除可能)。

オプション引数 *trash*が非 `nil`、かつ変数 `delete-by-moving-to-trash`が非 `nil`なら、このコマンドはファイルを削除するかわりにシステムの Trash(ゴミ箱) にファイルを移動する。Section "Miscellaneous File Operations" in *The GNU Emacs Manual* を参照のこと。インタラクティブに呼び出された際には、プレフィックス引数がなければ *trash*は `t`、それ以外は `nil`。

Section 25.11 [Create/Delete Dirs], page 568 の `delete-directory`も参照のこと。

`set-file-modes` *filename mode* [Command]

この関数は、*filename*のファイルモード (またはパーミッション) を *mode*にセットする。この関数はシンボリックリンクをフォローする。

非インタラクティブに呼び出された場合には、*mode*は整数でなければならない。その整数の下位 12 ビットだけが使用される。ほとんどのシステムでは意味があるのは下位 9 ビットのみ。*mode*を入力する Lisp 構文を使用できる。たとえば、

```
(set-file-modes #o644)
```

これはそのファイルにたいして所有者による読み取りと書き込み、グループメンバーによる読み取り、その他のユーザーによる読み取り可能であることを指定する。モードビットの仕様の説明は Section “File permissions” in *The* GNU `Coreutils` *Manual* を参照のこと。

インタラクティブに呼び出されると、*mode*は `read-file-modes`(以下参照) を使用してミニバッファーから読み取られる。この場合にはユーザーは整数、またはパーミッションをシンボルで表現する文字列をタイプできる。

ファイルのパーミッションをリターンする関数 `file-modes`については Section 25.6.4 [File Attributes], page 545 を参照のこと。

`set-default-file-modes` *mode* [Function]

この関数は Emacs および Emacs のサブプロセスが新たに作成するファイルにデフォルトのパーミッションをセットする。Emacs により作成されたすべてのファイルはこれらのパーミッション、およびそれらのサブセットとなるパーミッションをもつ (デフォルトファイルパーミッションが実行を許可しても、`write-region`は実行パーミッションを付与しないだろう)。GNU やその他の POSIX 準拠システムでは、デフォルトのパーミッションは ‘`umask`’の値のビット単位の補数で与えられる。すなわち引数 *mode*でセットされた各ビットは Emacs が作成するファイルのデフォルトパーミッション内ではリセットされる。

引数 *mode*は上記の `set-file-modes`と同様、パーミッションを指定する整数であること。意味があるのは下位 9 ビットのみ。

デフォルトのファイルパーミッションは、既存ファイルの変更されたバージョンを保存する際は効果がない。ファイルの保存では既存のパーミッションが保持される。

`with-file-modes` *mode body...* [Macro]

このマクロは新たなファイルにたいするデフォルトのパーミッションを一時的に *modes* (値は)`set-file-modes`にたいする値と同様) にセットしてフォーム *body*を評価する。終了時には元にデフォルトのファイルノパーミッションをリストアして、*body*の最後のフォームの値をリターンする。

これはたとえばプライベートファイルの作成に有用である。

`default-file-modes` [Function]

この関数はデフォルトのファイルのパーミッションを整数でリターンする。

`read-file-modes` **&optional** *prompt base-file* [Function]

この関数はミニバッファーからファイルモードのビットのセットを読み取る。1 つ目のオプション引数 *prompt*は非デフォルトのプロンプトを指定する。2 つ目のオプション引数 *base-file*はユーザーが既存ファイルのパーミッションに相対的なモードビット指定をタイプした場合に、この関数がリターンするモードビッの元となる権限をもつファイルの名前を指定する。

ユーザー入力が 8 進数で表される場合には、この関数はその数字をリターンする。それが`"u=rwx"`のようなモードビットの完全なシンボル指定なら、この関数は `file-modes-symbolic-to-number`を使用して、それを等価な数字に変換して結果をリターンする。`"o+g"`のように相対的な指定なら、その指定の元となるパーミッションは *base-file*のモードビットから取得される。*base-file*が省略または `nil`なら、この関数は元となるモードビットとして `0`を使用する。完全指定と相対指定は`"u+r,g+rx,o+r,g-w"`のように組み合わせることができる。ファイルモード指定の説明は Section “File permissions” in *The* GNU `Coreutils` *Manual* を参照のこと。

`file-modes-symbolic-to-number` *modes* **&optional** *base-modes* [Function]
この関数は *modes*内のシンボルによるファイルモード指定を等価な整数に変換する。シンボル指定が既存ファイルにもとづく場合には、オプション引数 *base-modes*からそのファイルのモードビットが取得される。その引数が省略または `nil`なら、0(すべてのアクセスが許可されない) がデフォルトになる。

`set-file-times` *filename* **&optional** *time* [Function]
この関数は *filename*のアクセス時刻と変更時刻を *time*にセットする。時刻が正しくセットされれば `t`、それ以外は `nil`がリターン値となる。*time*のデフォルトはカレント時刻であり time 値でなければならない (Section 40.5 [Time of Day], page 1104 を参照)。

`set-file-extended-attributes` *filename attribute-alist* [Function]
この関数は `filename`にたいして Emacs が認識する拡張ファイル属性をセットする。2 つ目の引数 *attribute-alist*は `file-extended-attributes`がリターンする alist と同じ形式であること。属性のセットが成功したら `t`、それ以外は `nil`がリターン値となる。Section 25.6.5 [Extended Attributes], page 548 を参照のこと。

`set-file-selinux-context` *filename context* [Function]
この関数は *filename*にたいする SELinux セキュリティコンテキストに *context*をセットする。*context*引数は各要素が文字列であるような `(user role type range)`というリストであること。Section 25.6.5 [Extended Attributes], page 548 を参照のこと。

この関数は *filename*の SELinux コンテキストのセットに成功したら `t`をリターンする。コンテキストがセットされなかった場合 (SELinux が無効、または Emacs が SELinux サポートなしでコンパイルされた場合等) には `nil`をリターンする。

`set-file-acl` *filename acl* [Function]
この関数は *filename*にたいする ACL に *acl*をセットする。*acl*引数は関数 `file-acl`がリターンするのと同じ形式であること。Section 25.6.5 [Extended Attributes], page 548 を参照のこと。

この関数は *filename*の ACL のセットに成功したら `t`、それ以外は `nil`をリターンする。

25.8 ファイルと二次媒体

Emacs がファイルを変更した後、その後に生じた電源喪失と媒体エラーにより行った変更が保存されないことには 2 つの理由があります。1 つ目は一方または他方のファイルが後で変更されるまで、オペレーティングシステムは書き込まれたデータを 2 次ストレージのどこかに格納済みのデータにエイリアスすることです。媒体エラーにより 2 次ストレージ上のコピーだけが失われると、ファイルは両方とも失われるでしょう。2 つ目はオペレーティングシステムがデータを 2 次ストレージに即座に書き込まないかもしれないので、電源喪失時にはデータが失われます。

これらはいずれも適切なファイルシステムの設定により回避され得る類のエラーですが、そのようなシステムは通常はより高価であるか非効率的です。より典型的なシステムでは媒体エラーに備えて異なるデバイスにファイルをコピーでき、電源喪失にたいしては変数 `write-region-inhibit-fsync`に `nil`をセットして `write-region`を使用できます。Section 25.4 [Writing to Files], page 538 を参照してください。

25.9 ファイルの名前

ファイルは一般的に名前で参照され、これは Emacs でも他と同様です。Emacs ではファイル名は文字列で表現されます。ファイルを操作する関数はすべてファイル名引数に文字列を期待します。

ファイル自体の操作に加えて、Emacs Lisp プログラムでファイル名を処理する必要 (ファイル名の一部を取得して関連するファイル名構築にその一部を使用する等) がしばしばあります。このセクションではファイル名を扱う方法を説明します。

このセクションの関数は実際にファイルにアクセスする訳ではないので、既存のファイルやディレクトリーを参照しないファイル名を処理できます。

MS-DOS や MS-Windows では、(実際にファイルを操作する関数と同様) これらの関数は MS-DOS と MS-Windows のファイル名構文を受け入れます。この構文は POSIX 構文のようにバックスラッシュでコンポーネントを区切りますが、これらの関数は常に POSIX 構文をリターンします。これにより POSIX 構文でファイル名を指定する Lisp プログラムが変更なしですべてのシステムで正しく機能することが可能になります [1]。

25.9.1 ファイル名の構成要素

オペレーティングシステムはファイルをディレクトリーにグループ化します。あるファイルを指定するためには、ディレクトリーとそのディレクトリー内でのファイルの名前を指定しなければなりません。それゆえ Emacs はファイル名をディレクトリー名パートと非ディレクトリー (またはディレクトリー内ファイル名) パートという、2 つの主要パートから判断します。どちらのパートも空の場合があり得ます。これら 2 つのパートを結合することによって元のファイル名が再構築されます。

ほとんどのシステムでは最後のスラッシュ(MS-DOS と MS-Windows ではバックスラッシュも許される) までのすべてがディレクトリーパートです。残りが非ディレクトリーパートです。

ある目的のために、非ディレクトリーパートはさらに正式名称 (the name proper) とバージョン番号に細分されます。ほとんどのシステムでは、名前にバージョン番号をもつのはバックアップファイルだけです。

`file-name-directory` *filename* [Function]

この関数は *filename*のディレクトリーパートをディレクトリー名 (Section 25.9.3 [Directory Names], page 558 を参照) としてリターンする。*filename*がディレクトリーパートを含まなければ `nil`をリターンする。

GNU や他の POSIX 準拠 ix システムでは、この関数がリターンする文字列は常にスラッシュで終わる。MS-DOS ではコロンで終わることもあり得る。

```
(file-name-directory "lewis/foo")  ; GNU の例
     ⇒ "lewis/"
(file-name-directory "foo")        ; GNU の例
     ⇒ nil
```

`file-name-nondirectory` *filename* [Function]

この関数は *filename*の非ディレクトリーパートをリターンする。

```
(file-name-nondirectory "lewis/foo")
     ⇒ "foo"
```

[1] MS-Windows バージョンの Emacs は Cygwin 環境用にコンパイルされており、2 つのファイル名構文の変換に `cygwin-convert-file-name-to-windows`と `cygwin-convert-file-name-from-windows`を使用できます。

```
(file-name-nondirectory "foo")
     ⇒ "foo"
(file-name-nondirectory "lewis/")
     ⇒ ""
```

`file-name-sans-versions` *filename* **&optional** *keep-backup-version* [Function]

この関数は、任意のファイルバージョン番号、バックアップバージョン番号、末尾のチルダを取り除いた *filename*をリターンする。

*keep-backup-version*が非`nil`なら、ファイルシステムなどが認識するような真のファイルバージョン番号は破棄されるが、バックアップバージョン番号は保持される。

```
(file-name-sans-versions "~rms/foo.~1~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo")
     ⇒ "~rms/foo"
```

`file-name-extension` *filename* **&optional** *period* [Function]

この関数は *filename*から、もしあればすべてのバージョン番号とバックアップ番号を取り除いた後の、終端の拡張子 (extension) をリターンする。ファイル名の拡張子とは最後の名前コンポーネント (からすべてのバージョン番号とバックアップ番号を取り去った後) の最後の‘.’に後続するパートのこと。

この関数は`foo`のような拡張子のないファイル名にたいしては`nil`、`foo.`のような null 拡張子にたいしては`""`をリターンする。ファイル名の最終コンポーネントが‘.’で始まる場合には、その‘.’は拡張子の開始とはみなされない。したがって`.emacs`の拡張子は‘`.emacs`’ではなく`nil`。

*period*が非`nil`なら、拡張子を区切るピリオドもリターン値に含まれる。その場合には、もし *filename*が拡張子をもたなければリターン値は`""`。

`file-name-sans-extension` *filename* [Function]

この関数は、もしあれば *filename*から拡張子を除いてリターンする。もしバージョン番号やバックアップ番号があるなら、ファイルが拡張子をもつ場合のみそれを削除する。たとえば、

```
(file-name-sans-extension "foo.lose.c")
     ⇒ "foo.lose"
(file-name-sans-extension "big.hack/foo")
     ⇒ "big.hack/foo"
(file-name-sans-extension "/my/home/.emacs")
     ⇒ "/my/home/.emacs"
(file-name-sans-extension "/my/home/.emacs.el")
     ⇒ "/my/home/.emacs"
(file-name-sans-extension "~/foo.el.~3~")
     ⇒ "~/foo"
(file-name-sans-extension "~/foo.~3~")
     ⇒ "~/foo.~3~"
```

最後の 2 つの例の‘`.~3~`’は拡張子ではなくバックアップ番号であることに注意。

`file-name-base` *filename* [Function]
これは `file-name-sans-extension`と `file-name-nondirectory`を組み合わせた関数。たとえば、

```
(file-name-base "/my/home/foo.c")
    ⇒ "foo"
```

25.9.2 絶対ファイル名と相対ファイル名

ファイルシステム内のすべてのディレクトリーはルートディレクトリーから開始されるツリーを形成します。このツリーのルートから開始されるすべてのディレクトリー名によりファイル名を指定でき、それを絶対 *(absolute)* ファイル名と呼びます。デフォルトディレクトリーからの相対的なツリー中の位置でファイルを指定することもでき、それらは相対 *(relative)* ファイル名と呼ばれます。GNU や他の POSIX 準拠システムでは '`~`'で開始されるすべてのファイル名は '`/`'で始まる絶対ファイル名 ([abbreviate-file-name], page 559 を参照) に展開されますが、相対ファイル名は展開されません。。MS-DOS と MS-Windows では絶対ファイル名はスラッシュ、バックスラッシュ、またはドライブ指定 '`x:/`'で始まります。ここで *x*はドライブ文字 *(drive letter)* です。

`file-name-absolute-p` *filename* [Function]
この関数はファイル *filename*が絶対ファイル名なら `t`、それ以外は `nil`をリターンする。ファイル名の最初のコンポーネントが '`~`'か '`~`*user*' (*user*は有効なログイン名) なら絶対ファイル名とみなす。以下の例では '`rms`'という名前のユーザーは存在するが、'`nosuchuser`'という名前のユーザーは存在しないものとする。

```
(file-name-absolute-p "~rms/foo")
     ⇒ t
(file-name-absolute-p "~nosuchuser/foo")
     ⇒ nil
(file-name-absolute-p "rms/foo")
     ⇒ nil
(file-name-absolute-p "/user/rms/foo")
     ⇒ t
```

相対ファイル名が与えられた場合には先頭に '`~`'があれば展開して、`expand-file-name`を使用して絶対ファイル名に変換できます (Section 25.9.4 [File Name Expansion], page 559 を参照)。この関数は絶対ファイル名を相対ファイル名に変換します:

`file-relative-name` *filename* **&optional** *directory* [Function]
この関数は *directory*(絶対ディレクトリー名かディレクトリーファイル名) から相対的なファイルと仮定して、*filename*と等価な相対ファイル名のリターンを試みる。*directory*が省略か `nil`なら、カレントバッファーのデフォルトディレクトリーがデフォルト。

絶対ファイル名がデバイス名で始まるオペレーティングシステムがいくつか存在する。そのようなシステムでは、2 つの異なるデバイス名から開始される *filename*は、*directory*にもといた等価な相対ファイル名をもたない。この場合には、`file-relative-name`は絶対形式で *filename* をリターンする。

```
(file-relative-name "/foo/bar" "/foo/")
     ⇒ "bar"
(file-relative-name "/foo/bar" "/hack/")
     ⇒ "../foo/bar"
```

25.9.3 ディレクトリーの名前

ディレクトリー名 *(directory name)* とは、ある文字列が何らかのファイルを命名する場合にはディレクトリーを命名する文字列のことです。ディレクトリーは実際にはファイルの一種なので、ファイル名 (ディレクトリーファイル名と呼ばれる) をもち、これはディレクトリー名と関係はあるものの通常は等価ではありません (これは POSIX の通常の用語と完全に同一ではない)。同じ実体にたいするこれら 2 つの異なる名前は構文的な変換により関連付けられます。GNU や他の POSIX システムではことは単純です。ディレクトリー名の最後が '/'でなければ、ディレクトリーファイル名に '/'を追加してディレクトリー名を取得できます。MS-DOS ではこの関連付けはより複雑です。

ディレクトリー名とディレクトリーファイル名の違いは些細ですが重要です。Emacs の変数や関数の引数を記述する際には、それがディレクトリー名であるとしており、ディレクトリーファイル名は許容されません。`file-name-directory`が文字列をリターンするときには常にディレクトリー名をリターンします。

以下の 2 つの関数は、ディレクトリー名とディレクトリーファイル名の間で変換を行います。これらの関数は '`$HOME`'のような環境変数や '`~`'、'`.`'、'`..`'などの構文にたいして、特別なことは何も行いません。

`file-name-as-directory` *filename* [Function]
この関数はオペレーティングシステムがディレクトリーの名前 (ディレクトリー名) と解釈する形式で *filename*を表す文字列をリターンする。これはほとんどのシステムでは、(もし終端にそれがなければ) これは文字列にスラッシュを追加することを意味する。

```
(file-name-as-directory "~rms/lewis")
     ⇒ "~rms/lewis/"
```

`directory-name-p` *filename* [Function]
この関数は *filename*の終端がディレクトリー区切り文字なら非 `nil`をリターンする。これは GNU や他の POSIX 準拠システムではスラッシュ'/'、MS-Windows と MS-DOS ではスラッシュとバックスラッシュ'\'がディレクトリー区切りとして認識される。

`directory-file-name` *dirname* [Function]
この関数はオペレーティングシステムがファイルの名前と解釈する形式 (ディレクトリーファイル名) で *dirname*を表す文字列をリターンする。ほとんどのシステムではこれは文字列すべてがディレクトリー区切り文字で構成されている場合を除き、文字列から最後のディレクトリー区切り文字を削除することを意味する。

```
(directory-file-name "~lewis/")
     ⇒ "~lewis"
```

ディレクトリー名にたいしては `concat`を使用して相対ファイルと組み合わせることができます:

```
(concat dirname relfile)
```

これを行う前にファイル名が相対的であることを確認してください。絶対ファイル名を使用すると構文的に不正な結果となったり、間違ったファイルを参照する可能性があります。

ディレクトリーファイル名の作成にこのような組み合わせを使用しなければ、最初に `file-name-as-directory`を使用してそれをディレクトリー名に変換しなければなりません:

```
(concat (file-name-as-directory dirfile) relfile)
```

以下のように手動でスラッシュの結合を試みてはなりません

```
;;; 間違い!
(concat dirfile "/" relfile)
```

なぜならこれには可搬性がないからです。常に `file-name-as-directory`を使用してください。

上述の問題の回避や *dirname*の値が `nil`かもしれない場合 (たとえば `load-path`の要素を使用する際) には以下を使用します:

```
(expand-file-name relfile dirname)
```

ただし `expand-file-name`は *relfile*の先頭の '`~`'を展開しますが、これは望んだ動作ではないかもしれません。Section 25.9.4 [File Name Expansion], page 559 を参照してください。

ディレクトリー名をディレクトリーの省略名に変換するには以下の関数を使用します:

`abbreviate-file-name` *filename* [Function]

この関数は *filename*の省略された形式をリターンする。これは `directory-abbrev-alist` (Section "File Aliases" in *The GNU Emacs Manual* を参照) で指定される省略名を適用して、引数で与えられるファイル名ががホームディレクトリーかそのサブディレクトリーにあれば、ユーザーのホームディレクトリーを '`~`'に置換する。ホームディレクトリーがルートディレクトリーなの場合には、多くのシステムでは結果が短縮されないので '`~`'で置き換えない。

これは名前の一部であるような省略形さえも認識するので、ディレクトリー名とファイル名にも使用できる。

25.9.4 ファイル名を展開する関数

ファイル名の展開 *(expanding)* と、相対ファイル名を絶対ファイル名に変換することを意味します。これはデフォルトディレクトリーから相対的に行われるため、展開されるファイル名と同様にデフォルトディレクトリーも指定しなければなりません。これは`~/`のような省略形の展開、および`./`や *name*`/../` のような冗長さの排除も行います。

`expand-file-name` *filename* **&optional** *directory* [Function]

この関数は *filename*を絶対ファイル名に変換する。*directory*が与えられた場合には *filename* が相対的であり先頭が '`~`'でなければ、それが開始点となるデフォルトディレクトリーになる (*directory*の値はそれ自体が絶対ディレクトリー名かディレクトリーファイル名であるべきで、それは '`~`'で始まるかもしれない)。それ以外ではカレントバッファーの `default-directory` の値が使用される。たとえば:

```
(expand-file-name "foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
(expand-file-name "../foo")
     ⇒ "/xcssun/users/rms/foo"
(expand-file-name "foo" "/usr/spool/")
     ⇒ "/usr/spool/foo"
```

*filename*の最初のスラッシュの前が '`~`'ならユーザーのホームディレクトリー (通常は環境変数 `HOME`の値で指定される) に展開される (Section "General Variables" in *The GNU Emacs Manual* を参照)。最初のスラッシュの前が '`~`*user*'で *user*が有効なログイン名なら *user*のホームディレクトリーに展開される。リテラル '`~`'で始まるかもしれない相対的な *filename*にたいして展開を望まない場合には `(expand-file-name filename directory)`のかわりに `(concat (file-name-as-directory directory) filename)`を使用できる。

'`.`'や '`..`'を含むファイル名は正規化形式に簡略化される:

```
(expand-file-name "bar/../foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
```

出力に‘..’部分が残り得る場合もある:

```
(expand-file-name "../home" "/")
     ⇒ "/../home"
```

これはルートディレクトリー/の上位のスーパールート (superroot) という概念をもつファイルシステムのためのものである。その他のファイルシステムでは/../は/とまったく同じに解釈される。

.や空文字列を展開するとデフォルトディレクトリーがリターンされる:

```
(expand-file-name "." "/usr/spool/")
     ⇒ "/usr/spool"
(expand-file-name "" "/usr/spool/")
     ⇒ "/usr/spool"
```

expand-file-nameは環境変数を展開しないことに注意。それを行うのは substitute-in-file-nameのみ。

```
(expand-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/lewis/$HOME/foo"
```

expand-file-nameはあらゆる階層においてシンボリックリンクをフォローしないことにも注意。これは‘..’の扱いが file-truenameと expand-file-nameで異なることに起因する。‘/tmp/bar’がディレクトリー‘/tmp/foo/bar’にたいするシンボリックリンクであると仮定すると:

```
(file-truename "/tmp/bar/../myfile")
     ⇒ "/tmp/foo/myfile"
(expand-file-name "/tmp/bar/../myfile")
     ⇒ "/tmp/myfile"
```

直接間接を問わず事前に expand-file-nameを呼び出さずに‘..’に先行するシンボリックリンクをフォローする必要があるかもしれない場合には、それを呼び出さずに確実に file-truenameを呼び出すこと。Section 25.6.3 [Truenames], page 544 を参照されたい。

`default-directory` [Variable]

このバッファーローカル変数の値はカレントバッファーにたいするデフォルトディレクトリー。これは絶対ディレクトリー名であること。これは‘~’で始まるかもしれない。この変数はすべてのバッファーにおいてバッファーローカル。

2 つ目の引数が nilなら、expand-file-nameはデフォルトディレクトリーを使用する。

値は常にスラッシュで終わる文字列。

```
default-directory
     ⇒ "/user/lewis/manual/"
```

`substitute-in-file-name` *filename* [Function]

この関数は *filename*内で参照される環境変数を環境変数の値に置き換える。標準的な Unix シェル構文にしたがい‘$’は環境変数値を置き換るプレフィックスである。入力に‘$$’が含まれる場合には、それらは‘$’に置き換えられる。これによりユーザーが‘$’をクォートする手段が与えられる。

環境変数名は‘$’の後に続く一連の英数字 (アンダースコアを含む) である。‘$’の後続文字が‘{’なら対応する‘}’までのすべてが変数名である。

substitute-in-file-nameにより生成された出力で substitute-in-file-nameを呼び出すと不正な結果となる傾向がある。たとえば単一の‘$’をクォートするために‘$$’を使用して

も正しく機能せずに環境変数値の中の‘$’は再帰的な置換を導くだろう。したがってこの関数を呼び出して出力をこの関数に渡すプログラムは、その後の不正な結果を防ぐためにすべての‘$’文字を二重化する必要がある。

以下ではユーザーのホームディレクトリーを保持する環境変数 HOMEの値が‘/xcssun/users/rms’だと仮定する。

```
(substitute-in-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
```

置き換え後には、‘/’の直後に‘~’や別の‘/’が出現すると、この関数は‘/’の前にあるすべてを無視する。

```
(substitute-in-file-name "bar/~/foo")
     ⇒ "~/foo"
(substitute-in-file-name "/usr/local/$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
     ;; /usr/local/は破棄された
```

ファイル名の展開が望ましくない場合もあります。そのような場合には展開を抑制するためにファイル名をクォートしてファイル名をそのままリテラルとして処理することができます。ファイル名の前に‘/:’を前置することによりクォートが行われます。

`file-name-quote` *name* [Macro]

このマクロはファイル *name*にクォーテーションプレフィクス‘/:’を付加する。ローカルファイル *name*には *name*の前にプレフィクス‘/:’を付加する。*name*がリモートファイル名なら *name*のローカル部分がクォートされる (Section 25.12 [Magic File Names], page 569 を参照)。*name*がクォート済みのファイル名なら、*name*は変更せずにリターンする。

```
(substitute-in-file-name (file-name-quote "bar/~/foo"))
     ⇒ "/:bar/~/foo"


(substitute-in-file-name (file-name-quote "/ssh:host:bar/~/foo"))
     ⇒ "/ssh:host:/:bar/~/foo"
```

マジックファイル名によるファイル名ハンドラーを抑制するためにこのマクロは使用できない (Section 25.12 [Magic File Names], page 569 を参照)。

`file-name-unquote` *name* [Macro]

このマクロはファイル *name*にクォーテーションプレフィクス‘/:’があれば削除する。*name*がリモートファイル名なら *name*のローカル部分を非クォート化する。

`file-name-quoted-p` *name* [Macro]

このマクロは *name*がプレフィクス‘/:’でクォートされていれば非 nilをリターンする。*name*がリモートファイル名なら *name*のローカル部分をチェックする。

25.9.5 一意なファイル名の生成

一時ファイルに書き込む必要があるプログラムがいくつかあります。以下は、そのようなファイルを構築する便利な方法です:

```
(make-temp-file name-of-application)
```

make-temp-fileの役目は、2 人の異なるユーザーやジョブが完全に一致する名前のファイルの使用を防ぐことです。

`make-temp-file` *prefix* **&optional** *dir-flag suffix text* [Function]

この関数は一時ファイルを作成して、その名前をリターンする。Emacs は Emacs の各ジョブごとに異なるランダムないくつかの文字を *prefix*に追加することにより一時ファイルの名前を作成する。結果として文字列として *text*が与えられた場合にはそれを含むファイル、それ以外は空ファイルが新たに作成されることが保障される。MS-DOS では、8+3 のファイル名制限に適合するように、文字列 *string*が切り詰められる可能性がある。*prefix*が相対ファイル名なら `temporary-file-directory`にたいして展開される。

```
(make-temp-file "foo")
     ⇒ "/tmp/foo232J6v"
```

`make-temp-file`がリターンした際には、一時ファイルは空で作成される。この時点でそのファイルに意図するコンテンツを書き込むこと。

*dir-flag*が `nil`なら、`make-temp-file`は空のファイルのかわりに空のディレクトリーを作成する。これはディレクトリー名ではなく、ディレクトリーのファイル名をリターンする。Section 25.9.3 [Directory Names], page 558 を参照のこと。

*suffix*が非 `nil`なら、`make-temp-file`はそれをファイル名の最後に追加する。

*text*が文字列なら `make-temp-file`はそれをファイルに挿入する。

同じ Emacs 内で実行される異なるライブラリー間での競合を防ぐために、`make-temp-file`を使用する各 Lisp プログラムがプログラム自身の *prefix*を使用すること。*prefix*の最後に追加される数字は、異なる Emacs ジョブ内で実行される同じアプリケーションを区別する。追加される文字により、同一の Emacs ジョブ内でも多数の名前を区別することが可能になる。

一時ファイル用のデフォルトディレクトリーは変数 `temporary-file-directory`により制御されます。この変数によりすべての一時ファイルにたいして、ユーザーがディレクトリーを指定する一貫した方法が与えられます。`small-temporary-file-directory`が非 `nil`なら、かわりにそれを使うプログラムもいくつかあります。これを使う場合には、`make-temp-file`を呼び出す前に正しいディレクトリーにたいしてプレフィックスを展開するべきです。

`temporary-file-directory` [User Option]

この変数は一時ファイル作成用のディレクトリー名を指定する。値はディレクトリー名であるべきだが、もし値がディレクトリーのファイル名 (Section 25.9.3 [Directory Names], page 558 を参照) ならば、Lisp プログラムがかわりに対処すればよい。`expand-file-name`の 2 つ目の引数としてその値を使用するのは、それを達成するよい方法である。

デフォルト値はオペレーティングシステムにたいして適切な方法により決定される。これは環境変数 `TMPDIR`、`TMP`、`TEMP`にもとづく値で、これらの変数が定義されていなければシステム依存の名前にフォールバックする。

一時ファイルの作成に `make-temp-file`を使用しない場合でも、一時ファイルを置くディレクトリーを判断するために依然としてこの変数を使用するべきである。しかし一時ファイルが小さくなることを求める場合には、`small-temporary-file-directory`が非 `nil`ならそれを使用すること。

`small-temporary-file-directory` [User Option]

この変数はサイズが小さいと予想される特定の一時ファイル作成用のディレクトリー名を指定する。

小さくなるかもしれない一時ファイルに書き込みたいなら、以下のようにディレクトリーを計算すること:

```
(make-temp-file
```

```
(expand-file-name prefix
                  (or small-temporary-file-directory
                      temporary-file-directory)))
```

`make-temp-name` *base-name* [Function]
この関数は一意なファイル名として使用できる文字列を生成する。この名前は *base-name*で始まり、それに各Emacsジョブごとに異なる複数のランダムな文字を追加したものである。これは `make-temp-file`と似ているが、(i) 名前だけを作成してファイルは作成せず、(ii)*base-name* はマジックファイル名ではない絶対ファイル名であること、(iii) リターンされるファイル名がマジックファイル名なら既存のファイルかもしれない、という点が異なる (MS-DOS システムでは 8+3 ファイル名制限に適合するように *base-name*が切り詰められる)。

警告: この関数を使用するべきではない。かわりに `make-temp-file`を使用すること! この関数は競合状態の影響を受けやすい。`make-temp-name`呼び出しと一時ファイル作成のタイムラグはセキュリティーホールとなり得る。

リモートホストやマウントされたディレクトリーで一時ファイルの作成を要する場合があります。以下の 2 つの関数はそれをサポートします。

`make-nearby-temp-file` *prefix* **&optional** *dir-flag suffix* [Function]
この関数は `make-temp-file`と同様だができ得るかぎり `default-directory`に近接した一時ファイルを作成する点が異なる。*prefix*が相対ファイル名で `default-directory`がリモートファイル名かマウントされたファイルシステムに配置されていれば、一時ファイルは関数 `temporary-file-directory`がリターンするディレクトリー内に作成される。それ以外なら関数 `make-temp-file`を使用する。*prefix*、*dir-flag*、*suffix*の意味は `make-temp-file`の場合と同様。

```
(let ((default-directory "/ssh:remotehost:"))
  (make-nearby-temp-file "foo"))
     ⇒ "/ssh:remotehost:/tmp/foo232J6v"
```

`temporary-file-directory` [Function]
`make-nearby-temp-file`を通じて一時ファイルを書き込むディレクトリー。`default-directory`がリモートの場合にはリモートホスト上の一時ファイル用のディレクトリー。そのようなディレクトリーが存在しない、または `default-directory`がマウントされるファイルシステム上に配置される場合 (`mounted-file-systems`を参照) には、この関数は `default-directory`をリターンする。リモートでもマウントされたファイルシステムでもない `default-directory`では `temporary-file-directory`の値がリターンされる。

一時ファイルのファイル名のローカル部分を抽出するためには `file-local-name`を使用してください (Section 25.12 [Magic File Names], page 569 を参照)。

25.9.6 ファイル名の補完

このセクションではファイル名を補完するための低レベルサブルーチンについて説明します。より高レベルの関数については Section 20.6.5 [Reading File Names], page 359 を参照してください。

`file-name-all-completions` *partial-filename directory* [Function]
この関数はディレクトリー *directory*内で *partial-filename*で始まる名前のファイルにたいする、すべての補完可能なリストをリターンする。補完の順番はそのディレクトリー内でのファイル順序であり、これは予測不能であり何の情報ももたない。

引数 *partial-filename*は非ディレクトリーパートを含むファイル名でなければならず、スラッシュ(いくつかのシステムではバックスラッシュ) が含まれていてはならない。*directory*が絶対ディレクトリーでなければ、*directory*の前にカレントバッファーのデフォルトディレクトリーが追加される。

以下の例では`~rms/lewis`がカレントデフォルトディレクトリーで、名前が‘`f`’で始まる 5 つのファイル `foo`、`file~`、`file.c`、`file.c.~1~`、`file.c.~2~`がある:

```
(file-name-all-completions "f" "")
     ⇒ ("foo" "file~" "file.c.~2~"
                "file.c.~1~" "file.c")

(file-name-all-completions "fo" "")
     ⇒ ("foo")
```

`file-name-completion` *filename directory* **&optional** *predicate* [Function]

この関数はディレクトリー *directory*内でファイル名 *filename*を補完する。これはディレクトリー *directory*内で、*filename*で始まるすべてのファイル名にたいして、最長の共通プレフィックスをリターンする。*predicate*が非 `nil`なら展開された絶対ファイル名を単一の引数として呼び出して、*predicate*を満足しない補完候補を無視する。

マッチが 1 つだけ存在して、かつ *filename*が正確にそれにマッチする場合には、この関数は `t`をリターンする。関数はディレクトリー *directory*が *filename*で始まる名前のファイルを含まなければ `nil`をリターンする。

以下の例では`~rms/lewis`がカレントデフォルトディレクトリーで、名前が‘`f`’で始まる 5 つのファイル `foo`、`file~`、`file.c`、`file.c.~1~`、`file.c.~2~`がある:

```
(file-name-completion "fi" "")
     ⇒ "file"

(file-name-completion "file.c.~1" "")
     ⇒ "file.c.~1~"

(file-name-completion "file.c.~1~" "")
     ⇒ t

(file-name-completion "file.c.~3" "")
     ⇒ nil
```

`completion-ignored-extensions` [User Option]

`file-name-completion`はこのリスト内の任意の文字列で終わるファイル名を通常は無視する。すべての可能な補完がこれらのサフィックスのいずれか 1 つで終わるときはそれらを無視しない。この変数は `file-name-all-completions`に影響しない。

以下は典型的な値:

```
completion-ignored-extensions
     ⇒ (".o" ".elc" "~" ".dvi")
```

`completion-ignored-extensions`のある要素がスラッシュ‘`/`’で終わる場合には、それはディレクトリーを示す。スラッシュで終わらない要素がディレクトリーにマッチすることは決してない。したがって上記の値は `foo.elc`という名前のディレクトリーを除外しないだろう。

25.9.7 標準的なファイル名

Emacs Lisp プログラムが特定の用途のために標準的なファイル名を指定することが必要な場合があります。典型的にはカレントユーザーによって指定された設定データを保持する場合が該当します。そのようなファイルは、通常は `user-emacs-directory`で指定されるディレクトリーに配置されて、デフォルトでは通常は`~/.config/emacs/`か`~/.emacs.d/`です (Section "How Emacs Finds Your Init File" in *The GNU Emacs Manual* を参照)。たとえば abbrev(abbreviation: 省略形) の定義は、デフォルトでは`~/.config/emacs/abbrev_defs`か`~/.emacs.d/abbrev_defs`に格納されます。このようなファイル名を指定するためには、関数 `locate-user-emacs-file`を使用するのがもっとも簡単な方法です。

`locate-user-emacs-file` *base-name* **&optional** *old-name* [Function]

この関数は Emacs 特有の設定ファイルやデータファイルにたいする絶対ファイル名をリターンする。引数 `base-name`は、相対ファイル名であること。リターン値は `user-emacs-directory`で指定されるディレクトリー内の絶対ファイル名。そのディレクトリーが存在しなければ、この関数はディレクトリーを作成する。

オプション引数 *old-name*が非 `nil`なら、それはユーザーのホームディレクトリー内のファイル`~/`*old-name*を指定する。そのようなファイルが存在すれば、リターン値は *base-name*で指定されるファイルではなくそのファイルの絶対ファイル名となる。これは Emacs パッケージが後方互換を提供するために使用されることを意図した引数。たとえば `user-emacs-directory`導入前には、abbrev ファイルは`~/.abbrev_defs`に置かれていた。以下は `abbrev-file-name`の定義である:

```
(defcustom abbrev-file-name
  (locate-user-emacs-file "abbrev_defs" ".abbrev_defs")
  "Default name of file from which to read abbrevs."
  ...
  :type 'file)
```

ファイル名の標準化のための低レベル関数は `convert-standard-filename`で、これはサブルーチンとして `locate-user-emacs-file`により使用されます。

`convert-standard-filename` *filename* [Function]

この関数は *filename*にもとづいたカレントオペレーティングシステムの慣習に適合するファイル名をリターンする。

GNU や他の POSIX 準拠システムでは単に *filename*をリターンする。その他のオペレーティングシステムではシステム固有のファイル名規約にしたがうだろう。たとえば MS-DOS では、この関数は MS-DOS ファイル名制限にしたがうように先頭の '.'を '_'に変換したり、'.'の後続の文字を 3 文字に切り詰める等、さまざまな変更を行う。

この関数で GNU と Unix システムの慣習に適合する名前を指定して、それを `convert-standard-filename`に渡すのが推奨される使用方法である。

25.10 ディレクトリーのコンテンツ

ディレクトリーとはファイルの一種であり、さまざまな名前のファイルを含んでいます。ディレクトリーはファイルシステムの機能です。

Emacs はディレクトリー内のファイル名を Lisp のリストとして一覧したり、シェルコマンド `ls`を使用してバッファー内にファイル名を表示することができます。後者の場合には、Emacs はオプションで各ファイルに関する情報も表示でき、それは `ls`コマンドに渡すオプションに依存します。

`directory-files` *directory* **&optional** *full-name match-regexp nosort* [Function]

この関数はディレクトリー *directory*内のファイルの名前のリストをリターンする。デフォルトではリストはアルファベット順。

この関数は *full-name*が非 `nil`ならファイルの絶対ファイル名、それ以外なら指定されたディレクトリーにたいする相対ファイル名をリターンする。

*match-regexp*が非 `nil`なら、この関数はその正規表現にたいするマッチを含むファイル名だけをリターンして、それ以外のファイル名はリストから除外される。case(大文字小文字) を区別するファイルシステムでは、case を区別する正規表現マッチングが行われる。

*nosort*が非 `nil`なら `directory-files`はリストをソートしないので、取得するファイル名に特定の順序はない。最大限の可能なスピードを得る必要がありファイル処理順を気にしなければこれを使用する。ユーザーから処理順が可視なら、名前をソートすれば多分ユーザーはより幸せになるだろう。

```
(directory-files "~lewis")
     ⇒ ("#foo#" "#foo.el#" "." ".."
         "dired-mods.el" "files-ja.texi"
         "files-ja.texi.~1~")
```

*directory*が読み取り可能なディレクトリー名でなければエラーがシグナルされる。

`directory-files-recursively` *directory regexp* **&optional** *include-directories predicate follow-symlinks* [Function]

*regexp*にマッチする名前をもつ *directory*配下のすべてのファイルをリターンする。この関数はベースネーム (basename: 先行するディレクトリー部分を除外したファイル名) が *regexp*にマッチするファイルを、*directory*とそのサブディレクトリーを再帰的に検索して、マッチしたファイルの絶対ファイル名 (Section 25.9.2 [Relative File Names], page 557 を参照) のリストをリターンする。ファイル名は深さ優先順でリターンされ、それは親ディレクトリーの前に任意のサブディレクトリー内のファイルが配置されることを意味する。加えて各ディレクトリー内で見つかったファイルはベースネームにもとづいてソートされる。デフォルトでは *regexp*にマッチする名前のディレクトリーはリストから省略されるが、オプション引数 *include-directories* が非 `nil`ならそれらも含まれる。

デフォルトではすべてのサブディレクトリーが含まれる。*predicate*が `t`ならサブディレクトリーを含める際のエラー (たとえばそのユーザーでは読み取り不可の場合) は無視される。`nil` や `t`以外なら、1 つのパラメーター (サブディレクトリー名) を受け取り、そのディレクトリーを含める場合には非 `nil`をリターンする関数であること。

デフォルトではサブディレクトリーへのシンボリックリンクはフォローしないが、*follow-symlinks*が非 `nil`ならフォローする。

`locate-dominating-file` *file name* [Function]

*file*から開始してディレクトリーツリー階層を上方に *name* (文字列) というディレクトリーを検索して、最初に見つけたディレクトリーをリターンする。*file*がファイルならファイルのディレクトリーが検索の開始位置、それ以外なら *file*は検索を開始するディレクトリーであること。この関数は開始ディレクトリーを調べて、それからその親ディレクトリー、更にその親ディレクトリー、... のように *name*というディレクトリーを見つけるか、あるいは *name*が見つけることなくファイルシステムのロートディレクトリーに到達するまで検索を行う。後者の場合には `nil`をリターンする。

引数 `name`は述語関数でもよい。この述語は関数により検査される、*file* (*file*がディレクトリーでない場合でも) から開始されるすべてのディレクトリーにたいして呼び出される。この述語は

1 つの引数 (ファイルかディレクトリー) で呼び出されて、それが検索しているディレクトリーなら非 nilをリターンすること。

directory-files-and-attributes *directory* **&optional** *full-name match-regexp nosort id-format* [Function]

これはどのファイルを報告するか、およびファイル名を報告する方法においてdirectory-filesと似ている。しかしこの関数はファイル名のリストをリターンするかわりに、各ファイルごとにリスト (*filename* . *attributes*)をリターンする。ここで *attributes*は、そのファイルにたいして file-attributesがリターンする値。オプション引数 *id-format*は、file-attributesの対応する引数と同じ意味をもつ ([Definition of file-attributes], page 546 を参照)。

file-expand-wildcards *pattern* **&optional** *full* [Function]

この関数はワイルドカードパターン *pattern*を展開して、それにマッチするファイル名のリストをリターンする。

絶対ファイル名として *pattern*が記述されると値も絶対ファイル名になる。

*pattern*が相対ファイル名で記述されていれば、それはカレントデフォルトディレクトリーにたいして相対的に解釈される。通常はリターンされるファイル名もカレントデフォルトディレクトリーにたいする相対ファイル名になる。しかし *full*が非 nilなら絶対ファイル名がリターンされる。

insert-directory *file switches* **&optional** *wildcard full-directory-p* [Function]

この関数は lsの *switches*に対応するフォーマットで、(カレントバッファー内に) ディレクトリー *file*のディレクトリーリストを挿入する。これは挿入したテキストの後にポイントを残す。*switches*にはオプション文字列、または個別のオプションを表す文字列リストを指定できる。

引数 *file*にはディレクトリー、またはワイルドカード文字を含むファイル名を指定できる。*wildcard*が非 nilなら *file*はワイルドカードを伴うファイル指定として扱われることを意味する。

*full-directory-p*が非 nilなら、ディレクトリーリストにたいしてディレクトリーの完全なコンテンツ表示を要求することを意味する。*file*がディレクトリーでスイッチに '-d'が含まれないときには、tを指定すること (lsへのオプション '-d'は、ディレクトリーのコンテンツではなくファイルとしてディレクトリーを表示するよう指定する)。

ほとんどのシステムでは、この関数は変数 insert-directory-programの名前のディレクトリーリスト用プログラムを実行することにより機能する。*wildcard*が非 nilなら、ワイルドカード展開するために shell-file-nameで指定されるシェルの実行も行う。

MS-DOS と MS-Windows システムは標準的な Unix プログラム lsを欠くので、この関数は Lisp コードで lsをエミュレートする。

技術的な詳細としては *switches*にロングオプション '--dired'が含まれる際に、insert-directoryは dired のためにこれを特別に扱う。しかし他のオプションと同様、通常は等価なショートオプション '-D'が単に insert-directory-programに渡されるだけである。

insert-directory-program [Variable]

この変数の値は関数 insert-directory用にディレクトリーリストを生成するプログラムである。この値は Lisp コードでこのリストを生成するシステムでは無視される。

25.11 ディレクトリーの作成・コピー・削除

Emacs Lisp のファイル操作関数のほとんどは、ディレクトリーであるようなファイルに使用されたときはエラーとなります。たとえば `delete-file`でディレクトリーの削除はできません。以下のスペシャル関数はディレクトリーの作成と削除を行うために存在します。

`make-directory` *dirname* **&optional** *parents* [Command]
このコマンドは *dirname*という名前のディレクトリーを作成する。*parents*が非 `nil`の場合 (インタラクティブな呼び出しでは常に非 `nil`) には、その親ディレクトリーがまだ存在しなければ最初にそれを作成することを意味する。`mkdir`はこのコマンドにたいするエイリアス。

`make-empty-file` *filename* **&optional** *parents* [Command]
このコマンドは *filename*という名前の空のファイルを作成する。このコマンドは `make-directory`と同様に、*parents*が非 *filename*がすでに存在する場合には、このコマンドはエラーをシグナルする。

`copy-directory` *dirname newname* **&optional** *keep-time parents* [Command]
copy-contents
このコマンドは *dirname*という名前のディレクトリーを *newname*にコピーする。*newname*がディレクトリー名なら *dirname*はそれのサブディレクトリーにコピーされる。Section 25.9.3 [Directory Names], page 558 を参照のこと。

これは常にコピーされるファイルのファイルモードを、対応する元のファイルモードと一致させる。

3 つ目の引数 *keep-time*が非 `nil`なら、それはコピーされるファイルの修正時刻を保持することを意味する。プレフィックス引数を与えると、*keep-time*は非 `nil`になる。

4 つ目の引数 *parents*は、親ディレクトリーが存在しない場合に作成するかどうかを指定する。インタラクティブな場合には、これはデフォルトで発生する。

5 つ目の引数 *copy-contents*が非 `nil`の場合には、それは *newname*がディレクトリー名ならば、そのサブディレクトリーとして *dirname*をコピーするかわりに *dirname*のコンテンツを *newname*にコピーする。

`delete-directory` *dirname* **&optional** *recursive trash* [Command]
このコマンドは *dirname*という名前のディレクトリーを削除する。関数 `delete-file`はディレクトリーであるようなファイルにたいしては機能しない。それらにたいしては `delete-directory`を使用しなければならない。*recursive*が `nil`でディレクトリー内にファイルが存在する場合には、`delete-directory`はエラーをシグナルする。*recursive*が非 `nil`なら、`delete-directory`の処理前にそのディレクトリーやディレクトリーのファイルを他のプロセスが削除したという稀な状況を除いてエラーは発生しない。

`delete-directory`は親ディレクトリーの階層のシンボリックリンクだけをフォローする。

オプション引数 *trash*が非 `nil`、かつ変数 `delete-by-moving-to-trash`が非 `nil`なら、このコマンドはファイルを削除するかわりにシステムの Trash(ゴミ箱) にファイルを移動する。Section “Miscellaneous File Operations” in *The GNU Emacs Manual* を参照のこと。インタラクティブに呼び出された際には、プレフィックス引数がなければ *trash*は `t`、それ以外は `nil`。

25.12 特定のファイル名の “Magic” の作成

特定のファイル名にたいして特別な処理を実装できます。これはそれらの名前にたいする *magic* 化と呼ばれます。この機能は主にリモートファイルにたいするアクセスの実装用に使用されます (Section “Remote Files” in *The GNU Emacs Manual* を参照)。

magic ファイル名を定義するには、名前クラスを定義するための正規表現とそれにマッチするファイル名用の Emacs ファイル操作プリミティブすべてを実装するハンドラーを定義しなければなりません。

変数`file-name-handler-alist`は各ハンドラーに適用するときを決定する正規表現とともにハンドラーのリストを保持します。各要素は以下の形式をもちます:

```
(regexp . handler)
```

ファイルアクセスとファイル名変換にたいするすべての Emacs プリミティブは、`file-name-handler-alist`にたいして与えられたファイル名をチェックします。そのファイル名が *regexp*にマッチしたら、そのプリミティブが *handler*を呼び出してファイルを処理します。

*handler*の 1 つ目の引数には、プリミティブの名前をシンボルとして与えます。残りの引数はそのプリミティブに引数として渡されます (これらの引数の 1 つ目はほとんどの場合はファイル名自身)。たとえば以下を行って:

```
(file-exists-p filename)
```

*filename*がハンドラー *handler*をもつなら、*handler*は以下のように呼び出されます:

```
(funcall handler 'file-exists-p filename)
```

関数が 2 つ以上の引数を受け取る場合には、それらはファイル名でなければならず、関数はそれらのファイル名それぞれにたいしてハンドラーをチェックします。たとえば、

```
(expand-file-name filename dirname)
```

以下を行うと、*filename*にたいするハンドラーをチェックした後に *dirname*にたいするハンドラーをチェックします。どちらの場合でも *handler*は以下のように呼び出されます:

```
(funcall handler 'expand-file-name filename dirname)
```

その後に *handler*は *filename*と *dirname*のいずれかを処理するか解決する必要があります。

指定されたファイル名が 2 つ以上のハンドラーにマッチする場合には、ファイル名の中で最後に開始するマッチが優先されます。リモートファイルアクセスのようなジョブにたいするハンドラーに先立って、解凍のようなジョブにたいするハンドラーが最初に処理されるようにこのルールが選択されました。

以下は magic ファイル名ハンドラーが処理する操作です:

`access-file`、`add-name-to-file`、
`byte-compiler-base-file-name`、
`copy-directory`、`copy-file`、
`delete-directory`、`delete-file`、
`diff-latest-backup-file`、
`directory-file-name`、
`directory-files`、
`directory-files-and-attributes`、
`dired-compress-file`、`dired-uncache`、
`exec-path`、`expand-file-name`、
`file-accessible-directory-p`、
`file-acl`、

file-attributes、
file-directory-p、
file-equal-p、
file-executable-p、file-exists-p、
file-in-directory-p、
file-local-copy、
file-modes、file-name-all-completions、
file-name-as-directory、
file-name-case-insensitive-p、
file-name-completion、
file-name-directory、
file-name-nondirectory、
file-name-sans-versions、file-newer-than-file-p、
file-notify-add-watch、file-notify-rm-watch、
file-notify-valid-p、
file-ownership-preserved-p、
file-readable-p、file-regular-p、
file-remote-p、file-selinux-context、
file-symlink-p、file-system-info、
file-truename、file-writable-p、
find-backup-file-name、
get-file-buffer、
insert-directory、
insert-file-contents、
load、
make-auto-save-file-name、
make-directory、
make-directory-internal、
make-process、
make-symbolic-link、
process-file、
rename-file、set-file-acl、set-file-modes、
set-file-selinux-context、set-file-times、
set-visited-file-modtime、shell-command、
start-file-process、
substitute-in-file-name、
unhandled-file-name-directory、
vc-registered、
verify-visited-file-modtime、
write-region。

insert-file-contentsにたいするハンドラーは *visit*引数が非 nilなら、通常は(set-buffer-modified-p nil)によりそのバッファーの変更フラグをクリアーする必要があります。これにはもしそのバッファーがロックされていたら、ロックを解除する効果もあります。

ハンドラー関数は上記すべての操作を処理しなければならず、他の操作が将来追加される可能性もあります。これらの操作自体すべてを実装する必要はありません— 特定の操作にたいして特別なことを行う必要がないときには、その操作を通常の方法で処理するように、そのプリミティブを再呼び出

しできます。認識できない操作にたいしては、常にそのプリミティブを再呼び出しするべきです。以下はこれを行う方法の 1 つです:

```
(defun my-file-handler (operation &rest args)
  ;; 特別に処理する必要がある、
  ;; 特別な操作を最初にチェックする
  (cond ((eq operation 'insert-file-contents) ...)
        ((eq operation 'write-region) ...)
        ...
        ;; 関知しないその他の操作を処理する
        (t (let ((inhibit-file-name-handlers
                  (cons 'my-file-handler
                        (and (eq inhibit-file-name-operation operation)
                             inhibit-file-name-handlers)))
                 (inhibit-file-name-operation operation))
             (apply operation args)))))
```

ハンドラー関数が通常の Emacs プリミティブを呼び出す決定をした際には、無限再帰を引き起こすような同一ハンドラーからのプリミティブの再呼び出しを防ぐ必要があります。上記の例では変数 `inhibit-file-name-handlers`と `inhibit-file-name-operation`によって、これを行う方法を示しています。上記の例のように、これらを正確に使用するよう注意してください。複数ハンドラーの正しい振る舞いと、それぞれがハンドラーをもつかもしれない 2 つのファイル名にたいする操作にたいする詳細は非常に重要です。

ファイルへの実アクセスにたいして実際には特別なことを行わないハンドラー (たとえばリモートファイル名にたいしてホスト名の補完を実装するハンドラー等) は、`safe-magic`プロパティに非 `nil`をもつべきです。たとえば Emacs は通常は `PATH`内で見い出されるようなディレクトリーがプレフィックス '`/:`'によって magic ファイル名に見えるようなら、magic ファイル名にならないように保護します。しかし `safe-magic`プロパティに非 `nil`をもつハンドラーがそれらにたいして使用された場合には、'`/:`'は追加されません。

ファイル名ハンドラーは普通とは異なる方法でそれを処理 (handle) するのがどの操作 (operation) なのかを宣言するために、`operations`プロパティをもつことができます。このプロパティが非 `nil`値をもつなら、それは操作のリストであるべきです。その場合には、それらの操作だけがハンドラーを呼び出すでしょう。これは無駄を省きますが、主な目的はオートロードされるハンドラー関数が実際に処理を行うとき以外はロードされないようにすることです。

通常のプリミティブにたいして単にすべての操作を延期しても機能しません。たとえばファイル名ハンドラーが `file-exists-p`にたいして適用された場合には、通常の `load`コードは正しく機能しないでしょうから、ハンドラー自身で `load`を処理しなければなりません。しかしハンドラーが `file-exists-p`プロパティを使用して `file-exists-p`を処理しないことを宣言した場合には、普通とは異なる方法で `load`を処理する必要はなくなります。

`inhibit-file-name-handlers` [Variable]
この変数は特定の操作にたいして現在のところ使用を抑制されているハンドラーのリストを保持する。

`inhibit-file-name-operation` [Variable]
特定のハンドラーにたいしてその時点で抑制されている操作。

`find-file-name-handler` *file operation* [Function]
この関数は *file*というファイル名にたいするハンドラー関数、それが存在しなければ `nil`をリターンする。引数 *operation*はそのファイルを処理する操作であること。これはハンドラー呼び

出し時に 1 つ目の引数として渡すことになる値である。*operation*が `inhibit-file-name-operation`と等しいか、そのハンドラーの `operations`内に存在しなければ、この関数は `nil`をリターンする。

`file-local-copy` *filename* [Function]

この関数はファイル *filename*がまだローカルマシン上になければ、それをローカルマシン上の通常の非 magic ファイルにコピーする。magic ファイル名は、それらが他のマシン上のファイルを参照する場合には、`file-local-copy`操作を処理するべきである。リモートファイルアクセス以外の目的にたいして使用される magic ファイル名は、`file-local-copy`を処理するべきではない。この場合には関数はそのファイルをローカルファイルとして扱うだろう。

*filename*がローカルなら、それが magic か否かにかかわらずこの関数は何も行わずに `nil`をリターンする。それ以外ならローカルコピーファイルのファイル名をリターンする。

`file-remote-p` *filename* **&optional** *identification connected* [Function]

この関数は *filename*がリモートファイルかどうかをテストする。*filename*がローカル (リモートではない) ならリターン値は `nil`、*filename*が正にリモートならリターン値はそのリモートシステムを識別する文字列。

この識別子文字列はホスト名とユーザー名、およびリモートシステムへのアクセスに使用されるメソッドを表す文字列も同様に含めることができる。たとえばファイル名`/sudo::/some/file`にたいするリモート識別子文字列は`/sudo:root@localhost:`。

2 つの異なるファイルにたいして `file-remote-p`が同じ識別子をリターンした場合には、それらが同じファイルシステム上に格納されていて互いに配慮しつつアクセス可能であることを意味する。これはたとえば同時に両方のファイルにアクセスするリモートプロセスを開始することが可能なことを意味する。ファイル名ハンドラーの実装者はこの方式を保証する必要がある。

*identification*は文字列としてリターンされるべき識別子の一部を指定する。*identification*には `method`、`user`、`host`のシンボルを指定できる。他の値はすべて `nil`のように扱われて、それは完全な識別子文字列をリターンすることを意味する。上記の例ではリモートの `user`識別子文字列は `root`になるだろう。

*connected*が非 `nil`なら、たとえ *filename*がリモートであっても Emacs がそのホストにたいする接続をもたなければ、この関数は `nil`をリターンする。これは接続が存在しない際の接続の遅延を回避したいときに有用。

`unhandled-file-name-directory` *filename* [Function]

この関数は非 magic のディレクトリーの名前をリターンする。これは非 magic の *filename*には対応するディレクトリー名 (Section 25.9.3 [Directory Names], page 558 を参照) をリターンする。magic の *filename*には、何の値をリターンするかを決定するためにファイル名ハンドラーを呼び出す。*filename*がローカルプロセスからアクセス不能なら、ファイル名ハンドラーは `nil`をリターンすることによってそれを示すこと。

これはサブプロセスの実行に有用。すべてのサブプロセスは自身が所属するカレントディレクトリーとして非 magic ディレクトリーをもたなければならず、この関数はそれを導出するよい手段である。

`file-local-name` *filename* [Function]

この関数は *filename*のローカル部分 *(local part)* をリターンする。これはリモートホスト上でファイル名を識別する部分であり、通常はリモートファイル名からリモートホストを指定する部分とアクセス方法を取り除いた部分である。たとえば:

```
(file-local-name "/ssh:user@host:/foo/bar")
```

```
⇒ "/foo/bar"
```

この関数はリモートの *filename*にたいして、リモートプロセス (Section 38.4 [Asynchronous Processes], page 941 と Section 38.3 [Synchronous Processes], page 937 を参照) やリモートホスト上で実行されるプログラムの引数として直接使用可能なファイル名をリターンする。*filename*がローカルなら、この関数はそれを変更せずにリターンする。

`remote-file-name-inhibit-cache` [User Option]

リモートファイルの属性は、よりよいパフォーマンスのためにキャッシュすることができる。キャッシュが Emacs の制御外で変更されると、そのキャッシュ値は無効になり再読込しなければならない。

この変数が `nil`にセットされているとキャッシュ値は決して失効しない。このセッティングは Emacs 以外にリモートファイルを変更するものがないことが確実な場合のみ慎重に使用すること。これが `t`にセットされているとキャッシュ値は決して使用されない。これはもっとも安全な値であるがパフォーマンスは低下するかもしれない。

折衷的な値としてはこれを正の数字にセットする。これはキャッシュされてからその数字の秒数の間は、ャッシュ値を使用することを意味する。リモートファイルが定期的にチェックされる場合には、この変数を定期的なチェックの間隔より小さい値に let バインドするのは、よい考えかもしれない。たとえば:

```
(defun display-time-file-nonempty-p (file)
  (let ((remote-file-name-inhibit-cache
         (- display-time-interval 5)))
    (and (file-exists-p file)
         (< 0 (file-attribute-size
               (file-attributes
                (file-chase-links file)))))))
```

25.13 ファイルのフォーマット変換

Emacs はバッファー内のデータ (テキスト、テキストプロパティ、あるいはその他の情報) とファイルへの格納に適した表現との間で双方向の変換をするために複数のステップを処理します。このセクションでは、このフォーマット変換 *(format conversion)* を行う基本的な関数、すなわちファイルをバッファーに読み込む `insert-file-contents`と、バッファーをファイルに書き込む `write-region`を説明します。

25.13.1 概要

関数 `insert-file-contents`:

- 最初に、ファイルからバイトをバッファーに挿入して
- バイトを適切な文字にデコードした後に
- `format-alist`のエントリーで定義されているようにフォーマット処理してから
- `after-insert-file-functions`内の関数を呼び出す。

関数 `write-region`:

- 最初に `write-region-annotate-functions`内の関数を呼び出して
- `format-alist`のエントリーで定義されているようにフォーマット処理してから
- 文字を適切なバイトにエンコードした後に

- そのバイトでファイルを変更する。

これはもっとも低レベルでの操作を対照的に示したもので、対象の読み取りと書き込みの処理が逆順で対応しています。このセクションの残りの部分では、上記で名前を挙げた 3 つの変数を取り巻く 2 つの機能と、関連するいくつかの関数を説明します。文字のエンコードとデコードについての詳細は Section 33.10 [Coding Systems], page 864 を参照してください。

25.13.2 ラウンドトリップ仕様

読み取りと書き込みのもっとも一般的な機能は変数`format-alist`で制御されます。これはファイルフォーマット *(file format)* 仕様のリストで、Emacs バッファー内のデータにたいしてファイル内で使用されるテキスト表現を記述します。読み取りと書き込みの仕様記述はペアーになっており、わたしたちがそれを “ラウンドトリップ (round-trip)” 仕様と呼ぶのはこれが理由です (非ペアー仕様については Section 25.13.3 [Format Conversion Piecemeal], page 576 を参照)。

`format-alist` [Variable]

このリストには定義されるファイルフォーマットごとに 1 つのフォーマット定義が含まれる。フォーマット定義はそれぞれ以下の形式のリスト:

```
(name doc-string regexp from-fn to-fn modify mode-fn preserve)
```

以下はフォーマット定義内で要素がもつ意味:

name フォーマットの名前。

doc-string フォーマットのドキュメント文字列。

regexp このフォーマットで表現されるファイルの認識に使用される正規表現。`nil`ならフォーマットが自動的に適用されることは決してない。

from-fn このフォーマットのデータをデコードする、(ファイルデータを通常の Emacs データ表現に変換するための) シェルコマンドか関数。

シェルコマンドは文字列として表され、Emacs はそのコマンドを変換処理用のフィルターとして実行する。

*from-fn*が関数なら、それは変換するべきバッファー部分を指定する 2 つの引数 *begin* と *end*で呼び出される。これはインプレースでテキストを編集することにより変換を行うこと。これはテキスト長を変更する可能性があるので *from-fn*は変更された end 位置をリターンすること。

ファイルの先頭が変換により *regexp*にマッチしないようにするのは *from-fn*の役目の 1 つである。そうでないとおそらく再度変換が呼び出される。さらに *from-fn*はデコードされるバッファーやバッファーではないこと。さもなければフォーマット用の内部バッファーが上書きされるかもしれない。

to-fn このフォーマットのデータをエンコード、すなわち通常の Emacs データ表現をこのフォーマットに変換するためのシェルコマンドか関数。

*to-fn*が文字列ならそれはシェルコマンドである。Emacs は変換処理のためのフィルターとしてこのコマンドを実行する。

*to-fn*が関数なら、それは 3 つの引数で呼び出される。*begin*と *end*は変換されるべきバッファー部分、*buffer*でそれがどのバッファーかを指定する。変換を行うには 2 つの方法がある:

- そのバッファー内でインプレースで編集を行う。*to-fn*はこの場合は変更にしたがいテキスト範囲の end 位置をリターンすること。

- 注釈 (annotation) のリストをリターンする。これは (*position* . *string*) という形式の要素をもつリストで、*position*は書き込まれるテキスト内での相対位置を指定する整数、*string*はそこに追加される注釈である。このリストは *to-fn*がそれをリターンする際には、位置順でソートされていなければならない。

  `write-region`が実際にバッファーからファイルにテキストを書き込む際には、指定された注釈を対応する位置に混合する。これはすべてバッファーを変更せずに行われる。

*from-fn*はデコードされるバッファーやバッファーではないこと。さもなければフォーマット用の内部バッファーが上書きされるかもしれない。

modify フラグ。エンコード関数がバッファーを変更するなら `t`、注釈リストをリターンすることによって機能するなら `nil`。

mode-fn このフォーマットから変換されたファイルを visit 後に呼び出されるマイナーモード関数。この関数は 1 つの引数で呼び出されて、それが整数 1 ならマイナーモード関数はそのモードを有効にする。

preserve フラグ。`format-write-file`が `buffer-file-format`からこのフォーマットを取り除くべきでなければ `t`。

関数 `insert-file-contents`は指定されたファイルを読み込む際にファイルフォーマットを自動的に認識します。これはフォーマット定義の正規表現にたいしてファイルの先頭テキストをチェックして、マッチが見つかったら、そのフォーマットにたいするデコード関数を呼び出します。その後は再度すべての既知のフォーマットをチェックします。適用できるフォーマットがない間はチェックを続行します。

`find-file-noselect`やそれを使用するコマンドでファイルを visit することにより、同じように変換が行われます (内部で `insert-file-contents`を呼び出すため)。さらにそれをデコードする各フォーマットのモード関数も呼び出します。これはバッファーローカル変数 `buffer-file-format`内にフォーマット名のリストを格納します。

`buffer-file-format` [Variable]
この変数は visit しているファイルのフォーマットを表す。より正確にはこれはカレントバッファーのファイルを visit に起因するデコードのファイルフォーマット名のリストである。これはすべてのバッファーにたいして常にローカル。

`write-region`がデータをファイルに書き込む際には、まず `buffer-file-format`にリストされたフォーマットにたいするエンコード関数をリスト内での出現順に呼び出します。

`format-write-file` *file format* **&optional** *confirm* [Command]
このコマンドはカレントバッファーのコンテンツをフォーマット名のリスト *format*にもとづいたフォーマットでファイル *file*に書き込む。これは *format*を起点に、`buffer-file-format`の値から *preserve*フラグ (上記参照) が非 `nil`の要素にたいして、それがまだ *format*内に存在しなければ任意の個数それらを追加する。その後に将来の保存においてデフォルトとなるように、このフォーマットで `buffer-file-format`を更新する。*format*引数を除けばこのコマンドは `write-file`と似ている。特に *confirm*は `write-file`での対応する引数と、意味や interactive での扱いが同じである。[Definition of write-file], page 535 を参照のこと。

`format-find-file` *file format* [Command]
このコマンドはファイル *file*を探してそれをフォーマット *format*にしたがって変換する。これは後でそのバッファーを保存する場合に *format*をデフォルトにすることも行う。

引数 *format*はフォーマット名のリスト。*format*が`nil`なら何の変換も行われない。interactive に呼び出した場合には、*format*にたいして単に`RET`をタイプすると`nil`が指定される。

`format-insert-file` *file format* **&optional** *beg end* [Command]
このコマンドはファイル *file*のコンテンツをフォーマット *format*にしたがって変換して挿入する。*beg*と *end*が非`nil`なら、それは`insert-file-contents`と同様、ファイルのどの部分を読み込むかを指定する (Section 25.3 [Reading from Files], page 537 を参照)。

リターン値は絶対ファイル名のリスト、および挿入されたデータの長さ (変換後) であり、これは`insert-file-contents`がリターンするものと同様。

引数 *format*はフォーマット名のリスト。*format*が`nil`なら何の変換も行われない。interactive に呼び出した場合には、*format*にたいして単に`RET`をタイプすると`nil`が指定される。

`buffer-auto-save-file-format` [Variable]
この変数は自動保存 (auto-saving) にたいして使用するフォーマットを指定する。値は`buffer-file-format`と同様、ファイル名のリストだが、これは auto-save ファイルへの書き込みで`buffer-file-format`のかわりに使用される。値が`t`(デフォルト) なら自動保存は当バッファーの通常の保存時と同じフォーマットを使用する。この変数はすべてのバッファーにおいて常にバッファーローカル。

25.13.3 漸次仕様

前のサブセクション (Section 25.13.2 [Format Conversion Round-Trip], page 574 を参照) で説明したラウンドトリップ指定とは対照的に、変数`after-insert-file-functions`と`write-region-annotate-functions`を使用して読み取りと書き込みの変換を個別に制御できます。

変換はある表現を起点として他の表現を生成します。これを行う変換が 1 つだけのときは、何を起点とするかに関して競合は存在しません。しかし複数の変換呼び出しが存在する場合には、同じデータを起点にする必要がある 2 つの変換の間に競合が発生するかもしれません。

この状況を理解するには、`write-region`中のテキストプロパティの変換コンテキストが最善です。たとえばあるバッファーの位置 42 の文字が‘`X`’で、それのテキストプロパティが`foo`だとします。`foo`にたいする変換が、たとえばそのバッファーに‘`FOO:`’を挿入することにより行われる場合には、それは位置 42 の文字‘`X`’を‘`F`’に変更します。そして次の変換は間違ったデータを起点に開始されるでしょう。

競合を避けるためには協調的な変換がバッファーを変更せずに、*position*昇順でソートされた`(`*position* `.` *string*`)`という形式の要素をもつリストを注釈 *(annotations)* に指定します。

2 つ以上の変換が存在する場合には、`write-region`はそれらの注釈を 1 つのソート済みリストに破壊的にマージします。後でそのバッファーのテキストを実際にファイルに書き込む際に、対応する位置にある指定された注釈を混合します。これはすべてバッファーを変更せずに行われます。

読み取り時にはこれとは対照的にそのテキストの混合された注釈は即座に処理されます。`insert-file-contents`は変更される何らかのテキストの先頭にポイントをセットしてから、そのテキストの長さで変換関数を呼び出します。これらの関数は常に挿入されるテキストの先頭のポイントをリターンするべきです。最初の変換により注釈が削除されても、その後の変換が誤って処理することはないので、このアプローチは読み取りに際しては正しく機能します。すべての変換関数は、それが認識する注釈のスキャン、その注釈の削除、バッファーテキストの変更 (たとえばテキストプロパティのセット等)、およびそれらの変更に由来する更新されたテキスト長のリターンを行うべきです。 1 つの関数によりリターンされた値は次の関数への引数になります。

`write-region-annotate-functions` [Variable]

`write-region`にたいして呼び出す関数のリスト。リスト内の各関数は書き込まれるリージョンの開始と終了の 2 つの引数で呼び出される。これらの関数はそのバッファーのコンテンツを変更するべきではない。かわりに注釈をリターンすること。

特別なケースとして、関数がカレントと異なるバッファーをリターンするかもしれない。Emacs はこれを、出力される変更されたテキストをカレントバッファーが含むものとして理解する。つまり Emacs は`write-region`呼び出しの引数 *start*と *end*を、新たなバッファーの`point-min`と`point-max`に変更して与える。さらに以前のすべての注釈はこの関数により処理されるので Emacs はそれらの破棄も行う。

`write-region-post-annotation-function` [Variable]

この変数の値が非`nil`なら、それは関数であること。この関数は`write-region`完了後に引数なしで呼び出される。

`write-region-annotate-functions`内のある関数がカレントと異なるバッファーをリターンした場合には、Emacs は`write-region-post-annotation-function`を複数回呼び出す。Emacs は最後にカレントだったバッファーでそれを呼び出し、その前にカレントだったバッファーで再度これを呼び出す、... のようにして元のバッファーに戻る。

したがって `write-region-annotate-functions`内の関数は、バッファーを作成して、`kill-buffer`のそのバッファーでのローカル値にこの変数を与え、変更されたテキストでそのバッファーをセットアップして、そのバッファーをカレントにすることができる。そのバッファーは、`write-region`完了後に kill されるだろう。

`after-insert-file-functions` [Variable]

このリスト内の各関数は、挿入されるテキストの先頭にポイントがある状態で、挿入される文字数を 1 つの引数として`insert-file-contents`により呼び出される。すべての関数はポイントを未変更のまま、その関数によって変更された挿入後テキストの新たな文字数をリターンすること。

わたしたちは、ユーザーがファイル内にテキストプロパティを格納したりそれらを取得するために、そしてさまざまなデータフォーマットを体験することにより適切なフォーマットを見つけるために、これらのフックを使用して Lisp プログラムを記述することを推奨します。最終的にはわたしたちが Emacs 内にインストールできる、良質で汎用性のある拡張をユーザーが開発することを望みます。

わたしたちはテキストプロパティの名前や値として、任意の Lisp オブジェクトの処理を試みることは推奨しません — なぜなら汎用的なプログラムはおそらく記述が困難かつ低速だからです。かわりに十分な柔軟性をもちエンコードが難しすぎない、想定されるデータ型のセットを選択してください。

26 バックアップと自動保存

バックアップファイルと auto-save(自動保存) ファイルは、Emacs のクラッシュやユーザー自身のエラーからユーザーの保護を試みるための 2 つの手段です。自動保存 (auto-saving) はカレントの編集セッションを開始した以降のテキストを保存します。一方バックアップファイルはカレントセッションの前のファイルコンテンツを保存します。

26.1 ファイルのバックアップ

バックアップファイル *(backup file)* とは編集中ファイルの古いコンテンツのコピーです。Emacs は visit されているファイルにバッファーを最初に保存するときにバックアップファイルを作成します。したがってバックアップファイルには、通常はカレント編集セッションの前にあったファイルのコンテンツが含まれています。バックアップファイルを一度存在したら、そのコンテンツは変更されずに残ります。

バックアップは通常は visit されているファイルを新たな名前にリネームすることによって作成されます。オプションでバックアップファイルが visit されているファイルをコピーすることにより作成されるように指定できます。この選択により、複数の名前をもつファイルの場合に違いが生じます。また編集中のファイルの所有者が元のオーナーのままか、それとも編集ユーザーになるかにも影響し得ます。

デフォルトでは Emacs は編集中のファイルごとに単一のバックアップファイルを作成します。かわりに番号付きバックアップ (numbered backup) を要求することもできます。その場合には新たなバックアップファイルそれぞれが新たな名前を得ます。必要なくなったときには古い番号付きバックアップを削除したり、Emacs にそれらを自動的に削除させることもできます。

性能的な理由によりオペレーティングシステムはバックアップファイルのコンテンツを二次ストレージに即座に書き込まないかもしれず、オリジナルデータとバックアップデータのいずれかが変更されるまでバックアップデータをエイリアスするかもしれません。Section 25.8 [Files and Storage], page 554 を参照してください。

26.1.1 バックアップファイルの作成

`backup-buffer` [Function]

この関数は、もしそれが適切ならカレントバッファーに visit されているファイルのバックアップを作成する。これは最初のバッファー保存を行う前に `save-buffer`により呼び出される。

リネームによりバックアップが作成されると、リターン値は (*modes extra-alist backupname*) という形式のコンスセルになる。ここで *modes*は `file-modes` (Section 25.6.1 [Testing Accessibility], page 541 を参照) でリターンされるような元ファイルのモードビット、*extra-alist*は `file-extended-attributes` (Section 25.6.5 [Extended Attributes], page 548 を参照) によりリターンされるような元ファイルの拡張属性を示す alist、そして *backupname* はバックアップの名前。

他のすべての場合 (コピーによりバックアップが作成された、またはバックアップが作成されなかった) には、この関数は `nil`をリターンする。

`buffer-backed-up` [Variable]

このバッファーローカル変数は、そのバッファーのファイルがバッファーによりバックアップされたかどうかを明示する。非 `nil`ならバックアップファイルは書き込み済み、それ以外なら (バックアップが有効なら) 次回保存時にファイルはバックアップされる。この変数は永続的にローカルであり `kill-all-local-variables`はこれを変更しない。

`make-backup-files` [User Option]

この変数はバックアップファイルを作成するかどうかを決定する。非`nil`なら、Emacs は初回保存時にすべてのファイルのバックアップを作成する— ただし`backup-inhibited`が`nil`の場合 (以下参照)。

以下の例は Rmail バッファーだけで変数`make-backup-files`を変更して、それ以外では変更しない方法を示す。この変数を`nil`にセットすると、Emacs はそれらのファイルのバックアップ作成をストップするのでディスク容量の消費を節約するだろう (あなたはこのコードを init ファイルに配置したいと思うかもしれない)。

```
(add-hook 'rmail-mode-hook
          (lambda () (setq-local make-backup-files nil)))
```

`backup-enable-predicate` [Variable]

この変数の値は、あるファイルがバックアップファイルをもつべきかどうかを決定するために、特定のタイミングで呼び出される関数ある。この関数は判断対象の絶対ファイル名という 1 つの引数を受け取る。この関数が`nil`をリターンすると、そのファイルにたいするバックアップは無効になる。それ以外なら、このセクション内の他の変数がバックアップ作成の是非と方法を指定する。

デフォルト値は `normal-backup-enable-predicate`で、これは `temporary-file-directory`と`small-temporary-file-directory`内のファイルをチェックする。

`backup-inhibited` [Variable]

この変数が非`nil`ならバックアップは抑制される。これは visit されているファイル名にたいする`backup-enable-predicate`のテスト結果を記録する。さらに visit されているファイルにたいするバックアップ抑制にもとづいたその他の機構からも使用され得る。たとえば VC はバージョンコントロールシステムに管理されるファイルのバックアップを防ぐために、この変数を非`nil`にセットする。

これは永続的にローカルなのでメジャーモード変更により値は失われない。メジャーモードはこの変数ではなく、かわりに`make-backup-files`をセットすること。

`backup-directory-alist` [User Option]

この変数の値はファイル名パターンとバックアップディレクトリーの alist。各要素は以下の形式をもつ

(*regexp* . *directory*)

この場合には名前が *regexp*にマッチするファイルのバックアップが、*directory*内に作成されるだろう。*directory*には相対ディレクトリーか絶対ディレクトリーを指定できる。絶対ディレクトリーなら、マッチするすべてのファイルが同じディレクトリー内にバックアップされる。このディレクトリー内でのファイル名はクラッシュを避けるために、バックアップされるファイルの完全名のすべてのディレクトリー区切りが‘`!`’に変更される。結果の名前を切り詰めるファイルシステムでは、これは正しく機能しないだろう。

すべてのバックアップが単一のディレクトリーで行われる一般的なケースでは、alist は‘`"."`’と適切なディレクトリーからなるペアーという単一の要素を含むこと。

この変数が`nil`(デフォルト)、またはファイル名のマッチに失敗するとバックアップは元のファイルのディレクトリーに作成される。

長いファイル名がない MS-DOS ファイルシステムでは、この変数は常に無視される。

`make-backup-file-name-function` [User Option]
この変数の値はバックアップファイル名を作成する関数。関数 `make-backup-file-name`はこれを呼び出す。Section 26.1.4 [Naming Backup Files], page 582 を参照のこと。

特定のファイルにたいして特別なことを行うために、これをバッファーローカルにすることもできる。変更する場合には、`backup-file-name-p`と `file-name-sans-versions`を変更する必要もあるかもしれない。

26.1.2 リネームかコピーのどちらでバックアップするか?

Emacs のバックアップファイル作成には 2 つの方法があります:

- Emacs は元のファイルをリネームすることができ、それがバックアップファイルになる。その後、バッファーの保存は新たなファイルに書き込まれる。この手順の後には、元ファイルの他のすべての名前 (ハードリンク) はバックアップファイルを参照することになる。新たなファイルの所有者は編集を行っているユーザーになり、グループはそのディレクトリー内でそのユーザーが新たなファイルを書き込んだときのデフォルトのグループになる。
- Emacs は元のファイルをバックアップファイルにコピーでき、新たな内容はその後は元のファイルに上書きされる。この手順の後は、元のファイルの他のすべての名前 (ハードリンク) は、そのファイルの (更新された) カレントバージョンを参照し続ける。ファイルの所有者とグループは変更されない。

デフォルトの方法は 1 つ目のリネームです。

変数 `backup-by-copying`が非 `nil`なら、それは 2 つ目の方法、つまり元のファイルをコピーして新たなバッファー内容で上書きすることを意味します。変数 `file-precious-flag`が非 `nil`の場合にも、(メイン機能の副作用として) この効果があります。Section 25.2 [Saving Buffers], page 534 を参照してください。

`backup-by-copying` [User Option]
この変数が非 `nil`なら、Emacs は常にコピーによりバックアップファイルを作成する。デフォルトは `nil`。

以下の 3 つの変数が非 `nil`の際は、ある特定のケースに 2 つ目の方法が使用されます。その特定のケースに該当しないファイルの処理には影響はありません。

`backup-by-copying-when-linked` [User Option]
この変数が非 `nil`なら、Emacs は複数名 (ハードリンク) をもつファイルにたいしてコピーによりバックアップを作成する。デフォルトは `nil`。

`backup-by-copying`が非 `nil`なら常にコピーによりバックアップが作成されるので、この変数は `backup-by-copying`が `nil`のときだけ意味がある。

`backup-by-copying-when-mismatch` [User Option]
この変数が非 `nil`(デフォルト) なら、リネームによりファイルの所有者やグループが変更されるケースでは Emacs はコピーによりバックアップを作成する。

リネームによりファイルの所有者やグループが変更されなければ、値に効果はない。つまり、そのディレクトリーで新たに作成されるファイルにたいするデフォルトのグループに属するユーザーにより所有されるファイルが該当する。

`backup-by-copying`が非 `nil`なら常にコピーによりバックアップが作成されるので、この変数は `backup-by-copying`が `nil`のときだけ意味がある。

`backup-by-copying-when-privileged-mismatch` [User Option]
この変数が非`nil`なら、特定のユーザー ID およびグループ ID の値 (具体的には特定の値以下の ID 数値) にたいしてのみ、`backup-by-copying-when-mismatch`と同じように振る舞うことを指定する。変数にはその数値をセットする。

したがってファイル所有者の変更を防ぐ必要がある際には、`backup-by-copying-when-privileged-mismatch`を 0 にセットすればスーパーユーザーとグループ 0 だけがコピーによるバックアップを行うことができる。

デフォルトは 200。

26.1.3 番号つきバックアップファイルの作成と削除

ファイルの名前が `foo`なら、番号付きバックアップのバージョン名は `foo.~v~`となります。*v*は`foo.~1~`、`foo.~2~`、`foo.~3~`、...、`foo.~259~`のように、さまざまな整数です。

`version-control` [User Option]
この変数は単一の非番号付きバックアップファイルを作成するか、それとも複数の番号付きバックアップを作成するかを制御する。

`nil` visit されたファイルが番号付きバックアップなら番号付きバックアップを作成して、それ以外は作成しない。これがデフォルト。

`never` 番号付きバックアップを作成しない。

anything else
番号付きバックアップを作成する。

番号付きバックアップを使用することにより、バックアップのバージョン番号は最終的には非常に大きな番号になるので、それらを削除しなければなりません。Emacs はこれを自動で行うことができ、ユーザーに削除するか確認することもできます。

`kept-new-versions` [User Option]
この変数の値は新たな番号付きバックアップが作成された際に保持するべき、もっとも新しいバージョンの個数。新たに作成されたバックアップもカウントされる。デフォルトは 2。

`kept-old-versions` [User Option]
この変数の値は新たな番号付きバックアップが作成された際に保持するべき、もっとも古いバージョンの個数。デフォルトは 2。

番号が 1、2、3、5、7 のバックアップがあり、かつこれらの変数が値 2 をもつ場合には、番号が 1 と 2 のバックアップは古いバージョンとして保持されて、番号が 5 と 7 のバックアップは新しいバージョンとして保持される。そして番号が 3 のバックアップは余分なバックアップとなる。関数`find-backup-file-name` (Section 26.1.4 [Backup Names], page 582 を参照) は、どのバージョンのバックアップを削除するかを決定する役目を負うが、この関数自身がバックアップを削除する訳ではない。

`delete-old-versions` [User Option]
この変数が`t`なら、ファイルの保存により余分なバージョンのバックアップは暗黙に削除される。`nil`なら余分なバックアップの削除前に確認を求めるて、それ以外なら余分なバックアップは削除されないことを意味する。

`dired-kept-versions` [User Option]

この変数は Dired 内のコマンド . (ピリオド。`dired-clean-directory`) で、もっとも新しいバージョンのバックアップをいくつ保持するかを指定する。これは新たにバックアップファイルを作成する際に `kept-new-versions`を指定するのと同等。デフォルトは 2。

26.1.4 バックアップファイルの命名

このセクションでは、主にバックアップファイルの命名規則を再定義してカスタマイズできる関数を記載します。これらの 1 つを変更した場合には、おそらく残りも変更する必要があります。

`backup-file-name-p` *filename* [Function]

この関数は *filename*がバックアップファイルとして利用可能なら非 `nil`値をリターンする。これは名前のチェックだけを行って、*filename*という名前のファイルが存在するかどうかはチェックしない。

```
(backup-file-name-p "foo")
     ⇒ nil
(backup-file-name-p "foo~")
     ⇒ 3
```

この関数の標準的な定義は、以下のようになる:

```
(defun backup-file-name-p (file)
  "FILE がバックアップファイルなら\
(番号付きか否かに関わらず) 非 nil をリターンする"
  (string-match "~\\'" file))
```

このようにファイル名が ‘~’で終われば、この関数は非 `nil`値をリターンする (ドキュメント文字列を分割するために 1 行目でバックスラッシュを使用しているが、これはドキュメント文字列内で単一行を生成する)。

この単純な式はカスタマイズのための再定義を簡便にするために、個々の関数内に配置されている。

`make-backup-file-name` *filename* [Function]

この関数はファイル *filename*の非番号付きバックアップファイル名として使用される文字列をリターンする。Unix ではこれは単に *filename*にチルダを追加する。

ほとんどのオペレーティングシステムでは、この関数の標準的な定義は以下のようになる:

```
(defun make-backup-file-name (file)
  "FILE にたいして非番号付きバックアップファイル名を作成する"
  (concat file "~"))
```

この関数を再定義することにより、バックアップファイルの命名規則を変更できる。以下はチルダの追加に加えて、先頭に ‘.’を追加するように `make-backup-file-name`を再定義する例:

```
(defun make-backup-file-name (filename)
  (expand-file-name
    (concat "." (file-name-nondirectory filename) "~")
    (file-name-directory filename)))

(make-backup-file-name "backups-ja.texi")
     ⇒ ".backups-ja.texi~"
```

Dired コマンドのいくつかを含む Emacs の一部では、バックアップファイル名が ‘~’で終わることを仮定している。この規則にしたがわない場合、深刻な問題とはならないだろうが、それらのコマンドが若干好ましくない結果をもたらすかもしれない。

`find-backup-file-name` *filename* [Function]

この関数は *filename*の新たなバックアップファイル用のファイル名を計算する。これは特定の既存バックアップファイルにたいする削除の提案も行うかもしれない。`find-backup-file-name`は CAR が新たなバックアップファイル名、CDR が削除を提案するバックアップファイルのリストであるようなリストをリターンする。値には `nil`も指定でき、これはバックアップが作成されないことを意味する。

`kept-old-versions`と `kept-new-versions`の 2 つの変数は、どのバージョンのバックアップを保持するべきかを決定する。この関数は値の CDR から該当するバージョンを除外することによってそれらを保持する。Section 26.1.3 [Numbered Backups], page 581 を参照のこと。

以下の例の値は新しいバックアップファイルに使用する名前が`~rms/foo.~5~`、`~rms/foo.~3~`は呼び出し側が削除を検討するべき余分なバージョンであることを示している。

```
(find-backup-file-name "~rms/foo")
     ⇒ ("~rms/foo.~5~" "~rms/foo.~3~")
```

`file-newest-backup` *filename* [Function]

この関数は *filename*にたいするもっとも最近のバックアップファイル名、バックアップファイルがなければ `nil`をリターンする。

ファイル比較関数のいくつかは、自動的にもっとも最近のバックアップを比較できるようにこの関数を使用している。

26.2 自動保存

Emacs は、visit しているすべてのファイルを定期的に保存します。これは自動保存 *(auto-saving)* と呼ばれます。自動保存はシステムがクラッシュした場合に失われる作業量を、一定の作業量以下にします。デフォルトでは自動保存は 300 キーストロークごと、または idle になった 30 秒後に発生します。自動保存に関するユーザー向けの情報については Section “Auto-Saving: Protection Against Disasters” in *The GNU Emacs Manual* を参照してください。ここでは自動保存の実装に使用される関数と、それらを制御する変数について説明します。

`buffer-auto-save-file-name` [Variable]

このバッファーローカル変数はカレントバッファーの自動保存に使用されるファイル名。そのバッファーが自動保存されるべきでなければ `nil`。

```
buffer-auto-save-file-name
     ⇒ "/xcssun/users/rms/lewis/#backups-ja.texi#"
```

`auto-save-mode` *arg* [Command]

これはバッファーローカルなマイナーモードである Auto Save モードにたいするモードコマンド。Auto Save モードが有効なときはそのバッファーで自動保存が有効。呼び出し方法は他のマイナーモードと同様 (Section 23.3.1 [Minor Mode Conventions], page 480 を参照)。

ほとんどのマイナーモードと異なり `auto-save-mode`変数は存在しない。`buffer-auto-save-file-name`が非 `nil`で `buffer-saved-size`(以下参照) が非 0 なら Auto Save モードが有効。

`auto-save-file-name-p` *filename* [Function]

この関数は *filename*が auto-save ファイルのような文字列なら非 `nil`をリターンする。先頭と末尾がハッシュマーク (‘`#`’) であるような名前は auto-save ファイルの可能性があるという、

auto-save ファイルにたいする通常の命名規則を想定する。引数 *filename*はディレクトリーパートを含まないこと。

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#backups-ja.texi#"
(auto-save-file-name-p "#backups-ja.texi#")
     ⇒ 0
(auto-save-file-name-p "backups-ja.texi")
     ⇒ nil
```

この関数の標準的な定義は、以下のようになる:

```
(defun auto-save-file-name-p (filename)
  "FILENAME が以下を満たすなら非 nil をリターンする"
  (string-match "^#.*#$" filename))
```

この関数は auto-save ファイルの命名規則規則を変更したいときにカスタマイズできるようにするために存在する。これを再定義するなら、それに対応して関数 `make-auto-save-file-name`も忘れずに再定義すること。

`make-auto-save-file-name` [Function]

この関数はカレントバッファーの自動保存に使用されるファイル名をリターンする。これはファイル名の先頭と末尾にハッシュマーク (‘`#`’) を単に追加する。この関数は変数 `auto-save-visited-file-name`(以下参照) を調べない。呼び出し側はまずその変数をチェックすること。

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#backups-ja.texi#"
```

以下はこの関数の標準的な定義の簡略版:

```
(defun make-auto-save-file-name ()
  "カレントバッファーの自動保存に使用される\
ファイル名をリターンする"
  (if buffer-file-name
      (concat
       (file-name-directory buffer-file-name)
       "#"
       (file-name-nondirectory buffer-file-name)
       "#")
    (expand-file-name
     (concat "#%" (buffer-name) "#"))))
```

auto-save ファイルの命名規則をカスタマイズして再定義できるように、これは独立した関数として存在している。ただしこれに対応した方法で `auto-save-file-name-p`も忘れずに変更すること。

`auto-save-visited-file-name` [User Option]

この変数が非 `nil`なら Emacs は visit 中のファイルにバッファーを自動保存する。つまり自動保存は編集中ファイルと同じファイルにたいして行われる。この変数は通常は `nil`なので、auto-save ファイルは `make-auto-save-file-name`で作成された別の名前をもつ。

この変数の値を変更した際、バッファー内で auto-save モードを再度有効にするまで、既存バッファーにたいして新たな値は効果をもたない。すでに auto-save モードが有効なら、再度 `auto-save-mode`が呼び出されるまで同じファイルに自動保存が行われる。

この変数を非 nilにセットしても自動保存とバッファーの保存は異なるという事実は変わらないことに注意 (Section 25.2 [Saving Buffers], page 534 で説明したフックはバッファーが自動保存された際には実行されない)。

recent-auto-save-p [Function]
この関数はカレントバッファーが最後に読み込み、または保存されて以降に自動保存されていればtをリターンする。

set-buffer-auto-saved [Function]
この関数はカレントバッファーを自動保存済みとマークする。そのバッファーは、バッファーテキストが再度変更されるまで自動保存されないだろう。この関数は nilをリターンする。

auto-save-interval [User Option]
この変数の値は自動保存の頻度を入力イベント数で指定する。この分の入力イベント読み取りごとに、Emacs は自動保存が有効なすべてのバッファーにたいして自動保存を行う。これを 0 にするとタイプした文字数にもとづいた自動保存は無効になる。

auto-save-timeout [User Option]
この変数の値は自動保存が発生すべき idle 時間の秒数。この秒数分ユーザーが休止するたびに、Emacs は自動保存が有効なすべてのバッファーにたいして自動保存を行う (カレントバッファーが非常に大きければ、指定されたタイムアウトはサイズ増加とともに増加される因子で乗ぜられる。この因子は 1MB のバッファーにたいしておよそ 4)。

値が 0 か nilなら idle 時間にもとづいた自動保存は行われず、auto-save-intervalで指定される入力イベント数の後のみ自動保存が行われる。

auto-save-hook [Variable]
このノーマルフックは自動保存が行われようとするたびに毎回実行される。

auto-save-default [User Option]
この変数が非 nilならファイルを visit するバッファーの自動保存がデフォルトで有効になり、それ以外では有効にならない。

do-auto-save **&optional** *no-message current-only* [Command]
この関数は自動保存される必要があるすべてのバッファーを自動保存する。これは自動保存が有効なバッファーであり、かつ前回の自動保存以降に変更されたすべてのバッファーを保存する。

いずれかのバッファーが自動保存される場合には、do-auto-saveは自動保存が行われる間、通常はそれを示すメッセージ‘Auto-saving...’をエコーエリアに表示する。しかし *no-message* が非 nilならこのメッセージは抑制される。

*current-only*が非 nilなら、カレントバッファーだけが自動保存される。

delete-auto-save-file-if-necessary **&optional** *force* [Function]
この関数は delete-auto-save-filesが非 nilならカレントバッファーの auto-save ファイルを削除する。これはバッファー保存時に毎回呼び出される。

この関数は *force*が nilなら最後に本当の保存が行われて以降、カレント Emacs セッションにより書き込まれたファイルだけを削除する。

delete-auto-save-files [User Option]
この変数は関数 delete-auto-save-file-if-necessaryにより使用される。これが非 nilなら、Emacs は (visit されているファイルに) 本当に保存が行われたときに auto-save ファイルを削除する。これはディスク容量を節約してディレクトリーを整理する。

`rename-auto-save-file` [Function]
この関数は visit されているファイルの名前が変更されていればカレントバッファーの auto-save ファイルの名前を調整する。これはカレント Emacs セッションで auto-save ファイルが作成されていれば、既存の auto-save ファイルのリネームも行う。visit されているファイルの名前が変更されていなければ、この関数は何も行わない。

`buffer-saved-size` [Variable]
このバッファーローカル変数の値はカレントバッファーが最後に読み取り、保存、または自動保存されたときのバッファーの長さ。これはサイズの大幅な減少の検知に使用され、それに応じて自動保存がオフに切り替えられる。

−1 なら、それはサイズの大幅な減少によりそのバッファーの自動保存が一時的に停止されていることを意味する。明示的な保存によりこの変数に正の値が格納されて、自動保存が再び有効になる。自動保存をオフやオンに切り替えることによってもこの変数は更新されるので、サイズの大幅な減少は忘れられさられる。

−2 なら、特にバッファーサイズの変更により一時的に自動保存を停止されないように、そのバッファーがバッファーサイズの変更を無視することを意味する。。

`auto-save-list-file-name` [Variable]
この変数は、(非`nil`なら) すべての auto-save ファイルの名前を記録するファイルを指定する。Emacs が自動保存を行うたびには自動保存が有効な各バッファーごとに 2 行ずつ書き込みを行う。1 行目は visit されているファイルの名前 (ファイルを visit しないバッファーの場合は空)、2 行目は auto-save ファイルの名前を示す。

Emacs を正常に exit した際にこのファイルは削除される。Emacs がクラッシュした場合にはこのファイルを調べることにより、失われるはずだった作業を含んだすべての auto-save ファイルを探すことができる。`recover-session`コマンドはそれらを見つけるためにこのファイルを使用する。

このファイルにたいするデフォルト名は、ユーザーのホームディレクトリーにある‘`.saves-`’で始まるファイルを指定する。この名前には Emacs のプロセス IDとホスト名も含まれる。

`auto-save-list-file-prefix` [User Option]
init ファイルを読み込んだ後、(`nil`にセット済みでなければ)Emacs はこのプレフィックスにもとづいたホスト名とプロセス ID を追加して、`auto-save-list-file-name`を初期化する。init ファイル内でこれを`nil`にセットした場合には、Emacs は`auto-save-list-file-name`を初期化しない。

26.3 リバート

あるファイルにたいして大きな変更を行った後、気が変わって元に戻したくなった場合は、`revert-buffer`コマンドでそのファイルの以前のバージョンを読み込むことにより、それらの変更を取り消すことができます。詳細は、Section “Reverting a Buffer” in *The GNU Emacs Manual* を参照してください。

`revert-buffer` **&optional** *ignore-auto noconfirm preserve-modes* [Command]
このコマンドはバッファーのテキストをディスク上の visit されているファイルのテキストで置き換える。これによりファイルが visit や保存された以降に行ったすべての変更はアンドゥ (undo: 取り消し) される。

デフォルトでは、最新の auto-save ファイルのほうが visit されているファイルより新しく引数 *ignore-auto*が`nil`なら、`revert-buffer`はユーザーにたいしてかわりに auto-save ファイル

を使用するかどうか確認を求める。このコマンドを interactive に呼び出したときプレフィックス数引数が指定されていなければ、*ignore-auto*は `t`となる。つまり interactive 呼び出しは、デフォルトでは auto-save ファイルのチェックを行わない。

`revert-buffer`は通常はバッファーを変更する前に確認を求める。しかし引数 *noconfirm*が非 `nil`なら `revert-buffer`は確認を求めない。

このコマンドは通常は `normal-mode`を使用することにより、そのバッファーのメジャーモードとマイナーモードを再初期化する。しかし *preserve-modes*が非 `nil`ならモードは変更されずに残る。

リバート (revert: 戻す、復元する) は `insert-file-contents`の置き換え機能を使用することにより、バッファー内のマーカー位置の保持を試みる。バッファーのコンテンツとファイルのコンテンツがリバート操作を行う前と等しければリバートはすべてのマーカーを保持する。等しくなければリバートによってバッファーは変更される。この場合は、(もしあれば) バッファーの最初と最後にある未変更のテキスト内にあるマーカーは保持される。他のマーカーを保持してもそれらは正しくないだろう。

`revert-buffer-in-progress-p` [Variable]
`revert-buffer`は処理を行っている間、この変数を非 `nil`値にバインドする。

このセクションの残りの部分で説明する変数をセットすることにより、`revert-buffer`が処理方法をカスタマイズできます。

`revert-without-query` [User Option]
この変数は問い合わせなしでリバートされるファイルのリストを保持する。値は正規表現のリスト。visit されているファイルの名前がこれらの正規表現のいずれかにマッチし、かつバッファーが未変更だがディスク上のファイルは変更されていれば、`revert-buffer`はユーザーに確認を求めることなくファイルをリバートする。

いくつかのメジャーモードは以下の変数をローカルにバインドすることにより `revert-buffer`をカスタマイズします:

`revert-buffer-function` [Variable]
この変数の値はそのバッファーをリバートするために使用する関数。これはリバート処理を行うために 2 つのオプション引数をとる関数であること。2 つのオプション引数 *ignore-auto*と *noconfirm*は `revert-buffer`が受け取る引数である。

Dired モードのような編集されるテキストにファイルのコンテンツが含まれず他の方式によって再生成され得るモードは、この変数のバッファーローカル値にコンテンツを再生成する特別な関数を与えることができる。

`revert-buffer-insert-file-contents-function` [Variable]
この変数の値はそのバッファーをリバートする際に更新されたコンテンツの挿入に使用される関数を指定する。その関数は 2 つの引数を受け取る。1 つ目は使用するファイル名で 2 つ目が `t`なら、ユーザーは auto-save ファイルの読み込みにたいして確認を求められる。

`revert-buffer-function`のかわりにこの変数をモードが変更する理由は、`revert-buffer`が行残りの処理 (ユーザーへの確認、アンドゥリストのクリアー、適切なメジャーモードの決定、以下のフックの実行) にたいする重複や置き換えを避けるためである。

`before-revert-hook` [Variable]
このノーマルフックは変更されたコンテンツを挿入する前に、デフォルトの `revert-buffer-function`により実行される。カスタマイズした `revert-buffer-function`は、このフックを実行するかどうか判らない。

`after-revert-hook` [Variable]

このノーマルフックは変更されたコンテンツを挿入した後に、デフォルトの `revert-buffer-function`によって実行される。カスタマイズした `revert-buffer-function`は、このフックを実行するかどうか判らない。

Emacs はバッファーを自動的にリバートできます。これはファイルを visit しているバッファーにはデフォルトで行われます。以下では新たなタイプのバッファーにたいして自動リバートのサポートを追加する方法を説明します。

そのようなバッファーはまず適切に定義された `revert-buffer-function`と `buffer-stale-function`をもたなければなりません。

`buffer-stale-function` [Variable]

この変数の値はバッファーがリバートを要するかどうかをチェックするために呼び出される関数を指定する。デフォルト値では、修正時刻をチェックすることによりファイルを visit するバッファーだけを処理する。ファイルを visit しないバッファーにはオプション引数を 1 つ受け取るカスタム関数が必要になる。この関数はバッファーをリバートする必要があれば非 `nil`をリターンする。関数の呼び出し時にはそのバッファーがカレントになる。

この関数は主として自動リバートを意図しているが、他の用途にも使用できる。たとえば自動リバートが有効でなければ、ユーザーにバッファーのリバートが必要なバッファーを警告するために使用できる。*noconfirm*引数の背景にあるアイデアは、ユーザーへの確認なしでバッファーがリバートされるようなら `t`、関数がバッファーの期限切れをユーザーに警告する。特に自動リバートにおける使用では *noconfirm*は `t`になる。関数が自動リバートにたいしてのみ使用されるようなら *noconfirm*引数は無視できる。

(Buffer Menu のように) `auto-revert-interval`秒ごとに自動リバートをしたければバッファーのモード関数で:

```
(setq-local buffer-stale-function
     (lambda (&optional noconfirm) 'fast))
```

を使用する。

特別なリターン値 '`fast`'はリバートの必要性はチェックしていないがバッファーのリバートは高速であることを告げる。さらに `auto-revert-verbose`が非 `nil`でもリバートメッセージを何もプリントしないように Auto Revert に指示する。`auto-revert-interval`秒ごとのリバートメッセージが非常に煩雑になり得ることから、これは重要である。自動リバート以外の目的でこの関数が考慮される場合には、リターン値が提供する情報は有用になるかもしれない。

バッファーが適切な `revert-buffer-function`と `buffer-stale-function`をもった後にも、通常はいくつかの問題が残ります。

未変更とマークされた場合のみバッファーは自動リバートされます。したがって種々の関数はリバートにより失われるかもしれない情報をバッファーが含む場合や、ユーザーがバッファーにたいして作業を行っていて自動リバートがユーザーにとって不便だと確信できる理由がある場合にのみバッファーを変更済みとマークすることを保証する必要があるでしょう。バッファーの変更状態を手動で調整することによりユーザーは常にこれをオーバーライドできます。これをサポートするために、未変更とマークされたバッファーでの `revert-buffer-function`呼び出しでは、常にバッファーの未変更とマークされた状態を保つ必要があります。

自動リバートの結果としてその前後で連続してポイントがジャンプしないことを保証することが重要です。もちろんバッファーが根本的に変更されればポイントの移動は必然的かもしれません。

`revert-buffer-function`が `auto-revert-verbose`が `t`の場合に表示される Auto Revert 自身のメッセージを不必要に重複してプリントしないことと、`auto-revert-verbose`にたいする `nil`値を効果的にオーバーライドすることを保証する必要があります。したがって自動リバートにたいするモードの調整には、このようなメッセージを取り除くことを要する場合が多々あります。とりわけこれは `auto-revert-interval`秒ごとに自動的にリバートされるバッファーにおいて重要です。

新しい自動リバートが Emacs の一部であるなら、`global-auto-revert-non-file-buffers` のドキュメント文字列でそれに言及するべきです。

同様に Emacs マニュアル内に追加でドキュメントするべきです。

27 バッファー

バッファー *(buffer)* とは編集されるテキストを含んだ Lisp オブジェクトのことです。バッファーは visit されるファイルのコンテンツを保持するために使用されます。しかしファイルを visit しないバッファーも存在します。一度に複数のバッファーが存在するかもしれませんが、カレントバッファー *(current buffer)* に指定できるのは常に 1 つのバッファーだけです。ほとんどの編集コマンドはカレントバッファーのコンテンツにたいして作用します。カレントバッファーを含むすべてのバッファーは任意のウィンドウ内に表示されるときもあれば、表示されない場合もあります。

27.1 バッファーの基礎

Emacs 編集におけるバッファーとは個別に名前をもち、編集可能なテキストを保持するオブジェクトです。Lisp プログラムにおけるバッファーはスペシャルデータ型として表されます。バッファーのコンテンツを拡張可能な文字列と考えることができます。挿入と削除はバッファー内の任意の箇所で発生し得ます。Chapter 32 [Text], page 777 を参照してください。

Lisp のバッファーオブジェクトは多くの情報要素を含んでいます。これらの情報のいくつかは変数を通じてプログラマーが直接アクセスできるのにたいして、その他の情報は特殊な目的のための関数を通じてのみアクセスすることができます。たとえば visit されているファイルの名前は変数を通じて直接アクセスできますが、ポイント値はプリミティブ関数からのみアクセスできます。

直接アクセス可能なバッファー固有の情報は、バッファーローカル *(buffer-local)* な変数バインディング内に格納されます。これは特定のバッファー内だけで効力のある変数値のことです。この機能により、それぞれのバッファーは特定の変数の値をオーバーライドすることができます。ほとんどのメジャーモードはこの方法で `fill-column`や `comment-column`のような変数をオーバーライドしています。バッファーローカルな変数、およびそれらに関連する関数についての詳細は Section 12.11 [Buffer-Local Variables], page 190 を参照してください。

バッファーからファイルを visit する関数および変数については Section 25.1 [Visiting Files], page 530 と Section 25.2 [Saving Buffers], page 534 を参照してください。ウィンドウ内へのバッファー表示に関連する関数および変数については Section 28.11 [Buffers and Windows], page 635 を参照してください。

`bufferp` *object* [Function]

この関数は *object*がバッファーなら `t`、それ以外は `nil`をリターンする。

27.2 カレントバッファー

一般的に 1 つの Emacs セッション内には多くのバッファーが存在します。常にそれらのうちの 1 つがカレントバッファー *(current buffer)* に指定されます。カレントバッファーとは、ほとんどの編集が行われるバッファーのことです。テキストを調べたり変更するプリミティブのほとんどは暗黙にカレントバッファーにたいして処理を行います (Chapter 32 [Text], page 777 を参照)。

通常は選択されたウィンドウ内に表示されるバッファーがカレントバッファーですが、常にそうではありません。Lisp プログラムはバッファーのコンテンツを処理するために、スクリーン上に表示されているものを変更することなく任意のバッファーを一時的にカレントに指定できます。カレントバッファーの指定にたいするもっとも基本的な関数は `set-buffer`です。

`current-buffer` [Function]

この関数はカレントバッファーをリターンする。

```
(current-buffer)
     ⇒ #<buffer buffers-ja.texi>
```

`set-buffer` *buffer-or-name* [Function]

この関数は *buffer-or-name*をカレントバッファーにする。*buffer-or-name*は既存のバッファー、または既存のバッファーの名前でなければならない。リターン値はカレントになったバッファー。

この関数はそのバッファーをどのウィンドウにも表示しないので、必然的にユーザーはそのバッファーを見ることはできない。しかし Lisp プログラムはその後に、そのバッファーにたいして処理を行うことになるだろう。

編集コマンドがエディターコマンドループにリターンする際、Emacs は選択されたウィンドウ内に表示されているバッファーにたいして、自動的に `set-buffer`を呼び出します。これは混乱を防ぐためであり、これにより Emacs がコマンドを読み取るときにカーソルのあるバッファーが、コマンドを適用されるバッファーになることが保証されます (Chapter 21 [Command Loop], page 375 を参照)。したがって異なるバッファーを指示して切り替える場合には `set-buffer`を使用するべきではありません。これを行うためには Section 28.12 [Switching Buffers], page 637 で説明されている関数を使用してください。

Lisp 関数を記述する際は、処理後にカレントバッファーをリストアするためにコマンドループのこの振る舞いに依存しないでください。編集コマンドはコマンドループだけではなく、他のプログラムから Lisp 関数としても呼び出されます。呼び出し側にとっては、そのサブルーチンがカレントだったバッファーを変更しないほうが便利です (もちろんそれがサブルーチンの目的でない場合)。

他のバッファーにたいして一時的に処理を行うには、`save-current-buffer`フォーム内に `set-buffer`を配置します。以下の例はコマンド `append-to-buffer`の簡略版です:

```
(defun append-to-buffer (buffer start end)
  "リージョンのテキストを BUFFER に追加する"
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-current-buffer
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

ここではカレントバッファーを記録するためにローカル変数にバインドしてから、後で `save-current-buffer`がそれを再びカレントにするようにアレンジしています。次に `set-buffer`が指定されたバッファーをカレントにして、`insert-buffer-substring`が元のバッファーの文字列を指定された (今はカレントの) バッファーにコピーします。

かわりに `with-current-buffer`マクロを使用することもできます:

```
(defun append-to-buffer (buffer start end)
  "BUFFER にリージョンのテキストを追加する"
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (with-current-buffer (get-buffer-create buffer)
      (insert-buffer-substring oldbuf start end))))
```

いずれのケースでも、追加されるバッファーが偶然他のウィンドウに表示されていると、次回の再表示でそのテキストがどのように変更されたか表示されるでしょう。どのウィンドウにも表示されていなければスクリーン上で即座に変更を目にすることはありません。コマンドはバッファーを一時的にカレントにしますが、そのことがバッファーの表示を誘発する訳ではありません。

バッファーローカルなバインディングをもつ変数にたいして、(`let`や関数引数などで) ローカルバインディングを作成する場合には、そのローカルバインディングのスコープの最初と最後で同じバッ

ファーがカレントとなることを確認してください。そうしないと、あるバッファーではバインドして他のバッファーではバインドされないことになるかもしれません!

`set-buffer`の使用において、カレントバッファーが戻ることに依存しないでください。なぜなら間違ったバッファーがカレントのときに quit が発生した場合には、その処理は行われないでしょう。たとえば上記の例に倣うと以下は間違ったやり方です:

```
(let ((oldbuf (current-buffer)))
  (set-buffer (get-buffer-create buffer))
  (insert-buffer-substring oldbuf start end)
  (set-buffer oldbuf))
```

例で示したように`save-current-buffer`や`with-current-buffer`を使用すれば、quit や`throw`を通常の評価と同様に処理できます。

`save-current-buffer` *body...* [Special Form]

スペシャルフォーム`save-current-buffer`はカレントバッファーの識別を保存して *body* フォームを評価し、最後にそのバッファーをカレントにリストアする。リターン値は *body*内の最後のフォームの値。`throw`やエラーを通じた異常 exit の場合にもカレントバッファーはリストアされる (Section 11.7 [Nonlocal Exits], page 162 を参照)。

カレントとして使用されていたバッファーが`save-current-buffer`による exit 時に kill されていたら、当然それが再びカレントとなることはない。かわりに exit 直前にカレントバッファーが何であれ、それがカレントになる。

`with-current-buffer` *buffer-or-name body...* [Macro]

`with-current-buffer`マクロはカレントバッファーの識別を保存して *buffer-or-name*をカレントにし、*body*フォームを評価してから最後にカレントバッファーをリストアする。*buffer-or-name*には既存のバッファー、または既存のバッファー名を指定しなければならない。

リターン値は *body*内の最後のフォームの値。`throw`やエラーを通じた異常 exit の場合にも、カレントバッファーはリストアされる (Section 11.7 [Nonlocal Exits], page 162 を参照)。

`with-temp-buffer` *body...* [Macro]

`with-temp-buffer`マクロは一時的なバッファーをカレントバッファーとして *body*フォームを評価する。これはカレントバッファーの識別を保存して一時的なバッファーを作成、それをカレントとして *body*フォームを評価して、一時バッファーを kill する間に以前のカレントバッファーをリストアする。デフォルトではこのマクロにより作成されたバッファー内のアンドゥ情報 (Section 32.9 [Undo], page 793 を参照) は記録されない (が必要なら *body*でそれを有効にできる)。

リターン値は *body*内の最後のフォームの値。最後のフォームとして`(buffer-string)`を使用することにより、一時バッファーのコンテンツをリターンできる。

`throw`やエラーを通じた異常 exit の場合にも、カレントバッファーはリストアされる (Section 11.7 [Nonlocal Exits], page 162 を参照)。

[Writing to Files], page 539 の`with-temp-file`も参照のこと。

27.3 バッファーの名前

それぞれのバッファーは文字列で表される一意な名前をもちます。バッファーにたいして機能する関数の多くは、引数としてバッファーとバッファー名の両方を受け入れます。*buffer-or-name*という名前の引数がこのタイプであり、それが文字列でもバッファーでもなければエラーがシグナルされます。*buffer*という名前の引数は名前ではなく実際のバッファーオブジェクトでなければなりません。

短命でユーザーが関心をもたないようなバッファーは名前がスペースで始まり、それらについては `list-buffers`と `buffer-menu`コマンドは無視します (がファイルを visit しているようなバッファーは無視されない)。スペースで始まる名前は初期状態ではアンドゥ情報の記録も無効になっています。Section 32.9 [Undo], page 793 を参照してください。

`buffer-name` **&optional** *buffer* [Function]

この関数は *buffer*の名前を文字列としてリターンする。*buffer*のデフォルトはカレントバッファー。

`buffer-name`が `nil`をリターンした場合、それは *buffer*が kill されていることを意味する。Section 27.10 [Killing Buffers], page 603 を参照のこと。

```
(buffer-name)
     ⇒ "buffers-ja.texi"

(setq foo (get-buffer "temp"))
     ⇒ #<buffer temp>
(kill-buffer foo)
     ⇒ nil
(buffer-name foo)
     ⇒ nil
foo
     ⇒ #<killed buffer>
```

`rename-buffer` *newname* **&optional** *unique* [Command]

この関数はカレントバッファーを *newname*にリネームする。*newname*が文字列でなければエラーをシグナルする。

*newname*がすでに使用済みなら、`rename-buffer`は通常はエラーをシグナルする。しかし *unique*が非 `nil`なら、未使用の名前となるように *newname*を変更する。interactive に呼び出した場合は、プレフィックス数引数により *unique*に非 `nil`を指定できる (この方法によってコマンド `rename-uniquely`は実装される)。

この関数は実際にバッファーに与えられた名前をリターンする。

`get-buffer` *buffer-or-name* [Function]

この関数は *buffer-or-name*で指定されたバッファーをリターンする。*buffer-or-name*が文字列で、かつそのような名前のバッファーが存在しなければ値は `nil`。*buffer-or-name*がバッファーなら与えられたバッファーをリターンする。これは有用とは言い難く、引数は通常は名前である。たとえば:

```
(setq b (get-buffer "lewis"))
     ⇒ #<buffer lewis>
(get-buffer b)
     ⇒ #<buffer lewis>
(get-buffer "Frazzle-nots")
     ⇒ nil
```

Section 27.9 [Creating Buffers], page 602 の関数 `get-buffer-create`も参照のこと。

`generate-new-buffer-name` *starting-name* **&optional** *ignore* [Function]

この関数は新たなバッファーにたいして一意となるような名前をリターンする — がバッファーは作成しない。この名前は *starting-name*で始まり内部が数字であるような '`<...>`'を追加す

ることにより、すべてのバッファーでカレントで使用されていない名前を生成する。この数字は 2 で始まり、既存バッファーの名前でないような名前になる数字まで増加される。

オプション引数 *ignore*が非 `nil`なら、それは潜在的にバッファー名であるような文字列であること。これは、たとえそれが (通常は拒絶されるであろう) 既存バッファーの名前であっても、試みられた場合には潜在的に受容可能なバッファーとして考慮することを意味する。つまり‘`foo`’、‘`foo<2>`’、‘`foo<3>`’、‘`foo<4>`’という名前のバッファーが存在する場合には、

```
(generate-new-buffer-name "foo")
     ⇒ "foo<5>"
(generate-new-buffer-name "foo" "foo<3>")
     ⇒ "foo<3>"
(generate-new-buffer-name "foo" "foo<6>")
     ⇒ "foo<5>"
```

Section 27.9 [Creating Buffers], page 602 の関連する関数 `generate-new-buffer`も参照のこと。

27.4 バッファーのファイル名

バッファーファイル名 *(buffer file name)* とは、そのバッファーに visit されているファイルの名前です。バッファーがファイルを visit していなければ、バッファーファイル名は `nil`です。バッファー名は大抵はバッファーファイル名の非ディレクトリーパートと同じですが、バッファーファイル名とバッファー名は別物であり個別にセットすることができます。Section 25.1 [Visiting Files], page 530 を参照してください。

`buffer-file-name` **&optional** *buffer* [Function]

この関数は *buffer*が visit しているファイルの絶対ファイル名をリターンする。*buffer*がファイルを visit していなければ、`buffer-file-name`は `nil`をリターンする。*buffer*が与えられない場合のデフォルトはカレントバッファー。

```
(buffer-file-name (other-buffer))
     ⇒ "/usr/user/lewis/manual/files-ja.texi"
```

`buffer-file-name` [Variable]

このバッファーローカル変数はカレントバッファーに visit されているファイルの名前、ファイルを visit していなければ `nil`。これは永続的なローカル変数であり `kill-all-local-variables`の影響を受けない。

```
buffer-file-name
     ⇒ "/usr/user/lewis/manual/buffers-ja.texi"
```

他のさまざまな事項を変更せずにこの変数を変更するのは危険である。通常は `set-visited-file-name`を使用するほうがよい (以下参照)。バッファー名の変更などのような、そこで行われることのいくつかは絶対必要という訳ではないが、その他の事項は Emacs が混乱するのを防ぐために必要不可欠である。

`buffer-file-truename` [Variable]

このバッファーローカル変数はカレントバッファーに visit されているファイルの省略された形式の実名 (truename)、ファイルを visit していなければ `nil`を保持する。これは永続的にローカルであり `kill-all-local-variables`の影響を受けない。See Section 25.6.3 [Truenames], page 544 と [abbreviate-file-name], page 559 を参照のこと。

`buffer-file-number` [Variable]

このバッファーローカル変数はカレントバッファーに visit されているファイルのファイル番号 (file number) とデバイス番号 (device number)、ファイルを visit していなければ `nil`を保持する。これは永続的にローカルであり `kill-all-local-variables`の影響を受けない。

値は通常は `(`*`filenum devnum`*`)`のような形式のリスト。この番号ペアはシステム上でアクセス可能なすべてのファイルの中からファイルを一意に識別する。より詳細な情報は Section 25.6.4 [File Attributes], page 545 の `file-attributes`を参照のこと。

`buffer-file-name`がシンボリックリンク名なら、いずれの番号も再帰的なターゲットを参照する。

`get-file-buffer` *filename* [Function]

この関数はファイル *filename*を visit しているバッファーをリターンする。そのようなバッファーが存在しなければ `nil`をリターンする。引数 *filename*は文字列でなければならず、展開 (Section 25.9.4 [File Name Expansion], page 559 を参照) された後に、kill されていないすべてのバッファーが visit しているファイル名と比較される。バッファーの `buffer-file-name`は *filename*の展開形と正確にマッチしなければならないことに注意。この関数は同じファイルにたいする他の名前は認識しないだろう。

```
(get-file-buffer "buffers-ja.texi")
    ⇒ #<buffer buffers-ja.texi>
```

特殊な状況下では、複数のバッファーが同じファイル名を visit することがあり得る。そのような場合には、この関数はバッファーリスト内の最初に該当するバッファーをリターンする。

`find-buffer-visiting` *filename* **&optional** *predicate* [Function]

これは `get-file-buffer`と似ているが、そのファイルを違う名前で visit しているかもしれないすべてのバッファーをリターンする。つまりバッファーの `buffer-file-name`は *filename*の展開形式と正確にマッチする必要はなく、同じファイルを参照することだけが要求される。*predicate*が非 `nil`なら、それは *filename*を visit しているバッファーを 1 つの引数とする関数であること。そのバッファーにたいして *predicate*が非 `nil`をリターンした場合のみ適切なリターン値と判断される。リターンすべき適切なバッファーが見つからなければ、`find-buffer-visiting`は `nil`をリターンする。

`set-visited-file-name` *filename* **&optional** *no-query* [Command]
along-with-file

*filename*が非空文字列なら、この関数はカレントバッファーに visit されているファイルの名前を *filename*に変更する (バッファーがファイルを visit していなければ visit するファイルとして *filename*を与える)。そのバッファーにたいする次回の保存では、新たに指定されたファイルに保存されるだろう。

このコマンドは、たとえそのバッファーのコンテンツがその前に visit されていたファイルとマッチしていても、(Emacs が関知するかぎり) *filename*のコンテンツとはマッチしないのでバッファーが変更されている (modified) とマークする。これはその名前がすでに使用されていなければ、新たなファイル名に対応してバッファーをリネームする。

*filename*が `nil`か空文字列なら、それは “visit されているファイルがない” ことを意味する。この場合には `set-visited-file-name`はバッファーの変更フラグを変更することなく、そのバッファーがファイルを visit していないとマークする。

この関数は *filename*を visit しているバッファーがすでに存在する場合は、通常はユーザーに確認を求める。しかし *no-query*が非 `nil`ならこの質問を行わない。*filename*を visit している

バッファーがすでに存在し、かつユーザーが承認するか *no-query*が非 `nil`なら、この関数は中に数字が入った '`<...>`'を *filename*に追加して新たなバッファーの名前を一意にする。

*along-with-file*が非 `nil`なら、それは前に visit されていたファイルが *filename*にリネームされたと想定することを意味する。この場合、コマンドはバッファーの修正フラグを変更せず、そのバッファーの記録されている最終ファイル変更時刻を `visited-file-modtime`が報告する時刻 (Section 27.6 [Modification Time], page 597 を参照) で変更することもしない。*along-with-file*が `nil`なら、この関数は `visited-file-modtime`が 0 をリターンした後に、記録済みの最終ファイル変更時刻をクリアーする。

関数 `set-visited-file-name`が interactive に呼び出されたときはミニバッファー内で *filename*の入力を求める。

`list-buffers-directory` [Variable]

このバッファーローカル変数は visit しているファイル名をもたないバッファーにたいして、バッファーリスト中の visit しているファイル名を表示する場所に表示する文字列を指定する。Dired バッファーはこの変数を使用する。

27.5 バッファーの変更

Emacs は各バッファーにたいしてバッファーのテキストを変更したかどうかを記録するために、変更フラグ *(modified flag)* と呼ばれるフラグを管理しています。このフラグはバッファーのコンテンツを変更すると常に `t`にセットされ、バッファーを保存したとき `nil`にクリアーされます。したがってこのフラグは保存されていない変更があるかどうかを表します。フラグの値は通常はモードライン内 (Section 23.4.4 [Mode Line Variables], page 488 を参照) に表示され、保存 (Section 25.2 [Saving Buffers], page 534 を参照) と自動保存 (Section 26.2 [Auto-Saving], page 583 を参照) を制御します。

いくつかの Lisp プログラムは、このフラグを明示的にセットします。たとえば、関数 `set-visited-file-name`は、このフラグを `t`にセットします。なぜなら、たとえその前に visit していたファイルが変更されていなくても、テキストは新たに visit されたファイルとマッチしないからです。

バッファーのコンテンツを変更する関数は Chapter 32 [Text], page 777 で説明されています。

`buffer-modified-p` **&optional** *buffer* [Function]

この関数はバッファー *buffer*が最後にファイルから読み込まれたか、あるいは保存されてから変更されていれば `t`、それ以外では `nil`をリターンする。*buffer*が与えられなければカレントバッファーがテストされる。

`set-buffer-modified-p` *flag* [Function]

この関数は *flag*が非 `nil`ならカレントバッファーを変更済みとして、`nil`なら未変更としてマークする。

この関数を呼び出すことによる別の効果は、それがカレントバッファーのモードラインの無条件な再表示を引き起こすことである。実際のところ関数 `force-mode-line-update`は以下を行うことにより機能する:

```
(set-buffer-modified-p (buffer-modified-p))
```

`restore-buffer-modified-p` *flag* [Function]

`set-buffer-modified-p`と同様だがモードラインにたいする強制的な再表示を行わない点が異なる。

`not-modified` **&optional** *arg* [Command]
このコマンドはカレントバッファーが変更されておらず保存する必要がないとマークする。*arg* が非 `nil`なら変更されているとマークするので、次回の適切なタイミングでバッファーは保存されるだろう。interactive に呼び出された場合には、*arg*はプレフィックス引数。

この関数はエコーエリア内にメッセージをプリントするのでプログラム内で使用してはならない。かわりに `set-buffer-modified-p` (上述) を使用すること。

`buffer-modified-tick` **&optional** *buffer* [Function]
この関数は *buffer*の変更カウント (modification-count) をリターンする。これはバッファーが変更されるたびに増加されるカウンター。*buffer*が `nil` (または省略) ならカレントバッファーが使用される。

`buffer-chars-modified-tick` **&optional** *buffer* [Function]
この関数は *buffer*の文字変更に関わる変更カウントをリターンする。テキストプロパティを変更してもこのカウンターは変化しない。しかしそのバッファーにテキストが挿入または削除されるたびに、このカウンターは `buffer-modified-tick`によりリターンされるであろう値にリセットされる。`buffer-chars-modified-tick`を 2 回呼び出してリターンされる値を比較することにより、その呼び出しの間にバッファー内で文字変更があったかどうかを知ることができる。*buffer*が `nil` (または省略) ならカレントバッファーが使用される。

テキストプロパティの変更の等、バッファーのテキストを実際には変更しない方法でバッファーを変更することを要する場合があります。プログラムがフックやバッファー変更にたいするリアクションを何もトリガーせずにバッファーを変更する必要がある場合には、`with-silent-modifications` マクロを使用します。

`with-silent-modifications` *body...* [Macro]
バッファーを変更しないように装って *body*を実行する。これにはバッファーのファイルがロックされているかどうかのチェック (Section 25.5 [File Locks], page 539 を参照)、バッファーの変更フック (Section 32.32 [Change Hooks], page 850 を参照) 等が含まれる。*body*が持実際にバッファーテキストを変更する場合にはアンドゥするデータが破損するかもしれないことに注意。

27.6 バッファーの変更 Time

あるファイルを visit してそのバッファー内で変更を行い、その一方ではディスク上でファイル自身が変更されたとします。この時点でバッファーを保存するとファイル内の変更は上書きされるでしょう。これが正に望んでいる動作のときもありますが、通常は有用な情報が失われてしまいます。したがって Emacs はファイルを保存する前に、以下で説明する関数を使用してファイルの変更時刻をチェックします (ファイルの変更時刻を調べる方法は Section 25.6.4 [File Attributes], page 545 を参照)。

`verify-visited-file-modtime` **&optional** *buffer* [Function]
この関数は *buffer*(デフォルトはカレントバッファー) に visit されているファイルにたいして記録されている変更時刻と、オペレーティングシステムにより記録された実際の変更時刻を比較する。これら 2 つの時刻は Emacs がそのファイルを visit か保存して以降、他のプロセスにより書き込みがされていなければ等しくなるはずである。

この関数は実際の最終変更時刻と Emacs が記録した変更時刻が同じなら `t`、それ以外は `nil`をリターンする。そのバッファーが記録済みの最終変更時刻をもたない、すなわち `visited-file-modtime`が 0 をリターンするような場合にも `t`をリターンする。

これはたとえ `visited-file-modtime`が非 0 の値をリターンしたとしても、ファイルを visit していないバッファーにたいしては常に `t`をリターンする。たとえば Dired バッファーにたいして、この関数は常に `t`をリターンする。また存在せず、以前に存在したこともなかったファイルを visit するバッファーにたいして `t`をリターンするが、visit しているファイルが削除されたバッファーにたいしては `nil`をリターンする。

`clear-visited-file-modtime` [Function]

この関数はカレントバッファーにより visit されているファイルの最終変更時刻の記録をクリアーする。結果としてこのバッファーにを次回の保存ではファイルの変更時刻の食い違いは報告されなくなる。

この関数は `set-visited-file-name`、および変更済みファイルの上書きを防ぐための通常テストを行わない例外的な箇所で呼び出される。

`visited-file-modtime` [Function]

この関数はカレントバッファーにたいして記録された最終ファイル変更時刻を Lisp タイムスタンプ (Section 40.5 [Time of Day], page 1104 を参照) としてリターンする。

バッファーが最終変更時刻の記録をもたなければこの関数は 0 をリターンする。これが発生するのは、たとえばバッファーがファイルを visit していなかったり、`clear-visited-file-modtime`で最終変更時刻が明示的にクリアーされた場合。しかし `visited-file-modtime`は、いくつかの非ファイルバッファーにたいするタイムスタンプをリターンすることに注意。たとえばディレクトリーをリストする Dired バッファーでは、Dired が記録するそのディレクトリーの最終変更時刻がリターンされる。

バッファーが存在しないファイルを visit している場合には、この関数は -1 をリターンする。

`set-visited-file-modtime` **&optional** *time* [Function]

この関数はバッファーが visit しているファイルの最終変更時刻の記録を、*time*が非 `nil`なら *time*、それ以外は visit しているファイルの最終変更時刻に更新する。

*time*が `nil`や `visited-file-modtime`がリターンする整数フラグでなければ、それは Lisp の time 値であること (Section 40.5 [Time of Day], page 1104 を参照)。

この関数はバッファーが通常のようにファイルから読み取られたものでない場合や、ファイル自身が害のない既知の理由により変更されている場合に有用。

`ask-user-about-supersession-threat` *filename* [Function]

これは visit しているファイル *filename*がバッファーのテキストより新しいときにバッファーの変更を試みた後に、ユーザーに処理方法を尋ねるために使用する関数。Emacs はディスク上のファイルの変更時刻がバッファーを最後に保存した時刻新しいかどうか、バッファーのコンテンツが変更されているかによりこれを検知する。これはおそらく他のプログラムがファイルを変更したことを意味する。

この関数が正常にリターンするかどうかは、ユーザーの応答に依存する。関数はバッファーの変更が処理された場合は正常にリターンし、バッファーの変更が許可されなかった場合はデータ `(`*`filename`*`)`とともにエラー `file-supersession`をシグナルする。

この関数は適切なタイミングで Emacs により自動的に呼び出される。これは再定義することにより Emacs をカスタマイズ可能にするために存在する。標準的な定義はファイル `userlock.el`を参照のこと。

Section 25.5 [File Locks], page 539 のファイルロックのメカニズムも参照されたい。

27.7 読み取り専用のバッファー

あるバッファーが読み取り専用 *(read-only)* の場合には、たとえスクロールやナローイングによってファイルのコンテンツのビューを変更しても、そのコンテンツを変更することはできません。

読み取り専用バッファーは、2 つのタイプの状況において使用されます:

- 書き込み保護されたファイルを visit するバッファーは、通常は読み取り専用になる。

  ここでの目的はユーザーにたいしてそのファイルへの保存を意図したバッファーの編集が無益、または望ましくないかもしれないことを伝えることである。それにも関わらずバッファーのテキストの変更を望むユーザーは、`C-x C-q`で読み取り専用フラグをクリアーした後にこれを行うことができる。

- Dired や Rmail のようなモードは、通常の編集コマンドによるコンテンツの変更がおそらく間違いであるようなときにバッファーを読み取り専用にする。

  このようなモードのスペシャルコマンドは、`buffer-read-only`を (`let`によって)`nil`にバインドしたり、テキストを変更する箇所では `inhibit-read-only`を `t`にバインドする。

`buffer-read-only` [Variable]
: このバッファーローカル変数は、そのバッファーが読み取り専用かどうかを指定する。この変数が非`nil`ならそのバッファーは読み取り専用。しかしテキストプロパティ`inhibit-read-only`をもつ文字は依然として編集可能。Section 32.19.4 [Special Properties], page 820 を参照のこと。

`inhibit-read-only` [Variable]
: この変数が非`nil`なら、読み取り専用バッファー、およびその実際の値に依存して、一部もしくはすべての読み取り専用文字が変更されている。バッファー内の読み取り専用文字とはテキストプロパティ`read-only`が非`nil`の文字。テキストプロパティについての詳細は Section 32.19.4 [Special Properties], page 820 を参照のこと。

  `inhibit-read-only`が `t`なら、すべての `read-only`文字プロパティは効果がなくなる。`inhibit-read-only`がリストの場合には、`read-only`文字プロパティがリストのメンバーなら効果がなくなる (比較は `eq`で行われる)。

`read-only-mode` **&optional** *arg* [Command]
: これはバッファーローカルなマイナーモード Read Only モードにたいするモードコマンド。このモードが有効なときは、そのバッファーの `buffer-read-only`は非 `nil`。無効なときは、そのバッファーの `buffer-read-only`は `nil`。呼び出す際の慣習は、他のマイナーモードコマンドの慣習と同じ (Section 23.3.1 [Minor Mode Conventions], page 480 を参照)。

  このマイナーモードは他のマイナーモードとは異なり、主に `buffer-read-only`にたいするラッパーの役目を果たし、別個に `read-only-mode`変数は存在しない。Read Only モードが無効なときでも、`read-only`テキストプロパティが非`nil`の文字は読み取り専用のままである。一時的にすべての読み取り専用ステータスを無視するには上述の `inhibit-read-only`をバインドすること。

  Read Only モードを有効にする際、このモードコマンドはオプション `view-read-only`が非`nil`なら View モードも有効にする。Section “Miscellaneous Buffer Operations” in *The GNU Emacs Manual* を参照のこと。Read Only モードを無効にする際に、もしも View モードが有効なら View モードも無効にする。

`barf-if-buffer-read-only` **&optional** *position* [Function]
この関数はカレントバッファーが読み取り専用なら`buffer-read-only`エラーをシグナルする。*position* (デフォルトはポイント位置) のテキストのテキストプロパティ`inhibit-read-only`がセットされていればエラーは発生しないだろう。

カレントバッファーが読み取り専用の場合にエラーをシグナルする他の方法については、Section 21.2.1 [Using Interactive], page 376 を参照のこと。

27.8 バッファーリスト

バッファーリスト *(buffer list)* とは、すべての生きた (kill されていない) バッファーのリストです。このリスト内のバッファーの順序は主に、それぞれのバッファーがウィンドウに表示されたのがどれほど最近なのかにもとづきます。いくつかの関数、特に`other-buffer`はこの順序を使用します。ユーザーに表示されるバッファーリストもこの順序にしたがいます。

バッファーを作成するとそれはバッファーリストの最後に追加されバッファーを kill することによってそのリストから削除されます。ウィンドウに表示するためにバッファーが選択されたとき (Section 28.12 [Switching Buffers], page 637 を参照)、あるいはバッファーを表示するウィンドウが選択されたとき (Section 28.9 [Selecting Windows], page 631 を参照)、そのバッファーは常にこのリストの先頭に移動します。バッファーがバリー (以下の`bury-buffer`を参照) されたときは、このリストの最後に移動します。バッファーリストを直接操作するために利用できる Lisp プログラマー向けの関数は存在しません。

説明した基本バッファーリスト (fundamental buffer list) に加えて、Emacs はそれぞれのフレームにたいしてローカルバッファーリスト (local buffer list) を保守します。ローカルバッファーリストでは、そのフレーム内で表示されていた (または選択されたウィンドウの) バッファーが先頭になります (この順序はそのフレームのフレームパラメーター`buffer-list`に記録される。Section 29.4.3.5 [Buffer Parameters], page 721 を参照)。並び順は基本バッファーリストにならい、そのフレームでは表示されていないフレームは後になになります。。

`buffer-list` **&optional** *frame* [Function]
この関数はすべてのバッファーを含むバッファーリストをリターンする (名前がスペースで始まるバッファーも含む)。リストの要素はバッファーの名前ではなく実際のバッファー。

*frame*がフレームなら、*frame*のローカルバッファーリストをリターンする。*frame*が`nil`か省略された場合は、基本バッファーリストが使用される。その場合には、そのバッファーを表示するフレームがどれかとは無関係に、もっとも最近に表示または選択されたバッファーの順になる。

```
(buffer-list)
     ⇒ (#<buffer buffers-ja.texi>
         #<buffer  *Minibuf-1*> #<buffer buffer.c>
         #<buffer *Help*> #<buffer TAGS>)

;; ミニバッファーの名前が
;;   スペースで始まることに注意!
(mapcar #'buffer-name (buffer-list))
    ⇒ ("buffers-ja.texi" " *Minibuf-1*"
        "buffer.c" "*Help*" "TAGS")
```

`buffer-list`からリターンされるリストはそれ専用に構築されたリストであって、Emacs の内部的なデータ構造ではなく、それを変更してもバッファーの並び順に影響はありません。基本バッファーリスト内のバッファーの並び順を変更したい場合に簡単なのは以下の方法です:

```
(defun reorder-buffer-list (new-list)
  (while new-list
    (bury-buffer (car new-list))
    (setq new-list (cdr new-list))))
```

この方法により、バッファーを失ったり有効な生きたバッファー以外の何かを追加する危険を犯さずにリストに任意の並び順を指定できます。

特定のフレームのバッファーリストの並び順や値を変更するには、`modify-frame-parameters`でそのフレームの`buffer-list`パラメーターをセットしてください (Section 29.4.1 [Parameter Access], page 713 を参照)。

`other-buffer` **&optional** *buffer visible-ok frame* [Function]

この関数はバッファーリスト中で *buffer*以外の最初のバッファーをリターンする。これは通常は選択されたウィンドウ (フレーム *frame*、または選択されたフレーム (Section 29.10 [Input Focus], page 732 を参照) にもっとも最近表示された *buffer*以外のバッファーである。名前がスペースで始まるバッファーは考慮されない。

*buffer*が与えられない (または生きたバッファーでない) 場合には、`other-buffer`は選択されたフレームのローカルバッファーリスト内の最初のバッファーをリターンする (*frame*が非`nil`なら *frame*のローカルバッファーリスト内の最初のバッファーをリターンする)。

*frame*が非`nil`の`buffer-predicate`パラメーターをもつ場合には、どのバッファーを考慮すべきかを決定するために`other-buffer`はその述語を使用する。これはそれぞれのバッファーごとにその述語を一度呼び出して、値が`nil`ならそのバッファーは無視される。Section 29.4.3.5 [Buffer Parameters], page 721 を参照のこと。

*visible-ok*が`nil`なら`other-buffer`はやむを得ない場合を除き、任意の可視のフレーム上のウィンドウ内で可視のバッファーをリターンすることを避ける。*visible-ok*が非`nil`なら、バッファーがどこかで表示されているかどうかは問題にしない。

適切なバッファーが存在しなければ、バッファー`*scratch*`を (必要なら作成して) リターンする。

`last-buffer` **&optional** *buffer visible-ok frame* [Function]

この関数は *frame*のバッファーリスト内から *buffer*以外の最後のバッファーをリターンする。*frame*が省略または`nil`なら選択されたフレームのバッファーリストを使用する。

引数 *visible-ok*は上述した`other-buffer`と同様に扱われる。適切なバッファーを見つけられなければバッファー`*scratch*`がリターンされる。

`bury-buffer` **&optional** *buffer-or-name* [Command]

このコマンドはバッファーリスト内の他のバッファーの並び順を変更することなく、*buffer-or-name*をバッファーリストの最後に配置する。つまりこのバッファーは`other-buffer`がリターンする候補でもっとも期待度が低くなる。引数はバッファー自身かバッファーの名前を指定できる。

この関数は基本バッファーリストと同様に、それぞれのフレームの`buffer-list`パラメーターを操作する。したがってバリー (bury: 埋める、隠す) したバッファーは`(buffer-list` *frame*`)`と`(buffer-list)`の値の最後に置かれるだろう。さらにバッファーが選択されたウィンドウ

に表示されていれば、ウィンドウのバッファーリストの最後にバッファーを置くことも行う (Section 28.14 [Window History], page 659 を参照)。

*buffer-or-name*が `nil`または省略された場合には、カレントバッファーをバリーすることを意味する。加えてカレントバッファーが選択されたウィンドウに表示されていれば、そのウィンドウを削除するか他のバッファーを表示する。より正確には選択されたウィンドウが専用 (dedicated) のウィンドウ (see Section 28.15 [Dedicated Windows], page 661) であり、かつそのフレーム上に他のウィンドウが存在する場合には専用ウィンドウは削除される。それがフレーム上で唯一のウィンドウであり、かつそのフレームが端末上で唯一のフレームでなければ、そのフレームは `frame-auto-hide-function`で指定される関数を呼び出すことにより開放される (Section 28.16 [Quitting Windows], page 662 を参照)。それ以外の場合はに、他のバッファーをそのウィンドウ内に表示するために `switch-to-prev-buffer`を呼び出す (Section 28.14 [Window History], page 659 を参照)。*buffer-or-name*が他のウィンドウで表示されていれば、そのまま表示され続ける。

あるバッファーにたいして、それを表示するすべてのウィンドウでバッファーを置き換えるには `replace-buffer-in-windows`を使用する。Section 28.11 [Buffers and Windows], page 635 を参照のこと。

`unbury-buffer` [Command]

このコマンドは選択されたフレームのローカルバッファーリストの最後のバッファーに切り替える。より正確には選択されたウィンドウ内で、`last-buffer` (上記参照) がリターンするバッファーを表示するために関数 `switch-to-buffer`を呼び出す (Section 28.12 [Switching Buffers], page 637 を参照)。

`buffer-list-update-hook` [Variable]

これはバッファーリストが変更されたときに常に実行されるノーマルフック。(暗黙に) このフックを実行する関数は `get-buffer-create` (Section 27.9 [Creating Buffers], page 602 を参照)、`rename-buffer` (Section 27.3 [Buffer Names], page 592 を参照)、`kill-buffer` (Section 27.10 [Killing Buffers], page 603 を参照)、`bury-buffer` (上記参照)、`select-window` (Section 28.9 [Selecting Windows], page 631 を参照)。

このフックにが実行する関数は無限再帰を引き起こすので、`nil`の *norecord*引数による `select-window`の呼び出しや `with-temp-buffer`の呼び出しを無効にする。

27.9 バッファーの作成

このセクションではバッファーを作成する 2 つのプリミティブについて説明します。`get-buffer-create`は指定された名前の既存バッファーが見つからなければ作成します。`generate-new-buffer`は常に新たにバッファーを作成してそれに一意な名前を与えます。

バッファーを作成するために使用できる他の関数には `with-output-to-temp-buffer` (Section 39.8 [Temporary Displays], page 996 を参照)、および `create-file-buffer` (Section 25.1 [Visiting Files], page 530 を参照) が含まれます。サブプロセスの開始によってもバッファーを作成することができます (Chapter 38 [Processes], page 934 を参照)。

`get-buffer-create` *buffer-or-name* [Function]

この関数は *buffer-or-name*という名前のバッファーをリターンする。リターンされたバッファーはカレントにならない — この関数はカレントがどのバッファーであるかを変更しない。

*buffer-or-name*は文字列、または既存バッファーのいずれかでなければならない。これが文字列で、かつ既存の生きたバッファーの名前なら、`get-buffer-create`はそのバッファーをリ

ターンする。そのようなバッファーが存在しなければ、新たにバッファーを作成する。*buffer-or-name*が文字列ではなくバッファーなら、たとえそのバッファーが生きていなくても与えられたバッファーをリターンする。

```
(get-buffer-create "foo")
     ⇒ #<buffer foo>
```

新たに作成されたバッファーにたいするメジャーモードは Fundamental モードにセットされる (変数 `major-mode`のデフォルト値はより高いレベルで処理される。Section 23.2.2 [Auto Major Mode], page 468 を参照)。名前がスペースで始まる場合には、そのバッファーのアンドゥ情報の記録は初期状態では無効である (Section 32.9 [Undo], page 793 を参照)。

`generate-new-buffer` *name* [Function]

この関数は新たに空のバッファーを作成してリターンするが、それをカレントにはしない。バッファーの名前は関数 `generate-new-buffer-name`に *name*を渡すことにより生成される (Section 27.3 [Buffer Names], page 592 を参照)。つまり *name*という名前のバッファーが存在しなければ、それが新たなバッファーの名前になり、その名前が使用されていたら '`<n>`'という形式のサフィックスが *name*に追加される。ここで *n*は整数。

*name*が文字列でなければエラーがシグナルされる。

```
(generate-new-buffer "bar")
     ⇒ #<buffer bar>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<2>>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<3>>
```

新たなバッファーにたいするメジャーモードは Fundamental モードにセットされる。変数 `major-mode`のデフォルト値は、より高いレベルで処理される。Section 23.2.2 [Auto Major Mode], page 468 を参照のこと。

27.10 バッファーの kill

バッファーの *kill(Killing a buffer)* により、そのバッファーの名前は Emacs にとって未知の名前となり、そのバッファーが占めていたメモリースペースは他の用途に使用できるようになります。

バッファーに対応するバッファーオブジェクトは、それを参照するものがあれば kill されても存在し続けますが、それをカレントにしたり表示することができないように特別にマークされます。とはいえ kill されたバッファーの同一性は保たれるので、2 つの識別可能なバッファーを kill した場合には、たとえ両方死んだバッファーであっても `eq`による同一性は残ります。

あるウィンドウ内においてカレント、あるいは表示されているバッファーを kill した場合、Emacs はかわりに他の何らかのバッファーを自動的に選択または表示します。これはバッファーの kill によってカレントバッファーが変更されることを意味します。したがってバッファーを kill する際には、(kill されるバッファーがカレントを偶然知っていた場合を除き) カレントバッファーの変更に関しても事前に注意を払うべきです。Section 27.2 [Current Buffer], page 590 を参照してください。

1 つ以上のインダイレクトバッファーのベースとなるバッファーを kill した場合には、同様にインダイレクトバッファーも自動的に kill されます。

バッファーの `buffer-name`が `nil`の場合のみバッファーは kill されます。kill されていないバッファーは生きた *(live)* バッファーと呼ばれます。あるバッファーにたいして、そのバッファーが生きているか、または kill されているかを確認するには `buffer-live-p`を使用します (下記参照)。

`kill-buffer` **&optional** *buffer-or-name* [Command]
この関数はバッファー *buffer-or-name*を kill して、そのバッファーのメモリーを他の用途のために開放、またはオペレーティングシステムに返却する。*buffer-or-name*が `nil`または省略された場合にはカレントバッファーを kill する。

そのバッファーを `process-buffer`として所有するすべてのプロセスには、通常はプロセスを終了させるシグナル `SIGHUP` (hangup) が送信される。Section 38.8 [Signals to Processes], page 951 を参照のこと。

バッファーがファイルを visit していて、かつ保存されていない変更が含まれる場合には、`kill-buffer`はバッファーを kill する前にユーザーにたいして確認を求める。これは `kill-buffer`が interactive に呼び出されていなくても行われる。この確認要求を抑制するには `kill-buffer`の呼び出し前に、変更フラグ (modified flag) をクリアーすればよい。Section 27.5 [Buffer Modification], page 596 を参照のこと。

kill されるバッファーをカレントで表示しているすべてのバッファーをクリーンアップするために、この関数は `replace-buffer-in-windows`を呼び出す。

すでに死んでいるバッファーを kill しても効果はない。

この関数は実際にバッファーを kill すると `t`をリターンする。ユーザーが確認で拒否を選択、または *buffer-or-name*がすでに死んでいる場合には `nil`をリターンする。

```
(kill-buffer "foo.unchanged")
     ⇒ t
(kill-buffer "foo.changed")

---------- Buffer: Minibuffer ----------
Buffer foo.changed modified; kill anyway? (yes or no) yes
---------- Buffer: Minibuffer ----------

     ⇒ t
```

`kill-buffer-query-functions` [Variable]
保存されていない変更について確認を求める前に、`kill-buffer`はリスト `kill-buffer-query-functions`内の関数を出現順に引数なしで呼び出す。それらが呼び出される際には kill されるバッファーがカレントになる。この機能はこれらの関数がユーザーに確認を求めるというアイデアが元となっている。これらの関数のいずれかが `nil`をリターンしたら、`kill-buffer`はそのバッファーを殺さない。

`kill-buffer-hook` [Variable]
これは尋ねることになっている質問をすべて終えた後、実際にバッファーを kill する直前に `kill-buffer`により実行されるノーマルフック。この変数は永続的にローカルであり、メジャーモードの変更により、そのローカルバインディングはクリアーされない。

`buffer-offer-save` [User Option]
特定のバッファーにおいてこの変数が非 `nil`なら、あたかもファイルを visit するバッファーにたいして提案するときのように、バッファーの保存を提案するように `save-buffers-kill-emacs`に指示する。2 つ目のオプション引数を `t`にセットして `save-some-buffers`を呼び出せばバッファーの保存も提案する。最後にこの変数をシンボル `always`にセットすると、`save-buffers-kill-emacs`と `save-some-buffers`は常に保存を提案する。[Definition of save-some-buffers], page 534 を参照のこと。何らかの理由により変数 `buffer-offer-save`がセットされると自動的にバッファーローカルになる。Section 12.11 [Buffer-Local Variables], page 190 を参照のこと。

`buffer-save-without-query` [Variable]
特定のバッファーにおいてこの変数が非 `nil`なら、`save-buffers-kill-emacs`と`save-some-buffers`は、(バッファーが変更されていれば) ユーザーに確認を求めることなくそのバッファーを保存する。何らかの理由によりこの変数をセットする際には自動的にバッファーローカルになる。

`buffer-live-p` *object* [Function]
この関数は *object*が生きたバッファー (kill されていないバッファー) なら`t`、それ以外は`nil`をリターンする。

27.11 インダイレクトバッファー

インダイレクトバッファー *(indirect buffer:* 間接バッファー*)* とは、ベースバッファー *(base buffer)* と呼ばれる他のバッファーとテキストを共有します。いくつかの点においてインダイレクトバッファーはファイル間でのシンボリックリンクに類似しています。ベースバッファー自身はインダイレクトバッファーではない可能性があります。

インダイレクトバッファーのテキストは、常にベースバッファーのテキストと同一です。編集により一方が変更されると、それは即座に他方のバッファーから可視になります。これには文字自体に加えてテキストプロパティも同様に含まれます。

他のすべての観点において、インダイレクトバッファーとそのベースバッファーは完全に別物です。それらは別の名前、独自のポイント値、ナローイング、マーカー、オーバーレイ、メジャーモード、バッファーローカルな変数バインディングをもちます (ただしどちらかのバッファーでのテキストの挿入や削除を行うと両方のバッファーでマーカーとオーバーレイが再配置される)。

インダイレクトバッファーはファイルを visit できませんがベースバッファーには可能です。インダイレクトバッファーの保存を試みると、実際にはベースバッファーが保存されます。

インダイレクトバッファーを kill してもベースバッファーに影響はありません。ベースバッファーを kill するとインダイレクトバッファーは kill されて再びカレントバッファーにすることはできません。

`make-indirect-buffer` *base-buffer name* **&optional** *clone* [Command]
これはベースバッファーが *base-buffer*であるような、*name*という名前のインダイレクトバッファーを作成してリターンする。引数 *base-buffer*は生きたバッファー、または既存バッファーの名前 (文字列) を指定できる。*name*が既存バッファーの名前ならエラーがシグナルされる。

*clone*が非`nil`ならインダイレクトバッファーは最初は *base-buffer*のメジャーモード、マイナーモード、バッファーローカル変数等の状態を共有する。*clone*が省略または`nil`なら、インダイレクトバッファーの情報は新たなバッファーにたいするデフォルト状態にセットされる。

*base-buffer*がインダイレクトバッファーなら、新たなバッファーのベースとしてそれのベースバッファーが使用される。さらに *clone*が非`nil`なら、初期状態は *base-buffer*ではなく実際のベースバッファーからコピーされる。

`clone-indirect-buffer` *newname display-flag* **&optional** *norecord* [Command]
この関数はカレントバッファーのベースバッファーを共有するインダイレクトバッファーを新たに作成して、カレントバッファーの残りの属性をコピーしてリターンする (カレントバッファーがインダイレクトバッファーでなければそれがベースバッファーとして使用される)。

*display-flag*が非`nil` (インタラクティブな呼び出しでは常に非`nil`) なら、それは`pop-to-buffer`を呼び出すことにより新しいバッファーを表示することを意味する。*norecord*が非`nil`なら、それは新しいバッファーをバッファーリストの先頭に置かないことを意味する。

`buffer-base-buffer` **&optional** *buffer* [Function]
この関数は *buffer* (デフォルトはカレントバッファー) のベースバッファーをリターンする。*buffer*がインダイレクトバッファーでなければ値は `nil`、それ以外では値は他のバッファーとなり、そのバッファーがインダイレクトバッファーであることは決してない。

27.12 2 つのバッファー間でのテキストの交換

特別なモードでは、ユーザーが同一のバッファーから複数の非常に異なったテキストにアクセスできるようにしなければならない場合があります。たとえばバッファーのテキストのサマリーを表示して、ユーザーがそのテキストにアクセスできるようにする場合です。

これは、(ユーザーがテキストを編集した際には同期を保つ) 複数バッファーや、ナローイング (Section 30.4 [Narrowing], page 766 を参照) により実装することができるかもしれません。しかしこれらの候補案はときに退屈になりがちであり、特にそれぞれのテキストタイプが正しい表示と編集コマンドを提供するために高価なバッファーグローバル操作を要求する場合には、飛び抜けて高価になる場合があります。

Emacs はそのようなモードにたいして別の機能を提供します。`buffer-swap-text`を使用すれば、2 つのバッファー間でバッファーテキストを素早く交換することができます。この関数はテキストの移動は行わずに異なるテキスト塊 (text chunk) をポイントするように、バッファーオブジェクトの内部的なデータ構造だけを変更するため非常に高速です。これを使用することにより、2 つ以上のバッファーグループから個々のバッファーのコンテンツすべてを併せもつような、単一の仮想バッファー (virtual buffer) が実在するように見せかけることができます。

`buffer-swap-text` *buffer* [Function]
この関数はカレントバッファーのテキストと、引数 *buffer*のテキストを交換する。2 つのバッファーのいずれか一方がインダイレクトバッファー (Section 27.11 [Indirect Buffers], page 605 を参照)、またはインダイレクトバッファーのベースバッファーの場合はエラーをシグナルする。

バッファーテキストに関連するすべてのバッファープロパティ、つまりポイントとマークの位置、すべてのマーカーとオーバーレイ、テキストプロパティ、アンドゥリスト、`enable-multibyte-characters`フラグの値 (Section 33.1 [Text Representations], page 852 を参照) 等も同様に交換される。

警告: この関数を `save-excursion`内部で呼び出すと、位置とバッファーを保存するために`save-excursion`が使用するマーカーも同様に交換されるので、そのフォームを抜ける際にはカレントバッファーは *buffer*にセットされるだろう。

ファイルを visit しているバッファーに `buffer-swap-text`を使用する場合には、交換されたテキストではなくそのバッファーの元のテキストを保存するようにフックをセットアップするべきです。`write-region-annotate-functions`は正にこの目的のために機能します。そのバッファーの`buffer-saved-size`を、おそらく交換されたテキストにたいする変更が自動保存に干渉しないであろう、−2 にセットするべきです。

27.13 バッファーのギャップ

Emacs のバッファーは挿入と削除を高速にするために不可視のギャップ *(gap)* を使用して実装されています。挿入はギャップ部分を充填、削除はギャップを追加することにより機能します。もちろんこれは最初にギャップを挿入や削除の部位 (locus) に移動しなければならないことを意味します。Emacs はユーザーが挿入か削除を試みたときだけギャップを移動します。大きなバッファー内の遠く離れた位置で編集した後に、他の箇所での最初の編集コマンドに無視できない遅延が発生する場合があるのはこれが理由です。

このメカニズムは暗黙に機能するものであり、Lisp コードはギャップのカレント位置に影響されるべきでは決してありませんが、以下の関数はギャップ状態に関する情報の取得に利用できます。

`gap-position` [Function]
この関数はカレントバッファー内のギャップのカレント位置をリターンする。

`gap-size` [Function]
この関数はカレントバッファー内のギャップのサイズをリターンする。

28 ウィンドウ

このチャプターではEmacsのウィンドウに関連する関数と変数について説明します。Emacsが利用可能なスクリーン領域にウィンドウが割り当てられる方法についてはChapter 29 [Frames], page 696 を参照してください。ウィンドウ内にテキストが表示される方法についての情報はChapter 39 [Display], page 981 を参照してください。

28.1 Emacs ウィンドウの基本概念

ウィンドウ *(window)* とは任意のバッファーを表示するために使用されるスクリーン領域です。Emacs Lispではウィンドウはスペシャル Lisp オブジェクトとして表現されます。

ウィンドウはフレームへとグループ化されます (Chapter 29 [Frames], page 696 を参照)。それぞれのフレームは最低でも 1 つのウィンドウを含みます。ユーザーは複数のバッファーを一度に閲覧するために、それを複数のオーバーラップしないウィンドウに分割することができます。Lisp プログラムはさまざまな目的にたいして複数のウィンドウを使用できます。たとえばRmailでは1つのウィンドウでメッセージタイトル、もう一方のウィンドウで選択したメッセージのコンテンツを閲覧できます。

Emacsはグラフィカルなデスクトップ環境やX Window Systemのようなウィンドウシステムとは異なる意味で“ウィンドウ (window)” という単語を使用します。EmacsがX上で実行されているときはXのグラフィカルなXウィンドウは、Emacsでの (1つ以上のEmacsウィンドウを含んだ) フレームになります。Emacsがテキスト端末上で実行されているときはフレームが端末スクリーン全体を占有します。

Xのウィンドウとは異なり、Emacsのウィンドウはタイル表示 *(tiled)* されるので、フレームの領域内でオーバーラップされることは決してありません。あるウィンドウが作成、リサイズ、削除されるとき変更されたウィンドウスペースの変更は各ウィンドウの調整により取得・譲与されるので、そのフレームの総領域に変化はありません。

`windowp` *object* [Function]
: この関数は *object*がウィンドウ (バッファーの表示有無に関わらず) なら `t`、それ以外は `nil`をリターンする。

生きたウィンドウ *(live window)* とは、あるフレーム内で実際にバッファーを表示しているウィンドウのことです。

`window-live-p` *object* [Function]
: この関数は *object*が生きたウィンドウなら `t`、それ以外は `nil`をリターンする。生きたウィンドウとはバッファーを表示するウィンドウのこと。

各フレーム内のウィンドウはウィンドウツリー *(window tree)* 内へと組織化されます。Section 28.2 [Windows and Frames], page 609 を参照してください。それぞれのウィンドウツリーのリーフノード (leaf nodes) は、実際にバッファーを表示している生きたウィンドウです。ウィンドウツリーの内部ノード (internal node) は内部ウィンドウ *(internal windows)* と呼ばれ、これらは生きたウィンドウではありません。

有効なウィンドウ *(valid window)* とは、生きたウィンドウか内部ウィンドウのいずれかです。有効なウィンドウにたいしては、それを削除 *(delete)*、すなわちそのウィンドウのフレームから削除することができます (Section 28.7 [Deleting Windows], page 625 を参照)。その場合、それは有効なウィンドウではなくなりますが、それを表すLispオブジェクトは依然として他のLispオブジェクトから参照されたままかもしれません。削除されたウィンドウは保存されたウィンドウ設定 (window

configuration) をリストアすることにより再び有効にすることができます (Section 28.26 [Window Configurations], page 686 を参照)。

`window-valid-p`により、削除されたウィンドウから有効なウィンドウを区別できます。

`window-valid-p` *object* [Function]
この関数は *object*が生きたウィンドウかウィンドウツリー内の内部ウィンドウなら `t`をリターンする。それ以外 (*object*が削除されたウィンドウの場合も含む) は `nil`をリターンする。

それぞれのフレーム内において、常にただ 1 つの Emacs ウィンドウがそのフレームで選択されている *(selected within the frame)* として指定されます。選択されたフレームにたいしてそのウィンドウは選択されたウィンドウ *(selected window)* と呼ばれます。選択されたウィンドウは編集のほとんどが行われるウィンドウであり、選択されたウィンドウに表示されるカーソルがあるウィンドウです (Section 29.4.3.9 [Cursor Parameters], page 725 を参照)。テキストの挿入や削除を行うキーボード入力もそのウィンドウにたいして行われます。選択されたウィンドウのバッファーは、通常は `set-buffer`が使用された場合を除いてカレントバッファーでもあります (Section 27.2 [Current Buffer], page 590 を参照)。選択されていないフレームでは、そのフレームが選択されたときはフレームで選択されていたウィンドウが選択されたウィンドウになります。Section 28.9 [Selecting Windows], page 631 を参照してください。

`selected-window` [Function]
この関数は選択されたウィンドウをリターンする (これは常に生きたウィンドウ)。

たとえば Follow モード (Section “Follow Mode” in `emacs`を参照) の管理下では、あるウィンドウが単独で表示可能な部分より大きい部分をそのウィンドウにまとめて表示するように、複数のウィンドウが集合かつ協調してバッファーを表示することがあります。そのようなウィンドウグループ *(window group)* を 1 つのエンティティーとしてとらえると便利なことがよくあります。`window-group-start` (Section 28.20 [Window Start and End], page 671 を参照) のようないくつかの関数では、グループ全体としてウィンドウの 1 つを引数に与えることにより、これを行うことができます。

`selected-window-group` [Function]
選択されたウィンドウがウィンドウグループのメンバーなら、この関数はそのバッファーの最前箇所を表示するウィンドウが先頭になる順序で、グループ内のウィンドウのリストをリターンする。それ以外なら、この関数は選択されたウィンドウだけを含むリストをリターンする。

バッファーローカル変数`selected-window-group-function`が関数にセットされているときは、選択されたウィンドウはグループの一部とみなされる。この場合には、`selected-window-group`はその関数を引数なしで呼び出し、その結果をリターンする (これはそのグループ内のウィンドウのリストであること)。

28.2 ウィンドウとフレーム

ウィンドウはそれぞれ厳密に 1 つのフレームに属します (Chapter 29 [Frames], page 696 を参照)。

`window-frame` **&optional** *window* [Function]
この関数はウィンドウ *window*が属するフレームをリターンする。*window*が `nil`の場合のデフォルトは選択されたウィンドウ。

`window-list` **&optional** *frame minibuffer window* [Function]
この関数はフレーム *frame*に属する生きたウィンドウのリストをリターンする。*frame*が省略または `nil`の場合のデフォルトは選択されたフレーム。

オプション引数 *minibuffer*はリターンされるリストにミニバッファーウィンドウを含めるべきかどうかを指定する。*minibuffer*が `t`ならミニバッファーウィンドウが含まれ、*minibuffer*が `nil`または省略された場合にはミニバッファーウィンドウがアクティブのときだけ含まれる。*minibuffer*が `nil`と `t`以外ならミニバッファーウィンドウは含まれない。

オプション引数 *window*が非 `nil`なら、それは指定されたフレーム上の生きたウィンドウであること。その場合には *window*がリターンされるリストの最初の要素になる。*window*が省略または `nil`なら、そのフレームの選択されたウィンドウが最初の要素になる。

同一フレーム内のウィンドウは、リーフノード (leaf nodes) が生きたウィンドウであるようなウィンドウツリー *(window tree)* 内に組織化されます。ウィンドウツリーの内部ノード (internal nodes) は生きたウィンドウではありません。これらのウィンドウは生きたウィンドウ間の関係を組織化するという目的のために存在します。ウィンドウツリーのルートノード (root node) はルートウィンドウ *(root window)* と呼ばれます。ルートノードは生きたウィンドウ (そのフレームにウィンドウが 1 つだけの場合)、または内部ウィンドウのいずれかです。

フレーム内で唯一ではないミニバッファーウィンドウ (Section 20.11 [Minibuffer Windows], page 371 を参照) は親ウィンドウをもたないので、厳密に言えばフレームのウィンドウツリーの一部ではありません。それでもフレームのルートウィンドウの兄弟ウィンドウなので、`window-next-sibling`を通じて到達することができます。さらにこのセクションの最後に説明する関数 `window-tree`は実際のウィンドウツリーと共にミニバッファーウィンドウをリスとします。

`frame-root-window` **&optional** *frame-or-window* [Function]

この関数は *frame-or-window*にたいするルートウィンドウをリターンする。引数 *frame-or-window*はウィンドウかフレームのいずれかであること。これが省略または `nil`の場合のデフォルトは選択されたフレーム。*frame-or-window*がウィンドウなら、リターン値はそのウィンドウのフレームのルートウィンドウ。

ウィンドウが分割 (split) されているときは、以前は 1 つだった 2 つの生きたウィンドウが存在します。これらのうちの一方は、元のウィンドウと同じ Lisp ウィンドウオブジェクトとして表され、もう一方は新たに作成された Lisp ウィンドウオブジェクトとして表されます。これらの生きたウィンドウはいずれも単一の内部ウィンドウの子ウィンドウ *(child windows)* として、ウィンドウツリーのリーフノードになります。もし必要なら Emacs はこの内部ウィンドウを自動的に作成します。この内部ウィンドウは親ウィンドウ *(parent window)* とも呼ばれ、ウィンドウツリー内の適切な位置に配置されます。同じ親を共有するウィンドウセットは兄弟 *(sibling)* と呼ばれます。

`window-parent` **&optional** *window* [Function]

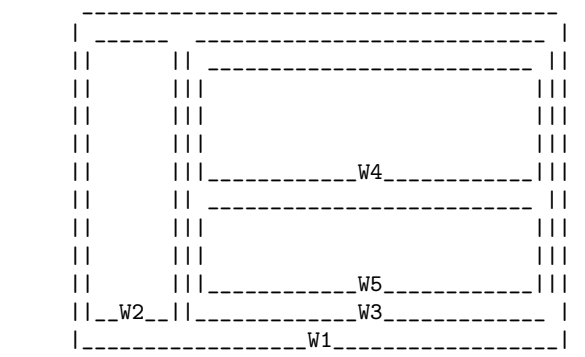
この関数は *window*の親ウィンドウ (parent window) をリターンする。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ。*window*が親をもたない場合 (ミニバッファーウィンドウやそのフレームのルートウィンドウ) にはリターン値は `nil`。

内部ウィンドウはそれぞれ、常に最低でも 2 つの子ウィンドウをもちます。ウィンドウ削除によりこの数値が 1 になると Emacs は自動的に内部ウィンドウを削除して、その残った単一の子ウィンドウがウィンドウツリー内のその位置に配置されます。

子ウィンドウはそれぞれ生きたウィンドウ、または (次に自身の子ウィンドウをもつであろう) 内部ウィンドウのいずれかです。したがって各内部ウィンドウは、最終的にはその内部ウィンドウの子孫であるような生きたウィンドウにより占有される領域を結合した、特定の矩形スクリーン領域 *(screen area)* を占有すると考えることができます。

内部ウィンドウそれぞれにたいして、近接する子たちのスクリーン領域は垂直 (vertically) か水平 (horizontally) のいずれかにより整列されます (両方で整列されることはない)。子ウィンドウが他の

子ウィンドウと上下に整列される場合、それらは垂直コンビネーション *(vertical combination)*、左右に整列される場合は水平コンビネーション *(horizontal combination)* を形成すると表現されます。以下の例で考えてみましょう:



このフレームのルートウィンドウは内部ウィンドウ *W1*です。これの子ウィンドウは、生きたウィンドウ *W2*と内部ウィンドウ *W3*からなる水平コンビネーションを形成します。*W3*の子ウィンドウは、生きたウィンドウ *W4*と *W5*からなる垂直コンビネーションを形成します。したがって、このウィンドウツリー内の生きたウィンドウは *W2*、*W4*、および *W5*です。

以下の関数は内部ウィンドウの子ウィンドウ、および子ウィンドウの兄弟を取得するために使用できます。

`window-top-child` **&optional** *window* [Function]
この関数は内部ウィンドウ *window*の子ウィンドウが垂直コンビネーションを形成する場合には、*window*の一番上の子ウィンドウをリターンする。他のタイプのウィンドウにたいするリターン値は `nil`。

`window-left-child` **&optional** *window* [Function]
この関数は内部ウィンドウ *window*の子ウィンドウが水平コンビネーションを形成する場合には、*window*の一番左の子ウィンドウをリターンする。他のタイプのウィンドウにたいするリターン値は `nil`。

`window-child` *window* [Function]
この関数は内部ウィンドウ *window*の最初の子ウィンドウをリターンする。これは垂直コンビネーションにたいしては一番上、水平コンビネーションにたいしては一番左の子ウィンドウ。*window*が生きたウィンドウならリターン値は `nil`。

`window-combined-p` **&optional** *window horizontal* [Function]
この関数は *window*が垂直コンビネーションの一部である場合のみ非 `nil`をリターンする。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ。

オプション引数 *horizontal*が非 `nil`なら、*window*が水平コンビネーションの一部である場合のみ非 `nil`をリターンすることを意味する。

`window-next-sibling` **&optional** *window* [Function]
この関数はウィンドウ *window*の次の兄弟をリターンする。省略または `nil`なら、*window*のデフォルトは選択されたウィンドウ。*window*がその親の最後の子ならリターン値は `nil`。

`window-prev-sibling` **&optional** *window* [Function]
この関数はウィンドウ *window*の前の兄弟をリターンする。省略または `nil`なら、*window*のデフォルトは選択されたウィンドウ。*window*がその親の最初の子ならリターン値は `nil`。

関数 `window-next-sibling`と `window-prev-sibling`を、ウィンドウのサイクル順 (Section 28.10 [Cyclic Window Ordering], page 633 を参照) で次や前のウィンドウをリターンする関数`next-window`と`previous-window`と混同しないでください。

以下の関数はフレーム内のウィンドウの配置に有用です。

`frame-first-window` **&optional** *frame-or-window* [Function]

この関数は *frame-or-window*により指定されたフレームの左上隅の生きたウィンドウをリターンする。引数 *frame-or-window*はウィンドウか生きたフレームを指定しなければならず、デフォルトは選択されたフレーム。*frame-or-window*がウィンドウを指定する場合には、この関数はそのウィンドウのフレームの最初のウィンドウをリターンする。前の例のフレームが`(frame-first-window)`で選択されたとすると *W2*がリターンされる。

`window-at-side-p` **&optional** *window side* [Function]

この関数は *window*のフレーム内で *window*が *side*に配置されていれば`t`をリターンする。引数 *window*は有効なウィンドウでなければならず、デフォルトは選択されたウィンドウ。引数 *side*はシンボル `left`、`top`、`right`、`bottom`のいずれか。デフォルト値の `nil`は `bottom`として扱われる。

この関数はミニバッファーウィンドウを無視することに注意 (Section 20.11 [Minibuffer Windows], page 371 を参照)。したがって *side*が `bottom`なら、*window*の右下にミニバッファーウィンドウがある際にも `t`をリターンするかもしれない。

`window-in-direction` *direction* **&optional** *window ignore sign wrap mini* [Function]

この関数はウィンドウ *window*内の位置 `window-point`から、方向 *direction*にあるもっとも近い生きたウィンドウをリターンする。引数 *direction*は `above`、`below`、`left`、`right`のいずれかでなければならない。オプション引数 *window*は生きたウィンドウでなければならず、デフォルトは選択されたウィンドウ。

この関数はパラメーター `no-other-window`が非 `nil`のウィンドウをリターンしない (Section 28.27 [Window Parameters], page 688 を参照)。もっとも近いウィンドウの`no-other-window`パラメーターが非`nil`なら、この関数は指定された方向で `no-other-window`パラメーターが`nil`であるような他のウィンドウを探す。オプション引数 *ignore*が非 `nil`なら、たとえ`no-other-window`パラメーターが非 `nil`のウィンドウでもリターンされるだろう。

オプション引数 *sign*が負の数値なら、それは参照位置として `window-point`のかわりに *window*の右端、または下端を使用することを意味する。*sign*が正の数値なら、それは参照位置として *window*の左端か上端を使用することを意味する。

オプション引数 *wrap*が非 `nil`なら、それはフレームのボーダー (borders: 枠線) を *direction*がラップアラウンド (wrap around: 最後に達したら最初に戻る) することを意味する。たとえば *window*はフレームの最上にあり *direction*が `above`なら、そのフレームのミニバッファーウィンドウがアクティブでそれがフレームの他のウィンドウの最下にある場合には、この関数は通常はミニバッファーウィンドウをリターンする。

オプション引数 *mini*が `nil`なら、それはミニバッファーがカレントでアクティブな場合のみミニバッファーウィンドウをリターンすることを意味する。*mini*が非 `nil`なら、たとえ非アクティブなときでも、この関数はミニバッファーウィンドウをリターンする。しかし *wrap*が非`nil`なら、常に *mini*が `nil`であるかのように振る舞う。

適切なウィンドウが見つからなければ、この関数は `nil`をリターンする。

この関数を *direction*にウィンドウが存在しないかどうかをチェックするために使用しないこと。それを行うには上述の `window-at-side-p`を呼び出すほうが効果的である。

以下の関数によりフレームのウィンドウツリー全体を取得できます:

`window-tree` **&optional** *frame* [Function]

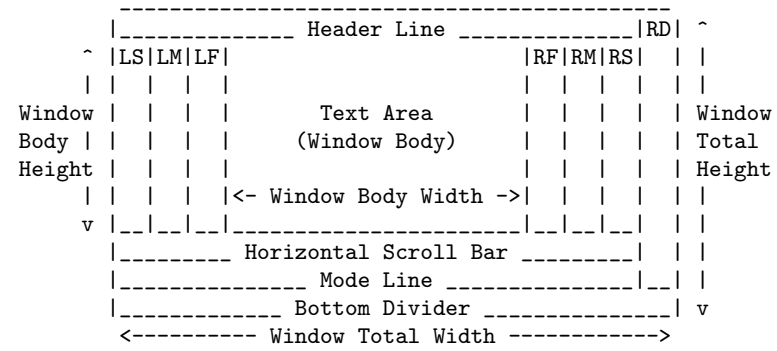
この関数はフレーム *frame*にたいするウィンドウツリーを表すリストをリターンする。*frame* が省略 `nil`の場合のデフォルトは選択されたフレーム。

リターン値は `(root mini)`という形式のリスト。ここで *root*はそのフレームのウィンドウツリーのルートウィンドウ、*mini*はそのフレームのミニバッファーウィンドウを表す。

ルートウィンドウが生きていれば *root*はそのウィンドウ自身、それ以外なら *root*はリスト `(dir edges w1 w2 ...)`。ここで *dir*は水平コンビネーションなら `nil`、垂直コンビネーションなら `t`となり、*edges*はそのコンビネーションのサイズと位置を与え、残りの要素は子ウィンドウである。子ウィンドウはそれぞれ、同じようにウィンドウオブジェクト (生きたウィンドウにたいして)、または上記フォーマットと同じ形式のリスト (内部ウィンドウにたいして) かもしれない。*edges*要素は `window-edges`がリターンする値のようなリスト `(left top right bottom)` (Section 28.24 [Coordinates and Windows], page 681 を参照)。

28.3 ウィンドウのサイズ

以下の図は生きたウィンドウの構造を示しています:



ウィンドウの中央はテキストエリア *(text area:* テキスト領域*)*、またはボディー *(body:* 本体、本文*)* と呼ばれるバッファーテキストが表示される場所です。テキストエリアは一連のオプションエリアで囲まれている可能性があります。左右には内側から外側に向かって図中に LF と RF で示される左右のフリンジ (Section 39.13 [Fringes], page 1034 を参照)、LM と RM で示される左右のマージン (Section 39.16.5 [Display Margins], page 1047 を参照)、そして LS と RS はスクロールバー (Section 39.14 [Scroll Bars], page 1039 を参照) で、これは常に表示されるのはいずれか一方だけです。さらに RD で示されるのが右ディバイダー (Section 39.15 [Window Dividers], page 1042 を参照) です。ウィンドウ上端にはヘッダーライン (Section 23.4.7 [Header Lines], page 493 を参照)、ウィンドウ下端には水平スクロールバー (Section 39.14 [Scroll Bars], page 1039 を参照)、モードライン (Section 23.4 [Mode Line Format], page 485 を参照)、下端ディバイダー (Section 39.15 [Window Dividers], page 1042 を参照) があります。

Emacs はウィンドウの高さと幅を求めるためのさまざまな関数を提供します。これらの関数がリターンする値の多くはピクセル単位、または行単位と列単位のいずれかにより指定できます。グラフィカルなディスプレイでは後者は実際には `frame-char-height`と `frame-char-width` (Section 29.3.2 [Frame Font], page 707 を参照) によりリターンされる、そのフレームのデフォルトフォントが指定するデフォルト文字の高さと幅に対応します。したがってあるウィンドウが異なるフォントやサイズでテキストを表示していると、そのウィンドウにたいして報告される行高さと列幅は、実際にウィンドウ内で表示されるテキスト行数と列数とは異なるかもしれません。

トータル高さ *(total height)* とは、そのウィンドウのボディーを構成する行数、ヘッダーライン、水平スクロールバー、モードライン、(もしあれば) 下端ディバイダーです。

`window-total-height` **&optional** *window round* [Function]
この関数はウィンドウ *window*のトータル高さを行数でリターンする。*window*が省略 `nil`の場合のデフォルトは選択されたウィンドウ。*window*が内部ウィンドウなら、リターン値はそのウィンドウの子孫となるウィンドウにより占有されるトータル高さになる。

ウィンドウのピクセル高さがそのウィンドウがあるフレームのデフォルト文字高さの整数倍でなければ、そのウィンドウが占有する行数が内部で丸められる。これはそのウィンドウが親ウィンドウの場合には、すべての子ウィンドウのトータル高さの合計が、親ウィンドウのトータル高さと内部的に等しくなるような方法により行われる。これはたとえ 2 つのウィンドウのピクセル高さが等しくでも、内部的なトータル高さは 1 行分異なるかもしれないことを意味する。さらにこれはそのウィンドウが垂直コンビネーションされていて、かつ次の兄弟をもつ場合には、その兄弟の上端行は、このウィンドウの上端行とトータル高さから計算されるかもしれないことも意味する (Section 28.24 [Coordinates and Windows], page 681 を参照)。

オプション引数 *round*が `ceiling`なら、この関数は *window*のピクセル高さをそのフレームの文字高さで除した数より大であるような最小の整数、`floor`なら除した数より小であるような最大の整数、それ以外の *round*にたいしては *windows*のトータル高さの内部値をリターンする。

トータル幅 *(total width)* とはそのウィンドウのボディーを構成する列数、マージン、フリンジ、スクロールバー、(もしあれば) 右ディバイダーです。

`window-total-width` **&optional** *window round* [Function]
この関数はウィンドウ *window*のトータル幅を列でリターンする。*window*が省略 `nil`の場合のデフォルトは選択されたウィンドウ。*window*が内部ウィンドウならリターン値はその子孫のウィンドウが占有するトータル幅になる。

ウィンドウのピクセル幅がそのウィンドウがあるフレームのデフォルト文字幅の整数倍でなければ、そのウィンドウが占有する列数が内部で丸められる。これはそのウィンドウが親ウィンドウの場合には、すべての子ウィンドウのトータル幅の合計が親ウィンドウのトータル幅と内部的に等しくなるような方法により行われる。これはたとえ 2 つのウィンドウのピクセル幅が等しくでも、内部的なトータル幅は 1 列分異なるかもしれないことを意味する。さらにこれはそのウィンドウが水平コンビネーションされていて、かつ次の兄弟をもつ場合、その兄弟の左端行はこのウィンドウの左端行とトータル幅から計算されるかもしれないことも意味する (Section 28.24 [Coordinates and Windows], page 681 を参照)。オプション引数 *round*は `window-total-height`の場合と同様に振る舞う。

`window-total-size` **&optional** *window horizontal round* [Function]
この関数はウィンドウ *window*のトータル高さを行数、またはトータル幅を列数でリターンする。*horizontal*が省略または `nil`なら *window*にたいして `window-total-height`を呼び出すのと等価、それ以外では *window*にたいして `window-total-width`を呼び出すのと等価である。オプション引数 *round*は `window-total-height`の場合と同様に振る舞う。

以下の 2 つの関数はウィンドウのトータルサイズをピクセル単位で取得するために使用できます。

`window-pixel-height` **&optional** *window* [Function]
この関数はウィンドウ *window*のトータル高さをピクセル単位でリターンする。*window*は有効なウィンドウでなければならずデフォルトは選択されたウィンドウ。

リターン値には、(もしあれば) モードライン、ヘッダーライン、水平スクロールバー、下端ディバイダーが含まれる。*window*が内部ウィンドウなら、そのピクセル高さは子ウィンドウたちによりスパンされるスクリーン領域のピクセル高さになる。

`window-pixel-width` **&optional** *window* [Function]
この関数はウィンドウ *window*の幅をピクセル単位でリターンする。*window*は有効なウィンドウでなければならずデフォルトは選択されたウィンドウ。

リターン値にはフリンジ、*window*のマージン、同様に *window*に属する垂直ディバイダーとスクロールバーが含まれる。*window*が内部ウィンドウなら、そのピクセル幅は子ウィンドウたちにより占有されるスクリーン領域の幅になる。

以下の関数は与えられたウィンドウに隣接するウィンドウがあるかどうかを判断するために使用できます。

`window-full-height-p` **&optional** *window* [Function]
この関数はフレーム内で *window*の上下に他のウィンドウがなければ非 `nil`をリターンする。より正確には、*window*のトータル高さがそのフレームのルートウィンドウの高さに等しいことを意味する。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ。

`window-full-width-p` **&optional** *window* [Function]
この関数はフレーム内で *window*の左右に他のウィンドウがなければ非 `nil`をリターンする (トータル幅がそのフレーム上のルートウィンドウと等しい)。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ。

ウィンドウのボディー高さ *(body height)* とはモードライン、ヘッダーライン、水平スクロールバー、下端ディバイダーを含まないテキスト領域の高さです。

`window-body-height` **&optional** *window pixelwise* [Function]
この関数はウィンドウ *window*のボディーの高さを行数でリターンする。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ、それ以外なら生きたウィンドウでなければならない。

オプション引数 *pixelwise*が非 `nil`なら、この関数はピクセルで計算 *window*のボディー高さをリターンする。

*pixelwise*が `nil`の場合には、必要ならリターン値はもっとも近い整数に切り下げられる。これはテキスト領域の下端行が部分的に可視の場合にその行は計数されないこと、さらに任意のウィンドウのボディー高さは `window-total-height`によりリターンされるそのウィンドウのトータル高さ決して超過し得ないことも意味する。

ウィンドウのボディー幅 *(body width)* とはスクロールバー、フリンジ、マージン、右ディバイダーを含まないテキスト領域の幅です。(幅を 0 にセットすることにより) 一方または両方のフリンジが削除されたときには、継続と切り詰めのグリフを表示するためにディスプレーエンジンが文字セル 2 つを予約するので、テキスト表示にたいして 2 列少なくなることに注意してください (以下で説明する `window-max-chars-per-line`はこの特性を考慮する)。

`window-body-width` **&optional** *window pixelwise* [Function]
この関数はウィンドウ *window*のボディーの幅を列数でリターンする。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ、それ以外なら生きたウィンドウでなければならない。

オプション引数 *pixelwise*が非 `nil`なら、この関数は *window*のボディーの幅をピクセル単位でリターンする。

*pixelwise*が `nil`なら、リターン値は必要に応じてもっとも近い整数に切り下げられる。これはテキスト領域の右端の列が部分的に可視な場合にその列が計数されないことを意味する。さらにこれはウィンドウのボディーの幅が `window-total-width`によりリターンされるウィンドウのトータル幅を決して超過し得ないことをも意味する。

`window-body-size` **&optional** *window horizontal pixelwise* [Function]
この関数は *window*のボディーの高さか幅をリターンする。*horizontal*が省略または `nil`なら *window*にたいして `window-body-height`、それ以外なら `window-body-width`を呼び出すのと同じ。いずれの場合もオプション引数 *pixelwise*は呼び出された関数に渡される。

以前のバージョンの Emacs との互換性のために `window-height`は `window-total-height`、`window-width`は `window-body-width`にたいするエイリアスです。これらのエイリアス時代遅れと考えられていて将来は削除されるでしょう。

ウィンドウのモードラインとヘッダーラインのピクセル高さは以下の関数により取得できます。それらのリターン値は、そのウィンドウが以前に表示されていない場合を除いて通常は加算されます。その場合のリターン値はそのウィンドウのフレームにたいして使用を予想されるフォントが元になります。

`window-mode-line-height` **&optional** *window* [Function]
この関数は *window*モードラインの高さをピクセルでリターンする。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。*window*にモードラインがなければリターン値は 0。

`window-header-line-height` **&optional** *window* [Function]
この関数は *window*のヘッダーラインの高さをピクセル単位でリターンする。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。*window*にヘッダーラインがない場合のリターン値は 0。

ウィンドウディバイダー (Section 39.15 [Window Dividers], page 1042 を参照)、フリンジ (Section 39.13 [Fringes], page 1034 を参照)、スクロールバー (Section 39.14 [Scroll Bars], page 1039 を参照)、ディスプレイマージン (Section 39.16.5 [Display Margins], page 1047 を参照) を取得する関数については、それぞれ対応するセクションで説明されています。

Lisp プログラムでレイアウト上の判断を要する場合には、以下の関数を有用と思うでしょう:

`window-max-chars-per-line` **&optional** *window face* [Function]
この関数は指定されたウィンドウ *window* (生きたウィンドウであること) 内で、指定されたフェイス *face*で表示される文字数をリターンする。*face*がリマップ (Section 39.12.5 [Face Remapping], page 1021 を参照) されていたらリマップされたフェイスの情報がリターンされる。省略または `nil`の場合、*face*のデフォルトはデフォルトフェイス、*window*のデフォルトは選択されたウィンドウ。

この関数は `window-body-width`と異なり、*window*のフレームの正準文字幅 (canonical character width) の単位ではなく、*face*のフォントの実サイズを考慮する。また *window*の一方または両方のフリンジがなければ、継続グリフに使用されるスペースも考慮する。

ウィンドウのサイズを変更 (Section 28.4 [Resizing Windows], page 617 を参照) したりウィンドウを分割 (split) するコマンド (Section 28.6 [Splitting Windows], page 622 を参照) は、指定で

きるウィンドウの最小の高さと幅を指定する変数 `window-min-height`と `window-min-width`にしたがう。これらのコマンドはウィンドウのサイズが *fixed(*固定*)* になる変数 `window-size-fixed`にもしたがう (Section 28.5 [Preserving Window Sizes], page 621 を参照)。

`window-min-height` [User Option]
: このオプションは任意のウィンドウの最小のトータル高さを行で指定する。この値は最低でも 1 つのテキスト行、同様に (もしあれば) モードライン、ヘッダーライン、水平スクロールバー、下端ディバイダーに対応する必要がある。

`window-min-width` [User Option]
: このオプションはすべてのウィンドウの最小のトータル幅を列で指定する。この値は 2 つのテキスト列、同様に (もしあれば) マージン、フリンジ、スクロールバー、右ディバイダーに対応する必要がある。

以下の関数は、ある特定の大きさのウィンドウにたいして、それの `window-min-height`と `window-min-width`、および `window-size-fixed` (Section 28.5 [Preserving Window Sizes], page 621 を参照) の値と領域のサイズを示す。

`window-min-size` **&optional** *window horizontal ignore pixelwise* [Function]
: この関数は *window*の最小のサイズをリターンする。*window*は有効なウィンドウでなければならず、デフォルトは選択されたウィンドウ。オプション引数 *horizontal*が非 `nil`なら *window* の最小の列数、それ以外は *window*の最小の行数をリターンすることを意味する。

  このリターン値によって *window*のサイズが実際にその値にセットされた場合に *window*のすべてのコンポーネントが完全に可視にとどまることが保証される。*horizontal*が `nil`なら、(もしあれば) モードライン、ヘッダーライン、水平スクロールバー、下端ディバイダーが含まれる。*horizontal*が非 `nil`なら、(もしあれば) マージン、フリンジ、垂直スクロールバー、右ディバイダーが含まれる。

  オプション引数 *ignore*が非 `nil`なら、`window-min-height`や `window-min-width`によりセットされる固定サイズのウィンドウに強いられる制限を無視することを意味する。*ignore*が `safe`なら、生きたウィンドウは可能な限り小さな `window-safe-min-height`の行、および `window-safe-min-width`の列を得る。*ignore*にウィンドウが指定されると、そのウィンドウにたいする制限だけを無視する。その他の非 `nil`値では、すべてのウィンドウにたいする上記制限のすべてが無視されることを意味する。

  オプション引数 *pixelwise*が非 `nil`なら、*window*の最小サイズがピクセルで計数されてリターンされることを意味する。

28.4 ウィンドウのリサイズ

このセクションではフレームのサイズを変更せずにウィンドウのサイズを変更する関数について説明します。生きたウィンドウはオーバーラップしないので、これらの関数は 2 つ以上のウィンドウを含む関数上でのみ意味があります (ウィンドウのリサイズにより隣接するウィンドウのサイズも変更される)。フレーム上に単一のウィンドウしか存在しない場合には、フレームの変更以外でウィンドウのサイズ変更はできません (Section 29.3.4 [Frame Size], page 709 を参照)。

注記した場合を除き、これらの関数は引数として内部ウィンドウも許容します。内部ウィンドウのリサイズにより、同じスペースにフィットするように子ウィンドウもリサイズされます。

`window-resizable` *window delta* **&optional** *horizontal ignore pixelwise* [Function]

この関数は *window*のサイズが *delta*行により垂直に変更され得る場合には *delta*をリターンする。オプション引数 *horizontal*が非 `nil`の場合には、*window*が *delta*列単位に水平方向にリサイズ可能ならかわりに *delta*をリターンする。これは実際にはウィンドウのサイズを変更しない。

*window*が `nil`の場合のデフォルトは選択されたウィンドウ。

*delta*が正の値ならそのウィンドウが行または列の単位で拡張可能かどうかをチェックすることを意味し、*delta*が負の値ならそのウィンドウが行または列の単位で縮小可能かどうかをチェックすることを意味する。*delta*が非 0 の場合のリターン値 0 は、そのウィンドウがリサイズ可能であることを意味する。

変数 `window-min-height`と `window-min-width`には通常は許容される最小のウィンドウサイズを指定する (Section 28.3 [Window Sizes], page 613 を参照)。しかしオプション引数 *ignore*が非 `nil`なら、この関数は `window-size-fixed`と同様に `window-min-height`と `window-min-width`を無視する。そのかわりに (もしあれば) ヘッダーライン、モードライン、水平スクロールバー、下端ディバイダーに加えて 1 行分の高さのテキストエリアから構成されるウィンドウを最小高さのウィンドウとし、(もしあれば) フリンジ、マージン、スクロールバー、右ディバイダーに加えて 1 列分の幅のテキストエリアから構成されるウィンドウを最小幅のウィンドウと判断する。

オプション引数 *pixelwise*が非 `nil`なら *delta*はピクセル単位として解釈される。

`window-resize` *window delta* **&optional** *horizontal ignore pixelwise* [Function]

この関数は *window*を *delta*増加することによりリサイズを行う。*horizontal*が `nil`なら高さを *delta*行、それ以外は幅を *delta*行変更する。正の *delta*はウィンドウの拡大、負の *delta*は縮小を意味する。

*window*が `nil`の場合のデフォルトは選択されたウィンドウ。要求されたようにウィンドウをリサイズできなければエラーをシグナルする。

オプション引数 *ignore*は上述の関数 `window-resizable`の場合と同じ意味をもつ。

オプション引数 *pixelwise*が非 `nil`なら *delta*はピクセル単位として解釈される。

この関数がどのウィンドウのエッジを変更するかの選択はオプション `window-combination-resize`の値、および関連するウィンドウのコンビネーションリミット (combination limits: 組み合わせ制限) に依存し、両方のエッジを変更するような場合もいくつかある。Section 28.8 [Recombining Windows], page 626 を参照のこと。ウィンドウの下端か右端のエッジを移動することだけでリサイズするには関数 `adjust-window-trailing-edge`を使用すること。

`adjust-window-trailing-edge` *window delta* **&optional** *horizontal pixelwise* [Function]

この関数は *window*の下端エッジを *delta*行分移動する。オプション引数 *horizontal*が非 `nil`なら、かわりに右端エッジを *delta*列分移動する。*window*が `nil`の場合のデフォルトは選択されたウィンドウ。

オプション引数 *pixelwise*が非 `nil`なら *delta*はピクセル単位として解釈される。

正の *delta*はエッジを下方か右方、負の *delta*はエッジを上方か左方へ移動する。*delta*で指定された範囲までエッジを移動できなければ、この関数はエラーをシグナルせずに可能な限りエッジを移動する。

この関数は移動されたエッジに隣接するウィンドウのリサイズを試みる。何らかの理由 (隣接するウィンドウが固定サイズの場合等) によりそれが不可能なら、他のウィンドウをリサイズするかもしれない。

`window-resize-pixelwise` [User Option]
このオプションの値が非 `nil`なら Emacs はウィンドウをピクセル単位でリサイズする。これは現在のところ `split-window` (Section 28.6 [Splitting Windows], page 622 を参照)、`maximize-window`、`minimize-window`、`fit-window-to-buffer`、`fit-frame-to-buffer`、`shrink-window-if-larger-than-buffer` (すべて以下に記述) のような関数に影響を与える。

あるフレームのピクセルサイズがそのフレームの文字サイズの整数倍でないときは、たとえこのオプションが `nil`であっても少なくとも 1 つのウィンドウがピクセル単位でリサイズされるであろうことに注意。デフォルト値は `nil`。

以下のコマンドは、より具体的な方法でウィンドウをリサイズします。これらがインタラクティブに呼び出されたときは選択されたウィンドウにたいして作用します。

`fit-window-to-buffer` **&optional** *window max-height min-height max-width min-width preserve-size* [Command]
このコマンドは *window*の高さか幅をウィンドウ内のテキストにフィットするように調整する。*window*がリサイズできたら非 `nil`、それ以外は `nil`をリターンする。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ、それ以外の場合には生きたウィンドウであること。

*window*が垂直コンビネーションの一部なら、この関数は *window*の高さを調整する。新たな高さはそのウィンドウのバッファーのアクセス可能な範囲の実際の高さから計算される。オプション引数 *max-height*が非 `nil`なら、それはこの関数が *window*に与えることができる最大のトータル高さを指定する。オプション引数 *min-height*が非 `nil`なら、それは与えることができる最小のトータル高さを指定して、それは変数 `window-min-height`をオーバーライドする。*max-height*と *min-height*はいずれも、(もしあれば) モードライン、ヘッダーライン、下端ディバイダーを含む行数で指定する。

*window*が水平コンビネーションの一部で、かつオプション `fit-window-to-buffer-horizontally` (以下参照) の値が非 `nil`なら、この関数は *window*の幅を調整する。新たな幅は *window*のカレントのスタート位置以降のバッファーの最長の行から計算される。オプション引数 *max-width*は最大幅を指定して、デフォルトは *window*のフレーム幅。オプション引数 *min-width*は最小幅を指定して、デフォルトは `window-min-width`。*max-width*と *min-width*はどちらも、(もしあれば) フリンジ、マージン、スクロールバーを含む列数で指定する。

オプション引数 *preserve-size*が非 `nil`なら、将来のリサイズ操作の間の *window*のサイズを予約するパラメーターをインストールする (Section 28.5 [Preserving Window Sizes], page 621 を参照)。

オプション `fit-frame-to-buffer` (以下参照) が非 `nil`なら、この関数は `fit-frame-to-buffer` (以下参照) を呼び出すことにより、*window*のコンテンツにフィットするように *window* のフレームのリサイズを試みるだろう。

`fit-window-to-buffer-horizontally` [User Option]
これが非 `nil`なら、`fit-window-to-buffer`はウィンドウを水平方向にリサイズできる。これが `nil` (デフォルト) なら `fit-window-to-buffer`はウィンドウ決して水平方向にリサイ

ズしない。これが `only`ならウィンドウを水平方向だけにリサイズできる。その他の値では`fit-window-to-buffer`がウィンドウをどちらの方向にもリサイズできることを意味する。

`fit-frame-to-buffer` [User Option]
このオプションが非`nil`なら、`fit-window-to-buffer`はフレームをフレームのコンテンツにフィットさせることができる。フレームは、フレームのルートウィンドウが生きたウィンドウで、かつこのオプションが非`nil`の場合のみフィットされる。`horizontally`ならフレームは水平方向にのみフィットされる。`vertically`ならフレームは垂直方向にのみフィットされる。その他の非`nil`値はフレームがどちらの方向にもフィットできることを意味する。

単一のウィンドウだけを表示するフレームではコマンド`fit-frame-to-buffer`を使用してそのバッファーにフレームをフィットできます。

`fit-frame-to-buffer` **&optional** *frame max-height min-height max-width min-width only* [Command]
このコマンドは*frame*のサイズを、表示しているバッファーのコテンツに正確に調整する。*frame*には任意の生きたフレームを指定できデフォルトは選択されたフレーム。*frame*のルートウィンドウが生きている場合のみフィットが行われる。引数*max-height*、*min-height*、*max-width*、*min-width*は*frame*のルートウィンドウの新たなトータルサイズの境界を指定する。*min-height*と*min-width*のデフォルトはそれぞれ`window-min-height`と`window-min-width`。

この関数はオプション引数*only*が`vertically`なら垂直方向のみ、*only*が`horizontally`なら水平方向のみフレームをリサイズする。

`fit-frame-to-buffer`の振る舞いは次にリストする 2 つのオプションで制御可能です。

`fit-frame-to-buffer-margins` [User Option]
このオプションはフレーム周辺のマージンを指定して`fit-frame-to-buffer`でフィットさせるために使用できる。このようなマージンはたとえばタスクバーや親フレームの一部とオーバーラップするフレームのリサイズを防ぐために有用かもしれない。

これはフィットされるフレームの左右上下にフリーとして残されるピクセル数を指定する。デフォルトの`nil`はそれぞれにたいしてマージンを使用しないことを指定する。ここで指定した値は、特定のフレームにたいしてもしそのフレームの`fit-frame-to-buffer-margins`が与えられればオーバーライドされ得る。

`fit-frame-to-buffer-sizes` [User Option]
このオプションは`fit-frame-to-buffer`にたいしてサイズ境界を指定する。これは自身のバッファーにフィットされる任意のフレームのルートウィンドウの最大行と最小行、最大列と最小列のトータルを指定する。これらの値のいずれかが非`nil`なら、`fit-frame-to-buffer`の対応する引数をオーバーライドする。

`shrink-window-if-larger-than-buffer` **&optional** *window* [Command]
このコマンドは*window*にたいしてそのバッファーを完全に表示できるが、`window-min-height`以上の行を表示できるまで可能な限り*window*の高さを縮小する。リターン値はそのウィンドウがリサイズされれば非`nil`、それ以外なら非`nil`。*window*が省略または`nil`の場合のデフォルトは選択されたウィンドウ。それ以外では生きたウィンドウであること。

このコマンドはそのウィンドウがバッファーのすべてを表示するにはすでに高さが低すぎる場合、バッファーのどこかがスクリーンからスクロールオフされている場合、またはそのウィンドウがフレーム内で唯一の生きたウィンドウの場合は何も行わない。

このコマンドは自身の処理を行うために`fit-window-to-buffer` (上記参照) を呼び出す。

`balance-windows` **&optional** *window-or-frame* [Command]
この関数は各ウィンドウにたいして完全な幅、および/または完全な高さを与えるような方法によって各ウィンドウのバランスをとる。*window-or-frame*にフレームを指定すると、そのフレーム上のすべてのウィンドウのバランスをとる。*window-or-frame*にウィンドウを指定すると、そのウィンドウとウィンドウの siblings(兄弟) にたいしてのみのバランスをとる (Section 28.2 [Windows and Frames], page 609 を参照)。

`balance-windows-area` [Command]
この関数は選択されたフレーム上のすべてのウィンドウにたいして、おおよそ同じスクリーンエリアを与えようと試みる。完全な幅か高さをもつウィンドウにたいしては、他のウィンドウと比較してより多くのスペースは与えられない。

`maximize-window` **&optional** *window* [Command]
この関数は、*window*にたいして、そのフレームをリサイズしたり他のウィンドウを削除することなく、水平垂直の両方向で可能な限り大きくなるように試みる。*window*が省略または `nil` の場合のデフォルトは選択されたウィンドウ。

`minimize-window` **&optional** *window* [Command]
この関数は *window*にたいして、そのフレームをリサイズしたりそのウィンドウを削除することなく、水平垂直の両方向で可能な限り小さくなるように試みる。*window*が省略または `nil` の場合のデフォルトは選択されたウィンドウ。

28.5 ウィンドウサイズの保持

前セクションのいずれかの関数を使用してウィンドウを明示的にリサイズしたり、たとえば隣接するウィンドウのリサイズ時、ウィンドウの分割や削除 (Section 28.6 [Splitting Windows], page 622 と see Section 28.7 [Deleting Windows], page 625 を参照)、あるいはそのウィンドウのフレームをリサイズ (Section 29.3.4 [Frame Size], page 709 を参照) する際に暗黙的にリサイズできます。

同一フレーム上に 1 つ以上のリサイズ可能なウィンドウが他に存在する際には、特定のウィンドウにたいして暗黙のリサイズを避けることが可能です。この目的にたいして、Emacs にそのウィンドウのサイズの予約 *(preserve)* をアドバイスしなければなりません。これを行うための基本的な方法が 2 つあります。

`window-size-fixed` [Variable]
このバッファーローカル変数が非 `nil`なら、通常はそのバッファーを表示するすべてのウィンドウのサイズが変更できなくなる。ウィンドウ削除やそのフレームのサイズ変更により、それ以外に方法がなければ依然としてウィンドウのサイズは変更され得る。

値が `height`ならそのウィンドウの高さのみ、値が `width`ならそのウィンドウの幅のみが固定される。その他の非 `nil`値では幅と高さの両方が固定される。

この変数が `nil`でも、そのバッファーを表示している任意のウィンドウを任意の方向にリサイズできるとはいえない。これを判断するには関数 `window-resizable`を使用する。Section 28.4 [Resizing Windows], page 617 を参照のこと。

影響を受けるウィンドウにたいする明示的なリサイズや分割の試みも同様に抑制するので、`window-size-fixed`の積極性が過度な場合が多々あります。これはたとえそのウィンドウが暗黙にリサイズされた後にも、たとえば隣接するウィンドウの削除やウィンドウのフレームのリサイズの際に発生するかもしれません。以下の関数では、そのようなウィンドウのリサイズを絶対禁止としないよう試みます:

`window-preserve-size` **&optional** *window horizontal preserve* [Function]
この関数は将来のリサイズ操作用にウィンドウ *window*の高さを予約済み (preserved) としてマーク (または非マーク) する。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。オプション引数 *horizontal*が非 `nil`なら *window*の幅を予約済みとしてマーク (または非マーク) する。

オプション引数 *preserve*が `t`なら *window*ボディーのカレントの高さまたは幅を予約することを意味する。*window*の高さまたは幅は Emacs が他によい選択をもたないときのみ変更される。この関数により予約されたウィンドウにたいする高さや幅のリサイズは決してエラーを throw しない。

*preserve*が `nil`なら、この関数の以前の呼び出しにより誘発された *window*にたいする任意の拘束を解除して、*window*の高さまたは幅の予約を停止することを意味する。引数に *window*を与えて `enlarge-window`、`shrink-window`、`fit-window-to-buffer`を呼び出すことによっても対応する高速を削除できる。

現在のところ `window-preserve-size`は以下の関数から呼び出されています:

`fit-window-to-buffer`
この関数のオプション引数 *preserve-size*が非 `nil`なら、この関数により確保されたサイズは予約される (Section 28.4 [Resizing Windows], page 617 を参照)。

`display-buffer`
この関数の *alist*引数に `preserve-size`エントリーがあれば、この関数により生成されるウィンドウサイズは予約される (Section 28.13.1 [Choosing Window], page 640 を参照)。

`window-preserve-size`はウィンドウリサイズ関数から参照される `window-preserved-size`と呼ばれるウィンドウパラメーターをインストールします (Section 28.27 [Window Parameters], page 688 を参照)。このパラメーターはウィンドウが `window-preserve-size`が呼び出されたときと異なるバッファーを表示していたり、呼び出し以降にウィンドウのサイズが変化していたらウィンドウのリサイズを妨げません。

以下の関数は特定のウィンドウの高さが予約済みかどうかチェックするために使用できます:

`window-preserved-size` **&optional** *window horizontal* [Function]
この関数はウィンドウ *window*の予約済み高さをピクセル単位でリターンする。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。オプション引数 *horizontal*が非 `nil`なら *window*の予約済み幅をリターンする。*window*のサイズが予約されていなければ `nil`をリターンする。

28.6 ウィンドウの分割

このセクションでは既存のウィンドウを分割 *(splitting)* することによりウィンドウを新たに作成する関数を説明します。ここで説明する理由により関数が失敗するという意味において、特別なウィンドウがいくつかあることに注意してください。このようなウィンドウの例としてサイドウィンドウ (Section 28.17 [Side Windows], page 664 を参照) やアトミックウィンドウ (Section 28.18 [Atomic Windows], page 669 を参照) があります。

`split-window` **&optional** *window size side pixelwise* [Function]
この関数はウィンドウ *window*の隣に生きたウィンドウを新たに作成する。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ。そのウィンドウは分割 (split) されてサイズは縮小される。そのスペースはリターンされる新たなウィンドウによって吸収される。

オプションの第 2 引数 *size*は、*window*および/または新たなウィンドウのサイズを決定する。これが省略または `nil`なら、両方のウィンドウに同じサイズが割り当てられる。行数が奇数なら、余りの 1 行は新たなウィンドウに割り当てられる。*size*が正の数値なら、*window*に *size*の行数 (*side*の値によっては列数) が与えられる。*size*が負の数値なら、新たなウィンドウに $-$*size*の行数 (または列数) が与えられる。

*size*が `nil`なら、この関数は変数 `window-min-height`と `window-min-width`にしたがう (Section 28.3 [Window Sizes], page 613 を参照)。つまり分割によりこれらの変数の指定より小さいウィンドウが作成されるようならエラーをシグナルする。しかし *size*にたいして非 `nil`値を指定すると、これらの変数は無視される。その場合には許容される最小のウィンドウはテキストエリアの高さが 1 行、および/または幅が 2 列のウィンドウとみなされる。

したがって *size*が指定された場合には、生成されるウィンドウがモードラインやスクロールバー等すべてのエリアを含むのに十分な大きさがあるかどうかチェックするのは呼び出し側の責任である。これに関して必要最小限の *window*を決定するために関数 `window-min-size` (Section 28.3 [Window Sizes], page 613 を参照) を使用できる。新たなウィンドウは通常はモードラインやスクロールバー等のエリアを *window*から継承するので、この関数は新たなウィンドウの最小サイズも良好に推定する。呼び出し側は、次回の再表示前にこれに応じて継承されたエリアを削除する場合のみ、より小さなサイズを指定すること。

オプションの第 3 引数 *side*は新たなウィンドウの位置を *window*から相対的に指定する。`nil`または `below`なら新たなウィンドウは *window*の下、`above`なら *window*の上に配置される。どちらの場合でも *size*はウィンドウのトータル高さを行数で指定する。

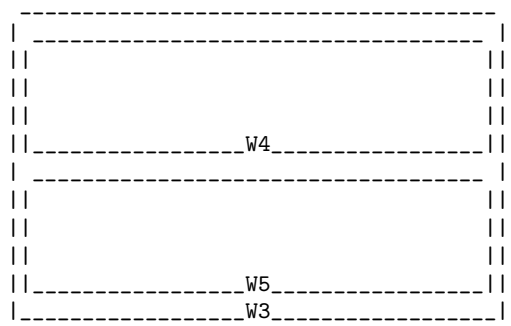
*side*が `t`か `right`なら新たなウィンドウは *window*の右、*side*が `left`なら *window*の左に配置される。どちらの場合でも *size*はウィンドウのトータル幅を列数で指定する。

オプションの第 4 引数 *pixelwise*が非 `nil`なら、*size*を行や列ではなくピクセル単位で解釈することを意味する。

*window*が生きたウィンドウの場合には、新たなウィンドウはマージンやスクロールバーを含むさまざまなプロパティを継承する。*window*が内部ウィンドウ (internal window) の場合には、新たなウィンドウは *window*のフレームのプロパティを継承する。

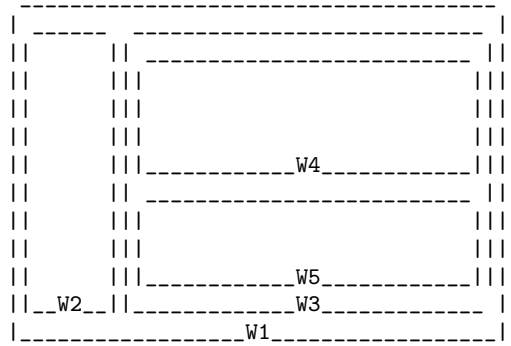
変数 `ignore-window-parameters`が `nil`の場合に限り、この関数の挙動は *window*なパラメーターにより変更されるかもしれない。ウィンドウパラメーター `split-window`の値が `t`なら、この関数はその他すべてのウィンドウパラメーターを無視する。それ以外ではウィンドウパラメーター `split-window`の値が関数の場合には、`split-window`の通常アクションのかわりに引数 *window*、*size*、*side*でその関数が呼び出される。値が関数以外なら、この関数は (もしあれば) ウィンドウパラメーター `window-atom`または `window-side`にしたがう。Section 28.27 [Window Parameters], page 688 を参照のこと。

例として Section 28.2 [Windows and Frames], page 609 で議論したウィンドウ構成 (window configuration) を得るための、一連の `split-window`呼び出しを以下に示します。この例では生きたウィンドウの分割と、内部ウィンドウの分割も示しています。最初は *W4*で表される単一のウィンドウ (生きたルートウィンドウ) を含むフレームから開始します。`(split-window W4)`を呼び出すことにより以下のウィンドウ構成が得られます。



split-window呼び出しにより *W5*で示す生きたウィンドウが新たに作成されました。*W3*で示される内部ウィンドウも新たに作成され、これはルートウィンドウかつ *W4*と *W5*の親ウィンドウになります。

次は引数として内部ウィンドウ *W3*を渡して `(split-window W3 nil 'left)`を呼び出します。



内部ウィンドウ *W3*の左に生きたウィンドウ *W2*が新たに作成されました。そして内部ウィンドウ *W1*が新たに作成され、これが新たにルートウィンドウになります。

インタラクティブな使用にたいして、Emacs は選択されたウィンドウを常に分割するコマンドを 2 つ提供します。これらは内部で split-windowを呼び出しています。

`split-window-right` **&optional** *size* [Command]
: この関数は選択されたウィンドウが左となるように、横並びの 2 つのウィンドウに分割する。*size*が正ならば左のウィンドウが *size*列、負ならば右のウィンドウが −*size*列を与えられる。

`split-window-below` **&optional** *size* [Command]
: この関数は選択されたウィンドウが上となるような、縦並びの 2 つのウィンドウに分割する。*size*が正ならば上のウィンドウが *size*行、負ならば下のウィンドウが −*size*行を与えられる。

`split-window-keep-point` [User Option]
: この変数の値が非 nil (デフォルト) なら split-window-belowは上述のように振る舞う。

  nilなら split-window-belowは再表示が最小となるように、2 つのウィンドウの各ポイントを調節する (これは低速な端末で有用)。これは何であれ、以前ポイントがあったスクリーン行 (screen line) を含むウィンドウを選択する。これは低レベル split-window関数ではなく split-window-belowだけに影響することに注意。

28.7 ウィンドウの削除

ウィンドウを削除 *(delete)* することにより、フレームのウィンドウツリーからウィンドウが取り除かれます。それが生きたウィンドウならスクリーンに表示されなくなります。内部ウィンドウならその子ウィンドウも削除されます。

ウィンドウを削除した後であっても、それへの参照が残っている限りは Lisp オブジェクトとして存在し続けます。ウィンドウ構成 (window configuration) をリストアすることにより、ウィンドウの削除は取り消すことができます (Section 28.26 [Window Configurations], page 686 を参照)。

`delete-window` **&optional** *window* [Command]

この関数は表示から *window* を削除して `nil` をリターンする。*window* が省略または `nil` の場合のデフォルトは選択されたウィンドウ。

ウィンドウ削除によりウィンドウツリーにウィンドウが何も残らなくなるか (それがフレーム内で唯一の生きたウィンドウの場合)、あるいは *window* のフレーム上の残りすべてのウィンドウがサイドウィンドウ (Section 28.17 [Side Windows], page 664 を参照) ならエラーがシグナルされる。*window* がアトミックウィンドウ (Section 28.18 [Atomic Windows], page 669 を参照) の一部なら、この関数はかわりにアトミックウィンドウのルートウィンドウの削除を試みる。

デフォルトでは *window* が占めていたスペースは、(もしあれば) 隣接する兄弟ウィンドウのうちの 1 つに与えられる。しかし変数 `window-combination-resize` が非 `nil` なら、そのスペースは同一ウィンドウコンビネーション内の残りのすべてのウィンドウに比例的に分配される。See Section 28.8 [Recombining Windows], page 626 を参照のこと。

変数 `ignore-window-parameters` が `nil` の場合に限り、この関数の振る舞いは *window* のウィンドウパラメーターにより変更される可能性がある。ウィンドウパラメーター `delete-window` の値が `t` なら、この関数はその他すべてのウィンドウパラメーターを無視する。ウィンドウパラメーター `delete-window` が関数なら、通常の `delete-window` のかわりに引数 *window* でその関数が呼び出される。Section 28.27 [Window Parameters], page 688 を参照のこと。

`delete-other-windows` **&optional** *window* [Command]

この関数は必要に応じて他のウィンドウを削除することにより、*window* でフレームを充填する。*window* が省略または `nil` の場合のデフォルトは選択されたウィンドウ。*window* がサイドウィンドウならエラーをシグナルする (Section 28.17 [Side Windows], page 664 を参照)。*window* がアトミックウィンドウの一部なら、この関数はアトミックウィンドウのルートウィンドウによるフレームの重点を試みる (Section 28.18 [Atomic Windows], page 669 を参照)。リターン値は `nil`。

変数 `ignore-window-parameters` が `nil` の場合に限り、この関数の振る舞いは変更される可能性がある。ウィンドウパラメーター `delete-other-windows` の値が `t` なら、この関数は他のすべてのウィンドウパラメーターを無視する。ウィンドウパラメーター `delete-other-windows` の値が関数なら、`delete-other-windows` の通常の動作のかわりに引数 *window* でその関数が呼び出される。Section 28.27 [Window Parameters], page 688 を参照のこと。

さらに `ignore-window-parameters` が `nil` なら、この関数は `no-delete-other-windows` パラメーターが非 `nil` のウィンドウを削除しない。

`delete-windows-on` **&optional** *buffer-or-name frame* [Command]

この関数は *buffer-or-name* を表示しているすべてのウィンドウにたいして `delete-window` を呼び出すことによってそれらを削除する。*buffer-or-name* はバッファー、またはバッファー名であること。省略または `nil` の場合のデフォルトはカレントバッファー。指定されたバッファー

を表示するウィンドウが存在しなければ、この関数は何も行わない。ミニバッファーが指定されるとエラーをシグナルする。

そのバッファーの表示に専用 (dedicated) のウィンドウがあり、フレーム上でそれが唯一のウィンドウの場合には、それが端末上で唯一のフレームでなければこの関数はそのフレームも削除する。

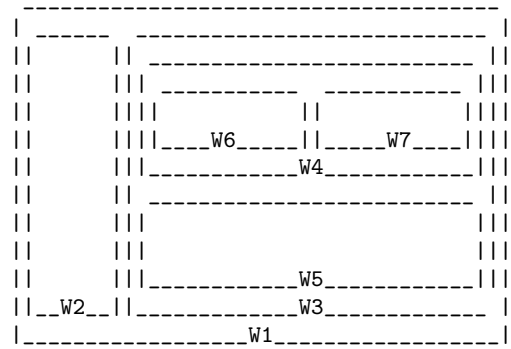
オプション引数 *frame*は操作を行うフレームがどれかを指定する:

- `nil` すべてのフレームを処理することを意味する。
- `t` 選択されたフレームを処理することを意味する。
- `visible` 可視なすべてのフレームを処理することを意味する。
- `0` 可視またはアイコン化されたすべてのフレームを処理することを意味する。
- フレームそのフレームを処理することを意味する。

この引数の意味は、すべての生きたウィンドウを走査する他の関数 (Section 28.10 [Cyclic Window Ordering], page 633 を参照) の場合とは異なることに注意。特にここでの `t`と `nil` のもつ意味は、これら他の関数の場合とは逆になる。

28.8 ウィンドウの再結合

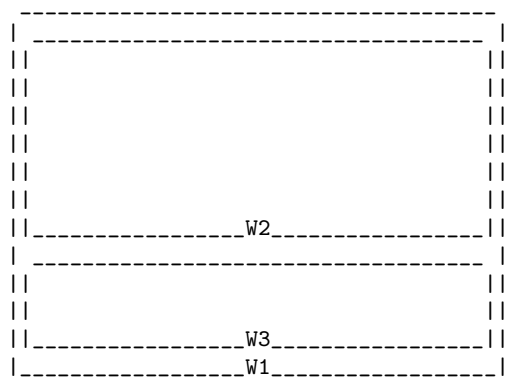
ウィンドウ *W*の最後の兄弟を削除したときは、ウィンドウツリー内の親ウィンドウを *W*を置き換えることにより、その親ウィンドウも削除されます。これは新たなウィンドウコンビネーションを形成するために、*W*がその親の兄弟たちと再結合されなければならないことを意味します。生きたウィンドウを削除することにより、必然的に 2 つの内部ウィンドウが削除されるかもしれない場合もあります。



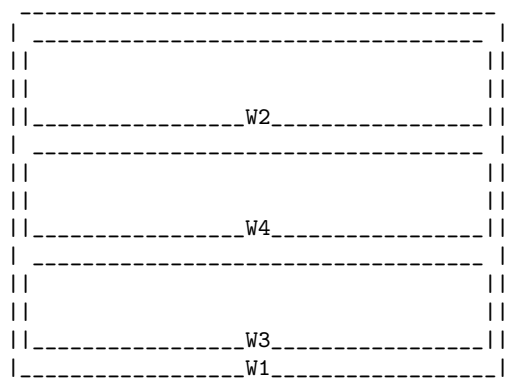
この構成における *W5*の削除は、通常は *W3*と *W4*の削除を誘発します。残りの生きたウィンドウ *W2*、*W6*、*W7*は親を *W7*とする水平コンビネーションを形成するために再結合されます。

しかし、ときには *W4*のような親ウィンドウを削除しないほうが合理的な場合もあります。特に親ウィンドウが同じタイプのコンビネーション内に埋め込まれるコンビネーションを保護するために使用されるときは、それを削除するべきではありません。そのような埋め込みは、あるウィンドウを分割した後に続けて新たなウィンドウを削除する際、Emacs が関連するフレームで分割前にあったレイアウトを確実に再構築するために意味があります。

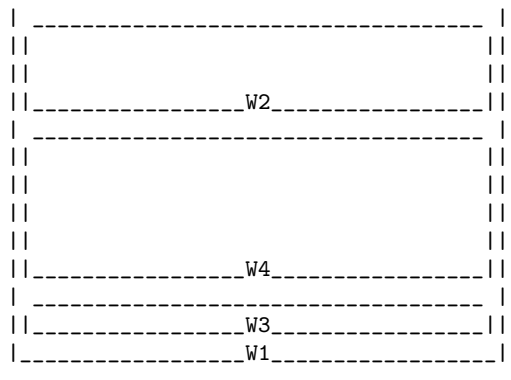
親が *W1*であるような 2 つの生きたウィンドウ *W2*と *W3*を出発点とするシナリオを考えてみましょう。



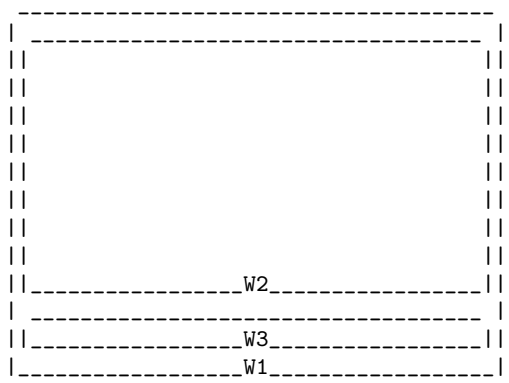
*W2*を分割すると以下のようにウィンドウ *W4*が新たに作成されます。



ここでウィンドウを垂直方向に拡大すると、Emacs はもしそのようなウィンドウがあれば下位の兄弟ウィンドウから対応するスペースを得ようと試みます。このシナリオでは *W4*の拡大により、*W3*からスペースが奪われます。



*W4*を削除すると、前に *W3*から奪ったスペースを含むスペース全体が *W2*に与えられるでしょう。



これは特に *W4*が一時的にバッファーを表示するために使用されていて (Section 39.8 [Temporary Displays], page 996 を参照)、かつ初期のレイアウトで作業を継続したい場合には直感に反するかもしれません。

この振る舞いは *W2*を分割する際に新たな親ウィンドウを作成することにより解決できます。

`window-combination-limit` [User Option]

この変数はウィンドウ分割により新たに親ウィンドウを作成させるかどうかを制御する。以下の値が認識される:

`nil`
: これは既存のウィンドウコンビネーションと同じ方向で分割が発生した場合 (これ以外の場合には、いずれにせよ内部ウィンドウが新たに作成される) は、既存の親ウィンドウが存在するなら新たな生きたウィンドウがそれを共有できることを意味する。

`window-size`
: これは `display-buffer`がウィンドウを分割する際に新たな親ウィンドウを作成して *alist*引数の `window-height`エントリーや `window-width`エントリーに渡すことを意味する (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照)。それ以外の場合にはウィンドウの分割は値が `nil`のときのように振る舞う。

`temp-buffer-resize`
: この場合には `with-temp-buffer-window`がウィンドウを分割して、かつ `temp-buffer-resize-mode`が有効なら新たに親ウィンドウを作成する (Section 39.8 [Temporary Displays], page 996 を参照)。それ以外ならウィンドウ分割は `nil`の場合のように振る舞う。

`temp-buffer`
: この場合には `with-temp-buffer-window`は既存のウィンドウの分割時に常に新たな親ウィンドウを作成する (Section 39.8 [Temporary Displays], page 996 を参照)。それ以外ならウィンドウ分割は `nil`の場合のように振る舞う。

`display-buffer`
: これは `display-buffer` (Section 28.13.1 [Choosing Window], page 640 を参照) がウィンドウを分割する際に、常に親ウィンドウを新たに作成することを意味する。それ以外ならウィンドウ分割は `nil`の場合のように振る舞う。

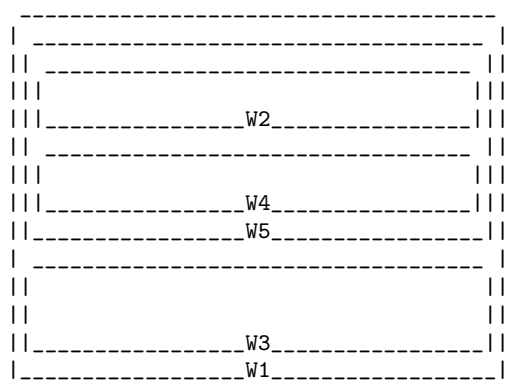
`t`
: これはウィンドウを分割することによって常に親ウィンドウが新たに作成されることを意味する。したがってこの変数の値が常に `t`なら、すべてのウィンドウツ

リーは常に 2 分木 (ルートウィンドウ以外のすべてのウィンドウが正確に 1 つの兄弟をもつようなツリー) になる。

デフォルトは `window-size`であり、それ以外の値は将来のために予約済み。

この変数のセッティングの結果として `split-window`が新たに親ウィンドウを作成したら、新たに作成された内部ウィンドウにたいして `set-window-combination-limit` (以下参照) も呼び出す。これは子ウィンドウが削除された際のウィンドウツリーの再配置に影響する (以下参照)。

`window-combination-limit`が `t`なら、このシナリオの初期構成では以下のようになるでしょう:



子として *W2*と新たな生きたウィンドウをもつ内部ウィンドウ *W5*が新たに作成されます。ここで *W2*は *W4*の唯一の兄弟なので、*W4*を拡大すると *W3*は変更せずに *W2*の縮小を試みるでしょう。*W5*は垂直コンビネーション *W1*に埋め込まれた 2 つのウィンドウからなる垂直コンビネーションを表すことに注意してください。

`set-window-combination-limit` *window limit* [Function]

この関数はウィンドウ *window*のコンビネーションリミット *(combination limit:* 結合限界*)* を *limit*にセットする。この値は関数 `window-combination-limit`を通じて取得できる。効果については以下を参照のこと。これは内部ウィンドウにたいしてのみ意味をもつことに注意。`split-window`は呼び出された際に変数 `window-combination-limit`が `t`なら、`t`を *limit* としてこの関数を呼び出す。

`window-combination-limit` *window* [Function]

この関数は *window*にたいするコンビネーションリミットをリターンする。

コンビネーションリミットは内部ウィンドウにたいしてのみ意味をもつ。これが `nil`なら Emacs はウィンドウ削除に応じて、兄弟同士で新たなウィンドウコンビネーションを形成することにより *window*の子ウィンドウをグループ化するために、*window*の自動的な削除を許す。コンビネーションリミットが `t`なら *window*の子ウィンドウがその兄弟と自動的に再結合されることは決してない。

このセクションの冒頭で示した構成の場合は、*W4* (*W6*と *W7*の親ウィンドウ) のコンビネーションリミットは `t`なので `t`を削除しても暗黙で *W4*も削除されることはない。

かわりに同じ構成内の中の 1 つのウィンドウが分割や削除されたときは、常に構成内のすべてのウィンドウをリサイズすることにより上記で示した問題を避けることができます。これはそのような操作にたいして、この方法以外では小さすぎるようなウィンドウの分割も可能にします。

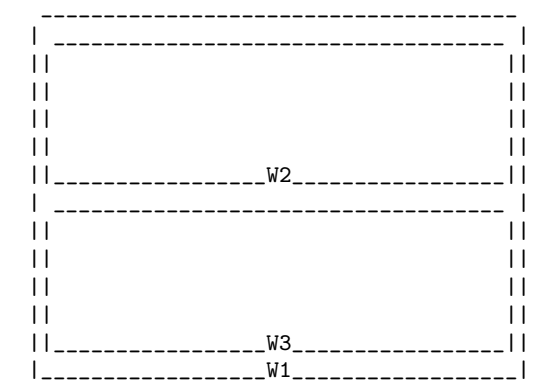
`window-combination-resize` [User Option]

この変数が`nil`なら、`split-window`はウィンドウ (以下 *window*) 自身と新たなウィンドウの両方にたいして、*window*のスクリーンエリアが十分大きい場合のみ *window*を分割できる。

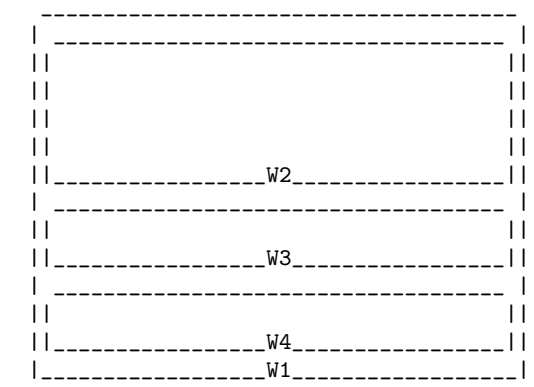
この変数が`t`なら、`split-window`は新たなウィンドウに対応するために *window*と同一コンビネーション内のすべてのウィンドウのリサイズを試みる。これは特に *window*が固定サイズウィンドウのときや、通常の分割には小さすぎるときも`split-window`をが成功することを許す。さらに続けて *window*のリサイズや削除を行うと、そのコンビネーション内のその他すべてのウィンドウをリサイズする。

デフォルトは`nil`であり、それ以外の値は将来の使用のため予約済みである。`window-combination-limit`の非`nil`値の影響を受ける場合には、この変数の値は特定の分割操作にたいして無視される。

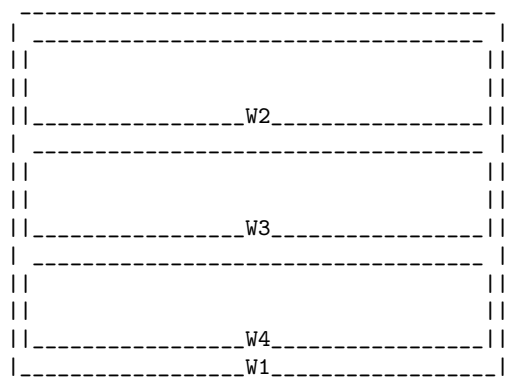
`window-combination-resize`の効果を説明するために以下のフレームレイアウトを考えてください。



`window-combination-resize`が`nil`なら、ウィンドウ *W3*を分割しても *W2*のサイズは変更されません:



`window-combination-resize`が`t`なら、*W3*を分割すると 3 つの生きたウィンドウすべてをおおよそ同じ高さにします:



生きたウィンドウ *W2*、*W3*、*W4*のいずれを削除しても、削除されたウィンドウのスペースは残りの 2 つの生きたウィンドウに相対的に分配されます。

28.9 ウィンドウの選択

`select-window` *window* **&optional** *norecord* [Function]

この関数は *window*を選択されたウィンドウにすることによりそのフレーム内で選択されたウィンドウ (Section 28.1 [Basic Windows], page 608 を参照) として、そのフレームを選択する。また *window*のバッファー (Section 28.11 [Buffers and Windows], page 635 を参照) をカレントにして、そのバッファーの `point`の値 (Section 28.19 [Window Point], page 671 を参照) を *window*の `window-point`の値にセットする。*window*は生きたウィンドウでなければならない。リターン値は *window*。

デフォルトではこの関数は *window*のバッファーをバッファーリストの先頭 (Section 27.8 [Buffer List], page 600 を参照) に移動して、*window*をもっとも最近選択されたウィンドウにする。オプション引数 *norecord*が非 `nil`なら、これらの追加処理は省略される。

加えてこの関数はデフォルトでは *window*のフレームの次回再表示の際に *window*を更新するようにディスプレイエンジンに指示する。*norecord*が非 `nil`なら、そのような更新は通常は行わない。しかし *norecord*が特別なシンボル `mark-for-redisplay`と等しければ上述の追加アクションは省略されるが、それにも関わらず *window*は更新される。

ウィンドウを選択することではウィンドウの表示やそのフレームをディスプレイ上の最上フレームにすることを満足しない場合があることに注意。さらにそのフレームのレイズやフレームにフォーカスが当たることを確実にする必要もあるだろう。Section 29.10 [Input Focus], page 732 を参照のこと。

歴史的な理由によりウィンドウ選択時に Emacs が個別にフックを実行することはありません。アプリケーションや内部ルーチンがあるウィンドウ上でいくつかの処理を行うために一時的にウィンドウを選択することはよくあります。これらは *window*引数が何も指定されていなければデフォルトでは選択されたウィンドウを処理する関数が多数存在するのでコーディングを単純化するために、あるいは引数としてウィンドウを受け取らないために常に選択されたウィンドウを処理する関数がいくつか存在した (そして今も存在する) ためにこれを行います。あるウィンドウが短時間選択されたときと、以前に選択されていたウィンドウがリストアされるたびに毎回フックを実行するのは有用ではありません。

しかし *norecord*引数が `nil`の際には `select-window`がバッファーリストを更新するので、間接的にノーマルフック `buffer-list-update-hook`が実行されます (Section 27.8 [Buffer List],

page 600 を参照)。結果としてこのフックは、あるウィンドウがより “永続的” に選択された際に関数を実行する 1 つの手段を提供します。

`buffer-list-update-hook`はウィンドウ管理とは無関係な関数によっても実行されるので、選択されたウィンドウの値を何かに保存しておいて、このフックの実行中に `selected-window`の値と比較することには意味があります。`buffer-list-update-hook`を使用する際の誤検出を防ぐためにも、一時的にのみウィンドウの選択を意図する `select-window`の呼び出しごとに *norecord*引数に非`nil`を渡すことはよい習慣です。そのような場合にはマクロ `with-selected-window` (以下参照) を使用するべきです。

最後の再表示以降に別ウィンドウが選択されたことを再表示ルーチンが検知した際には、常に Emacs はフック `window-selection-change-functions`も実行します。詳細な説明は Section 28.28 [Window Hooks], page 691 を参照してください。(同じセクションに記載されている) `window-state-change-functions`は別ウィンドウ選択後に実行される別のアブノーマルフックですが、これは他のウィンドウの変更時にも同様にトリガーされます。

引数 *norecord*に非 `nil`を指定した `select-window`の連続呼び出しは、ウィンドウの並び順を選択時刻により決定します。関数 `get-lru-window`は、もっとも昔に選択された生きたウィンドウ (Section 28.10 [Cyclic Window Ordering], page 633 を参照) を取得するために使用できます。

`save-selected-window` *forms...* [Macro]

このマクロは選択されたフレーム、同様に各フレームの選択されたウィンドウを記録して、*forms* を順に実行してから以前に選択されていたフレームとウィンドウをリストアする。これはカレントバッファーの保存とリストアも行う。リターン値は *forms*内の最後のフォームの値。

このマクロはウィンドウのサイズ、コンテンツ、配置についての保存やリストアは何も行わない。したがって *forms*がそれらを変更すると、その変更は永続化される。あるフレームにおいて以前に選択されていたウィンドウが *forms*の exit 時にすでに生きていなければ、そのフレームの選択されたウィンドウはそのまま放置される。以前に選択されていたウィンドウがすでに生きていなければ *forms*の最後に選択されていたウィンドウが何であれ、それが選択されたままになる。カレントバッファー *forms*の exit 時にそれが生きている場合のみリストアされる。

このマクロは、もっとも最近に選択されたウィンドウとバッファーリストの順番をいずれも変更しない。

`with-selected-window` *window forms...* [Macro]

このマクロは *window*を選択して *forms*を順に実行してから、以前に選択されていたウィンドウとカレントバッファーをリストアする。たとえば引数 *norecord*を `nil`で `select-window`を呼び出す等、*forms*内で故意に変更しない限り、もっとも最近に選択されたウィンドウとバッファーリストの順番は変更されない。したがってこのマクロは不要な `buffer-list-update-hook`の実行なしに、*window*を選択されたウィンドウとして一時的に作業するために好ましい手段である。

`frame-selected-window` **&optional** *frame* [Function]

この関数はフレーム *frame*内で選択されているウィンドウをリターンする。*frame*は生きたフレームであること。省略または `nil`の場合のデフォルトは選択されたフレーム。

`set-frame-selected-window` *frame window* **&optional** *norecord* [Function]

この関数は *window*をフレーム *frame*内で選択されたウィンドウにする。*frame*は生きたフレームであること。省略または `nil`の場合のデフォルトは選択されたフレーム。*window*は生きたウィンドウであること。省略または `nil`の場合のデフォルトは選択されたウィンドウ。

*frame*が選択されたフレームなら、*window*を選択されたウィンドウにする。

オプション引数 *norecord*が非 `nil`なら、この関数はもっとも最近に選択されたウィンドウのリストとバッファーリストをいずれも変更しない。

`window-use-time` **&optional** *window* [Function]
この関数はウィンドウ *window*の使用回数をリターンする。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。

ウィンドウの使用回数 *(use time)* は実際には time 値ではなく、`nil`の *norecord*引数による`select-window`の呼び出しごとに毎回単調に増加する整数。通常は最小の使用回数をもつウィンドウはもっとも最近使用されていないウィンドウ (the least recently used window)、最大の使用回数をもつウィンドウはもっとも最近使用されたウィンドウ (the most recently used window) と呼ばれる (Section 28.10 [Cyclic Window Ordering], page 633 を参照)。

28.10 ウィンドウのサイクル順

他のウィンドウを選択するためにコマンド *C*-x o (`other-window`) を使う際には、生きたウィンドウを特定の順番で巡回します。与えられた任意のウィンドウ構成にたいして、この順序は決して変更されません。これはウィンドウのサイクル順序 *(cyclic ordering of windows)* と呼ばれます。

この順序は各フレームのリーフノードである生きたウィンドウを取得するために、ツリーを深さ優先で走査することにより決定されます (Section 28.2 [Windows and Frames], page 609 を参照)。ミニバッファーがアクティブならミニバッファーウィンドウも含まれます。この順序は巡回的 (cyclic) なので、この順序の最後のウィンドウの次には最初のウィンドウが配置されます。

`next-window` **&optional** *window minibuf all-frames* [Function]
この関数はウィンドウのサイクル順で *window*の次の生きたウィンドウをリターンする。*window*は生きたウィンドウであること。省略または`nil`の場合のデフォルトは選択されたウィンドウ。

オプション引数 *minibuf*はサイクル順にミニバッファーウィンドウを含めるべきかどうかを指定する。通常は *minibuf*が`nil`のときは、ミニバッファーウィンドウがカレントでアクティブな場合のみミニバッファーウィンドウが含まれる。これは *C*-x o の振る舞いと合致する (ミニバッファーが使用されている限りミニバッファーウィンドウはアクティブであることに注意。Chapter 20 [Minibuffers], page 342 を参照のこと)。

*minibuf*が`t`なら、サイクル順にはすべてのミニバッファーウィンドウが含まれる。*minibuf*が`t`と`nil`のいずれとも異なる場合には、たとえアクティブであってもミニバッファーウィンドウは含まれない。

オプション引数 *all-frames*は考慮にするフレームを指定する:

- `nil` は *window*のフレーム上にあるウィンドウを考慮することを意味する。(*minibuf*引数で指定されたことにより) ミニバッファーウィンドウが考慮される場合には、ミニバッファーウィンドウを共有するフレームも考慮される。
- `t` はすべての既存フレーム上のウィンドウを考慮することを意味する。
- `visible` はすべての可視フレーム上のウィンドウを考慮することを意味する。
- 0 は可視またはアイコン化されたすべてのフレーム上のウィンドウを考慮することを意味する。
- フレームは指定されたフレーム上のウィンドウを考慮することを意味する。
- その他は *window*のあるフレーム上のウィンドウを考慮して、それ以外は考慮しないことを意味する。

複数のフレームが考慮される場合は、すべての生きたフレームのリストの順にしたがってそれらのフレームを順に追加することによりサイクル順を取得する (Section 29.8 [Finding All Frames], page 731 を参照)。

`previous-window` **&optional** *window minibuf all-frames* [Function]
この関数はウィンドウのサイクル順において *window*の前に位置する生きたウィンドウをリターンする。その他の引数は `next-window`の場合と同様に処理される。

`other-window` *count* **&optional** *all-frames* [Command]
この関数はウィンドウのサイクル順において、選択されたウィンドウから *count*番目に位置する生きたウィンドウをリターンする。*count*が正の数なら *count*個のウィンドウを前方にスキップし、負の数なら −*count*個のウィンドウを後方にスキップする。*count*が 0 なら選択されたウィンドウを単に再選択する．インタラクティブに呼び出された場合には、*count*はプレフィックス数引数。

オプション引数 *all-frames*は、`nil`の *minibuf*引数を指定したときの `next-window`の場合と同じ意味をもつ。

この関数は `ignore-window-parameters`が `nil`なら非 `nil`のウィンドウパラメーター `no-other-window`をもつウィンドウを選択しない。

選択されたウィンドウの `other-window`パラメーターが関数で `ignore-window-parameters`が `nil`なら、この関数の通常の処理のかわりに、`other-window`パラメーターの関数が引数 *count*と *all-frames*で呼び出される。

`walk-windows` *fun* **&optional** *minibuf all-frames* [Function]
この関数は生きたウィンドウそれぞれにたいしてウィンドウを引数に関数 *fun*を呼び出す。

これはウィンドウのサイクル順にしたがう。オプション引数 *minibuf*と *all-frames*には、含まれるウィンドウセットを指定する。これらは `next-window`の引数の場合と同じ意味をもつ。*all-frames*がフレームを指定する場合には、最初に処理されるのはそのフレームの最初のウィンドウ (`frame-first-window`がリターンするウィンドウ) であり、選択されたウィンドウである必要はない。

*fun*がウィンドウの分割や削除によりウィンドウ構成を変更する場合でも、処理するウィンドウセットは初回の *fun*呼び出しに先立ち決定されるため変更されない。

`one-window-p` **&optional** *no-mini all-frames* [Function]
この関数は選択されたウィンドウが唯一の生きたウィンドウなら `t`、それ以外は `nil`をリターンする。

ミニバッファーウィンドウがアクティブなら、ミニバッファーウィンドウは通常は考慮される (そのためこの関数は `nil`をリターンする)。しかしオプション引数 *no-mini*が非 `nil`なら、たとえアクティブであってもミニバッファーウィンドウは無視される。オプション引数 *all-frames*は `next-window`の場合と同じ意味をもつ。

以下は何らかの条件を満足するウィンドウを、それらを選択することなくリターンする関数です:

`get-lru-window` **&optional** *all-frames dedicated not-selected* [Function]
この関数は発見的に最近もっとも使用されていない生きたウィンドウをリターンする。オプション引数 *all-frames*は `next-window`の場合と同じ意味をもつ。

フル幅のウィンドウが存在する場合にはは、それらのウィンドウだけが考慮される。ミニバッファーが候補になることは決してない。オプション引数 *dedicated*が `nil`なら、専用バッファー

(Section 28.15 [Dedicated Windows], page 661 を参照) が候補になることは決してない。唯一の候補が選択されたウィンドウである場合以外は選択されたウィンドウを決してリターンしない。しかしオプション引数 *not-selected*が非 `nil`なら、そのような場合でもこの関数は `nil` をリターンする。

`get-mru-window` **&optional** *all-frames dedicated not-selected* [Function]
この関数は `get-lru-window`と同様だが、かわりにもっとも最近使用されたウィンドウをリターンする。引数の意味は `get-lru-window`で説明と同様。

`get-largest-window` **&optional** *all-frames dedicated not-selected* [Function]
この関数は、もっとも大きいエリア (高さと幅の乗) をもつウィンドウをリターンする。オプション引数 *all-frames*は検索するウィンドウを指定する。意味は `next-window`の場合と同様。

ミニバッファーウィンドウは決して候補とならない。オプション引数 *dedicated*が `nil`なら、専用ウィンドウ (Section 28.15 [Dedicated Windows], page 661 ウィンドウを参照) は決して候補とならない。オプション引数 *not-selected*が非 `nil`なら、選択されたウィンドウは決して候補にならない。オプション引数 *not-selected*が非 `nil`、かつ唯一の候補が選択されたウィンドウなら、この関数は `nil`をリターンする。

同サイズの候補ウィンドウが 2 つある場合には、この関数はウィンドウのサイクル順で選択されたウィンドウから数えて最初にあるウィンドウを優先する。

`get-window-with-predicate` *predicate* **&optional** *minibuf all-frames default* [Function]
この関数はウィンドウのサイクル順内の各ウィンドウにたいして、そのウィンドウを引数として関数 *predicate*を順に呼び出す。いずれかのウィンドウにたいして *predicate*が非 `nil`をリターンすると、この関数は処理を停止してそのウィンドウをリターンする。そのようなウィンドウが見つからなければリターン値は *default* (これのデフォルトは `nil`)。

オプション引数 *minibuf* と *all-frames*は検索するウィンドウを指定する。意味は `next-window`の場合と同様。

28.11 バッファーとウィンドウ

このセクションではウィンドウのコンテンツを調べたりセットするための低レベルな関数を説明します。ウィンドウ内に特定のバッファーを表示するための高レベルな関数については Section 28.12 [Switching Buffers], page 637 を参照してください。

`window-buffer` **&optional** *window* [Function]
この関数は *window*が表示しているバッファーをリターンする。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ。*window*が内部ウィンドウならこの関数は `nil`をリターンする。

`set-window-buffer` *window buffer-or-name* **&optional** *keep-margins* [Function]
この関数は *window*に *buffer-or-name*を表示させる。*window*は生きたウィンドウであること。`nil`の場合のデフォルトは選択されたウィンドウ。*buffer-or-name*はバッファー、あるいは既存のバッファー名であること。この関数は選択されていたウィンドウを変更せず、カレントバッファーも直接変更しない (Section 27.2 [Current Buffer], page 590 を参照)。リターン値は `nil`。

*window*があるバッファーにたいして特に専用 *(strongly dedicated)* であり、かつ *buffer-or-name*がそのバッファーを指定しなければ、この関数はエラーをシグナルする。Section 28.15 [Dedicated Windows], page 661 を参照のこと。

デフォルトではこの関数は指定されたバッファーのローカル変数にもとづいて *window*の位置、ディスプレイマージン、フリンジ幅、スクロールバーのセッティングをリセットする。しかしオプション引数 *keep-margins*が非 `nil`なら、*window*のディスプレイマージンおよびフリンジとスクロールバーのセッティングは未変更のままにする。

アプリケーションを記述する際には直接 `set-window-buffer`を呼び出すのではなく、通常は `display-buffer` (Section 28.13.1 [Choosing Window], page 640 を参照) や Section 28.12 [Switching Buffers], page 637 で説明する高レベルの関数を使用すること。

これは `window-scroll-functions`の後に `window-configuration-change-hook`を実行する。Section 28.28 [Window Hooks], page 691 を参照のこと。

`buffer-display-count` [Variable]

このバッファーローカル変数はウィンドウ内にバッファーが表示された回数を記録する。。これはそのバッファーにたいして `set-window-buffer`が呼び出されるたびに増分される

`buffer-display-time` [Variable]

このバッファーローカル変数はバッファーがウィンドウに最後に表示された時刻を記録する。バッファーが表示されたことがなければ `nil`をリターンする。これはそのバッファーにたいして `set-window-buffer`が呼び出されるたびに `current-time`がリターンする値により更新される (Section 40.5 [Time of Day], page 1104 を参照)。

`get-buffer-window` **&optional** *buffer-or-name all-frames* [Function]

この関数はウィンドウのサイクル順内で選択されたウィンドウを起点に、*buffer-or-name*を表示する最初のウィンドウをリターンする (Section 28.10 [Cyclic Window Ordering], page 633 を参照)。そのようなウィンドウが存在しなければリターン値は `nil`。

*buffer-or-name*はバッファーかバッファーの名前であること。省略または `nil`の場合のデフォルトはカレントバッファー。オプション引数 *all-frames*には考慮するウィンドウを指定する。

- `t`はすべての既存フレーム上のウィンドウを考慮することを意味する。
- `visible`はすべての可視フレーム上のウィンドウを考慮することを意味する。
- 0 はすべての可視またはアイコン化されたフレーム上のウィンドウを考慮することを意味する。
- フレームを指定すると、そのフレーム上のウィンドウだけを考慮することを意味する。
- その他の値は選択されたフレーム上のウィンドウを考慮することを意味する。

これらの意味は `next-window`の *all-frames*引数の場合とは若干異なることに注意 (Section 28.10 [Cyclic Window Ordering], page 633 を参照)。この不一致の解消のためにEmacs の将来のバージョンにおいて、この関数は変更されるかもしれない。

`get-buffer-window-list` **&optional** *buffer-or-name minibuf all-frames* [Function]

この関数はその時点で *buffer-or-name*を表示している、すべてのウィンドウのリストをリターンする。*buffer-or-name*はバッファーまたは既存バッファーの名前であること。省略または `nil`の場合のデフォルトはカレントバッファー。カレントで選択されたウィンドウが *buffer-or-name*を表示していれば、それはこの関数がリターンするリストの先頭となる。

引数 *minibuf*と *all-frames*は、関数 `next-window`の場合と同じ意味をもつ (Section 28.10 [Cyclic Window Ordering], page 633 を参照)。*all-frames*引数は、`get-buffer-window`の場合と正確に同じようには振る舞わないことに注意。

`replace-buffer-in-windows` **&optional** *buffer-or-name* [Command]

このコマンドは *buffer-or-name*を表示しているすべてのウィンドウで、それを他の何らかのバッファーに置き換える。*buffer-or-name*はバッファーまたは既存のバッファーの名前であること。省略または `nil`の場合のデフォルトはカレントバッファー。

各ウィンドウで置き換えられるバッファーは `switch-to-prev-buffer`を通じて選択される (Section 28.14 [Window History], page 659 を参照)。*buffer-or-name*を表示している専用ウィンドウは可能ならすべて削除される (Section 28.15 [Dedicated Windows], page 661 を参照)。そのようなウィンドウがそのフレームで唯一のウィンドウで、かつ同一端末上に他のフレームが存在する場合には、そのフレームも同様に削除される。その端末上の唯一のフレームの唯一のウィンドウの場合は、いずれにせよそのバッファーは置き換えられる。

28.12 ウィンドウ内のバッファーへの切り替え

このセクションでは、あるウィンドウ内で特定のバッファーにスイッチするための高レベルな関数について説明します。“バッファーをスイッチする” とは一般的に、(1) そのバッファーをあるウィンドウに表示して、(2) そのウィンドウを選択されたウィンドウとし (かつそのフレームを選択されたフレームとし)、(3) そのバッファーウィンドウカレントバッファーにすることを意味します。

Lisp プログラムがアクセスや変更できるように、バッファーを一時的にカレントにするためにこれらの関数を使用しないでください。これらはウィンドウヒストリー (Section 28.14 [Window History], page 659 を参照) の変更のような副作用をもつので、そのような方法での使用はユーザーを驚かせることになるでしょう。バッファーを Lisp で変更するためにカレントにしたければ `with-current-buffer`、`save-current-buffer`、`set-buffer`を使用してください。Section 27.2 [Current Buffer], page 590 を参照してください。

`switch-to-buffer` *buffer-or-name* **&optional** *norecord* *force-same-window* [Command]

このコマンドは選択されたウィンドウ内で *buffer-or-name*を表示して、それをカレントバッファーにしようと試みる。これはよくインタラクティブ (`C-x b`のバインディングで) に使用され、同様に Lisp プログラムでも使用される。リターン値はスイッチしたバッファー。

*buffer-or-name*が `nil`の場合のデフォルトは `other-buffer`によりリターンされるバッファー (Section 27.8 [Buffer List], page 600 を参照)。*buffer-or-name*が既存のバッファーの名前でない文字列なら、この関数はその名前で新たにバッファーを作成する。新たなバッファーのメジャーモードは変数 `major-mode`により決定される (Section 23.2 [Major Modes], page 463 を参照)。

通常は指定されたバッファーはバッファーリスト — グローバルバッファーリストと選択されたフレームのバッファーリストの両方の先頭に置かれる (Section 27.8 [Buffer List], page 600 を参照)。しかしオプション引数 *norecord*が非 `nil`なら、これは行われない。

選択されたウィンドウにそのバッファーを表示することが不適切なこともあるだろう。これは選択されたウィンドウがミニバッファーウィンドウの場合、および選択されたウィンドウがそのバッファーに特に専用 (Section 28.15 [Dedicated Windows], page 661 を参照) な場合に発生する。そのようなケースでは、このコマンドは `pop-to-buffer` (以下参照) を呼び出すことにより、通常は別のウィンドウにバッファーの表示を試みる。

オプション引数 *force-same-window*が非 `nil`、かつ選択されたウィンドウがそのバッファーの表示に不適切なら、非インタラクティブに呼び出された際にはこの関数は常にエラーをシグナルする。インタクラクティブな使用においては、もし選択されたウィンドウがミニバッファーウィンドウなら、この関数はかわりに別のウィンドウの使用を試みる。選択されたウィンドウが

そのバッファーにたいして特に専用なら、次に説明するオプション switch-to-buffer-in-dedicated-windowが使用される。

switch-to-buffer-in-dedicated-window [User Option]
このオプションが非 nilなら、switch-to-bufferがインタラクティブに呼び出されて、かつ選択されたウィンドウがそのバッファーに特に専用な際に、処理を先に進めることが許される、以下の値が許される:

nil 切り替えを許さず非インタラクティブな使用ではエラーをシグナルする。

prompt 切り替えを許すかどうかユーザーに確認を求める。

pop 処理を行うために pop-to-bufferを呼び出す。

t 選択されたウィンドウを非専用としてマークして処理を進める。

このオプションは非インタラクティブな switch-to-bufferの呼び出しには影響しない。

デフォルトでは switch-to-bufferはバッファーの point位置の維持を試みます。この振る舞いは以下のオプションを使用して調整できます。

switch-to-buffer-preserve-window-point [User Option]
この変数が nilなら、switch-to-bufferは *buffer-or-name*により指定されたバッファーを、そのバッファーの point位置で表示する。この変数が already-displayedなら、そのバッファーが任意の可視またはアイコン化されたフレーム上の他のウィンドウで表示されていれば、選択されたウィンドウ内の以前の位置でバッファーの表示を試みる。この変数が tなら、switch-to-bufferは選択されたウィンドウ内の以前の位置でそのバッファーを表示しようと試みる。

この変数はバッファーがすでに選択されたウィンドウに表示されている、これまで表示されたことがない、またはバッファーを表示するために switch-to-bufferが pop-to-bufferを呼び出した場合には無視される。

switch-to-buffer-obey-display-actions [User Option]
この変数が非 nilなら switch-to-bufferは display-buffer-overriding-actionや display-buffer-alist、およびその他の表示に関係する変数で指定されるディスプレイアクションにしたがいます。

以下の 2 つのコマンドは、説明している機能以外は switch-to-bufferと類似しています。

switch-to-buffer-other-window *buffer-or-name* **&optional** *norecord* [Command]
この関数は *buffer-or-name*で指定されたバッファーを、選択されたウィンドウ以外の別のウィンドウに表示する。これは関数 pop-to-buffer(以下参照) を内部で使用する。

選択されたウィンドウが指定されたバッファーをすでに表示していれば表示を続けるが、見つかった他のウィンドウも同様にそのバッファーを表示する。

引数 *buffer-or-name*と *norecord*は switch-to-bufferの場合と同じ意味をもつ。

switch-to-buffer-other-frame *buffer-or-name* **&optional** *norecord* [Command]
この関数は *buffer-or-name*で指定されたバッファーを新たなフレームに表示する。これは関数 pop-to-buffer (以下参照) を内部で使用する。

指定されたバッファーがすでにカレント端末上の任意のフレームの他のウィンドウに表示されている場合には、フレームを新たに作成せずにそのウィンドウに切り替える。しかしこれを行うために選択されたウィンドウを使用することは決してない。

引数 *buffer-or-name*と *norecord*は `switch-to-buffer`の場合と同じ意味をもつ。

上述したコマンドは任意のウィンドウにバッファーを柔軟に表示して、編集用にそのウィンドウを選択する関数 `pop-to-buffer`を使用しています。次に `pop-to-buffer`はバッファーの表示に `display-buffer`を使用します。したがって `display-buffer`に影響する変数も同様に影響します。`display-buffer`のドキュメントについては Section 28.13.1 [Choosing Window], page 640 を参照してください。

`pop-to-buffer` *buffer-or-name* **&optional** *action norecord* [Command]

この関数は *buffer-or-name*をカレントバッファーにして、なるべく選択されたウィンドウではないウィンドウにそれを表示する。そしてその後に表示しているウィンドウを選択する。そのウィンドウが別のグラフィカルなフレーム上にある場合には、可能ならそのフレームが入力フォーカスを与えられる (Section 29.10 [Input Focus], page 732 を参照)。

*buffer-or-name*が `nil`の場合のデフォルトは `other-buffer`によりリターンされるバッファー (Section 27.8 [Buffer List], page 600 を参照)。*buffer-or-name*が既存のバッファーの名前でない文字列なら、この関数はその名前で新たにバッファーを作成する。新たなバッファーのメジャーモードは変数 `major-mode`により決定される (Section 23.2 [Major Modes], page 463 を参照)。バッファーの表示に適したウィンドウが存在しなくても、すべてのケースにおいてそのバッファーがカレントになりリターンされる。

*action*が非 `nil`なら、それは `display-buffer`に渡すディスプレイアクション (display action) であること (Section 28.13.1 [Choosing Window], page 640 を参照)。非 `nil`か非リスト値なら、たとえそのバッファーがすでに選択されたウィンドウに表示されていたとしても、選択されたウィンドウではなく別のウィンドウをポップ (pop) することを意味する。

この関数は `switch-to-buffer`と同じように、*norecord*が `nil`ならバッファーリストを更新する。

28.13 適切なウィンドウへのバッファーの表示

このセクションでは特定のバッファーの表示にたいして Emacs が検索や作成に使用する低レベルの関数を説明します。これらの関数の共通点は、受け取ったすべてのバッファー表示要求を最終的に処理する `display-buffer` (Section 28.13.1 [Choosing Window], page 640 を参照) を主に用いるという点です。

`display-buffer`は適切なウィンドウを見つけるタスクを、いわゆるアクション関数に委譲します (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照)。まず `display-buffer`は、いわゆるアクション alist (アクション関数が振る舞いを微調整するために使用可能な連想リスト) をコンパイルします。それから呼び出す関数それぞれにたいして、その alist を渡します (Section 28.13.3 [Buffer Display Action Alists], page 645 を参照)。

`display-buffer`の動作は高度にカスタマイズ可能です。実際にカスタマイゼーションが使用される方法を理解するためには、`display-buffer`がアクション関数を呼び出す際に使用する優先順を示す例を学びたいと思うかもしれません (Section 28.13.5 [Precedence of Action Functions], page 651 を参照)。`display-buffer`を呼び出す Lisp プログラムや、`display-buffer`の動作にたいするユーザーのカスタマイズとの間の競合を避けるためには、このセクションの最後に示すいくつかのガイドラインにしたがうのが合理的かもしれません (Section 28.13.6 [The Zen of Buffer Display], page 655 を参照)。

28.13.1 バッファーを表示するウィンドウの選択

コマンド `display-buffer`は表示のために柔軟にウィンドウを選択して、そのウィンドウ内に指定されたバッファーを表示します。これはキーバインディング *C-x 4 C-o*を通じてインタラクティブに呼び出すことができます。また `switch-to-buffer`や `pop-to-buffer`を含む多くの関数やコマンドにからサブルーチンとしても使用されます (Section 28.12 [Switching Buffers], page 637 を参照)。

このコマンドはウィンドウ内に表示するウィンドウを探すために、いくつかの複雑なステップを実行します。これらのステップはディスプレイアクション *(display actions)* を用いて記述されます。ディスプレイアクションは `(`*`functions . alist`*`)`という形式をもちます。ここで *functions*は “アクション関数” と呼ばれる単一の関数か関数のリスト (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照)、*alist*は “アクション alist” と呼ばれる連想リストです (Section 28.13.3 [Buffer Display Action Alists], page 645 を参照)。ディスプレイ表示の例については Section 28.13.6 [The Zen of Buffer Display], page 655 を参照してください。

アクション関数は、表示するバッファーと、アクション alist という、2 つの引数を受け取ります。これは、自身の条件にしたがってウィンドウウィンドウ選択、または作成して、バッファーをウィンドウ内に表示します。成功した場合はそのウィンドウ、それ以外は `nil`をリターンします。

`display-buffer`は複数ソースからのディスプレイアクションを組み合わせて、アクション関数のいずれか 1 つがバッファーの表示を管理して非 `nil`値をリターンするまでアクション関数を順に呼び出します。

`display-buffer` *buffer-or-name* **&optional** *action frame* [Command]

このコマンドは、ウィンドウウィンドウ選択したり、そのバッファーをカレントにすることなく、*buffer-or-name*をウィンドウに表示させる。引数 *buffer-or-name*はバッファー、または既存のバッファーの名前でなければならない。リターン値はバッファーを表示するために選ばれたウィンドウ、適切なウィンドウが見つからなければ `nil`。

オプション引数 *action*が非 `nil`なら、それは通常はディスプレイアクション (上述) であること。`display-buffer`は以下のソース (優先度の高位順) からディスプレイアクションを集約してアクション関数リストとアクション alist を構築する:

- 変数 `display-buffer-overriding-action`。
- ユーザーオプション `display-buffer-alist`。
- *action*引数。
- ユーザーオプション `display-buffer-base-action`。
- 定数 `display-buffer-fallback-action`。

実際のところは、これらのディプレイアクションにより指定されるアクション関数すべてのリストを `display-buffer`が構築することを意味する。このリストの最初の要素は (もしあれば) `display-buffer-overriding-action`が指定する最初のアクション関数。`display-buffer-pop-up-frame`の最後の要素は `display-buffer-fallback-action`が指定する最後のアクション関数。このリストから重複要素は削除されないので、1 回の `display-buffer`呼び出しの間に同一のアクション関数が複数回呼び出されるかもしれない。

`display-buffer`はバッファーを 1 つ目の引数、組み合わされた alist を 2 つ目の引数として、いずれかの関数が非 `nil`をリターンするまで、このリスト内で指定されたアクション関数を順に呼び出す。異なるソースから指定されたディスプレイアクションを `display-buffer`がどのように処理するかの例は Section 28.13.5 [Precedence of Action Functions], page 651 を参照のこと。

2 つ目の引数は常に、上記に挙げたソースが指定するすべてのアクション alist エントリーのリストであることに注意。したがってこのリストの最初の要素は (もしあれば) `display-buffer-overriding-action`が指定する最初のアクション alist エントリー、最後の要素はもしあれば (`display-buffer-fallback-action`のアクション alist が空) `display-buffer-base-action`の最後の alist エントリーとなる。

組み合わされたアクション alist は重複したエントリーを含むかもしれず、同じキーにたするエントリーが異なる値をもつかもしれないことにも注意。アクション関数はルールとして見つかった最初のキーの連想値を常に使用する。したがってアクション関数の連想値が、そのアクション関数が指定するディスプレイアクションにより提供される連想値を使用する必要はない。

引数 *action*には非 `nil`の非 list 値も指定できる。これはたとえ選択されたウィンドウがすでにそのバッファーを表示していても、選択されたウィンドウではない別のウィンドウにバッファーが表示されるべきという特別な意味をもつ。プレフィックス引数とともにインタラクティブに呼び出された場合には、*action*は `t`である。Lisp プログラムは常にリス値を提供すること。

オプション引数 *frame*が非 `nil`の場合は、そのバッファーがすでに表示されているか判断する際、どのフレームをチェックするかを指定する。これは *action*のアクション alist に要素 (`reusable-frames . frame`) を追加するのと等価 (Section 28.13.3 [Buffer Display Action Alists], page 645 を参照)。*frame*は互換性のために提供される引数であり、Lisp プログラムが使用するべきではない。

`display-buffer-overriding-action` [Variable]
この変数の値は `display-buffer`により最高の優先順で扱われるディスプレイアクションであること。デフォルト値は空のディスプレイアクション (つまり (`nil . nil`))。

`display-buffer-alist` [User Option]
このオプションの値はディスプレイアクションにコンディション (condition: 状態) をマップする alist。コンディションはそれぞれバッファー名にマッチする正規表現、または 2 つの引数をとる関数であり、引数はバッファー名と `display-buffer`に渡す *action*引数。`display-buffer`に渡されたバッファー名がこの alist 内の正規表現にマッチするか、あるいはコンディションで指定された関数が非 `nil`をリターンした場合には、`display-buffer`はバッファーを表示するために対応するディスプレイアクションを使用する。

`display-buffer-base-action` [User Option]
このオプションの値はディスプレイアクションであること。このオプションは `display-buffer`呼び出しにたいする標準のディスプレイアクションを定義するために使用できる。

`display-buffer-fallback-action` [Constant]
このディスプレイアクションは `display-buffer`にたいして、他のディスプレイアクションが与えられなかった場合の代替え処理を指定する。

28.13.2 バッファー表示用のアクション関数

アクション関数 *(action function)* とはバッファーを表示するウィンドウを選択するために `display-buffer`が呼び出す関数です。アクション関数は *buffer* (表示するバッファー)、および *alist* (アクション alist、Section 28.13.3 [Buffer Display Action Alists], page 645 を参照) という 2 つの引数を受け取ります。これらの関数は成功時には *buffer*を表示するウィンドウ、失敗時には `nil`をリターンします。

以下の基本的なアクション関数が Emacs で定義されています。

`display-buffer-same-window` *buffer alist* [Function]
この関数は選択されたウィンドウ内に *buffer*の表示を試みる。選択されたウィンドウがミニバッファーウィンドウや他のバッファー専用 (Section 28.15 [Dedicated Windows], page 661 を参照) の場合には失敗する。*alist*に非 `nil`の `inhibit-same-window`エントリーがある場合にも失敗する。

`display-buffer-reuse-window` *buffer alist* [Function]
この関数はすでに *buffer*を表示しているウィンドウを探すことによりバッファーの表示を試みる。選択されたフレーム上のウィンドウは別フレーム上のウィンドウより優先される。

*alist*に非 `nil`の `inhibit-same-window`エントリーがある場合には、選択されたウィンドウは再利用に適さない。*buffer*をすでに表示しているウィンドウを検索するフレームセットは、アクション alist の `reusable-frames`エントリーで指定できる。*alist*に `reusable-frames`エントリーが含まれる場合には、この関数は選択されたフレームだけを検索する。

この関数が他のフレーム上のウィンドウを選択した場合には、そのフレームを可視にするとともに、*alist*が `inhibit-switch-frame`エントリーを含んでいなければ、必要ならそのフレームを最前面に移動 (raise) する。

`display-buffer-reuse-mode-window` *buffer alist* [Function]
この関数は与えられたモードですでに *buffer*を表示しているウィンドウを探すことによりバッファーの表示を試みる。

*alist*が `mode`エントリーを含んでいれば、その値がメジャーモード (シンボル)、またはメジャーモードのリストを指定する。*alist*に `mode`エントリーが含まれていなければ、かわりに *buffer*のカレントのメジャーモードが使用される。このように指定されたモードのいずれかから継承されたモードでバッファーを表示しているウィンドウは候補となる。

`display-buffer-reuse-window`のように関数の挙動は `inhibit-same-window`、`reusable-frames`、`inhibit-switch-frame`にたいする *alist*エントリーによっても制御される。

`display-buffer-pop-up-window` *buffer alist* [Function]
この関数は、最大もしくはもっとも長い間参照されていないウィンドウ (通常は選択されたフレームに配置されている) を分割することにより *buffer*の表示を試みる。これは実際には、`split-window-preferred-function` (Section 28.13.4 [Choosing Window Options], page 649 を参照) 内で指定された関数を呼び出すことにより分割を行う。

新たなウィンドウのサイズは *alist*にエントリー `window-height`と `window-width`を与えることにより調整できる。*alist*に `preserve-size`エントリーが含まれていれば、Emacs は将来のリサイズ操作の間に新たなウィンドウのサイズの維持も試みる (Section 28.5 [Preserving Window Sizes], page 621 を参照)。

この関数は分割可能なウィンドウがなければ失敗する。これの多くは、分割を許容するのに十分大きいウィンドウがない場合に発生する。この問題にたいしては `split-height-threshold`や `split-width-threshold`に小さい値をセットすることが助けになるかもしれない。選択されたフレームがフレームパラメーター `unsplittable`をもつ場合にも分割は失敗する。Section 29.4.3.5 [Buffer Parameters], page 721 を参照のこと。

`display-buffer-in-previous-window` *buffer alist* [Function]
この関数は *buffer*を以前に表示したウィンドウに *buffer*の表示を試みる。

*alist*に非 `nil`の `inhibit-same-window`エントリーが含まれる場合には、選択されたウィンドウは使用に適さない。専用ウィンドウ ()dedicated window は、すでに *buffer*を表示済み

の場合のみ使用可能。*alist*に `previous-window`エントリーが含まれる場合には、そのエントリーで指定されるウィンドウが以前に *buffer*を表示したことがなくても、そのウィンドウが使用される。

*alist*に `reusable-frames`エントリー (Section 28.13.3 [Buffer Display Action Alists], page 645 を参照) が含まれる場合には、その値が適切なウィンドウを検索するフレームを決定する。この関数は *alist*に `reusable-frames`エントリーが含まれず、`display-buffer-reuse-frames`と `pop-up-frames`がいずれも `nil`なら選択されたフレームのみ、いずれかが非 `nil`ならカレント端末上のすべてのフレームを検索する。

これらのルールに照らして 1 つ以上のウィンドウが使用に適している場合には、この関数は以下の優先順にしたがって選択を行う:

- *alist*の `previous-window`エントリーで指定されるウィンドウが選択されたウィンドウでなければそのウィンドウ。
- 以前に *buffer*を表示していたウィンドウが選択されたウィンドウでなければそのウィンドウ。
- *alist*の `previous-window`エントリーで指定されているか、あるいは以前に *buffer*を表示していれば選択されたウィンドウ。

`display-buffer-use-some-window` *buffer alist* [Function]

この関数は既存のウィンドウを選択して、そのウィンドウ内に *buffer*を表示することによりバッファーの表示を試みる。すべてのウィンドウが他のバッファー専用の場合には、この関数は失敗する可能性がある (Section 28.15 [Dedicated Windows], page 661 を参照)。

`display-buffer-in-direction` *buffer alist* [Function]

この関数は *alist*で指定した位置で *buffer*の表示を試みる。この目的のために、*alist*には値が `left`、`above` (か `up`)、`right`、`below` (か `down`) のいずれかであるような `direction`エントリーを含めること。それ以外の値は通常は `below`と解釈される。

*alist*に `window`エントリーも含まれている場合には、その値は参照ウィンドウを指定する。値には選択されたフレームのメインウィンドウ (Section 28.17.2 [Side Window Options and Functions], page 665 を参照) を意味する `main`、選択されたフレームのルートウィンドウ (Section 28.2 [Windows and Frames], page 609 を参照) を意味する `root`のような特別なシンボルを指定できる。任意の有効なウィンドウの指定も可能。それ以外の値 (または `window`エントリーを完全に省略) は参照ウィンドウとして選択されたウィンドウを使用することを意味する。

この関数は、指定方向でずすでに *buffer*を表示しているウィンドウの再利用を試みる。そのようなウィンドウが存在しなければ、指定方向に新たなウィンドウを生成するために参照ウィンドウの分割を試みる。これも同様に失敗したら指定方向にある既存ウィンドウに *buffer*の表示を試みる。いずれの場合でも `direction`エントリーで指定された参照ウィンドウ側に、少なくとも 1 辺を参照ウィンドウと共有したウィンドウが選ばれることになる。

参照ウィンドウが生きたウィンドウなら、選択されるウィンドウのエッジは `direction`エントリーで指定された方向と反対側が共有される。たとえば `direction`エントリーの値が `left`なら、選択されるウィンドウの右エッジ座標は、参照ウィンドウの左エッジ座標と等しくなる (Section 28.24 [Coordinates and Windows], page 681 を参照)。

参照ウィンドウが内部ウィンドウなら、再利用されるウィンドウは `direction`エントリーで指定されるエッジが共有されなければならない。したがって、たとえば参照ウィンドウがフレームのルートウィンフォウ、`direction`エントリーの値が `left`なら、再利用されるウィンドウ

はフレムの左側でなければならない。これは選択されるウィンドウと参照ウィンドウの左エッジ座標が等しいことを意味する。

しかし新たなウィンドウは選択したウィンドウが参照ウィンドウの反対側エッジを共有するように参照ウィンドウを分割して作成される。上記の例では参照ウィンドウを子ウィンドウとして、新たな生きたウィンドウと共にルートウィンドウが新たに作成される。選択されたウィンドウの右エッジ座標は、参照ウィンドウの左エッジ座標、左エッジ座標はフレームのルートウィンドウの左エッジ座標と等しくなる。

`direction`エントリーにたいする特別な 4 つの値 `leftmost`、`top`、`rightmost`、`bottom`では参照ウィンドウとして選択されたフレームのメインウィンドウを暗黙に指定できる。これはたとえば `(direction . left)` `(window . main)` のかわりに、単に `(direction . leftmost)` と指定することができることを意味する。このような場合では *alist*の既存の `window`エントリーは無視される。

`display-buffer-below-selected` *buffer alist* [Function]

この関数は選択されたウィンドウの下のウィンドウ内に *buffer*の表示を試みる。選択されたウィンドウの下にすでにそのバッファーを表示するウィンドウがあれば、そのウィンドウを再利用する。

そのようなウィンドウが存在しなければ、この関数は選択されたウィンドウを分割することにより新たなウィンドウを作成して *buffer*の表示を試みる。*alist*に適切な `window-height`か`window-width`のエントリーが含まれていれば、ウィンドウのサイズ調整も試みる (上記参照)。

選択されたウィンドウの分割に失敗、かつ選択されたウィンドウの下に別のバッファーを表示中の非専用ウィンドウがある場合には、この関数は *buffer*の表示にそのウィンドウの使用を試みる。

*alist*に `window-min-height`エントリーが含まれていると、この関数は少なくとも使用するウィンドウの高さがこのエントリーで指定された値になることを保証する。これは単なる保証であることに注意。使用するウィンドウを実際にリサイズするためには、*alist*で適切な `window-height`エントリーも提供しなければならない。

`display-buffer-at-bottom` *buffer alist* [Function]

この関数は選択されたフレームの最下にあるウィンドウ内に *buffer*の表示を試みる。

これはフレーム最下のウィンドウまたはフレームのルートウィンドウを分割、または選択されたフレーム最下の既存ウィンドウを試みる。

`display-buffer-pop-up-frame` *buffer alist* [Function]

この関数は新たにフレームを作成して、そのフレームのウィンドウ内にバッファーを表示する。これは実際には `pop-up-frame-function` (Section 28.13.4 [Choosing Window Options], page 649 を参照) 内で指定された関数を呼び出すことによりフレーム作成の処理を行う。*alist*が `pop-up-frame-parameters`エントリーを含む場合には、その連想値 (associated value) が新たに作成されたフレームのパラメーターに追加される。

`display-buffer-in-child-frame` *buffer alist* [Function]

この関数は選択されたフレームの既存の子フレーム、または子フレームを新たに作成して *buffer*の表示を試みる (Section 29.14 [Child Frames], page 739 を参照)。*alist*に非 `nil` の `child-frame-parameters`エントリーがあれば、対応する値が新たなフレームのフレームパラメーターの alist として与えられる。デフォルトとして選択されたフレームを指定する `parent-frame`パラメーターが提供される。その子フレームが別のフレームの子になる場合には、対応するエントリーを *alist*に追加しなければならない。

子フレームの外観は *alist*を通じて提供されるパラメーターに大きく依存する。子フレームが可視のままでいることを保証するために、少なくとも子フレームのサイズ (Section 29.4.3.3 [Size Parameters], page 717 を参照) と位置 (Section 29.4.3.2 [Position Parameters], page 715 を参照) を指定して比率 (ratio) を使用すること、および `keep-ratio`パラメーター (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照) の追加を推奨する。他に考慮すべきパラメーターについては Section 29.14 [Child Frames], page 739 を参照のこと。

`display-buffer-use-some-frame` *buffer alist* [Function]

この関数は述語を満足するフレーム (デフォルトは選択されたフレーム以外のフレーム) を探して *buffer*の表示を試みる。

この関数が他のフレーム上のウィンドウを選択した場合には、そのフレームを可視にするとともに、*alist*が `inhibit-switch-frame`エントリーを含んでいなければ、必要ならそのフレームを最前面に移動 (raise) する。

*alist*に非 `nil`の `frame-predicate`エントリーがあれば、その値は 1 つの引数 (フレーム) を受け取ってそのフレームが候補なら非 `nil`をリターンする、デフォルトの述語を置き換える関数。

*alist*に非 `nil`の `inhibit-same-window`エントリーがある場合には選択されたウィンドウは使用しない。したがって選択されたフレームに単一のウィンドウしかなければ使用しない。

`display-buffer-no-window` *buffer alist* [Function]

この関数は *alist*に非 `nil`の `allow-no-window`エントリーがあれば *buffer*を表示せずにシンボル `fail`をリターンする。この構成はアクション関数が `nil`か *buffer*を表示するウィンドウをリターンするという慣習の唯一の例外である。*alist*にそのような `allow-no-window`エントリーがなければ、この関数は `nil`をリターンする。

この関数が `fail`をリターンした場合には、`display-buffer`はそれ委譲のディスプレイアクションをスキップして即座に `nil`をリターンする。この関数が `nil`をリターンした場合には、`display-buffer`はもしあれば次のディスプレイアクションを継続する。

`display-buffer`の呼び出し側が非 `nil`の `allow-no-window`エントリーを指定した場合には、`nil`のリターン値の処理も可能とみなされる。

他の 2 つのアクション関数 `display-buffer-in-side-window`と `display-buffer-in-atom-window`については、それぞれ適正なセクションで説明します (Section 28.17.1 [Displaying Buffers in Side Windows], page 665 と Section 28.18 [Atomic Windows], page 669 を参照)。

28.13.3 バッファー表示用のアクション alist

アクション *alist(action alist)* とはアクション関数が認識可能な事前定義されたシンボルと、それに応じてそれらの関数が解釈することを意図した値をマップする連想リストです。各呼び出しにおいて `display-buffer`は新たなアクション alist(空の場合もある) を作成して、呼び出すすべてのアクション関数にそのリスト全体を渡します。

仕様によりアクション関数はアクション alist 全体を自由に解釈できます。実際のところ `allow-no-window`や `previous-window`のようないくつかのエントリーはいくつかのアクション関数によってのみ意味があり、他のアクション関数では無視されます。それ以外の `inhibit-same-window`や `window-parameters`のようなエントリーは、アプシケーションプログラムや外部パッケージから提供されるものも含めて、ほとんどのアクション関数がしたがいます。

前のサブセクションでは、個別のアクション関数がアクションエントリーを処理する方法の詳細を説明しました。ここでは既知のアクション alist エントリーに対応するすべてのシンボル、およびそ

れらの値とそれらを認識するアクション関数 (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照) とともに示すリファレンスリストを提供します。このリストにおいて用語 “バッファー” は `display-buffer`が表示しようとするバッファー、“値” はそのエントリーの値を意味しています。

`inhibit-same-window`
: 値が非`nil`なら、バッファーの表示に選択されたウィンドウを使用してはならないことを告げる。既存のウィンドウを (再) 使用するすべてのアクション関数は、このエントリーにしたがう必要がある。

`previous-window`
: 値には以前にバッファーを表示した可能性のあるウィンドウを指定しなければならない。そのようなウィンドウがまだ生きていて他のバッファー専用でなれば、`display-buffer-in-previous-window`はそのウィンドウを優先する。

`mode`
: 値はメジャーモードかメジャーモードのリスト。このエントリーが指定する値がウィンドウのバッファーのメジャーモードにマッチすれば、`display-buffer-reuse-mode-window`は常にそのウィンドウを再利用する。これ以外のアクション関数はこのエントリーを無視する。

`frame-predicate`
: 値はのそのフレームがバッファーを表示する候補なら非`nil`をリターンする単一の引数 (フレーム) を受け取る関数でなければならない。このエントリーは`display-buffer-use-some-frame`が使用する。

`reusable-frames`
: 値はバッファーをすでに表示していたことにより再利用可能なウィンドウを検索するためのフレームセットを指定する。以下をセット可能:

  - `nil`は選択されたフレーム (実際にはミニバッファーフレーム以外の最後に使用されたフレーム) 上のウィンドウだけを考慮することを意味する。
  - `t`はすべてのフレーム上のウィンドウを考慮することを意味する。
  - `visible`はすべての可視フレーム上のウィンドウを考慮することを意味する。
  - 0 はすべての可視またはアイコン化されたフレーム上のウィンドウを考慮することを意味する。
  - フレームを指定すると、そのフレーム上のウィンドウだけを考慮することを意味する。

  `nil`の意味は `next-window`にたいする *all-frames*引数の場合の意味とは若干異なることに注意 (Section 28.10 [Cyclic Window Ordering], page 633 を参照)。

  これの主要クライアントは`display-buffer-reuse-window`だが、ウィンドウの再利用を試みる他のすべてのアクション関数も同様に影響を受ける。`display-buffer-in-previous-window`はバッファーを以前に表示していたウィンドウを別フレーム上で検索する際にこれを参照する。

`inhibit-switch-frame`
: 非`nil`値の場合には、`display-buffer`により選択されたウィンドウがすでに表示されていれば別のフレームの raise や選択を抑制する。これにより主に影響を受けるのは `display-buffer-use-some-frame`と `display-buffer-reuse-window`。同様に`display-buffer-pop-up-frame`も影響を受けるべきだが、ウィンドウマネージャーがこれに準拠する保証はない。

`window-parameters`
: 値は選択したウィンドウに与えるウィンドウパラメーターの alist。ウィンドウを選択するすべてのアクション関数は、このエントリーを処理する必要がある。

`window-min-height`
: 値は使用するウィンドウの最小高さを行数で指定する。ウィンドウの高さがこのエントリーで指定した高さではない、あるいはできなければウィンドウの使用を考慮しない。このエントリーのクライアントは現在のところ `display-buffer-below-selected` のみ。

  このようなエントリーを単独で提供しても、ウィンドウがその値で指定した高さになるとは限らないことに注意。既存ウィンドウを実際にリサイズしたり新たなウィンドウの高さを指定した値にするためには、同様にこの値を指定する `window-height`エントリーを提供する必要がある。しかしこのような `window-height`エントリーでまったく異なる値を指定したり、ウィンドウ高さをバッファーにフィットするように要請することができる。そのような場合には `window-min-height`使用するウィンドウの最小高さにたいする保証を与える。

`window-height`
: 値は選択したウィンドウの高さを調節するかとその方法を指定する。以下のいずれか:

  - `nil`は選択したウィンドウの高さを変更しないことを意味する。
  - 整数値は選択したウィンドウの望ましい高さを行数で指定する。
  - 浮動小数点値は選択したウィンドウの望ましい高さをフレームのルートウィンドウのトータル高さにたいする比率で指定する。
  - 値が関数を指定する場合には、その関数は選択したウィンドウを引数として呼び出される関数。この関数はそのウィンドウの高さを調整することを期待されておりリターン値は無視される。これに適した関数は `shrink-window-if-larger-than-buffer`と `fit-window-to-buffer`。Section 28.4 [Resizing Windows], page 617 を参照のこと。

  慣例により選択したウィンドウの高さは、そのウィンドウが垂直コンビネーション (Section 28.2 [Windows and Frames], page 609 を参照) の一部であり、他の無関係のウィンドウの高さの変更を避ける場合のみ調整される。さらにこのエントリーはこのリストの後に指定する特定の条件下でのみ処理される必要がある。

`window-width`
: これは上述の `window-height`と同様だが、かわりに選択したウィンドウの幅の調節に使用される。値は以下のいずれか:

  - `nil`は選択したウィンドウの幅を変更しないことを意味する。
  - 整数値は選択したウィンドウの望ましい幅を列数で指定する。
  - 浮動小数点値は選択したウィンドウの望ましい幅をフレームのルートウィンドウのトータル幅にたいする比率で指定する。
  - 値が関数を指定する場合には、その関数は選択したウィンドウを引数として呼び出される関数。この関数はそのウィンドウの高さを調整することを期待されておりリターン値は無視される。

  慣例により選択したウィンドウの幅は、そのウィンドウが水平コンビネーション (Section 28.2 [Windows and Frames], page 609 を参照) の一部であり、他の無関係のウィ

ンドウの幅の変更を避ける場合のみ調整される。さらにこのエントリーはこのリストの後に指定する特定の条件下でのみ処理される必要がある。

`dedicated`
: このようなエントリーが非 `nil`なら、`display-buffer`は作成するすべてのウィンドウをそのバッファー専用であるとマークする (Section 28.15 [Dedicated Windows], page 661 を参照)。これは最初の引数に選択したウィンドウ、2 つ目の引数にエントリーの値を指定して `set-window-dedicated-p`を呼び出すことにより行われる。

`preserve-size`
: このようなエントリーが非 `nil`なら、選択したウィンドウのサイズを維持するように Emacs に指示する (Section 28.5 [Preserving Window Sizes], page 621 を参照)。値は `(t . nil)` (ウィンドウの幅を維持)、`(nil . t)` (高さを維持)、または `(t . t)` (幅と高さの両方を維持) のいずれか。このエントリーはこのリストの後に指定する特定の条件下でのみ処理される必要がある。

`pop-up-frame-parameters`
: この値は新たにフレームが作成された場合に与えるフレームパラメーターの alist。`display-buffer-pop-up-frame`がけが参照する。

`parent-frame`
: 値は子フレームにバッファーを表示時に使用する親フレームを指定する。このエントリーを使用するのは `display-buffer-in-child-frame`のみ。

`child-frame-parameters`
: 値は子フレームにバッファー表示時に使用するフレームパラメーターの alist を指定する。このエントリーを使用するのは `display-buffer-in-child-frame`のみ。

`side`
: 値はバッファーを表示する新たなウィンドウを、フレームやバッファーのどのサイドに作成するかを示す。このエントリーは新たなサイドウィンドウをフレームのどのサイドに配置するかを示すために `display-buffer-in-side-window`が使用する (Section 28.17.1 [Displaying Buffers in Side Windows], page 665 を参照)。さらに新たなサイドウィンドウを既存ウィンドウのどのサイドに配置するかを示すために `display-buffer-in-atom-window`にも使用される (Section 28.18 [Atomic Windows], page 669 を参照)。

`slot`
: 値が非 `nil`ならバッファーを表示するサイドウィンドウのスロットを指定する。このエントリーを使用するのは `display-buffer-in-side-window`のみ。

`direction`
: この値は `window`エントリーとともに、`display-buffer-in-direction`がバッファーを表示するウィンドウ位置を決定するための方向を指定する。

`window`
: 値は `display-buffer`で選択されたウィンドウと何らかの関連があるウィンドウを指定する。このエントリーは現在のところ新たなウィンドウが作成されるサイドのウィンドウを示すために `display-buffer-in-atom-window`が使用する。これは結果のウィンドウが出現する側の参照ウィンドウを指定するために `display-buffer-in-direction`も使用する。

`allow-no-window`
: 値が非 `nil`なら `display-buffer`は必ずしもバッファーを表示する必要はなく、呼び出し側はそれを受け入れる準備がある。`display-buffer`の任意の呼び出し手がバッ

ファーを表示するウィンドウが存在しないケースを処理できる保証がないので、、このエントリーはユーザーのカスタマイゼーションを意図したものではない。このエントリーを考慮するアクション関数は`display-buffer-no-window`のみ。

慣例によりエントリー`window-height`、`window-width`、`preserve-size`は選択したウィンドウのバッファーのセットアップ後、かつそのウィンドウが以前に他のバッファーを表示していない場合のみ適用されます。後者はより正確にはカレントの`display-buffer`呼び出しによりウィンドウが作成されたか、以前にそのバッファーを表示するために`display-buffer`によりウィンドウが作成されて、カレントの`display-buffer`呼び出しによる再利用まで他のバッファーの表示に使用されたことがないことを意味します。

28.13.4 バッファー表示の追加オプション

以下のユーザーオプションでバッファーのディスプレイアクション (Section 28.13.1 [Choosing Window], page 640 を参照) の振る舞いをさらに変更することができます。

`pop-up-windows` [User Option]

この変数の値が非`nil`なら、`display-buffer`は表示のために既存のバッファーを分割して新たなウィンドウを作成できる。

この変数は後方互換のためだけに提供される。このオプションの値が非`nil`のときはアクション関数`display-buffer-pop-up-window` (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照) を呼び出すだけの`display-buffer-fallback-action`内の特別なメカニズムを経由して`display-buffer`にしたがう。この変数は`display-buffer-alist`等により直接指定できる`display-buffer-pop-up-window`自体からは参照されない。

`split-window-preferred-function` [User Option]

この変数は、バッファーを表示する新たなウィンドウを作成するための、ウィンドウを分割する関数を指定する。これは、実際にウィンドウを分割するために、アクション関数`display-buffer-pop-up-window`により使用される。

値はウィンドウを単一の引数とする関数でなければならず、(要求されたバッファーを表示するために使用される) 新たなウィンドウ、または`nil` (分割の失敗を意味する) をリターンしなければならない。デフォルト値は`split-window-sensibly` (次に説明)。

`split-window-sensibly` **&optional** *window* [Function]

この関数は *window*を分割して新たに作成したウィンドウをリターンする。*window*を分割できなければ`nil`をリターンする。*window*が省略か`nil`の場合のデフォルトは選択されたウィンドウ。

この関数はウィンドウが分割できるかどうか判断する際の通常のルールにしたがう (Section 28.6 [Splitting Windows], page 622 を参照)。最初にまず`split-height-threshold`(以下参照)、およびその他が課す制約の元で新たなウィンドウが下になるように分割を試みる。これが失敗したら`split-width-threshold`(以下参照) が課す制約の元で新たなウィンドウが右になるように分割を試みる。これも失敗して、かつそのウィンドウがそのフレームの唯一のウィンドウなら、この関数は`split-height-threshold`を無視して新たなウィンドウが下になるように再度分割を試みる。これも同様に失敗したら、この関数は諦めて`nil`をリターンする。

`split-height-threshold` [User Option]

この変数は`split-window-sensibly`がウィンドウを分割して新たなウィンドウを下に配置できるかどうかを指定する。整数なら元のウィンドウが最低でもその行数なければ分割しないことを意味する。`nil`なら、この方法では分割しないことを意味する。

`split-width-threshold` [User Option]

この変数は `split-window-sensibly`がウィンドウを分割して新たなウィンドウを右に配置できるかどうかを指定する。整数なら元のウィンドウが最低でもその列数なければ分割しないことを意味する。`nil`なら、この方法では分割しないことを意味する。

`even-window-sizes` [User Option]

この変数が非 `nil`なら `display-buffer`が既存のウィンドウを再利用する際は常にウィンドウのサイズを均等にして、そのウィンドウを選択されたウィンドウに隣接させる。

値が `width-only`なら再利用されるウィンドウが選択されたウィンドウの左か右にあり、かつ選択されたウィンドウが再利用されるウィンドウより広い場合のみサイズは均等になる。値が `height-only`なら再利用されるウィンドウが選択されたウィンドウの上か下にあり、かつ選択されたウィンドウが再利用されるウィンドウより高い場合のみサイズは均等になる。その他の非 `nil`値は、選択されたウィンドウと再利用されるウィンドウをコンビネーション的に比較して選択されたウィンドウの方が大ならサイズを均等にすることを意味する。

`pop-up-frames` [User Option]

この変数の値が非 `nil`なら、新たにフレームを作成することにより `display-buffer`がバッファーを表示できることを意味する。デフォルトは `nil`。

非 `nil`値は `display-buffer`がすでに *buffer-or-name*を表示しているウィンドウを探す際に、選択されたフレームだけでなく可視およびアイコン化されたフレームを検索できることも意味する。

この変数は主に後方互換のために提供されている。値が非 `nil`のときは、アクション関数 `display-buffer-pop-up-frame` (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照) を呼び出すだけの `display-buffer-fallback-action`内の特別なメカニズムを経由して `display-buffer`にしたがう。この変数は `display-buffer-alist`等により直接指定できる、`display-buffer-pop-up-window`自体からは参照されない (これはウィンドウの分割前に行われる)。この変数は `display-buffer-alist`等により直接指定できる `display-buffer-pop-up-frame`自体からは参照されない。

`pop-up-frame-function` [User Option]

この変数はバッファーを表示する新たなウィンドウを作成するためのフレームを作成する関数を指定する。これはアクション関数 `display-buffer-pop-up-frame`により使用される。

値はフレーム、またはフレームを作成できなかったら `nil`をリターンする引数をとらない関数であること。デフォルト値は `pop-up-frame-alist` (以下参照) により指定されるパラメーターを使用してフレームを作成する関数。

`pop-up-frame-alist` [User Option]

この変数はフレームを新たに作成するための `pop-up-frame-function`で指定される関数が使用するフレームパラメーター (Section 29.4 [Frame Parameters], page 712 を参照) の alist を保持する。デフォルトは `nil`。

このオプションは後方互換のためだけに提供される。これは `display-buffer-pop-up-frame`が `pop-up-frame-function`の指定する関数を呼び出す際に、すべてのアクション alist の `pop-up-frame-parameters`エントリーの値に前置されるので、アクション alist のエントリーが指定する値が `pop-up-frame-alist`の対応する値を効果的にオーバーライドすることに注意。

したがってユーザーは `pop-up-frame-alist`をカスタマイズするのではなく、`display-buffer-alist`no アクション alist の `pop-up-frame-parameters`エント

リーをセットアップするべきである。ユーザーが指定したパラメーターの値だけが、display-bufferの呼び出し手が指定したパラメーターの値をオーバーライドするこが保証されている。

display-bufferのデザインでは pop-up-windows、pop-up-frames、pop-up-frame-alist、same-window-buffer-names、same-window-regexpsのような古いオプションを使用するコードにたいする保守に互換性を与えるために多くの努力が払われています。Lisp プログラムやユーザーはこれらのオプションの使用は控えるべきです。上述のように pop-up-frame-alistのカスタマイズにたいしては警告済みです。ここでは残りのオプションではなくディスプレイアクションを使用するように変換する方法を説明します。

pop-up-windows
: この変数のデフォルトは t。これを nilにカスタマイズして display-bufferに何を行うべきではないかを指示するよりも、かわりに試みるべきアクション関数をdisplay-buffer-base-action内にリストするほうがよい。たとえば:

```
(customize-set-variable
 'display-buffer-base-action
 '((display-buffer-reuse-window display-buffer-same-window
    display-buffer-in-previous-window
    display-buffer-use-some-window)))
```

pop-up-frames
: この変数をtにカスタマイズするのではなく、たとえば以下のように display-buffer-base-actionをカスタマイズすること:

```
(customize-set-variable
 'display-buffer-base-action
 '((display-buffer-reuse-window display-buffer-pop-up-frame)
   (reusable-frames . 0)))
```

same-window-buffer-names
same-window-regexps
: これらのオプションのいずれかにたいしてバッファー名や正規表現を追加するかわりに、そのバッファーにたいしてアクション関数 display-buffer-same-windowを指定する display-buffer-alistエントリーを使用すること。

```
(customize-set-variable
 'display-buffer-alist
 (cons '("\\*foo\\*" (display-buffer-same-window))
       display-buffer-alist))
```

28.13.5 アクション関数の優先順

これまでのサブセクションによってバッファーを表示するためには、display-bufferにいくつかのディスプレイアクションを提供しなければならないことがわかりました (Section 28.13.1 [Choosing Window], page 640 を参照)。まったくカスタマイズしていない Emacs では、これらのアクションは display-buffer-fallback-actionによってウィンドウの再利用、同一フレーム上での新たなウィンドウのポップアップ、そのバッファーを以前に表示したウィンドウの使用、新たなフレームをポップアップして何らかのウィンドウを使用するという優先順で指定されます (display-buffer-fallback-actionいより命名される残りのアクションは未カスタマイズの Emacs では void であることに注意)。

以下のフォームを考えてください:

```
(display-buffer (get-buffer-create "*foo*"))
```

このフォームをカスタマイズされていない Emacs の*scratch*バッファーで評価すると、通常はすでに*foo*を表示しているウィンドウの再利用は失敗しますが、新たなウィンドウのポップアップは成功するでしょう。そこで同じフォームを再度評価すると、今度は display-bufferはすでに*foo*を表示しているウィンドウを再利用して視覚的な変化は生じません。なぜならそれは許容され得るアクションであるとともに、すべての許容され得るアクションの中でもっとも高い優先度をもつからです。

選択されたフレームに十分なスペースがなければ、新たなウィンドウのポップアップは失敗します。カスタマイズされていない Emacs では、通常はフレームに 2 つのウィンドウがすでに存在すれば失敗します。たとえば今度は *C-x 1* の後に *C-x 2* をタイプしてからもう一度フォームを評価すると、*foo*は下側のウィンドウに表示されるはずです (display-bufferは単に “何らか” のウィンドウを使用した)。*C-x 2* をタイプする前に *C-x o* をタイプしていれば、*foo*は上側のウィンドウに表示されるでしょう。なぜなら “何らか” のウィンドウとは “最近もっとも使用されていない” ウィンドウという意味であり、選択されたウィンドウが最近もっとも使用されていないウィンドウになるのは、それがフレームで唯一のウィンドウの場合だけだからです。

C-x o をタイプせずに*foo*が下側のウィンドウに表示されているとしましょう。下側のウィンドウに移動するために *C-x o* の後に *C-x left* をタイプして、再びフォームを評価してみます。すると*foo*は同じ下側のウィンドウに表示されるはずです。なぜなら*foo*は以前にそのウィンドウで表示されているので何らかのウィンドウのかわりにそのウィンドウが選択されるからです。

ここまではカスタマイズしていない Emacs のデフォルトの動作を観察してきました。この動作のカスタマイズの仕方を確認するために、オプション display-buffer-base-actionについて考えてみましょう。これは概念的に任意のバッファーの表示に影響を与える、非常に大まかなカスタマイズを提供します。これは display-buffer-fallback-actionにより提供されるアクションの並び替えや、提供されていないもののユーザーの編集方法により密接に適合するアクションを追加することにより、display-buffer-fallback-actionの補強に使用することができます。しかしデフォルトの動作をより深く変更するためにも使用できます。

ルールとして別フレームにバッファーを表示することを好むユーザーを考えてみましょう。そのようなユーザーなら以下のようなカスタマイズを行うかもしれません:

```
(customize-set-variable
 'display-buffer-base-action
 '((display-buffer-reuse-window display-buffer-pop-up-frame)
   (reusable-frames . 0)))
```

このセッティングにより display-bufferはバッファーを表示するウィンドウを探すために、まず可視のフレームとアイコン化されたフレームを探して、そのようなフレームが存在しなければ新たなフレームをポップアップします。グラフィカルなシステム上で*scratch*を表示しているウィンドウで *C-x 1* をタイプして例の display-bufferフォームを評価して動作を観察できます。これは通常はルートウィンドウに*foo*を表示するフレームを新たに作成 (およびフォーカスを付与) します。このフレームをアイコン化して例のフォームを再度評価すると、display-bufferは新たなフレーム (通常はフレームを raise してフォーカスを付与) のウィンドウを再利用するでしょう。

display-bufferは新たなフレームの作成に失敗した場合のみ display-buffer-fallback-actionが提供するアクション (ウィンドウ再利用の再試行、新たなウィンドウのポップアップ等) を適用します。以下のフォームでフレーム作成を簡単に失敗させることができます:

```
(let ((pop-up-frame-function 'ignore))
  (display-buffer (get-buffer-create "*foo*")))
```

新たなフレーム作成に失敗してかわりにフォールバックアクションの使用を観察した直後にこのフォームを忘れてしまうでしょう。

`display-buffer-base-action`のカスタマイズにおいて `display-buffer-reuse-window`は冗長に思えることに注意してください。なぜならすでに `display-buffer-reuse-window`は`display-buffer-fallback-action`の一部であり、いずれにせよフォールバックアクションで試みられるはずだからです。しかしこれは `display-buffer-pop-up-frame`が優先順ですでに優っていた時点で、`display-buffer-base-action`が `display-buffer-fallback-action`より優先されることにより失敗するでしょう。実際のところ:

```
(customize-set-variable
 'display-buffer-base-action
 '(display-buffer-pop-up-frame (reusable-frames . 0)))
```

は`display-buffer`が常に新たなフレームをポップアップして、これはおそらくユーザーが望んでいない動作です。

ここまではユーザーが `display-buffer`のデフォルト動作をカスタマイズする方法だけを示しました。今度はアプリケーションが `display-buffer`の動作を変更する方法について見てみましょう。これを行う正規の手順は `display-buffer`や `pop-to-buffer`のような `display-buffer`を呼び出す関数の *action*引数を使用する方法です (Section 28.12 [Switching Buffers], page 637 を参照)。

あるアプリケーションが (新たなウィンドウにユーザーを即座に注目させるために) 可能なら選択されたウィンドウの下、それが失敗したらフレームの最下ウィンドウにに`*foo*`を表示したいと仮定してみましょう。これは以下のような呼び出しで行うことができます:

```
(display-buffer
 (get-buffer-create "*foo*")
 '((display-buffer-below-selected display-buffer-at-bottom)))
```

この新しい変更されたフォームがどのように機能するか確認するために`*foo*`を表示しているすべてのフレームを削除してから、`*scratch*`を表示中のウィンドウで *C-x 1* の後に *C-x 2* をタイプしてから、続けてそのフォームを評価してみてください。`display-buffer`は上側のウィンドウを分割して、新たなウィンドウに`*foo*`を表示するはずです。*C-x 2* の後に *C-x o* をタイプした場合には、`display-buffer`はかわりに最下にあるウィンドウを分割するでしょう。

今度は新たなフォームを評価する前に、(たとえば選択されたウィンドウで `(fit-window-to-buffer)`を評価して) 選択されたウィンドウを可能なかぎり小さくしたとします。この場合には`display-buffer`は選択されたウィンドウの分割に失敗して、フレームの最下に効果的に`*foo*`を表示するために、かわりにフレームのルートウィンドウを分割するでしょう。

いずれの場合においても新たなフォームの 2 回目の評価では、すでに`*foo*`を表示しているウィンドウの再利用を試みるはずです。これは *action*引数が提供するどちらの関数も、そのようなウィンドウの使用を最初に試みるからです。

*action*引数をセットすることにより、アプリケーションは `display-buffer-base-action`のすべてのカスタマイゼーションを効果的に無効にします。今度はユーザーはアプリケーションの選択を受け入れるか、以下のようにオプション `display-buffer-alist`をさらにカスタマイズすることができます:

```
(customize-set-variable
 'display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-reuse-window display-buffer-pop-up-frame))))
```

`*foo*`がどこにも表示されていない設定で新たな変更版のフォームを試みると、`display-buffer`の *action*を完全に無視して別フレームに`*foo*`が表示されます。

display-buffer-alistの仕様において,reusable-framesアクション alist の指定を気にしていない点に注意してください。display-bufferは常に最初に見つかったウィンドウ、この場合ではdisplay-buffer-base-actionで指定されたウィンドウを採用します。しかし異なる仕様を使用したいとき、たとえば再利用可能なウィンドウから*foo*を表示中のアイコン化されたフレームを除外したければ個別にそれを指定する必要があります:

```
(customize-set-variable
 'display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-reuse-window display-buffer-pop-up-frame)
    (reusable-frames . visible))))
```

これを試せば繰り返し*foo*の表示を試みた場合に、フレームが可視のときだけフレームの再利用に成功することに気がつくでしょう。

上記の例によりアプリケーションが選択した *action*引数を無視するという単一の目的にたいしてユーザーが display-buffer-alistをカスタマイズできるという結論が導かれるかもしれません。そのような結論は正しくありません。display-buffer-alistは表示が *action*引数によってもガイドされるかどうかに関わらず、ユーザーが好む方法で特定のバッファーの表示方法を指示するための標準オプションです。

しかし 2 つの主要な観点から、display-buffer-alistのカスタマイズは display-buffer-base-actionのカスタマイズとは異なると合理的に結論づけることができます。display-buffer-alistのカスタマイズは display-bufferの *action*引数をオーバーライドして、影響を受けるバッファーを明示的に指定できることからより強力です。実際のところ*foo*のカスタマイズによって他のバッファーの表示には何も影響がありません。たとえば、

```
(display-buffer (get-buffer-create "*bar*"))
```

は display-buffer-base-actionと display-buffer-fallback-actionのセッティングだけに管理されます。

ここで例を止めることもできますが、Lisp プログラムには display-buffer-alistにたいする任意のカスタマイズの無効化に使用できる取って置きの切り札があります。その切り札display-buffer-overriding-actionは以下のように display-buffer呼び出しの前後でバインドすることができます:

```
(let ((display-buffer-overriding-action
       '((display-buffer-same-window))))
  (display-buffer
   (get-buffer-create "*foo*")
   '((display-buffer-below-selected display-buffer-at-bottom))))
```

このフォームの評価により通常は *action*引数や任意のユーザーカスタマイゼーションとは無関係に、選択されたウィンドウに*foo*が表示されます (通常はアプリケーションが *action*を提供する必要もないが、ここではオーバーライドされる事実を示すために提供されている)。

ここで提供したカスタマイゼーションで*foo*の表示を試みたアクション関数のリストを調べれば、それが実例になるかもしれません。そのリスト (何がそれを追加したかと後続の要素を含む) は:

```
(display-buffer-same-window  ;; `display-buffer-overriding-action'
 display-buffer-reuse-window ;; `display-buffer-alist'
 display-buffer-pop-up-frame
 display-buffer-below-selected ;; ACTION argument
 display-buffer-at-bottom
 display-buffer-reuse-window ;; `display-buffer-base-action'
 display-buffer-pop-up-frame
 display-buffer--maybe-same-window ;; `display-buffer-fallback-action'
 display-buffer-reuse-window
 display-buffer--maybe-pop-up-frame-or-window
 display-buffer-in-previous-window
 display-buffer-use-some-window
 display-buffer-pop-up-frame)
```

ここで列挙した内部関数の中で display-buffer--maybe-pop-up-frame-or-windowが実際にdisplay-buffer-pop-up-windowを実行しているにも関わらず、display-buffer--maybe-same-windowが効果的に無視されていることに注意してください。

アクション関数の各呼び出しにおいて渡されるアクション alist は:

```
((reusable-frames . visible)
 (reusable-frames . 0))
```

これは上述の display-buffer-alistの 2 つ目の仕様を使用して、display-buffer-base-actionが提供する仕様をオーバーライドすることを示しています。これをユーザーが以下のように記述したと考えてみましょう

```
(customize-set-variable
 'display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-reuse-window display-buffer-pop-up-frame)
    (inhibit-same-window . t)
    (reusable-frames . visible))))
```

この場合には alist の inhibit-same-windowエントリーは display-buffer-overriding-actionから display-buffer-same-window仕様を成功裏に無効化して、display-bufferは別フレームに*foo*を表示するでしょう。この点において display-buffer-overriding-actionをより堅牢にするためには、アプリケーションはたとえば以下のように適切な inhibit-same-windowエントリーも指定する必要があるでしょう:

```
(let ((display-buffer-overriding-action
       '(display-buffer-same-window (inhibit-same-window . nil))))
  (display-buffer (get-buffer-create "*foo*")))
```

最後の例では Section 28.13.1 [Choosing Window], page 640 で説明したようにアクション関数の優先順は固定ではあるものの、優先順で低位のディスプレイアクションが指定したアクション alist のエントリーが、より高位のディスプレイアクションの実行に影響を与えれれることを示しています。

28.13.6 バッファー表示の思想

フレームのもっとも単純な形式では常にバッファー表示に使用可能な単一のウィンドウが収容されています。結果として display-bufferのもっとも最近の呼び出しは、常にそのウィンドウへのバッファーの配置に成功した呼び出しとなります。

そのようなフレームを処理することは実際的ではないので、デフォルトでは Emacs はフレームサイズのデフォルト値、`split-height-threshold`や `split-width-threshold`のオプションにより制御される、より複雑なレイアウトを許容しています。そのフレームでまだ表示されていないバッファーを表示すると、フレーム上の単一ウィンドウを分割するか、2 つのウィンドウのいずれかを (再) 利用します。

これらのしきい値のいずれかをカスタマイズしたり手動でフレームのレイアウトを変更するとデフォルトの動作は棄却されます。非 `nil`の *action*引数で `display-buffer`を呼び出したろ、前のサブセクションに示したオプションのいずれかをユーザーがカスタマイズした際にもデフォルト動作は棄却されます。`display-buffer`を習得次第、表示可能なディスプレイアクションとフレイムレイアウト結果の膨大さにフラストレーションを覚えるかもしれません。

しかしバッファー表示関の使用を控えてウィンドウのウィンドウの分割と削除のメタファーに逆行するのは良い考えではありません。Lisp プログラムやユーザーにたいしてバッファー表示関数は、異なるニーズを調整するフレームワークを提供します。ウィンドウの分割と削除にたいする同等なフレームワークは存在しません。バッファー表示関数ではフレームからバッファーを削除する際に、少なくともフレームレイアウトを部分的に後からリストアすることが可能です (Section 28.16 [Quitting Windows], page 662 を参照)。

上述したフラストレーションを埋め合わせるとともに、文字通りフレームのウィンドウ間でバッファーが失われることを避けるためるために、以下にいくつのガイダンスを示します。

Write display actions without stress

ディスプレイアクションの記述はアクション関数とアクション alist を 1 つの巨大なリストにまとめる必要があるので多きな苦痛をともなうかもしれません (歴史的な理由によって `display-buffer`の引数として個別にサポートすることができなかった)。以下のリストのような基本形式を覚えておくと便利かもしれません:

```
'(nil (inhibit-same-window . t))
```

アクション関数なしのアクション alist エントリーだけを指定する。これの唯一の目的は、どこかで指定された `display-buffer-same-window`関数が同一ウィンドウ内でのバッファー表示を抑制すること。前のサブセクションの最後の例も参照のこと。

```
'(display-buffer-below-selected)
```

一方こちらは 1 つのアクション関数と空のアクション alist を指定する。上記 2 つの指定の効果を 1 つ組み合わせるためには以下のようなフォームを記述

```
'(display-buffer-below-selected (inhibit-same-window . t))
```

別のアクション関数の追加は以下のように記述

```
'((display-buffer-below-selected display-buffer-at-bottom)
  (inhibit-same-window . t))
```

別のアクション alist を追加するには以下のように記述

```
'((display-buffer-below-selected display-buffer-at-bottom)
  (inhibit-same-window . t)
  (window-height . fit-window-to-buffer))
```

最後のフォームは以下の方法により `display-buffer`の *action*関数に使用できる:

```
(display-buffer
 (get-buffer-create "*foo*")
 '((display-buffer-below-selected display-buffer-at-bottom)
   (inhibit-same-window . t)
   (window-height . fit-window-to-buffer)))
```

`display-buffer-alist`のアスタマイズでは以下のように使用できる:

```
(customize-set-variable
 'display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-below-selected display-buffer-at-bottom)
    (inhibit-same-window . t)
    (window-height . fit-window-to-buffer))))
```

2 つ目のバッファーへのカスタマイズを追加するには以下のように記述:

```
(customize-set-variable
 'display-buffer-alist
 '(("\\*foo\\*"
    (display-buffer-below-selected display-buffer-at-bottom)
    (inhibit-same-window . t)
    (window-height . fit-window-to-buffer))
   ("\\*bar\\*"
    (display-buffer-reuse-window display-buffer-pop-up-frame)
    (reusable-frames . visible))))
```

互いを尊重して扱う

`display-buffer-alist`と `display-buffer-base-action`はユーザーオプションであって、Lisp プログラムはそれらのセットやリバインドを行ってはならない。一方で`display-buffer-overriding-action`はアプリケーション用に予約されていて滅多に使用されず、使用する場合には細心の注意を払うこと。

`display-buffer`の旧実装では、`pop-up-frames`や `pop-up-windows`のようなユーザーオプションのセッティングをめぐってユーザーとアプリケーションの競合が頻生した (Section 28.13.4 [Choosing Window Options], page 649 を参照)。これが`display-buffer`を再デザイン (ユーザーおよびアプリケーションにたいして何を行うことが許容されているかを指定する明快なフレームワークを提供する) した主な理由である。

Lisp プログラムは表示しようとしているバッファーにたいして予期せず方法ユーザーのカスタマイゼーションが影響を与えることに備えなければならない。`display-buffer`の後続の振る舞いにおいて *action*引数で要求した方法でバッファーが正確に表示されていると仮定しないこと。

ユーザーは任意のバッファーが表示される方法ひついて厳しすぎる制限を過度に多く設けるべきではない。さもないと特定の目的でバッファーを表示する際の特性を失うリスクがある。横並びの 2 つのウィンドウでバッファーの異なるバージョンを比較する Lisp プログラムを記述するとしよう。`display-buffer-alist`のカスタマイズによりそのようなすべてのバッファーは常に選択されたウィンドウの下に表示されるように定められていたら、`display-buffer`を通じて望ましいウィンドウ設定を設定するのはプログラムにとって困難だろう。

任意のバッファーを表示するための設定を指定するためには、ユーザーは`display-buffer-base-action`をカスタマイズする必要がある。複数のフレームで作業を行うことを好むユーザーについての例は前のサブセクションを参照のこと。`display-buffer-alist`は特定のバッファーを特定の方法で表示するために予約済みである。

すでにバッファーを表示しているウィンドウの再利用の考慮

一般的にユーザーと Lisp プログラムにとって、ウィンドウがすでに対象となるバッファーを表示していて、それを再利用するのは常に良いアイデアである。前のサブセクションではバッファーを表示しているフレームがすでに存在していても、正しく行うことに失敗すると `display-buffer`が継続的に新たなフレームをポップアップすることを示した。たとえばバッファーの異なる部分をそのウィンドウで表示する必要がある際のように、少数のケースにおいてはウィンドウの再利用は望ましくないかもしれない。

したがって `display-buffer-reuse-window`は *action*引数とカスタマイゼーションの両方で可能なかぎり使用するべきアクション関数の 1 つである。*action*引数の `inhibit-same-window`エントリーは、通常はバッファーをを表示中のウィンドウ、つまり対象となるウィンドウが選択されたウィンドウならそのウィンドウの再利用を避けるような、一般的なケースのほとんどを考慮する。

選択したウィンドウにフォーカスを当てる

これは複数フレームで作業を行う人にとっては思考を要しない。バッファーを表示中のフレームは自動的に raise されて `inhibit-switch-frame`が禁じていなければフォーカスを取得する。単一フレームのユーザーにとっては、このタスクは顕著に困難になり得る。この点において特に `display-buffer-pop-up-window`と `display-buffer-use-some-window`が厄介になる可能性がある。これらは表面上は任意に見えるウィンドウ (最大のウィンドウか最近もっとも使用されていないウィンドウ) を分割または使用してユーザーの注意を逸らす。

したがって Lisp プログラムのいくつかは、たとえば新たなウィンドウに関して問いに答える場所として期待されるミニバッファーウィンドウの近傍にバッファーを表示するためにフレーム最下のウィンドウの選択を試みる。選択されたウィンドウは通常はすでにユーザーの注意を喚起済みなので、入力とは無関係なアクション `display-buffer-below-selected`が好ましいかもしれない。

`display-buffer`に後続する呼び出しの処理

`display-buffer`は複数のバッファーを順番に表示するのに最適ではないので、結果となるウィンドウ構成でこれらのバッファーのすべてが順に表示されているか確認すること。繰り返しになるが標準のアクション関数 `display-buffer-pop-up-window`と `display-buffer-use-some-window`は、より複雑な構成における幾分混沌とした性質のために、この用途に最適とは言えない。

同一の表示サイクル 1 つで複数バッファーを表示するウィンドウ構成を生成するためには、Lisp プログラマーは自身でアクション関数を記述する必要を避けることはできないだろう。この問題において以下にリストしたいくつかのトリックが助けになるかもしれない。

- 新たなウィンドウのポップアップ時に既存のウィンドウ構成の破壊を回避するためにウィンドウをアトミックにする (Section 28.18 [Atomic Windows], page 669 を参照)。そうすれば新たなウィンドウは構成の外部にポップアップされる。
- 別のバッファーを表示中のウィンドウの使用を回避するために、一時的にウィンドウをバッファー専用にする (Section 28.15 [Dedicated Windows], page 661 を参照)。そうすれば非専用のウィンドウが使用される。
- `window-preserve-size`を呼び出すことにより新たなウィンドウのポップアップ時に引数のウィンドウのサイズの維持が試みられる (Section 28.5 [Preserving Window Sizes], page 621 を参照)。ただし同一構成内の別ウィンドウのサイズが縮小されるかもしれないことに留意する必要がある。

- 特定のバッファーをフレーム上の同一位置のウィンドウに常に表示するにはサイドウィンドウを使用できる (Section 28.17 [Side Windows], page 664 を参照)。これによりフレーム上で同時に表示する際に競合しないバッファーをグループ化して、他のバッファーの表示を妨害せずに同一ウィンドウでグループ化したバッファーを表示することが可能になる。
- フレームのウィンドウ構成の破壊や`display-buffer-pop-up-frame`により課せられる完全なフレームに付加されるオーバーヘッドなしで、選択されたフレームのスクリーン領域内にバッファー表示するために子フレームを使用できる (Section 29.14 [Child Frames], page 739 を参照)。

28.14 ウィンドウのヒストリー

ウィンドウはそれぞれ、リスト内に以前表示されていたバッファーと、それらのバッファーがウィンドウから削除された順序を記憶しています。このヒストリーが、たとえば`replace-buffer-in-windows` (Section 28.11 [Buffers and Windows], page 635 を参照) やウィンドウの quit (Section 28.16 [Quitting Windows], page 662 を参照) の際に使用されます。このリストは Emacs により自動的に保守されますが、これを明示的に調べたり変更するために、以下の関数を使用できます:

`window-prev-buffers` **&optional** *window* [Function]
この関数は *window*の前のコンテンツを指定するリストをリターンする。オプション引数 *window*には生きたウィンドウを指定すること。デフォルトは選択されたウィンドウ。

リスト要素はそれぞれ`(`*buffer window-start window-pos*`)`という形式をもつ。ここで *buffer*はそのウィンドウで前に表示されていたウィンドウ、*window-start*はそのバッファーが最後に表示されていたときのウィンドウのスタート位置 (Section 28.20 [Window Start and End], page 671 を参照)、*window-pos*は *window*内にそのバッファーが最後に表示されていたときのポイント位置 (Section 28.19 [Window Point], page 671 を参照)。

このリストは順序付きであり、より前の要素がより最近に表示されたバッファーに対応してして、通常は最初の要素がそのウィンドウからもっとも最近削除されたバッファーに対応する。

`set-window-prev-buffers` *window prev-buffers* [Function]
この関数は *window*の前のバッファーを *prev-buffers*の値にセットする。引数 *window*は生きたウィンドウでなければならず、デフォルトは選択されたウィンドウ。引数 *prev-buffers*は`window-prev-buffers`によりリターンされるリストと同じ形式であること。

これらに加えて各ウィンドウは次バッファー *(next buffers)* のリストを保守します。これは `switch-to-prev-buffer` (以下参照) により再表示されたバッファーのリストです。このリストは主に切り替えるバッファーを選択するために、`switch-to-prev-buffer`と`switch-to-next-buffer`により使用されます。

`window-next-buffers` **&optional** *window* [Function]
この関数は`switch-to-prev-buffer`を通じて *window*内に最近表示されたバッファーのリストをリターンする。*window*引数は生きたウィンドウか`nil` (選択されたウィンドウの意) でなければならない。

`set-window-next-buffers` *window next-buffers* [Function]
この関数は *window*の次バッファーリストを *next-buffers*にセットする。*window*引数は生きたウィンドウか`nil` (選択されたウィンドウの意)、引数 *next-buffers*はバッファーのリストであること。

以下のコマンドは`bury-buffer`や`unbury-buffer`のように、グローバルバッファーリストを巡回するために使用できます。ただしこれらはグローバルバッファーリストではなく、指定されたウィンドウのヒストリーリストのしたがって巡回します。それに加えてこれらはウィンドウ固有なウィンドウのスタート位置とポイント位置をリストアして、すでに他のウィンドウに表示されているバッファーをも表示できます。特に`switch-to-prev-buffer`コマンドは、ウィンドウにたいする置き換えバッファーを探すために`replace-buffer-in-windows`、`bury-buffer`、`quit-window`により使用されます。

`switch-to-prev-buffer` **&optional** *window bury-or-kill* [Command]

このコマンドは *window*内に前のバッファーを表示する。引数 *window*は生きたウィンドウか`nil` (選択されたウィンドウの意) であること。オプション引数 *bury-or-kill*が非`nil`なら、それは *window*内にカレントで表示されているバッファーは今まさにバリーもしくは kill されるバッファーであり、したがって将来におけるこのコマンドの呼び出しでこのバッファーに切り替えるべきではないことを意味する。

前のバッファーとは、通常は *window*内にカレントで表示されているバッファーの前に表示されていたバッファーである。しかしバリーや kill されたバッファー、または直近の`switch-to-prev-buffer`呼び出しですでに表示されたバッファーは前のバッファーとしては不適格となる。

このコマンドを繰り返して呼び出すことにより *window*内で前に表示されたすべてのバッファーが表示されてしまったら、将来の呼び出しでは *window*が表示されているフレームのバッファーリスト (Section 27.8 [Buffer List], page 600 を参照) からバッファーを表示する。

特定のバッファー、たとえば別ウィンドウに表示済みのバッファーへの切り替えを抑制するために、以下で説明するオプション`switch-to-prev-buffer-skip`を使用できる。同様に *window*のフレームが`buffer-predicate`パラメーター (Section 29.4.3.5 [Buffer Parameters], page 721 を参照) をもつ場合には、この述語は特定のバッファーへの切り替えを抑制する。

`switch-to-next-buffer` **&optional** *window* [Command]

このコマンドは *window*内の次バッファーに切り替える。つまり *window*内での最後の`switch-to-prev-buffer`コマンドの効果をアンドゥする。引数 *window*は生きたウィンドウであること。デフォルトは選択されたウィンドウ。

アンドゥ可能な`switch-to-prev-buffer`の直近の呼び出しが存在しなければ、この関数は *window*が表示されているフレームのバッファーリスト (Section 27.8 [Buffer List], page 600 を参照) からバッファーの表示を試みる。

*window*のフレームのオプション`switch-to-prev-buffer-skip`と`buffer-predicate` (Section 29.4.3.5 [Buffer Parameters], page 721 を参照) は`switch-to-prev-buffer`の場合のように、このコマンドに影響を与える。

デフォルトでは、`switch-to-prev-buffer`と`switch-to-next-buffer`は他のウィンドウで表示済みのバッファーに切り替えることができます。この挙動をオーバーライドするために以下のオプションを使用できます。

`switch-to-prev-buffer-skip` [User Option]

この変数が`nil`なら、`switch-to-prev-buffer`は別のウィンドウに表示済みのバッファーを含むすべてのバッファーに切り替えることができる。

この変数が非`nil`なら、`switch-to-prev-buffer`は特定のバッファーへの切り替えを抑制する。以下の値を使用できる:

- `this`は`switch-to-prev-buffer`が動作するウィンドウをホストするフレームで表示中のバッファーに切り替えないことを意味する。

- `visible`は可視フレームで表示中のバッファーに切り替えないことを意味する。
- 0(数値の 0) は可視やアイコン化されたフレームで表示中のバッファーに切り替えないことを意味する。
- `t`は生きたフレームで表示中のバッファーに切り替えないことを意味する。
- `switch-to-prev-buffer`の *window*引数、`switch-to-prev-buffer`が切り替えようとするバッファー、`switch-to-prev-buffer`の *bury-or-kill*引数という 3 つの引数を受け取る関数。この関数が非 `nil`をリターンすると、`switch-to-prev-buffer`は 2 つ目の引数で指定されたバッファーからの切り替えを抑制する。

コマンド `switch-to-next-buffer`は同様の方法でこのオプションにしたがう。このオプションに関数が指定されると、`switch-to-next-buffer`は 3 つ目の引数を常に `nil`にしてその関数を呼び出す。

`switch-to-prev-buffer`は `bury-buffer`、同じく `replace-buffer-in-windows`や `quit-restore-window`が呼び出すので、このオプションをカスタマイズすることによりウィンドウの quit やバッファーがバリーや kill される際の Emacs の挙動にも影響することに注意。

更に `switch-to-prev-buffer`や `switch-to-next-buffer`は特定の状況下、たとえばこれらの関数が切り替え可能なバッファーが 1 つしか残っていないときには、このオプションが無視されるかもしれないことにも注意。

28.15 専用のウィンドウ

特定のウィンドウがそのウィンドウのバッファーにたいして専用 *(dedicated)* であるとマークすることにより、バッファーを表示する関数にそのウィンドウを使用しないように告げることができます。`display-buffer` (Section 28.13.1 [Choosing Window], page 640 を参照) は、他のバッファーの表示に専用バッファーを決して使用しません。`get-lru-window`と `get-largest-window` (Section 28.10 [Cyclic Window Ordering], page 633 を参照) は、*dedicated*引数が非 `nil`のときは専用ウィンドウを候補とはみなしません。専用ウィンドウにたいする配慮に関して `set-window-buffer` (Section 28.11 [Buffers and Windows], page 635 を参照) の挙動は若干異なります。以下を参照してください。

ウィンドウからのバッファー削除、およびフレームからのウィンドウ削除を意図した関数は、処理するウィンドウが専用ウィンドウのときは特別な挙動を示す可能性があります。ここでは 3 つの基本ケース、すなわち (1) そのウィンドウがフレーム上で唯一のウィンドウの場合、(2) ウィンドウはフレーム上で唯一のウィンドウだが同一端末上に別のフレームがある場合、(3) そのウィンドウが同一端末上で唯一のフレームの唯一のウィンドウの場合、を明確に区別することにします。

特に `delete-windows-on` (Section 28.7 [Deleting Windows], page 625 を参照) は関連するフレームを削除する際にケース (2) を、フレーム上で唯一のウィンドウに他のバッファーを表示する際にケース (3) を処理します。バッファーが kill される際に呼び出される関数 `replace-buffer-in-windows`(Section 28.11 [Buffers and Windows], page 635 を参照) は、ケース (1) ではウィンドウを削除して、それ以外では `delete-windows-on`のように振る舞います。

`bury-buffer` (Section 27.8 [Buffer List], page 600 を参照) が選択されたウィンドウを操作する際は、選択されたフレームを処理するために `frame-auto-hide-function` (Section 28.16 [Quitting Windows], page 662 を参照) を呼び出すことによってケース (2) を取り扱います。他の 2 つのケースは `replace-buffer-in-windows`と同様に処理されます。

`window-dedicated-p` **&optional** *window* [Function]

この関数は *window*がそのバッファーにたいして専用なら非 `nil`、それ以外は `nil`をリターンする。より正確には最後の `set-window-dedicated-p`呼び出しで割り当てられた値、

`set-window-dedicated-p`が *window*を引数として呼び出されたことがなければ `nil`がリターン値となる。*window*のデフォルトは選択されたウィンドウ。

`set-window-dedicated-p` *window flag* [Function]
この関数は *flag*が非 `nil`なら *window*がそのバッファーに専用、それ以外は非専用とマークする。

特別なケースとして *flag*が `t`の場合には、*window*はそのバッファーにたいして特に専用 *(strongly dedicated)* になる。`set-window-buffer`は処理対象のウィンドウが特に専用のウィンドウで、かつ表示を要求されたバッファーが表示済みでなければエラーをシグナルする。その他の関数は `t`を他の非 `nil`値と区別して扱わない。

適切な`dedicated`アクション alist エントリー (Section 28.13.3 [Buffer Display Action Alists], page 645 を参照) を与えることにより、`display-buffer`が作成するウィンドウにたいしてそのバッファー専用であるとマークするよう指示することもできます。

28.16 ウィンドウの quit

バッファーを表示するために使用しているウィンドウを削除したいときには、フレームからそのウィンドウを削除するために `delete-window`や `delete-windows-on`を呼び出すことができます (Section 28.7 [Deleting Windows], page 625 を参照)。そのバッファーが別フレームで表示されているときには、かわりに `delete-frame`を呼び出したいと思うかもしれません (Section 29.7 [Deleting Frames], page 730 を参照)。その一方でバッファーを表示するためにウィンドウが再利用されている場合には、関数 `switch-to-prev-buffer`を呼び出して前に表示されていたバッファーを表示したいと思うかもしれません (Section 28.14 [Window History], page 659 を参照)。最終的にはそのウィンドウのバッファーをバリー (Section 27.8 [Buffer List], page 600 を参照) や kill(Section 27.10 [Killing Buffers], page 603 を参照) したいと思うかもしれません。

以下のコマンドは、最初にどのようにバッファーを表示するウィンドウを取得するかという情報を使用して、上述で説明した処理の自動化を試みます。

`quit-window` **&optional** *kill window* [Command]
このコマンドは *window*を quit してそのバッファーをバリーする。引数 *window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。プレフィックス引数 *kill*が非 `nil`ならバッファーをバリーするかわりに kill する。これはウィンドウとそのバッファーを処理するために、次に説明する関数 `quit-restore-window`を呼び出す。

`quit-window-hook`の関数は何か他のことを行う前に実行される。

`quit-restore-window` **&optional** *window bury-or-kill* [Function]
この関数は *window*の quit 後にウィンドウとウィンドウのバッファーを処理する。オプション引数 *window*は生きたウィンドウでなければならず、デフォルトは選択されたウィンドウ。この関数の挙動は *window*の `quit-restore`パラメーターにより指定される 4 要素のリストにより決定される (Section 28.27 [Window Parameters], page 688 を参照)。

`quit-restore`パラメーターの 1 つ目の要素は `window` (ウィンドウは `display-buffer`により特別に作られた)、`frame` (別のフレームが作られた)、`same` (ウィンドウは前と同じバッファーを表示する)、`other` (ウィンドウは前と異なるバッファーを表示する) のシンボルのいずれか。`frame`と `window`はウィンドウが quit される方法、`same`と `other`は *window*に以前に表示されていたバッファーの再表示に影響する。

パラメーターの 2 つ目の要素はシンボル `window`、`frame`、またはそのウィンドウに前に表示されていたバッファー、そのときのウィンドウ start 位置、ウィンドウポイント位置、ウィン

ドウの高さを要素にもつリストのいずれか。*window*の quit 時にバッファーがまだ生きていれば、この関数はバッファーの表示に *window*を再利用するかもしれない。

3 つ目の要素はパラメーター作成時に選択されていたウィンドウ。この関数が *window*を削除すると、それに続いてこの要素が指名するウィンドウを再選択する。

4 つ目の要素は、その表示がこのパラメーターの生成を引き起こしたバッファー。この関数は指定された *window*がまだそのバッファーを表示している場合のみそれを削除する。

この関数は、(1) *window*の `quit-restore`パラメーターの 1 つ目の要素が `window`か `frame`、(2) `window`が以前に表示したバッファーの履歴をもたない、(3) `quit-restore`の 4 つ目の要素が *window*にカレントで表示中のバッファーを指定している場合のみ *window*を削除する。*window*がアトミックウィンドウ (Section 28.18 [Atomic Windows], page 669 を参照) の一部なら、かわりにアトミックウィンドウのルートウィンドウの削除を試みる。いずれの場合でも *window*が削除できない場合にはエラーのシグナルを回避しようと試みる。

*window*がそのフレームで唯一のウィンドウであり、かつそのフレームの端末上に他のフレームが存在する場合には、オプション引数 *bury-or-kill*がそのウィンドウの処置を決定する。*bury-or-kill*が `kill`なら無条件でフレームは削除される。それ以外ではフレームの処遇はそのフレームを単一の引数とする `frame-auto-hide-function` (以下参照) の呼び出しにより決定される。

`quit-restore`パラメーターの 3 つ目の要素がバッファーのリストならウィンドウの開始 (Section 28.20 [Window Start and End], page 671 を参照)、ポイント (Section 28.19 [Window Point], page 671 を参照)、およびそのバッファーは死なずにバッファーは表示されてそれに応じてウィンドウの開始とポイントがセットされる。加えて *window*のバッファーが過去に一時的にリサイズされていた場合には、この関数は *window*の元の高さのリストアも試みる。

それ以外の場合には、以前に別のバッファーの表示に *window*が使用されていれば (Section 28.14 [Window History], page 659 を参照)、その履歴中でもっとも最近のバッファーが表示される。いずれの場合でも *window*が削除されなければ、`quit-restore`パラメーターは `nil`にリセットされる。

オプション引数 *bury-or-kill*には *window*を処理する方法を指定し、以下の値が処理される。

`nil` これはバッファーを特別な方法で処理しないことを意味する。その結果として *window*が削除されない場合には、`switch-to-prev-buffer`の呼び出しにより通常はそのバッファーが再び表示されるだろう。

`append` これは *window*が削除されない場合には、そのバッファーを *window*の前のバッファーリストの最後に移動するので、将来の `switch-to-prev-buffer`呼び出しでこのバッファーには切り替わることは少なくなる。これはそのバッファーをフレームのバッファーリストの最後への移動も行う。

`bury` これは *window*が削除されない場合には、そのバッファーを *window*の前のバッファーリストから削除する。これはそのバッファーをフレームのバッファーリストの最後への移動も行う。この値はバッファーを kill することなく `switch-to-prev-buffer`がこのバッファーに再び切り替えさせないようにする、もっとも信頼できる解決手段を提供する。

`kill` これは *window*のバッファーを kill することを意味する。

(interactive)

```
(display-buffer-record-window type window buffer)

(set-window-buffer window buffer)

(set-window-prev-buffers window nil)
```

ウィンドウのヒストリーを `nil`にセットすることにより？、将来の `quit-window`呼び出しが一緒にウィンドウを削除することが保証される。

以下のオプションは quit すべきウィンドウ、あるいはバリーすべきバッファーをもつウィンドウを 1 つだけ含むフレームを処理する方法を指定します。

`frame-auto-hide-function` [User Option]

このオプションで指定された関数は自動的にフレームを隠すために呼び出される。この関数はフレームを唯一の引数として呼び出される。

ここで指定される関数は選択されたウィンドウが専用 (dedicated) であり、かつバリーされるバッファーを表示しているときに `bury-buffer` (Section 27.8 [Buffer List], page 600 を参照) から呼び出される。また quit されるウィンドウのフレームがそのウィンドウのバッファーを表示するために特別に作成されたフレームで、かつそのバッファーが kill されないときにも `quit-restore-window` (上記) から呼び出される。

デフォルトでは `iconify-frame` (Section 29.11 [Visibility of Frames], page 736 を参照) を呼び出す。かわりにフレームをディスプレイから削除する `delete-frame` (Section 29.7 [Deleting Frames], page 730 を参照)、フレームを不可視にする `make-frame-invisible`、フレームを変更せずに残す `ignore`、またはフレームを唯一の引数とする任意の関数のいずれかを指定できる。

このオプションで指定された関数は指定されたフレームが生きたウィンドウただ 1 つを含み、かつ同一端末上に少なくとも 1 つ他のフレームが存在する場合のみ呼び出されることに注意。

特定のフレームにたいしてここで指定した値は、そのフレームのフレームパラメーター `auto-hide-function`でオーバーライドされるかもしれない (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照)。

28.17 サイドウィンドウ

サイドウィンドウ (side window) とは、フレームのルートウィンドウ (Section 28.2 [Windows and Frames], page 609 を参照) の 4 辺のいずれかに位置する特別なウィンドウです。これは実際にはフレームのルートウィンドウ領域は、メインウィンドウとメインウィンドウ周囲のいくつかのサイドウィンドウに分割されることを意味します。メインウィンドウは “通常” の生きたウィンドウ、またはすべての通常ウィンドウを含んだ領域を指定します。

この形式のもっともシンプルな使用では、サイドウィンドウによって特定のバッファーを常にフレームの同一領域に表示することが可能です。したがってこれは `display-buffer-at-bottom` (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照) によって提供される概念を残りのサイド (訳注: 下辺以外) に一般化したものとみなすことができます。しかし適切にカスタマイズすることにより、いわゆる統合開発環境 (IDE) で見い出されるようなフレームレイアウトを提供するためにも、サイドウィンドウを使用できます。

28.17.1 サイドウィンドウへのバッファーの表示

以下の `display-buffer` (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照) 用のアクション関数は特定のバッファーを表示するためにサイドウィンドウの作成や再利用を行います。

`display-buffer-in-side-window` *buffer alist* [Function]

この関数は選択されたフレームのサイドウィンドウに *buffer*を表示する。*buffer*の表示に使用したウィンドウをリターンする。そのようなウィンドウが見つからない、または作成できなければ `nil`をリターンする。

*alist*は `display-buffer`の場合と同様のシンボルと値からなる連想リスト。*alist*内で以下のシンボルはこの関数では特別な意味をもつ:

`side` ウィンドウを配置するフレームのサイド (側面) を表す。有効な値は `left`、`top`、`right`、`bottom`。未指定ならウィンドウはフレームの底部 (bottom) に配置される。

`slot` 指定したウィンドウを配置するサイドのスロットを表す。値 0 は指定したサイドのおおよそ中央にウィンドウを配置する。負の値は中央スロットの前 (上方か左方)、正の値は中央スロットの後 (下方か右方) を意味する。つまり特定のサイド上にあるすべてのウィンドウは `slot`の値の順になる。未指定ならウィンドウは指定したサイドの中央に配置される。

同一サイドの同一スロットに 2 つ以上の異なるバッファーすると、最後に表示されたバッファーが対応するウィンドウに表示される。したがってバッファー間で同じサイドウィンドウを共有するためにスロットを使用できる。

この関数はパラメーター `window-side`と `window-slot`をインストールして永続化する (Section 28.27 [Window Parameters], page 688 を参照)。*alist*内の `window-parameters`エントリーを通じて明示的に提供されない限り、他のウィンドウパラメーターは何もインストールしない。

デフォルトではサイドウィンドウは `split-window` (Section 28.6 [Splitting Windows], page 622 を参照) で分割できません。さらにサイドウィンドウはアクションのターゲットとして明示的に指定されていなければ、バッファーディスプレイアクション (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照) によって再利用や分割されることはありません。`delete-other-windows`はサイドウィンドウをフレーム上で唯一ののウィンドウにできないことにも注意してください (Section 28.7 [Deleting Windows], page 625 を参照)。

一度セットアップされれば、サイドウィンドウにたいする `switch-to-prev-buffer`と `switch-to-next-buffer` (Section 28.14 [Window History], page 659 を参照) の挙動も変更されます。特にサイドウィンドウでは、これらのコマンドは以前にそのウィンドウに表示されたことのないバッファーの表示を抑制します。さらにすでにサイドウィンドウに表示されているバッファーの、通常の非サイドウィンドウでの表示も抑制します。後者ルールの明記するべき例外は、アプリケーションがバッファー表示後にバッファーのローカル変数をリセットしたときに発生します。

28.17.2 サイドウィンドウのオプションと関数

以下のオプションはサイドウィンドウ配置において更なる制御を提供します。

`window-sides-vertical` [User Option]

非 `nil`ならフレームの左右のサイドウィンドウはフレームの全高さを占有する。それ以外ならフレームの上下のサイドウィンドウはフレームの全幅を占有する。

`window-sides-slots` [User Option]
このオプションはフレーム各サイドのサイドウィンドウの最大数を指定する。値は各フレームのサイドウィンドウのスロット数を左辺、上辺、右辺、下辺の順で指定する 4 要素のリスト。要素が数値なら対応するサイドで表示できる最大のウィンドウ数を意味する。要素が `nil`ならそのサイドのスロット数に制限がないことを意味する。

指定された値のいずれかが 0 なら、対応するサイドへのウィンドウは作成できない。この場合には `display-buffer-in-side-window`はエラーをシグナルしないが `nil`をリターンする。指定した値が単にサイドウィンドウの追加作成を禁止する場合には、そのサイド上にあるもっとも適したウィンドウが再利用されて、それに応じてウィンドウの `window-slot`も変更される。

`window-sides-reversed` [User Option]
このオプションは上や下のサイドウィンドウが逆順で表示されるかどうかを指定する。`nil`ならフレームの上や下にあるサイドウィンドウはスロット値の増加にともない常に左から右に描画される。`t`なら描画順は反転してフレームの上や下にあるサイドウィンドウはスロット値の増加にともない右から左に描画される。

これが `bidi`なら描画順はフレームのメインウィンドウ内でもっとも最近選択されたウィンドウに表示されるバッファーの `bidi-paragraph-direction` (Section 39.26 [Bidirectional Display], page 1085 を参照) の値が `right-to-left`の場合のみ逆順になる。このウィンドウを見つけるのが困難なときがあるかもしれないので、別のウィンドウ選択時に意図せず描画順が変更されることを避けるために経験則が使用される。

フレームの左右にあるサイドウィンドウのレイアウトは、この変数の値から影響を受けない。

以下の関数はサイドウィンドウをもつフレームのメインウィンドウをリターンします。

`window-main-window` **&optional** *frame* [Function]
この関数は指定した *frame*のメインウィンドウをリターンする。オプション引数 *frame*は生きたフレームでなければならず、デフォルトは選択されたフレーム。

*frame*にサイドウィンドウがなければ *frame*のルートウィンドウをリターンする。それ以外なら *frame*上にある他のすべての非サイドウィンドウの系統元であるようなサイドウィンドウではない内部ウィンドウ、または *frame*の単一の生きた非サイドウィンドウのいずれか。フレームのメインウィンドウは `delete-window`で削除できないことに注意。

以下のコマンドは指定したフレーム上にあるすべてのサイドウィンドウの外観を手軽にトグル (toggle: オンとオフを切り替える) できます。

`window-toggle-side-windows` **&optional** *frame* [Command]
この関数は指定された *frame*のサイドウィンドウをトグルする。オプション引数 *frame*は生きたフレームでなければならずデフォルトは選択された *frame*。

*frame*に少なくとも 1 つのサイドウィンドウがあれば、このコマンドは *frame*のルートウィンドウの状態を *frame*の `window-state`パラメーターに保存して、その後に *frame*のサイドウィンドウをすべて削除する。

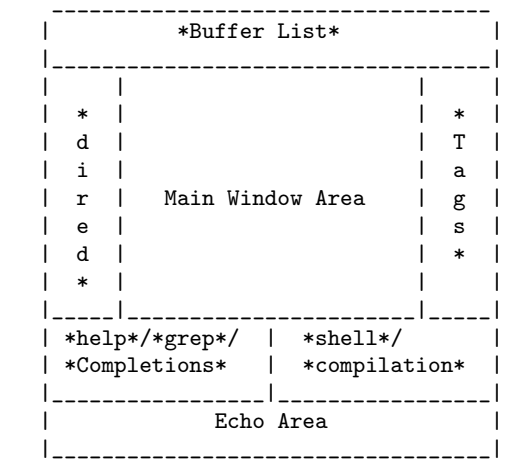
*frame*にサイドウィンドウがなく、しかし `window-state`パラメーターがあれば、このコマンドはパラメーターの値を使用して *frame*のメインウィンドウを残しつつ *frame*のサイドウィンドウをリストアする。

*frame*にサイドウィンドウがなく保存した状態も見つからなければエラーをシグナルする。

28.17.3 サイドウィンドウによるフレームのレイアウト

サイドウィンドウは統合開発環境 (IDE) が提供するような、より複雑なフレームレイアウトの作成に使用できます。そのようなレイアウトでは通常の編集アクティビティが行われるのはメインウィンドウ領域になります。サイドウィンドウは通常の意味においての編集は意図していません。それよりはカレント編集アクティビティを補足するためのファイルやタグやバッファーのリスト、ヘルプ情報、検索や grep の結果、シェル出力などの情報の表示を意図しています。

そのようなフレームのレイアウトは以下のような外観になるでしょう:



以下は上図フレームレイアウトを作成するコードをセットアップするために `display-buffer-in-side-window`とともにウィンドウパラメーター (see Section 28.27 [Window Parameters], page 688) を使用する方法を説明するための例です。

```
(defvar parameters
  '(window-parameters . ((no-other-window . t)
                         (no-delete-other-windows . t))))

(setq fit-window-to-buffer-horizontally t)
(setq window-resize-pixelwise t)

(setq
 display-buffer-alist
 `(("\\*Buffer List\\*" display-buffer-in-side-window
    (side . top) (slot . 0) (window-height . fit-window-to-buffer)
    (preserve-size . (nil . t)) ,parameters)
   ("\\*Tags List\\*" display-buffer-in-side-window
    (side . right) (slot . 0) (window-width . fit-window-to-buffer)
    (preserve-size . (t . nil)) ,parameters)
   ("\\*\\(?:help\\|grep\\|Completions\\)\\*"
    display-buffer-in-side-window
    (side . bottom) (slot . -1) (preserve-size . (nil . t))
    ,parameters)
   ("\\*\\(?:shell\\|compilation\\)\\*" display-buffer-in-side-window
    (side . bottom) (slot . 1) (preserve-size . (nil . t))
    ,parameters)))
```

これは固定化された名前をもつバッファーにたいして`display-buffer-alist`エントリー (Section 28.13.1 [Choosing Window], page 640 を参照) を指定します。特にフレーム上辺に高さ調節可能な`*Buffer List*`と、フレーム右辺に幅調節可能な`*Tags List*`の表示を要求します。さらにフレーム下辺左側のウィンドウでバッファー`*help*`と`*grep*`と`*Completions*`の共有、フレーム下辺右側のウィンドウではバッファー`*shell*`と`*compilation*`の表示を要求します。

ウィンドウの水平調節を可能にするためにオプション`fit-window-to-buffer-horizontally`が非`nil`値をもたなければならないことに注意してください。フレームの上下にあるサイドウィンドウの高さとフレームの左右にあるサイドウィンドウの幅を保持するためのエントリーも追加しています。フレーム最大化時にサイドウィンドウがそれに応じたサイズを維持することを保証するために、変数`window-resize-pixelwise`に非`nil`値をセットしています。Section 28.4 [Resizing Windows], page 617 を参照してください。

最後のフォームではこれらの各ウィンドウに`no-other-window`パラメーターをインストールすることによって、作成したサイドウィンドウにたいする *C-x o* を介してアクセスできないことも保証しています。さらにこれらの各ウィンドウに`no-delete-other-windows`パラメーターをインストールして、*C-x 1* によるサイドウィンドウの削除ができないことを保証しています。

`dired`バッファーは固定化された名前をもたないので、フレーム左辺の細いディレクトリーバッファーを表示するためにスペシャル関数`dired-default-directory-on-left`を使用しています。

```
(defun dired-default-directory-on-left ()
  "左側サイドウィンドウに詳細を隠して`default-directory'を表示する。"
  (interactive)
  (let ((buffer (dired-noselect default-directory)))
    (with-current-buffer buffer (dired-hide-details-mode t))
    (display-buffer-in-side-window
     buffer `((side . left) (slot . 0)
              (window-width . fit-window-to-buffer)
              (preserve-size . (t . nil)) ,parameters))))
```

これまでのフォームを評価して任意の順序で *M-x list-buffers*、*C-h f*、*M-x shell*、*M-x list-tags*、*M-x dired-default-directory-on-left*を評価すれば上図のフレームレイアウトが再作成されるはずです。

28.18 アトミックウィンドウ

アトミックウィンドウ (atomic window) とは少なくとも 2 つ以上の生きたウィンドウから組成された矩形領域であり以下のような特性をもちます:

- アトミックウィンドウの構成にたいして関数 `split-window` (Section 28.6 [Splitting Windows], page 622 を参照) を適用するとアトミックウィンドウ外部に新たなウィンドウの作成を試みる。
- アトミックウィンドウの構成にたいして関数 `delete-window` (Section 28.7 [Deleting Windows], page 625 を参照) を適用するとアトミックウィンドウ全体の削除を試みる。
- アトミックウィンドウの構成にたいして関数`delete-other-windows` (Section 28.7 [Deleting Windows], page 625 を参照) を適用するとアトミックウィンドウによるフレームの充填、またはメインウィンドウ化を試みる (Section 28.17 [Side Windows], page 664 を参照)。

これはウィンドウ構造を変更する基本的な関数グループはアトミックウィンドウを生きたウィンドウのように扱い、それゆえアトミックウィンドウ内部の構造を保持することを意味しています。

アトミックウィンドウはファイルのリビジョン間の差異、異なる言語やマークアップでの同一テキストの表示のように、関連するバッファーを特定のマナーにしたがって同時に表示する際にのみ有意義なウィンドウレイアウトの構築と保持に有用です。特定のウィンドウの側面上のバーでウィンドウの情報を永続的に表示するためにも使用できます。

アトミックウィンドウは予約済みのウィンドウパラメーター `window-atom` (Section 28.27 [Window Parameters], page 688 を参照) の助けを借りて実装されていて、内部ウィンドウ (Section 28.1 [Basic Windows], page 608 を参照) はアトミックウィンドウのルートウィンドウと呼ばれます。同じアトミックウィンドウの一部であるようなすべてのウィンドウは共通の祖先としてこのルートウィンドウをもち、`window-atom`パラメーターに非 `nil`が割り当てられます。

以下の関数は指定したウィンドウを一部とするアトミックウィンドウのルートをリターンします:

`window-atom-root` **&optional** *window* [Function]
: この関数は *window*を一部とするようなアトミックウィンドウのルートウィンドウをリターンする。*window*には有効なウィンドウを指定しなければならず、デフォルトは選択されたウィンドウ。*window*がアトミックウィンドウの一部でなければ `nil`をリターンする。

アトミックウィンドウを新たに作成するには既存の内部ウィンドウを選んで以下の関数を適用するのがもっともシンプルなアプローチです:

`window-make-atom` *window* [Function]
この関数は *window*をアトミックウィンドウに変換する。*window*には内部ウィンドウを指定しなければならない。この関数が行うのは *window*の子孫それぞれの `window-atom`パラメーターに `t`をセットすることだけである。

既存の生きたウィンドウから新たにアトミックウィンドウを作成したり、既存のアトミックウィンドウに新たにウィンドウを追加するには、以下のバッファーディスプレイアクション関数を使用できます (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照):

`display-buffer-in-atom-window` *buffer alist* [Function]
この関数は既存のウィンドウと組み合わせてアトミックウィンドウを形成することになる新たなウィンドウ内で *buffer*を表示する。既存のウィンドウがすでにアトミックウィンドウの一部なら、そのアトミックウィンドウに新たなウィンドウを追加する。

*alist*にはシンボルと値からなる連想リストを指定する。以下は特別な意味をもつシンボル:

`window` このような要素は新たなウィンドウを組み合わせる既存のウィンドウを指定する。内部ウィンドウを指定すると、そのウィンドウのすべての子ウィンドウもアトミックウィンドウの一部になる。ウィンドウを何も指定しなければ新たなウィンドウは選択されたウィンドウの兄弟ウィンドウになる。既存ウィンドウの `window-atom`パラメーターはそれが生きたウィンドウであり、かつ `window-atom`がすでにセット済みでなければ `main`にセットされる。

`side` このような要素は既存ウィンドウで新たなウィンドウが配置されるサイドを表す。有効な値は `below`、`right`、`above`、`left`。デフォルトは `below`。この値は新たなウィンドウの `window-atom`パラメーターにセットされる。

リターン値は新たなウィンドウ、ウィンドウ作成に失敗すると `nil`。

非 `nil`である限り `window-atom`パラメーターの値は問題ではないことに注意してください。`display-buffer-in-atom-window`が割り当てる値は、関数の適用後に元のウィンドウと新たなウィンドウを簡単に取得することだけが目的です。`display-buffer-in-atom-window`が割り当てるウィンドウパラメーターは `window-atom`パラメーターだけであることにも注意してください。それ以外のパラメーターは *alist*内の `window-parameters`エントリーを介してアプリケーションがセットする必要があります。

アトミックウィンドウはそれを構成するウィンドウのいずれかが削除された際には存在を終えます。アトミックウィンドウを手動で分解するためには、それを構成するウィンドウ (アトミックウィンドウのルートウィンドウと子孫) の `window-atom`パラメーターをリセットしてください。

以下のスニペットコードを単一ウィンドウのフレームに適用すると、最初に選択されたウィンドウを分割して選択されたウィンドウと新たなウィンドウの親をルートとしてアトミックウィンドウを構成します。それからフレーム下辺にある新たなウィンドウでバッファー`*Messages*`を表示して、新たなウィンドウを作成したアトミックウィンドウの一部にします。

```
(let ((window (split-window-right)))
  (window-make-atom (window-parent window))
  (display-buffer-in-atom-window
   (get-buffer-create "*Messages*")
   `((window . ,(window-parent window)) (window-height . 5))))
```

この時点においてフレーム内の任意のウィンドウで *C-x 2* をタイプすると、フレーム下辺に新たなウィンドウが作成されます。かわりに *C-x 3* をタイプすれば新たなウィンドウはフレーム右辺に配

置されるでしょう。いずれのケースでもここでアトミックウィンドウ内の任意のウィンドウで `C-x 1` をタイプすると、新たなウィンドウだけが削除されます。アトミックウィンドウ内の任意のウィンドウで `C-x 0` をタイプすればフレームは新たなウィンドウで充填されるでしょう。

28.19 ウィンドウとポイント

それぞれのウィンドウは独自のポイント値 (Section 30.1 [Point], page 755 を参照) をもち、同じバッファーを表示する他のウィンドウの間でも、ポイント値はそれぞれ独立しています。これは 1 つのバッファーを複数ウィンドウで表示するのに有用です。

- ウィンドウポイント (window point) は、ウィンドウが最初に作成されたときに設定される。ウィンドウポイントはバッファーのポイント、またはそのバッファーからオープンされたウィンドウがあればそのウィンドウのウィンドウポイントにより初期化される。
- ウィンドウの選択により、ウィンドウのポイント値からそのバッファーのポイント値がセットされる。反対にウィンドウの非選択により、ウィンドウのポイント値にバッファーのポイント値がセットされる。つまり与えられたバッファーを表示するウィンドウ間で切り替えを行ったときには、そのバッファーでは選択されたウィンドウのポイント値が効力をもつが、他のウィンドウのポイント値はそのウィンドウに格納される。
- 選択されたウィンドウがカレントバッファーの表示を続ける限り、そのウィンドウのポイントとバッファーのポイントは常に連動して移動して等しく保たれる。

デフォルトでは Emacs は塗りつぶした矩形ブロックで各ウィンドウのポイント位置にカーソルを表示します。あるウィンドウでユーザーが別のバッファーに切り替えた際には、そのウィンドウのカーソルはそのバッファーのポイント位置に移動します。display 文字列やイメージ等の何らかのディスプレイ要素により正確な位置が隠れている場合には、Emacs はその display 要素の直前か直後にカーソルを表示する。

`window-point` **&optional** *window* [Function]
この関数は *window*内のカレントのポイント位置をリターンする。選択されていないウィンドウでは、そのウィンドウが選択された場合の、(そのウィンドウのバッファーの) ポイント値である。*window*にたいするデフォルトは選択されたウィンドウ。

*window*が選択されたウィンドウのときのリターン値は、そのウィンドウのバッファーのポイント値。厳密にはすべての `save-excursion`フォームの外側のトップレベルのポイント値のほうがより正確であろう。しかしこの値は見つけるのが困難である。

`set-window-point` *window position* [Function]
この関数は *window*内のポイントを *window*のバッファー内の位置 *position*に配置する。リターン値は *position*。

*window*が選択されていれば単に *window*内で `goto-char`を行う。

`window-point-insertion-type` [Variable]
この変数は `window-point`のマーカー挿入型 (Section 31.5 [Marker Insertion Types], page 771 を参照) を指定する。デフォルトは `nil`で、`window-point`は挿入されたテキストの後に留まるだろう。

28.20 ウィンドウの開始位置と終了位置

ウィンドウはそれぞれバッファー位置を追跡するために、バッファー内で表示を開始すべき位置を指定するマーカーを保守しています。この位置はそのウィンドウの *display-start*(表示開始)、または単

に *start(*開始*)* と呼ばれます。この位置の後の文字がウィンドウの左上隅に表示される文字となります。これは通常はテキスト行の先頭になりますが必須ではありません。

ウィンドウやバッファーの切り替え後やいくつかのケースにおいては、ウィンドウが行の途中で開始される場合に Emacs がィンドウの開始を行の開始に調整します。これは行中で無意味な位置のウィンドウ開始のまま特定の操作が行われるのを防ぐためです。この機能は Lisp モードのコマンドを使用して実行することによりある種の Lisp コードをテストする場合には、それらのコマンドがこの再調整を誘発してしまうので邪魔かもしれません。そのようなコードをテストするためには、それをコマンド内に記述して何らかのキーにバインドしてください。

`window-start` **&optional** *window* [Function]

この関数はウィンドウ *window*の表示開始位置をリターンする。*window*が `nil`なら選択されたウィンドウが使用される。

ウィンドウを作成したり他のバッファーをウィンドウ内に表示する際、display-start 位置は同じバッファーにたいしてもっとも最近に使用された display-start 位置、そのバッファーがそれをもたなければ `point-min`にセットされる。

ポイントがスクリーン上に確実に現れるように、再表示は window-start 位置を更新する (前の再表示以降に window-start 位置を明示的に指定していない場合)。再表示以外に window-start 位置を自動的に変更するものはない。ポイントを移動した場合には、次の再表示後までポイントの移動に応じて window-start が変更されることを期待してはならない。

`window-group-start` **&optional** *window* [Function]

この関数は `window-start`と同様だが、*window*がウィンドウグループ ([Window Group], page 609 を参照) の一部なら、`window-group-start`はグループ全体の開始位置をリターンする点が異なる。この条件はバッファーローカル変数 `window-group-start-function`に関数がセットされている際に保持される。この場合には、`window-group-start`はその関数を単一の引数 *window*で呼び出して結果をリターンする。

`window-end` **&optional** *window update* [Function]

この関数は *window*のバッファーの最後を表示する位置をリターンする。*window*にたいするデフォルトは選択されたウィンドウ。

バッファーテキストの単なる変更やポイントの移動では `window-end`がリターンする値は更新されない。この値は Emacs が再表示を行って、妨害されることなく再表示が完了したときのみ更新される。

*window*の最後の再表示が妨害されて完了しなかったら、Emacs はそのウィンドウ内の表示の end 位置を知らない。関数はこの場合は `nil`をリターンする。

*update*が非 `nil`なら、`window-end`は `window-start`のカレント値にもとづき、どこが表示の end なのか最新の値をリターンする。以前に保存された位置の値がまだ有効なら、`window-end`はその値をリターンする。それ以外はバッファーのテキストをスキャンして正しい値を計算する。

たとえ *update*が非 `nil`であっても、`window-end`はポイントが画面外に移動した場合に実際の再表示が行うような表示のスクロールを試みない。これは `window-start`の値を変更しない。これは実際にはスクロールが要求されない場合の表示されたテキストの end がどこかを報告する。リターンされる位置は部分的に可視なだけかもしれないことに注意。

`window-group-end` **&optional** *window update* [Function]

この関数は `window-end`と同様だが、*window*がウィンドウグループ ([Window Group], page 609 を参照) の一部なら、`window-group-end`はグループ全体の終了位置をリターンする点が異なる。この条件はバッファーローカル変数 `window-group-end-function`に関数が

セットされている際に保持される。この場合には、`window-group-end`はその関数を 2 つの引数 *window*と *update*で呼び出して結果をリターンする。引数 *update*は `window-end`の場合と同じ意味をもつ。

`set-window-start` *window position* **&optional** *noforce* [Function]

この関数は *window*の display-start 位置を *window*のバッファーの *position*にセットする。リターン値は *position*。

表示ルーチンはバッファーが表示されたときにポイント位置が可視になることを強要する。通常これらは内部ロジックに応じてポイントを可視にするために display-start 位置を選択 (および必要ならつまりウィンドウをスクロール) する。しかしこの関数で *noforce*に `nil`を使用して start 位置を指定した場合は、たとえポイントを画面外になるような場所に配置したとしても、*position*での表示開始を望んでいることを意味する。これによりポイントが画面外に配置されると、表示ルーチンはポイントをウィンドウ内の中央行の左マージンに移動しようと試みる。

たとえばポイントが 1 のときにウィンドウの start を次行の開始 37 にセットすると、ポイントはウィンドウの最上端より上になるだろう。表示ルーチンは再表示が発生したときにポイントが 1 のままならポイントを動かすことになる。以下は例:

```
;; 以下は式 set-window-start実行前
;;   'foo'の様子

---------- Buffer: foo ----------
*This is the contents of buffer foo.
2
3
4
5
6
---------- Buffer: foo ----------

(set-window-start
 (selected-window)
 (save-excursion
   (goto-char 1)
   (forward-line 1)
   (point)))
⇒ 37

;; 以下は式 set-window-start実行後の
;;   'foo'の様子
---------- Buffer: foo ----------
2
3
*4
5
6
---------- Buffer: foo ----------
```

ポイントを可視 (つまり完全に可視なスクリーン行内にポイントを配置) にする試みが失敗すると、表示ルーチンは要求された window-start 位置を無視して、とにかく新しい位置を計算する。したがってこの関数を呼び出す Lisp プログラムが信頼できる結果を得るためには、表示が *position*で始まるウィンドウ内部に常にポイントを移動すること。

*noforce*が非 `nil`で、かつ次回の再表示でポイントが画面外に配される場合、再表示はポイントと協調して機能する位置となるような新たな window-start を計算するので、*position*は使用されない。

`set-window-group-start` *window position* **&optional** *noforce* [Function]

この関数は `set-window-start`と同様だが、*window*がウィンドウグループ ([Window Group], page 609 を参照) の一部なら、`set-window-group-start`はグループ全体の開始位置をリターンする点が異なる。この条件はバッファーローカル変数 `set-window-group-start-function`に関数がセットされている際に保持される。この場合には、`set-window-group-start`はその関数を 3 つの引数 *window*、*position*、*noforce*で呼び出して結果をリターンする。この関数の引数 *position*と *noforce*は `set-window-start`の場合と同じ意味をもつ。

`pos-visible-in-window-p` **&optional** *position window partially* [Function]

この関数は *window*内の *position*が画面上カレントで可視のテキスト範囲内にあれば非 `nil`、*position*が表示範囲のスクロール外にあれば `nil`をリターンする。*partially*が `nil`なら部分的に不明瞭な位置は可視とは判断されない。引数 *position*のデフォルトは *window*内のポイントのカレント位置、*window*のデフォルトは選択されたウィンドウ。*position*が `t`なら、それは *window*の最後に可視だった行の位置、または EOB(end-of-buffer: バッファー終端位置のいずれか前方になる位置をチェックすることを意味する。

この関数は垂直スクロールだけを考慮する。*position*が表示範囲外にある理由が、*window*が水平にスクロールされただけなら、いずれにせよ `pos-visible-in-window-p`は非 `nil`をリターンする。Section 28.23 [Horizontal Scrolling], page 680 を参照のこと。

*position*が可視で *partially*が `nil`なら、`pos-visible-in-window-p`は `t`をリターンする。*partially*が非 `nil`で *position*以降の文字が完全に可視なら、`(`*x y*`)`という形式のリストをリターンする。ここで *x*と *y*はウィンドウの左上隅からの相対的なピクセル座標。*position*以降の文字が完全に可視ではなければ、拡張された形式のリスト `(`*x y rtop rbot rowh vpos*`)`をリターンする。ここで *rtop*と *rbot*は *position*でウィンドウ外となった上端と下端のピクセル数、*rowh*はその行の可視な部分の高さ、*vpos*はその行の垂直位置 (0 基準の行番号) を示す。

以下は例:

```
;; ポイントが画面外なら recenter する
(or (pos-visible-in-window-p
     (point) (selected-window))
    (recenter 0))
```

`pos-visible-in-window-group-p` **&optional** *position window partially* [Function]

この関数は `pos-visible-in-window-p`と同様だが、*window*がウィンドウグループ ([Window Group], page 609 を参照) の一部なら、`pos-visible-in-window-group-p`は *window*単独ではなく、グループ全体で *pos*の可視性をテストする点が異なる。この条件はバッファーローカル変数 `pos-visible-in-window-group-p-function`に関数がセットされている際に保持される。この場合には、`pos-visible-in-window-group-p`はその関数を 3 つの引数

position、*window*、*partially*で呼び出して結果をリターンする。この関数の引数 *position*と *partially*は `pos-visible-in-window-p`の場合と同じ意味をもつ。

`window-line-height` **&optional** *line window* [Function]

この関数は *window*内のテキスト行 *line*の高さをリターンする。*line*が `header-line`、`mode-line`、`window-line-height`のいずれかなら、そのウィンドウの対応する行についての情報をリターンする。それ以外では、*line*は 0 から始まるテキスト行番号。負数ならそのウィンドウの end から数える。*line*にたいするデフォルトは *window*内のカレント行、*window*にたいするデフォルトは選択されたウィンドウ。

表示が最新でなければ `window-line-height`は `nil`をリターンする。その場合には関連する情報を入手するために `pos-visible-in-window-p`を使用できる。

指定された *line*に対応する行がなければ、`window-line-height`は `nil`をリターンする。それ以外では、リスト `(`*`height vpos ypos offbot`*`)`をリターンする。ここで *height*はその行の可視部分のピクセル高さ、*vpos*と *ypos*は最初のテキスト行上端からのその行への相対的な垂直位置の行数とピクセル数、*offbot*はそのテキスト行下端のウィンドウ外のピクセル数。(最初の) テキスト行上端にウィンドウ外のピクセルがある場合には *ypos*は負となる。

28.21 テキスト的なスクロール

テキスト的なスクロール *(textual scrolling)* とは、ウィンドウ内のテキストを上や下に移動することを意味します。これはそのウィンドウの display-start を変更することにより機能します。これはポイントを画面上に維持するために `window-point`の値も変更するかもしれません (Section 28.19 [Window Point], page 671 を参照)。

テキスト的なスクロールの基本的な関数は、(前方にスクロールする) `scroll-up`、および (後方にスクロールする) `scroll-down`です。これらの関数の名前の “up” と “down” は、バッファーテキストのそのウィンドウにたいする相対的な移動方向を示しています。そのテキストが長いロール紙に記述されていて、スクロールコマンドはその上を上下に移動すると想像してみてください。つまりバッファーの中央に注目している場合には、繰り返して `scroll-down`を呼び出すと最終的にはバッファーの先頭を目にすることになるでしょう。

これは残念なことに時折混乱を招きます。なぜならある人はこれを逆の慣習にもとづいて考える傾向があるからです。彼らはテキストがその場所に留まりウィンドウが移動して、“down” コマンドによりバッファー終端に移動するだろうと想像します。この慣習はそのようなコマンドが現代風のキーボード上の `PageDown`という名前のキーにバインドされているという事実と一致しています。

選択されたウィンドウ内で表示されているバッファーがカレントバッファーでなければ、(`scroll-other-window`以外の) テキスト的スクロール関数の結果は予測できません。Section 27.2 [Current Buffer], page 590 を参照してください。

(たとえば大きなイメージがある等で) ウィンドウにウィンドウの高さより高い行が含まれる場合には、スクロール関数は部分的に可視な行をスクロールするためにそのウィンドウの垂直スクロール位置を調整します。Lisp 呼び出し側は変数 `auto-window-vscroll`を `nil`にバインドすることにより、この機能を無効にできます (Section 28.22 [Vertical Scrolling], page 679 を参照)。

`scroll-up` **&optional** *count* [Command]

この関数は選択されたウィンドウ内で *count*行前方にスクロールする。

*count*が負ならかわりに後方へスクロールする。*count*が `nil` (または省略) ならスクロールされる距離は、そのウィンドウのテキストエリアの高さより小さい `next-screen-context-lines`となる。

この関数は選択されたウィンドウがそれ以上スクロールできなければエラーをシグナルして、それ以外は nilをリターンする。

scroll-down &optional *count* [Command]
この関数は選択されたウィンドウ内で *count*行後方にスクロールする。

*count*が負ならかわりに前方へスクロールする。それ以外の点ではこれは scroll-upと同様に振る舞う。

scroll-up-command &optional *count* [Command]
これは scroll-upと同様に振る舞うが選択されたウィンドウがそれ以上スクロールできず、かつ変数 scroll-error-top-bottomの値が tなら、かわりにそのバッファーの終端への移動を試みる。ポイントがすでに終端にあればエラーをシグナルする。

scroll-down-command &optional *count* [Command]
これは scroll-downと同様に振る舞うが選択されたウィンドウがそれ以上スクロールできず、かつ変数 scroll-error-top-bottomの値が tなら、かわりにそのバッファーの先頭への移動を試みる。ポイントがすでに先頭にあればエラーをシグナルする。

scroll-other-window &optional *count* [Command]
この関数は他のウィンドウ内のテキストを上方に *count*行スクロールする。*count*が負か nilなら scroll-upのように処理される。

変数 other-window-scroll-bufferにバッファーをセットすることにより、どのバッファーをスクロールするかを指定できる。そのバッファーが表示されていなければ、scroll-other-windowはそれを何らかのウィンドウにそれを表示する。

選択されたウィンドウがミニバッファーのとき、次ウィンドウは通常はそのウィンドウの直上最左のウィンドウである。変数 minibuffer-scroll-windowをセットすることにより、スクロールする別のウィンドウを指定できる。この変数はミニバッファー以外のウィンドウが選択されているときは効果がない。これが非 nil、かつミニバッファーが選択されているときには other-window-scroll-bufferより優先される。[Definition of minibuffer-scroll-window], page 374 を参照のこと。

ミニバッファーがアクティブのとき選択されたウィンドウが下端右角のウィンドウなら、ミニバッファーが次ウィンドウになる。この場合には scroll-other-windowはミニバッファーのスクロールを試みる。ミニバッファーに含まれるのが 1 行だけならどこにもスクロールできないので、エコーエリアにメッセージ‘End of buffer’を瞬時表示した後にその行を再表示する。

scroll-other-window-down &optional *count* [Command]
この関数は他のウィンドウ内のテキストを下方に *count*行スクロールする。*count*が負か nilなら scroll-downのように処理される。それ以外の点においては scroll-other-windowと同様の方法で振る舞う。

other-window-scroll-buffer [Variable]
この変数が非 nilなら、それは scroll-other-windowがどのバッファーのウィンドウをスクロールするかを指定する。

scroll-margin [User Option]
このオプションはスクロールマージン (ポイントとウィンドウの上端/下端との最小行数) のサイズを指定する。ポイントがウィンドウの上端/下端からその行数になったとき、(可能なら) 再表示はポイントをそのマージン外のウィンドウ中央付近に移動するためにテキストを自動的にスクロールする。

`maximum-scroll-margin` [User Option]

この変数は`scroll-margin`の実効値をカレントウィンドウの行高さの割合に制限する。たとえばカレントウィンドウが 20 行で`maximum-scroll-margin`が 0.1 なら、`scroll-margin`がどれほど大きくてもスクロールマージンが 2 より大きくなることはない。

`maximum-scroll-margin`自体は最大値として 0.5 という値をもち、これはウィンドウの中央行 (ウィンドウが偶数行なら中央の 2 行) までカーソルを維持するためにマージンを大きくセットすることを可能にする。大きな値 (または 0.0 から 0.5 までの浮動小数点数以外の値) をセットするとデフォルト値 0.25 がかわりに使用される。

`scroll-conservatively` [User Option]

この変数はポイントがスクリーン外 (またはスクロールマージン内) に移動したときに自動的にスクロールを行う方法を指定する。値が正の整数 *n*なら再表示はそれが正しい表示範囲内にポイントを戻すなら、いずれかの方向に *n*行以下のテキストをスクロールする。この振る舞いは保守的なスクロール *(conservative scrolling)* と呼ばれる。それ以外ならスクロールは`scroll-up-aggressively`や`scroll-down-aggressively`のような他の変数の制御の下に通常の方法で発生する。

デフォルトの値は 0 でこれは保守的スクロールが発生し得ないことを意味する。

`scroll-down-aggressively` [User Option]

この変数の値は`nil`、または 0 から 1 までの小数点数 *f*であること。小数点数ならスクリーン上でポイントが置かれたとき下にスクロールする場所を指定する。より正確にはポイントがウィンドウ start より上という理由でウィンドウが下にスクロールされるときには、新たな start 位置がウィンドウ上端からウィンドウ高さの *f*の箇所にポイントが置かれるように選択される。より大きな *f*なら、より aggressive(積極的) にスクロールする。

その効果はポイントを中央に配置することであり、値`nil`は.5 と等価である。どのような方法によりセットされたときでも、この変数は自動的にバッファーローカルになる。

`scroll-up-aggressively` [User Option]

`scroll-up-aggressively`の場合と同様。値 *f*はポイントがウィンドウ下端からどれほどの位置に置かれるべきかを指定する。つまり、`scroll-up-aggressively`と同様に大きな値ではより aggressive(積極的) になる。

`scroll-step` [User Option]

この変数は`scroll-conservatively`の古い変種である。違いは値が *n*なら *n*以下の値ではなく、正確に *n*だけのスクロールを許容することである。この機能は`scroll-margin`とは共に機能しない。デフォルトは 0。

`scroll-preserve-screen-position` [User Option]

このオプションが`t`なら、スクロールによりポイントがウィンドウ外に移動したとき、Emacs は常にポイントがポイントの上下端ではなくカーソルがそのウィンドウ内の元の垂直位置に保たれるようポイントの調整を試みる。

値が非`nil`かつ非`t`なら、たとえスクロールコマンドによりポイントがウィンドウ外に移動していなくとも、Emacs はカーソルが同じ垂直位置に保たれるようにポイントを調整する。

このオプションはシンボルプロパティ`scroll-command`が非`nil`であるような、すべてのスクロールコマンドに影響する。

`next-screen-context-lines` [User Option]
この変数の値は全画面スクロールされたときに継続して残される行数を指定する。たとえば引数が `nil`の `scroll-up`はウィンドウ上端ではなく下端に残される行数でスクロールする。デフォルト値は 2。

`scroll-error-top-bottom` [User Option]
このオプションが `nil`(デフォルト) なら、それ以上のスクロールが不可能な際に `scroll-up-command`と `scroll-down-command`は単にエラーをシグナルする。

値が `t`なら、これらのコマンドはかわりにポイントをバッファーの先頭か終端 (スクロール方向に依存する) に移動する。ポイントがすでにその位置にある場合のみエラーをシグナルする。

`recenter` **&optional** *count redisplay* [Command]
この関数は選択されたウィンドウ内の指定された垂直位置にポイントを表示するようにウィンドウ内のテキストをスクロールする。これはテキストに応じたポイント移動を行わない。

*count*が非負の数なら、そのウィンドウ上端から *count*行下にポイントを含む行を配置する。*count*が負ならウィンドウ下端から上に数えるので、−1 はそのウィンドウ内で最後の利用可能な行となる。

*count*が `nil` (または非 `nil`のリスト) なら、`recenter`はポイントを含む行をウィンドウの中央に配置する。*count*と *redisplay*が非 `nil`なら、この関数は `recenter-redisplay`の値に応じてフレームを再描画するかもしれない。`recenter-redisplay`が非 `nil`の場合の効果を打ち消すために、したがって 2 つ目の引数の省略を使用できる。インタラクティブな呼び出しでは *redisplay*に非 ‘ nil ’ が渡される。

`recenter`がインタラクティブに呼び出されたときは raw プレフィックス引数が *count*となる。したがってプレフィックスとして *C-u*をタイプすると *count*に非 `nil`、*C-u 4*では *count*に 4 がセットされ te、後者ではカレント行を上端から 4 行目にセットする。

引数 0 では `recenter`はカレント行をウィンドウ上端に配置する。コマンド `recenter-top-bottom`はこれを達成するためにより簡便な方法を提供する。

`recenter-window-group` **&optional** *count* [Function]
この関数は `recenter`と同様だが、選択されたウィンドウがウィンドウグループ ([Window Group], page 609 を参照) の一部の際には、`recenter-window-group`がグループ全体をスクロールする点が異なる。この条件はバッファーローカル変数 `recenter-window-group-function`が関数にセットされている際に保持される。この場合には `recenter-window-group`はその関数を引数 *count*で呼び出して結果をリターンする。引数 *count*は `recenter`の場合と同様の意味をもつが、ウィンドウグループ全体に作用する。

`recenter-redisplay` [User Option]
この変数が非 `nil`なら引数 *redisplay*を `nil`、引数 *count*を非 `nil`で `recenter`を呼び出すことによりフレームを再描画する。デフォルト値は `tty`で、これはフレームが tty フレームのときだけフレームを再描画することを意味する。

`recenter-top-bottom` **&optional** *count* [Command]
デフォルトでは *C-l*にバインドされているこのコマンドは、`recenter`と同様に動作するが引数なしで呼び出されたときの動作が異なる。この場合には連続して呼び出すことにより変数 `recenter-positions`で定義されるサイクル順に応じてポイントを配置する。

`recenter-positions` [User Option]
これは`recenter-top-bottom`を引数なしで呼び出したときの挙動を制御する。デフォルト値は`(middle top bottom)`で、これは引数なしで`recenter-top-bottom`を連続して呼び出すとポイントをウィンドウの中央、上端、下端と巡回して配置することを意味する。

28.22 割り合いによる垂直スクロール

垂直フラクショナルスクロール *(vertical fractional scrolling)* とは、指定された値を行に乗ずることによりウィンドウ内のテキストを上下にシフトすることを意味します。たとえばEmacsはウィンドウより高さが大きいイメージやスクリーン行でこれを使用します。ウィンドウはそれぞれ決して0より小さくなることはない、垂直スクロール位置 *(vertical scroll position)* という数値をもっています。これはコンテンツを表示しているウィンドウにたいして、コンテンツをどこから表示開始 (raise) するかを指定します。ウィンドウのコンテンツの表示開始により一般的には上端の何行かのすべて、または一部が表示されなくなり他の何行かのすべて、または一部が下端に表示されるようになります。通常の値は0です。

垂直スクロール位置は行の通常高さ (デフォルトフォントの高さ) の単位で数えられます。したがって値が.5なら、それはウィンドウのコンテンツが通常行の半分の高さで上にスクロール、3.3なら通常行の3倍を若干超える高さで上にスクロールされていることを意味します。

垂直スクロールが覆い隠す (cover) のがどれほどの行断片 (fraction of a line) なのか、あるいは行数かはそれらの行に何が含まれるかに依存します。3.3という値により高い行やイメージの一部だけを画面外にスクロールできることもあれば、.5という値が非常に小さい高さの行を画面外にスクロールできることもあります。

`window-vscroll` **&optional** *window pixels-p* [Function]
この関数は *window*のカレントの垂直スクロール位置をリターンする。*window*のデフォルトは選択されたウィンドウ。*pixels-p*が非`nil`ならリターン値は通常行高さ単位ではなくピクセル単位で測定される。

```
(window-vscroll)
     ⇒ 0
```

`set-window-vscroll` *window lines* **&optional** *pixels-p* [Function]
この関数は *window*の垂直スクロール位置を *lines*にセットする。*window*が`nil`なら選択されたウィンドウが使用される。引数 *lines*は0または正であること。それ以外は0として扱われる。

実際の垂直スクロール位置は常にピクセルの整数に対応しなければならないため、指定した値はそれに応じて丸められる。

この丸め結果がリターン値となる。

```
(set-window-vscroll (selected-window) 1.2)
     ⇒ 1.13
```

*pixels-p*が非`nil`なら *lines*はピクセル数を指定する。この場合にはリターン値は *lines*。

`auto-window-vscroll` [Variable]
この変数が非`nil`なら関数`line-move`、`scroll-up`、`scroll-down`は、たとえば大きなイメージが存在する等でウィンドウ高さより高いディスプレイ行をスクロールするために垂直スクロール位置を自動的に変更するだろう。

28.23 水平スクロール

水平スクロール *(horizontal scrolling)* とは指定された通常文字幅の倍数でウィンドウ内のイメージを左右にシフトすることを意味します。ウィンドウはそれぞれ、決して 0 より小さくなることはない水平スクロール位置 *(horizontal scroll position)* という数値をもっています。これはコンテンツをどれほど左にシフトするかを指定します。ウィンドウのコンテンツを左にシフトすることにより一般的には左にある文字のすべて、または一部が表示されなくなり右にある文字のすべて、または一部が表示されることを意味します。通常の値は 0 です。

水平スクロール位置は通常の文字幅を単位として数えられます。したがって値が 5 なら、それはウィンドウのコンテンツは通常文字幅の 5 倍左にスクロールされることを意味します。左の何文字が表示されなくなるかは、それらの文字の文字幅とに依存していて、それは行ごとに異なります。

読み取りを行う際には内側のループ (inner loop) で横方向、外側のループ (outer loop) で上から下に読み取るため、水平スクロールの効果はテキスト的スクロールや垂直スクロールとは異なります。テキスト的スクロールは表示するためのテキスト範囲の選択を引き起こし、垂直スクロールはウィンドウコンテンツを連続して移動します。しかし水平スクロールはすべての行の一部をスクリーン外へスクロールします。

通常は水平スクロールは行われないので、ウィンドウ左端には最左列があります。この状態では右スクロールにより左端に新たに表示されるデータは存在しないので、右へのスクロールはできません。左スクロールによってテキストの 1 列目がウィンドウ端からウィンドウ外にスクロールされ、右端にはその前は切り詰められていた (truncated) 列が新たに表示されるので左へのスクロールはできます。ウィンドウが左へ非 0 の値で水平スクロールされていれば右スクロールしてそれを戻すことができますが、正味の水平スクロールが 0 に減少するまでの間のみ右スクロールができます。左へどれほどスクロールできるかに制限はありませんが、最終的にはすべてのテキストが左端の外に消えるでしょう。

`auto-hscroll-mode`がセットされている場合には、再表示はポイントが常に可視となることを保証するために必要に応じて水平スクロールを自動的に変更する。とはいえ依然として水平スクロール位置を明示的に指定するのは可能である。指定した値は自動スクロールの下限値としての役目を果たす (自動スクロールは指定された値より小さい列にウィンドウをスクロールしない)。

`auto-hscroll-mode`のデフォルト値は `t`です。これを `current-line`にセットするとカーソルのある行だけがポイントが可視になるように水平スクロールされて、ウィンドウの残りはスクロールされないか `scroll-left`と `scroll-right` (以下参照) にセットされた最小量だけスクロールされる自動水平スクロールの変種がアクティブになります。

`scroll-left` **&optional** *count set-minimum* [Command]

この関数は選択されたウィンドウを左 (*count*が負なら右) に *count*列スクロールする。*count* のデフォルトはウィンドウ幅から 2 を減じた値。

リターン値は `window-hscroll`(以下参照) がリターンする値と同じように、変更後に実際に左に水平スクロールされたトータル量。

基本方向が R2L(Section 39.26 [Bidirectional Display], page 1085 を参照) のパラグラフ内のテキストは、正の *count*値で `scroll-left`が呼び出された際には右へと移動するように、反対方向に移動することに注意。

ウィンドウを可能な限り右にスクロールした後は、左スクロールの合計が 0 であるような通常の位置に戻り、右へのそれ以上のスクロールの試みは効果をもたない。

*set-minimum*が非 `nil`なら新たなスクロール量は自動スクロールの下限値となる。つまり自動スクロールはこの関数がリターンする値より小さい列にウィンドウをスクロールしないだろう。インタラクティブに呼び出すと *set-minimum*に非 `nil`を渡す。

`scroll-right` **&optional** *count set-minimum* [Command]
この関数は選択されたウィンドウを右 (*count*が負なら左) に *count*列スクロールする。*count* のデフォルトはウィンドウ幅から 2 を減じた値。スクロール方向を除けばこれは `scroll-left` と同様に機能する。

`window-hscroll` **&optional** *window* [Function]
この関数は *window*の左への水平スクロールのトータル (左マージンを超えて左にスクロールされた *window*内のテキスト列数) をリターンする (R2L パラグラフでの値はかわりに右方向への総スクロール量となる)。*window*のデフォルトは選択されたウィンドウ。

リターン値が負になることは決してない。*window*で水平スクロールが行われていない場合 (これが通常) にはリターン値は 0。

```
(window-hscroll)
     ⇒ 0
(scroll-left 5)
     ⇒ 5
(window-hscroll)
     ⇒ 5
```

`set-window-hscroll` *window columns* [Function]
この関数は *window*の水平スクロールをセットする。*columns*の値はスクロール量を左マージン (R2L パラグラフでは右マージン) からの列数で指定する。引数 *columns*は 0 または正の数であること。そうでない場合ニは 0 とみなされる。小数点数の *columns*値は現在のところサポートされない。

シンプルに *M-:*を呼び出して評価する方法でテストすると、`set-window-hscroll`が機能していないように見えるかもしれないことに注意。ここで何が発生しているかというと、この関数は水平スクロール値をセットしてリターンするが、その後にポイントを可視にするために水平スクロールを調整するよう再表示が行なわれて、これが関数の行った処理をオーバーライドしている。この関数の効果は左マージンからポイントまでのスクロール量が、ポイントが可視のまま留まるように関数を呼び出すことにより観察できる。

リターン値は *columns*。

```
(set-window-hscroll (selected-window) 10)
     ⇒ 10
```

以下は与えられた位置 *position*が水平スクロールによりスクリーン外にあるかどうかを判断する例です:

```
(defun hscroll-on-screen (window position)
  (save-excursion
    (goto-char position)
    (and
     (>= (- (current-column) (window-hscroll window)) 0)
     (< (- (current-column) (window-hscroll window))
        (window-width window)))))
```

28.24 座標とウィンドウ

このセクションでは位置とそのウィンドウを報告する関数を説明します。これらの関数のほとんどはウィンドウのフレームのネイティブ位置の原点から相対的な位置を報告します (Section 29.3 [Frame

Geometry], page 701 を参照)。いくつかの関数はウィンドウのフレームのディスプレイの原点から相対的な位置を報告します。いずれのケースにおいても原点は座標 (0, 0) をもち、X 座標と Y 座標はそれぞれ右方向と下方向で増加します。

以下の関数では X 座標と Y 座標は整数の文字単位 (行数と列数) で報告されます。グラフィカルなディスプレイ上での “行” と “列” はそれぞれ、そのフレームのデフォルトフォントにより指定されるデフォルト文字の高さと幅に対応します (Section 29.3.2 [Frame Font], page 707 を参照)。

`window-edges` **&optional** *window body absolute pixelwise* [Function]

この関数は *window*端の座標のリストをリターンする。*window*が省略または `nil`の場合のデフォルトは選択されたウィンドウ。

リターン値は `(left top right bottom)`という形式をもつ。リストの要素は順にそのウィンドウにより占有される最左列の X 座標、最上行の Y 座標、最右列より 1 列右の X 座標、最下行より 1 行下の Y 座標。

これらはヘッダーライン、モードライン、スクロールバー、ウィンドウディバイダー、ディスプレイマージンを含むウィンドウの実際の端であることに注意。テキスト端末ではそのウィンドウの右に隣接するウィンドウがあれば、ウィンドウの右端にはそのウィンドウと隣接するウィンドウの間のセパレーターラインが含まれる。

オプション引数 *body*が `nil`なら、それは *window*の総サイズに対応する端をリターンすることを意味する。*body*が非 `nil`なら *window*のボディー (テキスト領域) の端をリターンすることを意味する。*body*が非 `nil`なら *window*には生きたウィンドウを指定しなければならない。

オプション引数 *absolute*が `nil`なら、それは *window*のフレームのネイティブ位置に相対的な端のリターンを意味する。*absolute*が非 `nil`なら *window*のディスプレーの原点 (0, 0) からの相対座標のリターンを意味する。非グラフィカルなシステムではこの引数に効果はない。

オプション引数 *pixelwise*が `nil`なら、それは *window*のフレームのデフォルト文字の幅と高さの単位で座標をリターンすることを意味する。*pixelwise*が非 `nil`ならピクセル単位で座標をリターンすることを意味する。*right*と *bottom*で指定されるピクセルはこれらの端の外側であることに注意。*absolute*が非 `nil`なら、*pixelwise*も暗黙に非 `nil`となる。

`window-body-edges` **&optional** *window* [Function]

この関数は *window*のボディーの端をリターンする (Section 28.3 [Window Sizes], page 613 を参照)。`(window-body-edges window)`の呼び出しは `(window-edges window t)` (上記参照) の呼び出しと等価。

以下の関数は一連のフレーム相対座標 (frame-relative coordinates) からウィンドウへの関連付けに使用できます:

`window-at` *x y* **&optional** *frame* [Function]

この関数は *frame*のネイティブ位置 (Section 29.3 [Frame Geometry], page 701 を参照) から相対的に、デフォルト文字単位 (Section 29.3.2 [Frame Font], page 707 を参照) で与えられる座標 *x*と *y*にある生きたウィンドウをリターンする。

その位置にウィンドウがなければリターン値は `nil`。*frame*が省略または `nil`の場合のデフォルトは選択されたフレーム。

`coordinates-in-window-p` *coordinates window* [Function]

この関数はウィンドウ *window*がフレーム相対座標 *coordinates*を占有するかどうかをチェックして、もしそうならウィンドウのどの部分かをチェックする。*window*は生きたウィンドウであること。

*coordinates*は `(x . y)` という形式のコンスセルであること。ここで *x* と *y* は *window* のフレームのネイティブ位置 (Section 29.3 [Frame Geometry], page 701 を参照) から相対的に、デフォルト文字サイズ (Section 29.3.2 [Frame Font], page 707 を参照) の単位で与えられる。

指定された位置にウィンドウが存在しなければリターン値は `nil`。それ以外ではリターン値は以下のいずれか:

`(relx . rely)`
: その座標は *window* 内にある。数値 *relx* と *rely* は指定された位置にたいする、ウィンドウ左上隅を原点に 0 から数えたウィンドウ相対座標と等価。

`mode-line`
: その座標は *window* のモードライン内にある。

`header-line`
: その座標は *window* のヘッダーライン内にある。

`tab-line`
: その座標は *window* のタブライン内にある。

`right-divider`
: その座標は *window* と右に隣接するウィンドウを分けるディバイダー内にある。

`bottom-divider`
: その座標は *window* と下にあるウィンドウを分けるディバイダー内にある。

`vertical-line`
: その座標は *window* と右に隣接するウィンドウを分ける垂直ライン内にある。この値はウィンドウにスクロールバーがないときのみ発生し得る。スクロールバー内の位置はこれらの目的にたいしてはウィンドウ外側と判断される。

`left-fringe`
`right-fringe`
: その座標はウィンドウの左か右のフリンジ内にある。

`left-margin`
`right-margin`
: その座標はウィンドウの左か右のマージン内にある。

`nil`
: その座標は *window* のいずれの部分でもない。

関数 `coordinates-in-window-p` は *window* のあるフレームを使用するので引数としてフレームを要求しない。

以下の関数は文字単位ではなくピクセル単位でウィンドウ位置をリターンします。主にグラフィカルなディスプレイで有用ですがテキスト端末上でも呼び出すことができ、その場合は各文字の占めるスクリーン領域が 1 ピクセルとなります。

`window-pixel-edges` **&optional** *window* [Function]
この関数は *window* 端にたいするピクセル座標のリストをリターンする。`(window-pixel-edges window)` の呼び出しは、`(window-edges window nil nil t)` (上記参照) の呼び出しと等価。

`window-body-pixel-edges` **&optional** *window* [Function]
この関数は *window* のボディー端をピクセルでリターンする。`(window-body-pixel-edges window)` の呼び出しは、`(window-edges window t nil t)` (上記参照) の呼び出しと等価。

以下の関数はフレーム原点ではなく、ディスプレイ画面 (display screen) の原点に相対的なウィンドウ位置をピクセルでリターンします。

`window-absolute-pixel-edges` **&optional** *window* [Function]
この関数は *window*のフレームのディスプレイの (0, 0) にある原点から相対的な *window* のピクセル座標をリターンする。`(window-absolute-pixel-edges)`の呼び出しは `(window-edges window nil t t)`の呼び出しと等価。上記を参照のこと。

`window-absolute-body-pixel-edges` **&optional** *window* [Function]
この関数は *window*のフレームのディスプレイの (0, 0) にある原点から相対的な *window*の body のピクセル座標のリストをリターンする。`(window-absolute-body-pixel-edges window)`の呼び出しは`(window-edges window t t t)`の呼び出しと等価。上記を参照のこと。

`set-mouse-absolute-pixel-position`と組み合わせることにより、あるウィンドウ内で可視な任意のバッファー位置にマウスポインターを移動するためにこの関数を使用できる。

```
(let ((edges (window-absolute-body-pixel-edges))
      (position (pos-visible-in-window-p nil nil t)))
  (set-mouse-absolute-pixel-position
   (+ (nth 0 edges) (nth 0 position))
   (+ (nth 1 edges) (nth 1 position))))
```

グラフィカルな端末では上記フォームは選択されたウィンドウのポイントにあるグリフ左上隅にマウスカーソルを “ワープ” させる。この方法で計算される位置は、そこにツールチップウィンドウを表示するためにも使用できる。

以下の関数はウィンドウ内で可視なバッファー位置のスクリーン座標をリターンします。

`window-absolute-pixel-position` **&optional** *position window* [Function]
バッファー位置 *position*がウィンドウ *window*内で可視なら、この関数は *position*にあるグリフの左上隅のディスプレー座標をリターンする。リターン値は *window*のディスプレー原点 (0, 0) から相対的な、X 座標と Y 座標からなるコンスセル。*window*内で *position*が不可視なら `nil`をリターンする。

*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。*position* のデフォルトは *window*の `window-point`の値。

これは選択されたウィンドウ内のポイント位置へのマウスポインターの移動は、以下のような記述で足りることを意味する:

```
(let ((position (window-absolute-pixel-position)))
  (set-mouse-absolute-pixel-position
   (car position) (cdr position)))
```

以下の関数はそのウィンドウに表示するテキストを変換することなくウィンドウに内接する最大の矩形をリターンします。

`window-largest-empty-rectangle` **&optional** *window count min-width min-height positions left* [Function]
この関数は指定死 *window*に内接する空の最大の矩形のディメンションをリターンする。*window* は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。

リターン値は *window*のテキスト領域の空スペース (テキストを何も表示していないスペース) に内接可能な最大矩形の幅、Y 座標の開始と終了からなる 3 要素のリスト。このような矩形は

*window*のテキスト領域の右エッジで終端されるとみなされるので、この関数はX座標をリターンしない。空のスペースが見つからなければリターン値は`nil`。

オプション引数 *count*が非`nil`なら、それはリターンする矩形の最大数を指定する。これはリターン値が最大の矩形が1つ目になるような矩形 (3要素のリスト) のリストであることを意味する。*count*はコンスセルでもよく、この場合にはCARがリターンする矩形数、CDRが非`nil`ならリターンされるすべての矩形が互いに素でなければならないことを示す。

オプション引数 *min-width*と *min-height*が非`nil`なら、リターンされる矩形の最小の幅と高さを指定する。

オプション引数 *positions*非`nil`なら、それはリターンされる矩形がカバーしなければならない最上のピクセル位置がCAR、最下のピクセル位置がCDRであるようなコンスセル。これらの位置は *window*のテキスト領域の開始から計測される。

オプション引数 *left*が非`nil`なら、それはリターン値がテキストを右から左に表示するバッファーに適していることを意味する。この場合にはリターンされる矩形はすべて *window*のテキスト領域の左端から開始するとみなされる。

この関数は`window-lines-pixel-dimensions` (Section 39.10 [Size of Displayed Text], page 1007 を参照) を通じて *window*のグリフマトリクスの各行のディメンションを取得する必要があることに注意。したがってこの関数は *window*のグリフマトリクスが最新でなければ`nil`をリターンするかもしれない。

28.25 マウスによるウィンドウの自動選択

以下のオプションによりマウスポインターの下のウィンドウを自動的に選択することができます。これはウィンドウシステムのウィンドウにマウスポインターがエンターした際は常にフレームにフォーカスを与える (それゆえ以降の選択をトリガーする) ウィンドウマネージャーと同様なポリシーを実現します (Section 29.10 [Input Focus], page 732 を参照)。

`mouse-autoselect-window` [User Option]

この変数が非`nil`なら、Emacsはマウスポインターの下のウィンドウの自動的な選択を試みる。有意な値は以下のとおり:

正の数値 これは自動選択トリガー後の遅延秒数を指定する。マウスポインター下のウィンドウはこの遅延秒数の間マウスが留まった後に選択される。

負の数値 負の数値は正の数値と同様の効果をもつが、数値の絶対数の遅延秒数の間マウスポインターがウィンドウ内に留まり加えて移動を停止した後にウィンドウを選択する点が異なる。

その他の値 その他の非`nil`値はマウスポインターがウィンドウ上にエンターすると即座にウィンドウを選択することを意味する。

いずれのケースでもウィンドウの選択をトリガーするために、マウスポインターはウィンドウのテキスト領域にエンターしなければならない。スクロールバーのスライダーのドラッグやウィンドウのモードラインではコンセプト的には自動選択を発生させるべきではない。

マウスによる自動選択はミニバッファーウィンドウがアクティブなときのみ選択を行い、アクティブなミニバッファーウィンドウの選択解除は決して行わない。

マウスによる自動選択はウィンドウマネージャーが追跡しない子フレーム (Section 29.14 [Child Frames], page 739 を参照) にたいしてマウスフォーカス追従ポリシーをエミュレートするために使用できます。これを行うためには`focus-follows-mouse` (Section 29.10 [Input Focus], page 732

を参照) に非nil値をセットすることが必要です。focus-follows-mouseの値がauto-raiseなら、マウスで子フレームにエンターすることにより親フレームの他のすべての子フレームの前面にその子フレームが自動的にレイズされます。

28.26 ウィンドウの構成

ウィンドウ構成 *(window configuration)* とは 1 つのフレーム上全体のレイアウト — すべてのウィンドウとサイズ、どんなバッファーを含んでいるか、それらのバッファーがスクロールされる方法、およびポイント値を記録して、更にフリンジ、マージン、スクロールバーのセッティングも記録します。これにはminibuffer-scroll-windowの値も含まれます。特別な例外としてウィンドウ構成には選択されたウィンドウのカレントバッファーのポイント値は記録されません。

以前に保存されたウィンドウ構成をリストアすることにより、フレーム全体のレイアウトをリストアすることができます。1 つだけではなくすべてのフレームのレイアウトを記録したければ、ウィンドウ構成のかわりにフレーム構成 (frame configuration) を使用します。Section 29.13 [Frame Configurations], page 739 を参照してください。

current-window-configuration **&optional** *frame* [Function]
この関数は *frame*のカレントのウィンドウ構成を表す新たなオブジェクトをリターンする。*frame* のデフォルトは選択されたフレーム。変数 window-persistent-parameteresはこの関数により保存されるウィンドウパラメーター (もしあれば) を指定する。Section 28.27 [Window Parameters], page 688 を参照のこと。

set-window-configuration *configuration* [Function]
この関数は *configuration*が作成されたフレームにたいして、そのフレームが選択されているかどうかに関わらず、ウィンドウとバッファーの構成を *configuration*で指定された構成にリストアする。引数 *configuration*はそのフレームにたいして以前に current-window-configuration がリターンした値でなければならない。

*configuration*が保存されたフレームが死んでいる (生きていない) 場合には、この関数が行うのは変数 minibuffer-scroll-windowの値のリストア、minibuffer-selected-window がリターンした値の調整のみ。この場合には関数は nil、それ以外は tをリターンする。

*configuration*のウィンドウのバッファーが *configuration*作成後に kill されていたら、そのウィンドウは規則としてリストアされる構成から削除される。しかしリストアされる構成内でそのウィンドウが最後に残ったウィンドウなら、そのウィンドウに別の生きたバッファーが表示される。

以下は save-window-excursionと同様な効果を得るためにこの関数を使用する例:

```
(let ((config (current-window-configuration)))
  (unwind-protect
      (progn (split-window-below nil)
             ...)
    (set-window-configuration config)))
```

save-window-excursion *forms...* [Macro]
このマクロは選択されたフレームのウィンドウ構成を記録して、*forms*を順に実行してから以前のウィンドウ構成をリストアする。リターン値は *forms*内の最後のフォームの値。

Lisp コードのほとんどはこのマクロを使用するべきではない。大抵は save-selected-window で十分であろう。特にこのマクロは *forms*内で新たなウィンドウをオープンするコードを確実に防ぐことができず、新たなウィンドウは別のフレーム内でオープンされるかもしれないが

(Section 28.13.1 [Choosing Window], page 640 を参照)、`save-window-excursion`が保存とリストアするのはカレントフレーム上のウィンドウ構成だけだからである。

`window-configuration-p` *object* [Function]
この関数は *object*がウィンドウ構成なら `t`をリターンする。

`compare-window-configurations` *config1 config2* [Function]
この関数はポイント値、保存されたスクロール位置を無視して (つまりこれらの観点が異なっても `t`をリターンし得る)、ウィンドウ構造の観点から 2 つのウィンドウ構成を比較する。

関数 `equal`も 2 つのウィンドウ構成を比較できる。これはすべての点からたとえ 1 つでも異なるものがあれば等しい構成とはみなさず、たとえ保存されたポイント値が異なるだけでも等しくないとみなす。

`window-configuration-frame` *config* [Function]
この関数はウィンドウ構成 *config*が作成されたフレームをリターンする。

ウィンドウ構成の内部を調べる他のプリミティブも有用かもしれませんが、わたしたちはこれらを必要としないので実装されていません。ウィンドウ構成にたいしてより多くの操作を知りたければ、ファイル `winner.el`を参照してください。

`current-window-configuration`がリターンするオブジェクトは Emacs プロセスとともに死滅します。ウィンドウ構成をディスク上に格納してそれを別の Emacs セッションに読み戻すために、次に説明する関数を使用できます。これらの関数はフレームの状態を任意の生きたウィンドウにクローンする場合にも有用です (`set-window-configuration`はフレームのウィンドウをそのフレームのルートウィンドウだけに効果的にクローンする)。

`window-state-get` **&optional** *window writable* [Function]
この関数は *window*の状態を Lisp オブジェクトとしてリターンする。引数 *window*は有効なウィンドウでなければならずデフォルトは選択されたフレームのルートウィンドウ。

オプション引数 *writable*が非 `nil`なら、それは `window-point`や `window-start`のようなサンプリング位置にたいするマーカーを使用しないことを意味する。この状態をディスクに書き込んで別のセッションに読み戻すなら、この引数は非 `nil`であること。

この関数によりどのウィンドウパラメーターが保存されるかは、引数 *writable*と変数 `window-persistent-parameters`の両方で指定する。Section 28.27 [Window Parameters], page 688 を参照のこと。

`window-state-get`によりリターンされる値は、同一セッション内の他のウィンドウ内にあるウィンドウのクローンを作成するために使用できます。これはディスクに書き込んで別のセッションに読み戻すこともできます。いずれの場合にもウィンドウの状態をリストアするためには以下の関数を使用します。

`window-state-put` *state* **&optional** *window ignore* [Function]
この関数はウィンドウ状態 *state*を *window*内に put する。引数 *state*は以前に呼び出した `window-state-get`がリターンしたウィンドウ状態であること。オプション引数 *window*には生きたウィンドウか内部ウィンドウ (Section 28.2 [Windows and Frames], page 609 を参照) のいずれかを指定できる。*window*が生きていなければ、*state*を put する前に生きたウィンドウに置き換える。*window*が生きたウィンドウでなければ、それに *state*を put する前に同一フレーム上に作成された新たな生きたウィンドウに置き換えられる。*window*が `nil`ならウィンドウの状態を新たなウィンドウに put する。

オプション引数 *ignore*が非 `nil`なら、それは最小ウィンドウサイズと固定サイズの制限を無視することを意味する。*ignore*が `safe`なら、それは 1 行および/または 2 列までできる限り小さくできることを意味する。

関数 `window-state-get`と `window-state-put`では 2 つの生きたウィンドウ間でのコンテンツの交換も可能です。以下の関数はこれを正確に行います:

`window-swap-states` **&optional** *window-1 window-2 size* [Command]
このコマンドは 2 つの生きたウィンドウ *window-1*と *window-2*の状態を交換する。*window-1*には生きたウィンドウを指定しなければならずデフォルトは選択されたウィンドウ。*window-2*には生きたウィンドウを指定しなければならず、デフォルトはミニバッファーウィンドウを除き、すべての可視なフレーム上の生きたウィンドウを含むウィンドウサイクル順において *window-1*の次のウィンドウ。

オプション引数 *size*が非 `nil`なら、それは *window-1*と *window-2*のサイズも同様に交換を試みることを意味する。値 `height`は高さのみ、値 `width`は幅のみ、`t`は幅と高さの両方を可能なら交換することを意味する。この関数はフレームをリサイズしない。

28.27 ウィンドウのパラメーター

このセクションではウィンドウに追加の情報を関連付けるために使用できるウィンドウパラメーターを説明します。

`window-parameter` *window parameter* [Function]
この関数は *window*の *parameter*の値をリターンする。*window*のデフォルトは選択されたウィンドウ。*window*に *parameter*にたいするセッティングがなければ、この関数は `nil`をリターンする。

`window-parameters` **&optional** *window* [Function]
この関数は *window*のすべてのパラメーターと値をリターンする。*window*のデフォルトは選択されたウィンドウ。リターン値は `nil`、または `(`*`parameter . value`*`)`という形式をもつ要素からなる連想リスト。

`set-window-parameter` *window parameter value* [Function]
この関数は *window*の *parameter*の値に *value*をセットして *value*をリターンする。*window*のデフォルトは選択されたウィンドウ。

デフォルトではウィンドウ構成 (window configuration) やウィンドウ状態 (states of windows) の保存とリストアを行う関数は、ウィンドウパラメーターについては関知しません (Section 28.26 [Window Configurations], page 686 を参照)。これは `save-window-excursion`の body 内でパラメーターの値を変更したときは、そのマクロの exit 時に以前の値がリストアされないことを意味します。これはまた以前に `window-state-get`で保存されたウィンドウ状態を `window-state-put`でリストアしたときは、クローンされたすべてのウィンドウのパラメーターが `nil`にリセットされることも意味します。以下の変数によってこの標準の挙動をオーバーライドできます:

`window-persistent-parameters` [Variable]
この変数は `current-window-configuration`と `window-state-get`により保存、`set-window-configuration`と `window-state-put`によりリストアされるパラメーターを指定する alist である。Section 28.26 [Window Configurations], page 686 を参照のこと。

この alist の各エントリーの CAR はパラメーターを指定するシンボル。CDR は以下のいずれかであること:

`nil` この値はそのパラメーターが `window-state-get`と `current-window-configuration`のいずれによっても保存されていないことを意味する。

`t` この値はそのパラメーターが `current-window-configuration`、および(*writable*引数が `nil`なら)`window-state-get`により保存されたことを意味する。

`writable` これはそのパラメーターが無条件で `current-window-configuration`と `window-state-get`の両方により保存されたことを意味する。この値は入力構文 (read syntax) をもたないパラメーターに使用するべきではない。使用した場合には、別のセッションで `window-state-put`を呼び出すと`invalid-read-syntax`エラーで失敗するだろう。

いくつかの関数 (特に `delete-window`、`delete-other-windows`、`split-window`) は、*window*引数で指定されたウィンドウがその関数の名前と同じ名前のパラメーターセをもつ場合には特別な挙動を示すかもしれません。以下の変数を非 `nil`値にバインドすることにより、そのような特別な挙動をオーバーライドできます:

`ignore-window-parameters` [Variable]

この変数が非 `nil`なら、いくつかの標準関数はウィンドウパラメーターを処理しない。現在のところ影響を受ける関数は `split-window`、`delete-window`、`delete-other-windows`、`other-window`。

これらの関数の呼び出し周辺でアプリケーションはこの変数を非 `nil`にバインドできる。これを行うと、そのアプリケーションはその関数の exit 時に関連するすべてのウィンドウのパラメーターを正しく割り当てる責任をもつ。

以下のパラメーターは現在のところウィンドウ管理コードにより使用されています:

`delete-window`
: このパラメーターは `delete-window`の実行に影響する (Section 28.7 [Deleting Windows], page 625 を参照)。

`delete-other-windows`
: このパラメーターは `delete-other-windows`の実行に影響する (Section 28.7 [Deleting Windows], page 625 を参照)。

`no-delete-other-windows`
: このパラメーターはそのウィンドウを `delete-other-windows`により削除できないことをマークする (Section 28.7 [Deleting Windows], page 625 を参照)。

`split-window`
: このパラメーターは `split-window`の実行に影響する (Section 28.6 [Splitting Windows], page 622 を参照)。

`other-window`
: このパラメーターは `other-window`の実行に影響する (Section 28.10 [Cyclic Window Ordering], page 633 を参照)。

`no-other-window`
: このパラメーターはそのウィンドウを `other-window`による選択が不可だとマークする (Section 28.10 [Cyclic Window Ordering], page 633 を参照)。

`clone-of`
: このパラメーターはそのウィンドウがクローンされたことを指定する。これは `window-state-get`によりインストールされる (Section 28.26 [Window Configurations], page 686 を参照)。

`window-preserved-size`
: このパラメーターはバッファー、方向 (`nil`は垂直で `t`は水平)、ピクセル単位のサイズを指定する。そのウィンドウが指定されたバッファーを表示していて、かつ指示された方向のサイズがこのパラメーターで指定されたサイズと等しければ、Emacs はそのウィンドウの指示された方向のサイズを予約する。関数 `window-preserve-size`によりこのパラメーターのインストールと更新が行われる (Section 28.5 [Preserving Window Sizes], page 621 を参照)。

`quit-restore`
: このパラメーターはバッファー表示関数によりインストールされて、`quit-restore-window`により参照される (Section 28.16 [Quitting Windows], page 662 を参照)。これは 4 つの要素をもつリストであり、詳細は Section 28.16 [Quitting Windows], page 662 の `quit-restore-window`の説明を参照のこと。

`window-side`
`window-slot`
: これらはサイドウィンドウを実装するために内部的に使用される (Section 28.17 [Side Windows], page 664 を参照)。

`window-atom`
: このパラメーターはアトミックウィンドウを実装するために内部的に使用される (Section 28.18 [Atomic Windows], page 669 を参照)。

`mode-line-format`
: このパラメーターはウィンドウが表示されるたびにそのウィンドウのバッファーのバッファーローカル変数 `mode-line-format` (Section 23.4.1 [Mode Line Basics], page 485 を参照) の値を置き換える。シンボル `none`はそのウィンドウのモードライン表示の抑制を意味する。そのバッファーを表示する他のウィンドウのモードラインの表示とコンテンツは影響を受けない。

`header-line-format`
: このパラメーターはウィンドウが表示されるたびにそのウィンドウのバッファーのバッファーローカル変数 `header-line-format` (Section 23.4.1 [Mode Line Basics], page 485 を参照) の値を置き換える。シンボル `none`はそのウィンドウのヘッダーライン表示の抑制を意味する。そのバッファーを表示する他のウィンドウのヘッダーラインの表示とコンテンツは影響を受けない。

`tab-line-format`
: このパラメーターはウィンドウが表示されるたびにそのウィンドウのバッファーのバッファーローカル変数 `tab-line-format` (Section 23.4.1 [Mode Line Basics], page 485 を参照) の値を置き換える。シンボル `none`はそのウィンドウのタブライン表示の抑制を意味する。そのバッファーを表示する他のウィンドウのタブラインの表示とコンテンツは影響を受けない。

`min-margins`
: このパラメーターの値は CAR と CDR が非 `nil`のコンスセルなら、そのウィンドウの左ージンと右マージンの最小値を列数で指定する (Section 39.16.5 [Display Margins], page 1047 を参照)。与えられた場合には、Emacs はウィンドウを分割するか水平方向に縮小するかの判断にたいして、実際のマージン幅のかわりにこれらの値を使用する。

すべてのウィンドウにたいして、分割やリサイズ後に Emacs がマージンの自動調整をすることは決してない。これはそのウィンドウト分割のためにこのウィンドウのマージンを継承する新たなウィンドウニタイシテ、マージンを調整するためにこのパラメーターをセットするすべてのアプリケーション単独の責任である。`window-configuration-change-hook`と `window-size-change-functions`はいずれもこの用途に使用すること (Section 28.28 [Window Hooks], page 691 を参照)。

これはウィンドウ内でバッファーをセンタリングするために大きなマージンを使用するアプリケーションのサポート用に Emacs のバージョン 25.1 から導入されたパラメーターであり、それらのアプリケーションと排他となるよう留意して使用すること。Emacs の将来のバージョンで改善策により置換され得る。

28.28 ウィンドウのスクロールと変更のためのフック

このセクションでは Lisp プログラムがウィンドウのスクロール後や、その他のウィンドウ変更が発生した際にどのようにしてアクションを起こすことができるかを説明します。最初はウィンドウがバッファーの異なる部分を表示するケースについて考えてみます。

`window-scroll-functions` [Variable]
: この変数はウィンドウのスクロールにより Emacs が再表示前に呼び出すべき関数のリストを保持する。そのウィンドウ内への異なるバッファーの表示や新たなウィンドウの作成でもこれらの関数が実行される。

この変数はそれぞれの関数がウィンドウとウィンドウの display-start 位置という 2 つの引数を受け取るのでノーマルフックではない。呼び出し時には window 引数の display-start 位置は新たな値、そのウィンドウに表示されるバッファーがカレントバッファーとしてセットされる。

これらの関数は `window-end` (Section 28.20 [Window Start and End], page 671 を参照) を使用する際には気をつけなければならない。最新の値が必要なら、それを確実に入力するために *update*引数を使用しなければならない。

警告**:** ウィンドウのスクロール方法を変更するためにこの機能を使用してはならない。これはそのような用途のためにデザインされておらず、そのような使用では機能しないだろう。

加えて Font Lock フォント表示関数 (Font Lock fontification function) を登録するために `jit-lock-register`を使用できる。バッファーの一部が (再) フォント表示されたときは、ウィンドウがスクロールまたはサイズ変更されたという理由で、これが常に呼び出されるだろう。Section 23.6.4 [Other Font Lock Variables], page 502 を参照のこと。

このセクションの残りの部分ではウィンドウのスクロールをともなわないウィンドウの有意な変更を検知した際の再表示の間に呼び出される 6 つのフックについて補足します。単純化するために、これらのフックおよびそれらが呼び出す関数をまとめてウィンドウ変更関数 *(window change functions)* と呼ぶことにします。

これらのフックのうち最初のフックはウィンドウバッファー変更 *(window buffer change)* を検知した後に実行されます。ウィンドウバッファー変更とはウィンドウの作成や削除、他のバッファーの割り当てを意味します。

`window-buffer-change-functions` [Variable]

この変数はウィンドウバッファー変更時の再表示の間に呼び出す関数を指定する。値は 1 つの引数を受け取る関数のリストであること。

バッファーローカルに指定された関数は、対応するバッファーを表示するすべてのウィンドウにたいして、最後にウィンドウ変更関数が実行されて以降にウィンドウが作成されたり、そのバッファーが割り当てられた場合に呼び出される。この場合にはそのウィンドウが引数として渡される。

デフォルト値として指定された関数はフレームにたいして、最後にウィンドウ変更関数が実行されて以降にそのフレームに少なくとも 1 つのウィンドウの追加か削除が行われるか、別のバッファーが割り当てられた場合に呼び出される。この場合にはフレームが引数として渡される。

2 つ目のフックはウィンドウサイズ変更 *(window size change)* の検出時に実行されます。ウィンドウサイズ変更とはウィンドウ作成、別バッファーの割り当て、合計サイズやテキストエリアの合計サイズの変更を意味します。

`window-size-change-functions` [Variable]

この変数はウィンドウサイズ変更発生時の再表示の間に呼び出される関数を指定する。値は 1 つの引数を受け取る関数のリストであること。

バッファーローカルに指定された関数は、対応するバッファーを表示するすべてのウィンドウにたいして、最後にウィンドウ変更関数が実行されて以降にウィンドウの追加や別バッファーの割り当て、合計サイズやボディーサイズが変更された場合に呼び出される。この場合にはそのウィンドウが引数として渡される。

デフォルト値として指定された関数は、最後にフレームにたいしてウィンドウ変更関数が実行されて以降に、そのフレーム上の少なくとも 1 つのウィンドウ追加や別バッファーの割り当て、または合計サイズやボディーサイズが変更された場合に呼び出される。この場合にはフレームが引数として渡される。

3 つ目のフックは前回の再表示以降にウィンドウ選択変更 *(window selection change)* により別のウィンドウが選択された際に実行されます。

`window-selection-change-functions` [Variable]

この変数は選択されたウィンドウやフレームの選択されたウィンドウの変更時の再表示の間に実行される関数を指定する。値は 1 つの引数を受け取る関数のリストであること。

バッファーローカルに指定された関数は、対応するバッファーを表示するすべてのウィンドウにたいして、最後にウィンドウ変更関数が実行されて以降に (すべてのウィンドウあるいはウィンドウのフレーム上のすべてのウィンドウの中から) ウィンドウが選択されたり選択解除された場合に呼び出される。この場合にはそのウィンドウが引数として渡される。

デフォルト値として指定された関数はフレームにたいして、最後にウィンドウ変更関数が実行されて以降のそのフレームの選択や非選択、あるいはフレームの選択されたウィンドウが変更された場合に呼び出される。この場合にはフレームが引数として渡される。

4 つ目のフックはウィンドウ状態変更 *(window state change)* の検出時に実行されます。ウィンドウ状態変更とは上述の 3 つのウィンドウ変更のうち少なくとも 1 つが発生したことを意味します。

`window-state-change-functions` [Variable]

この変数はウィンドウのバッファーかサイズの変更発生時、または選択されたウィンドウやフレームの選択されたウィンドウの変更時の再表示の間に呼び出される関数を指定する。値は 1 つの引数を受け取る関数のリストであること。

バッファーローカルに指定された関数は、対応するバッファーを表示するすべてのウィンドウにたいして、最後にウィンドウ変更関数が実行されて以降にそのウィンドウの追加や別バッファーの割り当て、合計サイズやボディーサイズが変更されたり、(すべてのウィンドウあるいはウィンドウのフレーム上のすべてのウィンドウの中から) ウィンドウが選択されたり選択解除された場合に呼び出される。この場合にはそのウィンドウが引数として渡される。

デフォルト値として指定された関数はフレームにたいして、最後にウィンドウ変更関数が実行されて以降にそのフレームの少なくとも 1 つのウィンドウが追加や削除、別バッファーを割り当てられるか、合計サイズやボディーサイズの変更、またはフレームの選択または選択されていないウィンドウが変更された場合に呼び出される。この場合にはフレームが引数として渡される。

デフォルト値として指定された関数は、最後の再表示以降にそのフレームのウィンドウ状態変更フラグ (以下参照) がセットされた場合にもフレームにたいして実行される。

5 つ目のフックはウィンドウ構成変更 *(window configuration change)* の検出時に実行されます。ウィンドウ構成変更とはバッファーやウィンドウサイズの変更を意味します。このフックは実行方向において前記 4 つのフックとは異なります。

`window-configuration-change-hook` [Variable]

この変数はウィンドウのバッファーかサイズの変更時の再表示の間に呼び出される関数を指定する。値は引数を受け取らない関数のリストであること。

バッファーローカルに指定された関数は、対応するバッファーを表示するすべてのウィンドウにたいして、最後にウィンドウ変更関数が実行されて以降に、そのフレームの少なくとも 1 つのウィンドウの追加や削除、別バッファーの割り当て、あるいは合計サイズやボディーサイズが変更された場合に呼び出される。各呼び出しは一時的にそのバッファーを表示するウィンドウを選択して、そのバッファーをカレントにして行われる。

デフォルト値として指定された関数は各フレームにたいして、最後にウィンドウ変更関数が実行されて以降に、そのフレームに少なくとも 1 つのウィンドウの追加や削除、別バッファーの割り当て、あるいは合計サイズやボディーサイズが変更された場合に呼び出される。各呼び出しは一時的にそのフレームを選択して、その選択されたウィンドウのバッファーをカレントにして行われる。

最後に Emacs は `window-state-change-functions`の振る舞いを一般化したノーマルフックを実行します。

`window-state-change-hook` [Variable]

この変数のデフォルト値はウィンドウ状態変更、または少なくとも 1 つのフレームでウィンドウ状態変更フラグのセットを検出した際の再表示の間に呼び出される関数を指定する。値は引数を受け取らない関数のリストであること。

アプリケーションは最後の再表示以降に複数のフレームで発生した (またはシグナルされた) 変更に対処したい場合のみ、このフックに関数を配置すること。他のすべてのケースにたいしては `window-state-change-functions`に関数を配置するほうが好ましい。

ウィンドウ変更関数は各フレームにたいする再表示の間に次のように呼び出されます。まずすべてのバッファーローカルなウィンドウバッファー変更関数、ウィンドウサイズ変更関数、ウィンドウ選択変更関数、ウィンドウ状態変更関数を順番に呼び出します。次にこれらの関数のデフォルト値を同じ順番で呼び出します。これらの関数のデフォルト値で指定された関数の呼び出した後に、すべてのバッファーローカルなウィンドウ構成変更関数を呼び出します。そして最後に `window-state-change-hook`の関数が実行されます。

ウィンドウ変更関数は対応する変更が前に登録されている特定のフレームにたいしてのみ実行されます。そのような変更にはウィンドウの作成や削除、ウィンドウへの別バッファーやサイズの割り当てが含まれます。そのような変更が登録されていたとしても、それは上述したフックのいずれかが実行されることを意味しないことに注意してください。たとえばある変更がウィンドウエクスカーション (Section 28.26 [Window Configurations], page 686 を参照) のスコープ内で登録されると、ウィンドウ変更関数呼び出しはウィンドウ変更関数実行時にそのエクスカーションがまだ持続している場合のみトリガーされるでしょう。それ以前にエクスカーションが exit されていれば、そのエクスカーションのスコープ外で登録された変更の場合のみフックが実行されます。

フレームのウィンドウ状態変更フラグ *(window state change flag)* をセットすることにより、そのフレームにたいしてウィンドウ状態変更が実際に発生したかどうかに関わらず、次回の再表示の間に (そのフレームにたいする) `window-state-change-functions`と `window-state-change-hook` のデフォルト値が実行されます。これらのフックのすべての関数を実行した後に、各フレームにたいしてフラグをリセットを行います。アプリケーションは以下の関数を使用してこのフラグのセットや検査を行うことができます。

`set-frame-window-state-change` **&optional** *frame arg* [Function]
この関数は *arg*が非 `nil`なら *frame*のウィンドウ状態変更フラグをセット、それ以外はリセットする。*frame*は生きたフレームでなければならず、デフォルトは選択されたフレーム。

`frame-window-state-change` **&optional** *frame* [Function]
この関数は *frame*のウィンドウ状態変更フラグがセットされていれば `t`、それ以外は `nil`をリターンする。*frame*は生きたフレームでなければならず、デフォルトは選択されたフレーム。

ウィンドウ変更関数の実行中に下記の関数を呼び出すことにより、最後の再表示以降に特定のウィンドウやフレームにたいして何が変更されたかについてより詳細な知見を得ることができます。これらの関数はすべて生きたウィンドウを単一かつオプションの引数として受け取ります (デフォルトは選択されたウィンドウ)。

`window-old-buffer` **&optional** *window* [Function]
この関数は *window*のフレームにたいして最後にウィンドウ変更関数が実行された時点において、*window*に表示されていたバッファーをリターンする。その時点より後に *window*が作成された場合には `nil`をリターンする。その時点では *window*は表示されていなかったが、後から以前に保存されたウィンドウ構成がリストアされた場合には `t`をリターンする。それ以外ならリターン値はその時点で `window`が表示していたバッファー。

`window-old-pixel-width` **&optional** *window* [Function]
この関数はウィンドウ変更関数がフレーム上で `window`が生きていることを確認した最終時点における `window`のトータルピクセル幅をリターンする。その後に `window`が作成された場合は 0。

`window-old-pixel-height` **&optional** *window* [Function]
この関数はウィンドウ変更関数がフレーム上で `window`が生きていることを確認した最終時点における `window`のトータルピクセル高さをリターンする。その後に `window`が作成された場合は 0。

`window-old-body-pixel-width` **&optional** *window* [Function]
この関数はウィンドウ変更関数がフレーム上で `window`が生きていることを確認した最終時点における `window`のテキストエリアのピクセル幅をリターンする。その後に `window`が作成された場合は 0。

`window-old-body-pixel-height` **&optional** *window* [Function]
この関数はウィンドウ変更関数がフレーム上で `window`が生きていることを確認した最終時点における `window`のテキストエリアのピクセル高さをリターンする。その後に `window`が作成された場合は 0。

ウィンドウ変更関数が最後に実行された時点でどのウィンドウまたはフレームが選択されていたかを調べるために以下の関数を使用できます:

`frame-old-selected-window` **&optional** *frame* [Function]
この関数はウィンドウ変更関数が最後に実行された時点における *frame*の選択されていたウィンドウをリターンする。*frame*が省略または `nil`の場合のデフォルトは選択されたフレーム。

`old-selected-window` [Function]
この関数はウィンドウ変更関数が最後に実行された時点において選択されていたウィンドウをリターンする。

`old-selected-frame` [Function]
この関数はウィンドウ変更関数が最後に実行された時点において選択されていたフレームをリターンする。

ウィンドウ変更関数は最後に実行されて以降に削除されたウィンドウに関する情報は提供しないことに注意してください。アプリケーションがそれを必要とする場合には、特定のバッファーを表示するすべてのウィンドウをバッファーのローカル変数に記憶するとともに、それをウィンドウバッファー変更の検出時に実行されるすべてのフックのデフォルト値の関数で更新する必要があります。

ウィンドウ変更関数に関数を追加する際には、以下の点を考慮する必要があります:

- ウィンドウ変更関数の呼び出しをトリガーしない操作がある。これにはミニバッファーウィンドウでの別バッファー表示やツールチップウィンドウのすべての変更が含まれる。
- ウィンドウ変更関数はウィンドウの作成や削除、バッファー変更、ウィンドウのサイズや選択状態を何も変更するべきではない。なぜならそのような変更に関する情報が他のウィンドウ変更関数に伝播される保証は何もないから。もしそのような変更行うなら `window-state-change-hook`のデフォルト値の最後にリストされる関数で行うこと。
- ウィンドウ変更関数の実行時には `save-window-excursion`、`with-selected-window`、`with-current-buffer`のようなマクロを使用できる。
- マッチデータはウィンドウ変更関数の実行により保存やリストアされない。`window-configuration-change-hook`の実行以外では選択されたウィンドウやフレーム、カレントバッファーの保存やリストアは行われない。
- ウィンドウ変更関数の実行をトリガーするすべての再表示は abort(異常終了) する可能性がある。ウィンドウ変更関数が完了する前に abort が発生すると、ウィンドウ変更関数は以前の値、すなわち再表示が行われなかったかのような値で再実行される。その後に abort した場合には新たな値、すなわち再表示が実際に処理されたかのような値で実行される

29 フレーム

フレーム *(frame)* とは、1 つ以上の Emacs ウィンドウを含むスクリーンオブジェクトです (Chapter 28 [Windows], page 608 を参照)。これはグラフィカル環境では “ウィンドウ” と呼ばれる類のオブジェクトです。しかし Emacs はこの単語を異なる方法で使用しているので、ここではそれを “ウィンドウ” と呼ぶことはできません。Emacs Lisp においてフレームオブジェクト *(frame object)* とは、スクリーン上のフレームを表す Lisp オブジェクトです。Section 2.5.4 [Frame Type], page 26 を参照してください。

フレームには最初は 1 つのメインウィンドウおよび/またはミニバッファーウィンドウが含まれます。メインウィンドウは、より小さいウィンドウに垂直か水平に分割することができます。Section 28.6 [Splitting Windows], page 622 を参照してください。

端末 *(terminal)* とは 1 つ以上の Emacs フレームを表示する能力のあるデバイスのことです。Emacs Lisp において端末オブジェクト *(terminal object)* とは端末を表す Lisp オブジェクトです。Section 2.5.5 [Terminal Type], page 26 を参照してください。

端末にはテキスト端末 *(text terminals)* とグラフィカル端末 *(graphical terminals)* という 2 つのクラスがあります。テキスト端末はグラフィック能力をもたないディスプレイであり、`xterm`やその他の端末エミュレーターが含まれます。テキスト端末上ではそれぞれの Emacs フレームはその端末のスクリーン全体を占有します。たとえ追加のフレームを作成してそれらを切り替えることができたとしても、端末が表示するのは一度に 1 つのフレームだけです。一方でグラフィカル端末は X ウィンドウシステムのようなグラフィカルディスプレイシステムにより管理されています。これにより Emacs は同一ディスプレイ上に複数のフレームを同時に表示することができます。

GNU および Unix systems システムでは、単一の Emacs セッション内でその Emacs がテキスト端末とグラフィカル端末のいずれで開始されたかに関わらず、任意の利用可能な端末上で追加のフレームを作成することができます。Emacs は、グラフィカル端末とテキスト端末の両方を同時に表示することができます。これはたとえばリモートから同じセッションに接続する際などに便利でしょう。Section 29.2 [Multiple Terminals], page 698 を参照してください。

`framep` *object* [Function]

この述語 (predicate) は *object*がフレームなら非 `nil`、それ以外は `nil`をリターンする。フレームにたいしてはフレームが使用するディスプレイの種類が値:

- `t` そのフレームはテキスト端末上で表示されている。
- `x` そのフレームは X グラフィカル端末上で表示されている。
- `w32` そのフレームは MS-Windows グラフィカル端末上で表示されている。
- `ns` そのフレームは GNUstep か Macintosh Cocoa グラフィカル端末上で表示されている。
- `pc` そのフレームは MS-DOS 端末上で表示されている。

`frame-terminal` **&optional** *frame* [Function]

この関数は *frame*を表示する端末オブジェクトをリターンする。*frame*が `nil`または未指定の場合のデフォルトは選択されたフレーム。

`terminal-live-p` *object* [Function]

この述語は *object*が生きた (削除されていない) 端末なら非 `nil`、それ以外は `nil`をリターンする。生きた端末にたいしては、リターン値はその端末上で表示されているフレームの種類を示す。可能な値は上述の `framep`と同様。

グラフィカルな端末ではフレームを 2 つのタイプに区別しています。通常のトップレベルフレーム *(top-level frame)* は、ウィンドウ (ウィンドウシステム) としては端末にたいするルートウィンドウ (ウィンドウシステム) の子であるようなフレームです。子フレーム (child frame) は、ウィンドウ (ウィンドウシステム) としては Emacs の別フレームのウィンドウ (ウィンドウシステム) の子であるようなフレームです。Section 29.14 [Child Frames], page 739 を参照してください。

29.1 フレームの作成

新たにフレームを作成するためには関数 `make-frame`を呼び出します。

`make-frame` **&optional** *parameters* [Command]

この関数はカレントバッファーを表示するフレームを作成してそれをリターンする。

*parameters*引数は新たなフレームのフレームパラメーターを指定する alist。Section 29.4 [Frame Parameters], page 712 を参照のこと。*alist*内で `terminal`パラメーターを指定すると新たなフレームはその端末上で作成される。それ以外の場合には、*alist*内で `window-system` フレームパラメーターを指定すると、それはフレームがテキスト端末とグラフィカル端末のどちらで表示されるべきかを決定する。Section 39.24 [Window Systems], page 1083 を参照のこと。どちらも指定しなければ新たなフレームは選択されたフレームと同じ端末上に作成される。

*parameters*で指定されなかったパラメーターのデフォルトは連想リスト `default-frame-alist`内の値となる。そこでも指定されないパラメーターのデフォルトは X リソース、またはそのオペレーティングシステムで同等なものの値 (Section “X Resources” in *The GNU Emacs Manual* を参照)。フレームが作成された後に、この関数は `frame-inherited-parameters` (以下参照) 内で指定されたパラメーターのうち未割り当てのパラメーターにたいして、`make-frame`呼び出し時に選択されていたフレームから値を取得して適用する。

マルチモニターディスプレイ (Section 29.2 [Multiple Terminals], page 698 を参照) では、ウィンドウマネージャーが *parameters*内の位置パラメーター (Section 29.4.3.2 [Position Parameters], page 715 を参照) の指定とは異なる位置にフレームを配置するかもしれないことに注意。たとえばウィンドウの大きな部分、いわゆる支配モニター *(dominating monitor)* 上のフレームを表示するポリシーをもつウィンドウマネージャーがいくつかある。

この関数自体が新たなフレームを選択されたフレームにする訳ではない。Section 29.10 [Input Focus], page 732 を参照のこと。以前に選択されていたフレームは選択されたままである。しかしグラフィカル端末上ではウィンドウシステム自身の理由によって新たなフレームが選択されるかもしれない。

`before-make-frame-hook` [Variable]

`make-frame`がフレームを作成する前に、それにより実行されるノーマルフック。

`after-make-frame-functions` [Variable]

`make-frame`がフレームを作成した後に実行するアブノーマルフック。`after-make-frame-functions`内の各関数は作成された直後のフレームを単一の引数として受け取る。

init ファイルでこれらのフックに追加した関数は、初期フレームの作成後に Emacs が init ファイルを読み込むために、通常は初期フレームにたいして実行されないことに注意してください。しかし別のミニバッファーフレーム (Section 29.9 [Minibuffers and Frames], page 732 を参照) を使用して初期フレームを指定すれば、これらの関数はミニバッファーなしのフレームとミニバッファーフレームの両方にたいして実行されます。

`frame-inherited-parameters` [Variable]
この変数はカレントで選択されているフレームから継承して新たに作成されたフレームのフレームパラメーターのリストを指定する。リスト内の各要素は `make-frame`の処理において前に割り当てられていないパラメーター (シンボル) であり、`make-frame`は新たに作成されたフレームのそのパラメーターに選択されたフレームの値をセットする。

`server-after-make-frame-hook` [User Option]
Emacs サーバーがクライアントフレームを作成する際に実行されるノーマルフック。このフックの呼び出し時には作成されたフレームが選択されたフレームとなる。Section “Emacs Server” in *The GNU Emacs Manual* を参照のこと。

29.2 複数の端末

Emacs はそれぞれの端末を端末オブジェクト *(terminal object)* というデータ型で表します (Section 2.5.5 [Terminal Type], page 26 を参照)。GNU および Unix システムでは Emacs はそれぞれのセッション内で複数の端末を同時に実行できます。その他のシステムでは単一の端末だけが使用できます。端末オブジェクトはそれぞれ以下の属性をもちます:

- その端末により使用されるデバイスの名前 (たとえば‘`:0.0`’や`/dev/tty`)。
- その端末により使用される端末とキーボードのコーディングシステム。Section 33.10.8 [Terminal I/O Encoding], page 877 を参照のこと。
- その端末に関連付けられたディスプレイの種類。これは関数`terminal-live-p`によりリターンされるシンボル (たとえば`x`、`t`、`w32`、`ns`、`pc`)。Chapter 29 [Frames], page 696 を参照のこと。
- 端末パラメーターのリスト。Section 29.5 [Terminal Parameters], page 729 を参照のこと。

端末オブジェクトを作成するプリミティブはありません。`make-frame-on-display` (以下参照) を呼び出したときなどに、Emacs が必要に応じてそれらを作成します。

`terminal-name` **&optional** *terminal* [Function]
この関数は *terminal*により使用されるデバイスのファイル名をリターンする。*terminal*が省略または`nil`の場合のデフォルトは選択されたフレームの端末。*terminal*はフレームでもよく、その場合はそのフレームの端末。

`terminal-list` [Function]
この関数はすべての生きた端末オブジェクトのリストをリターンする。

`get-device-terminal` *device* [Function]
この関数は *device*により与えられたデバイス名の端末をリターンする。*device*が文字列なら端末デバイス名、または‘*host*`:`*server*`.`*screen*’という形式の X ディスプレイのいずれかを指定できる。*device*ならこの関数はそのフレームの端末をリターンする。`nil`は選択されたフレームを意味する。最後にもし *device*が生きた端末を表す端末オブジェクトなら、その端末がリターンされる。引数がこれらのいずれとも異なれば、この関数はエラーをシグナルする。

`delete-terminal` **&optional** *terminal force* [Function]
この関数は *terminal*上のすべてのフレームを削除して、それらが使用していたリソースを解放する。これらはアブノーマルフック`delete-terminal-functions`を実行して、各関数の引数として *terminal*を渡す。

*terminal*が省略または`nil`の場合のデフォルトは選択されたフレームの端末。*terminal*はフレームでもよく、その場合はそのフレームの端末を意味する。

この関数は通常は唯一アクティブな端末の削除を試みるとエラーをシグナルするが、*force*が非`nil`ならこれを行うことができる。端末上で最後のフレームを削除した際には、Emacs は自動的にこの関数を呼び出す (Section 29.7 [Deleting Frames], page 730 を参照)。

`delete-terminal-functions` [Variable]
`delete-terminal`により実行されるアブノーマルフック。各関数は`delete-terminal`に渡された *terminal*を唯一の引数として受け取る。技術的な詳細により、この関数は端末の削除の直前または直後のいずれかに呼び出される。

数は多くありませんが、Lisp 変数のいくつかは端末ローカル *(terminal-local)* です。つまりそれらは端末それぞれにたいして個別にバインディングをもちます。いかなるときも実際に効果をもつバインディングはカレントで選択されたフレームに属する端末にたいして 1 つだけです。これらの変数には`default-minibuffer-frame`、`defining-kbd-macro`、`last-kbd-macro`、`system-key-alist`が含まれます。これらは常に端末ローカルであり、決してバッファーローカル (Section 12.11 [Buffer-Local Variables], page 190 を参照) にはできません。

GNU および Unix システムでは、X ディスプレイはそれぞれ別のグラフィカル端末になります。X ウィンドウシステム内で Emacs が開始された際は環境変数`DISPLAY`、または '`--display`'オプション (Section "Initial Options" in *The GNU Emacs Manual* を参照) により指定された X ディスプレイを使用します。Emacs はコマンド`make-frame-on-display`を通じて別の X ディスプレイに接続できます。それぞれの X ディスプレイは、それぞれが選択されたフレームとミニバッファーをもちます。しかしあらゆる瞬間 (Section 29.10 [Input Focus], page 732 を参照) において、それらのフレームのうちの 1 つだけが、いわゆる選択されたフレームになります。`emacsclient`との対話により、Emacs が別のテキスト端末と接続することさえ可能です。Section "Emacs Server" in *The GNU Emacs Manual* を参照してください。

1 つの X サーバーが 1 つ以上のディスプレイを処理できます。各 X ディスプレイには '*hostname*`:`*displaynumber*`.`*screennumber*'という 3 つの部分からなる名前があります。1 つ目の部分の *hostname*はその端末が物理的に接続されるマシン名です。2 つ目の部分の *displaynumber* は同じキーボードとポインティングデバイス (マウスやタブレット等) を共有するマシンに接続された 1 つ以上のモニターを識別するための 0 基準の番号です。3 つ目の部分の *screennumber*は、その X サーバー上の単一のモニターコレクション (a single monitor collection) の一部である 0 基準のスクリーン番号 (個別のモニター) です。1 つのサーバー配下にある 2 つ以上のスクリーンを使用する際には、Emacs はそれらの名前の同一部分から、それらが単一のキーボードを共有することを知ることができるのです。

MS-Windows のように X ウィンドウシステムを使用しないシステムは X ディスプレイの概念をサポートせず、各ホスト上には 1 つのディスプレイだけがあります。これらのシステム上のディスプレイ名は上述したような 3 つの部分からなる名前にしたがいません。たとえば MS-Windows システム上のディスプレイ名は文字列定数 '`w32`'です。これは互換性のために存在するものであり、ディスプレイ名を期待する関数にこれを渡すことができます。

`make-frame-on-display` *display* **&optional** *parameters* [Command]
この関数は *display* 上に新たにフレームを作成してそれをリターンする。その他のフレームパラメーターは、*parameters*という alist から取得する。*display*は X ディスプレイの名前 (文字列) であること。

この関数はフレーム作成前に Emacs がグラフィックを表示するためにセットアップされることを保証する。(テキスト端末上で開始された等で) たとえば Emacs が X リソースを未処理ならこの時点で処理を行う。他のすべての点においては、この関数は`make-frame`(Section 29.1 [Creating Frames], page 697 を参照) と同様に振る舞う。

`x-display-list` [Function]
この関数は Emacs がどの X ディスプレイに接続したかを識別するリストをリターンする。このリストの要素は文字列で、それぞれがディスプレイ名を表す。

`x-open-connection` *display* **&optional** *xrm-string must-succeed* [Function]
この関数はディスプレイ上にフレームを作成することなく、X ディスプレイ *display*への接続をオープンする。通常は `make-frame-on-display`が自動的に呼び出すので、Emacs Lisp プログラムがこの関数を呼び出す必要はない。これを呼び出す唯一の理由は、与えられた X ディスプレイにたいして通信を確立できるかどうかチェックするためである。

オプション引数 *xrm-string*が非 `nil`なら、それは`.Xresources`ファイル内で使用されるフォーマットと同一なリソース名とリソース値。Section "X Resources" in *The GNU Emacs Manual* を参照のこと。これらの値はその X サーバー上で記録されたリソース値をオーバーライドして、このディスプレイ上で作成されるすべての Emacs フレームにたいして適用される。以下はこの文字列がどのようなものかを示す例:

```
"*BorderWidth: 3\n*InternalBorder: 2\n"
```

*must-succeed*が非 `nil`なら、接続オープンの失敗により Emacs が終了させられる。それ以外なら通常の Lisp エラーとなる。

`x-close-connection` *display* [Function]
この関数はディスプレイ *display*への接続をクローズする。これを行う前には、まずそのディスプレイ上でオープンしたすべてのフレームを削除しなければならない (Section 29.7 [Deleting Frames], page 730 を参照)。

マルチモニターのセットアップにおいて、単一の X ディスプレイが複数の物理モニターに出力される場合があります。そのようなセットアップを取得するために関数 `display-monitor-attributes-list`と `frame-monitor-attributes`を使用できます。

`display-monitor-attributes-list` **&optional** *display* [Function]
この関数は *display* 上の物理モニターの属性のリストをリターンする。*display*にはディスプレイ名 (文字列)、端末、フレームを指定でき、省略または `nil`の場合のデフォルトは選択されたフレームのディスプレイ。このリストの各要素は物理モニターの属性を表す連想リスト。1 つ目の要素はプライマリーモニターである。以下は属性のキーと値:

'`geometry`'
'`(x y width height)`'のようなピクセル単位でのそのモニターのスクリーンの左上隅の位置とサイズ。そのモニターがプライマリーモニターでなければ、いくつかの座標が負になり得る。

'`workarea`'
'`(x y width height)`'のようなピクセル単位でのワークエリア (使用可能なスペース) の左上隅の位置とサイズ。ウィンドウマネージャーのさまざまな機能 (dock、taskbar 等) によりそのスペースが占有される '`geometry`'とは異なり、これはワークエリアから除外され得る。そのような機能が実際にワークエリアから差し引かれるかどうかは、そのプラットフォームと環境に依存する。繰り返しになるが、そのモニターがプライマリーモニターでなければ、いくつかの座標は負になり得る。

'`mm-size`' '`(width height)`'<のようなミリメートル単位での幅と高さ。

'`frames`' その物理モニターが支配 (dominate) するフレームのリスト (以下参照)。

‘name’ *string*のようなその物理モニターの名前。

‘source’ *string*のようなマルチモニターの情報ソース (例: ‘XRandr’、‘Xinerama’等)。

x、*y*、*width*、*height*は整数。‘name’と‘source’は欠落しているかもしれない。

あるモニター内にフレームの最大領域がある、または (フレームがどの物理モニターにも跨がらないなら) そのモニターがフレームに最も近いとき、フレームは物理モニターにより支配 *(dominate)* される。グラフィカルなディスプレイ内の (ツールチップではない) すべてのフレームは、たとえそのフレームが複数の物理モニターに跨がる (または物理モニター上にない) としても、(可視か否かによらず) 正確に 1 つの物理モニターにより支配される。

以下は 2 つのモニターディスプレイ上でこの関数により生成されたデータの例:

```
(display-monitor-attributes-list)
⇒
(((geometry 0 0 1920 1080) ;; 左手側プライマリーモニター
  (workarea 0 0 1920 1050) ;; タスクバーが幾分かの高さを占有
  (mm-size 677 381)
  (name . "DISPLAY1")
  (frames #<frame emacs@host *Messages* 0x11578c0>
          #<frame emacs@host *scratch* 0x114b838>))
 ((geometry 1920 0 1680 1050) ;; 右手側モニター
  (workarea 1920 0 1680 1050) ;; スクリーン全体を使用可
  (mm-size 593 370)
  (name . "DISPLAY2")
  (frames)))
```

`frame-monitor-attributes` **&optional** *frame* [Function]
この関数は *frame*を支配 (上記参照) する物理モニターの属性をリターンする。 *frame*のデフォルトは選択されたフレーム。

マルチモニターディスプレーではフレームを指定したモニター上にするためにコマンド`make-frame-on-monitor`を使用することが可能です。

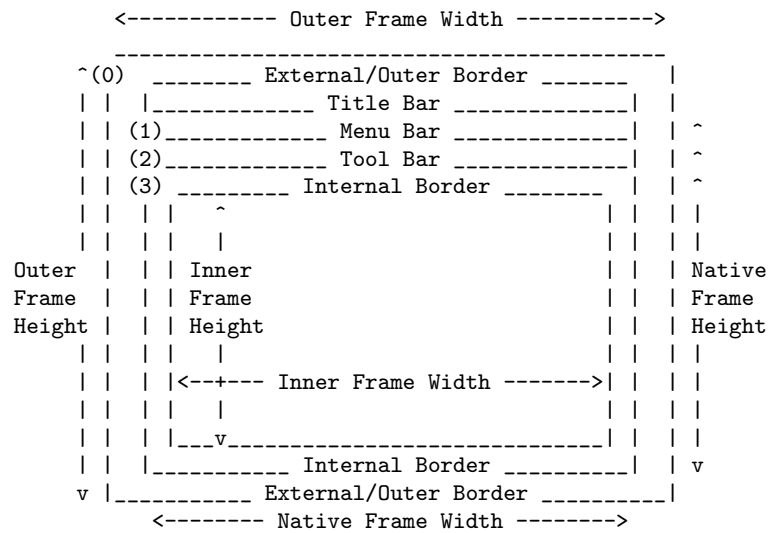
`make-frame-on-monitor` *monitor* **&optional** *display parameters* [Command]
この関数は *display*上に配置される *monitor*に新たにフレームを作成してそれをリターンする。その他のフレームパラメーターは、*parameters*という alist から取得する。*monitor*は物理モニター名であり、`display-monitor-attributes-list`関数のリターン値の属性`name`の文字列と同一の文字列であること。*display*は X ディスプレイの名前 (文字列) であること。

29.3 フレームのジオメトリー

フレームのジオメトリー (geometry) は、その Emacs インスタンスのビルドに使用されたツールキット、およびそのフレームを表示する端末に依存します。このチャプターではこれらの依存関係とそれらを処理するいくつかの関数を説明します。これらの関数すべてにたいして、*frame*引数には生きたフレームを指定する必要があることに注意してください (Section 29.7 [Deleting Frames], page 730 を参照)。省略または`nil`なら選択されたフレーム (Section 29.10 [Input Focus], page 732 を参照) が指定されます。

29.3.1 フレームのレイアウト

可視なフレームは端末のディスプレイの矩形 (rectangular) の領域を占有します。この領域にはそれぞれが異なる用途をサービスする、いくつかのネストされた矩形を含むことができます。以下のスケッチはグラフィカル端末上でのレイアウトを描いたものです:



実際のところ上図で示した領域すべてが存在するわけではありません。これらの領域については以下で説明します。

アウターフレーム (Outer Frame)

> アウターフレーム *(outer frame)* とは上図で示すすべての領域を網羅する矩形。この矩形の端はそのフレームのアウターエッジ *(outer edges)* と呼ばれる。フレームのアウター幅 *(outer width)* とアウター高さ *(outer height)* は併せて矩形のアウターサイズ *(outer size)* を指定する。
>
> フレームのアウターサイズを把握することはフレームのディスプレイの作業領域にフレームをフィットさせたり、スクリーンで 2 つのフレームを隣接して配置するのに有用である (Section 29.2 [Multiple Terminals], page 698 を参照)。アウターサイズは通常はフレームが少なくとも 1 回マップ (可視にすること。Section 29.11 [Visibility of Frames], page 736 を参照) された後でなければ利用できない。初期フレームやまだ作成されていないフレームにたいするアウターサイズは予想するかウィンドウシステムやウィンドウマネージャーのデフォルトから計算しなければならない。回避策としてはマップ済みフレームのアウターサイズとネイティブサイズを取得して、新たなフレームのアウターサイズの計算に使用すればよい。
>
> (上図 '`(0)`'で示される) アウターフレームの左上隅の位置はフレームのアウター位置 *(outer position)* と呼ばれる。グラフィカルなフレームのアウター位置は、ディスプレイではフレームのリサイズやレイアウト変更では未変更のままなので、そのフレームの "位置" としても参照される。
>
> アウター位置はフレームパラメーター `left`と `top`を通じて指定とセットができる (Section 29.4.3.2 [Position Parameters], page 715 を参照)。これらのパラメーターは通常のトップレベルのフレームにたいして、通常はフレームのディスプレイの原点からみた絶対位置 (以下参照) を表す。子フレーム (Section 29.14 [Child Frames], page 739

を参照) にたいしては親フレームのネイティブ位置 (以下参照) から相対的な位置を表す。これらのパラメーターの値はテキスト端末のフレームでは意味がなく常に 0 である。

エクスターナルボーダー (External Border)
: エクスターナルボーダー *(external border)* はウィンドウマネージャーにより提供される装飾の一部。典型的にはマウスによるフレームのリサイズで典型的に使用される。そのため “全画面化 (fullboth)” や最大化されたフレームでは表示されない (Section 29.4.3.3 [Size Parameters], page 717 を参照)。エクスターナルボーダーの幅はウィンドウマネージャーにより決定されて、Emacs の関数では変更できない。

  テキスト端末のフレームではエクスターナルボーダーは存在しない。グラフィカルなフレームではフレームパラメーター `override-redirect`と `undecorated`をセットすることにより表示を抑制できる (Section 29.4.3.8 [Management Parameters], page 723 を参照)。

アウターボーダー (Outer Border)
: アウターボーダー *(outer border)* はフレームパラメーター `border-width` (Section 29.4.3.4 [Layout Parameters], page 719 を参照) で指定可能な分割線。実際にはエクスターナルボーダーとアウターボーダーのいずれかが表示されるが、両方が同時に表示されることはない。アウターボーダーはツールチップフレーム (Section 39.25 [Tooltips], page 1084 を参照)、子フレーム (Section 29.14 [Child Frames], page 739 を参照)、および `undecorated`や `override-redirect`をセットされたフレーム (Section 29.4.3.8 [Management Parameters], page 723 を参照) のようにウィンドウマネージャーにより (完全に) 制御される特別なフレームでは通常は表示されない。

  アウターボーダーはテキスト端末のフレームと GTK+ルーチンが生成したフレームでは表示されない。MS-Windows ではアウターボーダーはピクセル幅が 1 のエクスターナルボーダーの助けを借りてエミュレートされる。X 上でのツールキットによらないビルドではフレームパラメーター `border-color`をセットすることによりアウターボーダーのカラーを変更できる (Section 29.4.3.4 [Layout Parameters], page 719 を参照)。

タイトルバー (Title Bar)
: タイトルバー *(title bar)*、いわゆるキャプションバー *(caption bar)* もウィンドウマネージャーの装飾の一部であり通常はフレームのタイトル (Section 29.6 [Frame Titles], page 730 を参照)、同様に最小化や最大化、削除のボタンを表示する。これはマウスによるフレームのドラッグにも使用できる。タイトルバーは通常は全画面化 (fullboth) されたフレーム (Section 29.4.3.3 [Size Parameters], page 717 を参照)、ツールチップフレーム (Section 39.25 [Tooltips], page 1084 を参照)、子フレーム (see Section 29.14 [Child Frames], page 739 を参照) では表示されず、端末フレームでは存在しない。タイトルバーの表示はフレームパラメーター `override-redirect`や `undecorated`をセットすることにより抑制できる (Section 29.4.3.8 [Management Parameters], page 723 を参照)。

メニューバー (Menu Bar)
: メニューバー (Section 22.17.5 [Menu Bar], page 453 を参照) にはインターナル (Emacs 自身が描画) とエクスターナル (ツールキトが描画) がある。ほとんどのビルド (GTK+、Lucid、Motif、MS-Windows) ではエクスターナルメニューバーを依拠とする。NS もエクスターナルメニューバーを使用するが、これはアウターフレームの一部ではない。非ツールキットのビルドはインターナルメニューバーを提供できる。テキスト端末フレームではメニューバーはフレームのルートウィンドウの一部である (Section 28.2

[Windows and Frames], page 609 を参照)。ルールとして子フレームでメニューバーが表示されることはない (Section 29.14 [Child Frames], page 739 を参照)。パラメーター `menu-bar-lines` (Section 29.4.3.4 [Layout Parameters], page 719 を参照) をセットすることによりメニューバーの表示は抑制できる。

メニューバーの幅がフレームにフィットするには大きくなりすぎた際に折り返されるか (wrapped)、それとも切り詰められるか (truncated) はツールキットに依存する。通常は Motif と MS-Windows のビルドだけがニューバーを折り返すことができる。これらはメニューバーの折り返し、またはそれを解除する際にフレームのアウター高さの保持を試みるの、でかわりにフレームのネイティブ高さ (以下参照) が変更される。

ツールバー (Tool Bar)

メニューバーと同じように、ツールバーにはインターナルツールバー (Emacs が描画) とエクスターナルツールバー (ツールキットが描画) がある。GTK+と NS のビルドにはそれらのツールキットが描画するツールバーがある。その他のビルドはインターナルツールバーを使用する。GTK+ではフレームのインタナルボーダー (以下参照) のすぐ外側のいずれかのサイドにツールバーを配置できる。子フレームは通常はツールバーを表示しない (Section 29.14 [Child Frames], page 739 を参照)。パラメーター `tool-bar-lines` (Section 29.4.3.4 [Layout Parameters], page 719 を参照) に 0 をセットすることでツールバーの表示を抑制できる。

変数 `auto-resize-tool-bars`が非 `nil`なら、フレームに収まらないほど幅が大きくなると Emacs はツールバーを折り返す。Emacs がインターナルツールバーの折り返しや折り返しの解除を行う場合には、デフォルトではフレームのアウター高さを未変更に保つので、かわりにフレームのネイティブ高さ (以下参照) が変更される。一方 GTK+とともにビルドされた Emacs ではツールバーの折り返しは決して行われないが、長くなりすぎたツールバーが収まるようにフレームのアウター幅が自動的に増加される。

ネイティブフレーム (Native Frame)

ネイティブフレーム *(native frame)* は全体的にアウターフレーム内に配置される。ネイティブフレームにはエクスターナルボーダーとアウターボーダー、タイトルバーとエクスターナルメニューバーとツールバーが占有する領域は含まれない。ネイティブフレームのエッジ (edge: 端) はフレームのネイティブエッジ *(native edges)* と呼ばれる。フレームのネイティブサイズ *(native size)* ハ、フレームのネイティブ幅 *(native width)* とネイティブ高さ *(native height)* で指定される。

フレームのネイティブサイズは Emacs 内のフレームの作成やリサイズを Emacs が行う際にウィンドウシステムやウィンドウマネージャーに渡すサイズである。これはたとえばタイトルバーの対応するボタンのクリックによりフレームを最大化した後やマウスでエクスターナルボーダーをドラッグした際等、フレームの (ウィンドウシステムの) ウィンドウをリサイズする際にウィンドウシステムやウィンドウマネージャーに渡すサイズでもある。

ネイティブフレームの左上隅の位置はフレームのネイティブ位置 *(native position)* を指定する。上図の (1) から (3) は種々のビルドにたいするネイティブ位置を示す。

(1) 非ツールキットと端末フレーム

(2) Lucid、Motif、MS-Windows のフレーム

(3) GTK+と NS のフレーム

したがって Lucid、Motif、MS-Windows ではネイティブ高さにツールバーの高さは含まれるがメニューバーの高さは含まれず、非ツールキットおよび端末のフレームではメニューバーとツールバーの高さは含まれない。

フレームのネイティブ位置はマウスのカレント位置 (Section 29.16 [Mouse Position], page 743 を参照) のセットやリターンを行う関数や `window-edges`、`window-at`、`coordinates-in-window-p`のようにウィンドウ位置 (Section 28.24 [Coordinates and Windows], page 681 を参照) を扱う関数にたいして参照位置となる。これはフレームの子フレームの配置や位置にたいする原点 (0, 0) も指定する。

フレームパラメーター `override-redirect`や `undecorated` (Section 29.4.3.8 [Management Parameters], page 723 を参照) を変更してウィンドウマネージャーの装飾の削除や追加を行う際にも、フレームのネイティブ位置は未変更のままであることにも注意。

インターナルボーダー (Internal Border)
: インターナルボーダーはインナーフレーム (以下参照) の周辺に Emacs が描画するボーダーである。インターナルボーダーの幅はフレームパラメーター `internal-border-width`により指定される (Section 29.4.3.4 [Layout Parameters], page 719 を参照)。インターナルボーダーのカラーは `internal-border`フェイスのバックグラウンドで指定される。

インナーフレーム (Inner Frame)
: インナーフレーム *(inner frame)* はフレームのウィンドウにたいして予約された矩形のこと。インナーフレームはインターナルボーダー (これはインナーフレームの一部ではない) に囲われている。インナーフレームのエッジはフレームのインナーエッジ *(inner edges)* と呼ばれる。この矩形のインナーサイズ *(inner size)* はインアー幅 *(inner width)* とインナー高さ *(inner height)* により指定される。このインナーフレームはフレームのディスプレイエリア *(display area)* として参照されることもある。

  ルールとしてインナーフレームはフレームのルートウィンドウ (Section 28.2 [Windows and Frames], page 609 を参照) とミニバッファーウィンドウ (Section 20.11 [Minibuffer Windows], page 371 を参照) に細分される。この 2 つには注目すべき 2 つの例外がある。それはミニバッファーウィンドウをもたないルートウィンドウのみのミニバッファー *less* フレーム *(minibuffer-less frame)* と、ミニバッファーウィンドウだけをもち、それがフレームのルートウィンドウの役目も果たすミニバッファー *only* ウィンドウ *(minibuffer-only frame)* である。そのようなフレーム構成を作成する方法は Section 29.4.2 [Initial Parameters], page 713 を参照のこと。

テキストエリア (Text Area)
: フレームのテキストエリア *(text area)* はネイティブフレームに埋め込み可能な一種の架空領域である。テキストエリアの位置は指定されない。テキストエリアの幅はネイティブ幅からインターナルボーダーの幅、そのフレームに指定されていれば (Section 29.4.3.4 [Layout Parameters], page 719 を参照)1 つの垂直スクロールバーの幅、左右のフリンジ各 1 の幅を取り除くことにより取得できる。テキストエリアの高さはネイティブ高さからインターナルボーダーの幅、そのフレームに指定されていればフレームのインターナルメニューバーとインターナルツールバーの高さ、1 つの水平スロールバーの高さを取り除くことにより取得できる。

フレームの絶対位置 *(absolute position)* は (X, Y) というペア、またはフレームのディスプレイの原点 (0, 0) から相対的な水平および垂直のピクセル単位のオフセットにより与えられる。これに対応してフレームの絶対エッジ *(absolute edges)* はこの原点からのピクセル単位のオフセットにより与えられる。

マルチモニターではディスプレイの原点が端末の利用可能な表示エリア全体の左上隅と一致する必要はない。したがってそのような環境では、たとえそのフレームが完全に可視であってもフレームの絶対位置は負の値になり得る。

慣例により垂直方向のオフセットは “下方向” にたいして増加する。これはフレームの下エッジから上エッジのオフセットを減ずることによりフレームの高さが得られることを意味する。期待されるように水平方向のオフセットは “右方向” にたいして増加するので、フレームの右エッジから左エッジのオフセットを減ずることによりフレームの幅を計算できる。

以下の関数はグラフィカル端末上のフレームにたいして上述したエリアのサイズをリターンします:

`frame-geometry` **&optional** *frame* [Function]

この関数は *frame*の幾何学的な属性をリターンする。リタ^ン値は以下のような属性のリストの連想リスト。すべての座標、高さや幅の値はピクセル単位の整数。まだ *frame*がマップされていなければ (Section 29.11 [Visibility of Frames], page 736 を参照)、いくつかのリターン値は実際の値の近似値しか表していないかもしれない (それらの値はフレームのマップ後に確認可能になる)。

`outer-position`
: *frame*のアウターフレームの絶対位置を表すコンスであり、*frame*のディスプレイの原点 (0, 0) から相対的な位置。

`outer-size`
: *frame*のアウター幅とアウター高さを表すコンス。

`external-border-size`
: ウィンドウマネージャーにより与えられる、*frame*エクスターナルボーダーの水平幅と垂直幅を表すコンス。ウィンドウマネージャーによりこれらの値が提供されなければ、Emacs はアウターフレームとインナーフレームの座標からそれらの推測を試みる。

`outer-border-width`
: *frame*のアウターボーダーの幅。この値は非 GTK+の X ビルドでのみ意味がある。

`title-bar-size`
: ウィンドウマネージャーまたはオペレーティングシステムが与える、*frame*のタイトルバーの幅と高さを表すコンス。いずれも 0 なら、そのフレームにタイトルバーはない。幅だけが 0 なら、Emacs が幅の情報を取得できなかったことを意味する。

`menu-bar-external`
: 非`nil`なら、それはメニューバーがエクスターナルである (*frame*のネイティブフレームの一部ではない) ことを意味する。

`menu-bar-size`
: *frame*のメニューバーの幅と高さを表すコンス。

`tool-bar-external`
: 非`nil`なら、それはツールバーがエクスターナルである (*frame*のネイティブフレームの一部ではない) ことを意味する。

`tool-bar-position`
: これはツールバーが *frame*のどの端に配置されているかを示し `left`、`top`、`right`、`bottom`のいずれか。現在のところ `top`以外の値をサポートするツールキットは GTK+のみ。

`tool-bar-size`
: *frame*のツールバーの幅と高さを表すコンス。

`internal-border-width`
: *frame*のインターナルボーダーの幅。

以下の関数はフレームにたいするアウター、ネイティブ、インナーのエッジの取得に使用できます。

`frame-edges` **&optional** *frame type* [Function]
: この関数は *frame*のアウター、ネイティブ、インナーフレームの絶対エッジをリターンする。*frame*は生きたフレームでなければならずデフォルトは選択されたフレーム。リターンされるリストは (*left top right bottom*) という形式をもつ。すべて *frame*のディスプレイの原点から相対的なピクセル単位の値。端末フレームでは *left*と *top*にたいしてリターンされる値は常に 0。

オプション引数 *type*はリターンするエッジのタイプを指定する。`outer-edges`は *frame*のアウターエッジ、`native-edges` (か `nil`) はネイティブエッジ、`inner-edges`はインナーエッジをリターンすることを意味する。

慣例により *left*と *top*にたいしてリターンされたディスプレイのピクセル位置は *frame*の内部 (一部) とみなされる。したがって *left*と *top*がいずれも 0 なら、ディスプレイの原点のピクセル位置は *frame*の一部となる。その一方で *bottom*と *right*のピクセル位置は *frame*のすぐ外側にあるとみなされる。これはたとえば 2 つの横並びのフレームがあり、左フレームの右アウトーエッジが右フレームの左エッジと等しければ、そのエッジ上のピセルは右フレームの一部として表されることを意味する。

29.3.2 フレームのフォント

フレームにはそれぞれ、そのフレームにたいするデフォルト文字サイズを指定するデフォルトフォント *(default font)* があります。このサイズは、行や列の単位でのフレームサイズの取得や変更での使用を意図したものです (Section 29.4.3.3 [Size Parameters], page 717 を参照)。これはウィンドウのリサイズ (Section 28.3 [Window Sizes], page 613 を参照) や分割 (Section 28.6 [Splitting Windows], page 622 を参照) の際にも使用されます。

“デフォルト文字高さ (default character height)t” のかわりに行高さ *(line height)* や正準文字高さ *(canonical character height)* という用語を使用するときがあります。同様に “デフォルト文字幅 (default character width)” のかわりに列幅 *(column width)* や正準文字幅 *(canonical character width)* という用語も使用されます。

`frame-char-height` **&optional** *frame* [Function]
`frame-char-width` **&optional** *frame* [Function]
: これらの関数はピクセルで測った *frame*内の文字のデフォルトの高さまたは幅をリターンする。両者をあわせたサイズにより *frame*の *frame*のサイズが確立される。値は *frame*にたいして選択されたフォントに依存する。Section 29.4.3.10 [Font and Color Parameters], page 726 を参照のこと。

以下の関数でデフォルトフォントを直接セットすることもできます:

`set-frame-font` *font* **&optional** *keep-size frames* [Command]
: これはデフォルトフォントに *font*をセットする。インタラクティブに呼び出された際にはフォント名の入力を求めて、選択されたフレームにそのフォントを使用する。Lisp から呼び出す際には、*font*はフォント名 (文字列)、フォントオブジェクト、フォントエンティティー、フォント spec のいずれかであること。

オプション引数 *keep-size*が `nil`ならフレームの行数と列数を固定に保つ (非 `nil`なら次セクションで説明するオプション `frame-inhibit-implied-resize`がこれをオーバーライドするだろう)。*keep-size*が非 `nil` (またはプレフィクス引数を指定) なら行数と列数を調節することにより、カレントフレームのディスプレーエリアのサイズの維持を試みる。

オプション引数 *frames*が `nil`なら、そのフォントは選択されたフレームだけに適用される。*frames*が非 `nil`ならそれは作用するフレームのリスト、またはすべての既存フレームおよび将来のすべてのグラフィカルフレームを意味する `t`のいずれかであること。

29.3.3 フレームの位置

グラフィカルなシステムでは通常のトップレベルのフレームの位置はアウターフレームの絶対位置として指定されます (Section 29.3 [Frame Geometry], page 701 を参照)。子フレーム (Section 29.14 [Child Frames], page 739 を参照) の位置は親フレームのネイティブ位置から子フレームのアウターエッジまでのピクセル単位のオフセットとして相対的に指定されます。

フレーム位置はフレームパラメーター `left`と `top`(Section 29.4.3.2 [Position Parameters], page 715 を参照) を使用すれば変更やアクセスができます。既存の可視なフレームの位置を処理するために追加で 2 つの関数が存在します。いずれの関数でも引数 *frame*は生きたフレームでなければならず、デフォルトは選択されたフレームです。

`frame-position` **&optional** *frame* [Function]

この関数は子フレームではない通常のフレームにたいしてフレームのアウター位置 (Section 29.3.1 [Frame Layout], page 702 を参照) を、フレームのディスプレイの原点 `(0, 0)`からのピクセル座標をコンスセルでリターンする。子フレーム (Section 29.14 [Child Frames], page 739 を参照) にたいしてはフレームのアウター位置を、*frame*のネイティブ位置を原点 `(0, 0)`として、そこからのピクセル座標をリターンする。

負の値は *frame*のディスプレイまたは親フレームの右エッジか下エッジからのオフセットではない。これらは *frame*のアウター位置が、フレームのデイスプレイの原点または親フレームのネイティブ位置の左および/または上にあることを意味する。これは通常は *frame*の一部だけが可視 (または完全に不可視) であることを意味している。しかしディスプレイの原点がディスプレイの左上隅と一致しないシステムではセカンダリーモニター上ではフレームは可視かもしれない。

てきすと端末ではいずれの値も 0。

`set-frame-position` *frame x y* [Function]

この関数は *frame*のアウターフレームの位置を (*x*, *y*) にセットする。後の引数は通常は *frame*のディスプレイの位置 (0, 0) にある原点からのピクセル数、子フレームでは *frame*の親フレームのネイティブ位置からのピクセル数。

負のパラメーター値はアウターフレームの右エッジをスクリーンの右エッジ (または親フレームのネイティブ矩形位置) から左に-*x*ピクセル、または下エッジをスクリーンの下エッジ (または親フレームのネイティブ矩形位置) から上に-*y*ピクセルの位置にセットする。

負の値では *frame*の右エッジや下エッジを性格にディスプレイや親フレームの右エッジや下エッジには揃えられないことに注意。負の値ではディスプレイや親フレーム内にあるエッジにある位置は指定できない。フレームパラメーター `left`と `top` (Section 29.4.3.2 [Position Parameters], page 715 を参照) でこれを行うことはできるものの、初期フレームや新たなフレームでは依然として良好な結果は得られないかもしれない。

この関数はテキスト端末フレームでは効果がない。

`move-frame-functions` [Variable]
このフックはウィンドウシステムやウィンドウマネージャーがフレームを (新たな位置を割り当てて) 移動した際に実行される関数を指定する。関数は移動されたフレームを単一の引数として実行される。子フレーム (Section 29.14 [Child Frames], page 739 を参照) では親フレームとの関連性においてのフレーム位置が変更されたときだけ関数が実行される。

29.3.4 フレームのサイズ

Emacs 内からフレームサイズ *(size of a frame)* を指定する正規の方法はテキストサイズ *(text size)* の指定による方法です (フレームのテキストエリアの幅と高さの組み合わせ。Section 29.3.1 [Frame Layout], page 702 を参照)。これはピクセルやフレームの標準の文字サイズ (Section 29.3.2 [Frame Font], page 707 を参照) で計ることができます。

インターナルメニューやインターナルツールバーのあるフレームでは実際にフレームが描画されるまでフレームのネイティブ高さを告げることはできません。一般的にこれはフレームの初期サイズの指定にネイティブサイズを使用できないことを意味しています。可視フレームのネイティブフレームが解り次第、`frame-geometry`のリターン値から残りのコンポーネントを追加してアウターサイズ (Section 29.3.1 [Frame Layout], page 702 を参照) を計算できます。しかし不可視のフレームやまだ作成されていないフレームにたいするアウターサイズは推定しかできません。これはスクリーンの右エッジや下エッジからのオフセットの指定ではフレームの正確な初期位置を計算することが不可能であることも意味しています (Section 29.3.3 [Frame Position], page 708 を参照)。

フレームのテキストサイズはフレームパラメーター `height`と `width`を使用してセットや取得が可能です (Section 29.4.3.3 [Size Parameters], page 717 を参照)。初期フレームのテキストサイズは X 様式のジオメトリー仕様でもセットや取得が可能です。Section “Command Line Arguments for Emacs Invocation” in *The GNU Emacs Manual* を参照してください。以下に既存で可視なフレーム (デフォルトは選択されたフレーム) のサイズにたいしてセットやアクセスを行う関数をいくつかリストします。

`frame-height` **&optional** *frame* [Function]
`frame-width` **&optional** *frame* [Function]
これらの関数は *frame*のテキストエリアの高さと幅を、*frame*のデフォルトフォントの高さと幅を単位に計測してリターンする。これらの関数は単に `(frame-parameter frame 'height)` と `(frame-parameter frame 'width)`を略記したもの。

ピクセルで計測した *frame*のテキストエリアがデフォルトフォントサイズの倍数でなければ、これらの関数がリターンする値はテキストエリアに完全に収まるデフォルトフォントの文字数に切り捨てられる。

以下の関数は与えられたフレームのネイティブフレーム、アウターフレーム、インナーフレーム、テキストエリアのピクセル幅とピクセル高さをリターンします (Section 29.3.1 [Frame Layout], page 702 を参照)。テキスト端末では結果はピクセルではなく文字単位になります。

`frame-outer-width` **&optional** *frame* [Function]
`frame-outer-height` **&optional** *frame* [Function]
これらの関数は *frame*のアウター幅やアウター高さをピクセル単位でリターンする。

`frame-native-height` **&optional** *frame* [Function]
`frame-native-width` **&optional** *frame* [Function]
これらの関数は *frame*のネイティブ幅やネイティブ高さをピクセル単位でリターンする。

`frame-inner-width` **&optional** *frame* [Function]
`frame-inner-height` **&optional** *frame* [Function]
これらの関数は *frame*のインナー幅やインナー高さをピクセル単位でリターンする。

`frame-text-width` **&optional** *frame* [Function]
`frame-text-height` **&optional** *frame* [Function]
これらの関数は *frame*のテキストエリアの幅や高さをピクセル単位でリターンする。

ウィンドウシステムがサポートしていれば、Emacs はデフォルトでフレームのピクセル単位でのテキストサイズをフレームの文字サイズの倍数にしようと試みます。しかし通常これはエクスターナルボーダーのドラッグ時にフレームが文字サイズの増減だけでリサイズできることを意味しています。さらにこれはフレームの下および/または右に空のスペースが残ることにより、フレームを最大化したり “fullheight” や “fullwidth” にする試みを阻害するかもしれません。このような場合には以下のオプションが助けになるでしょう。

`frame-resize-pixelwise` [User Option]
このオプションが `nil` (デフォルト) ならフレームのテキストのピクセルサイズは、フレームのリサイズの際に通常は `frame-char-height`と `frame-char-width`のカレント値の倍数に丸められる。非 `nil`なら丸めは行われず、フレームのサイズはピクセル単位で増加/減少が可能になる。

この変数をセットすることにより次回のリサイズ処理では、通常はウィンドウマネージャーにこれに相当するサイズのヒントを渡す。これはユーザーの初期ファイル内でのみこの変数をセットすべきで、アプリケーションが一時的にこれをバインドすべきではないことを意味する。

このオプションにたいして `nil`値がもつ正確な意味は使用されるツールキットに依存する。マウスによるエクスターナルボーダーのドラッグは、ウィンドウマネージャーが対応するサイズヒントを処理する意思があれば文字単位で行われる。文字サイズの整数倍ではないフレームサイズを引数として `set-frame-size` (以下参照) を呼び出すと、もしかしたら丸められたり (GTK+)、あるいは受容される (Lucid、Motif、MS-Windows) かもしれない。

いくつかのウィンドウマネージャーでは、フレームを本当に最大化や全画面で表示させるために、これを非 `nil`にセットする必要があるかもしれない。

`set-frame-size` *frame width height* **&optional** *pixelwise* [Function]
この関数は *frame*のテキストエリアのサイズを、*frame*の文字の正準高さと正準幅で計測した単位でセトする (Section 29.3.2 [Frame Font], page 707 を参照)。

オプション引数 *pixelwise*が非 `nil`なら、かわりにピクセル単位で新たな幅と高さを測ることを意味する。`frame-resize-pixelwise`が `nil`の場合には、それが文字の整数倍でフレームサイズを増加あるいは減少させないなら、この要求を完全にはしたがわずに拒絶するツールキットがいくつかあることに注意。

`set-frame-height` *frame height* **&optional** *pretend pixelwise* [Function]
この関数は *frame*のテキストエリアを *height*行の高さにリサイズする。*frame*内の既存ウィンドウのサイズはフレームにフィットするよう比例して変更される。

*pretend*が非 `nil`なら、Emacs は *frame*内で *height*行の出力を表示するが、そのフレームの実際の高さにたいする値は変更しない。これはテキスト端末上でのみ有用。端末が実際に実装するより小さい高さの使用は、より小さいスクリーン上での振る舞いの再現したり、スクリーン全体を使用時の端末の誤動作を観察するとき有用かもしれない。フレームの高さの直接セットは常に機能するとは限らない。なぜならテキスト端末上でのカーソルを正しく配置するために、正確な実サイズを知る必要があるかもしれないからである。

オプションの第 4 引数 *pixelwise*が非 `nil`なら、それは *frame*の高さが *height*ピクセル高くなることを意味する。`frame-resize-pixelwise`が `nil`の場合、それが文字の整数倍でフレームサイズを増加あるいは減少させないなら、この要求に完全にはしたがわずに拒絶するウィンドウマネージャーがいくつかあることに注意。

インタラクティブにこのコマンドを使用時は、カレントで選択されたフレーム高さを数プレフィクスで指定された行数にセットする。

`set-frame-width` *frame width* **&optional** *pretend pixelwise* [Function]

この関数は文字単位で *frame*のテキストエリアの幅をセットする。引数 *pretend*は `set-frame-height`のときと同じ意味をもつ。

オプションの第 4 引数 *pixelwise*が非 `nil`なら、それは *frame*の幅が *height*ピクセル広くなることを意味する。`frame-resize-pixelwise`が `nil`の場合には、それが文字の整数倍でフレームサイズを増加あるいは減少させないなら、この要求に完全にはしたがわずに拒絶するウィンドウマネージャーがいくつかあることに注意。

インタラクティブにこのコマンドを使用時は、カレントで選択されたフレーム幅を数プレフィクスで指定された列数にセットする。

これらの 3 つの関数はスクロールバー、フリンジ、マージン、ディバイダー、モードラインやヘッダーラインと一緒にすべてのウィンドウを表示するために必要な最小よりフレームを小さくしません。これはたとえばマウスによるフレームのエクスターナルボーダーのドラッグなどによるウィンドウマネージャーがトリガーとなる要求と対照的です。このような要求は、もし必要なら表示できないフレームの右下隅の部分をクリッピングすることにより常に尊重されます。Emacs 内からのフレームサイズの変更時に同様の振る舞いを得るには、パラメーター `min-width`と `min-height`を使用できます (Section 29.4.3.3 [Size Parameters], page 717 を参照)。

アブノーマルフック `window-size-change-functions` (Section 28.28 [Window Hooks], page 691 を参照) はウィンドウシステムやウィンドウマネージャーに起因するのもを含む、フレームのインナーサイズの変更のすべてを追跡します。実際にインナーフレームのサイズを変更せずにフレームのウィンドウのサイズだけを変更した際に発生するかもしれない誤検出を除外するために以下の関数を使用できます。

`frame-size-changed-p` **&optional** *frame* [Function]

この関数は *frame*にたいして最後に `window-size-change-functions`が実行されて以降に *frame*のインナー幅かインナー高さが変更されていれば非 `nil`をリターンする。これは *frame*にたいする `window-size-change-functions`の実行直後は常に `nil`をリターンする。

29.3.5 フレームの暗黙的なリサイズ

たとえばメニューバーやツールバーの表示切り替え、デフォルトフォントの変更、フレームのスクロールバーの幅のセットの際には、Emacs はデフォルトではフレームのテキストエリアの行数と列数を未変更に保つように試みます。しかしこれはそのような場合のサイズ変更を調停するために、Emacs がウィンドウマネージャーにフレームのウィンドウのリサイズを依頼しなければならないことを意味します。

たとえばフレームの最大化や全画面化の際のように、そのような暗黙なフレームのリサイズ *(implied frame resizing)* がおそらく望ましくないケースもあります (デフォルトではオフになっている)。一般的には以下のオプションで暗黙のリサイズを無効にできます。

`frame-inhibit-implied-resize` [User Option]

このオプションが `nil`ならフレームのフォント、メニューバー、ツールバー、インターナルボーダー、フリンジ、スクロールバーを変更においてフレームのテキストエリアの列数と行数を維

持するために、アウターフレームがリサイズされるかもしれない。このオプションが tならそのようなリサイズは行われない。

このオプションの値はフレームパラメーターのリストでもよい。この場合にはリスト内に出現するパラメーター変更にたいする暗黙のリサイズは抑制される。このオプションで処理されるフレームパラメーターは現在のところ font、font-backend、internal-border-width、menu-bar-lines、tool-bar-lines。

フレームパラメーター scroll-bar-width、scroll-bar-height、vertical-scroll-bars、horizontal-scroll-bars、left-fringe、right-fringeのいずれかにたいする変更は、あたかもそのフレームが単一の生きたウィンドウを含むかのように処理される。これはたとえば複数の横並びのウィンドウをフレームで垂直スクロールバーを削除すると、このオプションが nilなら nilはスクロールバーの幅の分だけ縮小されて、tや vertical-scroll-barsを含む場合には未変更に保たれることを意味する。

Lucid、Motif、MS-Windows のデフォルト値は'(tab-bar-lines tool-bar-lines)であり、これはツールバーやタブバーの追加や削除でアウターフレーム高さが変更されないことを意味する。GTK+を含むその他すべてのウィンドウシステムでは'(tab-bar-lines)であり、これは上記リストの tab-bar-lines以外のパラメーターのいずれかを変更するとアウターフレームのサイズは変更されるかもしれないことを意味する。それ以外では tであり、これはウィンドウシステムのサポートがなければ暗黙にアウターフレームのサイズが変更されることはないことを意味する。

フレームが上述のいずれかのパラメーターにたいする変更の調停に不十分な際には、たとえこのオプションが非 nilでも Emacs がフレームの拡大を試みるかもしれないことに注意。

ウィンドウマネージャーは通常はエクスターナルメニューバーやエクスターナルツールバーが占有する行数の変更時にフレームのリサイズを要求しないことにも注意。典型的にはこのような “折り返し (wrappings)” はユーザーがフレームのメニューバーやツールバーのすべての要素を表示できないほどフレームを水平方向に縮小しや際に発生する。これはメニューバーやツールバーのアイテム数を変化させるようなメジャーモードの変更によっても発生し得る。このような折り返しはフレームのテキストエリアの行数を暗黙に変更するかもしれ、このオプションのセットによる影響を受けない。

29.4 フレームのパラメーター

フレームにはその外見と挙動を制御する多くのパラメーターがあります。フレームがどのようなパラメーターをもつかは、そのフレームが使用するディスプレイのメカニズムに依存します。

フレームパラメーターは主にグラフィカルディスプレイのために存在します。ほとんどのフレームパラメーターはテキスト端末上のフレームへの適用時には効果がありません。テキスト端末上のフレームで何か特別なことを行うパラメーターは height、width、name、title、menu-bar-lines、buffer-list、buffer-predicateだけです。その端末がカラーをサポートする場合には foreground-color、background-color、background-mode、display-typeなどのパラメーターも意味をもちます。その端末が透過フレーム (frame transparency) をサポートする場合には、パラメーター alphaにも意味があります。

デフォルトでは変数 desktop-restore-framesが非 nilのときには、フレームパラメーターはデスクトップライブラリー関数 (Section 23.8 [Desktop Save Mode], page 518 を参照) が保存とリストアを行います。リストアされたセッションでは無意味や有害になることを避けるためにパラメーターを frameset-persistent-filter-alistに含めるのはアプリケーションの責任です。

29.4.1 フレームパラメーターへのアクセス

以下の関数でフレームのパラメーター値の読み取りと変更ができます。

`frame-parameter` *frame parameter* [Function]
この関数は *frame*のパラメーター *parameter* (シンボル) の値をリターンする。*frame*が `nil` なら選択されたフレームのパラメーターをリターンする。*frame*が *parameter*にたいするセッティングをもたなければ、この関数は `nil`をリターンする。

`frame-parameters` **&optional** *frame* [Function]
関数 `frame-parameters`は *frame*のすべてのパラメーターとその値をリストする alist をリターンする。*frame*が省略または `nil`なら選択されたフレームのパラメーターをリターンする。

`modify-frame-parameters` *frame alist* [Function]
この関数は *alist*の要素にもとづきフレーム *frame*を変更する。*alist*内の要素はそれぞれ `(parm . value)`という形式をもつ。ここで *parm*はパラメーターを名付けるシンボルである。*alist*内に指定されないパラメーターの値は変更されない。*frame*が `nil`の場合のデフォルトは選択されたフレーム。

いくつかのパラメーターは特定の種類のディスプレー上のフレーム (Chapter 29 [Frames], page 696 を参照) でのみ意味がある。*frame*のディスプレーで意味をもたないパラメーターが *alist*に含まれているようなら、この関数はそのフレームのパラメーターリスト内の値を変更するが、その他の値を変更しないパラメーターは無視するだろう。

*frame*の新たなサイズに影響し得る値をもつようなパラメーターが *alist*で複数指定されている際には、フレームの最終的なサイズは使用しているツールキットに応じて異なるかもしれない。たとえばあるフレームにたいしてメニューバーおよび/またはツールバーをもたない状態から保有するように指定して、同時に新たなフレーム高さを指定すると、必然的にフレームの高さの再計算を招くだろう。そのようなケースでは、概念的にはこの関数は明示的な高さ指定を優先するよう試みる。しかしツールキットによりその後に処理されるメニューバーやツールバーの追加 (や削除) を除外することはできないので、高さ指定を優先するという意図は打ち消されてしまうだろう。

ここで概略した問題は、この関数の呼び出し前後に `frame-inhibit-implied-resize` (Section 29.3.5 [Implied Frame Resizing], page 711 を参照) を非 `nil`値にバインドすることで訂正できる場合がある。しかしそのようなバインディングが正に問題を引き起こす場合もある。

`set-frame-parameter` *frame parm value* [Function]
この関数はフレームパラメーター *parm*に指定された *value*をセットする。*frame*が `nil`の場合のデフォルトは選択されたフレーム。

`modify-all-frames-parameters` *alist* [Function]
この関数は *alist*に応じて既存のフレームすべてのフレームパラメーターを変更してから、今後に作成されるフレームに同じパラメーター値を適用するために、`default-frame-alist` (必要なら `initial-frame-alist`も) を変更する。

29.4.2 フレームの初期パラメーター

init ファイル (Section 40.1.2 [Init File], page 1093 を参照) の内部で `initial-frame-alist`をセットすることにより、フレームの初期スタートアップにパラメーターを指定できます。

`initial-frame-alist` [User Option]

この変数の値は初期フレーム作成時に使用されるパラメーター値の alist。以降のフレームを変更することなく初期フレームの外見を指定するためにこの変数を使用できる。要素はそれぞれ以下の形式をもつ:

```
(parameter . value)
```

Emacs は init ファイル読み取り前に初期フレームを作成する。Emacs はこのファイル読み取り後に `initial-frame-alist`をチェックして、変更する値に含まれるパラメーターのセッティングを作成済みの初期フレームに適用する。

これらのセッティングがフレームのジオメトリーと外見に影響する場合には、間違った外見のフレームを目にした後に、指定した外見に変更される様を目にするだろう。これが煩わしければ、X リソースで同じジオメトリーと外見を指定できる。これらはフレーム作成前に効果をもつ。Section “X Resources” in *The GNU Emacs Manual* を参照のこと。

X リソースセッティングは、通常はすべてのフレームに適用される。初期フレームのためにある X リソースを単独で指定して、それ以降のフレームには適用したくなければ、次の方法によりこれを達成できる。それ以降のフレームにたいする X リソースをオーバーライドするために `default-frame-alist`内でパラメーターを指定してから、それらが初期フレームに影響するのを防ぐために `initial-frame-alist`内の同じパラメーターにたいして X リソースにマッチする値を指定すればよい。

これらのパラメーターに `(minibuffer . nil)`が含まれていれば、それは初期フレームがミニバッファーをもつべきではないことを示しています。この場合には、Emacs は同じようにミニバッファー *only* フレーム *(minibuffer-only frame)* を個別に作成します。

`minibuffer-frame-alist` [User Option]

この変数の値は、初期ミニバッファー only フレーム (`initial-frame-alist`がミニバッファーのないフレームを指定する場合に Emacs が作成するミニバッファー only フレーム) を作成時に使用されるパラメーター値の alist。

`default-frame-alist` [User Option]

これはすべての Emacs フレーム (最初のフレームとそれ以降のフレーム) にたいしてフレームパラメーターのデフォルト値を指定する alist。X ウィンドウシステム使用時には、大抵は X リソースで同じ結果を得られる。

この変数のセットは既存フレームに影響しない。さらに別フレームにバッファーを表示する関数は、自身のパラメーターを提供することによりデフォルトパラメーターをオーバーライドできる。

フレームの外見を指定するコマンドラインオプションとともに Emacs を呼び出すと、これらのオプションは `initial-frame-alist`か `default-frame-alist`のいずれかに要素を追加することにより効果を発揮します。‘`--geometry`’や‘`--maximized`’のような初期フレームだけに影響するオプションは `initial-frame-alist`、その他のオプションは `default-frame-alist`に要素を追加します。Section “Command Line Arguments for Emacs Invocation” in *The GNU Emacs Manual* を参照してください。

29.4.3 ウィンドウフレームパラメーター

フレームがどんなパラメーターをもつかは、どのようなディスプレイのメカニズムがそれを使用するかに依存します。このセクションでは一部、またはすべての端末種類において特別な意味をもつパラメーターを説明します。これらのうち `name`、`title`、`height`、`width`、`buffer-list`、`buffer-predicate`

は端末フレームにおいて意味をもつ情報を提供し、`tty-color-mode`はテキスト端末上のフレームにたいしてのみ意味があります。

29.4.3.1 基本パラメーター

以下のフレームパラメーターはフレームに関するっとも基本的な情報を提供します。`title`と `name`はすべての端末において意味をもちます。

`display` このフレームをオープンするためのディスプレイ。これは環境変数 `DISPLAY`のような‘*host*:*dpy*.*screen*’という形式の文字列であること。ディスプレイ名についての詳細は、Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`display-type`
このパラメーターはこのフレーム内で使用できる利用可能なカラーの範囲を記述する。値は `color`、`grayscale`、`mono`のいずれか。

`title` フレームが非 `nil`の title をもつ場合には、それはフレーム上端にあるウィンドウシステムのタイトルバーに表示され、`mode-line-frame-identification`に‘`%F`’ (Section 23.4.5 [%-Constructs], page 491 を参照) を使用していればそのフレーム内のウィンドウのモードラインにも表示される。これは通常は Emacs がウィンドウシステムを使用しておらず、かつ同時に 1 つのフレームのみ表示可能なケースが該当する。Section 29.6 [Frame Titles], page 730 を参照のこと。

`name` そのフレームの名前。`title`が未指定か `nil`ならフレーム名はフレームタイトルにたいしてデフォルトの役割りを果たす。name を指定しなければ Emacs は自動的にフレーム名をセットする (Section 29.6 [Frame Titles], page 730 を参照)。

フレーム作成時に明示的にフレーム名を指定すると、そのフレームにたいして X リソースを照合する際にも、(Emacs 実行可能形式名のかわりに) その名前が使用される。

`explicit-name`
フレーム作成時にフレーム名が明示的に指定されると、このパラメーターはその名前。明示的に名付けられなかったら、このパラメーターは `nil`。

29.4.3.2 位置のパラメーター

フレームの XY 方向のオフセットを記述するパラメーターは常にピクセル単位です。子フレームではない通常のフレームでは、フレームのアウター位置 (Section 29.3 [Frame Geometry], page 701 を参照) はディスプレイの原点から相対的に指定されます子フレーム (Section 29.14 [Child Frames], page 739 を参照) では、フレームのアウター位置は親フレームのネイティブ位置から相対的に指定されます (これらのパラメーターに TTY フレームで意味のあるものはないことに注意)。

`left` フレームのディスプレイ (か親フレーム) の左エッジからフレームの左アウターエッジまでのピクセル単位での位置。以下のいずれかの方法で指定できる。

整数 正の整数はフレームの左エッジをディスプレイ (か親フレーム) の左エッジ、負の整数はフレームの右エッジをディスプレイ (か親フレーム) の右エッジに相対的に指定する。

`(+` *pos*`)` これはディスプレイ (か親フレーム) の左エッジにたいしフレームの左エッジの相対的位置を指定する。整数 *pos*は正と負の値をとり得る。負の値はスクリーン (か親フレーム) の外側、または (マルチモニターディスプレイにたいしては) プライマリーモニター以外のモニター上の位置を指定する。

(- *pos*) これはディスプレイ (か親フレーム) ノ右エッジにたいしフレームノ右エッジの相対的位置を指定する。整数 *pos*は正ト負の値をとり得る。負の値はスクリーン外側、または (マルチモニターディスプレイにたいしては) プライマリーモニター以外のモニター上の位置を指定する。

a floating-point value

0.0 から 1.0 の範囲の浮動小数点数はフレームの左位置比率 *(left position ratio)* を通じて左エッジのオフレットを指定する。これはアウターフレームの左エッジの位置にたいする、フレームのワークエリア (Section 29.2 [Multiple Terminals], page 698 を参照) または親のネイティブフレーム (Section 29.14 [Child Frames], page 739 を参照) からアウターフレームの幅を減じた値との比率。したがって左位置比率 0.0 はディスプレイ (か親フレーム) の左、比率 0.5 は中央、1.0 は右に揃える。同様に上位置比率 *(top position ratio)* はアウターフレームの上エッジの位置にたいする、フレームのワークエリア (か親のネイティブフレーム) からフレームの高さを減じた値との比率。

Emacs は子フレームのパラメーター `keep-ratio` (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照) が非 `nil`なら、親フレームがリサイズされた場合に子フレームの位置比率を未変更のままにしようと試みる。

通常はフレームが可視になるまでフレームのアウターサイズ (Section 29.3 [Frame Geometry], page 701 を参照) は利用できないので、装飾つきのフレーム作成時には浮動小数点値の一般的には使用はできない。浮動小数点値は親フレームの領域内で子フレームの適切な配置を保証したい場合により適している。

プログラム指定の位置を無視するウィンドウマネージャーがいくつかある。指定した位置が無視されない保証を望む場合には、パラメーター `user-position`にも以下の例のように非 `nil`値を指定すること。

```
(modify-frame-parameters
  nil '((user-position . t) (left . (+ -4))))
```

一般的にフレームをディスプレイの右エッジや下エッジから相対的に配置するのは良いアイデアではない。初期フレームや新たなフレームの配置は不正確になる (フレームが可視になるまでアウターフレームのサイズは完全には解らない) か、または (可視になった後にフレームが再配置されると) 追加のちらつき (flickering) をもたらすだろう。

さらにディスプレイ (やワークエリア、親フレーム) の右エッジや下エッジから相対的に指定された位置、同様に浮動小数点数のオフセットは内部的にはディスプレイ (やワークエリア、親フレーム) の左エッジや上エッジから相対的な整数のオフセットとして格納されることに注意。これらの値は `frame-parameters`のような関数によるリターン値やデスクトップ保存ルーチンのリストアにも使用される。

`top` ディスプレイ (か親フレーム) の上エッジ (下エッジ) にたいして、上エッジ (下エッジ) のスクリーン位置をピクセル単位で指定する。方向が水平ではなく垂直である点を除イテ、これは `left`と同様に機能する。

`icon-left`

スクリーン左エッジから数えた、フレームアイコン左エッジのピクセル単位のスクリーン位置。ウィンドウマネージャーがこの機能をサポートすれば、これはフレームをアイ

コン化したとき効果を発揮する。このパラメーターに値を指定する場合には `icon-top` にも値を指定しなければならず、その逆も真。

`icon-top` スクリーン上端から数えたフレームアイコン上端のピクセル単位のスクリーン位置。ウィンドウマネージャーがこの機能をサポートすれば、これはフレームをアイコン化したときに効果を発揮する。

`user-position`
フレームを作成してパラメーター `left`と `top`で位置を指定する際は、指定した位置がユーザー指定 (人間であるユーザーにより明示的に要求された位置) なのか、それとも単なるプログラム指定 (プログラムにより選択された位置) なのかを告げるためにこのパラメーターを使用する。非 `nil`値はそれがユーザー指定の位置であることを告げる。

ウィンドウマネージャーは一般的にユーザー指定位置に留意するとともに、プログラム指定位置にも幾分か留意する。しかし多くのウィンドウマネージャーはプログラム指定位置を無視して、ウィンドウをウィンドウマネージャーのデフォルトの方法で配置するかユーザーのマウスによる配置に任せる。`twm`を含むウィンドウマネージャーのいくつかは、プログラム指定位置にしたがうか無視するかをユーザーの指定に任せる。

`make-frame`を呼び出す際にパラメーター `left`や `top`の値がそのユーザーにより示される嗜好を表すなら、このパラメーターに非 `nil`値、それ以外は `nil`を指定すること。

`z-group` このんパラメーターはフレームのウィンドウシステム的なウィンドウの相対位置を、フレームのディスプレイのスタック順 (Z オーダー) で指定する。

これが `above`ならウィンドウシステムは、`above`プロパティがセットされていない他のすべてのウィンドウシステムのウィンドウの前面にフレームに対応するウィンドウを表示する。`nil`ならフレームのウィンドウは `above`プロパティがセットされたすべてのウィンドウの背後、かつ `below`プロパティがセットされたすべてのウィンドウの前面に表示される。`below`ならフレームのウィンドウは `below`プロパティがセットされていないすべてのウィンドウの背後に表示される。

特定のフレームの前面や背後にフレームを配置するためには関数 `frame-restack`を使用すること (Section 29.12 [Raising and Lowering], page 737 を参照)。

29.4.3.3 サイズのパラメーター

フレームパラメーターは通常はフレームのサイズを文字単位で指定します。グラフィカルなディスプレイ上では `default`フェイスがこれら文字単位の実際のピクセルサイズを決定します (Section 39.12.1 [Face Attributes], page 1012 を参照)。

`width` このパラメーターはフレームの幅を指定する。これは以下の方法でも指定できる:

整数 フレームのテキストエリア (Section 29.3 [Frame Geometry], page 701 を参照) の幅を文字単位で指定する正の整数。

a cons cell これが CAR にシンボル `text-pixels`をもつコンスセルなら、CDR はフレームのテキストエリアの幅をピクセル単位で指定する。

a floating-point value
0.0 から 1.0 の範囲の浮動小数点数はフレームの幅比率 *(width ratio)* を通じてフレームの幅の指定に使用できる。これはアウター幅 (Section 29.3 [Frame Geometry], page 701 を参照) にたいする、フレームのワークエリア (Section 29.2 [Multiple Terminals], page 698 を参照) または親フレームのネイティブフレーム (Section 29.14 [Child Frames], page 739

を参照) からアウターフレームの幅を減じた値との比率。したがって値 0.5 はワークエリア (か親フレーム) の半分の幅、値 1 は全幅をフレームに占有させる。同様に高さ比率 *(height ratio)* はアウターフ高さにたいする、フレームのワークエリア (か親のネイティブフレーム) からフレームの高さとの比率。

Emacs は子フレームのパラメーター `keep-ratio` (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照) が非 `nil`なら、親フレームがリサイズされた場合に子フレームの幅と高さの比率を未変更のままにしようと試みる。

通常はフレームが可視になるまでフレームのアウターサイズ (Section 29.3 [Frame Geometry], page 701 を参照) は利用できないので、装飾つきのフレーム作成時には浮動小数点値の一般的には使用はできない。浮動小数点値は、たとえば `display-buffer-in-child-frame`を通じて `display-buffer-alist` (Section 28.13.1 [Choosing Window], page 640 を参照) をカスタマイズする際に親フレームの領域内へ常に子フレームを確実にフィットさせたい場合により適している。

パラメーターの指定方法にかかわらず、このパラメーターの値を報告する `frame-parameters`のような関数は、常にフレームのデフォルトの文字幅の倍数にするために必要に応じて丸めを行いフレームのテキストエリアの幅を整数でリターンする。この値はデスクトップ保存ルーチンでも使用される。

`height` このパラメーターはフレームの高さを指定する。これは水平ではなく垂直であるという点を除き `width`のように機能する。

`user-size`
これはサイズパラメーター `height`と `width`にたいして、`user-position` (Section 29.4.3.2 [Position Parameters], page 715 を参照) が `top`と `left`が行うのと同じことを行う。

`min-width`
このパラメーターはフレームの最小ネイティブ幅 (Section 29.3 [Frame Geometry], page 701 を参照) を文字単位で指定する。フレームの初期幅やフレームを水平方向にリサイズする関数は、通常はフレームのすべてのウィンドウ、垂直スクロールバー、フリンジ、マージン、垂直ディバイダーが表示できるよう保証する。このパラメーターが非 `nil`なら、収まりきらないコンポーネントが結果としてウィンドウマネージャーにクリップされてしまうようなフレームより小さいフレームを作成できる。

`min-height`
このパラメーターはフレームの最小ネイティブ高さ (Section 29.3 [Frame Geometry], page 701 を参照) を文字単位で指定する。フレームの初期幅やフレームを水平方向にリサイズする関数は、通常はフレームのすべてのウィンドウ、水平スクロールバー、水平ディバイダー、モードライン、ヘッダーライン、エコーエリアー、インターナルメニューバー、インターナルツールバーが表示できるよう保証する。このパラメーターが非 `nil`なら、収まりきらないコンポーネントが結果としてウィンドウマネージャーにクリップされてしまうようなフレームより小さいフレームを作成できる。

`fullscreen`
このパラメータはフレームの幅、高さ、またはその両方を最大化するかどうかを指定する。値は `fullwidth`、`fullheight`、`fullboth`、または `maximized`。フレームが

fullwidth なら幅、*fullheight* なら高さ、*fullboth* まら幅と高さが可能なかぎり大きくなる。*maximized* は “fullboth” と同様だが、通常はタイトルバーとフレームのリサイズやクローズ用のボタンが維持される点が異なる。同様に最大化されたフレームでは、デスクトップ上に表示されているタスクバーやパネルが見えなくなること通常はない。一方で “fullboth(全画面)” のフレームは、通常はタイトルバーは省略されて、利用可能なスクリーンスペース全体を占有する。

この点では fullheight や fullwidth のフレームは最大化されたフレームと類似している。しかしこれらは通常は最大化フレームでは存在しないエクスターナルボーダーを表示する。したがって最大化されたフレームの高さと幅は、fullheight のフレームの高さや fullwidth のフレームの幅とあ数ピクセル異なることがある。

いくつかのウィンドウマネージャーではフレームを真に最大化やフルスクリーンで表示させるために、変数 `frame-resize-pixelwise` (Section 29.3.4 [Frame Size], page 709 を参照) をカスタマイズする必要があるかもしれない。さらにフルスクリーンや最大化の種々の状態間でスムーズな遷移をサポートしないウィンドウマネージャーもいくつかある。これの回避には変数 `x-frame-normalize-before-maximize`のカスタマイズが助けになるかもしれない。

macOS のフルスクリーンではツールバーとメニューバーの両方が非表示になるが、マウスポインターがスクリーン上端に移動すればどちらも表示される。

`fullscreen-restore`
: このパラメータは `toggle-frame-fullscreen`コマンド (Section “Frame Commands” in *The GNU Emacs Manual* を参照) 呼び出し後の “fullboth” 状態での、望ましい全画面状態を指定する。このパラメーターは Section “Frame Commands” in *The GNU Emacs Manual* を参照の切り替え時に、このコマンドによって自動的にインストールされる。しかし Emacs が “fullboth” の状態で開始された場合には、以下の例のように初期ファイル内で望ましい挙動を指定する必要がある

```
(setq default-frame-alist
    '((fullscreen . fullboth)
      (fullscreen-restore . fullheight)))
```

これは `F11`の初回タイプ後に、フレームに新たに fullheight を与えるだろう。

`fit-frame-to-buffer-margins`
: このパラメーターは `fit-frame-to-buffer` (Section 28.4 [Resizing Windows], page 617 を参照) でフレームをルートウィンドウのバッファーにフィットさせる際に、オプション `fit-frame-to-buffer-margins`の値のオーバーライドを可能にする。

`fit-frame-to-buffer-sizes`
: このパラメーターは `fit-frame-to-buffer` (Section 28.4 [Resizing Windows], page 617 を参照) でフレームをルートウィンドウのバッファーにフィットさせる際に、オプション `fit-frame-to-buffer-sizes`の値のオーバーライドを可能にする。

29.4.3.4 レイアウトのパラメーター

以下のフレームパラメーターによりフレームのさまざまなパーツを有効または無効にしたりサイズを制御できます。

`border-width`
: ピクセル単位でのフレームのアウターボーダー幅 (Section 29.3 [Frame Geometry], page 701 を参照)。

`internal-border-width`
: ピクセル単位でのフレームのインターナルボーダー幅 (Section 29.3 [Frame Geometry], page 701 を参照)。

`vertical-scroll-bars`
: フレームが垂直スクロール用のスクロールバー (Section 39.14 [Scroll Bars], page 1039 を参照) をもつべきか否か、およびスクロールバーをフレームのどちら側に置くか。可能な値は `left`、`right`、スクロールバーなしは `nil`。

`horizontal-scroll-bars`
: フレームが水平スクロール用のスクロールバーをもつべきかと、スクロールバーをフレームのどちら側に置くか (`t`と `bottom`はスクロールバーあり、`nil`はスクロールバーなしを意味する)。

`scroll-bar-width`
: 垂直スクロールバーのピクセル単位による幅。`nil`はデフォルト幅の使用を意味する。

`scroll-bar-height`
: 垂直スクロールバーのピクセル単位による高さ。`nil`はデフォルト高さの使用を意味する。

`left-fringe`
`right-fringe`
: そのフレーム内のウィンドウの左右フリンジのデフォルト幅 (Section 39.13 [Fringes], page 1034 を参照)。いずれかが 0 なら対応するフリンジを削除する効果がある。

  これら 2 つのフレームパラメーターの値を問い合わせるために `frame-parameter`を使用する際のリターン値は常に整数。`nil`値を渡して `set-frame-parameter`を使用する際には、実際のデフォルト値 8 ピクセルが課せられる。

`right-divider-width`
: フレーム上のすべてのウィンドウの右ディバイダー (Section 39.15 [Window Dividers], page 1042 を参照) 用に予約されるピクセル単位の幅 (厚さ)。値 0 は右ディバイダーを描画しないことを意味する。

`bottom-divider-width`
: フレーム上のすべてのウィンドウの下ディバイダー (Section 39.15 [Window Dividers], page 1042 を参照) 用に予約されるピクセル単位の幅 (厚さ)。値 0 は下ディバイダーを描画しないことを意味する。

`menu-bar-lines`
: メニューバー用にフレーム上端に割り当てる行数 (Section 22.17.5 [Menu Bar], page 453 を参照)。デフォルトは Menu Bar モードが有効なら 1、それ以外なら 0。Section “Menu Bars” in *The GNU Emacs Manual* を参照のこと。エクスターナルメニューバー (Section 29.3.1 [Frame Layout], page 702 を参照) では、メニューバーが複数行に折り返されても値は変更されない。この場合には `frame-geometry`がリターンする `menu-bar-size`の値で実際にメニューバーが占有する行数を導出できる (Section 29.3 [Frame Geometry], page 701 を参照)。

`tool-bar-lines`
: ツールバー用に使用する行数 (Section 22.17.6 [Tool Bar], page 455 を参照)。デフォルトは Tool Bar モードが有効なら 1、それ以外は 0。Section “Tool Bars” in *The GNU Emacs Manual* を参照のこと。ツールバーが折り返されているかどうかで値は変化するかもしれない (Section 29.3.1 [Frame Layout], page 702 を参照)。

`tool-bar-position`
: Emacs が GTK+とともにビルドされた際のツールバーの位置。値は `top`、`bottom`、`left`、`right`のいずれか (デフォルトは `top`)。

`line-spacing`
: 各テキスト行の下に残すピクセル単位の追加スペース (正の整数)。詳細は Section 39.11 [Line Height], page 1010 を参照のこと。

`no-special-glyphs`
: 非`nil`ならそのフレームで表示するすべてのバッファーにたいする切り詰めと継続のグリフ (Section 39.3 [Truncation], page 982 を参照) の表示をすべて抑制する。これは `fit-frame-to-buffer` (Section 28.4 [Resizing Windows], page 617 を参照) を通じてフレームをバッファーにフィットさせる際のグリフ排除に有用。

29.4.3.5 バッファーのパラメーター

以下はフレーム内でどのバッファーが表示されているか、表示されるべきかを扱うためのフレームパラメーターであり、すべての種類の端末上で意味があります。

`minibuffer`
: そのフレームが自身のミニバッファーをもつか否か。もつ場合には `t`、もたない場合は `nil`、`only`ならそのフレームが正にミニバッファーであることを意味する。値が (別フレーム内の) ミニバッファーウィンドウなら、そのフレームはそのミニバッファーを使用する。

  このパラメーターはフレームの作成時に効果をもつ。`nil`を指定すると Emacs は `default-minibuffer-frame`のミニバッファーウィンドウ (Section 29.9 [Minibuffers and Frames], page 732 を参照) にそのフレームをセットしようと試みる。このパラメーターは既存のフレームでは別のミニバッファーウィンドウを指定するためにのみ使用できる。`nil`から `t`、およびその逆の変更も許されていない。このパラメーターがすでにミニバッファーウィンドウを指定していれば、これを `nil`にセットしても効果はない。

  特別な値 `child-frame`は作成されるフレームが親となるようなミニバッファーのみの子フレーム (Section 29.14 [Child Frames], page 739 を参照) を作成することを意味する。Emacs は `nil`が指定されたかのように子フレームのイニバッファーウィンドウにこのパラメーターをセットするが、作成後に子フレームを選択しない。

`buffer-predicate`
: このフレームにたいするバッファー述語関数。関数 `other-buffer`はこの述語が非 `nil`なら、(選択されたフレームから) どのバッファーを考慮すべきか決定するためにこれを使用する。これは各バッファーにたいして、そのバッファーを唯一の引数としてこの述語を 1 回呼び出す。この述語が非 `nil`値をリターンしたら、そのバッファーは考慮される。

`buffer-list`
: そのフレーム内で選択されたことのあるバッファーにたいする、もっとも最近選択されたバッファーが先頭になるような順のリスト。

`unsplittable`
: 非`nil`なら、このフレームのウィンドウは決して自動的に分割されることはない。

29.4.3.6 フレームとの相互作用のためのパラメーター

以下のパラメーターは別のフレームとの間での相互作用するためのフォームを提供します。

`parent-frame`
: 非`nil`ならそのフレームが子フレーム (Section 29.14 [Child Frames], page 739 を参照) であることを意味しており、そのパラメーターは親フレームを指定する。`nil`ならそのフレームが通常のトップレベルのフレームであることを意味する。

`delete-before`
: このパラメーターが非`nil`なら、それは削除によってこのウレームも自動的にトリガーされるようなフレームを指定する。Section 29.7 [Deleting Frames], page 730 を参照のこと。

`mouse-wheel-frame`
: このパラメーターが非`nil`ならフレーム上でマウスホイールをスクロールするたびにウィンドウがスクロールされるようなフレームを指定する。Section “Mouse Commands” in *The GNU Emacs Manual* を参照のこと。

`no-other-frame`
: 非`nil`ならそのフレームは関数`next-frame`、`previous-frame` (Section 29.8 [Finding All Frames], page 731 を参照)、`other-frame`の候補として適格ではない。Section “Frame Commands” in *The GNU Emacs Manual* を参照のこと。

`auto-hide-function`
: このパラメーターが関数を指定する場合には、他のフレームが存在しないときにフレーム唯一のウィンドウを quit(Section 28.16 [Quitting Windows], page 662 を参照) した際に変数`frame-auto-hide-function`で指定された関数のかわりに呼び出される関数を指定する。

`minibuffer-exit`
: このパラメーターが非`nil`なら、Emacs はミニバッファーの exit(Chapter 20 [Minibuffers], page 342 を参照) の際は常にそのフレームを不可視にする。かわりに関数`iconify-frame`と`delete-frame`も指定できる。このパラメーターはミニバッファーの exit 時に (Emacs がウィンドウを処理する際のように) 自動的に子フレームを非表示にするために有用。

`keep-ratio`
: このパラメーターは現在のところ子フレーム (Section 29.14 [Child Frames], page 739 を参照) にたいしてのみ意味がある。非`nil`なら親フレームのリサイズ時に、Emacs はフレームのサイズ比率 (幅と高さ。Section 29.4.3.3 [Size Parameters], page 717 を参照) と位置比率 (左と右。Section 29.4.3.2 [Position Parameters], page 715 を参照のこと) を変更しないよう試みる。

  このパラメーターの値が`nil`なら親フレームのリサイズ時にフレームのサイズと位置は変更されないので、位置とサイズの比率は変更されるかもしれない。パラメーターの値が`t`なら、Emacs はフレームのサイズと位置の比率を維持しようと試みるので、親フレームから相対的なフレームのサイズと位置は変更されるかもしれない。

  コンスセルを使用することにより、さらに特化した制御が可能になる。この場合にはコンスセルの CAR が`t`か`width-only`ならフレームの幅比率、CAR が`t`か`height-only`なら高さ比率、CDR が`t`か`left-only`なら左位置比率、CDR が`t`か`top-only`なら上位置比率が維持される。

29.4.3.7 マウスドラッグのパラメーター

以下のパラメーターはマウスでのインターナルボーダーのドラッグによるフレームのリサイズにたいするサポートを提供します。これらは最上ウィンドウのヘッダーラインや最下ウィンドウのモードラインのマウスドラッグによるフレームの移動も可能にします。

これらのパラメーターはウィンドウマネージャーによる装飾がない子フレーム (Section 29.14 [Child Frames], page 739 を参照) にたいしてもっとも有用です。もし必要なら未装飾のトップレベルのフレームにたいしても使用できます。

`drag-internal-border`
: 非`nil`なら、(もしあれば) マウスでインターナルボーダーをドラッグしてフレームをリサイズできる。

`drag-with-header-line`
: 非`nil`なら、マウスで最上ウィンドウのヘッダーラインをドラッグしてフレームを移動できる。

`drag-with-mode-line`
: 非`nil`なら、マウスで最下ウィンドウのモードラインをドラッグしてフレームを移動できる。このようなフレームは自身のミニバッファーウィンドウをもつことが許されないことに注意。

`snap-width`
: マウスで移動されるフレームはディスプレイ (か親ウィンドウ) のいずれかのエッジにこのパラメーターで指定されたピクセル数までドラッグされるとディスプレイ (か親ウィンドウ) のボーダーに “snap する (とびつく)”。

`top-visible`
: このパラメーターが数値ならフレームの上エッジがディスプレイ (か親フレーム) の上エッジより上に表示されることは決してない。さらにディスプレイ (か親フレーム) の他のエッジにたいしてフレームが移動された際には、フレームの指定されたピクセル数分が可視に留まる。このパラメーターのセットは非`nil`の`drag-with-header-line`パラメーターをもつ子フレームが、ドラッグにより完全に親フレームのエリア外になることを防ぐために有用。

`bottom-visible`
: このパラメーターが数値ならフレームの下エッジがディスプレイ (か親フレーム) の下エッジより下に表示されることは決してない。さらにディスプレイ (か親フレーム) の他のエッジにたいしてフレームが移動された際には、フレームの指定されたピクセル数分が可視に留まる。このパラメーターのセットは非`nil`の`drag-with-mode-line`パラメーターをもつ子フレームが、ドラッグにより完全に親フレームのエリア外になることを防ぐために有用。

29.4.3.8 ウィンドウ管理のパラメーター

以下のフレームパラメーターはウィンドウマネージャーやウィンドウシステムとフレームとの相互作用のさまざまな面を制御します。これらはテキスト端末上では効果がありません。

`visibility`
: フレームの可視性 (visibility) の状態。可能な値は 3 つあり`nil`は不可視、`t`は可視、`icon`はアイコン化されていることを意味する。Section 29.11 [Visibility of Frames], page 736 を参照のこと。

`auto-raise`
: 非`nil`なら、Emacs はそのフレーム選択時に自動的にそれを前面に移動 (raise) する。これを許さないウィンドウマネージャーがいくつかある。

`auto-lower`
: 非`nil`なら、Emacs はそのフレームの選択解除時に自動的にそれを背面に移動 (lower) する。これを許さないウィンドウマネージャーがいくつかある。

`icon-type`
: そのフレームに使用するアイコンのタイプ。値が文字列なら使用するビットマップを含むファイル、`nil`ならアイコンなしを指定する (何を表示するかはウィンドウマネージャーが決定する)。その他の非`nil`値はデフォルトの Emacs アイコンを指定する。

`icon-name`
: このフレームにたいするアイコンで使用する名前。アイコンを表示する場合は、その際に表示される。これが`nil`ならフレームのタイトルが使用される。

`window-id`
: グラフィカルディスプレイがこのフレームにたいして使用する ID 番号。Emacs はフレーム作成時にこのパラメーターを割り当てる。このパラメーターを変更しても実際の ID 番号に効果はない。

`outer-window-id`
: そのフレームが存在する最外殻のウィンドウシステムのウィンドウの ID 番号。`window-id`と同じように、このパラメーターを変更しても実際の効果はない。

`wait-for-wm`
: 非`nil`ならジオメトリー変更を確認するために、ウィンドウマネージャーを待機するよう Xt に指示する。Fvwm2 および KDE のバージョンを含むウィンドウマネージャーのいくつかは確認に失敗して Xt がハングする。これらウィンドウマネージャーのハングを防ぐためには、これを`nil`にセットすること。

`sticky`
: 非`nil`なら仮想デスクトップを伴うシステム上のすべての仮想デスクトップ上でそのフレームが可視になる。

`inhibit-double-buffering`
: 非`nil`ならフレームはダブルバッファリングなしでスクリーンに描画される。通常はちらつきを減少させるために、利用可能なら Emacs はダブルバッファリングの使用を試みる。ディスプレイのバグや昔のようにちらつく感覚があるようならこのプロパティをセットすること。

`skip-taskbar`
: 非`nil`ならウィンドウマネージャーならフレームのディスプレイに関連するタスクバーからフレームのアイコンを削除して、*`Alt-TAB`*のコンビネーションキーを通じたフレームへの切り替えを抑制するようにウィンドウマネージャーに指示する。MS-Windows ではこのようなフレームをアイコン化するとデスクトップ下部にある、フレームにたいするウィンドウシステムのウィンドウに"ロールイン (roll in)"される。いくつかのウィンドウマネージャーはこのパラメーターにしたがわない。

`no-focus-on-map`
: 非`nil`ならフレームはマップ時に入力フォーカスの受け取りを希望しないことを意味する。いくつかのウィンドウマネージャーはこのパラメーターにしたがわない。

`no-accept-focus`
: 非 `nil`ならフレームは明示的なマウスクリックや `focus-follows-mouse` (Section 29.10 [Input Focus], page 732 を参照)、`mouse-autoselect-window` (Section 28.25 [Mouse Window Auto-selection], page 685 を参照) によりマウスがフレーム内に移動した際に入力フォーカスの受け取りを希望しないことを意味する。これはユーザーが選択されていないフレームをマウスでスクロールできないという、望ましくない副作用をもたらすかもしれない。いくつかのウィンドウマネージャーはこのパラメーターにしたがわない。

`undecorated`
: 非`nil`ならフレームのウィンドウシステムのウィンドウはタイトル、最小化ボックスや最大化ボックス、エクスターナルボーダーのような装飾なしで描画される。これは通常はマウスによるそのウィンドウシステムのウィンドウのドラッグ、リサイズ、アイコン化、最大化、削除ができないことを意味する。`nil`の場合にはウィンドウマネージャーのセッティングでディスプレイがサスペンドされていなければ、フレームのウィンドウは上述のすべての要素とともに描画される。

X では装飾をオフに切り替えるために、Emacs は Motif ウィンドウマネージャーのヒントを使用する。いくつかのウィンドウマネージャーはこれらのヒントにしたがわない。

NS ビルドはツールバーを装飾とみなすので未装飾のフレームでは表示されない。

`override-redirect`
: 非`nil`ならオーバーライド・リダイレクト *(override redirect)* のフレームであることを意味する。これはフレームが X 配下のウィンドウマネージャーで処理されないことを意味する。オーバーライド・リダイレクト・フレームはウィンドウマネージャーの装飾をもたず、Emacs の配置関数やリサイズ関数でしか配置やリサイズができない。また他のすべてのフレームの最上位に通常は描画される。このパラメーターをセットしても MS-Windows では効果がない。

`ns-appearance`
: macOS のみ利用可能であり `dark`にセットすると “vibrant dark” テーマ、それ以外ならシステムのデフォルトを使用してフレームのウィンドウシステムのウィンドウを描画する。暗色 (dark) のバックグラウンドの Emacs テーマ使用時にツールバーやスクロールバーに暗色の外観をセットするために “vibrant dark” テーマを使用できる。

`ns-transparent-titlebar`
: macOS でのみ利用可能であり非`nil`ならタイトルバーとツールバーをトランスペアレント (transparent: 透明) にセットする。これは Emacs のバックグラウンドカラーにマッチするようにタイトルバーとツールバーのバックグラウンドカラーを効果的にセットする。

29.4.3.9 カーソルのパラメーター

このフレームパラメーターはカーソルの外見を制御します。

`cursor-type`
: カーソルの表示方法。適正な値は:

  `box` 塗りつぶされた四角形 (filled box) を表示する (デフォルト)。

  `hollow` 中抜きの四角形 (hollow box) を表示する。

  `nil` カーソルを表示しない。

`bar` 文字間に垂直バー (vertical bar) を表示する。

`(bar . width)`
文字間に幅が *width*ピクセルの垂直バー (vertical bar) を表示する。

`hbar` 文字間に水平バー (horizontal bar) を表示する。

`(hbar . height)`
文字間に高さが *height*ピクセルの水平バー (horizontal bar) を表示する。

フレームパラメーター `cursor-type`は変数 `cursor-type`と `cursor-in-non-selected-windows`によりオーバーライドされるかもしれません。

`cursor-type` [User Option]
このバッファーローカル変数は選択されたウィンドウ内で表示されているそのバッファーのカーソルの外見を制御する。この値が `t`なら、それはフレームパラメーター `cursor-type`で指定されたカーソルのーを使用することを意味する。それ以外では値は上記リストのカーソルタイプのいずれかであるべきであり、これはフレームパラメーター `cursor-type`をオーバーライドする。

`cursor-in-non-selected-windows` [User Option]
このバッファーローカル変数は選択されていないウィンドウ内でのカーソルの外見を制御する。これはフレームパラメーター `cursor-type`と同じ値をサポートする。さらに `nil`は選択されていないウィンドウ内にはカーソルを表示せず、`t`は通常のカーソルタイプの標準的な変更 (塗りつぶされた四角形は中抜きの四角形、バーはより細いバーになる) の使用を意味する。

`x-stretch-cursor` [User Option]
この変数はタブや伸長された空白文字のようなエクストラワイドグリフで表示される block カーソルの幅を制御する。デフォルトではそのフォントのデルト文字だけの幅で、これはカーソル一のグリフがエクストラワイドなら幅を完全にカバーしないだろう。この変数にたいする非 `nil`値はカーソル位置のグリフの幅に応じて block カーソルを描画することを意味する。デフォルト値は `nil`。

テキストモードのフレームでは Emacs の制御外部の端末によりカーソルが描画されるので、この変数に効果はない。

`blink-cursor-alist` [User Option]
この変数はカーソルのブリンク (blink: 点滅) 方法を指定する。各要素は `(on-state . off-state)`という形式をもつ。カーソルタイプが *on-state*と等しい (`equal`を用いて比較) ときは、これに対応する *off-state*がブリンクが “off” の際のカーソルの外見を指定する。*on-state*と *off-state*はどちらもフレームパラメーター `cursor-type`に適した値であること。

それぞれのカーソルタイプのブリンク方法にたいして、そのタイプがここで *on-state*として指定されていなければ、さまざまなデフォルトが存在する。フレームパラメーター `cursor-type`で指定した際に限り、この変数内での変更は即座に効果を発揮しない。

29.4.3.10 フォントとカラーのパラメーター

以下のフレームパラメーターはフォントとカラーの使用を制御します。

`font-backend`
フレーム上で文字の描画に使用するフォントバックエンド *(font backends)* を優先順に指定するシンボルのリスト。Cairo なしでビルドされた Emacs の X での描画では現

在のところ `x` (X のコアフォントドライバー)、`xft` (Xft フォントドライバー)、`xfthb` (HarfBuzz テキストシェイピングをもつ Xft フォントドライバー) という 3 つのフォントバックエンドが潜在的に利用できる。Cairo 描画つきでビルドされた場合にも X 上のフォントバックエンドとして `x`、`ftcr` (Cairo の FreeType フォントドライバー)、`ftcrhb` (HarfBuzz テキストシェイピングをもつ Cairo 上の FreeType フォントドライバー) の 3 つが潜在的に利用できる。HarfBuzz つきで Emacs をビルドした場合には、非推奨の `ftcr`ドライバーの使用が可能であっても、デフォルトのフォントドライバーは `ftcrhb` となる。MS-Windows では現在のところ `gdi` (MS-Windows のコアフォントドライバー)、`uniscribe` (OTF フォントと TTF フォントにたいする Uniscribe エンジンによるテキストシェイピングをもつフォントドライバー)、`harfbuzz` (OTF フォントと TTF フォントにたいする HarfBuzz テキストシェイピングをもつフォントドライバー) という 3 つのフォントバックエンドが利用できる (Section "Windows Fonts" in *The GNU Emacs Manual* を参照)。同様に `harfbuzz`も推奨される。それ以外のシステムでは利用可能なフォントバックエンドは 1 つだけなので、このフレームパラメーターの変更は意味をもたない。

`background-mode`
: このパラメーターは `dark`か `light`のいずれかで、それぞれバックグラウンドを暗く (dark) するか、明るく (light) するかに対応する。

`tty-color-mode`
: このパラメーターは端末上で使用するカラーモードを指定して、そのシステムの端末機能データベース (terminal capabilities database、termcap) により与えられた端末のカラーサポートをオーバーライドする。値にはシンボルか数値を指定できる。数値なら使用するカラー数 (および間接的にはそれぞれのカラーを生成するためのコマンド) を指定する。たとえば `(tty-color-mode . 8)`は標準的なテキストカラーにたいして ANSI エスケープシーケンスの使用を指定する。値 −1 はカラーサポートをオフに切り替える。

  このパラメーターの値がシンボルなら、それは `tty-color-mode-alist`の値を通じて数値を指定するもので、そのシンボルに割り当てられた数値がかわりに使用される。

`screen-gamma`
: これが数値なら Emacs はすべてのカラーの輝度を調整するガンマ補正 (gamma correction) を行う。値はディスプレイのスクリーンのガンマであること。

  通常の PC モニターはスクリーンガンマが 2.2 なので、Emacs と X ウィンドウのカラー値は一般的にそのガンマ値のモニター上で正しく表示するよう校正されている。`screen-gamma`にたいして 2.2 を指定すると、それは補正が不必要であることを意味する。その他の値は通常のモニター上のガンマ値 2.2 で表示されるように、補正したカラーがスクリーン上に表示されることを意図された補正を要求する。

  モニターが表示するカラーが明るすぎる場合には、`screen-gamma`に 2.2 より小さい値を指定すること。これはカラーをより暗くする補正を要求する。スクリーンガンマの値 1.5 は、LCD カラーディスプレイにたいして良好な結果を与えるだろう。

`alpha`
: このパラメーターは可変透明度 (variable opacity) をサポートするグラフィカルディスプレイ上でそのフレームの透明度を指定する。これは 0 から 100 の整数であるべきで 0 は完全な透明、100 は完全な不透明を意味する。`nil`値をもつこともでき、これは Emacs にフレームの opacity をセットしないよう告げる (ウィンドウマネージャーに委ねる)。

  フレームが完全に見えなくなるのを防ぐために、変数 `frame-alpha-lower-limit`は透明度の最低限度を定義する。フレームパラメーターの値がこの変数の値より小さければ Emacs は後者を使用する。デフォルトの `frame-alpha-lower-limit`は 20。

フレームパラメーター alphaにはコンスセル (active . inactive)も指定できる。ここで activeは選択時のフレームの透明度、inactiveは未選択時の透明度。

いくつかのウィンドウシステムは子フレーム (Section 29.14 [Child Frames], page 739 を参照) にたいして alphaパラメーターをサポートしない。

以下は特定のフェイスの特定のフェイス属性と自動的に等しくなるので、ほぼ時代遅れとなったフレームパラメーターです (Section “Standard Faces” in *The Emacs Manual* を参照)。

`font` フレーム内でテキストを表示するためのフォントの名前。これはシステムで有効なフォント名か、Emacs フォントセット名 (Section 39.12.11 [Fontsets], page 1027 を参照) のいずれかであるような文字列。これは defaultフェイスの font属性と等価。

`foreground-color`
文字のイメージに使用するカラー。これは defaultフェイスの :foreground属性と等価。

`background-color`
文字のバックグラウンドに使用するカラー。これは defaultフェイスの :background属性と等価。

`mouse-color`
マウスポインターのカラー。これは mouseフェイスの :background属性と等価。

`cursor-color`
ポイントを表示するカーソルのカラー。これは cursorフェイスの :background属性と等価。

`border-color`
これはフレームのボーダーのカラー。これは borderフェイスの :background属性と等価。

`scroll-bar-foreground`
非 nilならスクロールバーのフォアグラウンドカラー。これは scroll-barフェイスの :foreground属性と等価。

`scroll-bar-background`
非 nilならスクロールバーのバックグラウンドカラー。これは scroll-barフェイスの :background属性と等価。

29.4.4 ジオメトリー

以下は X スタイルのウィンドウジオメトリー指定によるアクションのデータを調べる方法です:

`x-parse-geometry` *geom* [Function]

関数 x-parse-geometryは標準的な X ウィンドウのジオメトリー文字列を make-frameの引数の一部として使用できる alist に変換する。

この alist は *geom*内で指定されたパラメーターと、そのパラメーターに指定された値を記述する。各要素は (*parameter* . *value*)のような形式。可能な *parameter*の値は left、top、width、height。

サイズのパラメーターの値は整数でなければならない。位置のパラメーター leftと topの名前に関しては、かわりに右端または下端の位置を示す値もいくつかあるので完全に正確ではない。位置パラメーターにたいして可能な *value*は前述したような整数 (Section 29.4.3.2 [Position Parameters], page 715 を参照)、リスト (+ *pos*)、リスト (- *pos*)である。

以下は例:

```
(x-parse-geometry "35x70+0-0")
     ⇒ ((height . 70) (width . 35)
         (top - 0) (left . 0))
```

29.5 端末のパラメーター

端末はそれぞれ関連するパラメーターのリストをもっています。これら端末パラメーター *(terminal parameters)* は主に端末ローカル変数を格納するための便利な手段ですが、いくつかの端末パラメーターは特別な意味をもっています。

このセクションでは端末のパラメーター値の読み取りや変更を行う関数を説明します。これらはすべて引数として端末かフレームいずれかを受け入れます。フレームならそれはそのフレームの端末の使用を意味します。引数 `nil`は選択されたフレームの端末という意味です。

`terminal-parameters` **&optional** *terminal* [Function]
この関数は *terminal*n のすべてのパラメーターとその値をリストする alist をリターンする。

`terminal-parameter` *terminal parameter* [Function]
この関数は *terminal*のパラメーター *parameter* (シンボル) の値をリターンする。*terminal* が *parameter*にたいするセッティングをもたなければ、この関数は `nil`をリターンする。

`set-terminal-parameter` *terminal parameter value* [Function]
この関数は *terminal*のパラメーター *parameter*に指定された *value*をセットしてパラメーターの以前の値をリターンする。

以下は特別な意味をもついくつかの端末パラメーターのリストです:

`background-mode`
端末のバックグラウンドカラーの区分で `light`か `dark`のいずれか。

`normal-erase-is-backspace`
値は 1 か 0 で、これはその端末上で `normal-erase-is-backspace-mode`がオンまたはオフのいずれに切り替えられたかに依存する。Section “DEL Does Not Delete” in *The Emacs Manual* を参照のこと。

`terminal-initted`
端末の初期化後に端末固有の初期化関数にセットされる。

`tty-mode-set-strings`
与えられた際には、読み取り用に tty を設定時に Emacs が出力するエスケープシーケンスを含む文字列のリスト。Emacs は端末の設定時のみこれらの文字列を発行する。すでにアクティブな端末上 (たとえば `tty-setup-hook`内) でモードを有効にしたければ、`tty-mode-set-strings`へのシーケンスの追加に加えて、`send-string-to-terminal`を使用して明示的に必要なエスケープシーケンスを出力すること。

`tty-mode-reset-strings`
与えられた際には、`tty-mode-set-strings`内の文字列の効果をアンドゥする文字列のリスト。Emacs は exit、端末の削除、Emacs 自身のサスペンドの際にそれらの文字列を発行する。

29.6 フレームのタイトル

フレームにはそれぞれ`name`というパラメーターがあります。これはウィンドウシステムが通常フレーム上端に表示するフレームタイトルにたいするデフォルトとしての役割をもちます。フレームプロパティ`name`をセットすることにより明示的に名前を指定できます。

通常は名前を明示的に指定せずに、Emacs が変数`frame-title-format`に格納されたテンプレートにもとづいて自動的にフレーム名を計算します。Emacs はフレームが再表示されるたびに名前を再計算します。

`frame-title-format` [Variable]
この変数はフレーム名が明示的に指定されないときにフレーム名を計算する方法を指定する。この変数の値は実際には`mode-line-format`のようなモードライン構文 (mode line construct) だが '`%c`'、'`%C`'、'`%l`'の構文は無視される。Section 23.4.2 [Mode Line Data], page 486 を参照のこと。

`icon-title-format` [Variable]
この変数はフレームタイトルを明示的に指定しないときの、アイコン化されたフレームの名前の計算方法を指定する。このタイトルはアイコン自体に表示される。

`multiple-frames` [Variable]
この変数は Emacs により自動的にセットされる。フレームが 2 つ以上 (ミニバッファーのみのフレームと不可視のフレームは勘定に入らない) のとき、値は`t`となる。`frame-title-format`のデフォルト値はフレームが複数存在する場合のみ、フレーム名にバッファー名を入れるために`multiple-frames`を使用する。

この変数の値は`frame-title-format`と`icon-title-format`の処理中を除き正確である保証はない。

29.7 フレームの削除

生きたフレーム *(live frame)* とは削除されていないフレームのことです。フレームが削除される際には、たとえそれへの参照元がなくなるまで Lisp オブジェクトとして存在し続けるとしても端末ディスプレイからは削除されます。

`delete-frame` **&optional** *frame force* [Command]
この関数はフレーム *frame*を削除する。引数 *frame*は生きたフレーム (以下参照) を指定しなければならず、デフォルトは選択されたフレーム。

この関数はまず *frame*のすべての子フレーム (Section 29.14 [Child Frames], page 739 を参照) とフレームパラメーター`delete-before` (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照) が *frame*を指定するすべてのフレームを削除する。祖先として *frame*をもつフレームが他に存在しないことを保証するために、このような削除はすべて再帰的に行われる。その後に *frame*がツールチップを指定していなければ、実際にフレームを kill する前にフック`delete-frame-functions`を実行する (フックの各関数は単一の引数として *frame*を受け取る)。`delete-frame`は実際にフレームを kill してフレームリストからフレームを削除した後に`after-delete-frame-functions`を実行する。

フレームのミニバッファーが別のフレームの代替えミニバッファー (Section 29.9 [Minibuffers and Frames], page 732 を参照) の役割をもつかぎりフレームを削除できないことに注意。他のフレームすべてが不可視なら通常はフレームは削除できないが、*force*が非`nil`なら削除が可能になる。

`frame-live-p` *frame* [Function]
この関数はフレーム *frame*が削除されていなければ非`nil`をリターンする。リターンされ得る非`nil`の値は`framep`と同様。Chapter 29 [Frames], page 696 を参照のこと。

いくつかのウィンドウマネージャーはウィンドウを削除するコマンドを提供します。これらはそのウィンドウを操作するプログラムに特別なメッセージを送ることにより機能します。Emacs がそれらメッセージのいずれかを受け取ったときは`delete-frame`イベントを生成します。このイベントの通常の定義は関数`delete-frame`を呼び出すコマンドです。Section 21.7.10 [Misc Events], page 395 を参照してください。

`delete-other-frames` **&optional** *frame* [Command]
このコマンドは *frame*の端末上から *frame*以外のすべてのフレームを削除する。*frame*が別のフレームのミニバッファーを使用している場合には、そのミニバッファーフレームは処理せずに残る。引数 *frame*は生きたフレームを指定しなければならず、デフォルトは選択されたフレーム。このコマンドは内部的には削除するすべてのフレームにたいして、*force*に`nil`を指定して`delete-frame`を呼び出すことにより機能する。

この関数は *frame*の子フレームは削除しない (Section 29.14 [Child Frames], page 739 を削除)。*frame*が子フレームなら *frame*の兄弟だけを削除する。

29.8 すべてのフレームを探す

`frame-list` [Function]
この関数はすべての生きたフレーム (削除されていないフレーム) のリストをリターンする。これはバッファーにたいする`buffer-list`に類似しており、すべての端末上のフレームが含まれる。リターンされるリストは新たに作成されたものであり、このリストを変更しても Emacs 内部への影響はない。

`visible-frame-list` [Function]
この関数はカレントで可視なフレームだけのリストをリターンする。See Section 29.11 [Visibility of Frames], page 736 を参照のこと。テキスト端末上のフレームは、実際に表示されるのが選択されたフレームだけだとしても常に可視であるとみなされる。

`frame-list-z-order` **&optional** *display* [Function]
この関数は Z オーダー (重なり) の順で Emacs のフレームのリストをリターンする (Section 29.12 [Raising and Lowering], page 737 を参照)。オプション引数 *display*は調査するディスプレイを指定する。*display*はフレームかディスプレイ名 (文字列) であること。省略か`nil`なら選択されたフレームのディスプレイを意味する。*display*に Emacs フレームが含まれていなければ`nil`をリターンする。

フレームは最前面 (最初) から最背面 (最後) の順にリストされる。特別なケースとして *display*が非`nil`で生きたフレームを指定する場合には、そのフレームの子フレームを Z オーダー (重なり順) でリターンする。

この関数はテキスト端末では意味がない。

`next-frame` **&optional** *frame minibuf* [Function]
この関数により特定の端末上のすべてのフレームを任意の開始位置から簡単に巡回できる。これは *frame*の端末上のすべての生きたフレームのリストから *frame*の後のフレームをリターンする。引数 *frame*は生きたフレームを指定しなければならず、デフォルトは選択されたフレー

ム。`no-other-frame`パラメーター (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照) が非`nil`であるようなフレームをリターンすることは決してない。

2 つ目の引数 *minibuf* はどのフレームを考慮するかを示す:

`nil`	ミニバッファーのみのフレームを除外。
`visible`	すべての可視フレームを考慮する。
0	すべての可視フレームとアイコン化されたフレームを考慮する。
ウィンドウ	特定のウィンドウをミニバッファーとして使用するフレームだけを考慮する。
その他	すべてのフレームを考慮する。

`previous-frame` **&optional** *frame minibuf* [Function]

`next-frame`と同様だがすべてのフレームを逆方向に巡回する。

Section 28.10 [Cyclic Window Ordering], page 633 の`next-window`と`previous-window`も参照してください。

29.9 ミニバッファーとフレーム

それぞれのフレームは通常は下端に自身のミニバッファーウィンドウをもち、フレームが選択された際は常にそれを使用します。このウィンドウは関数`minibuffer-window`で取得できます (Section 20.11 [Minibuffer Windows], page 371 を参照)。

しかしミニバッファーをもたないフレームも作成できます。そのようなフレームは、別のフレームのミニバッファーを使用しなければなりません。この別フレームはそのフレームにたいする代替えミニバッファーフレーム *(surrogate minibuffer frame)* としての役目を果たし、そのフレームが生きているかぎり`delete-frame`で削除することはできなくなります (Section 29.7 [Deleting Frames], page 730 を参照)。

フレームパラメーター`minibuffer`により、フレーム作成時に (別フレーム上にある) 使用するミニバッファーを明示的に指定できます (Section 29.4.3.5 [Buffer Parameters], page 721 を参照)。これを行わない場合には変数`default-minibuffer-frame`の値であるようなフレーム内でミニバッファーを探します。この値はミニバッファーをもつフレームである必要があります。

ミニバッファーのみのフレームを使用する場合には、ミニバッファーにエンター時にそのフレームを前面に移動 (raise) したいと思うかもしれません。その場合には変数`minibuffer-auto-raise`に`t`をセットします。Section 29.12 [Raising and Lowering], page 737 を参照してください。

`default-minibuffer-frame` [Variable]

この変数はデフォルトでミニバッファーウィンドウとして使用するフレームを指定する。これは既存のフレームには影響しない。これはカレント端末にたいして常にローカルであり、バッファーローカルにはできない。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

29.10 入力のフォーカス

どんなときでも Emacs 内のただ 1 つのフレームが選択されたフレーム *(selected frame)* です。選択されたウィンドウは常に選択されたフレーム上にあります。

Emacs がフレームを複数端末 (Section 29.2 [Multiple Terminals], page 698 を参照) 上に表示する際には、各端末は自身の選択されたフレームをもちます。しかしそれらのうち 1 つだけが、いわゆる選択されたフレームであり、それはもっとも最近に入力があった端末に属すフレームです。つま

り特定の端末からのコマンドを Emacs が実行する際には、その端末上の 1 つが選択されたフレームです。Emacs が実行するコマンドは常に 1 つだけなので、選択されたフレームは常に 1 つだけだと考える必要があります。このフレームこそが、このマニュアルで選択されたフレームと呼ぶフレームです。選択されたフレームを表示するディスプレイは、選択されたフレームのディスプレイ *(selected frame's display)* です。

`selected-frame` [Function]
この関数は選択されたフレームをリターンする。

いくつかのウィンドウシステムおよびウィンドウマネージャーは、マウスがあるウィンドウオブジェクトにキーボード入力をダイレクトします。それ以外は、さまざまなウィンドウオブジェクトにフォーカスをシフト *(shift the focus)* するために、明示的なクリックやコマンドを要求します。どちらの方法でも Emacs はフォーカスをもつフレーム (複数形の frames) を自動的に追跡します。Lisp 関数から別フレームに明示的に切り替えるためには、`select-frame-set-input-focus`を呼び出します。

前のパラグラフ中の "複数形の frames" は意図したものです。Emacs 自身は選択されたフレームを 1 つしかもたないのにたいして、Emacs は多くの異なる端末上にフレームをもつことができ (ウィンドウシステムへの接続は端末とみなされることを思い出してほしい)、各端末は入力フォーカスをもつフレームにたいして独自のアイデアをもっています。あるフレームに入力フォーカスをセットする際にはフレームの端末にフォーカスをセットしますが、他の端末上のフレームのフォーカスは残り続けます。

関数 `select-frame`を呼び出すことにより、Lisp プログラムが一時的にフレームを切り替えることができます。これはそのウィンドウシステムのフォーカス概念を変更はしません。変更ではなく何らかの方法により制御が再確認 (reasserted) されるまで、ウィンドウマネージャーの制御から抜け出す (escape) のです。

テキスト端末使用時はその端末上で一度に表示できるフレームは 1 つだけなので、`select-frame` 呼び出し後に次回の再表示で新たに選択されたフレームが実際に表示されます。このフレームは次の `select-frame`呼び出しまで選択されたままです。テキスト端末上の各フレームはバッファー名の前に表示される番号をもちます (Section 23.4.4 [Mode Line Variables], page 488 を参照)。

`select-frame-set-input-focus` *frame* **&optional** *norecord* [Function]
この関数は *frame*を選択して、(他のフレームのせいで不明瞭な場合には) それを前面に移動 (raise) してウィンドウシステムのフォーカス授与を試みる。テキスト端末上では、次回再表示時に端末スクリーン全体に新たにフレームが表示される。オプション引数 *norecord*は `select-frame`(下記参照) のときと同じ意味をもつ。この関数のリターン値に意味はない。

以下で説明する関数は理想的には他のフレームを前面にレイズすることなくフレームにフォーカスするべきです。残念ながらウィンドウシステムやウィンドウマネージャーの多くはこの要求を拒絶するかもしれません。

`x-focus-frame` *frame* **&optional** *noactivate* [Function]
この関数は *frame*のレイズを要さずに *frame*に X サーバーのフォーカスを与える。*frame*が `nil`なら選択されたフレームを意味する。X の配下ではオプション引数 *noactivate*が非 `nil`なら、*frame*のウィンドウシステムのウィンドウが "アクティブ" なウィンドウになることを防ぐことを意味する。これは *frame*が他のフレームの前面にならないようさらに強く主張する。

MS-Windows では *noactivate*引数に効果はない。しかし *frame*が子フレーム (Section 29.14 [Child Frames], page 739 を参照) なら、この関数は通常は他の子レームの前面にレイズすることなく *frame*にフォーカスする。

この関数はウィンドウシステムのサポートがなければ何もしない。

`select-frame` *frame* **&optional** *norecord* [Command]

この関数はフレーム *frame*を選択して、X サーバーのフォーカスがあればそれを一時的に無視する。*frame*にたいする選択は次回ユーザーが別フレームに何かを行うか、この関数の次回呼び出しまで継続する (ウィンドウシステムを使用する場合には以前に選択されていたフレームに依然としてウィンドウシステムの入力フォーカスがあるかもしれないので、コマンドループからリターン後にそのフレームが選択されたフレームとしてリストアされるかもしれない)。

指定された *frame*は選択されたフレームとなり、その端末が選択された端末になる。この関数はその後に *frame*内で選択されていたウィンドウを第 1 引数、*norecord*を第 2 引数にして `select-window`をサブルーチンとして呼び出す (したがって *norecord*が非 `nil`なら、もっとも最近に選択されたウィンドウとバッファーリストの変更を避ける)。Section 28.9 [Selecting Windows], page 631 を参照のこと。

この関数は *frame*、*frame*が削除されていれば `nil`をリターンする。

一般的には実行後に端末を戻すよう切り替えることなく、別の端末に切り替えるのが可能な手段として `select-frame`を決して使用しないこと。

Emacs は選択されたフレームをサーバーとしてアレンジしてウィンドウシステムに要求することによりウィンドウシステムと協調します。Emacs のいずれかのフレームが選択されたことをあるウィンドウシステムが通知すると、Emacs は内部的に *focus-in* イベントを生成します。Emacs フレームをフォーカス変更イベントの通知をサポートする `xterm`のようなテキスト端末エミュレーター上で表示している際には、テキストモードのフレームでも focus-in と focus-out のイベントを利用できます。フォーカスイベントは通常は `handle-focus-in`で処理されます。

`handle-focus-in` *event* [Command]

この関数は明示的なフォーカス通知をサポートするウィンドウシステムと端末からの focus-in イベントを処理する。これは `frame-focus-state`が問い合わせて `after-focus-change-function`を呼び出すフレームごとのフォーカスフラグを更新する。加えて Emacs が認識する選択されたフレームを、いくつかの端末でもっとも最近フォーカスされたフレームに切り替えるために `switch-frame`イベントを生成する。Emacs の選択されたフレームをもっとも最近フォーカスされたフレームに切り替えることは、それぞれの端末にある他のフレームがフォーカスをもち続けることを意味しないことに注意することが大事。自分でこの関数を呼び出してはならない。かわりにロジックを `after-focus-change-function`につけ加えること。

`handle-switch-frame` *frame* [Command]

この関数はフォーカス通知や最後のイベントとは異なるフレームから到着した入力イベントを含むさまざまな状況において Emacs が自身のために生成する switch-frame イベントを処理する。自分でこの関数を呼び出してはならない。

`redirect-frame-focus` *frame* **&optional** *focus-frame* [Function]

この関数は *frame*から *focus-frame*にフォーカスをリダイレクトする。これは *frame*にかわって *focus-frame*が以降のキーストロークとイベントを受け取るであろうことを意味する。そのようなイベント後には `last-event-frame`の値は *focus-frame*になるだろう。また *frame*を指定した switch-frame イベントも、かわりに *focus-frame*を選択するだろう。

*focus-frame*が省略または `nil`なら、*frame*にたいするすべての既存のリダイレクションがキャンセルされるので、*frame*が自身のイベントを再度受け取ることになる。

フォーカスリダイレクトの用途の 1 つは、ミニバッファーをもたないフレームにたいしてである。これらのフレームは別フレーム上のミニバッファーを使用する。別フレーム上のミニバッ

ファーをアクティブにすることは、そのフレームにフォーカスをリダイレクトすることである。これはたとえマウスがミニバッファーをアクティブにしたフレーム内に留まっていても、ミニバッファーが属すフレームにフォーカスを置く。

フレーム選択はフォーカスリダイレクションの変更も可能にする。`foo`が選択されているときにフレーム `bar`を選択することにより、`foo`を指すすべてのリダイレクションはかわりに `bar`を指す。これはユーザーが `select-window`を使用してあるフレームから別のフレームに切り替えた際に、フォーカスのリダイレクトが正しく機能することを可能にする。

これはフォーカスが自身にリダイレクトされたフレームが、フォーカスがリダイレクトされていないフレームとは異なる扱いを受けることを意味する。前者にたいして `select-frame`は影響するが、後者には影響がない。

このリダイレクションは、それを変更するために `redirect-frame-focus`が呼び出されるまで継続する。

`frame-focus-state` *frame* [Function]

この関数は *frame*の既知の最後のフォーカス状態を取得する。

フレームにフォーカスがないと解っていれば `nil`、フォーカスがあると解っていれば `t`、フレームのフォーカス状態を Emacs が知らなければ `unknown`をリターンする (この最後の状態は明示的なフォーカス通知をサポートしない端末で実行中の TTY フレームで見ることがあるかもしれない)。

`after-focus-change-function` [Variable]

この関数はフォーカス変更の通知の受信に使用可能な拡張ポイントである。

このフックの関数はフォーカスのあるフレームのセットが変更されたかもしれないことを Emacs が認識した際に引数なしで呼び出される。フレームのフォーカス変更時にコードが何かを行いたければ `add-function`を使用して関数をフックに追加して、追加した関数内で各フレームの最後の既知のフォーカス状態を取得するために `frame-focus-state`を呼び出して、フォーカスされたフレームのセットを再スキャンすること。フォーカスイベントは非同期に配信され、外部システムによるフレームの入力フォーカスは Emacs の選択されたフレームの概念には対応していないかもしれない。フォーカスイベントの配信の差異、複数の Emacs 端末の存在、それ以外の要因のために複数フレームが同時に入力フォーカスをもつように見えるかもしれないので、コードはこの状況に直面しても堅牢であること。

ウィンドウシステムによってはフォーカスイベントが繰り返し配信されて、期待した値にセットする前に異なるフォーカス状態が配信されるかもしれない。フォーカス通知にもとづくコードはおそらく再表示まで処理を遅延することにより、フォーカス変更から生じるユーザーに可視の更新を “デバウンス (debounce)” する必要がある。

この関数は `read-event`の内部を含む任意のコンテキストで呼び出されるかもしれないので、プロセスフィルター記述時と同様の注意を払うこと。

`focus-follows-mouse` [User Option]

このオプションはフレーム内にマウスポインターを移動した際にウィンドウマネージャーがフォーカスを転送するかどうかと、その方法について Emacs に知らせる。意味がある値は以下の 3 つ:

`nil` デフォルト値 `nil`はウィンドウマネージャーが “click-to-focus(フレームがフォーカスを取得するためにフレーム内部でマウスをクリックする必要がある)” のポリシーにしたがう際に使用すること。

`t`
: 値 `t`はマウスポインターの位置にしたがってウィンドウマネージャーが自動的にフォーカスするが、フォーカスを得たフレームは自動的にレイズされず別のウィンドウシステムのウィンドウに隠されたままかもしれないときに使用すること。

`auto-raise`
: 値 `auto-raise`はマウスポインターの位置にしたがってウィンドウマネージャーが自動的にフォーカスして、フォーカスを得たフレームは自動的にレイズされる際に使用すること。

このオプションが非 `nil`なら、Emacs は `select-frame-set-input-focus`が選択したフレームにマウスポインターを移動する。この関数は `other-frame`や `pop-to-buffer`などいくつかのコマンドで使用される。

ウィンドウマネージャーは “通常” のフレームのレイズには通常は配慮するので、値 `t`と `auto-raise`を区別する必要はない。これは `mouse-autoselect-window` (Section 28.25 [Mouse Window Auto-selection], page 685 を参照) を通じて子フレームをレイズするために有用。

このオオプションは “sloppy” なフォーカス (別のウィンドウマネージャーウィンドウにマウスポインターが移動しないかぎり以前にフォーカスのあったフレームがフォーカスを保持する) と “strict” なフォーカス (マウスポインターが去るとフレームは即座にフォーカスを失う) を区別しないことに注意。ウィンドウマネージャーがフォーカスや自動レイズの遅延をサポートしていなくても、新たなフレームがフォーカスを取得 (あるいは自動レイズ) するまでの時間を明示的に指定できる。

変数 `mouse-autoselect-window` (Section 28.25 [Mouse Window Auto-selection], page 685 を参照) をカスタマイズすることにより、個別の Emacs ウィンドウにたいして “focus follows mouse” ポリシーを提供できる。

29.11 フレームの可視性

グラフィカルなディスプレイ上のフレームは可視 *(visible)*、不可視 *(invisible)*、またはアイコン化 *(iconified)* されているかもしれません。可視ならそのコンテンツは通常の方法により表示されます。アイコン化されている場合にはそのコンテンツは表示されませんが、ビュー内にフレームを戻すための小さいアイコンがどこかにあります (いくつかのウィンドウマネージャーはこの状態をアイコン化ではなく最小化と呼ぶが Emacs の見地ではこれらは同等である)。フレームが不可視ならまったく表示されません。

可視性の概念はマップ (または非マップ) されたフレームと強い関連性があります。フレーム (より正確にはウィンドウシステムのウィンドウ) は最初に表示されるときにマップ済み *(mapped)* となり、状態が `iconified`や `invisible`から `visible`に変化します。それとは逆に状態が `visible`から `iconified`や `invisible`に変化したときは、フレームは常に非マップ済み *(unmapped)* になります。

テキスト端末では実際に表示されるのは常に選択されたフレームだけなので可視性に意味はありません。

`frame-visible-p` *frame* [Function]
: この関数はフレーム *frame*の可視性の状態をリターンする。値は *frame*が可視なら `t`、不可視なら `nil`、アイコン化されていれば `icon`。

テキスト端末上ではたとえ 1 つのフレームだけが表示されているとしても、この関数の目的にたいしてはすべてのフレームが可視とみなされる。Section 29.12 [Raising and Lowering], page 737 を参照のこと。

`iconify-frame` **&optional** *frame* [Command]
この関数はフレーム *frame*をアイコン化する。*frame*を省略すると選択されたフレームをアイコン化する。これにより通常は *frame*のすべての子フレーム (と子孫) は不可視になる (Section 29.14 [Child Frames], page 739 を参照)。

`make-frame-visible` **&optional** *frame* [Command]
この関数はフレーム *frame*を可視にする。*frame*を省略すると選択されたフレームを可視にする。これはフレームを前面に移動しないが、望むなら `raise-frame`でこれを行うことができる (Section 29.12 [Raising and Lowering], page 737 を参照)。

通常はフレームを可視とすることにより、すべての子フレーム (と子孫) も同様に可視になる (Section 29.14 [Child Frames], page 739 を参照)。

`make-frame-invisible` **&optional** *frame force* [Command]
この関数はフレーム *frame*を不可視にする。*frame*を省略すると選択されたフレームを不可視にする。これにより通常は *frame*のすべての子フレーム (と子孫) も不可視になる (Section 29.14 [Child Frames], page 739 を参照)。

この関数は *force*が `nil`の場合には、他のすべてのフレームが不可視なら *frame*を不可視にすることを拒絶する。

フレームの可視性の状態はフレームパラメーターとしても利用可能である。つまりフレームパラメーターとして読み取りと変更ができる。Section 29.4.3.8 [Management Parameters], page 723 を参照のこと。ウィンドウマネージャーによりユーザーがフレームのアイコン化や非アイコン化を行うこともできる。これは Emacs が何らかの制御を及ぼすのが可能なレベルより下のレベルにおいて発生するが、Emacs はそのような変化を追跡するために使用するイベントを提供する。Section 21.7.10 [Misc Events], page 395 を参照のこと。

`x-double-buffered-p` **&optional** *frame* [Function]
この関数は *frame*がカレントでダブルバッファリングで描画されていれば非 `nil`をリターンする。*frame*のデフォルトは選択されたフレーム。

29.12 フレームの raise、lower、re-stack

ほとんどのウィンドウシステムではデスクトップというメタファー (metaphor: 比喩的概念) が使用されています。このメタファーの一部はシステムレベルのウィンドウ (Emacs ではフレーム) がスクリーン表面に向かって、概念的 3 次元の垂直方向にスタッキングされる (重ねて表示される) というアイデアにもとづいています。スタッキングによる順序は合計であり、通常はスタッキングの順序 (または Z オーダー) として参照されます。2 つのウィンドウがオーバーラップする領域では、このオーダーにおいて高位のウィンドウが、下位のウィンドウ (の一部) をカバーします。

関数 `raise-frame`と `lower-frame`を使用してフレームの Z オーダーの一番上に *raise* したり一番下に *lower* することができます。関数 `frame-restack`を使用すれば別のフレームの上や下にフレームを直接 *restack* できます。

以下で説明するすべての関数はフレーム (および他のウィンドウシステムのウィンドウすべて) の対応する Z グループを尊重することに注意してください (Section 29.4.3.2 [Position Parameters], page 715 を参照)。たとえば通常はフレームをデスクトップウィンドウより lower(下位) にすることや、`z-group`パラメータ 0 が非 `nil`のフレームをウィンドウシステムのタスクバーやツールチップウィンドウより raise(上位) にすることはできません。

`raise-frame` **&optional** *frame* [Command]
この関数はを *frame*の z-group と同じか lower(下位) にある他のすべてのフレームの上位にフレーム *frame* (デフォルトは選択されたフレーム) を raise する。*frame*が不可視やアイコン化されていたら可視になる。*frame*が子フレーム (Section 29.14 [Child Frames], page 739 を参照) なら、親フレームの他のすべての子フレームの上位に *frame*を raise する。

`lower-frame` **&optional** *frame* [Command]
この関数はを *frame*の z-group と同じか上位にある他のすべてのフレームの下位へフレーム *frame* (デフォルトは選択されたフレーム) を lower にする。*frame*が子フレーム (Section 29.14 [Child Frames], page 739 を参照) なら、親フレームの他のすべての子フレームの下位に *frame*を lower する。

`frame-restack` *frame1 frame2* **&optional** *above* [Function]
この関数は *frame2*の下に *frame1*を再スタックする (restack: 再び重ねる)。いずれのフレームも可視で表示エリアがオーバーラップしていたら、*frame2*が *frame1*を (部分的に) 隠すことを暗に示している。オプションの 3 つ目の引数 *above*が非 `nil`なら、この関数は *frame2*の上に *frame1*を再スタックする。これはいずれのフレームも可視で表示エリアがオーバーラップしていたら、*frame1*が *frame2*を (部分的に) 隠すことを意味している。

この関数は最初のステップではディスプレイから *frame1*のウィンドウシステムのウィンドウを削除して、2 つ目のステップで (*above*が真なら)*frame2*のウィンドウの下に *frame1*のウィンドウを再挿入を行う、2 つのステップによりアトミックなアクションを処理すると技術的に考えることができる。したがって *frame1*を除く他のすべてのフレームに相対的な *frame2*のディスプレイ内での Z オーダー (スタッキングオーダー) は変更されない。

いくつかのウィンドウマネージャーはウィンドウの再スタックを拒絶する。

再スタックの効果は関連するフレームがアイコン化されたり不可視になったりしないかぎり継続することに注意してください。フレームを他のフレームの上位 (や下位) に永続的に表示されるフレームグループに追加するためにフレームパラメーター `z-group` (Section 29.4.3.2 [Position Parameters], page 715 を参照) を使用できます。これらのグループのいずれかにフレームが所属するかぎり、再スタックはそのグループ内での相対的なスタッキング位置にのみ効果があります。異なる Z グループに所属するフレームにたいする再スタックの効果は未定義です。関数 `frame-list-z-order`でカレントのフレームスタッキングオーダーをリストできます (Section 29.8 [Finding All Frames], page 731 を参照)。

`minibuffer-auto-raise` [User Option]
これが非 `nil`ならミニバッファーをアクティブにすることにより、ミニバッファーウィンドウのあるフレームが前面に移動される。

ウィンドウシステム上ではフレームパラメーターを使用して、(フレーム選択時に)auto-raising、(フレーム選択解除時に)auto-lowering を有効にできます。Section 29.4.3.8 [Management Parameters], page 723 を参照してください。

フレームを前面や背面に移動するという概念は、テキスト端末のフレームにも適用できます。各テキスト端末上では一度に表示されるのは常に最前面のフレームだけです。

`tty-top-frame` **&optional** *terminal* [Function]
この関数は *terminal*上の最前面のフレームをリターンする。*terminal*は端末オブジェクト、フレーム (そのフレームの端末を意味する)、または `nil` (選択されたフレームの端末を意味する) であること。これがテキスト端末を参照しなければリターン値は `nil`。

29.13 フレーム構成

フレーム構成 *(frame configuration)* はフレームのカレント配置、すべてのプロパティ、および各ウィンドウのウィンドウ構成 (Section 28.26 [Window Configurations], page 686 を参照) を記録します。

`current-frame-configuration` [Function]
この関数はフレームのカレント配置とそのコンテンツを記述するフレーム構成のリストをリターンする。

`set-frame-configuration` *configuration* **&optional** *nodelete* [Function]
この関数はフレームの状態を *configuration*の記述にリストアする。ただしこの関数は削除されたフレームはリストアしない。

この関数は通常は *configuration*内にリストされない既存フレームすべてを削除する。しかし *nodelete*が非 `nil`なら、それらのフレームはかわりにアイコン化される。

29.14 子フレーム

子フレームはウィンドウ (Chapter 28 [Windows], page 608 を参照) と “通常” のフレームとの中間にあるオブジェクトです。ウィンドウのように所属するフレームにアタッチされますが、ウィンドウとは異なり互いにオーバーラップすることができます。子フレームの 1 つのサイズや位置を変更しても兄弟となる他の子フレームのサイズや位置は変化しません。

仕様により子フレームの作成や変更を行う操作は特別な関数やカスタマイズ可能な変数ではなくフレームパラメーター (Section 29.4 [Frame Parameters], page 712 を参照) の助けを借りて実装されています。子フレームはグラフィカル端末でのみ意味があることに注意してください。

子フレームを新たに作成したり通常のフレームを子フレームに変換するためには、そのフレームの `parent-frame`パラメーター (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照) にすでに存在するフレームをセットします。このパラメーターで指定されたフレームは、パラメーターが変更やリセットされるまでフレームの親フレームになります。これにより技術的には子フレームのウィンドウシステムのウィンドウは、親フレームのウィンドウシステムのウィンドウの子ウィンドウになります。

`parent-frame`パラメーターはいつでも変更できます。これを他のフレームにセットすれば子フレームが *reparent(*親を変更*)* されます。別の子フレームにセットすればフレームをネストされた *(nested)* 子フレームにします。にセットすればフレームの状態をトップレベルのフレーム (ウィンドウシステムのウィンドウがディスプレイーのルートウィンドウの子であるようなフレーム) にリストアします。

子フレームは任意にネスト (入れ子) させることができるので、フレームは子フレームと親フレームの両方になることができます。また子フレームと親フレームの相対的な役割はいつでも逆転させることができます (たとえ子フレームを親フレームより十分小さいサイズに保つことが通常はよいアイデアであるとしても)。フレームをそのフレームの祖先にしようと試みるとエラーがシグナルされます。

ほとんどのウィンドウシステムは親フレームのネイティブエッジ (Section 29.3 [Frame Geometry], page 701 を参照) で子フレームをクリップします (これらのエッジの外側は通常は不可視になる)。子フレームのパラメーター `left`と `top`は親のネイティブフレームの左上隅から相対的な位置を指定します。親フレームがリサイズされたとき、この位置は概念的には変更されません。

NS ビルドは親フレームのエッジで子フレームをクリップしないので、子フレーム自身が可視であっても親フレームを隠さないように配置することができます。

通常は親フレームを移動することにより、すべての子フレームとその子孫も相対的な位置が変化しないように一緒に移動されます。フック `move-frame-functions` (Section 29.3.3 [Frame Position],

page 708 を参照) は子フレームの親フレームにたいする相対的な位置が変化したときだけ実行されます。

親フレームがリサイズされた際には、子フレームは概念的には以前のサイズと親フレームの左上隅からの相対的な位置を保ちます。これは親フレームが縮小されると子フレームが (部分的に) 不可視になるかもしれないことを意味しています。親フレームのリサイズ時に常に子フレームを比例してリサイズおよび再配置するためにパラメーター `keep-ratio`を使用できます (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照)。これにより親フレームが縮小された際にフレームの一部が隠されることを防ぐことができます。

可視な子フレームは常に親フレームの最上位に表示されるので、親フレームの下位に表示可能な NS ビルド以外では親フレームの一部を隠すことになります。これは常に親ウィンドウであるデスクトップのルートウィンドウの最上位に表示されるトップレベルフレームのウィンドウシステムのウィンドウに相当します。親フレームがアイコン化されたり不可視 (Section 29.11 [Visibility of Frames], page 736 を参照) になったときには子フレームは不可視になります。親フレームが非アイコン化されたり可視になると子フレームは可視になります。

親フレームが削除される際には、その前に子フレームが再帰的に削除されます (Section 29.7 [Deleting Frames], page 730 を参照)。この規則には 1 つの例外があります。子フレームが他のフレームの代理ミニバッファーフレーム (Section 29.9 [Minibuffers and Frames], page 732 を参照) を果たす際には、親フレームが削除されるまで削除されずに留まります。。この時点でその子フレームをミニバッファーとして使用するフレームが残っていなければ、Emacs は子フレームの削除も試みます。理由は何であれこの削除が失敗すると、その子フレームがトップレベルのフレームになります。

子フレームがメニューバーやツールバーをもてるかどうかはウィンドウシステムやウィンドウマネージャーに依存します。ほとんどのウィンドウシステムは子フレームのメニューバーを明示的に許可していません。フレームの初期パラメーターのセッティングでメニューバーとツールバーの両方を無効にすることを推奨します。

子フレームは通常はタイトルバーやエクスターナルボーダー (Section 29.3 [Frame Geometry], page 701 を参照) のようなウィンドウマネージャーの装飾を表示しません。子フレームがメニューバーやツールバーを表示しないときには他の種類のフレームのボーダーをエクスターナルボーダーのかわりに使用できます (Section 29.4.3.4 [Layout Parameters], page 719 を参照)。

特に X(ただし GTK+ビルド以外) ではフレームのアウターボーダーを使用できます。MS-Windows では非 0 のアウターボーダーを指定することにより、幅が 1 ピクセルのエクスターナルボーダーが表示されます。すべてのウィンドウシステムにおいてインターナルボーダーを使用できます。いずれのケースでもフレームパラメーター `undecorated` (Section 29.4.3.8 [Management Parameters], page 723 を参照) で子フレームにたいするウィンドウマネージャーの装飾を無効にすることを推奨します。

マウスで装飾されていない子フレームのリサイズや移動を行うためには、特別なフレームパラメーターを使う必要があります (Section 29.4.3.7 [Mouse Dragging Parameters], page 723 を参照)。子フレームのインターナルボーダーが存在する場合には、そのフレームが非`nil`の`drag-internal-border`パラメーターをもっていればマウスによるフレームのリサイズに使用できます。`snap-width`がセットされていれば、それは親フレームのエッジやコーナーそれぞれでフレームをスナップ *(snaps)* するピクセル数を表します。

マウスで子フレーム全体をドラッグするためには 2 つの方法があります。`drag-with-mode-line`パラメーターが非`nil`なら、ミニバッファーウィンドウのないフレーム (Section 20.11 [Minibuffer Windows], page 371 を参照) の最下ウィンドウのモードラインエリアを通じてドラッグできます。`drag-with-header-line`パラメーターが非`nil`なら、フレームの最上ウィンドウのヘッダーラインを通じたドラッグが可能です。

子フレームにドラッグ可能なヘッダーラインやモードラインを与えるためには、ウィンドウパラメーター `mode-line-format`と `header-line-format` (Section 28.27 [Window Parameters], page 688 を参照) を使用するのが手軽です。これらにより (`drag-with-header-line`の選択時に) 不要なモードラインを削除したり、フレームのドラッグと干渉するマウス感応エリアを削除できます。

親のネイティブフレームの完全に外部にフレームがドラッグで移動 (マウスカーソルが行き過ぎてしまいマウスボタンのリリース時にフレームの取得が困難になる際に発生し得る) されないようにするためには、それに応じてフレームのパラメーター `top-visible`と `bottom-visible`をセットすることを推奨します。

`top-visible`パラメーターは親のネイティブフレーム内でドラッグ中にフレームの上部で常に可視に留まるピクセル数を指定するもので、これは `drag-with-header-line`パラメーターに非 `nil`を指定する際にセットする必要があります。`bottom-visible`パラメーターは親のネイティブフレーム内でドラッグ中にフレームの下部で常に可視に留まるピクセル数を指定するもので、これは `drag-with-mode-line`パラメーターに非 `nil`を指定する際にセットする必要があります。

`display-buffer-in-child-frame` (Section 28.13.2 [Buffer Display Action Functions], page 641 を参照) を介してバッファー表示に子フレームが使用されている際には、バッファーを表示中のウィンドウが quit される際にフレームを適切に処理するために、フレームの `auto-hide-function`パラメーターに関数をセットできます (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照)。

たとえば別のウィンドウに補完を表示する等でミニバッファーとの相互作用中子フレームを使用する際には、ミニバッファーの exit 時にフレームを適切に処理するために `minibuffer-exit`パラメーター (Section 29.4.3.6 [Frame Interaction Parameters], page 721 を参照) が便利です。

子フレームの振る舞いは他のいくつかの点においても、トップレベルのフレームから逸脱しています。それらのいくつかを以下に挙げます:

- 子フレームにたいする最大化とアイコン化の意味はウィンドウシステムに大きく依存する。原則としてアプリケーションは子フレームでこれらのオプションを決して呼び出すべきではない。デフォルトでは子フレームで `iconify-frame`を呼び出すと子フレームにに対応するトップレベルのフレームにたいしてアイコン化を試みる。異なる挙動を得るためには、以下で説明するオプション `iconify-child-frame`をカスタマイズできる。
- 子フレームの raise、lower、restack (Section 29.12 [Raising and Lowering], page 737 を参照) や `z-group` (Section 29.4.3.2 [Position Parameters], page 715 を参照) の変更では、同じ親をもつ子フレームの stack 順だけが変更される。
- ウィンドウシステムの多くは子フレームの透明度 (Section 29.4.3.10 [Font and Color Parameters], page 726 を参照) を変更できない。
- いくつかのウィンドウシステムでは、祖先のウィンドウの可視部分のマウスクリックによる、子フレームから親以外の祖先へのフォーカスの移動は失敗する。最初に直接的な親のウィンドウシステムのウィンドウを直接クリックする必要があるだろう。
- ウィンドウマネージャーマウスポリシーにしたがってフォーカスを子フレームに拡大しないかもしれない。この問題にたいして `mouse-autoselect-window`のカスタマイズが助けになるかもしれない (Section 28.25 [Mouse Window Auto-selection], page 685 を参照)。
- 子フレームへのドロップ (Section 29.21 [Drag and Drop], page 748 を参照) はすべてのウィンドウシステムで動作を保証されない。親フレームにオブジェクトをドロップしたり、他の祖先にドロップするものもある。

以下の 2 つの関数は子フレームと親フレームで処理を行う際に役に立つかもしれません:

`frame-parent` **&optional** *frame* [Function]
この関数は *frame*の親フレームをリターンする。*frame*の親フレームはウィンドウシステムのウィンドウが *frame*のウィンドウシステムのウィンドウの親ウィンドウであるような Emacs フレーウである。そのようなフレームが存在すれば、*frame*はそのフレームの子フレームとみなされる。

この関数は *frame*に親フレームがなければ `nil`をリターンする。

`frame-ancestor-p` *ancestor descendant* [Function]
この関数は *ancestor*が *descendant*の祖先なら非`nil`をリターンする。*ancestor*が *descendant*の親フレームか *descendant*の親フレームの祖先なら、*ancestor*は *descendant*の祖先である。*ancestor*と *descendant*にはいずれも生きたフレームを指定しなければならない。

既存ウィンドウの最大の空エリア内への子フレームの描画に使用できる関数 `window-largest-empty-rectangle`にも注意してください (Section 28.24 [Coordinates and Windows], page 681 を参照)。これは子フレームがウィンドウ内に表示されているテキストを隠さないようにするために有用です。

子フレームにたいする `iconify-frame`の挙動の調整に以下のオプションのカスタマイズが役に立つかもしれません。

`iconify-child-frame` [User Option]
このオプションは Emacs にたいして子フレームのアイコン化を要求された際に処理を行う方法を指定する。`nil`なら `iconify-frame`が子フレームに呼び出された際には何も行わない。`iconify-top-level`なら子フレームの祖先であるトップレベルのフレームをアイコン化する。`make-invisible`ならアイコン化せずに子フレームを不可視にしようと試みる。

その他の値は子フレームのアイコン化を試みることを意味する。そのような試みはすべてのウィンドウマネージャーで許容されるとはかぎらず、子フレームがユーザーのアクションに無応答になることさえあり得るので、デフォルトではトップレベルのフレームをアイコン化する。

29.15 マウスの追跡

マウスをトラック *(track:* 追跡*)* するのが有用なことが時折あります。マウスのトラックとはマウスの位置を示す何かを表示して、マウス移動とともにそのインジケーターを移動するという意味です。効果的にマウスをトラックするためには、マウスが実際に移動するまで待機する手段が必要になります。

マウスをトラックするためには、マウスのモーション (motion: 移動) を表すイベントを問い合わせるのが便利な方法です。その後はそのイベントを待機することによりモーションを待機できます。それに加えて発生し得る他の類のイベントも簡単に処理できます。ボタンのリリースのような何か他のイベントだけを待機してマウスを永久にトラックすることは、通常は望ましくないのでこれは有用です。

`track-mouse` *body...* [Macro]
このマクロはマウスモーションイベントの生成を有効にして *body*を実行する。*body*はモーションイベントを読み取るために、通常は `read-event`を使用してそれに対応して表示を変更する。マウスモーションイベントのフォーマットについては Section 21.7.8 [Motion Events], page 395 を参照のこと。

`track-mouse`の値は *body*内の最後のフォームの値。ボタンのリリースを示す up-event、またはトラックを止めるべきタイミングを意味する類のイベントを確認した際にはリターンするように *body*をデザインすること。

`track-mouse`フォームでは変数`track-mouse`を非`nil`値にバインドすることにより、Emacsにマウスモーションイベントを生成させる。この変数が特別な値`dragging`をもつなら、ディスプレーエンジンにマウスポインターのシェイプ (形状) の変更を控えるように追加で指示する。これはEmacsが表示する大きな範囲を横断するマウスドラッグを要するLispプログラムでは、そうしなければ表示箇所に応じてマウスポインターのシェイプが変更されてしまうので望ましいだろう (Section 29.19 [Pointer Shape], page 746 を参照)。したがってドラッグ中にオリジナルのマウスポインターシェイプを保つ必要があるLispプログラムは、*body*の先頭で`track-mouse`を値`dragging`にバインドすること。

マウスモーションをトラックする通常の目的は、それ以降に発生するボタンのプッシュやリリースをカレント位置に示すことです。

多くの場合はテキストプロパティ`mouse-face`(Section 32.19.4 [Special Properties], page 820 を参照) を使用することにより、マウスをトラックする必要性を回避できます。これはより低レベルで機能して、かつLispレベルのマウストラッキングよりスムーズに実行されます。

29.16 マウスの位置

関数`mouse-position`と`set-mouse-position`はマウスのカレント位置にたいするアクセスを提供します。

`mouse-position` [Function]
この関数はマウス位置の記述子をリターンする。値は`(`*`frame x . y`*`)`のような形式であり、*x*と*y*は*frame*のネイティブ位置 (Section 29.3 [Frame Geometry], page 701 を参照) から相対的に、*frame*のデフォルト文字サイズ (Section 29.3.2 [Frame Font], page 707 を参照) の単位で位置を与える整数 (丸められている可能性あり)。

`mouse-position-function` [Variable]
この変数の値は非`nil`なら`mouse-position`にたいして呼び出される関数。`mouse-position`はリターン直前に、自身の通常のリターン値を唯一の引数としてこの関数を呼び出して、それが何であれその関数がリターンした値をリターンする。

このアブノーマルフックは`xt-mouse.el`のようにLispレベルでマウス処理を行う必要があるパッケージのために存在する。

`set-mouse-position` *frame x y* [Function]
この関数はフレーム*frame*内の位置*x*、*y*にマウスをワープ *(warps the mouse)* する。引数*x*と*y*は*frame*のネイティブ位置 (Section 29.3 [Frame Geometry], page 701 を参照) から相対的に、*frame*のデフォルト文字サイズ (Section 29.3.2 [Frame Font], page 707 を参照) の単位で位置を与える整数 (丸められている可能性あり)。

結果となるマウス位置は*frame*のネイティブフレームに拘束される。この関数は*frame*が不可視なら何も行わない。リターン値に意味はない。

`mouse-pixel-position` [Function]
この関数は`mouse-position`と似ているが文字単位ではなくピクセル単位の座標をリターンする。

`set-mouse-pixel-position` *frame x y* [Function]
この関数は`set-mouse-position`のようにマウスをワープするが、*x*と*y*が文字単位ではなくピクセル単位である点が異なる。

結果となるマウス位置は*frame*のネイティブフレームに拘束される。この関数は*frame*が不可視なら何も行わない。リターン値に意味はない。

フラフィカルな端末上では、以下の 2 つの関数によりマウスカーソルの絶対位置の取得とセットができます。

`mouse-absolute-pixel-position` [Function]
この関数は選択されたフレームのディスプレーの位 (0, 0) から相対的に、マウスカーソルの位置の座標をピクセル単位のコンスセル (*x* . *y*) でリターンする。

`set-mouse-absolute-pixel-position` *x y* [Function]
この関数はマウスカーソルを位置 (x, *y*) に移動する。座標 *x* と *y* は、選択されたフレームのディスプレーの位置 (0, 0) から相対的なピクセル値と解釈される。

以下の関数はフレーム上のマウスカーソルがレントで可視かどうかを確認します:

`frame-pointer-visible-p` **&optional** *frame* [Function]
この述語関数は *frame* 上に表示されたマウスポインターが可視なら非 `nil`、それ以外は `nil` をリターンする。*frame* が省略または `nil` ならそれは選択されたフレームを意味する。これは `make-pointer-invisible` が `t` にセットされているときに有用。これによりポインターが隠されていることを知ることができる。Section “Mouse Avoidance” in *The Emacs Manual* を参照のこと。

29.17 ポップアップメニュー

Lisp プログラムがポップアップメニューを表示して、ユーザーがマウスで候補を選択できます。テキスト端末上では、マウスが利用不可ならキーボードのモーションキー *C-n* や *C-p*、上矢印キーや下矢印キーで候補を選択できます。

`x-popup-menu` *position menu* [Function]
この関数はポップアップメニューを表示して、ユーザーが何を選択したかの指標をリターンする。

引数 *position* には、メニュー左上隅をスクリーン上のどこに置くか指定する。これはマウスボタンイベント (ユーザーがボタンを操作した位置にメニューを置くよう指示する)、または以下の形式のリストのいずれか:

```
((xoffset yoffset) window)
```

ここで *xoffset* と *yoffset* は *window* の左上隅からピクセル単位で測られた座標である。*window* はウィンドウかフレーム。

position が `t` なら、それはマウスのカレント位置の使用を意味する (テキスト端末上でマウスが利用不可ならフレーム左上隅)。*position* が `nil` なら、それは実際にメニューをポップアップせずに、*menu* 内で指定されたキーマップと等価なキーバインディングを事前に計算することを意味する。

引数 *menu* はメニュー内で何を表示するかを意味する。これはキーマップかキーマップのリストを指定できる (Section 22.17 [Menu Keymaps], page 447 を参照)。この場合にはリターン値はユーザー選択に対応するイベントのリスト。選択がサブメニュー内で発生した場合には、このリストには複数の要素がある (`x-popup-menu` はそのイベントシーケンスにバインドされたコマンドを実際には実行しないことに注意)。テキスト端末やメニュータイトルをサポートするツールキットでは、*menu* がキーマップならタイトルは *menu* のプロンプト文字列、*menu* がキーマップのリストなら最初のキーマップのプロンプト文字列から取得される (Section 22.17.1 [Defining Menus], page 447 を参照)。

かわりに *menu* は以下の形式をもつこともできる:

```
(title pane1 pane2...)
```

ここで pane はそれぞれ以下の形式のリストである

```
(title item1 item2...)
```

*item*はそれぞれコンスセル (`line . value`)であること。ここで *line*は文字列、*value*は *line* が選択された場合にリターンされる値。メニューキーマップとは異なり `nil`の *value*は選択不可のメニューアイテムを作成しない。かわりに *item*にコンスセルではなく文字列を指定できる。これは選択不可のメニューアイテムを作成する。

たとえば有効な選択からマウスを外してクリックしたり、*C-g*をタイプすることにより、有効な選択を行うことなくユーザーがメニューを取り除いた場合は、通常は quit して `x-popup-menu` はリターンしない。しかし *position*がマウスボタンイベント (ユーザーがマウスでメニューを呼び出したことを示す) なら、quit は発生せずに `x-popup-menu`はリターンする。

使用上の注意**:** メニューキーマップで定義したプレフィクスキー処理を行える場合には、メニューの表示に `x-popup-menu`を使用しないでください。メニューの実装にメニューキーマップを使用する場合には、*C-h c*と *C-h a*でメニュー内の個別アイテムの確認、およびそれらにたいするヘルプを提供できます。かわりに `x-popup-menu`を呼び出すコマンドを定義することによりメニューを実装した場合には、ヘルプ機能はそのコマンド内部で何が起こっているか知ることができず、そのメニューアイテムのヘルプを何も与えることはできません。

マウス移動によってサブメニュー間を切り替えるメニューバーのメカニズムは、それが `x-popup-menu`を呼び出すか確認するためにコマンドの定義を見ることができません。したがって `x-popup-menu` を使用してサブメニューの実装を試みた場合には、それは統合された方式でメニューバーとともに機能しません。メニューバーのすべてのサブメニューは親メニューのメニューキーマップにより実装されて、決して `x-popup-menu`で実装されないのはこれが理由です。Section 22.17.5 [Menu Bar], page 453 を参照してください。

メニューバーのサブメニューのコンテンツを変化させたい場合にも、その実装には依然としてメニューキーマップを使用するべきです。コンテンツを変化させるためには、必要に応じてメニューキーマップのコンテンツを更新するためにフック関数を `menu-bar-update-hook`に追加してください。

29.18 ダイアログボックス

ダイアログボックスとはポップアップメニューの一種です。外見は多少異なり常にフレーム中央に表示されて、階層を 1 つしかもたず 1 つ以上のボタンがあります。ユーザーが “yes”、“no”、および別のいくつかの候補で応答ができる質問を尋ねるのがダイアログボックスの主な用途です。単一のボタンではユーザーに重要な情報の確認を強いることもできます。関数 `y-or-n-p`や `yes-or-no-p`は、マウスのクリックで呼び出されたコマンドから呼び出された際には、キーボードのかわりにダイアログボックスを使用します。

`x-popup-dialog` *position contents* **&optional** *header* [Function]

この関数はポップアップダイアログボックスを表示してユーザーが何を選択したかの指標をリターンする。引数 *contents*は提供する選択肢を指定する。これは以下のフォーマットをもつ:

```
(title (string . value)...)
```

これは `x-popup-menu`にたいして単一の pane を指定するリストのように見える。

リターン値は選択された候補の *value*。

`x-popup-menu`の場合と同じように、このリストの要素はコンスセル (`string . value`)のかわりに単なる文字列かもしれない。これは選択不可のボックスを作成する。

このリスト内に nilがある場合には、それは左手側と右手側のアイテムを分ける。つまり nil より前のアイテムは左、nilより後のアイテムは右に表示される。リスト内に nilを含めない場合には、およそ半数ずつが両サイドに表示される。

ダイアログボックスは常にフレームの中央に表示される。引数 *position*はどのフレームかを指定する。可能な値は x-popup-menuの場合と同様だが、正確な座標や個別のウィンドウは問題ではなくフレームだけが問題となる。

*header*が非 nilならボックスのフレームタイトルは 'Information'、それ以外は 'Question' になる。前者は message-box ([message-box], page 985 を参照) にたいして使用される (テキスト端末上ではボックスタイトルは表示されない)。

いくつかの構成では Emacs は本当のダイアログボックスを表示できないので、かわりにフレーム中央のポップアップメニュー内に同じアイテムを表示する。

たとえばウィンドウマネージャーを使用して有効な選択を行うことなくユーザーがダイアログボックスを取り除いた場合には、通常は quit して x-popup-dialogはリターンしない。

29.19 ポインターの形状

テキストプロパティpointerや、イメージならイメージプロパティ:pointerと:mapを使用して、特定のテキストやイメージにたいしてマウスポインターのスタイルを指定できます。これらのプロパティに使用できる値は text (または nil)、arrow、hand、vdrag、hdrag、modeline、hourglassです。textはテキスト上で使用される通常のマウスポインタースタイルを意味します。

ウィンドウの空部分 (void parts: バッファーコンテンツのどの部分にも対応しない部分) の上では、マウスポインターは通常 arrowスタイルを使用しますが、void-text-area-pointerをセットすることにより異なるスタイルを指定できます。

void-text-area-pointer [User Option]
この変数は空テキストエリアにたいするマウスポインタースタイルを指定する。このエリアには行末の後やバッファー終端行の下が含まれる。デフォルトでは arrow(non-text) ポインタースタイルを使用する。

X を使用する際は変数 x-pointer-shapeをセットすることにより textの実際の外見を指定できます。

x-pointer-shape [Variable]
この変数は Emacs フレーム内でポインタースタイル textに通常使用するポインターシェイプを指定する。

x-sensitive-text-pointer-shape [Variable]
この変数はマウスがマウスセンシティブテキスト上にあるときのポインターシェイプを指定する。

これらの変数は新たに作成されるフレームに影響します。これらは通常は既存のフレームに効果はありませんが、フレームのマウスカラーのインストール時にはこれら 2 つ変数のカレント値もインストールされます。Section 29.4.3.10 [Font and Color Parameters], page 726 を参照してください。

これらのポインターシェイプのいずれかを指定するために使用可能な値はファイル lisp/term/x-win.el内で定義されています。それらのリストを確認するには *M-x apropos RET x-pointer RET*を使用してください。

29.20 ウィンドウシステムによる選択

X のようなウィンドウシステムでは、異なるアプリケーション間のデータ転送は選択 *(selections)* により行われます。X は任意の数の選択タイプ *(selection types)* を定義し、それぞれが独自にデータを格納できます。しかし一般的に使用されるのはクリップボード *(clipboard)*、プライマリー選択 *(primary selection)*、セカンダリー選択 *(secondary selection)* の 3 つだけです。それ以外のウィンドウシステムではクリップボードだけがサポートされます。これら 3 つの選択を使用する Emacs コマンドについては Section "Cut and Paste" in *The GNU Emacs Manual* を参照してください。このセクションではウィンドウシステムによる選択の読み取りとセットを行う低レベル関数について説明します。

`gui-set-selection` *type data* [Command]
この関数はウィンドウシステムの選択をセットする。これは選択タイプ *type*、それに割り当てる値 *data*という 2 つの引数を受け取る。

*type*はシンボルであること。通常は `PRIMARY`、`SECONDARY`、`CLIPBOARD`のいずれかである。これらは X ウィンドウシステムの慣例に対応する大文字のシンボル名である。*type*が `nil`ならそれは `PRIMARY`を意味する。

*data*が `nil`なら、それはその選択をクリアーすることを意味する。それ以外なら *data*は文字列、シンボル、整数 (2 つの整数からなるコンスかリスト)、オーバーレイ、同じバッファーを指す 2 つのマーカーのコンスを指定できる。オーバーレイとマーカーのペアは、そのオーバーレイまたはマーカー間のテキストを意味する。引数 *data*には有効な非ベクターの選択のベクターも指定できる。

この関数は *data*をリターンする。

`gui-get-selection` **&optional** *type data-type* [Function]
この関数は Emacs や他のプログラムによりセットアップされた選択にアクセスする。これは *type*と *data-type*の 2 つの引数を受け取る。*type*は選択のタイプでありデフォルトは `PRIMARY`。

*data-type*引数は別の X クライアントから取得した raw データを Lisp データに変換するためにデータ変換に使用する形式を指定する。有意義な値には `TEXT`、`STRING`、`UTF8_STRING`、`TARGETS`、`LENGTH`、`DELETE`、`FILE_NAME`、`CHARACTER_POSITION`、`NAME`、`LINE_NUMBER`、`COLUMN_NUMBER`、`OWNER_OS`、`HOST_NAME`、`USER`、`CLASS`、`ATOM`、`INTEGER`が含まれる (これらは対応する X 慣習の大文字シンボル名である)。*data-type*のデフォルトは `STRING`。X 以外のウィンドウシステムは、通常は `STRING`に加えて、少数の部分集合だけをサポートする。

`selection-coding-system` [User Option]
この変数は選択やクリップボードに読み書きする際のコーディングシステムを指定する。Section 33.10 [Coding Systems], page 864 を参照のこと。デフォルトは `compound-text-with-extensions`で、これは X11 が通常使用するテキスト表現に変換する。

Emacs が MS-Windows 上で実行されている際には、一般的に X 選択はサポートされませんがクリップボードはサポートされます。MS-Windows での `gui-get-selection`と `gui-set-selection`は、テキストデータタイプだけをサポートします。クリップボードが他のタイプのデータを保持している場合には、Emacs はクリップボードを空として扱います。サポートされるデータタイプは `STRING`です。

後方互換性のために `gui-get-selection`と `gui-set-selection`にたいして、Emacs 25.1 以前の名前 `x-get-selection`と `x-set-selection`が時代遅れのエイリアスとして存在します。

29.21 ドラッグアンドドロップ

ユーザーが別のアプリケーションから Emacs に何かをドラッグをした際には、その別アプリケーションは Emacs がドラッグされたデータを処理可能か告げることを期待します。変数 `x-dnd-test-function`は何を応答するか決定するために Emacs により使用されます。デフォルト値は `x-dnd-default-test-function`で、これはドロップされたデータのタイプが `x-dnd-known-types`内にあればドロップを受け入れます。何か別の条件にもとづいて Emacs にドロップを許容または拒絶させたければ、`x-dnd-test-function`および/または `x-dnd-known-types`をカスタマイズできます。

Emacs が異なるタイプのドロップを処理する方法を変更したり新たなタイプを追加したければ `x-dnd-types-alist`をカスタマイズします。これには他のアプリケーションがドラッグアンドドロップに使用するタイプが何なのか詳細な知識が要求されます。

Emacs に URL がドロップされたとき、それはファイルかもしれませんが他の URL タイプ (http、...) であるかもしれません。Emacs はまずその URL に何を行うべきか判断するために、`dnd-protocol-alist`をチェックします。それにマッチがなく、かつ `browse-url-browser-function`が alist なら Emacs はそこでマッチを探します。それでもマッチが見つからなければ、その URL にたいするテキストを挿入します。これらの変数をカスタマイズすれば Emacs の挙動を変更できます。

29.22 カラー名

カラー名 (color name) とはカラーを指定するテキスト (通常は文字列) です。'`black`'、'`white`'、'`red`'等を指定できます。定義された名前のリストは *`M-x list-colors-display`*を使用して確認できます。'`#rgb`'や'`RGB:r/g/b`'のような数値的な形式でカラーを指定することもできます。ここで *r* は赤 (red)、*g* は緑 (green)、*b* は青 (blue) のレベルを指定します。1 桁、2 桁、3 桁、または 4 桁の 16 進数を *r* に使用できます。その後の *g* と *b* には同じ桁数の 16 進数を同様に使用しなければなりません。これにより総桁数が 3^^ef^^bd^^a46^^ef^^bd^^a49^^ef^^bd^^a4 または 12 桁の 16 進数となります (カラーの数値的な RGB 指定についての詳細は X ウィンドウシステムのドキュメントを参照)。

以下の関数は有効なカラー名と、それらの外見を判断する手段を提供します。以下で説明するようにその値は選択されたフレーム *(selected frame)* に依存する場合があります。"選択されたフレーム"という用語の意味については Section 29.10 [Input Focus], page 732 を参照してください。

補完付きでカラー名のユーザー入力を読み取るには `read-color`を使用します (Section 20.6.4 [High-Level Completion], page 357 を参照)。

`color-defined-p` *color* **&optional** *frame* [Function]

この関数はカラー名が有意かどうかを報告する。もし有意なら `t`、それ以外は `nil`をリターンする。引数 *frame*はどのフレームのディスプレイにたいして問い合わせるかを指定する。*frame* が省略または `nil`の場合は選択されたフレームが使用される。

これは使用しているディスプレイがそのカラーをサポートするかどうかは告げないことに注意。X 使用時にはすべての種類のディスプレイ上のすべての定義されたカラーを問い合わせることができ、何らかの結果 (通常は可能な限り近いカラー) を得ることができるだろう。あるフレームが特定のカラーを実際に表示できるかどうか判断するためには `color-supported-p` (以下参照) を使用する。

この関数は以前は `x-color-defined-p`と呼ばれており、その名前は今でもエイリアスとしてサポートされている。

`defined-colors` **&optional** *frame* [Function]
この関数は *frame*(デフォルトは選択されたフレーム) 上で定義されていて、かつサポートされるカラー名のリストをリターンする。*frame*がカラーをサポートしなければ値は `nil`。

この関数は以前は `x-defined-colors`と呼ばれており、その名前は今でもエイリアスとしてサポートされている。

`color-supported-p` *color* **&optional** *frame background-p* [Function]
これは、*frame*が実際にカラー *color* (または最低でもそれに近いカラー) を表示可能なら `t`をリターンする。*frame*が省略または `nil`ならこの問いは選択されたフレームに適用される。

フォアグラウンドとバックグラウンドにたいして異なるカラーセットをサポートする端末がいくつかある。*background-p*が非 `nil`なら、それは *color*がバックグラウンドとして、それ以外はフォアグラウンドとして使用可能かどうかを問うことを意味する。

引数 *color*は有効なカラー名でなければならない。

`color-gray-p` *color* **&optional** *frame* [Function]
これは *color*が *frame*のディスプレイ上の定義としてグレイスケールなら `t`をリターンする。*frame*が省略または `nil`なら、この問いは選択されたフレームに適用される。*color*が有効なカラー名でなければ、この関数は `nil`をリターンする。

`color-values` *color* **&optional** *frame* [Function]
この関数は *frame*上で理想的には *color*がどのように見えるべきかを記述する値をリターンする。*color*が定義済みなら値は赤、緑、青の割合を与える 3 つの整数からなるリストとなる。それぞれの整数の範囲は原則として 0 から 65535 だが、この範囲全体を使用しないディスプレイもいくつか存在するだろう。この 3 要素のリストはカラーの *RGB* 値 *(rgb values)* と呼ばれる。

*color*が未定義なら値は `nil`。

```
(color-values "black")
     ⇒ (0 0 0)
(color-values "white")
     ⇒ (65280 65280 65280)
(color-values "red")
     ⇒ (65280 0 0)
(color-values "pink")
     ⇒ (65280 49152 51968)
(color-values "hungry")
     ⇒ nil
```

カラーの値は *frame*のディスプレイにたいしてリターンされる。*frame*が省略または `nil`の場合には、この情報は選択されたフレームのディスプレイにたいしてリターンされる。このフレームがカラーを表示できなければ値は `nil`。

この関数は以前は `x-color-values`と呼ばれており、その名前は今でもエイリアスとしてサポートされている。

29.23 テキスト端末のカラー

テキスト端末は通常は少しのカラーしかサポートせず、コンピューターはカラー選択に小さい整数を使用します。これは選択したカラーがどのように見えるかコンピューターが信頼性をもって告げることができず、どのカラーがどのような小さい整数に対応するかという情報をアプリケーションに伝え

る必要があることを意味します。しかし Emacs は標準的なカラーセットを知っており、それらの自動的な使用を試みるでしょう。

このセクションで説明する関数は Emacs が端末カラーを使用する方法を制御します。

これらの関数のうちのいくつかは Section 29.22 [Color Names], page 748 で説明した *RGB* 値 *(rgb values)* を使用またはリターンします。

これらの関数はオプション引数としてディスプレイ (フレームまたは端末名のいずれか) を受け取ります。わたしたちは将来には異なる端末上で異なるカラーを Emacs にサポートさせたいと望んでいます。そうすればこの引数はどの端末を処理するか (デフォルトは選択されたフレームの端末。Section 29.10 [Input Focus], page 732 を参照) を指定するようになるでしょう。しかし現在のところ *frame*引数に効果はありません。

`tty-color-define` *name number* **&optional** *rgb frame* [Function]
この関数はカラー名 *name*をその端末上のカラー値 *number*に関連付ける。

オプション引数 *rgb*が指定された場合、それはそのカラーが実際にどのように見えるかを指定する 3 つの数値のリストからなる RGB 値である。*rgb*を指定しなければ Emacs はそれがどのように見えるか知らないので、そのカラーを他のカラーに近似するために `tty-color-approximate` で使用することができない。

`tty-color-clear` **&optional** *frame* [Function]
この関数はテキスト端末の定義済みカラーのテーブルをクリアーする。

`tty-color-alist` **&optional** *frame* [Function]
この関数はテキスト端末がサポートする既知のカラーを記録した alist をリターンする。

それぞれの要素は `(name number . rgb)`、または `(name number)`という形式をもつ。ここで *name*はカラー名、*number*はその端末でカラー指定に使用される数値。*rgb*が与えられたら、それはそのカラーが実際にどのように見えるかを告げる 3 つのカラー値 (赤、緑、青) のリストである。

`tty-color-approximate` *rgb* **&optional** *frame* [Function]
この関数は *display*にたいしてサポートされた既知のカラーの中から、RGB 値 *rgb* (カラー値のリスト) で記述されたもっとも近いカラーを探す。リターン値は `tty-color-alist`の要素。

`tty-color-translate` *color* **&optional** *frame* [Function]
この関数は *display*にたいしてサポートされた既知のカラーの中から、もっとも近いカラーのインデックス (整数) をリターンする。名前 *color*が未定義なら値は `nil`。

29.24 X リソース

このセクションでは X リソース、または他のオペレーティングシステム上での等価物を問い合わせたり使用する関数および変数をいくつか説明します。X リソースにたいする詳細な情報は Section “X Resources” in *The GNU Emacs Manual* を参照してください。

`x-get-resource` *attribute class* **&optional** *component subclass* [Function]
関数 `x-get-resource`は X ウィンドウのデフォルトデータベースからリソース値を取得する。

リソースはキー *(key)* とクラス *(class)* の組み合わせによりインデックス付けされている。この関数は‘*instance*`.`*attribute*’という形式をキー (*instance*は Emacs が呼び出されたときの名前)、クラスとして‘`Emacs.`*class*’として使用することにより検索を行う。

オプション引数 *component*と *subclass*は、それぞれキーとクラスを追加する。指定する場合には両方を指定するか、さもなくばどちらも指定してはならない。これらを指定した場合にはキーは‘*instance.component.attribute*’、クラスは‘Emacs.*class.subclass*’となる。

`x-resource-class` [Variable]
この変数は `x-get-resource`が照会すべきアプリケーション名を指定する。デフォルト値は `"Emacs"`。`x-get-resource`の呼び出し周辺で、この変数を他のアプリケーション名の文字列にバインドすることにより、アプリケーション名にたいして X リソースを調べることができる。

`x-resource-name` [Variable]
この変数は `x-get-resource`が照会すべきインスタンス名を指定する。デフォルト値は Emacs 呼び出し時の名前、またはスイッチ‘`-name`’、または‘`-rn`’で指定された値。

上述のいくつかを説明するために X リソースファイル (通常は`~/.Xdefaults`や`~/.Xresources`) 内に以下のような行があるとしましょう:

```
xterm.vt100.background: yellow
```

その場合は:

```
(let ((x-resource-class "XTerm") (x-resource-name "xterm"))
  (x-get-resource "vt100.background" "VT100.Background"))
     ⇒ "yellow"
(let ((x-resource-class "XTerm") (x-resource-name "xterm"))
  (x-get-resource "background" "VT100" "vt100" "Background"))
     ⇒ "yellow"
```

`inhibit-x-resources` [Variable]
この変数が非 `nil`なら Emacs は X リソースを照会せず、新たなフレーム作成時に X リソースは何も効果をもたない。

29.25 ディスプレー機能のテスト

このセクションの関数は特定のディスプレイの基本的な能力を説明します。Lisp プログラムはそのディスプレイが行えることに挙動を合わせるために、それらを使用できます。たとえばポップアップメニューがサポートされなければ、通常はポップアップメニューを使用するプログラムはミニバッファーを使用できます。

これらの関数のオプション引数 *display*は問い合わせるディスプレイを指定します。これにはディスプレイ名、フレーム (フレームがあるディスプレイを指定)、または `nil` (選択されたフレームのディスプレイを参照する。Section 29.10 [Input Focus], page 732 を参照) を指定できます。

ディスプレイに関する情報を取得するその他の関数については Section 29.22 [Color Names], page 748 を参照してください。

`display-popup-menus-p` **&optional** *display* [Function]
この関数は *display*上でポップアップメニューがサポートされていれば `t`、それ以外は `nil`をリターンする。Emacs ディスプレイのある部分をマウスでクリックすることによりメニューがポップアップするので、ポップアップメニューのサポートにはマウスが利用可能であることが要求される。

`display-graphic-p` **&optional** *display* [Function]
この関数は *display*が一度に複数フレームおよび複数の異なるフォントを表示する能力を有すグラフィックディスプレイなら `t`をリターンする。これは X のようなウィンドウシステムのディスプレイにたいしては真、テキスト端末にたいしては偽となる。

`display-mouse-p` **&optional** *display* [Function]
この関数は *display*でマウスが利用可能なら `t`、それ以外は `nil`をリターンする。

`display-color-p` **&optional** *display* [Function]
この関数はそのスクリーンがカラースクリーンなら `t`をリターンする。これは以前は `x-display-color-p`と呼ばれており、その名前はエイリアスとして今でもサポートされる。

`display-grayscale-p` **&optional** *display* [Function]
この関数はスクリーンがグレースケールを表示可能なら `t`をリターンする (カラーディスプレイはすべてこれを行うことができる)。

`display-supports-face-attributes-p` *attributes* **&optional** *display* [Function]
この関数は *attributes*内のすべてのフェイス属性がサポートされていれば非 `nil`をリターンする (Section 39.12.1 [Face Attributes], page 1012 を参照)。

幾分発見的ではあるが “サポートされる” という言葉は、基本的にはあるフェイスが *attributes* 内のすべての属性を含み、ディスプレイにたいしてデフォルトフェイスにマージ時に、

1. デフォルトフェイスとは異なる外見で表示でき、かつ
2. 指定した属性と正確に一致しない場合はより近い (close in spirit) 外見で表示する

2 つ目のポイントによると属性`:weight black`は太字 (bold) 表示可能、同様に属性`:foreground "yellow"`は黄色がかった何らかのカラーを表示可能なすべてのディスプレイで満たされるだろうが、属性`:slant italic`は斜体 (italic) を自動的に淡色 (dim) に置き換える tty の表示コードでは満たされないであろうことを暗示している。

`display-selections-p` **&optional** *display* [Function]
この関数は *display*が選択 (selections) をサポートすれば `t`をリターンする。ウィンドウ化されたディスプレイでは通常は選択がサポートされるが、他の場合にもサポートされ得る。

`display-images-p` **&optional** *display* [Function]
この関数は *display*がイメージを表示可能なら `t`をリターンする。ウィンドウ化されたディスプレイは原則イメージを処理するが、イメージにたいするサポートを欠くシステムもいくつかある。イメージをサポートしないディスプレイ上では Emacs はツールバーを表示できない。

`display-screens` **&optional** *display* [Function]
この関数はそのディスプレイに割り当てられたスクリーンの数をリターンする。

`display-pixel-height` **&optional** *display* [Function]
この関数はスクリーンの高さをピクセルでリターンする。文字端末では文字数で高さを与える。

マルチモニターにセットアップされているグラフィカル端末では、*display*に割り当てられたすべての物理モニターのピクセル幅を参照することに注意。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`display-pixel-width` **&optional** *display* [Function]
この関数はスクリーンの幅をピクセルでリターンする。文字端末では文字数で幅を与える。

マルチモニターにセットアップされているグラフィカル端末では、*display*に割り当てられたすべての物理モニターのピクセル幅を参照することに注意。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`display-mm-height` **&optional** *display* [Function]
この関数はスクリーンの高さをミリメートルでリターンする。`nil`なら Emacs がその情報を取得できなかったことを意味する。

マルチモニターにセットアップされているグラフィカル端末では、*display*に割り当てられたすべての物理モニターのピクセル幅を参照することに注意。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`display-mm-width` **&optional** *display* [Function]
この関数はスクリーンの幅をミリメートルでリターンする。`nil`なら Emacs がその情報を取得できなかったことを意味する。

マルチモニターにセットアップされているグラフィカル端末では、*display*に割り当てられたすべての物理モニターのピクセル幅を参照することに注意。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`display-mm-dimensions-alist` [User Option]
この変数はシステムの提供する値が不正な場合に `display-mm-height`と `display-mm-width`がリターンするグラフィカルなディスプレイのサイズをユーザーが指定できるようにする。

`display-backing-store` **&optional** *display* [Function]
この関数はそのディスプレイのバッキングストアー (backing store) の能力をリターンする。バッキングストアーとは非露出ウィンドウ (およびウィンドウの一部) のピクセルを記録しておいて、露出時に素早く表示できるようにすることを意味する。

値はシンボル `always`、`when-mapped`、`not-useful`。特定の種類のディスプレイにたいしてこの問いが適用外の際には、この関数は `nil`をリターンすることもある。

`display-save-under` **&optional** *display* [Function]
この関数はそのディスプレイが SaveUnder 機能をサポートすれば非 `nil`をリターンする。この機能はポップアップウィンドウに隠されるピクセルを保存して素早くポップダウンができるようにするために使用される。

`display-planes` **&optional** *display* [Function]
この関数はそのディスプレイがサポートする平面数 (number of planes) をリターンする。これは通常はピクセルごとのビット数 (bits per pixel: 色深度 [bpp])。tty ディスプレイではサポートされるカラー数の 2 進対数 (log to base two)。

`display-visual-class` **&optional** *display* [Function]
この関数はそのスクリーンのビジュアルクラスをリターンする。値はシンボル `static-gray` (カラー数変更不可の限定されたグレイ)、`gray-scale` (フルレンジのグレイ)、`static-color` (カラー数変更不可の限定されたカラー)、`pseudo-color` (限定されたカラー数のカラー)、`true-color` (フルレンジのカラー)、および `direct-color` (フルレンジのカラー) のいずれか。

`display-color-cells` **&optional** *display* [Function]
この関数はそのスクリーンがサポートするカラーのセル数をリターンする。

以下の関数は Emacs が指定された *display*を表示する場所に使用されるウィンドウシステムの追加情報を取得します (関数名先頭の `x-`は歴史的理由による)。

`x-server-version` **&optional** *display* [Function]

この関数は GNU および Unix システム上の X サーバーのような、*display* 上で実行されている GUI ウィンドウシステムのバージョン番号のリストをリターンする。値は 3 つの整数からなるリストで、1 つ目と 2 つ目の整数はそのプロトコルのメジャーバージョン番号とマイナーバージョン番号、3 つ目の整数はウィンドウシステムソフトウェア自体のディストリビューター固有のリリース番号。GNU および Unix システムでは、通常これらは X プロトコルのバージョン番号と、X サーバーソフトウェアのディストリビューター固有のリリース番号。MS-Windows では Widows の OS バージョン番号。

`x-server-vendor` **&optional** *display* [Function]

この関数はウィンドウシステムソフトウェアを提供するベンダー (文字列) をリターンする。GNU および Unix システムでは、それが誰であれその X サーバーを配布するベンダーを意味する。MS-Windows では Widows OS のベンダー ID 文字列 (Microsoft)。

X 開発者がソフトウェア配布者を “vendors” とラベル付けしたことは、いかなるシステムも非商業的に開発および配布できないと彼らが誤って仮定したことを示している。

30 ポジション

位置 *(position)* とは、バッファーのテキストの文字のインデックスです。より正確には、位置とは 2 つの文字間 (または最初の文字の前か最後の文字の後) の箇所を識別して、与えられた位置の前あるいは後の文字のように表現することができます。しかし “ある位置にある文字” のように表現することもあり、その場合にはその位置の後にある文字を意味します。

位置は通常は 1 から始まる整数として表されますが、マーカー *(markers)* として表現することもできます。関数は引数に位置 (整数) を期待しますが、代替としてマーカーも受け入れ、通常はそのマーカーが指すのがどのバッファーなのかは無視します。これらの関数はマーカーを整数に変換して、たとえそのマーカーが誤ったバッファーを指していたとしても、まるで引数として単にその整数が渡されたかのようにその整数を使用します。整数に変換できない場所を指すマーカーを整数のかわりに使用するとエラーとなります。Chapter 31 [Markers], page 768 を参照してください。

多くのカーソルモーションコマンドにより使用される関数を提供するフィールド (field) 機能も参照してください (Section 32.19.9 [Fields], page 831 を参照)。

30.1 ポイント

ポイント *(point)* とは多くの編集コマンドにより使用されるバッファーの特別な位置のことです。これらのコマンドには自己挿入型のタイプ文字やテキスト挿入関数が含まれます。その他のコマンドは別の箇所でテキストの編集や挿入ができるようにポイントを移動します。

ポイントは他の位置と同様に特定の文字ではなく、2 つの文字の間 (または最初の文字の前か最後の文字の後) を指します。端末では通常はポイント直後の文字の上にカーソルを表示します。つまりポイントは実際はカーソルのある文字の前にあります。

ポイントの値は 1 より小さくなることはなく、そのバッファーのサイズに 1 を加えた値より大きくなることはありません。ナローイング (Section 30.4 [Narrowing], page 766 を参照) が効力をもつ場合には、ポイントはそのバッファーのアクセス可能な範囲内 (範囲の境界はバッファーの先頭か終端のいずれかの可能性がある) に拘束されます。

バッファーはそれぞれ自身のポイント値をもち、それは他のバッファーのポイント値とは無関係です。ウィンドウもそれぞれポイント値をもち、他のウィンドウ内の同じバッファー上のポイント値とは無関係です。同じバッファーを表示する種々のウィンドウが異なるポイント値をもてるのはこれが理由です。あるバッファーがただ 1 つのウィンドウに表示されているときは、そのバッファーのポイントとそのウィンドウのポイントは通常は同じ値をもち、区別が重要になることは稀です。詳細は Section 28.19 [Window Point], page 671 を参照してください。

`point` [Function]

この関数はカレントバッファー内のポイントの値を整数でリターンする。

```
(point)
     ⇒ 175
```

`point-min` [Function]

この関数はカレントバッファー内のアクセス可能なポイントの最小値をリターンする。これは通常は 1 だがナローイングが効力をもつ場合は、ナローイングしたリージョンの開始位置となる (Section 30.4 [Narrowing], page 766 を参照)。

`point-max` [Function]

この関数はカレントバッファー内のアクセス可能なポイントの最大値をリターンする。これはナローイングされていなければは `(1+ (buffer-size))` だが、ナローイングが効力をもつ場

合は、ナローイングしたリージョンの終端位置となる (Section 30.4 [Narrowing], page 766 を参照)。

`buffer-end` *flag* [Function]
この関数は *flag*が 0 より大なら `(point-max)`、それ以外は `(point-min)`をリターンする。引数 *flag*は数値でなければならない。

`buffer-size` **&optional** *buffer* [Function]
この関数はカレントバッファー内の文字数のトータルをリターンする。ナローイング (Section 30.4 [Narrowing], page 766 を参照) されていなければ、`point-max`はこれに 1 を加えた値をリターンする。

*buffer*にバッファーを指定すると値は *buffer*のサイズになる。

```
(buffer-size)
     ⇒ 35
(point-max)
     ⇒ 36
```

30.2 モーション

モーション関数はポイントのカレント値、バッファーの先頭か終端、または選択されたウィンドウ端のいずれかより相対的にポイントの値を変更します。Section 30.1 [Point], page 755 を参照してください。

30.2.1 文字単位の移動

以下の関数は文字数にもとづいてポイントを移動します。 `goto-char`は基本的なプリミティブであり、その他の関数はこれを使用しています。

`goto-char` *position* [Command]
この関数はカレントバッファー内のポイントの値を *position*にセットする。

ナローイングが効力をもつ場合でも *position*は依然としてバッファー先頭から数えられるが、ポイントをアクセス可能な範囲外に移動することはできない。*position*が範囲外なら、`goto-char`はアクセス可能な範囲の先頭または終端にポイントを移動する。

この関数がインタラクティブに呼び出された際は、*position*の値は数プレフィクス引数、プレフィクス引数が与えられなかった場合はミニバッファーから値を読み取る。

`goto-char`は *position*をリターンする。

`forward-char` **&optional** *count* [Command]
この関数は前方、すなわちバッファーの終端方向にポイントを *count*文字移動する (*count*が負なら後方、すなわちバッファーの先頭方向にポイントを移動する)。*count*が `nil`の場合のデフォルトは 1。

バッファー (ナローイングが効力をもつ場合はアクセス可能な範囲の境界) の先頭か終端を超えて移動を試みるとエラーシンボル `beginning-of-buffer`か `end-of-buffer`のエラーをシグナルする。

インタラクティブな呼び出しでは数プレフィクス引数が *count*となる。

`backward-char` **&optional** *count* [Command]
移動方向が逆であることを除いて、これは `forward-char`と同様。

30.2.2 単語単位の移動

以下で説明する単語をパースする関数は、与えられた文字が単語の一部かどうかを判断するために構文テーブルと `char-script-table`を使用します。Chapter 35 [Syntax Tables], page 906 と Section 33.6 [Character Properties], page 857 を参照してください。

`forward-word` **&optional** *count* [Command]

この関数は *count*の単語数分ポイントを前方に移動する。(*count*が負なら後方に移動する)。*count*が省略または `nil`の場合のデフォルトは 1。インタラクティブな呼び出しでは、*count*は数プレフィクス引数により指定される。

“単語 1 つ移動” とは単語構成文字を横断 (これは単語の先頭を示す) して単語の終わりまで移動を継続することを意味する。デフォルトでは単語を開始や終了させる文字は単語境界 *(word boundaries)* として知られており、これはカレントバッファーの構文テーブル (Section 35.2.1 [Syntax Class Table], page 907 を参照) で定義されるが、後述の `find-word-boundary-function-table`を適切にセットすることによりモードはこれをオーバーライドできる。(`char-script-table`により定義される) 異なるスクリプトに属する文字も単語境界を定義できる (Section 33.6 [Character Properties], page 857 を参照)。いずれにせよ、この関数はバッファーのアクセス可能範囲の境界やフィールド境界 (Section 32.19.9 [Fields], page 831 を参照) を超えてポイントを移動できない。フィールド境界のもっとも一般的な例はミニバッファー内のプロンプト終端である。

バッファー境界やフィールド境界により途中で停止することなく単語 *count*個分の移動が可能なら値は `t`となる。それ以外ではリターン値は `nil`となり、ポイントはバッファー境界またはフィールド境界で停止する。

`inhibit-field-text-motion`が非 `nil`なら、この関数はフィールド境界を無視する。

`backward-word` **&optional** *count* [Command]

この関数は単語の前に遭遇するまで前方ではなく後方に移動することを除いて `forward-word`と同様。

`words-include-escapes` [User Option]

この変数は、`forward-word`と `backward-word`、およびそれらを使用するすべての関数の挙動に影響する。これが非 `nil`なら、構文クラス escape と character-quote 内の文字は単語の一部とみなされる。それ以外なら単語の一部とはみなされない。

`inhibit-field-text-motion` [Variable]

この変数が非 `nil`なら `forward-word`、`forward-sentence`、`forward-paragraph`を含む特定のモーション関数はフィールド境界を無視する。

`find-word-boundary-function-table` [Variable]

この変数は`forward-word`と `backward-word`、およびそれらを使用するすべての挙動に影響する。値は単語境界を検索するための関数の文字テーブル (Section 6.6 [Char-Tables], page 110 を参照)。このテーブル内である文字が非 `nil`のエントリーをもつ場合には、単語がその文字で開始または終了する際に対応する関数が 2 つの引数 *pos*と *limit*で呼び出される。この関数は別の単語境界の位置をリターンすること。具体的には、*pos*が *limit*より小さければ *pos*は単語の先頭にあり関数はその単語の最後の文字の後の位置、それ以外なら *pos*は単語の最後の文字にあり関数はその単語の最初の文字の位置をリターンすること。

`forward-word-strictly` **&optional** *count* [Function]

この関数は `forward-word`と同様だが `find-word-boundary-function-table`による影響を受けない点が異なる。このテーブルをセットする `subword-mode`のようなモードにより単語単

位の移動が変更されている際に挙動を変えるべきではない Lisp プログラムは、`forward-word` のかわりにこの関数を使用すること。

`backward-word-strictly` **&optional** *count* [Function]
この関数は `backward-word`と同様だが、`find-word-boundary-function-table`の影響を受けない点が異なる。`forward-word-strictly`と同様に、構文テーブルだけを考慮して単語単位の移動を行う必要がある際には、`backward-word`のかわりにこの関数を使用すること。

30.2.3 バッファー終端への移動

バッファーの先頭にポイントを移動するには以下のように記述します:

```
(goto-char (point-min))
```

同様にバッファーの終端に移動するには以下を使用します:

```
(goto-char (point-max))
```

以下の 2 つのコマンドは、ユーザーがこれらを行うためのコマンドです。これらはマークをセットしてメッセージをエコーエリアに表示するため、Lisp プログラム内で使用しないよう警告するためにここに記述します。

`beginning-of-buffer` **&optional** *n* [Command]
この関数はバッファー (ナローイングが効力をもつ場合はアクセス可能範囲の境界) の先頭にポイントを移動して、以前の位置にマークをセットする (Transient Mark モードの場合にはマークがすでにアクティブならマークはセットしない)。

*n*が非 `nil`ならバッファーのアクセス可能範囲の先頭から $n/10$ の位置にポイントを配置する。インタラクティブな呼び出しでは *n*は数プレフィクス引数が与えられればその値、それ以外でのデフォルトは `nil`。

警告**:** この関数を Lisp プログラム内で使用してはならない。

`end-of-buffer` **&optional** *n* [Command]
この関数はバッファー (ナローイングが効力をもつ場合はアクセス可能範囲の境界) の終端にポイントを移動して、以前の位置にマークをセットする (Transient Mark モードの場合にはマークがすでにアクティブならマークはセットしない)。*n*が非 `nil`ならバッファーのアクセス可能範囲の終端から $n/10$ の位置にポイントを配置する。

インタラクティブな呼び出しでは *n*は数プレフィクス引数が与えられればその値、それ以外でのデフォルトは `nil`。

警告**:** この関数を Lisp プログラム内で使用してはならない。

30.2.4 テキスト行単位の移動

テキスト行とは改行で区切られたバッファーの範囲です。改行は前の行の一部とみなされます。最初のテキスト行はバッファー先頭で始まり、最後のテキスト行は最後の文字が改行かどうかは関係なくバッファー終端で終わります。バッファーからテキスト行への分割はそのウィンドウの幅、表示の行継続、タブやその他の制御文字の表示方法に影響されません。

`beginning-of-line` **&optional** *count* [Command]
この関数はカレント行の先頭にポイントを移動する。引数 *count*が非 `nil`または 1 以外なら前方に $count-1$ 行移動してから、その行の先頭に移動する。

この関数は別の行に移動する場合を除いてフィールド境界 (Section 32.19.9 [Fields], page 831 を参照) を超えてポイントを移動しない。したがって *count*が `nil`か 1 で、かつポイ

ントがフィールド境界で開始される場合にはポイントを移動しない。フィールド境界を無視させるには `inhibit-field-text-motion`を `t`にバインドするか、かわりに `forward-line`関数を使用する。たとえばフィールド境界を無視することを除けば、`(forward-line 0)`は`(beginning-of-line)`と同じことを行う。

この関数がバッファー (ナローイングが効力をもつ場合はアクセス可能範囲) の終端に到達したらポイントをその位置に配置する。エラーはシグナルされない。

`line-beginning-position` **&optional** *count* [Function]

`(beginning-of-line` *count*`)`が移動するであろう位置をリターンする。

`end-of-line` **&optional** *count* [Command]

この関数は、カレント行の終端にポイントを移動する。引数 *count*が非 `nil`または 1 以外なら前方に *count*−1 行移動してから、その行の終端に移動する。

この関数は別の行に移動する場合を除いてフィールド境界 (Section 32.19.9 [Fields], page 831 を参照) を超えてポイントを移動しない。したがって *count*が `nil`または 1 で、かつポイントがフィールド境界で開始される場合にはポイントを移動しない。フィールド境界を無視させるには `inhibit-field-text-motion`を `t`にバインドする。

この関数がバッファー (ナローイングが効力をもつ場合はアクセス可能範囲) の終端に到達したらポイントをその位置に配置する。エラーはシグナルされない。

`line-end-position` **&optional** *count* [Function]

`(end-of-line` *count*`)`が移動するであろう位置をリターンする。

`forward-line` **&optional** *count* [Command]

この関数は後続行へ前方に *count*行移動して、その行の先頭にポイントを移動する。*count*が負なら先行行へ後方に −*count*行移動して、その行の先頭にポイントを移動する。*count*が 0 ならカレント行の先頭にポイントを移動する。*count*が `nil`ならそれは 1 を意味する。

`forward-line`が指定された行数を移動する前にバッファー (またはアクセス可能範囲) の先頭か終端に遭遇したら、そこにポイントをセットする。エラーはシグナルされない。

`forward-line`は *count*と実際に移動した行数の差をリターンする。3 行しかないバッファーの先頭から 5 行下方への移動を試みると、ポイントは最終行の終端で停止して値は 2 となるだろう。明示的な例外としてアクセス可能な最終行が空ではなく改行がなければ (バッファーが改行で終わらない場合)、この関数はその行の終端にポイントをセットして、この関数がリターンする値はその行を移動に成功した 1 行として計数する。

インタラクティブな呼び出しでは数プレフィクス引数が *count*となる。

`count-lines` *start end* [Function]

この関数はカレントバッファー内の位置 *start*と *end*の間の行数をリターンする。*start*と *end*が等しければリターン値は 0。それ以外は、たとえ *start*と *end*が同一行にあっても最小でも 1 をリターンする。これらの間にあるテキストは、それだけを孤立して考えたると、それが空でない限りは最小でも 1 行を含まなければならないからである。

`count-words` *start end* [Command]

この関数はカレントバッファー内の位置 *start*と *end*の間にある単語の数をリターンする。

この関数はインタラクティブに呼び出すこともできる。その場合はバッファー、またはリージョンがアクティブならリージョン内の行数、単語数、文字数を報告するメッセージをプリントする。

`line-number-at-pos` **&optional** *pos absolute* [Function]
この関数はカレントバッファーのバッファー位置 *pos*に対応する行番号をリターンする。*pos* が `nil`か省略ならカレントのバッファー位置を使用する。*absolute*が `nil` (デフォルト) なら `(point-min)`から数え始めるので、値は (ナローイングされているかもしれない) バッファーのアクセス可能範囲を参照する。*absolute*が非 `nil`ならすべてのナローイングを無視して行の絶対番号をリターンする。

Section 32.1 [Near Point], page 777 の関数 `bolp`と `eolp`も参照してください。これらの関数はポイントを移動しませんが、ポイントがすでに行頭または行末にあるかどうかをテストします。

30.2.5 スクリーン行単位の移動

前のセクションの行関数は、改行文字で区切られたテキスト行だけを数えました。それらとは対照的に以下の関数はスクリーン行を数えます。スクリーン行はスクリーン上でテキストが表示される方法にしたがって定義されます。あるテキスト行 1 行が選択されたウィンドウの幅にフィット可能な程に十分短ければそれはスクリーン行で 1 行になりますが、それ以外は複数のスクリーン行になり得ます。

テキスト行が追加スクリーン行に継続されずに、そのスクリーンで切り詰められる (truncated) 場合があります。そのような場合には `vertical-motion`で `forward-line`のようにポイントを移動します。Section 39.3 [Truncation], page 982 を参照してください。

文字列が与えられると、その幅は文字の外見を制御するフラグに依存するために与えられたテキスト断片にたいして、たとえそれが選択されたウィンドウ上でさえも (幅、切り詰めの有無、ディスプレイテーブルはウィンドウごとに異なり得るので)、そのテキストがあるバッファーに応じて `vertical-motion`の挙動は異なります。Section 39.22.1 [Usual Display], page 1078 を参照してください。

以下の関数はスクリーン行のブレーク位置を判断するためにテキストをスキャンするために、スキャンする長さに比例して時間を要します。

`vertical-motion` *count* **&optional** *window cur-col* [Function]
この関数はポイントのあるスクリーン行からスクリーン行で *count*行下方に移動して、そのスクリーン行の先頭にポイントを移動する。*count*が負ならかわりに上方に移動する。

*count*引数には整数のかわりにコンスセル `(cols . lines)`を指定できる。その場合には関数はスクリーン行で *lines*行移動して、そのスクリーン行の視覚的な行頭 (visual start) から *cols* 列目にポイントを配置する。*cols*はその行の視覚的 *(visual)* な開始から数えられることに注意。そのウィンドウが水平スクロール (Section 28.23 [Horizontal Scrolling], page 680 を参照) されていれば、ポイントが配置される列は、スクロールされたテキストの列数が加えられるだろう。

リターン値はポイントが移動したスクリーン行の行数。バッファーの先頭か終端に到達していたら、この値は絶対値では *count*より小になるかもしれない。

ウィンドウ *window*は幅、水平スクロール、ディスプレイテーブルのようなパラメーターの取得に使用される。しかし `vertical-motion`は、たとえ *window*がカレントで他のバッファーを表示していたとしても、常にカレントバッファーにたいして処理を行う。

オプション引数 *cur-col*は関数呼び出し時のカレント列を指定する。これはフレームのデフォルトフェイスのフォント幅の単位で計測したウィンドウに相対的なポイントの水平座標である。これを提供することにより関数はカレント列を判断するためにバッファーを戻る必要がなくなるので、特に長い行において関数が高速化される。*cur-col*は行のビジュアル的な開始からも計測されることに注意。

`count-screen-lines` **&optional** *beg end count-final-newline window* [Function]
この関数は *beg*から *end*のテキスト内のスクリーン行の行数をリターンする。スクリーン行数は行継続やディスプレイテーブル等により実際の行数とは異なるかもしれない。*beg*と *end*が `nil`、または省略された場合のデフォルトは、そのバッファーのアクセス可能範囲の先頭と終端。

そのリージョンが改行で終わる場合には、オプションの第 3 引数 *count-final-newline*が `nil` ならそれは無視される。

オプションの第 4 引数 *window*は幅や水平スクロール等のパラメーターを取得するウィンドウを指定する。デフォルトは選択されたウィンドウのパラメーターを使用する。

`vertical-motion`と同じうように、`count-screen-lines`は *window*内にどのバッファーが表示されていようと常にカレントバッファーを使用する。これによりバッファーが何らかのウィンドウにカレントで表示されているか否かにかかわらず、任意にバッファーにたいして `count-screen-lines`の使用が可能になる。

`move-to-window-line` *count* [Command]
この関数は選択されたウィンドウ内にカレントで表示されているテキストに応じてポイントを移動する。これはウィンドウ上端からスクリーン行で *count*行目 (0 は最上行を意味する) の先頭にポイントを移動する。*count*が負ならバッファー下端 (バッファーが指定されたスクリーン位置の上で終わる場合にはバッファーの最終行) から $-$*count*行目の位置を指定する。したがって *count*が -1 ならウィンドウの最後の完全に可視なスクリーン行を指定する。

*count*が `nil`ならポイントはウィンドウ中央の行の先頭に移動する。*count*の絶対値がウィンドウサイズより大の場合には、ウィンドウが十分に高かったらそのスクリーン行は表示されていたであろう位置にポイントを移動する。これはおそらく次回の再表示の際に、その箇所がスクリーン上になるようなスクロールを発生させるだろう。

インタラクティブな呼び出しでは数プレフィクス引数が *count*となる。

リターン値はポイントが移動した先の、ウィンドウ上端行から相対的なスクリーン行番号。

`move-to-window-group-line` *count* [Function]
この関数は `move-to-window-line`と同様だが、選択されたウィンドウがウィンドウグループ ([Window Group], page 609 を参照) の一部なら、`move-to-window-group-line`は単一のウィンドウではなくグループ全体にたいする位置に移動する。この条件はバッファーローカル変数 `move-to-window-group-line-function`に関数がセットされている際に保持される。この場合には `move-to-window-group-line`は引数 *count*でその関数を呼び出して、その結果をリターンする。

`compute-motion` *from frompos to topos width offsets window* [Function]
この関数はカレントバッファーをスキャンしてスクリーン位置を計算する。これは位置 *from*がスクリーン座標 *frompos*にあると仮定して、そこから位置 *to*または座標 *topos*のいずれか先に到達したほうまでバッファーを前方にスキャンする。これはスキャン終了のバッファー位置とスクリーン座標をリターンする。

座標引数 *frompos*と *topos*は、`(hpos . vpos)`という形式のコンスセル。

引数 *width*はテキストを表示するために利用可能な列数。これは継続行の処理に影響する。`nil` はそのウィンドウ内で使用可能な実際のテキスト列数であり、`(window-width window)`がリターンする値と等しい。

引数 *offsets*は `nil`、または `(hscroll . tab-offset)`という形式のコンスセルのいずれかであること。ここで *hscroll*は左マージンのために表示されない列数であり、呼び出し側のほとんどは `window-hscroll`を呼び出すことによりこれを取得する。一方 *tab-offset*はスクリーン上

の列数とバッファー内の列数の間のオフセットである。これは継続行において前のスクリーン行の幅が`tab-width`の整数倍でないときは非 0 になる可能性がある。非継続行ではこれは常に 0。

ウィンドウ *window*の唯一の役割は使用するディスプレイテーブルの指定である。`compute-motion`は *window*内に表示されているのがどのバッファーであろうとカレントバッファーを処理する。

リターン値は 5 つの要素をもつリストである:

```
(pos hpos vpos prevhpos contin)
```

ここで *pos*はスキャンが停止したバッファー位置、*vpos*は垂直スクリーン位置、*hpos*は水平スクリーン位置である。

結果の *prevhpos*は *pos*から 1 文字戻った水平位置、*contin*は最後の行が前の文字の後 (または中) から継続されていれば`t`となる。

たとえばあるウィンドウのスクリーン行 *line*の列 *col*のバッファー位置を求めるには、そのウィンドウの display-start(表示開始) 位置を *from*、そのウィンドウの左上隅の座標を *frompos*として渡す。スキャンをそのバッファーのアクセス可能範囲の終端に制限するために、バッファーの`(point-max)`を *to*、*line*と *col*を *topos*に渡す。以下はこれを行う関数:

```
(defun coordinates-of-position (col line)
  (car (compute-motion (window-start)
                       '(0 . 0)
                       (point-max)
                       (cons col line)
                       (window-width)
                       (cons (window-hscroll) 0)
                       (selected-window))))
```

ミニバッファーにたいして`compute-motion`を使う際には、最初のスクリーン行の先頭の水平位置を取得するために`minibuffer-prompt-width`を使用する必要がある。

30.2.6 バランスのとれたカッコを越えた移動

以下はバランスの取れたカッコ式 (balanced-parenthesis。これらの式を横断して移動することと関連して Emacs では *sexp(S 式)* とも呼ばれる) と関係のあるいくつかの関数です。これらの関数がさまざまな文字を処理する方法は構文テーブル (syntax table) が制御します。Chapter 35 [Syntax Tables], page 906 を参照してください。sexp やその一部にたいする低レベルのプリミティブについては Section 35.6 [Parsing Expressions], page 914 を参照してください。ユーザーレベルのコマンドについては Section “Commands for Editing with Parentheses” in *The GNU Emacs Manual* を参照してください。

`forward-list` **&optional** *arg* [Command]
この関数はバランスの取れたカッコのグループを *arg* (デフォルトは 1) グループ前方に移動する (単語やクォート文字のペアーでクォートされた文字列は無視される)。

`backward-list` **&optional** *arg* [Command]
この関数はバランスの取れたカッコのグループを *arg* (デフォルトは 1) グループ後方に移動する (単語やクォート文字のペアーでクォートされた文字列は無視される)。

`up-list` **&optional** *arg escape-strings no-syntax-crossing* [Command]
この関数は *arg* (デフォルトは 1) の外側のカッコへ前方に移動する。負の引数では後方へ移動するが、それでもより浅いスポットへと移動する。*escape-strings*が非`nil` (インタラクティ

ブ時が該当) なら、取り囲まれた文字列の外側にも同様に移動する。*no-syntax-crossing*が非 `nil` (インタラクティブ時が該当) なら、複数の文字列を横断してリスト先頭に移動するかわりに、取り囲む文字列から脱け出すことを優先する。エラー時にはポイントの位置は未定義。

`backward-up-list` **&optional** *arg escape-strings no-syntax-crossing* [Command]
この関数は `up-list`とど同様だが引数の正負が逆。

`down-list` **&optional** *arg* [Command]
この関数はカッコを *arg* (デフォルトは 1) レベル内側、前方に移動する。負の引数では後方に移動するが、それでも深いレベル (−*arg*レベル) に移動する。

`forward-sexp` **&optional** *arg* [Command]
この関数はバランスの取れた式 (balanced expressions) を *arg* (デフォルトは 1) 前方に移動する。バランスの取れた式にはカッコ等で区切られた式、および単語や文字列定数のようなものも含まれる。Section 35.6 [Parsing Expressions], page 914 を参照のこと。たとえば、

```
---------- Buffer: foo ----------
(concat⋆ "foo " (car x) y z)
---------- Buffer: foo ----------

(forward-sexp 3)
     ⇒ nil

---------- Buffer: foo ----------
(concat "foo " (car x) y⋆ z)
---------- Buffer: foo ----------
```

`backward-sexp` **&optional** *arg* [Command]
この関数はバランスの取れた式 (balanced expressions) を、*arg* (デフォルトは 1) 後方に移動する。

`beginning-of-defun` **&optional** *arg* [Command]
この関数は後方に *arg*番目の defun の先頭に移動する。*arg*が負なら実際には前方に移動するが、defun の終端ではなく先頭に移動することは変わらない。*arg*のデフォルトは 1。

`end-of-defun` **&optional** *arg* [Command]
この関数は前方に *arg*番目の defun の終端に移動する。*arg*が負なら実際には後方に移動するが、defun の先頭ではなく終端に移動することは変わらない。*arg*のデフォルトは 1。

`defun-prompt-regexp` [User Option]
このバッファーローカル変数は非 `nil`なら defun の始まりとなる開きカッコの前に出現し得るテキストを指定する正規表現を保持する。つまりこの正規表現にたいするマッチで始まり、その後に開きカッコ構文 (open-parenthesis syntax) が続くものが defun である。

`open-paren-in-column-0-is-defun-start` [User Option]
この変数の値が非 `nil`なら列 0 にある開きカッコは defun の始まりとみなされる。`nil`なら列 0 の開きカッコは特別な意味をもたない。デフォルトは `t`。リテラル文字列の列 0 にカッコがあるような場合には偽陽性を回避するためにバックスラッシュでエスケープすること。

`beginning-of-defun-function` [Variable]
この変数は非 `nil`なら defun の開始を見つける関数を保持する。関数 `beginning-of-defun` は通常の手法を使うかわりに、この関数に自身のオプション引数を渡して呼び出す。引数が非 `nil`なら、その関数はその回数分の関数呼び出しによって `beginning-of-defun`が行うように後方に移動すること。

`end-of-defun-function` [Variable]
この変数は非 `nil`なら defun の終端を見つける関数を保持する。関数 `end-of-defun`は、通常の手法を使うかわりにその関数を呼び出す。

30.2.7 文字のスキップ

以下の 2 つの関数は指定された文字セットを超えてポイントを移動します。これらの関数は、たとえば空白文字をスキップするためによく使用されます。関連する関数については Section 35.5 [Motion and Syntax], page 913 を参照してください。

これらの関数は検索関数 (Chapter 34 [Searching and Matching], page 881 を参照) が行うように、そのバッファーがマルチバイト (multibyte) ならマルチバイトに、ユニバイト (unibyte) ならユニバイトにその文字列セットを変換します。

`skip-chars-forward` *character-set* **&optional** *limit* [Function]
この関数は与えられた文字セットをスキップしてカレントバッファー内のポイント前方に移動する。これはポイントの後の文字を調べて、その文字が *character-set*にマッチすればポイントを進める。そしてマッチしない文字に到達するまでこれを継続する。この関数は飛び超えて移動した文字数をリターンする。

引数 *character-set*が正規表現での '`[...]`'内部と同じだが、'`]`'で終端されず '`\`'が '`^`'、'`-`'、'`\`'をクォートする点が異なる。つまり`"a-zA-Z"`はすべての英字をスキップして最初の非英字の前で停止して、`"^a-zA-Z"`はすべての非英字をスキップして最初の英字の前で停止する (Section 34.3 [Regular Expressions], page 883 を参照)`"[:alnum:]"`のような文字クラスも使用できる (Section 34.3.1.2 [Char Classes], page 887 を参照)。

limit (数字かマーカー) が与えられたら、それはポイントがスキップして到達できるそのバッファー内の最大位置を指定する。ポイントは *limit*、または *limit*の前でストップするだろう。

以下の例ではポイントは最初 '`T`'の直前に置かれている。フォーム評価後にポイントはその行の末尾 ('`hat`'の '`t`'と改行の間) に配置される。この関数はすべての英字とスペースをスキップするが改行はスキップしない。

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(skip-chars-forward "a-zA-Z ")
     ⇒ 18

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

`skip-chars-backward` *character-set* **&optional** *limit* [Function]
この関数は *limit*に至るまで *character-set*にマッチする文字をスキップしてポイントを後方に移動する。これは `skip-chars-forward`と同様だがポイントを移動する方向が異なる。
リターン値は移動した距離を示す。これは 0 以上の整数。

30.3 エクスカーション

プログラム中の限定された部分でポイントを一時的に移動するのが便利なことが時折あります。これはエクスカーション *(excursion:* 遠足、小旅行*)* と呼ばれるもので、スペシャルフォーム `save-excursion` により行われます。この構文は初期のカレントバッファー自体とポイントの値を記憶して、そのエクスカーション完了時にそれらをリストアします。これはプログラムのある部分においてプログラムの他の部分に影響を与えることなくポイントを移動する標準的な手段であり、Emacs の Lisp ソース内では何度も使用されています。

カレントバッファー自体のみの保存やリストアが必要なら、かわりに `save-current-buffer`や `with-current-buffer`を使用してください (Section 27.2 [Current Buffer], page 590 を参照)。ウィンドウ構成の保存やリストアが必要なら、Section 28.26 [Window Configurations], page 686 と Section 29.13 [Frame Configurations], page 739 で説明されているフォームを参照してください。

`save-excursion` *body...* [Special Form]
このスペシャルフォームはカレントバッファー自体とポイント値を保存、*body*を評価してから最後にバッファーと保存したポイントとマークの値をリストアする。`throw`やエラーを通じたアブノーマル exit(Section 11.7 [Nonlocal Exits], page 162 を参照) の場合にも、保存されたいずれの値もリストアされる。
`save-excursion`がリターンする値は *body*内の最後のフォームの結果、または *body*フォームが与えられなければ `nil`をリターンする。

`save-excursion`はエクスカーション開始時にカレントだったバッファーのポイントだけを保存するために、そのエクスカーション中に変更された他のバッファーのポインはその後も効果が残るでしょう。これはしばしば予期せぬ結果を招くので、エクスカーション中に `set-buffer`を呼び出すとバイトコンパイラーは警告を発します:

```
Warning: Use `with-current-buffer' rather than
         save-excursion+set-buffer
```

このような問題を回避するためには、以下の例のように望むカレントバッファーをセット後にのみ `save-excursion`を呼び出すべきです:

```
(defun append-string-to-buffer (string buffer)
  "BUFFER 末尾に STRING を追加"
  (with-current-buffer buffer
    (save-excursion
      (goto-char (point-max))
      (insert string))))
```

同様に `save-excursion`は `switch-to-buffer`のような関数が変更したウィンドウ/バッファーの対応をリストアしません。

警告**:** 保存されたポイント値に隣接する通常のテキスト挿入は、それがすべてのマーカーを再配置するのと同じように、保存されたポイントカーを再配置します。より正確には保存される値は挿入タイプ `nil`のマーカーです。Section 31.5 [Marker Insertion Types], page 771 を参照してください。したがって保存されたポイント値は、リストア時には通常は挿入されたテキストの直前になります。

`save-mark-and-excursion` *body...* [Macro]
このマクロは `save-excursion`と同様だが、マークの位置と `mark-active`の保存とリストアも行う点が異なる。このマクロは Emacs のバージョン 25.1 以前に `save-excursion`が行っていたことを行う。

30.4 ナローイング

ナローイング *(narrowing)* とは Emacs 編集コマンドがアドレス指定可能なテキストを、あるバッファー内の制限された文字範囲に限定することを意味します。アドレス可能なテキストは、そのバッファーのアクセス可能範囲 *(accessible portion)* と呼ばれます。

ナローイングは 2 つのバッファー位置により指定されるもので、それらの位置がアクセス可能範囲の開始と終了になります。ほとんどの編集コマンドやプリミティブにたいして、これらの位置はそれぞれそのバッファーの先頭と終端に置き換えられます。ナローイングが効果をもつ間にはアクセス可能範囲外のテキストは表示されず、その外部にポイントを移動することはできません。ナローイングは実際のバッファー位置 (Section 30.1 [Point], page 755 を参照) を変更しないことに注意してください。ほとんどの関数はアクセス可能範囲外のテキストにたいする操作を受け入れません。

バッファーを保存するコマンドはナローイングの影響を受けません。どんなナローイングであろうと、それらはバッファー全体を保存します。

単一バッファー内にタイプが大きく異なるテキストを複数表示する必要がある場合には、Section 27.12 [Swapping Text], page 606 で説明する代替機能の使用を考慮してください。

`narrow-to-region` *start end* [Command]
この関数はアクセス可能範囲の開始と終了にカレントバッファーの *start*と *end*をセットする。どちらの引数も文字位置で指定すること。

インタラクティブな呼び出しでは、*start*と *end*はカレントリージョンにセットされる (ポイントとマークで小さいほうが前者)。

`narrow-to-page` **&optional** *move-count* [Command]
この関数はカレントページだけを含むようにカレントバッファーのアクセス可能範囲をセットする。1 つ目のオプション引数 *move-count*が非 `nil`なら、*move-count*で前方か後方へ移動後に 1 ページにナローすることを意味する。変数 `page-delimiter`はページの開始と終了の位置を指定する (Section 34.8 [Standard Regexps], page 905 を参照)。

インタラクティブな呼び出しでは *move-count*には数プレフィクス引数がセットされる。

`widen` [Command]
この関数はカレントバッファーにたいするすべてのナローイングをキャンセルする。これはワイドニング *(widening)* と呼ばれる。これは以下の式と等価:

```
(narrow-to-region 1 (1+ (buffer-size)))
```

`buffer-narrowed-p` [Function]
この関数はそのバッファーがナローされていれば非 `nil`、それ以外は `nil`をリターンする。

`save-restriction` *body...* [Special Form]
このスペシャルフォームはアクセス可能範囲のカレントのバインドを保存して *body*を評価、その後に以前有効だったナローイング (またはナローイングがない状態) と同じ状態になるように、最後に保存された境界をリストアする。ナローイングの状態は、`throw`やエラーを通じたアブノーマル exit(Section 11.7 [Nonlocal Exits], page 162 を参照) イベント内においてもリストアされる。したがってこの構文は一時的にバッファーをナローする明快な手段である。

save-restrictionがリターンする値は *body*内の最後のフォームのリターン値、*body*フォームが与えられなければ nil。

注意: save-restriction使用の際は間違いを起こしやすい。これを試みる前にこの説明全体に目を通すこと。

*body*がカレントバッファーを変更する場合でも save-restrictionは依然として元のバッファー (その制限が保存されたバッファー) 上の制限をリストアするが、カレントバッファー自体はリストアしない。

save-restrictionはポイントをリストアしない。これを行うには save-excursionを使用する。save-restrictionと save-excursionの両方を共に使用するなら、始め (外側) に save-excursionを記述すること。それ以外では一時的なナローイング影響下で古いポイント値がリストアされる。古いポイント値が一時的なナローイング境界外なら、それを実際にリストアすることは失敗するだろう。

以下は save-restrictionの正しい使い方の簡単な例:

```
---------- Buffer: foo ----------
This is the contents of foo
This is the contents of foo
This is the contents of foo*
---------- Buffer: foo ----------

(save-excursion
  (save-restriction
    (goto-char 1)
    (forward-line 2)
    (narrow-to-region 1 (point))
    (goto-char (point-min))
    (replace-string "foo" "bar")))

---------- Buffer: foo ----------
This is the contents of bar
This is the contents of bar
This is the contents of foo*
---------- Buffer: foo ----------
```

31 マーカー

マーカー *(marker)* とは、あるバッファー内で取り囲んでいるテキストにたいして相対的な位置を指定するために使用されるオブジェクトです。テキストが挿入や削除される際には、常にマーカーは自動的にそのバッファーの先頭からのオフセットを自動的に変更して自身の左右にある文字の間に留まります。

31.1 マーカーの概要

マーカーはバッファーとそのバッファー内の位置を指定します。マーカーは位置を要求する関数内において、整数と同じように位置を表すために使用することができます。その場合には、そのマーカーのバッファーは通常は無視されます。この方法で使用されるマーカーは、通常はその関数が処理するバッファー内の位置を指しますが、それは完全にプログラマーの責任です。位置についての完全な説明は Chapter 30 [Positions], page 755 を参照してください。

マーカーはマーカー位置 (marker position)、マーカーバッファー (marker buffer)、挿入タイプ (insertion type) という 3 つの属性をもちます。マーカー位置はそのバッファー内の位置としてのマーカーと (その時点において) 等しい整数です。しかしマーカー位置はマーカーの生存期間中に変化し得るものであり頻繁に変更されます。バッファー内でのテキストの挿入や削除によってマーカーは再配置されます。マーカー前後の 2 文字以外の場所で挿入や削除がおこなわれても、マーカー位置はその 2 文字間に留まるというのがこのアイデアです。再配置によってマーカーと等価な整数は変更されます。

マーカー位置周辺のテキストを削除することにより、そのマーカーは削除されたテキストの直前と直後にある文字の間に残されます。マーカー位置へのテキスト挿入では、マーカーは通常は新たなテキストの前か後のいずれかに配置されます。その挿入が `insert-before-markers` (Section 32.4 [Insertion], page 781 を参照) で行われたものでなければ、どちらに配置されるかはマーカーの挿入タイプ (Section 31.5 [Marker Insertion Types], page 771 を参照) に依存します。

バッファーでの挿入と削除では、すべてのマーカーをチェックして必要ならそれらを再配置しなければなりません。これは多数のマーカーをもつバッファーでの処理を低速にします。この理由によりそれ以上マーカーが不必要なことが確信できるなら、存在しない場所を指さないようにマーカーを設定することはよいアイデアといえるでしょう。それ以上アクセスされる可能性がないマーカーは最終的には削除されます (Section E.3 [Garbage Collection], page 1178 を参照)。

マーカー位置にたいして算術演算を行うことは一般的なので、それらの演算子のほとんど (+や-を含む) が引数としてマーカーに渡すことができます。そのような場合でのマーカーはカレント位置を意味します。

以下ではマーカーの作成とセットを行ってポイントをマーカーに移動しています:

```
;; 最初はどこも指さない新たなマーカーを作成:
(setq m1 (make-marker))
     ⇒ #<marker in no buffer>

;; カレントバッファーの 99 と 100 番目の
;;    文字間を指すよう m1をセット:
(set-marker m1 100)
     ⇒ #<marker at 100 in markers-ja.texi>
```

```
;; ここでバッファー先頭に 1 文字挿入:
(goto-char (point-min))
     ⇒ 1
(insert "Q")
     ⇒ nil

;; m1は適切に更新された
m1
     ⇒ #<marker at 101 in markers-ja.texi>

;; 同じ位置を指す 2 つのマーカーは
;;   equalだが eqに非ず
(setq m2 (copy-marker m1))
     ⇒ #<marker at 101 in markers-ja.texi>
(eq m1 m2)
     ⇒ nil
(equal m1 m2)
     ⇒ t

;; マーカー使用終了時、存在しない場所を指すようセット
(set-marker m1 nil)
     ⇒ #<marker in no buffer>
```

31.2 マーカーのための述語

あるオブジェクトがマーカーなのか、それとも整数かマーカーのいずれかであるかを確認するためのテストを行うことができます。後者のテストはマーカーと整数の両方にたいして機能する算術関数において有用です。

`markerp` *object* [Function]
この関数は *object*がマーカーなら `nil`、それ以外は `t`をリターンする。多くの関数はマーカーか整数のいずれかを受け入れるだろうが、整数はマーカーとは異なることに注意。

`integer-or-marker-p` *object* [Function]
この関数は *object*が整数かマーカーなら `t`、それ以外は `nil`をリターンする。

`number-or-marker-p` *object* [Function]
この関数は *object*が数値 (整数か浮動小数点数) またはマーカーなら `t`、それ以外は `nil`をリターンする。

31.3 マーカーを作成する関数

マーカーを新たに作成する際には存在しない場所、ポイントの現在位置、バッファーのアクセス可能範囲の先頭や終端、または別の与えられたマーカーと同じ箇所を指すようにすることができます。

以下の 4 つの関数はすべて挿入タイプ `nil`のマーカーをリターンします。Section 31.5 [Marker Insertion Types], page 771 を参照してください。

`make-marker` [Function]
この関数はどこも指さないマーカーを新たに作成してリターンする。

```
(make-marker)
     ⇒ #<marker in no buffer>
```

`point-marker` [Function]
この関数はカレントバッファーのポイント現在位置を指すマーカーを新たに作成してリターンする。Section 30.1 [Point], page 755 を参照のこと。例は以下の `copy-marker`を参照のこと。

`point-min-marker` [Function]
この関数はバッファーのアクセス可能範囲の先頭を指すマーカーを新たに作成してリターンする。ナローイングが効力をもたなければ、これはバッファーの先頭になるだろう。Section 30.4 [Narrowing], page 766 を参照のこと。

`point-max-marker` [Function]
この関数はバッファーのアクセス可能範囲の終端を指すマーカーを新たに作成してリターンする。ナローイングが効力をもたなければ、これはバッファーの終端になるだろう。Section 30.4 [Narrowing], page 766 を参照のこと。

以下はこのチャプターのテキストのソースファイルのバージョンを含むバッファーにたいして、この関数と `point-min-marker`を使用する例。

```
(point-min-marker)
     ⇒ #<marker at 1 in markers-ja.texi>
(point-max-marker)
     ⇒ #<marker at 24080 in markers-ja.texi>

(narrow-to-region 100 200)
     ⇒ nil
(point-min-marker)
     ⇒ #<marker at 100 in markers-ja.texi>
(point-max-marker)
     ⇒ #<marker at 200 in markers-ja.texi>
```

`copy-marker` **&optional** *marker-or-integer insertion-type* [Function]
引数としてマーカーを渡されると、`copy-marker`は *marker-or-integer*が行うように同じバッファーの同じ位置を指すマーカーを新たに作成してリターンする。整数を渡されると、`copy-marker`はカレントバッファーの位置 *marker-or-integer*を指すマーカーを新たに作成してリターンする。

新たなマーカーの挿入タイプは引数 *insertion-type*により指定される。Section 31.5 [Marker Insertion Types], page 771 を参照のこと。

```
(copy-marker 0)
     ⇒ #<marker at 1 in markers-ja.texi>

(copy-marker 90000)
     ⇒ #<marker at 24080 in markers-ja.texi>
```

*marker*がマーカーと整数のいずれでもなければエラーがシグナルされる。

2 つのマーカーはそれらが同じバッファーの同じ位置、またはどちらも存在しない場所を指す場合には、(`eq`ではないが)`equal`とみなされます。

```
(setq p (point-marker))
     ⇒ #<marker at 2139 in markers-ja.texi>
```

```
(setq q (copy-marker p))
     ⇒ #<marker at 2139 in markers-ja.texi>

(eq p q)
     ⇒ nil

(equal p q)
     ⇒ t
```

31.4 マーカーからの情報

このセクションではマーカーオブジェクトの構成要素にアクセスする関数を説明します。

`marker-position` *marker* [Function]
この関数は *marker*が指す位置、存在しない場所なら `nil`をリターンする。

`marker-buffer` *marker* [Function]
この関数は *marker*がその内部を指すバッファー、存在しない場所を指す場合には `nil`をリターンする。

```
(setq m (make-marker))
     ⇒ #<marker in no buffer>
(marker-position m)
     ⇒ nil
(marker-buffer m)
     ⇒ nil

(set-marker m 3770 (current-buffer))
     ⇒ #<marker at 3770 in markers-ja.texi>
(marker-buffer m)
     ⇒ #<buffer markers-ja.texi>
(marker-position m)
     ⇒ 3770
```

31.5 マーカーの挿入タイプ

マーカーが指す位置に直接テキストを挿入する際には、そのマーカーを再配置するために利用可能な手段が 2 つあります。そのマーカーは挿入されたテキストの前か後を指すことができます。マーカーの挿入タイプ *(insertion type)* を指定することにより、マーカーがどちらを行うか指定できます。`insert-before-markers`を使用する場合には、マーカーの挿入タイプを無視して常にマーカーが挿入されたテキストの後を指すよう再配置されることに注意してください。

`set-marker-insertion-type` *marker type* [Function]
この関数はマーカー *marker*の挿入タイプを *type*にセットする。*type*が `t`なら、テキスト挿入時に *marker*はその位置まで進められるだろう。*type*が `nil`なら、テキスト挿入時に *marker*はそこまで進められることはない。

`marker-insertion-type` *marker* [Function]
この関数は *marker*のカレント挿入タイプを報告する。

挿入タイプを指定するための引数を受け取らずにマーカーを作成するすべての関数は、挿入タイプ`nil`でマーカーを作成します。マークもデフォルトでは挿入タイプ`nil`をもちます。

31.6 マーカー位置の移動

このセクションでは既存マーカーの位置を変更する方法について説明します。これを行う際にはそのマーカーがあなたのプログラム外部に使用されているかどうか、もし使用されているならマーカーを移動した結果どのような影響が生じるかを確実に理解する必要があります。さもないと Emacs の他の部分で混乱した出来事が発生するかもしれません。

`set-marker` *marker position* **&optional** *buffer* [Function]

この関数は *buffer*内で *marker*を *position*に移動する。*buffer*が与えられなかった場合のデフォルトはカレントバッファー。

*position*が`nil`、または存在しない場所を指すマーカーなら、*marker*は存在しない場所を指すようにセットされる。

リターン値は *marker*。

```
(setq m (point-marker))
     ⇒ #<marker at 4714 in markers-ja.texi>
(set-marker m 55)
     ⇒ #<marker at 55 in markers-ja.texi>
(setq b (get-buffer "foo"))
     ⇒ #<buffer foo>
(set-marker m 0 b)
     ⇒ #<marker at 1 in foo>
```

`move-marker` *marker position* **&optional** *buffer* [Function]

これは`set-marker`の別名。

31.7 マーク

バッファーはそれぞれマーク *(mark)* というバッファー専用の特別なマーカーをもちます。バッファーが新たに作成される際には、このマーカーはすでに存在していますがどこも指していません。これはそのバッファーにはまだマークが存在しないことを意味します。それ以降のコマンドがマークをセットできます。

マークは`kill-region`や`indent-rigidly`のような多くのコマンドにたいしてテキスト範囲をバインドするための位置を指定します。これらのコマンドは、通常はポイントとマークの間のリージョン *(region)* と呼ばれるテキストに作用します。リージョンを操作するコマンドを記述する場合にはマークを直接調べず、かわりに‘`r`’指定とともに`interactive`を使用してください。このようにすればインタラクティブな呼び出しではコマンドの引数としてポイントとマークの値が提供され、かつ他の Lisp プログラムは引数を明示的に指定できます。Section 21.2.2 [Interactive Codes], page 378 を参照してください。

いくつかのコマンドは副作用 (side-effect) としてマークをセットします。コマンドはユーザーがそれを使用する可能性がある場合のみマークをセットするべきであって、決してコマンドの内部的な目的にたいして使用してはなりません。たとえば`replace-regexp`コマンドは何らかの置換を行う前にマークにポイントの値をセットしますが、その理由はこれによりユーザーが置換を終えた後に簡単にその位置に戻ることが可能になるからです。

一度バッファー内にマークが存在すれば、その存在は通常は決して消えることはありません。しかし Transient Mark モードが有効だとマークが非アクティブ *(inactive)* になることはあります。バッファーローカル変数 `mark-active`が非 `nil`なら、それはマークがアクティブであることを意味します。コマンドはマークを直接非アクティブにするために関数 `deactivate-mark`を呼び出すことができ、変数 `deactivate-mark`を非 `nil`値にセットすることにより、エディターコマンドループ (editor command loop) にリターン時にマークの非アクティブ化を要求できます。

Transient Mark モードが有効だと、通常ならポイント近傍に適用される特定の編集コマンドはマークがアクティブなときはかわりにリージョンに適用されます。これが Transient Mark モードを使用する主な動機です (他にもマークアクティブ時にはリージョンのハイライトが有効になるという理由もある。Chapter 39 [Display], page 981 を参照)。

マークに加えてバッファーはそれぞれマークリング *(mark ring)* をもっています。これは以前のマーク値を含むマーカーのリストです。編集コマンドがマークを変更する際には、それらのコマンドは通常はマークの旧値をマークリングに保存するべきです。変数 `mark-ring-max`はマークリング内のエントリー最大数を指定します。リストがこの長さに達すると最後の要素を削除して新たな要素が追加されます。

これとは別にグローバルマークリング (global mark ring) がありますが、それは少数の特定のユーザーレベルコマンドでのみ使用されて、Lisp プログラムとは関連しないのでここでは説明しません。

`mark` **&optional** *force* [Function]

この関数はカレントバッファーのマーク位置を整数でリターンする。そのバッファー内でそれまでマークがセットされていなければ `nil`をリターンする。

Transient Mark モードが有効、かつ `mark-even-if-inactive`が `nil`の場合、マークが非アクティブなら `mark`はエラーをシグナルする。しかし、*force*が非 `nil`なら、`mark`はマークの非アクティブ性を無視して、何にせよマーク位置 (か `nil`) をリターンする。

`mark-marker` [Function]

この関数はカレントバッファーのマークを表すマーカーをリターンする。これはコピーではなく内部的に使用されるマーカー。したがってこのマーカー位置にたいする変更は、そのバッファーのマークに直接影響する。それが望む効果でなければこれを行ってはならない。

```
(setq m (mark-marker))
     ⇒ #<marker at 3420 in markers-ja.texi>
(set-marker m 100)
     ⇒ #<marker at 100 in markers-ja.texi>
(mark-marker)
     ⇒ #<marker at 100 in markers-ja.texi>
```

他のマーカー同じように、このマーカーを任意のバッファー位置にセットできる。このマーカーにたいして、これがマークする以外のバッファーを指すようにすると、完全に整合性があるものの、いささか奇妙な結果を得ることになるだろう。わたしたちはこれを行わないことを推奨する!

`set-mark` *position* [Function]

この関数はマークを *position*にセットして、そのマークをアクティブにする。マークの旧値はマークリングに *push* されない。

注意**:** マークが移動したことをユーザーに確認させて、かつ前のマーク位置が失われることを望む場合のみこの関数を使用すること。通常はマークセット時に古いマークを `mark-ring`に push

すること。この理由により、ほとんどのアプリケーションは `set-mark`ではなく、`push-mark` と `pop-mark`を使用するべきである。

Emacs Lisp 初心者のプログラマーは誤った用途にマークの使用を試みがちである。ユーザーの利便のために位置を保存するのがマークである。編集コマンドはマーク変更がコマンドのユーザーレベル機能の一部でない限りマークを変更しないこと (そのような場合にはその効果をドキュメントするべきである)。Lisp プログラムの内部的な使用のために位置を記憶するためには、マークを Lisp 変数に格納すること。たとえば:

```
(let ((beg (point)))
  (forward-line 1)
  (delete-region beg (point)))
```

`push-mark` **&optional** *position nomsg activate* [Function]

この関数はカレントバッファーのマークを *position*にセットして、前のマークを `mark-ring` に push する。*position*が `nil`ならポイントの値を使用する。

関数`push-mark`は通常はマークをアクティブにしない。アクティブにする場合には引数 *activate* に `t`を指定する。

*nomsg*が `nil`ならメッセージ '`Mark set`'が表示される。

`pop-mark` [Function]

この関数は `mark-ring`のトップ要素を pop して、そのマークをバッファーの実際のマークにする。これはバッファー内のポイントを移動せず、`mark-ring`が空なら何も行わない。これはマークを非アクティブ化する。

`transient-mark-mode` [User Option]

この変数が非 `nil`なら Transient Mark モードを有効にする。Transient Mark モードでは、すべてのバッファー変更プリミティブが `deactivate-mark`をセットする。結果としてバッファーを変更するほとんどのコマンドもマークを非アクティブにする。

Transient Mark モードが有効かつマークがアクティブなら、通常はポイント近傍に適用されるコマンドの多くは、かわりにリージョンに適用される。そのようなコマンド、リージョンを処理すべきかどうかをテストするために、関数 `use-region-p`を使用すること。Section 31.8 [The Region], page 776 を参照のこと。

Lisp プログラムは一時的に Transient Mark モードを有効にするために、`transient-mark-mode`を `nil`でも `t`でもない値にセットできる。値が `lambda`なら、通常ならマークを非アクティブ化するバッファー変更ような操作の後に、Transient Mark モードを自動的にオフに切り替える。値が `(only . oldval)` なら後続のコマンドがポイントを移動かつシフト変換 (Section 21.8.1 [Key Sequence Input], page 404 を参照) されていない場合、あるいは通常はマークを非アクティブにするその他の操作の場合にに、`transient-mark-mode`に値 *oldval*をセットする。

`mark-even-if-inactive` [User Option]

これが非 `nil`な Lisp プログラムおよび Emacs ユーザーは、たとえ非アクティブでもマークを使用できる。このオプションは Transient Mark モードの動作に影響を及ぼす。このオプションが非 `nil`ならマークの非アクティブ化によりリージョンのハイライトはオフに切り替えられるが、マークを使用するコマンドは、あたかもマークがアクティブであるかのように振る舞う。

`deactivate-mark` [Variable]

エディターコマンドがこの変数を非 `nil`にセットすると、エディターコマンドループはコマンドのリターン後に、(Transient Mark モードが有効なら) マークを非アクティブにする。バッ

ファーを変更するすべてのプリミティブは、コマンド終了時にマークを非アクティブにするために deactivate-markをセットする。この変数はセットすらことによりバッファーローカルになる。

コマンド終了時にマークを非アクティブにすることなくバッファーを変更する Lisp コードを記述するためには、変更を行うコードの周辺で deactivate-markを nilにバインドすること。たとえば:

```
(let (deactivate-mark)
  (insert " "))
```

deactivate-mark **&optional** *force* [Function]
Transient Mark モードが有効、または *force*が非 nilなら、この関数はマークを非アクティブにしてノーマルフック deactivate-mark-hookを実行して、それ以外は何も行わない。

mark-active [Variable]
この変数が非 nilならマークはアクティブ。この変数はそれぞれのバッファーにたいして常にローカル。通常はポイント近傍を操作するコマンドが、かわりにリージョンを操作すべきかどうかを判断するためにこの変数の値を使用してはならない。その目的にたいしては関数 use-region-pを使用すること (Section 31.8 [The Region], page 776 を参照)。

activate-mark-hook [Variable]
deactivate-mark-hook [Variable]
これらのノーマルフックはマークがアクティブや非アクティブになった際に順次実行される。マークがアクティブかつリージョンが変更された可能性があるなら、コマンドループの最後にフック activate-mark-hookも実行される。

handle-shift-selection [Function]
この関数はポイント移動コマンドのシフト選択 (shift-selection) の動作を実装する。Section “Shift Selection” in *The GNU Emacs Manual* を参照のこと。これは interactive指定に文字 ‘^’を含むコマンドの呼び出し時は常に、そのコマンド自身を実行する前に Emacs コマンドループにより自動的に呼び出される (Section 21.2.2 [Interactive Codes], page 378 を参照)。

shift-select-modeが非 nil、かつカレントコマンドがシフト変換 (Section 21.8.1 [Key Sequence Input], page 404 を参照) を通じて呼び出された場合には、この関数はマークをセットして一時的にリージョンをアクティブにする (すでにこの方法によりリージョンが一時的にアクティブにされている場合を除く)。それ以外ではリージョンが一時的にアクティブにされていればマークを非アクティブにして、変数 transient-mark-modeに前の値をリストアする。

mark-ring [Variable]
このバッファーローカル変数の値は、もっとも最近のものが先頭となるような、以前に保存されたカレントバッファーのマークのリスト。

```
mark-ring
⇒ (#<marker at 11050 in markers-ja.texi>
    #<marker at 10832 in markers-ja.texi>
    ...)
```

mark-ring-max [User Option]
この変数の値は mark-ringの最大サイズ。これより多くのマークが mark-ringに push されると、push-mark新たなマーク追加時には古いマークを破棄する。

Delete Selection モード (Section “Using Region” in *The GNU Emacs Manual* を参照) が有効な際には、アクティブリージョン (いわゆる “選択”) を操作するコマンドは若干異なる振る舞いをします。これは `pre-command-hook`に関数 `delete-selection-pre-hook`を追加することにより機能します (Section 21.1 [Command Overview], page 375 を参照)。この関数はそのコマンドにたいして適切なように選択を削除するために `delete-selection-helper`を呼び出します。コマンドを Delete Selection モードに適応させたければ、その関数シンボルの `delete-selection`プロパティに put してください (Section 9.4.1 [Symbol Plists], page 129 を参照)。シンボルにこのプロパティをもたないコマンドは選択を削除しません。そのコマンドに期待される挙動を調整するために、このプロパティはいくつかの値のいずれかをもつことができます。詳細は `delete-selection-pre-hook`と `delete-selection-helper`のドキュメント文字列を参照してください。

31.8 リージョン

ポイントとマークの間のテキストは、リージョン *(region)* という名で知られています。さまざまな関数がポイントとマークで区切られたテキストを操作しますが、ここではリージョンそのものに特に関連する関数だけを説明します。

以下の 2 つの関数はマークが何処も指していなければエラーをシグナルします。Transient Mark モードが有効、かつ `mark-even-if-inactive`が `nil`な、マークが非アクティブな場合にエラーをシグナルします。

`region-beginning` [Function]
: この関数はリージョンの先頭位置を、(整数として) リターンする。これはポイントかマークのいずれか小さいほうの位置。

`region-end` [Function]
: この関数はリージョンの終端位置を、(整数として) リターンする。これはポイントかマークのいずれか大きいほうの位置。

リージョンにたいして操作を行うようにデザインされたコマンドがリージョンの先頭と終端を探すためには、`region-beginning`や `region-end`を使用するかわりに、通常は ‘`r`’指定とともに `interactive`を使用するべきです。これにより他の Lisp プログラムが引数として明示的にリージョンの境界を指定できるようになります。Section 21.2.2 [Interactive Codes], page 378 を参照してください。。

`use-region-p` [Function]
: この関数は Transient Mark モードが有効でマークがアクティブであり、かつバッファー内に有効なリージョンがあれば `t`をリターンする。この関数はマークアクティブ時にはポイント近傍のテキストのかわりにリージョンを操作するコマンドにより使用されることを意図している。

  リージョンはそれが非 0 のサイズをもつか、あるいはユーザーオプション `use-empty-active-region`が非 `nil` (デフォルトは `nil`) なら有効。関数 `region-active-p`は `use-region-p`と同様だが、すべてのリージョンを有効とみなす。リージョンが空ならポイントにたいして操作を行うほうが適切な場合が多いために、ほとんどの場合は `region-active-p`を使用するべきではない。

32 テキスト

このチャプターではバッファー内のテキストを扱う関数を説明します。ほとんどはカレントバッファー内のテキストにたいして検査、挿入、削除を行ってポイント位置やポイントに隣接するテキストを操作することが多々あります。その多くはインタラクティブ (interactive: 対話的) です。テキストを変更するすべての関数は、その変更にたいする undo(アンドゥ、取り消し) を提供します (Section 32.9 [Undo], page 793 を参照)。

テキストに関連する関数の多くが、*start*と *end*という名前の引数として渡された 2 つのバッファー位置により定義されるテキストのリージョンを操作します。これらの引数はマーカー (Chapter 31 [Markers], page 768 を参照) か数値的な文字位置 (Chapter 30 [Positions], page 755 を参照) のいずれかであるべきです。これらの引数の順序は関係ありません。*start*がリージョンの終端で *end*がリージョンの先頭であっても問題はありません。たとえば `(delete-region 1 10)`と `(delete-region 10 1)`は等価です。*start*と *end*のいずれかがバッファーのアクセス可能範囲の外部なら `args-out-of-range`エラーがシグナルされます。インタラクティブな呼び出しでは、これらの引数にポイントとマークが使用されます。

このチャプターを通じて、“テキスト (text)” とは (関係あるときは) そのプロパティも含めたバッファー内の文字を意味します。ポイントは常に 2 つの文字の間にあり、カーソルはポイントの後の文字上に表示されることを覚えておいてください。

32.1 ポイント近傍のテキストを調べる

ポイント付近にある文字を調べるための関数が数多く提供されています。簡単な関数のいくつかはここで説明します。Section 34.4 [Regexp Search], page 894 の `looking-at`も参照してください。

以下の 4 つの関数でのバッファーの “先頭 (beginning)” と “終端 (end)” はそれぞれ、アクセス可能範囲の先頭と終端を意味します。

`char-after` **&optional** *position* [Function]

この関数はカレントバッファーの位置 *position* (つまり直後) の文字をリターンする。*position* がこの目的にたいする範囲の外にある場合、すなわちバッファーの先頭より前、またはバッファーの終端以降にあるなら値は `nil`。*position*のデフォルトはポイント。

以下の例ではバッファーの最初の文字が ‘`@`’であると仮定する:

```
(string (char-after 1))
     ⇒ "@"
```

`char-before` **&optional** *position* [Function]

この関数はカレントバッファーの位置 *position*の直前の文字をリターンする。*position*がこの目的にたいする範囲の外にある場合、すなわちバッファーの先頭より前、またはバッファーの終端より後にあるなら値は `nil`。*position*のデフォルトはポイント。

`following-char` [Function]

この関数はカレントバッファーのポイントの後にある文字をリターンする。これは `(char-after (point))`と同様。ただしポイントがバッファー終端にある場合には、`following-char`は 0 をリターンする。

ポイントが常に 2 つの文字の間にあり、カーソルは通常はポイント後の文字上に表示されることを思い出してほしい。したがって `following-char`がリターンする文字はカーソル上の文字となる。

以下の例では‘a’と‘c’の間にポイントがある。

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea⋆ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(string (preceding-char))
     ⇒ "a"
(string (following-char))
     ⇒ "c"
```

`preceding-char` [Function]

この関数はカレントバッファーのポイントの前の文字をリターンする。上記`following-char`の下の例を参照のこと。ポイントがバッファー先頭にあれば、`preceding-char`は 0 をリターンする。

`bobp` [Function]

この関数はポイントがバッファー先頭にあれば`t`をリターンする。ナローイングが効力をもつなら、これはテキストのアクセス可能範囲の先頭を意味する。Section 30.1 [Point], page 755 の`point-min`も参照のこと。

`eobp` [Function]

この関数はポイントがバッファー終端にあれば`t`をリターンする。ナローイングが効力をもつなら、これはテキストのアクセス可能範囲の終端を意味する。Section 30.1 [Point], page 755 の`point-max`も参照のこと。

`bolp` [Function]

この関数はポイントが行の先頭にあれば`t`をリターンする。Section 30.2.4 [Text Lines], page 758 を参照のこと。バッファー (またはアクセス可能範囲) の先頭は、常に行の先頭とみなされる。

`eolp` [Function]

この関数はポイントが行の終端にあれば`t`をリターンする。Section 30.2.4 [Text Lines], page 758 を参照のこと。バッファー (またはアクセス可能範囲) の終端は常に行の先頭とみなされる。

32.2 バッファーのコンテンツを調べる

このセクションでは Lisp プログラムがバッファー内の任意の範囲にあるテキストを文字列に変換するための関数を説明します。

`buffer-substring` *start end* [Function]

この関数はカレントバッファー内の位置 *start*と *end*で定義されるリージョンのテキストのコピーを含む文字列をリターンする。引数がバッファーのアクセス可能範囲内の位置でなければ、`buffer-substring`は`args-out-of-range`エラーをリターンする。

以下の例では Font-Lock モードが有効でないものとする:

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------
```

```
(buffer-substring 1 10)
     ⇒ "This is t"
(buffer-substring (point-max) 10)
     ⇒ "he contents of buffer foo\n"
```

コピーされるテキストが何らかのテキストプロパティをもっていたら、それらのプロパティが属する文字とともに文字列にコピーされる。しかしバッファー内のオーバーレイ (Section 39.9 [Overlays], page 999 を参照)、およびそれらのプロパティは無視されるためコピーされない。

たとえば Font-Lock モードが有効なら以下のような結果を得るだろう:

```
(buffer-substring 1 10)
     ⇒ #("This is t" 0 1 (fontified t) 1 9 (fontified t))
```

`buffer-substring-no-properties` *start end* [Function]
これは `buffer-substring`と同様だが、テキストプロパティはコピーせずに文字自体だけをコピーする点が異なる。Section 32.19 [Text Properties], page 813 を参照のこと。

`buffer-string` [Function]
この関数はカレントバッファーのアクセス可能範囲全体のコンテンツを文字列としてリターンする。

異なる場所からのコピー時に双方向テキストの再配置によって結果の文字列の視覚的外見が変更されないように保証する必要があるなら、`buffer-substring-with-bidi-context`関数を使用すること (Section 39.26 [Bidirectional Display], page 1085 を参照)。

`filter-buffer-substring` *start end* **&optional** *delete* [Function]
この関数は変数 `filter-buffer-substring-function`により指定された関数を使用して、*start*と *end*の間のバッファーテキストをフィルターしてその結果をリターンする。

デフォルトのフィルター関数は時代遅れとなったラッパーフック `filter-buffer-substring-functions` (この時代遅れの機能に関する詳細はマクロ `with-wrapper-hook`のドキュメント文字列を参照のこと)、および同様に時代遅れとなった変数 `buffer-substring-filters`を参照する。これら両者が `nil`なら、バッファーから未変更のテキスト、すなわち `buffer-substring`がリターンするであろうテキストをリターンする。

*delete*が非 `nil`なら、この関数は `delete-and-extract-region`と同じように、コピー後に *start*と *end*の間のテキストを削除する。

Lisp コードは kill リング、X クリップボード、レジスターのようなユーザーがアクセス可能なデータ構造内にコピーする際には `buffer-substring`、`buffer-substring-no-properties`、`delete-and-extract-region`のかわりにこの関数を使用すること。メジャーモードとマイナーモードはバッファー外部にコピーするテキストを変更するために `filter-buffer-substring-function`を変更することができる。

`filter-buffer-substring-function` [Variable]
この変数の値は実際の処理を行うために `filter-buffer-substring`が呼び出す関数。その関数は `filter-buffer-substring`と同じように 3 つの引数を受けとり、それらは `filter-buffer-substring`にドキュメントされているように扱うこと。関数はフィルターされたテキストをリターン (およびオプションでソーステキストを削除) すること。

The following two variables are obsoleted by 以下の 2 つの変数は`filter-buffer-substring-function`により時代遅れになりましたが、後方互換のために依然としてサポートされます。

`filter-buffer-substring-functions` [Variable]

これは時代遅れとなったラッパーフックであり、このフックのメンバーは *fun*、*start*、*end*、*delete*の 4 つの引数を受け取る関数であること。*fun*は 3 つの引数 (*start*、*end*、*delete*) を受け取り、文字列をリターンする関数。いずれも引数 *start*、*end*、*delete*は `filter-buffer-substring`のときと同様の意味をもつ。

1 つ目のフック関数は `filter-buffer-substring`のデフォルトの処理と同じく *start*と *end* の間の (任意の `buffer-substring-filters`により処理された) バッファー部分文字列をリターン、オプションでバッファーから元テキストを削除する関数であり、それが *fun*に渡される。ほとんどの場合にはフック関数は *fun*を 1 回だけ呼び出してから、その結果にたいして自身の処理を行う。次のフック関数はこれと等しい *fun*を受け取って、それが順次繰り返されていく。実際のリターン値はすべてのフック関数が順次処理した結果。

`buffer-substring-filters` [Variable]

時代遅れとなったこの変数の値は、文字列を唯一の引数ちして別の文字列をリターンする関数のリストであること。デフォルトの `filter-buffer-substring`関数は、バッファー部分文字列をこのリストの 1 つ目の関数に渡して、そのリターン値を次の関数に渡して、これがそれぞれの関数にたいして順次繰り返される。最後の関数のリターン値は `filter-buffer-substring-functions`に渡される。

`current-word` **&optional** *strict really-word* [Function]

この関数はポイント位置またはその付近のシンボル (または単語) を文字列としてリターンする。リターン値にテキストプロパティは含まれない。

オプション引数 *really-word*が非 `nil`なら単語、それ以外はシンボル (単語文字とシンボル構成文字の両方を含む) を探す。

オプション引数 *strict*が非 `nil`のならポイントは単語 (またはシンボル) の内部にあるか隣接しなければならない。そこに単語 (またはシンボル) がなければ、この関数は `nil`をリターンする。*strict*が `nil`ならポイントと同一行にある近接する単語 (またはシンボル) を許容する。

`thing-at-point` *thing* **&optional** *no-properties* [Function]

ポイントに隣接または周辺にある *thing*を文字列としてリターンする。

引数 *thing*は構文エンティティの種別を指定するシンボルである。可能なシンボルとしては `symbol`、`list`、`sexp`、`defun`、`filename`、`url`、`word`、`sentence`、`whitespace`、`line`、`page`、および他が含まれる。

オプション引数 *no-properties*は非 `nil`なら、この関数はリターン値からテキストプロパティを取り除く。

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea⋆ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(thing-at-point 'word)
     ⇒ "Peace"
(thing-at-point 'line)
     ⇒ "Gentlemen may cry ``Peace! Peace!,''\n"
(thing-at-point 'whitespace)
     ⇒ nil
```

32.3 テキストの比較

以下の関数により最初にバッファー内のテキストを文字列内にコピーすることなく、バッファー内のテキスト断片を比較することが可能になります。

`compare-buffer-substrings` *buffer1 start1 end1 buffer2 start2 end2* [Function]

この関数により 1 つのバッファー、または 2 つの異なるバッファーの 2 つの部分文字列 (substrings) を比較できる。最初の 3 つの引数はバッファーとそのバッファー内の 2 つの位置を与えることにより、1 つの部分文字列を指定する。最後の 3 つの引数は、同様の方法によりもう一方の部分文字列を指定する。*buffer1*と *buffer2*のいずれか、または両方にたいしてカレントバッファーを意味する `nil`を使用できる。

1 つ目の部分文字列が 2 つ目の部分文字列より小なら負、大なら正、等しければ値は 0 となる。結果の絶対値は部分文字列内で最初に異なる文字のインデックスに 1 を和した値。

`case-fold-search`が非 `nil`なら、この関数は case(大文字小文字) の違いを無視する。テキストプロパティは常に無視される。

カレントバッファー内にテキスト '`foobarbar haha!rara!`' がある。そしてこの例では 2 つの部分文字列が '`rbar `'と '`rara!`'だとする。1 つ目の文字列の 2 つ目の文字が大きいので値は 2 となる。

```
(compare-buffer-substrings nil 6 11 nil 16 21)
     ⇒ 2
```

32.4 テキストの挿入

挿入 *(insertion)* とはバッファーへの新たなテキストの追加を意味します。テキストはポイント位置、すなわちポイント前の文字とポイント後の文字の間に追加されます。挿入関数は挿入されたテキストの後にポイントを残しますが、前にポイントを残す関数もいくつかあります。前者の挿入をポイント後挿入 *(after point)*、後者をポイント前挿入 *(before point)* と呼びます。

挿入により挿入位置の後にあったマーカーは、テキストを取り囲むように移動されます (Chapter 31 [Markers], page 768 を参照)。マーカーが挿入箇所をさしている際には、挿入によるマーカーの再配置の有無はそのマーカーの挿入タイプに依存します (Section 31.5 [Marker Insertion Types], page 771 を参照)。`insert-before-markers`のような特定のスペシャル関数は、マーカーの挿入タイプとは関係なく挿入されたテキストの後にそのようなすべてのマーカーを再配置します。

カレントバッファーが読み取り専用 (Section 27.7 [Read Only Buffers], page 599 を参照)、または読み取り専用テキスト (Section 32.19.4 [Special Properties], page 820 を参照) を挿入しようとすると、挿入関数はエラーをシグナルします。

以下の関数は文字列やバッファーからプロパティとともにテキスト文字をコピーします。挿入される文字はコピー元の文字と完全に同一のプロパティをもちます。それとは対照的に文字列やバッファーの一部ではない個別の引数として指定された文字は、隣接するテキストからテキストプロパティを継承します。

テキストが文字列かバッファー由来なら、マルチバイトバッファーに挿入するために挿入関数はユニバイトからマルチバイトへの変換、およびその逆も行います。しかしたとえカレントバッファーがマルチバイトバッファーであったとしても、コード 128 から 255 までのユニバイトはマルチバイトに変換しません。Section 33.3 [Converting Representations], page 854 を参照してください。

`insert` **&rest** *args* [Function]
この関数は文字列および/または 1 つ以上の文字 *args*をカレントバッファーのポイント位置に挿入して、ポイントを前方に移動する。言い換えるとポイントの前にテキストを挿入する。すべての *args*が文字列が文字列と文字のいずれでもなければエラーをシグナルする。値は `nil`。

`insert-before-markers` **&rest** *args* [Function]
この関数は文字列および/または 1 つ以上の文字 *args*をカレントバッファーのポイント位置に挿入して、ポイントを前方に移動する。すべての *args*が文字列が文字列と文字のいずれでもなければエラーをシグナルする。値は `nil`。

この関数は他の挿入関数と異なり、挿入されたテキストの後を指すように、まずマーカーが挿入位置を指すよう再配置する。挿入位置からオーバーレイが開始される場合には、挿入されたテキストはそのオーバーレイの外側に出される。空でないオーバーレイが挿入位置で終わる場合には、挿入されたテキストはそのオーバーレイの内側に入れられる。

`insert-char` *character* **&optional** *count inherit* [Command]
このコマンドはカレントバッファーのポイントの前に、*character*のインスタンスを *count*個挿入する。引数 *count*は整数、*character*は文字でなければならない。

インタラクティブに呼び出された際には、このコマンドは *character*にたいしてコードポイントか Unicode 名による入力を求める。Section “Inserting Text” in *The GNU Emacs Manual* を参照のこと。

この関数はたとえカレントバッファーがマルチバイトバッファーであっても、コード 128 から 255 のユニバイト文字をマルチバイト文字に変換しない。Section 33.3 [Converting Representations], page 854 を参照のこと。

*inherit*が非 `nil`なら、挿入された文字は挿入位置前後の 2 文字からステッキーテキストプロパティ(sticky text properties) を継承する。Section 32.19.6 [Sticky Properties], page 827 を参照のこと。

`insert-buffer-substring` *from-buffer-or-name* **&optional** *start end* [Function]
この関数はカレントバッファーのポイント前に、バッファー *from-buffer-or-name*の一部を挿入する。挿入されるテキストは *start* (を含む) から *end* (を含まない) の間のリージョン (これらの引数のデフォルトは、そのバッファーのアクセス可能範囲の先頭と終端)。この関数は `nil` をリターンする。

以下の例ではバッファー ‘`bar`’をカレントバッファーとしてフォームを実行する。バッファー ‘`bar`’は最初は空であるものとする。

```
---------- Buffer: foo ----------
We hold these truths to be self-evident, that all
---------- Buffer: foo ----------

(insert-buffer-substring "foo" 1 20)
     ⇒ nil

---------- Buffer: bar ----------
We hold these truth⋆
---------- Buffer: bar ----------
```

`insert-buffer-substring-no-properties` *from-buffer-or-name* **&optional** *start end* [Function]
これは `insert-buffer-substring`と似ているが、テキストプロパティをコピーしない点が異なる。

テキスト挿入に加えて、隣接するテキストからテキストプロパティを継承する他の関数については Section 32.19.6 [Sticky Properties], page 827 を参照のこと。インデント関数により挿入された空白文字もテキストプロパティを継承する。

32.5 ユーザーレベルの挿入コマンド

このセクションではテキスト挿入のための高レベルコマンド、ユーザーによる使用を意図しているが Lisp プログラムでも有用なコマンドについて説明します。

`insert-buffer` *from-buffer-or-name* [Command]
このコマンドは *from-buffer-or-name* (存在しなければならない) のアクセス可能範囲全体をカレントバッファーのポイントの後に挿入する。マークは挿入されたテキストの後に残される。値は `nil`。

`self-insert-command` *count* **&optional** *char* [Command]
このコマンドは文字 *char*を挿入する。これをポイント前で *count*回繰り返して `nil`をリターンする。ほとんどのプリント文字はこのコマンドにバインドされる。通常の使用では `self-insert-command`は Emacs でもっとも頻繁に呼び出される関数だが、Lisp プログラムではそれをキーマップにインストールする場合を除いて使用されるのは稀。

インタラクティブな呼び出しでは *count*は数プレフィクス引数。

自己挿入では入力文字は `translation-table-for-input`を通じて変換される。Section 33.9 [Translation of Characters], page 863 を参照のこと。

これは、入力文字がテーブル `auto-fill-chars`内にあり、`auto-fill-function`が非 `nil` なら常にそれを呼び出す (Section 32.14 [Auto Filling], page 802 を参照)。

このコマンドは Abbrev モードが有効で、かつ入力文字が単語構成構文をもたなければ abbrev 展開を行う (Chapter 36 [Abbrevs], page 922 と Section 35.2.1 [Syntax Class Table], page 907 を参照)。さらに入力文字が閉カッコ構文 (close parenthesis syntax) をもつ場合には `blink-paren-function`を呼び出す責任もある (Section 39.21 [Blinking], page 1077 を参照)。

このコマンドは最後にフック `post-self-insert-hook`を実行する。たとえばテキストがタイプされたときのように自動的に再インデントするためにこれを使用できる。このフックのいずれかの関数がリージョン (Section 31.8 [The Region], page 776 を参照) にたいして作用する必要があるなら、`post-self-insert-hook`の関数が呼び出される前に Delete Selection モード (Section “Using Region” in *The GNU Emacs Manual* を参照) がリージョンを削除しないようにする必要がある。これを行うには Delete Selection モードにリージョンを削除しないように告げる特別なフック `self-insert-uses-region-functions`に `nil`をリターンする関数を追加すること。

`self-insert-command`の標準的な定義にたいして、独自の定義による置き換えを試みてはならない。エディターコマンドループはこのコマンドを特別に扱うからだ。

`newline` **&optional** *number-of-newlines interactive* [Command]
このコマンドはカレントバッファーのポイントの前に改行を挿入する。*number-of-newlines*が与えられたら、その個数の改行文字が挿入される。インタラクティブな呼び出しでは、*number-of-newlines*はプレフィクス数引数。

この関数はカレント列数が `fill-column`より大、かつ *number-of-newlines*が `nil`なら `auto-fill-function`を呼び出す。このコマンドは改行を挿入するために `self-insert-command`を呼び出して、続けて `auto-fill-function`を呼び出すことにより前の行をブレークする (Section 32.14 [Auto Filling], page 802 を参照)。`auto-fill-function`が通常行うのは改行の挿入であり、最終的な結果としてはポイント位置と、その行のより前方の位置という 2 つの異なる箇所に改行を挿入する。*number-of-newlines*が非 `nil`なら `newline`は auto-fill を行わない。

このコマンドはインタラクティブな呼び出し、または *interactive*が非 `nil`ならフック `post-self-insert-hook`を実行する。

このコマンドは左マージンが 0 でなければ、左マージンにインデントする。Section 32.12 [Margins], page 800 を参照のこと。

リターン値は `nil`。

`overwrite-mode` [Variable]
この変数は overwrite モードが効力をもつかどうかを制御する。値は `overwrite-mode-textual`、`overwrite-mode-binary`、または `nil`。`overwrite-mode-textual`はテキスト的な overwrite モード (改行とタブを特別に扱う)、`overwrite-mode-binary`はバイナリー overwrite モード (改行とタブを普通の文字と同様に扱う) を指定する。

32.6 テキストの削除

削除とはバッファー内のテキストの一部を kill リングに保存せずに取り除くことを意味します (Section 32.8 [The Kill Ring], page 788 を参照)。削除されたテキストを yank することはできませんが、undo メカニズム (Section 32.9 [Undo], page 793 を参照) を使用すれば再挿入が可能です。特別なケースにおいては kill リングにテキストの保存を行う削除関数がいくつかあります。

削除関数はすべてカレントバッファーにたいして処理を行います。

`erase-buffer` [Command]
この関数はカレントバッファーのテキスト全体 (アクセス可能範囲だけではない) を削除してバッファーが読み取り専用なら `buffer-read-only`、バッファー内の一部テキストが読み取り専用なら `text-read-only`をシグナルする。それ以外では確認なしでテキストを削除する。リターン値は `nil`。

バッファーからの大量テキストの削除では、バッファーが大幅に縮小されたという理由により、通常はさらなる自動保存が抑制される。しかし `erase-buffer`は将来のテキストが以前のテキストと関連があるのは稀であり、以前のテキストのサイズと比較されるべきではないというアイデアにもとづいてこれを行わない。

`delete-region` *start end* [Command]
このコマンドはカレントバッファー内の位置 *start*から *end*までの間のテキストを削除して `nil`をリターンする。削除されるリージョン内にポイントがあれば、リージョン削除後のポイントの値は *start*。それ以外の場合は、マーカーが行うようにポイントはテキストを取り囲むように再配置される。

`delete-and-extract-region` *start end* [Function]
この関数はカレントバッファー内の位置 *start*から *end*までの間のテキストを削除して、削除されたテキストを含む文字列をリターンする。

削除されるリージョン内にポイントがあれば、リージョン削除後のポイントの値は *start*。それ以外ならマーカーが行うようにポイントはテキストを取り囲むように再配置される。

`delete-char` *count* **&optional** *killp* [Command]
このコマンドはポイント直後の *count*文字、*count*が負なら直前の *count*文字を削除する。*killp*が非 `nil`なら削除した文字を kill リングに保存する。

インタラクティブな呼び出しでは、*count*は数プレフィクス引数、*killp*は未処理プレフィクス引数 (unprocessed prefix argument)。すなわちプレフィクス引数が与えられたらそのテキストは kill リングに保存され、与えられなければ 1 文字が削除されて、それは kill リングに保存されない。

リターン値は常に `nil`。

`delete-backward-char` *count* **&optional** *killp* [Command]
このコマンドはポイント直前の *count*文字、*count*が負なら直後の *count*文字を削除する。*killp*が非 `nil`なら、削除した文字を kill リングに保存する。

インタラクティブな呼び出しでは、*count*は数プレフィクス引数、*killp*は未処理プレフィクス引数 (unprocessed prefix argument)。すなわちプレフィクス引数が与えられたらそのテキストは kill リングに保存され、与えられなければ 1 文字が削除されて、それは kill リングに保存されない。

リターン値は常に `nil`。

`backward-delete-char-untabify` *count* **&optional** *killp* [Command]
このコマンドはタブをスペースに変換しながら、後方に *count*文字を削除する。次に削除する文字がタブなら、まず適正な位置を保つような数のスペースに変換してから、それらのうちのスペース 1 つをタブのかわりに削除する。*killp*が非 `nil`なら、このコマンドは削除した文字を kill リングに保存する。

タブからスペースへの変換は *count*が正の場合のみ発生する。負の場合はポイント後の正確に −*count*文字が削除される。

インタラクティブな呼び出しでは、*count*は数プレフィクス引数、*killp*は未処理プレフィクス引数 (unprocessed prefix argument)。すなわちプレフィクス引数が与えられたらそのテキストは kill リングに保存され、与えられなければ 1 文字が削除されて、それは kill リングに保存されない。

リターン値は常に `nil`。

`backward-delete-char-untabify-method` [User Option]
このオプションは `backward-delete-char-untabify`が空白文字を扱う方法を指定する。可能な値には `untabify` (タブを個数分のスペースに変換してスペースを 1 つ削除。これがデフォルト)、`hungry` (1 コマンドでポイント前のタブとスペースすべてを削除する)、`all` (ポイント前のタブとスペース、および改行すべてを削除する)、`nil` (空白文字にたいして特に何もしない)。

32.7 ユーザーレベルの削除コマンド

このセクションでは、主にユーザーにたいして有用ですがLispプログラムでも有用なテキストを削除するための高レベルのコマンドを説明します。

`delete-horizontal-space` **&optional** *backward-only* [Command]

この関数はポイント近辺のすべてのスペースとタブを削除する。リターン値は`nil`。

*backward-only*が非`nil`なら、この関数はポイント前のスペースとタブを削除するがポイント後のスペースとタブは削除しない。

以下の例では、各行ごとに2番目と3番目の間にポイントを置いて、`delete-horizontal-space`を4回呼び出している。

```
---------- Buffer: foo ----------
I ⋆thought
I ⋆     thought
We⋆ thought
Yo⋆u thought
---------- Buffer: foo ----------

(delete-horizontal-space)   ; Four times.
     ⇒ nil

---------- Buffer: foo ----------
Ithought
Ithought
Wethought
You thought
---------- Buffer: foo ----------
```

`delete-indentation` **&optional** *join-following-p beg end* [Command]

この関数はポイントのある行をその前の行に結合(join)する。結合においてはすべての空白文字を削除、特定のケースにおいてはそれらを1つのスペースに置き換える。*join-following-p*が非`nil`なら、`delete-indentation`はかわりに後続行と結合を行う。それ以外の場合には*beg*と*end*が非`nil`なら、この関数は*beg*と*end*で定義されるリージョン内のすべての行を結合する。

インタラクティブな呼び出しでは*join-following-p*はプレフィクス引数、*beg*と*end*はリージョンがアクティブならリージョンの開始と終了、それ以外は`nil`。この関数は`nil`をリターンする。

fillプレフィクスがあり、結合される2つ目の行もそのプレフィクスで始まる場合には、行の結合前に`delete-indentation`はその fillプレフィクスを削除する。Section 32.12 [Margins], page 800 を参照のこと。

以下の例では‘`events`’で始まる行にポイントがあり、前の行の末尾に1つ以上のスペースが存在しても違いは生じない。

```
---------- Buffer: foo ----------
When in the course of human
⋆    events, it becomes necessary
---------- Buffer: foo ----------

(delete-indentation)
     ⇒ nil
```

```
---------- Buffer: foo ----------
When in the course of human⋆ events, it becomes necessary
---------- Buffer: foo ----------
```

行の結合後に結合点に単一のスペースを残すか否かを決定するのは、関数 `fixup-whitespace` の責任である。

`fixup-whitespace` [Command]

この関数はポイントを取り囲むすべての水平スペースを、コンテキストに応じて 1 つのスペースまたはスペースなしに置き換える。リターン値は `nil`。

行の先頭や末尾において、スペースの適正な数は 0。閉カッコ構文 (close parenthesis syntax) の前の文字、開カッコの後の文字、式プレフィクス構文 (expression-prefix syntax) においても、スペースの適正な数は 0。それ以外ではスペースの適正な数は 1。Section 35.2.1 [Syntax Class Table], page 907 を参照のこと。

以下の例では最初に 1 行目の単語 '`spaces`'の前にポイントがある状態で、`fixup-whitespace` を呼び出している。2 回目の呼び出しでは '`(`'の直後にポイントがある。

```
---------- Buffer: foo ----------
This has too many     ⋆spaces
This has too many spaces at the start of (⋆   this list)
---------- Buffer: foo ----------

(fixup-whitespace)
     ⇒ nil
(fixup-whitespace)
     ⇒ nil

---------- Buffer: foo ----------
This has too many spaces
This has too many spaces at the start of (this list)
---------- Buffer: foo ----------
```

`just-one-space` **&optional** *n* [Command]

このコマンドはポイントを取り囲むすべてのスペースを 1 つのスペース、*n*が指定された場合は *n*個のスペースで置き換える。リターン値は `nil`。

`delete-blank-lines` [Command]

この関数はポイントを取り囲む空行を削除する。ポイントが前後に 1 行以上の空行がある空の行にある場合には、1 行を除いてそれらすべてを削除する。ポイントが孤立した空行にあればその行を削除する。ポイントが空でない行にあれば、その直後にあるすべての空白を削除する。

空行とはタブまたはスペースのみを含む行として定義される。

`delete-blank-lines`は `nil`をリターンする。

`delete-trailing-whitespace` **&optional** *start end* [Command]

*start*と *end*で定義されるリージョン内から末尾の空白文字を削除する。

このコマンドはリージョン内の各行の最後の非空白文字後にある空白文字を削除する。

このコマンドがバッファー全体 (マークが非アクティブな状態で呼び出された場合や Lisp から *end*と `nil`で呼び出された場合) にたいして動作する場合には、変数 `delete-trailing-lines` が非 `nil`ならバッファーの終端行の末尾の行も削除する。

32.8 kill リング

kill 関数 *(kill functions)* は削除関数のようにテキストを削除しますが、ユーザーが *yank* により再挿入できるようにそれらを保存する点が異なります。これらの関数のほとんどは '`kill-`'という名前をもちます。対照的に名前が '`delete-`'で始まる関数は、(たとえ削除を undo できるとしても) 通常は yank 用にテキストを保存しません。それらは削除 (deletion) 関数です。

ほとんどの kill コマンドは主にインタラクティブな使用を意図しており、ここでは説明しません。ここで説明するのは、そのようなコマンドの記述に使用されるために提供される関数です。テキストを kill するために、これらの関数を使用できます。Lisp 関数の内部的な目的のためにテキストの削除を要するときは、kill リング内のコンテンツに影響を与えないように通常は削除関数を使用するべきでしょう。Section 32.6 [Deletion], page 784 を参照してください。

kill されたテキストは後の yank 用に *kill* リング *(kill ring)* 内に保存されます。これは直前の kill だけでなく直近の kill のいくつかを保持するリストです。yank がそれをサイクル順に要素をもつリストとして扱うので、これを "リング (ring)" と称しています。このリストは変数 `kill-ring`に保持されており、リスト用の通常関数で操作可能です。このセクションで説明する、これをリングとして扱うために特化された関数も存在します。

特に kill された実体が破壊されてしまわないような操作を参照するという理由から、"kill" という単語の使用が不適切だと考える人もいます。これは通常の生活において死は永遠であり、kill された実体は生活に戻ることはないことと対照的です。したがって他の比喩表現も提案されてきました。たとえば、"cut リング (cut ring)" という用語は、コンピューター誕生前に原稿を再配置するためにハサミで切り取って貼り付けていたような人に意味があるでしょう。しかし今となってはこの用語を変更するのは困難です。

32.8.1 kill リングの概念

kill リングはリスト内でもっとも最近に kill されたテキストが先頭になるように、kill されたテキストを記録します。たとえば短い kill リングは以下のようになるでしょう:

```
("some text" "a different piece of text" "even older text")
```

このリストのエントリー長が `kill-ring-max`に達すると、新たなエントリー追加により最後のエントリーが自動的に削除されます。

kill コマンドが他のコマンドと混ざり合っているときは、各 kill コマンドは kill リング内に新たなエントリーを作成します。連続する複数の kill コマンドは単一の kill リングエントリーを構成します。これは 1 つの単位として yank されます。2 つ目以降の連続する kill コマンドは、最初の kill により作成されたエントリーにテキストを追加します。

yank にたいしては、kill リング内のただ 1 つのエントリーが、そのリングの先頭のエントリーとなります。いくつかの yank コマンドは、異なる要素を先頭に指定することにより、リングを回転 (rotate) させます。しかしこの仮想的回転はリスト自身を変更しません。もっとも最近のエントリーが、常にリスト内の最初に配置されます。

32.8.2 kill 用の関数

`kill-region`はテキスト kill 用の通常サブルーチンです。この関数を呼び出すすべてのコマンドは kill コマンドです (そして恐らくは名前に '`kill`'が含まれる)。`kill-region`は新たに kill されたテキストを kill リング内の最初の要素内に置くか、それをもっとも最近の要素に追加します。これは前のコマンドが kill コマンドか否かを、(`last-command`を使用して) 自動的に判別して、もし kill コマンドなら kill されたテキストをもっとも最近のエントリーに追加します。

以下で説明するコマンドは kill されるテキストが kill リングに保存される前に、それらをフィルターできます。これらの関数は、このフィルタリングを行うために `filter-buffer-substring`を

呼び出します (Section 32.2 [Buffer Contents], page 778 を参照)。デフォルトではこれらはフィルタリングを行いませんが、バッファーにあったときと異なるようにテキストを kill リングに保存するために、マイナーモードとフック関数はフィルタリングをセットアップできます。

`kill-region` *start end* **&optional** *region* [Command]
この関数は *start*と *end*の間のテキスト範囲を kill するが、オプション引数 *region*が非 `nil`なら、かわりにカレントリージョンのテキストを kill する。そのテキストは削除されるが、そのテキストプロパティと共に kill リングに保存される。値は常に `nil`。

インタラクティブな呼び出しでは *start*と *end*はポイントとマークで *region*は常に非 `nil`なので、このコマンドは常にカレントリージョン内のテキストを kill する。

バッファーまたはテキストが読み取り専用なら、`kill-region`は同じように kill リングを変更後に、バッファーを変更せずにエラーをシグナルする。これはユーザーが一連の kill コマンドで、読み取り専用バッファーから kill リングにテキストをコピーするのに有用。

`kill-read-only-ok` [User Option]
このオプションが非 `nil`なら、バッファーやテキストが読み取り専用でも `kill-region`はエラーをシグナルしない。かわりにバッファーを変更せずに kill リングを更新して単にリターンする。

`copy-region-as-kill` *start end* **&optional** *region* [Command]
この関数は *start*と *end*の間のテキスト範囲 (テキストプロパティを含む) を kill リングに保存するが、バッファーからそのテキストを削除しない。しかしオプション引数 *region*が非 `nil`なら、この関数は *start*と *end*を無視して、かわりにカレントリージョンを保存する。この関数は常に `nil`をリターンする。

インタラクティブな呼び出しでは *start*と *end*はポイントとマークで *region*は常に非 `nil`なので、このコマンドは常にカレントリージョン内のテキストを kill する。

このコマンドは後続の kill コマンドが同一の kill リングエントリーに追加しないように、`this-command`に `kill-region`をセットしない。

32.8.3 yank

yank とは kill リングからテキストを挿入しますが、それが単なる挿入ではないことを意味します。`yank`とそれに関連するコマンドは、テキスト挿入前に特別な処理を施すために `insert-for-yank`を使用します。

`insert-for-yank` *string* [Function]
この関数は `insert`と同様に機能するが、結果をカレントバッファーに挿入する前にテキストプロパティ`yank-handler`、同様に変数 `yank-handled-properties`と `yank-excluded-properties`に応じて *string*内のテキストを処理する点が異なる。

`insert-buffer-substring-as-yank` *buf* **&optional** *start end* [Function]
この関数は `insert-buffer-substring`と似ているが、`yank-handled-properties`と `yank-excluded-properties`に応じてテキストを処理する点が異なる (これは `yank-handler`プロパティを処理しないが、いずれにせよバッファー内のテキストでは通常は発生しない)。

文字列の一部またはすべてにテキストプロパティ`yank-handler`を put すると、`insert-for-yank`が文字列を挿入する方法が変更されます。文字列の別の箇所が異なる `yank-handler`の値をも

つ場合 (比較は eq)、部分文字列はそれぞれ個別に処理されます。プロパティ値は以下の形式からなる 1 から 4 要素のリストでなければなりません (2 番目以降の要素は省略可):

```
(function param noexclude undo)
```

これらの要素が何を行うかを以下に示します:

function *function*が非 nilなら、insertのかわりに文字列を挿入するために、挿入する文字列を単一の引数として、その関数が呼び出される。

param 非 nilの *param*が与えられた場合には、それは *string* (または処理される *string*の部分文字列) を置き換えるオブジェクトとして *function* (または insert) に渡される。たとえば *function*が yank-rectangleなら、*param*は矩形 (rectangle) として挿入されるべき文字列のリスト。

noexclude 非 nilの *noexclude*が与えられたら、挿入される文字列にたいする yank-handled-propertiesと yank-excluded-propertiesの通常の動作を無効にする。

undo 非 nilの *undo*が与えられたら、それはカレントオブジェクトの挿入を undo するために yank-popが呼び出す関数。この関数はカレントリージョンの start と end という 2 つの引数で呼び出される。*function*は yank-undo-functionをセットすることにより *undo*の値をオーバーライドできる。

yank-handled-properties [User Option]
この変数は yank されるテキストの状態を処理するスペシャルテキストプロパティを指定する。これは (通常の方法、または yank-handlerを通じた) テキストの挿入後、yank-excluded-propertiesが効力をもつ前に効果を発揮する。

値は要素が (*prop* . *fun*)であるような alist であること。alist の各要素は順番に処理される。挿入されるテキストはテキスト範囲にたいして、テキストプロパティが *prop*と eqなものがスキャンされる。そのような範囲にたいしてプロパティの値、そのテキストの開始と終了の位置という 3 つの引数により *fun*が呼び出される。

yank-excluded-properties [User Option]
この変数の値は挿入されるテキストから削除するためのプロパティのリスト。デフォルト値にはマウスに応答したりキーバインディングの指定を引き起こすテキストのような、煩わしい結果をもたらすかもしれないプロパティが含まれる。これは yank-handled-propertiesの後に効果を発揮する。

32.8.4 yank 用の関数

このセクションでは yank 用の高レベルなコマンドを説明します。これらのコマンドは主にユーザー用に意図されたものですが、Lisp プログラム内での使用にたいしても有用です。yankと yank-popはいずれも、変数 yank-excluded-propertiesとテキストプロパティ yank-handlerにしたがいます (Section 32.8.3 [Yanking], page 789 を参照)。

yank **&optional** *arg* [Command]
このコマンドは kill リングの先頭にあるテキストをポイントの前に挿入する。これは push-mark(Section 31.7 [The Mark], page 772 を参照) を使用して、そのテキストの先頭にマークをセットする。

*arg*が非 nilのリスト (これはユーザーがインタラクティブに数字を指定せずに *C-u*タイプ時に発生する) なら、yankは上述のようにテキストを挿入するがポイントは yank されたテキストの前、マークは yank されたテキストの後に置かれる。

*arg*が数字なら `yank`は *arg*番目に最近 kill されたテキスト、すなわち kill リングリストの *arg* 番目の要素を挿入する。この順番はコマンドの目的にたいして 1 番目の要素としてみなされるリスト先頭の要素から巡回的に数えられる。

`yank`は、それが他のプログラムから提供されるテキストを使用しないかぎり (使用する場合はそのテキストを kill リングに push する)、kill リングのコンテンツを変更しない。しかし *arg* が非 1 の整数なら、kill リングを転回 (rotate) して yank されるテキストをリング先頭に置く。

`yank`は `nil`をリターンする。

`yank-pop` **&optional** *arg* [Command]

このコマンドは kill リング上の正に yank されたばかりのエントリーを kill リングの別エントリーで置き換える。

このコマンドは `yank`か別の `yank-pop`の直後のみ許される。そのような際にそのリージョンには、yank により正に挿入されたテキストが含まれる。`yank-pop`はそのテキストを削除して、kill された別のテキスト片をその位置に挿入する。そのテキスト片はすでに kill リング内のどこか別の箇所にあるので、これは削除されたテキストを kill リングに追加しない。しかし新たに yank されたテキストが先頭になるように、kill リングの転回は行う。

*arg*が `nil`なら置換テキストは kill リングの 1 つ前の要素。*arg*が数字なら置換テキストは kill リングの *arg*個前の要素である。*arg*が負なら、より最近の kill が置換される。

kill リング内の kill されたエントリーの順序はラップする。すなわちもっとも古い kill の次にもっとも新しい kill、もっとも新しい kill の前はもっとも古い kill となる。

リターン値は常に `nil`である。

`yank-undo-function` [Variable]

この変数が非 `nil`なら、関数 `yank-pop`は前の `yank`や `yank-pop`により挿入されたテキストを削除するために、`delete-region`のかわりにこの変数の値を使用する。値はカレントリージョンの開始と終了という 2 つの引数をとる関数でなければならない。

関数 `insert-for-yank`はテキストプロパティ`yank-handler`の要素 *undo*に対応して、この変数を自動的にセットする。

32.8.5 低レベルの kill リング

以下の関数と変数は kill リングにたいして低レベルなアクセスを提供しますが、それらはウィンドウシステムの選択 (Section 29.20 [Window System Selections], page 747 を参照) との相互作用にも留意するので、Lisp プログラム内での使用に関しても依然として有用です。

`current-kill` *n* **&optional** *do-not-move* [Function]

関数 `current-kill`は kill リングの先頭を指す yank ポインターを、(新しい kill から古い kill に) *n*個転回して、リング内のその箇所のテキストをリターンする。

オプションの第 2 引数 *do-not-move*が非 `nil`なら、`current-kill`は yank ポインターを変更しない。カレント yank ポインターから *n*個目の kill を単にリターンする。

*n*が 0 ならそれは最新の kill の要求を意味しており、`current-kill`は kill リング照会前に `interprogram-paste-function`(以下参照) の値を呼び出す。その値が関数で、かつそれが文字列か複数の文字列からなるリストをリターンすると、`current-kill`はその文字列を kill リング上に push して最初の文字列をリターンする。これは *do-not-move*の値に関わらず、`interprogram-paste-function`がリターンする最初の文字列の kill リングエントリーを指すように yank ポインターのセットも行う。それ以外では `current-kill`は *n*にたいする 0 値を特別に扱うことはなく、yank ポインターが指すエントリーをリターンして yank ポインターの移動は行わない。

`kill-new` *string* **&optional** *replace* [Function]
この関数はテキスト *string*を kill リング上に push して yank ポインターがそれを指すようにセットする。それが適切なら、もっとも古いエントリーを破棄する。`interprogram-paste-functioninterprogram-paste-function` (ユーザーオプション `save-interprogram-paste-before-kill`にしたがう) と `interprogram-cut-function` (以下参照) の値の呼び出しも行う。

*replace*が非 `nil`なら `kill-new`は kill リング上に *string*を push せずに、kill リングの 1 つ目の要素を *string*に置き換える。

`kill-append` *string before-p* [Function]
この関数は kill リング内の最初のエントリーにテキスト *string*を追加して、その結合されたエントリーを指すように yank ポインターをセットする。通常はそのエントリーの終端に *string*が追加されるが、*before-p*が非 `nil`ならエントリーの先頭に追加される。この関数はサブルーチンとして `kill-new`も呼び出すので `interprogram-cut-function`とおそらく `interprogram-paste-function`の値 (以下参照) が拡張により呼び出される。

`interprogram-paste-function` [Variable]
この変数は他のプログラムから kill リングへ kill されたテキストを転送する方法を提供する。値は `nil`、または引数のない関数であること。

値が関数なら、もっとも最近の kill を取得するために `current-kill`はそれを呼び出す。その関数が非 `nil`値をリターンすると、その値がもっとも最近の kill として使用される。`nil`をリターンしたら kill リングの先頭が使用される。

複数選択をサポートするウィンドウシステムのサポートを容易にするために、この関数は文字列のリストをリターンすることもある。その場合には 1 つ目の文字列がもっとも最近の kill として使用され、その他の文字列はすべて `yank-pop`によるアクセスを容易にするために kill リング上に push される。

この関数の通常の用途は、たとえそれが他アプリケーションに属する選択であっても、もっとも最近の kill としてウィンドウシステムのクリップボードからそれを取得することである。しかしクリップボードのコンテンツがカレント Emacs セッションに由来するなら、この関数は `nil`をリターンする筈である。

`interprogram-cut-function` [Variable]
この変数はウィンドウシステム使用時に、他のプログラムに kill されたテキストを転送する方法を提供する。値は `nil`、または 1 つの引数を要求する関数であること。

値が関数なら `kill-new`と `kill-append`は kill リングの新たな 1 つ目要素を引数としてそれを呼び出す。

この関数の通常の用途は、新たに kill されたテキストをウィンドウシステムのクリップボードに配置することである。Section 29.20 [Window System Selections], page 747 を参照のこと。

32.8.6 kill リングの内部

変数 `kill-ring`は、文字列リスト形式で kill リングのコンテンツを保持します。もっとも最近の kill が常にこのリストの先頭になります。

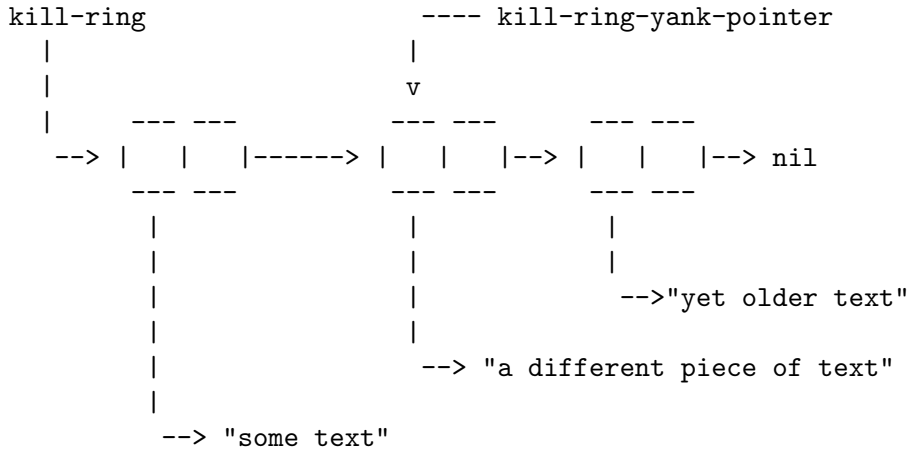
変数 `kill-ring-yank-pointer`は、CAR が次の yank のテキストであるような、kill リングリスト内のリンクを point します。これをリングの先頭を識別すると言います。そして、`kill-ring-yank-pointer`を異なるリンクに移動することを *kill* リングの転回 *(rotating the kill ring)* と呼びます。yank ポインターを移動する関数は yank ポインターをリスト終端から先頭、またはその逆へラッ

プするので kill リングを “ring” と呼びます。kill リングの転回は仮想的なものであって `kill-ring` の値は変更しません。

`kill-ring`と `kill-ring-yank-pointer`はいずれも、通常は値がリストであるような Lisp 変数です。`kill-ring-yank-pointer`の名前にある単語 “pointer” は、その変数の目的が次回 yank コマンドにより使用されるリストの最初の要素を指すことであることを示します。

`kill-ring-yank-pointer`の値は常に kill リングリスト内の 1 つのリンクと `eq`です。それが指す要素は、そのリンクの CAR です。kill リングを変更する kill コマンドも、この変数に `kill-ring` の値をセットします。その効果は新たに kill された先頭になるように、リングを転回することです。

以下は変数 `kill-ring-yank-pointer`が、kill リング (`"some text" "a different piece of text" "yet older text"`)内の 2 番目のエントリーを指すことを表すダイアグラムです。



この状態は *C-y* (`yank`) の直後に *M-y* (`yank-pop`) を行うことにより発生し得ます。

`kill-ring` [Variable]
この変数はもっとも最近に kill されたテキストが先頭になるように、kill されたテキストのシーケンスのリストを保持する。

`kill-ring-yank-pointer` [Variable]
この変数の値は、yank にたいして使用される kill リングの先頭にある要素を示す。より正確には値は `kill-ring`の値の tail 値であり、その CAR が *C-y*により yank されるはずの kill 文字列。

`kill-ring-max` [User Option]
この変数の値は、リング終端の要素を破棄する前に kill リングが成長し得る最大長。`kill-ring-max`のデフォルト値は 60。

32.9 アンドゥ

ほとんどのバッファーは、バッファーのテキストにたいして行われた変更を undo できるように、すべての変更を記録する *undo* リスト *(undo list)* をもっています (undo リストをもたないバッファーとは通常は Emacs が undo を有用とみなさない特殊用途のバッファーである。特に名前がスペースで始まるバッファーはすべて undo の記録がデフォルトでオフになっている。Section 27.3 [Buffer Names], page 592 を参照)。バッファー内でテキストを変更するすべてのプリミティブは undo リストの先頭に自動的に要素を追加して、それは変数 `buffer-undo-list`に格納されます。

`buffer-undo-list` [Variable]
このバッファーローカル変数の値は、カレントバッファーの undo リスト。値が `t`なら undo 情報の記録を無効にする。

以下は undo リストが保有可能な要素の種類です:

position この種の要素は前のポイント値を記録する。この要素を undo することによりポイントは *position*に移動する。通常のカーソル移動はどのような類の undo 記録も作成しないが、削除操作はそのコマンド以前にポイントがあった場所を記録するためにこのエントリーを使用する。

(*beg* . *end*)
この種の要素は挿入されたテキストを削除する方法を示す。挿入においてそのテキストはバッファー内の範囲 *beg*から *end*を占める。

(*text* . *position*)
この種の要素は削除されたテキストを再度挿入する方法を示す。文字列 *text*は削除されたテキストそのもの。削除されたテキストを再挿入する位置は `(abs` *position*`)`。*position* が正ならポイントがあったのは削除されたテキストの先頭、それ以外では末尾。この要素の直後に 0 個以上の (*marker* . *adjustment*) 要素が続く。

(t . *time-flag*)
この種の要素は未変更のバッファーが変更されたことを示す。*time-flag* (非整数の Lisp タイムスタンプ) は、visit されたファイルにたいしてそれが以前に visit や保存されたときの更新時刻 (modification time) を、`current-time`と同じ形式を用いて表す。Section 40.5 [Time of Day], page 1104 を参照のこと。*time-flag*が 0 ならそのバッファーに対応するファイルがないことを、−1 なら visit されたファイルは以前は存在しなかったことを意味する。`primitive-undo`はバッファーを再度未変更とマークするかどうかを判断するために、これらの値を使用する (ファイルの状態が *time-flag*のそれとマッチする場合のみ未変更とマーク)。

(nil *property value beg* . *end*)
この種の要素はテキストプロパティの変更を記録する。変更を undo する方法は以下のようになる:

```
(put-text-property beg end property value)
```

(*marker* . *adjustment*)
この種の要素はマーカー *marker*がそれを取り囲むテキストの削除により再配置されて、*adjustment*文字位置を移動したということを記録する。undo リスト内の前にある要素 (*text* . *position*) とマーカーの位置が一致する場合には、この要素を undo することにより *marker* − *adjustment*文字移動する。

(apply *funname* . *args*)
これは拡張可能な undo アイテムであり、引数 *args*とともに *funname*を呼び出すことにより undo が行われる。

(apply *delta beg end funname* . *args*)
これは拡張可能な undo アイテムであり、*beg*から *end*までに限定された範囲にたいして、そのバッファーのサイズを *delta*文字増加させる変更を記録する。これは引数 *args* とともに *funname*を呼び出すことにより undo が行われる。

この種の要素は、それがリージョンと関係するか否かを判断することによりリージョンに限定された undo を有効にする。

`nil` この要素は境界 (boundary) である。2 つの境界の間にある要素を変更グループ *(change group)* と呼び、それぞれの変更グループは通常 1 つのキーボードコマンドに対応するとともに、undo コマンドは通常はグループを 1 つの単位として全体を undo を行う。

`undo-boundary` [Function]
この関数は undo リスト内に境界を配置する。このような境界ごとに undo コマンドは停止して、連続する undo コマンドは、より以前の境界へと undo を行っていく。この関数は `nil`をリターンする。

この関数を明示的に呼び出すことは、あるコマンドの効果を複数単位に分割するために有用である。たとえば `query-replace`はユーザーが個別に置換を undo できるように、それぞれの置換後に `undo-boundary`を呼び出している。

しかしほとんどの場合には、この関数は適切なタイミングで自動的に呼び出される。

`undo-auto-amalgamate` [Function]
エディターコマンドループは各アンドゥがが通常はそれぞれ 1 つのコマンドの効果をアンドゥするように、各キーシーケンスを実行する直前に `undo-boundary`を呼び出す。少数の例外は融合 *(amalgamating)* コマンドである。これらのコマンドは一般的にバッファーにたいして小さい変更を発生させるので、変更をグループとしてアンドゥできるように、20 回目のコマンドごとに境界が挿入される。デフォルトでは自己挿入入力文字を生成するコマンド `self-insert-command` (Section 32.5 [Commands for Insertion], page 783 を参照)、文字を削除するコマンド `delete-char` (Section 32.6 [Deletion], page 784 を参照) は融合コマンドである。複数バッファーのコンテンツに影響するコマンド、たとえば発生し得るとすれば `post-command-hook`上の関数が `current-buffer`以外のバッファーに影響を及ぼす場合には、影響を受ける各バッファーごとに `undo-boundary`が呼び出されるだろう。

この関数を融合コマンドの前に呼び出すことができる。そのような一連の呼び出しが行われていると、以前の `undo-boundary`は削除される。

融合可能な最大の変更数は `amalgamating-undo-limit`変数で制御される。この変数が 1 なら変更は融合されない。

`undo-auto-current-boundary-timer` [Variable]
プロセスバッファーのようないくつかのバッファーでは、何もコマンドを実行していなくても変更が発生し得る。このような場合には、通常は `undo-boundary`この変数内のタイマーにより定期的に呼び出される。この挙動を抑制するには、この変数を非 `nil`にセットすること。

`undo-in-progress` [Variable]
この変数は通常は `nil`だが、undo コマンドはこれを `t`にバインドする。これによりさまざまな種類の変更フックが undo により呼び出された際に、それを告げることが可能になる。

`primitive-undo` *count list* [Function]
これは undo リストの要素の undo にたいする基本的な関数。これは *list*の最初の *count*要素を undo して *list*の残りをリターンする。

`primitive-undo`はバッファー変更時に、そのバッファーの undo リストに要素を追加する。undo コマンドは混乱を避けるために undo 操作シーケンス冒頭に undo リストの値を保存する。その後で undo 操作は保存された値の使用と更新を行う。undo により追加された新たな要素はこの保存値の一部でないので継続する undo と干渉しない。

この関数は `undo-in-progress`をバインドしない。

いくつかのコマンは、コマンドの選択的なアンドゥを妨害する方法により、実行後にリージョンをアクティブなままにします。undoをそのようなコマンドの直後に呼び出した際にアクティブなリージョンを無視するには、コマンドの関数シンボルの undo-inhibit-regionプロパティに非 nil 値をセットします。Section 9.4.2 [Standard Properties], page 130 を参照してください。

32.10 アンドゥリストの保守

このセクションでは与えられたバッファーにたいして undo 情報を有効や無効にする方法を説明します。undo リストが巨大化しないように undo リストを切り詰める方法も説明します。

新たに作成されたバッファー内の undo 情報記録は、通常は開始とともに有効になります。しかしバッファー名がスペースで始まる場合には、undo の記録は初期状態では無効になっています。以下の 2 つの関数、または自身で buffer-undo-listをセットすることにより、undo 記録の有効化や無効化を明示的に行うことができます。

buffer-enable-undo **&optional** *buffer-or-name* [Command]
: このコマンドは以降の変更を undo 可能にするように、バッファー *buffer-or-name*の undo 情報記録を有効にする。引数が与えられなければカレントバッファーを使用する。そのバッファー内の undo 記録がすでに有効ならこの関数は何も行わない。リターン値は nil。

  インタラクティブな呼び出しでは *buffer-or-name*はカレントバッファーであり、他のバッファーを指定することはできない。

buffer-disable-undo **&optional** *buffer-or-name* [Command]
: この関数は *buffer-or-name*の undo リストを破棄して、それ以上の undo 情報記録を無効にする。結果として以前の変更と以後のすべての変更にたいするそれ以上の undo は不可能になる。*buffer-or-name*の undo リストがすでに無効ならこの関数に効果はない。

  インタラクティブな呼び出しでは BUFFER-OR-NAME はカレントバッファー。他のバッファーを指定することはできない。リターン値は nil。

編集が継続されるにつれて undo リストは次第に長くなっていきます。利用可能なメモリー空間すべてを使い尽くすのを防ぐために、ガベージコレクションが undo リストを設定可能な制限サイズに切り詰めて戻します (この目的のために undo リストのサイズはリストを構成するコンスセルに加えて削除された文字列により算出される)。undo-limit、undo-strong-limit、undo-outer-limit の 3 つの変数は、許容できるサイズの範囲を制御します。これらの変数においてサイズは専有するバイト数で計数され、それには保存されたテキストとその他データが含まれます。

undo-limit [User Option]
: これは許容できる undo リストサイズのソフトリミット。このサイズを超過した箇所の変更グループは最新の変更グループ 1 つが保持される。

undo-strong-limit [User Option]
: これは undo リストの許容できるサイズの上限。このサイズを超過する箇所の変更グループは (その他すべてのより古い変更グループとともに) 自身を破棄する。1 つ例外があり undo-outer-limitを超過すると最新の変更グループだけが破棄される。

undo-outer-limit [User Option]
: ガベージコレクション時にカレントコマンドの undo 情報がこの制限を超過したら、Emacs はその情報を破棄して警告を表示する。これはメモリーオーバーフローを防ぐための最後の回避用リミットである。

`undo-ask-before-discard` [User Option]

この変数が非`nil`なら undo 情報の`undo-outer-limit`超過時に、Emacs はその情報を破棄するかどうかをエコーエリアで尋ねる。デフォルト値は`nil`でこれは自動的な破棄を意味する。

このオプションは主にデバッグを意図している。これを尋ねる際にはガベージコレクションは抑制されており、もしユーザーがその問にたいして答えるのをあまりに長くかかるなら、Emacs がメモリーリークを起こすかもしれないことを意味する。

32.11 fill

フィル *(fill:* 充填*)* とは、指定された最大幅付近 (ただし超過せず) に、(行ブレークを移動することにより) 行の長さを調整することを意味します。加えて複数行を位置揃え *(justify)* することもできます。位置揃えとはスペースを挿入して左および/または右マージンを正確に整列させることを意味します。その幅は変数`fill-column`により制御されます。読みやすくするために行の長さは 70 列程度を超えないようにするべきです。

テキストの挿入とともに自動的にテキストをフィルする Auto Fill モードを使用できますが、既存テキストの変更では不適切にフィルされたままになるかもしれません。その場合にはテキストを明示的にフィルしなければなりません。

このセクションのコマンドのほとんどは有意な値をリターンしません。フィルを行うすべての関数はカレント左マージン、カレント右マージン、カレント位置揃えスタイルに留意します (Section 32.12 [Margins], page 800 を参照)。カレント位置揃えスタイルが`none`なら、フィル関数は実際には何も行いません。

フィル関数のいくつかは引数 *justify* を受け取ります。これが非`nil`なら、それは何らかの類の位置揃えを要求します。特定の位置揃えスタイルを要求するために`left`、`right`、`full`、`center`を指定できます。これが`t`なら、それはそのテキスト部分にたいしてカレント位置揃えスタイルを使用することを意味します (以下の`current-justification`を参照)。その他すべての値は`full`として扱われます。

インタラクティブにフィル関数を呼び出すには際、プレフィクス引数の使用は *justify* にたいして暗に値`full`を指定します。

`fill-paragraph` **&optional** *justify region* [Command]

このコマンドはポイント位置、またはその後のパラグラフ (paragraph: 段落) をフィルする。*justify* が非`nil`なら、同様に各行が位置揃えされる。これはパラグラフ境界を探すために、通常のパラグラフ移動コマンドを使用する。Section “Paragraphs” in *The GNU Emacs Manual* を参照のこと。

もし *region* が非`nil`で、Transient Mark モードが有効かつマークがアクティブなら、このコマンドはカレントパラグラフのみフィルするかわりに、リージョン内すべてのパラグラフをフィルするためにコマンド`fill-region`を呼び出す。このコマンドがインタラクティブに呼び出された際は、*region* は`t`。

`fill-region` *start end* **&optional** *justify nosqueeze to-eop* [Command]

このコマンドは *start* から *end* のリージョン内のすべてのパラグラフをフィルする。*justify* が非`nil`なら同様に位置揃えも行う。

nosqueeze が非`nil`なら、それは行ブレーク以外の空白文字を残すことを意味する。*to-eop* が非`nil`なら、それはパラグラフ終端 (以下の`use-hard-newlines`が有効なら次の hard 改行) までのフィルを維持することを意味する。

変数`paragraph-separate`はパラグラフを分割する方法を制御する。Section 34.8 [Standard Regexps], page 905 を参照のこと。

`fill-individual-paragraphs` *start end* **&optional** *justify citation-regexp* [Command]

このコマンドはリージョン内の各パラグラフを、それの固有なフィルプレフィクスに応じてフィルする。したがってパラグラフの行がスペースでインデントされていれば、フィルされたパラグラフは同じ様式でインデントされた状態に保たれるだろう。

最初の 2 つの引数 *start* と *end* はフィルするリージョンの先頭と終端。3 つ目の引数 *justify*、4 つ目の引数 *citation-regexp* はオプション。*justify* が非 `nil` なら、そのパラグラフはフィルと同様に位置揃えも行われる。*citation-regexp* が非 `nil` なら、それはこの関数がメールメッセージを処理しているのでヘッダーラインをフィルするべきではないことを意味する。*citation-regexp* が文字列なら、それは正規表現として扱われる。それが行の先頭にマッチすれば、その行は引用マーカー (citation marker) として扱われる。

`fill-individual-paragraphs` は通常はインデントの変更を新たなパラグラフの開始とみなす。`fill-individual-varying-indent` が非 `nil` ならセパレーターラインだけがパラグラフを分割する。その場合には、最初の行からさらにインデントが追加されたパラグラフを処理することが可能になる。

`fill-individual-varying-indent` [User Option]

この変数は上述のように `fill-individual-paragraphs` の動作を変更する。

`fill-region-as-paragraph` *start end* **&optional** *justify nosqueeze squeeze-after* [Command]

このコマンドはテキストのリージョンを 1 つのパラグラフとみなしてそれをフィルする。そのリージョンが多数のパラグラフから構成されていたらパラグラフ間の空行は削除される。*justify* が非 `nil` ならフィルとともに位置揃えも行う。

nosqueeze が非 `nil` なら、それは改行以外の空白に手を加えずに残すことを意味する。*squeeze-after* が非 `nil` なら、それはリージョン内の位置を指定して、その位置より前にあるスペースについては標準化を行わないことを意味する。

Adaptive Fill モードでは、このコマンドはフィルプレフィクスを選択するためにデフォルトで `fill-context-prefix` を呼び出す。Section 32.13 [Adaptive Fill], page 801 を参照のこと。

`justify-current-line` **&optional** *how eop nosqueeze* [Command]

このコマンドはその行が正確に `fill-column` で終わるように単語間にスペースを挿入する。リターン値は `nil`。

引数 *how* が非 `nil` なら、それは位置揃えスタイルを明示的に指定する。指定できる値は `left`、`right`、`full`、`center`、または `none`。値が `t` なら指定済みの位置揃えスタイル (以下の `current-justification` を参照) にしたがうことを意味する。`nil` は位置揃え `full` と同じ。

eop が非 `nil` なら、それは `current-justification` が full 位置揃えを指定する場合に left 位置揃えだけを行うことを意味する。これはパラグラフ最終行にたいして使用される。パラグラフ全体が full 位置揃えだったとしても最終行は full 位置揃えであるべきではない。

nosqueeze が非 `nil` なら、それは内部のスペースを変更しないことを意味する。

`default-justification` [User Option]

この変数の値は位置揃えに使用するスタイルをテキストプロパティで指定しないテキストにたいするスタイルを指定する。可能な値は `left`、`right`、`full`、`center`、または `none`。デフォルト値は `left`。

`current-justification` [Function]
この関数はポイント周辺のフィルに使用するための適正な位置揃えスタイルをリターンする。

これはポイント位置のテキストプロパティ`justification`の値、そのようなテキストプロパティが存在しなければ変数`default-justification`の値をリターンする。しかし “位置揃えなし” なら、`none`ではなく`nil`をリターンする。

`sentence-end-double-space` [User Option]
この変数が非`nil`ならピリオドの後の単一のスペースをセンテンスの終わりとみなさず、フィル関数はそのような箇所でのラインブレークを行わない。

`sentence-end-without-period` [User Option]
この変数が非`nil`なら、ピリオドなしでセンテンスは終了できる。これはたとえばピリオドなしの 2 連スペースでセンテンスが終わるタイ語などに使用される。

`sentence-end-without-space` [User Option]
この変数が非`nil`なら、それは後にスペースをともなうことなくセンテンスを終了させ得る文字列であること。

`fill-separate-heterogeneous-words-with-space` [User Option]
この変数が非`nil`なら異種 (たとえば英語の CJK) の 2 つの単語は一方が行末にあり、もう一方が次行の先頭にある場合にはフィルでの結合時にスペースで分割される。

`fill-paragraph-function` [Variable]
この変数はパラグラフのフィルをオーバーライドする手段を提供する。この値が非`nil`なら、`fill-paragraph`はその処理を行うためにその関数を呼び出す。その関数が非`nil`値をリターンすると、`fill-paragraph`は処理が終了したとみなして即座にその値をリターンする。

この機能の通常の用途はプログラミング言語のモードにおいてコメントをフィルすることである。通常の方法でその関数がパラグラフをフィルする必要があるなら、以下のようにそれを行うことができる:

```
(let ((fill-paragraph-function nil))
  (fill-paragraph arg))
```

`fill-forward-paragraph-function` [Variable]
この変数は`fill-region`や`fill-paragraph`のようなフィル関数が次のパラグラフへ前方に移動する方法をオーバーライドするための手段を提供する。値は移動するパラグラフの数 nを唯一の引数として呼び出される関数であり、nと実際に移動したパラグラフ数の差をリターンすること。この変数のデフォルト値は`forward-paragraph`。Section “Paragraphs” in *The GNU Emacs Manual* を参照のこと。

`use-hard-newlines` [Variable]
この変数が非`nil`なら、フィル関数はテキストプロパティ`hard`をもつ改行を削除しない。これらの hard 改行、パラグラフのセパレーターとして機能する。Section “Hard and Soft Newlines” in *The GNU Emacs Manual* を参照のこと。

32.12 fill のマージン

`fill-prefix` [User Option]
このバッファーローカル変数が非 `nil`なら、それは通常のテキスト行の先頭に出現して、それらのテキスト行をフィルする際には無視されるべきテキスト文字列を指定する。そのフィルプレフィクスで始まらない行はパラグラフの開始とみなされ、フィルプレフィクスで始まる行はその後にスペースが追加される。フィルプレフィクスで始まりその後に追加のスペースがない行はフィル可能な通常のテキスト行。結果となるフィル済みの行もフィルプレフィクスで開始される。

もしあればフィルプレフィクスは左マージンのスペースの後になる。

`fill-column` [User Option]
このバッファーローカル変数はフィルされる行の最大幅を指定する。値は列数を表す整数であること。Auto Fill モード (Section 32.14 [Auto Filling], page 802 を参照) を含むフィル、位置揃え、センタリングを行うすべてのコマンドがこの変数の影響を受ける。

実際の問題として他の人が読むためのテキストを記述する場合には、`fill-column`を 70 より大きくするべきではない。これにしたがわないと人が快適に読むには行が長くなり過ぎてしまい、下手に記述されたテキストに見えてしまうだろう。

`fill-column`のデフォルト値は 70。

`set-left-margin` *from to margin* [Command]
これは *from*から *to*のテキストの `left-margin`プロパティに値 *margin*をセットする。Auto Fill モードが有効なら、このコマンドは新たなマージンにフィットするようにリージョンの再フィルも行う。

`set-right-margin` *from to margin* [Command]
これは *from*から *to*のテキストの `right-margin`プロパティに値 *margin*をセットする。Auto Fill モードが有効なら、このコマンドは新たなマージンにフィットするようにリージョンの再フィルも行う。

`current-left-margin` [Function]
この関数はポイント周辺をフィルするために使用する、適切な左マージン値をリターンする。値はカレント行開始文字の `left-margin`プロパティの値 (なければ 0) と変数 `left-margin`の値の合計。

`current-fill-column` [Function]
この関数はポイント周辺のテキストをフィルするために使用する、適切なフィル列値をリターンする。値は変数 `fill-column`からポイント後の文字の `right-margin`プロパティの値を減じた値。

`move-to-left-margin` **&optional** *n force* [Command]
この関数はカレント行の左マージンにポイントを移動する。移動先の列は関数 `current-left-margin`により決定される。引数 *n*が非 `nil`なら、まず `move-to-left-margin`は *n*行前方に移動する。

*force*が非 `nil`なら、それは行のインデントが左マージン値とマッチしなければインデントを修正するように指定する。

`delete-to-left-margin` **&optional** *from to* [Function]
この関数は *from*から *to*の間のテキストから左マージンのインデントを取り除く。削除するインデントの量は `current-left-margin`を呼び出すことにより決定される。この関数が非空白文字を削除することはない。*from*と *to*が省略された場合のデフォルトはそのバッファー全体。

`indent-to-left-margin` [Function]
この関数はカレント行の先頭のインデントを変数 `left-margin`に指定された値に調整する (これにより空白文字の挿入や削除が起こるかもしれない)。Paragraph-Indent Text モード内の変数 `indent-line-function`の値はこの関数。

`left-margin` [User Option]
この変数は左マージンの基本列を指定する。Fundamental モードでは、`RET`はこの列にインデントする。この変数は手段の如何を問わずセットされると自動的にバッファーローカルになる。

`fill-nobreak-predicate` [User Option]
この変数はメジャーモードにたいして、特定の箇所で行ブレークしないように指定する手段を提供する。値は関数のリストであること。フィルがバッファー内の特定箇所で行ブレークすると判断されるときは、常にその箇所にポイントを置いた状態でこれらの関数を引数なしで呼び出す。これらの関数のいずれかが非 `nil`をリターンすると、その行のその箇所では行ブレークしない。

32.13 Adaptive Fill モード

Adaptive Fill モードが有効なとき、Emacs は事前定義された値を使用するのではなく、フィルされる各パラグラフのテキストから自動的にフィルプレフィクスを決定します。Section 32.11 [Filling], page 797 と Section 32.14 [Auto Filling], page 802 で説明されているように、このフィルプレフィクスはフィルの間にそのパラグラフの 2 行目以降の行頭に挿入されます。

`adaptive-fill-mode` [User Option]
この変数が非 `nil`なら Adaptive Fill モードは有効。デフォルトは `t`。

`fill-context-prefix` *from to* [Function]
この関数は Adaptive Fill モード実装の肝である。これは *from*から *to*、通常はパラグラフの開始から終了にあるテキストにもとづいてフィルプレフィクスを選択する。これは以下で説明する変数にもとづき、そのパラグラフの最初の 2 行を調べることによりこれを行う。

この関数は通常は文字列としてフィルプレフィクスをリターンする。しかしこれを行う前に、この関数はそのプレフィクスで始まる行がパラグラフの開始とは見えないだろうか、最終チェックを行う (以降では特に明記しない)。これが発生した場合には、この関数はかわりに `nil`をリターンすることにより異常を通知する。

以下は `fill-context-prefix`が行う詳細:

1. 1 行目からフィルプレフィクス候補を取得するために、(もしあれば) まず `adaptive-fill-function`内の関数、次に `adaptive-fill-regexp` (以下参照) の正規表現を試みる。これらの非 `nil`の最初の結果、いずれも `nil`なら空文字列が 1 行目の候補となる。
2. そのパラグラフが 1 行だけなら、関数は見つかったプレフィクス候補の妥当性をテストする。その後でこの関数はそれが妥当ならその候補を、それ以外はスペース文字列をリターンする (以下の `adaptive-fill-first-line-regexp`の説明を参照)。
3. すでにそのパラグラフが 2 行以上なら、この関数は次に 1 行目にたいして行なったのとまったく同じ方法で 2 行目でプレフィクス候補を探す。見つからなければ `nil`をリターンする。

4. ここでこの関数は発見的手法により 2 つのプレフィクス候補を比較する。2 行目の候補の非空白文字の並びが 1 行目の候補と同じなら、この関数は 2 行目の候補をリターンする。それ以外では 2 つの候補に共通するもっとも長い先頭の部分文字列 (これは空文字列かもしれない) をリターンする。

`adaptive-fill-regexp` [User Option]
Adaptive Fill モードは、(もしあれば) 行の左マージン空白文字の後から開始されるテキストにたいしてこの正規表現をマッチする。マッチする文字列がその行のフィルプレフィクス候補。

デフォルト値は空白文字と特定の句読点文字が混在した文字列にマッチする。

`adaptive-fill-first-line-regexp` [User Option]
この正規表現は 1 行だけのパラグラフに使用され、1 つの可能なフィルプレフィクス候補の追加の妥当性評価として機能する。その候補はこの正規表現にマッチするか、`comment-start-skip`にマッチしなければならない。マッチしなければ `fill-context-prefix`はその候補を同じ幅のスペース文字列に置き換える。

この変数のデフォルト値は `"\\`[\t]*\\'"`であり、これは空白文字列だけにマッチする。このデフォルトの効果は 1 行パラグラフで見つかったフィルプレフィクスが、常に純粋な空白文字となるよう強制することである。

`adaptive-fill-function` [User Option]
この変数に関数をセットすることにより、自動的なフィルプレフィクス選択にたいしてより複雑な方法を指定することが可能になる。その関数は、(もしあれば) 行の左マージンの後のポイントで呼び出され、かつポイントを保たなければならない。その関数はその行のフィルプレフィクス、またはプレフィクスの判断に失敗したことを意味する `nil`のいずれかをリターンすること。

32.14 オート fill

Auto Fill モードはテキスト挿入とともに自動的に行をフィルするマイナーモードです。Section “Auto Fill” in *The GNU Emacs Manual* を参照してください。このセクションでは Auto Fill モードにより使用される変数をいくつか説明します。既存テキストを明示的にフィルしたり位置揃えすることができる関数の説明は Section 32.11 [Filling], page 797 を参照してください。

Auto Fill モードではテキスの一部を再フィルするためにマージンや位置揃えを変更する関数も利用できます。Section 32.12 [Margins], page 800 を参照してください。

`auto-fill-function` [Variable]
このバッファーローカル変数の値はテーブル `auto-fill-chars` (以下参照) からの文字の自己挿入後に呼び出される関数 (引数なし) であること。`nil`でもよく、その場合は特に何もしない。

Auto Fill モードが有効なら `auto-fill-function`の値は `do-auto-fill`。これは行ブレークにたいする通常のストラテジーを実装することを唯一の目的とする関数である。

`normal-auto-fill-function` [Variable]
この変数は Auto Fill がオンのときは `auto-fill-function`にたいして使用する関数を指定する。Auto Fill の動作方法を変更するためにメジャーモードはこの変数にバッファーローカル値をセットできる。

`auto-fill-chars` [Variable]
文字が自己挿入された際に `auto-fill-function`を呼び出す文字からなる文字テーブル (ほとんどの言語環境においてはスペースと改行)。

`comment-auto-fill-only-comments` [User Option]
この変数が非 `nil`なら行のフィルは自動的にコメントだけになる。より正確にはカレントバッファーにたいしてコメント構文が定義されていれば、コメントの外側の自己挿入文字が`auto-fill-function`を呼び出さないことを意味する。

32.15 テキストのソート

このセクションで説明するソート関数は、すべてバッファー内のテキストを再配置します。これはリスト要素を再配置する `sort`関数とは対照的です (see Section 5.6.3 [Rearrangement], page 84)。これらの関数がリターンする値に意味はありません。

`sort-subr` *reverse nextrecfun endrecfun* **&optional** *startkeyfun endkeyfun predicate* [Function]
この関数はバッファーをレコードに細分してそれらをソートする一般的なテキストソートルーチン。このセクションのコマンドのほとんどは、この関数を使用する。

`sort-subr`が機能する方法を理解するためには、バッファーのアクセス可能範囲をソートレコード *(sort records)* と呼ばれる分離された断片に分割すると考えればよい。レコードは連続、あるいは非連続かもしれないがオーバーラップしてはならない。各ソートレコードの一部 (全体かもしれない) はソートキーとして指定される。これらソートキーによるソートによりレコードは再配置される。

レコードは通常はソートキー昇順で再配置される。`sort-subr`の 1 つ目の引数 *reverse*が非`nil`ならレコードはソートキー降順にソートされて再配置される。

`sort-subr`にたいする以下の 4 つの引数は、ソートレコード間でポイントを移動するために呼び出される。これらは `sort-subr`内で頻繁に呼び出される。

1. *nextrecfun*はレコード終端のポイントで呼び出される。この関数は次のレコードの先頭にポイントを移動する。`sort-subr`が呼び出された際には、ポイント位置が 1 つ目のレコードの開始とみなされる。したがって `sort-subr`を呼び出す前は、通常はそのバッファーの先頭にポイントを移動すること。

   この関数はバッファー終端にポイントを残すことにより、それ以上のソートレコードがないことを示すことができるできる。

2. *endrecfun*はレコード内にあるポイントで呼び出される。これはレコード終端にポイントを移動する。

3. *startkeyfun*はポイントをレコード先頭からソートキー先頭に移動する。この引数はオプションで、省略された場合はレコード全体がソートキーとなる。もし与えられた場合には、その関数はソートキーとして使用する非 `nil`値、または `nil` (ソートキーはそのバッファー内のポイント位置から始まることを示す) のいずれかをリターンすること。後者の場合にはソートキー終端を見るけるために *endkeyfun*が呼び出される。

4. *endkeyfun*はソートキー先頭からソートキー終端にポイントを移動するために呼び出される。引数はオプション。*startkeyfun*が `nil`をリターンし、かつこの引数が省略 (または `nil`) の場合には、そのソートキーはレコード終端まで拡張される。*startkeyfun*が非 `nil`値をリターンした場合には *endkeyfun*は不要。

引数 *predicate*はキーの比較に使用する関数。この関数は引数として比較する 2 つのキーを受け取り、1 つ目のキーが 2 つ目のキーよりソート順で前なら非 `nil`をリターンすること。引数のキーが正確に何であるかは *startkeyfun*と *endkeyfun*がリターンする値に依存する。*predicate*が省略または `nil`の場合のデフォルトはキーが数値なら`<`、キーがコンスセル (`car`がキーのバッ

ファー位置の開始で cdrが終了) なら compare-buffer-substrings、それ以外なら string< (キーを文字列とみなす)。

sort-subrの例として以下は sort-lines関数の完全な定義である:

```
;; ドキュメント文字列の冒頭 2 行は
;; ユーザー閲覧時には 1 行となることに注意
(defun sort-lines (reverse beg end)
  "リージョン内の行をアルファベット順にソート;\
 引数は降順を意味する
プログラムから呼び出す場合は、以下の 3 つの引数がある:
REVERSE(非 nil は逆順の意)、\
および BEG と END(ソートするリージョン)
変数`sort-fold-case'は英字\
大文字小文字の違いが
ソート順に影響するかどうかを決定する"
  (interactive "P\nr")
  (save-excursion
    (save-restriction
      (narrow-to-region beg end)
      (goto-char (point-min))
      (let ((inhibit-field-text-motion t))
        (sort-subr reverse 'forward-line 'end-of-line)))))
```

ここで forward-lineは次のレコードの先頭にポイントを移動して、end-of-lineはレコードの終端にポイントを移動する。レコード全体をソートキーとするので引数 *startkeyfun*と *endkeyfun*は渡していない。

sort-paragraphsはほとんど同じだが、sort-subrの呼び出しが以下のようになる:

```
(sort-subr reverse
           (lambda ()
             (while (and (not (eobp))
                         (looking-at paragraph-separate))
               (forward-line 1)))
           'forward-paragraph)
```

ソートレコード内を指す任意のマーカーは、sort-subrリターン後は無意味なマーカー位置のまま取り残される。

`sort-fold-case` [User Option]

この変数が非 nilなら、sort-subrとその他のバッファーソート関数は文字列比較時に case(大文字小文字) の違いを無視する。

`sort-regexp-fields` *reverse record-regexp key-regexp start end* [Command]

このコマンドは *start*から *end*の間のリージョンを、*record-regexp*と *key-regexp*で指定されたようにアルファベット順にソートする。*reverse*が負の整数なら逆順にソートする。

アルファベット順のソートとは 2 つのソートキーにたいして、それぞれの 1 つ目の文字同士、2 つ目の文字同士、... のように比較することにより、キーを比較することを意味する。文字が一致しなければ、それはソートキーが不等なことを意味する。最初の不一致箇所で文字が小さいソートキーが小さいソートキーとなる。個別の文字は Emacs 文字セット内の文字コードの数値に応じて比較される。

引数 *record-regexp*の値はバッファーをソートレコードに分割する方法を指定する。各レコードの終端で、この正規表現にたいする検索は完了して、これにマッチするテキストが次のレコードとして採用される。たとえば改行の前に少なくとも 1 つの文字がある行にマッチする正規表現 '`^.+$`'は、そのような行をソートレコードとするだろう。正規表現の構文と意味については Section 34.3 [Regular Expressions], page 883 を参照のこと。

引数 *key-regexp*の値は各レコードのどの部分がソートキーかを指定する。*key-regexp*はレコード全体、またはその一部にマッチすることができる。後者の場合にはレコードの残りの部分はソート順に影響しないが、レコードが新たな位置に移動される際はともに移動される。

引数 *key-regexp*は *record-regexp*の部分式 (subexpression)、またはその正規表現自体にマッチしたテキストを参照できる。

*key-regexp*には以下を指定できる:

'`\`*digit*' *record-regexp*内で *digit*番目のカッコ '`\(...\)`'でグループ化によりマッチしたテキストがソートキーになる。

'`\&`' レコード全体がソートキーとなる。

正規表現 `sort-regexp-fields`は、そのレコード内で正規表現にたいするマッチを検索する。そのようなマッチがあればそれがソートキー。レコード内に *key-regexp*にたいするマッチがなければそのレコードは無視されて、そのバッファー内でのレコードの位置は変更されないことを意味する (他のレコードがそのレコードを移動するかもしれない)。

たとえばリージョン内のすべての行にたいして、最初の単語が文字 '`f`'で始まる行をソートすることを目論むなら、*record-regexp*を '`^.*$`'、*key-regexp*を '`\<f\w*\>`'にセットすること。結果は以下のような式になる

```
(sort-regexp-fields nil "^.*$" "\\<f\\w*\\>"
                    (region-beginning)
                    (region-end))
```

`sort-regexp-fields`をインタラクティブに呼び出した場合にはミニバッファー内で *record-regexp*と *key-regexp*の入力を求める。

`sort-lines` *reverse start end* [Command]
このコマンドは *start*と *end*の間のリージョン内の行をアルファベット順にソートする。*reverse*が非 `nil`なら逆順にソートする。

`sort-paragraphs` *reverse start end* [Command]
このコマンドは *start*と *end*の間のリージョン内のパラグラフをアルファベット順にソートする。*reverse*が非 `nil`なら逆順にソートする。

`sort-pages` *reverse start end* [Command]
このコマンドは *start*と *end*の間のリージョン内のページをアルファベット順にソートする。*reverse*が非 `nil`なら逆順にソートする。

`sort-fields` *field start end* [Command]
このコマンドは *start*と *end*の間のリージョン内の行にたいして、各行の *field*番目のフィールドをアルファベット順に比較することに行をソートする。*field*は空白文字により区切られて、1 から数えられる。*field*が負なら行の終端から −*field*番目のフィールドでソートする。このコマンドはテーブルのソートに有用。

`sort-numeric-fields` *field start end* [Command]
このコマンドは *start*と *end*の間のリージョン内の行にたいして、各行の *field*番目のフィールドを数値的に比較することにより行をソートする。*field*は空白文字により区切られて、1 から数えられる。リージョン内の各行の指定されたフィールドは数字を含んでいなければならない。0 で始まる数字は 8 進数、'`0x`'で始まる数字は 16 進数として扱われる。

*field*が負なら行の終端から −*field*番目のフィールドでソートする。このコマンドはテーブルのソートに有用。

`sort-numeric-base` [User Option]
この変数は `sort-numeric-fields`にたいして数字を解析するための基本基数を指定する。

`sort-columns` *reverse* **&optional** *beg end* [Command]
このコマンドは *beg*と *end*の間にある行にたいして特定の列範囲をアルファベット順に比較することによりソートする。*beg*と *end*の列位置はソートが行われる列範囲にバインドされる。

*reverse*が非 `nil`なら逆順にソートする。

このコマンドが通常と異なるのは、位置 *beg*を含む行全体と位置 *end*を含む行全体がソートされるリージョンに含まれることである。

タブは指定された列に分割される可能性があるので、`sort-columns`はタブを含むテキストを受け付けないことに注意。ソート前に *M-x untabify*を使用してタブをスペースに変換すること。

可能ならユーティリティプログラム `sort`を呼び出すことにより、このコマンドは実際に機能する。

32.16 列を数える

列関数は文字位置 (バッファー先頭から数えた文字数) と列位置 (行先頭から数えたスクリーン文字数) を変換する関数です。

これら列関数はスクリーン上占める列数に応じて各文字を数えます。これはコントロール文字は `ctl-arrow`の値に応じて 2 列または 4 列を、タブは `tab-width`の値と、タブが始まる列の位置に依存する列数を占めるものとして数えられることを意味します。Section 39.22.1 [Usual Display], page 1078 を参照してください。

列数計算はウィンドウ幅と水平スクロール量を無視します。結果として列値は任意に大きくなる可能性があります。最初 (または左端) の列は 0 と数えられます。列値は不可視性を別としてオーバーレイとテキストプロパティを無視します。

`current-column` [Function]
この関数は左マージンを 0 として列単位で数えたポイントの水平位置をリターンする。列の位置はカレント行の開始からポイントまでの間の文字の表示上の表現すべての幅の和。

`move-to-column` *column* **&optional** *force* [Command]
この関数はカレント行の *column*にポイントを移動する。*column*の計算には行の開始からポイントまでの文字の表示上の表現の幅が考慮される。

インタラクティブに呼び出された際には、*column*はプレフィクス数引数の値。*column*が整数でなければエラーがシグナルされる。

列 *column*がタブのような複数列を占める文字の中間にあるために列を移動することが不可能な場合には、ポイントはその文字の終端に移動される。しかし *force*が非 `nil`、かつ *column*がタブの中間にあるなら、(`indent-tabs-mode`が `nil`なら)`move-to-column`はタブをスペー

スに変換するか、(それ以外なら) その前に十分なスペースを挿入するので正確に列 *column*に移動することができる。それ以外の複数列文字については分割する手段がないので、*force*指定に関わらず異常を引き起こす恐れがある。

その行が列 *column*に達するほど長くない場合にも引数 *force*は効果をもつ。*column*が `t`ならその列に達するよう行端に空白を追加することを意味する。

リターン値は実際に移動した列番号。

32.17 インデント

インデント関数は行の先頭にある空白文字の調査、移動、変更に使用されます。行の他の箇所にある空白文字を変更できる関数もいくつかあります。列とインデントは左マージンを 0 として数えられます。

32.17.1 インデント用のプリミティブ

このセクションではインデントのカウントと挿入に使用されるプリミティブ関数について説明します。以降のセクションの関数はこれらのプリミティブを使用します。関連する関数については Section 39.10 [Size of Displayed Text], page 1007 を参照してください。

`current-indentation` [Function]
この関数はカレント行のインデント、すなわち最初の非ブランク文字の水平位置をリターンする。行のコンテンツ全体がブランクなら、それは行終端の水平位置である。

`indent-to` *column* **&optional** *minimum* [Command]
この関数はポイントから *column*に達するまでタブとスペースでインデントを行う。*minimum*が指定されて、かつそれが非 `nil`なら、たとえ *column*を超えることが要求される場合であっても、少なくともその個数のスペースが挿入される。それ以外ではポイントがすでに *column*を超える場合には、この関数は何も行わない。値は挿入されたインデントの終端列。

挿入される空白文字は周囲のテキスト (通常は先行するテキストのみ) のテキストプロパティを継承する。Section 32.19.6 [Sticky Properties], page 827 を参照のこと。

`indent-tabs-mode` [User Option]
この変数が非 `nil`なら、インデント関数はスペースと同じようにタブを挿入でき、それ以外ではスペースだけを挿入できる。この変数はセットすることにより自動的にカレントバッファー内でバッファーローカルになる。

32.17.2 メジャーモードが制御するインデント

すべてのメジャーモードにとって重要な関数は、編集対象の言語にたいして正しくインデントを行うように `TAB`キーをカスタマイズします。このセクションでは `TAB`キーのメカニズムと、それを制御する方法について説明します。このセクションの関数は予期せぬ値をリターンします。

`indent-for-tab-command` **&optional** *rigid* [Command]
これはほとんどの編集用モードで `TAB`にバインドされるコマンド。これの通常の動作はカレント行のインデントだが、かわりにタブ文字の挿入やリージョンのインデントを行うこともできる。

これは以下のことを行う:

- まず Transient Mark モードが有効かどうか、そしてリージョンがアクティブかどうかをチェックする。もしそうならリージョン内のテキストすべてをインデントするために `indent-region`を呼び出す (Section 32.17.3 [Region Indent], page 809 を参照)。

- それ以外なら `indent-line-function`内のインデント関数が `indent-to-left-margin`の場合、または変数`tab-always-indent`が挿入する文字としてタブ文字を指定する場合 (以下参照) にはタブ文字を挿入する。
- それ以外ならカレント行をインデントする。これは `indent-line-function`内の関数を呼び出すことにより行われる。その行がすでにインデント済みで、かつ `tab-always-indent`の値が `complete`(以下参照) ならポイント位置のテキストの補完を試みる。

*rigid*が非 `nil` (インタラクティブな場合はプレフィクス引数) なら、このコマンドが行をインデントした後、あるいはタブを挿入後に新たなインデントを反映するために、このコマンドはカレント行先頭にあるバランスされた式全体も厳正にインデントする。この引数はコマンドがリージョンをインデントする場合は無視される。

`indent-line-function` [Variable]
この変数の値はカレント行をインデントするために `indent-for-tab-command`、およびその他種々のインデントコマンドにより使用される関数。これは通常はメジャーモードにより割り当てられ、たとえばLisp モードはこれを `lisp-indent-line`、C モードは `c-indent-line`のようにセットする。デフォルト値は `indent-relative`。Section 23.7 [Auto-Indentation], page 508 を参照のこと。

`indent-according-to-mode` [Command]
このコマンドはカレントのメジャーモードに適した方法でカレント行をインデントするために `indent-line-function`内の関数を呼び出す。

`newline-and-indent` [Command]
この関数は改行を挿入後に、メジャーモードに応じて新たな行 (挿入した改行の次の行) をインデントする。これは `indent-according-to-mode`を呼び出すことによりインデントを行う。

`reindent-then-newline-and-indent` [Command]
このコマンドはカレント行の再インデント、ポイント位置への改行の挿入、その後に新たな行 (挿入した改行の次の行) のインデントを行う。これは `indent-according-to-mode`を呼び出すことにより両方の行をインデントする。

`tab-always-indent` [User Option]
この変数は `TAB` (`indent-for-tab-command`) コマンドの挙動のカスタマイズに使用できる。値が `t`(デフォルト) ならコマンドは通常はカレント行だけをインデントする。値が `nil`ならコマンドはポイントが左マージン、またはその行のインデント内ににあるときのみカレント行をインデントして、それ以外はタブ文字を挿入する。値が `complete`ならコマンドはまずカレント行のインデントを試みて、その行がすでにインデント済みならポイント位置のテキストを補完するために `completion-at-point`を呼び出す (Section 20.6.8 [Completion in Buffers], page 365 を参照)。

いくつかのメジャーモードでは、異なるメジャーモードに所属する構文をもつ埋め込みのテキストリージョンをサポートする必要があります。これらの例にはドキュメントとソースコード断片の組み合わせである文芸的プログラミング *(literate programming)* のソースファイル、Python や JS のコード断片を含んだ Yacc/Bison プログラムが含まれます。埋め込みチャンクをを正しくインデントするためには、主モードがインデントをガイドする何らかのコンテキストを提供しつつ、他のモードのインデントエンジン (JS では `js-indent-line`、Python では `python-indent-line`の呼び出し) にインデントを委譲する必要があります。メジャーモードはインデントコードで `widen`の呼び出しを避けて `prog-first-column`にしたがうべきです。

`prog-indentation-context` [Variable]
この変数が非`nil`なら副モードのインデントエンジンにたいして上位モードが提供するインデントのコンテキストを保持する。値は`(first-column . rest)`という形式のリストであること。リストのメンバーは以下の意味をもつ:

first-column
トップレベルの構文にたいして使用する列。これは副モードが使用するトップレベルの列のデフォルト値 (通常は 0) を置き換える。

rest 現在のところ使用しない。

以下は他のメジャーモードの副モードとしての呼び出しをサポートする際に、そのメジャーモードのインデントエンジンが使用するべき便宜用の関数です。

`prog-first-column` [Function]
トップレベルのプログラム上のコンテキストの列値にたいしてリテラル値 (通常は 0) を使用するかわりにこの関数を呼び出す。関数の値はトップレベルに使用する列数。上位モードの影響下になければ関数は 0 をリターンする。

32.17.3 リージョン全体のインデント

このセクションではリージョン内すべての行をインデントするコマンドを説明します。これらは予期せぬ値をリターンします。

`indent-region` *start end* **&optional** *to-column* [Command]
このコマンドは *start* (含む) から *end* (含まず) で始まる非ブランク行すべてをインデントする。*to-column*が`nil`なら`indent-region`はカレントモードのインデント関数、すなわち`indent-line-function`の値を呼び出すことにより非ブランク行すべてをインデントする。

*to-column*が非`nil`なら、それはインデントの列数を指定する整数であること。その場合には、この関数は空白文字を追加か削除することにより正確にその量のインデントを各行に与える。

フィルプレフィクスがある場合には、`indent-region`はそのフィルプレフィクスで開始されるように各行をインデントする。

`indent-region-function` [Variable]
この変数の値はショートカットとして`indent-region`により使用されるかもしれない関数。その関数はリージョンの開始と終了という 2 つの引数を受け取ること。その関数はリージョンの行を 1 行ずつインデントするときと同じような結果を生成するようにデザインするべきだが、おそらくより高速になるであろう。

値が`nil`ならショートカットは存在せず`indent-region`は実際に 1 行ずつ機能する。

ショートカット関数は`indent-line-function`が関数定義先頭をスキャンしなければならない C モードや Lisp モードのようなモードにたいして有用であり、それを各行に適用するためには行数の 2 乗に比例する時間を要するだろう。ショートカットは各行のインデントとともに移動してスキャン情報を更新でき、それは線形時間である。行を個別にインデントするのが高速なモードではショートカットの必要性はない。

引数 *to-column*が非`nil`の`indent-region`では意味は異なり、この変数は使用しない。

`indent-rigidly` *start end count* [Command]
この関数は *start* (含む) から *end* (含まず) までのすべての行を横に *count*列インデントする。これは影響を受けるリージョンの外観を保ち、それを厳密な単位として移動する。

これはインデントされていないテキストリージョンのインデントだけでなく、フォーマット済みコードのリージョンにたいするインデントにも有用。たとえば *count*が 3 なら、このコマンドは指定されたリージョン内で始まるすべての行のインデントに 3 を追加する。

プレフィクス引数なしでインタラクティブに呼び出された場合には、このコマンドはインデントを厳密に調整するために Transient Mark モードを呼び出す。Section “Indentation Commands” in *The GNU Emacs Manual* を参照のこと。

`indent-code-rigidly` *start end columns* **&optional** *nochange-regexp* [Command]

これは `indent-rigidly`と似ているが文字列やコメントで始まる行を変更しない点が異なる。

加えて (*nochange-regexp*が非 `nil`なら) *nochange-regexp*が行先頭にマッチする場合にはその行を変更しない。

32.17.4 前行に相対的なインデント

このセクションでは前の行のコンテンツにもとづいてカレント行をインデントするコマンドを 2 つ説明します。

`indent-relative` **&optional** *unindented-ok* [Command]

このコマンドは前の非ブランク行の次のインデントポイント *(indent point)* と同じ列に拡張されるように、ポイント位置に空白文字を挿入する。インデントポイントとは後に空白文字をともなった非空白文字。次のインデントポイントはポイントのカレント列より大きい、最初のインデントポイントになる。たとえばポイントがテキスト行の最初の非ブランク文字の下と左にある場合には、空白文字を挿入してその列に移動する。

前の非ブランク行に次のインデントポイントがない (列の位置が十分大きくない) 場合には、(*unindented-ok*が非 `nil`なら) 何もしないか、あるいは `tab-to-tab-stop`を呼び出す。したがってポイントが短いテキスト行の最後の列の下と右にある場合には、このコマンドは通常は空白文字を挿入することにより次のタブストップにポイントを移動する。

`indent-relative`のリターン値は予測できない。

以下の例ではポイントは 2 行目の先頭にある:

```
                This line is indented twelve spaces.
     ⋆The quick brown fox jumped.
```

式 `(indent-relative nil)`の評価により以下が生成される:

```
                This line is indented twelve spaces.
                ⋆The quick brown fox jumped.
```

次の例ではポイントは ‘`jumped`’の ‘`m`’と ‘`p`’の間にある:

```
                This line is indented twelve spaces.
     The quick brown fox jum⋆ped.
```

式 `(indent-relative nil)`の評価により以下が生成される:

```
                This line is indented twelve spaces.
     The quick brown fox jum  ⋆ped.
```

`indent-relative-first-indent-point` [Command]

このコマンドは引数 *first-only*に `t`を指定して `indent-relative`を呼び出すことにより、前の非ブランク行に倣ってカレント行をインデントする。リターン値は予測できない。

カレント列より先のインデントポイントが前の非ブランク行に存在しなければこのコマンドは何もしない。

32.17.5 調整可能なタブストップ

このセクションではユーザー指定のタブストップ (tab stops) と、それらの使用やセットするメカニズムについて説明します。“タブストップ” という名前はタイプライターのタブストップと機能が類似しているため使用されています。この機能は次のタブストップ列に到達するために、適切な数のスペースとタブを挿入することにより機能します。これはバッファー内のタブ文字の表示に影響を与えません (Section 39.22.1 [Usual Display], page 1078 を参照)。Text モードのような少数のメジャーモードだけが、TAB文字を入力としてこのタブストップ機能を使用することに注意してください。Section “Tab Stops” in *The GNU Emacs Manual* を参照してください。

`tab-to-tab-stop` [Command]
: このコマンドは`tab-stop-list`により定義される次のタブストップ列までポイント前にスペースかタブを挿入する。

`tab-stop-list` [User Option]
: この変数は`tab-to-tab-stop`により使用されるタブストップ列を定義する。これは`nil`、もしくは増加 (均等に増加する必要はない) していく整数のリストであること。このリストは暗黙に、最後の要素と最後から 2 番目の要素の間隔 (またはリストの要素が 2 未満なら`tab-width`) を繰り返すことにより無限に拡張される。値`nil`は列`tab-width`ごとにタブストップすることを意味する。

  インタラクティブにタブストップの位置を編集するには *`M-x edit-tab-stops`*を使用すればよい。

32.17.6 インデントにもとづくモーションコマンド

以下は主にインタラクティブに使用されるコマンドであり、テキスト内のインデントにもとづいて動作します。

`back-to-indentation` [Command]
: このコマンドはカレント行 (ポイントのある行のこと) の最初の非空白文字にポイントを移動する。リターン値は`nil`。

`backward-to-indentation` **&optional** *arg* [Command]
: このコマンドは後方へ *arg*行ポイントを移動した後に、その行の最初の非ブランク文字にポイントを移動する。リターン値は`nil`。*arg*が省略または`nil`のときのデフォルトは 1。

`forward-to-indentation` **&optional** *arg* [Command]
: このコマンドは前方へ *arg*行ポイントを移動した後に、その行の最初の非ブランク文字にポイントを移動する。リターン値は`nil`。*arg*が省略または`nil`のときのデフォルトは 1。

32.18 大文字小文字の変更

ここで説明する case(大文字小文字) 変換コマンドはカレントバッファー内のテキストに作用します。文字列と文字の case 変換コマンドは Section 4.9 [Case Conversion], page 67、大文字や小文字に変換する文字やその変換方法のカスタマイズは Section 4.10 [Case Tables], page 69 を参照してください。

`capitalize-region` *start end* [Command]
: この関数は *start*と *end*で定義されるリージョン内のすべての単語を capitalize する。capitalize とは各単語の最初の文字を大文字、残りの文字を小文字に変換することを意味する。この関数は`nil`をリターンする。

リージョンのいずれかの端が単語の中間にある場合には、リージョン内にある部分を単語全体として扱う。

インタラクティブに `capitalize-region`が呼び出された際には、*start*と *end*はポイントとマークになり小さいほうが先になる。

```
---------- Buffer: foo ----------
This is the contents of the 5th foo.
---------- Buffer: foo ----------

(capitalize-region 1 37)
⇒ nil

---------- Buffer: foo ----------
This Is The Contents Of The 5th Foo.
---------- Buffer: foo ----------
```

`downcase-region` *start end* [Command]

この関数は *start*と *end*で定義されるリージョン内のすべての英文字を小文字に変換する。この関数は `nil`をリターンする。

インタラクティブに `downcase-region`が呼び出された際には、*start*と *end*はポイントとマークになり小さいほうが先になる。

`upcase-region` *start end* [Command]

この関数は *start*と *end*で定義されるリージョン内のすべての英文字を大文字に変換する。この関数は `nil`をリターンする。

インタラクティブに `upcase-region`が呼び出された際には、*start*と *end*はポイントとマークになり小さいほうが先になる。

`capitalize-word` *count* [Command]

この関数はポイントの後の *count*単語を capitalize して、変換後その後にポイントを移動する。capitalize とは各単語の先頭を大文字、残りを小文字に変換することを意味する。*count*が負なら、この関数は前の −*count*単語を capitalize するがポイントは移動しない。値は `nil`。

ポイントが単語の中間にある場合には、ポイントの前にある単語部分は前方に移動する際は無視される。そして残りの部分が単語全体として扱われる。

インタラクティブに `capitalize-word`が呼び出された際には、*count*に数プレフィクス引数がセットされる。

`downcase-word` *count* [Command]

この関数はポイントの後の *count*単語を小文字に変換して、変換後その後にポイントを移動する。*count*が負なら、この関数は前の −*count*単語を小文字に変換するがポイントは移動しない。値は `nil`。

インタラクティブに `downcase-word`が呼び出された際には、*count*に数プレフィクス引数がセットされる。

`upcase-word` *count* [Command]

この関数はポイントの後の *count*単語を大文字に変換して、変換後その後にポイントを移動する。*count*が負なら、この関数は前の −*count*単語を小文字に変換するがポイントは移動しない。値は `nil`。

インタラクティブに `upcase-word`が呼び出された際には、*count*に数プレフィクス引数がセットされる。

32.19 テキストのプロパティ

バッファーや文字列内の各文字位置は、シンボルにおけるプロパティリスト (Section 5.9 [Property Lists], page 92 を参照) のようにテキストプロパティリスト *(text property list)* をもつことができます。特定の位置の特定の文字に属するプロパティ、たとえばこのセンテンス先頭の文字 '`T`' (訳注: 翻訳前のセンテンスは"The properties belong to a ..."で始まる)、または '`foo`'の最初の '`o`'など、もし同じ文字が異なる 2 箇所に存在する場合には、2 つの文字は一般的に異なるプロパティをもちます。

それぞれのプロパティには名前と値があります。どちらも任意の Lisp オブジェクトをもつことができますが、名前は通常はシンボルです。典型的にはそれぞれのプロパティ名シンボルは特定の目的のために使用されます。たとえばテキストプロパティ`face`は、文字を表示するためのフェイスを指定します (Section 32.19.4 [Special Properties], page 820 を参照)。名前を指定してそれに対応する値を尋ねるのが、このプロパティリストにアクセスするための通常の方法です。

ある文字が`category`プロパティをもつ場合は、それをその文字のプロパティカテゴリー *(property category)* と呼びます。これはシンボルであるべきです。そのシンボルのプロパティはその文字のプロパティにたいしてデフォルトとしての役割をもちます。

文字列とバッファーの間でのテキストのコピーでは、文字とともにそのプロパティが保持されます。これには `substring`、`insert`、`buffer-substring`のようなさまざまな関数が含まれます。

32.19.1 テキストプロパティを調べる

テキストプロパティを調べるもっともシンプルな方法は、特定の文字の特定のプロパティの値を尋ねる方法です。これを行うには `get-text-property`を使用します。ある文字のプロパティリスト全体を取得するには `text-properties-at`を使用します。複数の文字のプロパティを一度に調べる関数については Section 32.19.3 [Property Search], page 817 を参照してください。

以下の関数は文字列とバッファーの両方を処理します。バッファー内の位置は 1 から始まりますが、文字列内の位置は 0 から始まることに留意してください。カレントバッファー以外のバッファーのパースは低速になるかもしれません。

`get-text-property` *pos prop* **&optional** *object* [Function]

この関数は *object* (バッファーか文字列) 内の位置 *pos*の後にある文字のプロパティ *prop*の値をリターンする。引数 *object*はオプションでありデフォルトはカレントバッファー。

厳密な意味で *prop*プロパティは存在しないが、その文字がシンボルのプロパティカテゴリーをもつなら、`get-text-property`はそのシンボルの *prop*プロパティをリターンする。

`get-char-property` *position prop* **&optional** *object* [Function]

この関数は `get-text-property`と似ているが、まずオーバーレイをチェックして次にテキストプロパティをチェックする点が異なる。Section 39.9 [Overlays], page 999 を参照のこと。

引数 *object*は文字列、バッファー、あるいはウィンドウかもしれない。ウィンドウならそのウィンドウ内に表示されているバッファーのテキストプロパティとオーバーレイが使用されるが、そのウィンドウにたいしてアクティブなオーバーレイだけが考慮される。*object*がバッファーなら、そのバッファー内のオーバーレイがまず優先的に考慮されて、その後にテキストプロパティが考慮される。*object*が文字列なら文字列がオーバーレイをもつことは決してないのでテキストプロパティだけが考慮される。

`get-pos-property` *position prop* **&optional** *object* [Function]
この関数は `get-char-property`と似ているが、*position* (すぐ右) にある文字のプロパティのかわりにプロパティの stickiness(粘着性) とオーバーレイの advancement(前向的) のセッティングに注意を払う点が異なる。

`get-char-property-and-overlay` *position prop* **&optional** *object* [Function]
これは `get-char-property`と似ているが、そのプロパティ値が由来するオーバーレイについて追加情報を与える点が異なる。

値は CAR がプロパティ値であるようなコンスセルであり、これは同じ引数により `get-char-property`がリターンするであろう値と同じ。CDR はそのプロパティが見つかった箇所のオーバーレイ、テキストプロパティとして見つかった場合や見つからなかった場合には `nil`。

*position*が *object*の終端なら CAR と CDR の値はどちらも `nil`。

`char-property-alias-alist` [Variable]
この変数はプロパティ名と代替となるプロパティ名リストをマップする alist を保持する。文字があるプロパティにたいして直接値を指定しなければ、順に代替プロパティ名が調べられて最初の非 `nil`値が使用される。この変数は `default-text-properties`より優先されて、この変数より `category`プロパティが優先される。

`text-properties-at` *position* **&optional** *object* [Function]
この関数は文字列かバッファー *object*内の位置 *position*にある文字のプロパティリスト全体をリターンする。*object*が `nil`ならデフォルトはカレントバッファー。

`default-text-properties` [Variable]
この変数はテキストプロパティにたいしてデフォルト値を与えるプロパティリストを保持する。あるプロパティにたいして文字が直接、あるいはカテゴリーシンボルや `char-property-alias-alist`を通じて値を指定しないときは、常にこのリストに格納された値がかわりに使用される。以下は例:

```
(setq default-text-properties '(foo 69)
      char-property-alias-alist nil)
;; 文字 1 は自身のプロパティをもたない
(set-text-properties 1 2 nil)
;; 取得される値はデフォルト値
(get-text-property 1 'foo)
     ⇒ 69
```

32.19.2 テキストプロパティの変更

プロパティを変更するプリミティブは、バッファーや文字列内の指定されたテキスト範囲に適用されます。関数 `set-text-properties` (セクションの最後を参照) は、その範囲内のテキストのプロパティリスト全体をセットします。名前を指定することにより特定のプロパティだけを追加、変更、削除するためにも有用です。

テキストプロパティはバッファー (か文字列) のコンテンツの一部とみなされ、かつスクリーン上でのバッファーの見栄えに影響を与えることができるので、バッファー内のテキストプロパティの変更はすべてバッファーを変更済みとマークします。バッファーテキストプロパティの変更もアンドゥできます (Section 32.9 [Undo], page 793 を参照)。バッファー内の位置は 1 から始まりますが、文字列内の位置は 0 から始まります。

`put-text-property` *start end prop value* **&optional** *object* [Function]
この関数は文字列かバッファー *object*内の *start*と *end*の間のテキストにたいして、プロパティ *prop*に *value*をセットする。*object*が `nil`ならデフォルトはカレントバッファー。

`add-text-properties` *start end props* **&optional** *object* [Function]
この関数は文字列かバッファー *object*内の *start*と *end*の間のテキストにたいして、テキストプロパティを追加またはオーバーライドする。*object*が `nil`ならデフォルトはカレントバッファー。

引数 *props*には追加するプロパティを指定する。これはプロパティリストの形式 (Section 5.9 [Property Lists], page 92 を参照)、つまりプロパティ名と対応する値が交互に出現するような要素を含むリストであること。

関数が実際に何らかのプロパティの値を変更したら `t`、それ以外 (*props*が `nil`、またはプロパティの値がテキスト内のプロパティの値と一致している場合) は `nil`がリターン値となる。

たとえば以下はテキストの範囲に `comment`と `face`のプロパティをセットする例:

```
(add-text-properties start end
                     '(comment t face highlight))
```

`remove-text-properties` *start end props* **&optional** *object* [Function]
この関数は文字列かバッファー *object*内の *start*と *end*の間のテキストから、指定されたテキストプロパティを削除する。*object*が `nil`ならデフォルトはカレントバッファー。

引数 *props*は削除するプロパティを指定する。これはプロパティリストの形式 (Section 5.9 [Property Lists], page 92 を参照)、つまりプロパティ名と対応する値が交互に出現するような要素を含むリストであること。しかし問題となるのは名前であって付随する値は無視される。たとえば `face`プロパティを削除するには以下のようにすればよい。

```
(remove-text-properties start end '(face nil))
```

関数が実際に何らかのプロパティの値を変更したら `t`、それ以外 (*props*が `nil`、または指定されたテキスト内にそれらのプロパティをもつ文字がない場合) は `nil`がリターン値となる。

特定のテキストからすべてのテキストプロパティを削除するには、新たなプロパティリストに `nil`を指定して `set-text-properties`を使用すればよい。

`remove-list-of-text-properties` *start end list-of-properties* [Function]
&optional *object*
`remove-text-properties`と同様だが、*list-of-properties*がプロパティ名と値が交互になったリストではなくプロパティ名だけのリストである点が異なる。

`set-text-properties` *start end props* **&optional** *object* [Function]
この関数は文字列かバッファー *object*内の *start*から *end*の間のテキストにたいするテキストプロパティリストを完全に置き換える。*object*が `nil`ならデフォルトはカレントバッファー。

引数 *props*は新たなプロパティリスト。これはプロパティ名と対応する値が交互となるような要素のリストであること。

`set-text-properties`のリターン後には、指定された範囲内のすべての文字は等しいプロパティをもつ。

*props*が `nil`なら、指定されたテキスト範囲からすべてのプロパティを取り除く効果がある。以下は例:

```
(set-text-properties start end nil)
```

この関数のリターン値を信用してはならない。

`add-face-text-property` *start end face* **&optional** *appendp object* [Function]
この関数は *start*と *end*の間のテキストのテキストプロパティ`face`にフェイス *face*を追加するように動作する。*face*はフェイス名、もしくは anonymous フェイス (anonymous face: 無名フェイス) のような`face`プロパティ(Section 32.19.4 [Special Properties], page 820 を参照) にたいして有効な値であること (Section 39.12 [Faces], page 1011 を参照)。

リージョン内の任意のテキストがすでに非`nil`の`face`プロパティをもつ場合には、それらのフェイスは保たれる。この関数は`face`プロパティに最初の要素 (デフォルト) が *face*、以前に存在していたフェイスが残りの要素であるようなフェイスのリストをセットする。オプション引数 *appendp*が非`nil`なら、*face*はかわりにリストの最後に追加される。フェイスリスト内では、各属性にたいして最初に出現する値が優先されることに注意。

たとえば以下のコードでは *start*と *end*の間のテキストにグリーン斜体のフェイスを割り当てるだろう:

```
(add-face-text-property start end 'italic)
(add-face-text-property start end '(:foreground "red"))
(add-face-text-property start end '(:foreground "green"))
```

オプション引数 *object*が非`nil`なら、それはカレントバッファーではなく動作するバッファーか文字列を指定する。*object*が文字列なら *start*と *end*は 0 基準で文字列内をインデックス付けする。

文字列にテキストプロパティを付するもっとも簡単な方法は`propertize`です:

`propertize` *string* **&rest** *properties* [Function]
この関数はテキストプロパティ*properties*を追加した *string*のコピーをリターンする。これらのプロパティはリターンされる文字列内のすべての文字に適用される。以下は`face`プロパティと`mouse-face`プロパティとともに文字列を構築する例:

```
(propertize "foo" 'face 'italic
            'mouse-face 'bold-italic)
     ⇒ #("foo" 0 3 (mouse-face bold-italic face italic))
```

文字列のさまざまな部分に異なるプロパティを put するには、それぞれの部分を`propertize`で構築して、それらを`concat`で結合すればよい:

```
(concat
 (propertize "foo" 'face 'italic
             'mouse-face 'bold-italic)
 " and "
 (propertize "bar" 'face 'italic
             'mouse-face 'bold-italic))
     ⇒ #("foo and bar"
                 0 3 (face italic mouse-face bold-italic)
                 3 8 nil
                 8 11 (face italic mouse-face bold-italic))
```

プロパティではなくバッファーからテキストをコピーする関数`buffer-substring-no-properties`については Section 32.2 [Buffer Contents], page 778 を参照してください。

バッファーを変更せずにバッファーにテキストプロパティを追加したり削除したければ、その呼び出しを上記の`with-silent-modifications`マクロでラップできます。Section 27.5 [Buffer Modification], page 596 を参照してください。

32.19.3 テキストプロパティの検索関数

テキストプロパティの通常の使用では、ほとんどの場合は複数または多くの連続する文字が同じ値のプロパティをもちます。文字を 1 つずつ調べるプログラムを記述するよりも、同じプロパティ値をもつテキスト塊 (chunks of text) を処理するほうがより高速です。

以下はこれを行うことに使用できる関数です。これらはプロパティ値の比較に `eq`を使用します。すべての関数において *object*のデフォルトはカレントバッファーです。

より良いパフォーマンスのためには、特に単一のプロパティを検索する関数における *limit*引数の使用が重要です。さもないと興味のあるプロパティが変化しない場合に、バッファー終端までのスキャンで長い時間を要するでしょう。

これらの関数はポイントを移動しません。そのかわりに位置 (または `nil`) をリターンします。ポイントは常に文字と文字の間にあることを思い出してください。これらの関数によりリターンされる位置は、異なるプロパティをもつ 2 つの文字の間にあります。

`next-property-change` *pos* **&optional** *object limit* [Function]

この関数は文字列かバッファー *object*内の位置 *pos*から、何らかのテキストプロパティの変化が見つかるまでテキストを前方にスキャンして、変化のあった位置をリターンする。言い換えると *pos*の直後の文字とプロパティが等しくない、*pos*の先にある最初の文字の位置をリターンする。

*limit*が非 `nil`ならスキャンは位置 *limit*で停止する。そのポイントより前にプロパティが変化しなければ、この関数は *limit*をリターンする。

プロパティが *object*終端まで変化せず、かつ *limit*が `nil`なら値は `nil`。値が非 `nil`なら、それは *pos*以上の位置。*limit*が *pos*と等しいときのみ値は *pos*になる。

以下はすべてのプロパティが定数であるようなテキスト塊によりバッファーをスキャンする方法の例:

```
(while (not (eobp))
  (let ((plist (text-properties-at (point)))
        (next-change
         (or (next-property-change (point) (current-buffer))
             (point-max))))
    ポイントから next-changeへテキストを処理...
    (goto-char next-change)))
```

`previous-property-change` *pos* **&optional** *object limit* [Function]

これは `next-property-change`と似ているが、*pos*から前方ではなく後方にスキャンする点が異なる。値が非 `nil`なら、それは *pos*以下の位置。*limit*と *pos*が等しい場合のみ *pos*をリターンする。

`next-single-property-change` *pos prop* **&optional** *object limit* [Function]

この関数はプロパティ *prop*内の変化にたいしてテキストをスキャンして、変化があった位置をリターンする。このスキャンは文字列かバッファー *object*内の位置 *pos*から前方に行われる。言い換えると *pos*の直後の文字とプロパティ *prop*が等しくない、*pos*の先にある最初の文字の位置をリターンする。

*limit*が非 `nil`ならスキャンは位置 *limit*で終了する。そのポイントより前にプロパティの変化がなければ、`next-single-property-change`は *limit*をリターンする。

プロパティが *object*終端まで変化せず、かつ *limit*が `nil`なら値は `nil`。値が非 `nil`なら、それは *pos*以上の位置。*limit*が *pos*と等しいときのみ値は *pos*になる。

`previous-single-property-change` *pos prop* **&optional** *object limit* [Function]
これは `next-single-property-change`と似ているが、*pos*から前方ではなく後方にスキャンする点が異なる。値が非 `nil`なら、それは *pos*以下の位置。*limit*と *pos*が等しい場合のみ *pos*をリターンする。

`next-char-property-change` *pos* **&optional** *limit* [Function]
`next-property-change`と似ているが、これはテキストプロパティと同様にオーバーレイも考慮して、バッファー終端より前に変化が見つからなければ、`nil`ではなくバッファー位置の最大をリターンする点が異なる (この点では `next-property-change`よりも対応するオーバーレイ関数 `next-overlay-change`と似ている)。この関数はカレントバッファーだけを処理するので *object*オペランドは存在しない。これはいずれかの種類のプロパティが変化した、次のアドレスをリターンする。

`previous-char-property-change` *pos* **&optional** *limit* [Function]
これは `next-char-property-change`と似ているが、*pos*から前方ではなく後方へスキャンすること、および変化が見つからなければバッファー位置の最小をリターンする点が異なる。

`next-single-char-property-change` *pos prop* **&optional** *object limit* [Function]
`next-single-property-change`と似ているが、これはテキストプロパティと同様にオーバーレイも考慮して、*object*終端より前に変化が見つからなければ、`nil`ではなく *object*内の有効な位置の最大をリターンする点が異なる。`next-char-property-change`と異なり、この関数は *object*オペランドをもつ。*object*が非バッファーならテキストプロパティだけが考慮される。

`previous-single-char-property-change` *pos prop* **&optional** *object limit* [Function]
これは `next-single-char-property-change`と似ているが、*pos*から前方ではなく後方へスキャンすること、および変化が見つからなければ *object*内の有効な位置の最小をリターンする点が異なる。

`text-property-any` *start end prop value* **&optional** *object* [Function]
この関数は *start*と *end*の間に少なくともプロパティ *prop*に値 *value*をもつ文字が 1 つあれば非 `nil`をリターンする。より正確には、これはそのような最初の文字の位置、それ以外は `nil`をリターンする。

5 つ目のオプション引数 *object*はスキャンする文字列かバッファーを指定する。位置は *object*にたいして相対的。*object*のデフォルトはカレントバッファー。

`text-property-not-all` *start end prop value* **&optional** *object* [Function]
この関数は *start*と *end*の間に少なくともプロパティ *prop*に値 *value*をもたない文字が 1 つあれば非 `nil`をリターンする。より正確には、これはそのような最初の文字の位置、それ以外は `nil`をリターンする。

5 つ目のオプション引数 *object*はスキャンする文字列かバッファーを指定する。位置は *object*にたいして相対的。*object*のデフォルトはカレントバッファー。

`text-property-search-forward` *prop* **&optional** *value predicate not-current* [Function]
テキストプロパティ *prop*に *value*がセットされている次のリージョンを *predicate*にしたがい検索する。

この関数はポイントを移動するという点において search-forwardや類似関数をモデルとするが、match-beginningや類似関数とは異なりマッチを記述する構造をリターンする。

テキストプロパティが見つからなければ、この関数は nilをリターンする。見つかったらマッチしたテキストプロパティをもつリージョンの最後にポイントを移動して prop-match構造をリターンする。

*predicate*は t (equalのシノニム)、nil ("not equal" を意味する)、または 2 つのパラメーター (1 つ目は *value*で 2 つ目は検査するテキストプロパティの値) で呼び出される述語のいずれかを指定できる。

*not-current*の場合には、もしポイントがマッチしたリージョン内にあれば、それをスキップして次のインスタンスを探す。

prop-match構造は prop-match-beginning (マッチ先頭)、prop-match-end (マッチ終端)、prop-match-value (マッチ先頭の *property*の値) というアクセサをもつ。

以下の例では以下のようなバッファー内にいると想像してほしい:

```
This is a bold and here's bolditalic and this is the end.
```

すなわち単語 "bold" は boldフェイス、単語 "italic" は italicフェイスをもつものとする。

まず最初は:

```
(while (setq match (text-property-search-forward 'face 'bold t))
  (push (buffer-substring (prop-match-beginning match)
                          (prop-match-end match))
        words))
```

これは boldフェイスを使用するすべての単語を選択する。

```
(while (setq match (text-property-search-forward 'face nil t))
  (push (buffer-substring (prop-match-beginning match)
                          (prop-match-end match))
        words))
```

これはフェイスプロパティをもたないすべての断片を選択して、結果はリスト'("This is a " "and here's " "and this is the end")'となるだろう (pushを使用しているので逆転される)。

```
(while (setq match (text-property-search-forward 'face nil nil))
  (push (buffer-substring (prop-match-beginning match)
                          (prop-match-end match))
        words))
```

これは faceに何らかがセットされているすべてのリージョンを選択するがプロパティが変化する箇所で分割するので結果は'("bold" "bold" "italic")'になるだろう。

これを使用するかもしれないより現実的な例して URL を表す特定のセクションがあり、それらが shr-urlでタグ付けされているバッファーがあると仮定してみる。

```
(while (setq match (text-property-search-forward 'shr-url nil nil))
  (push (prop-match-value match) urls))
```

これはそれらすべての URL のリストを与えるだろう。

`text-property-search-backward` *prop* **&optional** *value predicate not-current* [Function]

これは text-property-search-backwardと同様だが、後方に検索する。ポイントはマッチしたリージョン終端ではなく先頭に配置される。

32.19.4 特殊な意味をもつプロパティ

以下はビルトインで特別な意味をもつテキストプロパティ名のテーブルです。以降のセクションではフィルとプロパティ継承を制御する特別なプロパティ名をいくつか追加でリストしています。これ以外のすべての名前は特別な意味をもたず自由に使用できます。

注意: プロパティ`composition`、`display`、`invisible`、`intangible`はすべての Emacs コマンドの後に好ましい箇所にポイントを移動させることもできます。Section 21.6 [Adjusting Point], page 387 を参照してください。

`category` ある文字が`category`プロパティをもつ場合には、それをその文字のプロパティカテゴリー *(property category)* と呼ぶ。これはシンボルであること。このシンボルのプロパティはその文字のプロパティのデフォルトとしての役割をもつ。

`face` `face`プロパティはその文字の外観を制御する (Section 39.12 [Faces], page 1011 を参照)。このプロパティの値は以下が可能:

- フェイス名 (シンボルか文字列)。
- anonymous フェイス: `(`*`keyword value ...`*`)`形式のプロパティリスト。*keyword* はそれぞれフェイス属性名、*value*はその属性の値。
- フェイスのリスト。各リスト要素はフェイス名か anonymous フェイスであること。これはリストされた各フェイス属性を集計したフェイスを指定する。このリスト内で最初にあるフェイスがより高い優先度をもつ。
- `(foreground-color .` *`color-name`*`)`または`(background-color .` *`color-name`*`)`という形式のコンスセル。これは`(:foreground` *`color-name`*`)`や`(:background` *`color-name`*`)`と同じようにフォアグラウンドやバックグラウンドを指定する。この形式は後方互換のためだけにサポートされており無視すること。
- `(:filtered` *`filter face-spec`*`)` という形式のコンスセル。これは *face-spec* で与えられたフェイスを指定するが、フェイスを表示に使用する際に *filter*がマッチした場合のみ。*face-spec*には上述した任意のフォームを使用できる。*filter*は`(:window` *`param value`*`)` という形式であること。これはパラメーター *param*が *value*に`eq`であるようなウィンドウにマッチする。変数`face-filters-always-match`が非`nil`なら、すべてのフェイスフィルターがマッチしたとみなす。

Font Lock モード (Section 23.6 [Font Lock Mode], page 496 を参照) はほとんどのバッファーにおいて、コンテキストにもとづき文字の`face`プロパティを動的に更新することにより機能する。

`add-face-text-property`関数は、このプロパティをセットする便利な手段を提供する。Section 32.19.2 [Changing Properties], page 814 を参照のこと。

`font-lock-face`
このプロパティは Font Lock モードが配下にあるテキストに適用すべき`face`プロパティにたいして値を指定する。これは Font Lock モードに使用されるフォント表示手法の 1 つであり、独自のハイライトを実装する特別なモードにたいして有用。Section 23.6.6 [Precalculated Fontification], page 504 を参照のこと。Font Lock モードが無効なら`font-lock-face`に効果はない。

`mouse-face`
このプロパティは文字上または近傍にマウスがあるとき、`face`のかわりに使用される。この目的にたいして “近傍” とは文字とマウス位置の間のすべてのテキストが同じ`mouse-face`プロパティの値をもつことを意味する。

Emacs はテキストサイズ (`:height`、`:weight`、`:slant`) を変更する `mouse-face`プロパティ由来の属性すべてを無視する。これらの属性はハイライトされていないテキストと常に等しい。

`fontified`
このプロパティはそのテキストの表示準備が整っているかどうかを告げる。`nil`なら Emacs の再表示ルーチンはバッファーの該当部分を表示する前に、準備のために `fontification-functions` (Section 39.12.7 [Auto Faces], page 1024 を参照) の中の関数を呼び出す。これはフォントロックのコードの just-in-time により内部的に使用される。

`display` このプロパティはテキストが表示される方法を変更するさまざまな機能をアクティブ化する。たとえばこれによりテキスト外観を縦長 (taller) または縦短 (short) したり、高く (higher) または低く (lower)、太く (wider) または細く (narrower) したり、あるいはイメージに置き換えることができる。Section 39.16 [Display Property], page 1043 を参照のこと。

`help-echo`
テキストが `help-echo`プロパティに文字列をもつ場合には、そのテキスト上にマウスを移動した際には、`substitute-command-keys`を通じて文字列を渡した後に Emacs はエコーエリアかツールチップウィンドウ (Section 39.25 [Tooltips], page 1084 を参照) にその文字列を表示する。

`help-echo`プロパティの値が関数なら、その関数は *window*、*object*、*pos*の 3 つの引数で呼び出されてヘルプ文字列、ヘルプ文字列が存在しなければ `nil`をリターンすること。1 つ目の引数 *window*はそのヘルプが見つかったウィンドウ。2 つ目の引数 *object* は `help-echo`プロパティをもつバッファー、オーバーレイ、または文字列。*pos*引数は以下のとおり:

- *object*がバッファーなら *pos*はそのバッファー内の位置。
- *object*がオーバーレイなら、そのオーバーレイは `help-echo`プロパティをもち *pos* はそのオーバーレイのバッファー内の位置。
- *object*が文字列 (オーバーレイ文字列、または `display`プロパティにより表示された文字列) なら *pos*はその文字列内の位置。

`help-echo`プロパティの値が関数と文字列のいずれでもなければ、それはヘルプ文字列を得るために評価される。

変数 `show-help-function`をセットすることにより、ヘルプテキストが表示される方法を変更できる ([Help display], page 826 を参照)。

この機能はモードライン内、およびその他のアクティブテキストにたいして使用される。

`help-echo-inhibit-substitution`
`help-echo`文字列の最初の文字が非 `nil`の `help-echo-inhibit-substitution` プロパティをもつ場合には、`substitute-command-keys`を通じて渡すことなく `show-help-function`が行うように表示される。

`keymap` `keymap`プロパティはコマンドにたいして追加のキーマップを指定する。このキーマップを適用する際には、マイナーモードキーマップとバッファーのローカルマップの前に、このマップがキー照合のために使用される。Section 22.7 [Active Keymaps], page 429 を参照のこと。プロパティ値がシンボルなら、そのシンボルの関数定義がキーマップとして使用される。

ポイントの前の文字のプロパティの値は、それが非nilで rear-sticky であり、かつポイントの後の文字のプロパティ値が非nilで front-sticky なら適用される (マウスクリックではポイント位置のかわりにクリック位置が使用される)。

local-map
: このプロパティはkeymapと同じように機能するが、これはそのバッファーのローカルマップのかわりに使用するキーマップを指定する点が異なる。ほとんど (もしかするとすべて) の目的にたいしてはkeymapを使用するほうが良いだろう。

syntax-table
: syntax-tableプロパティは特定の文字にたいしてどのシンタックステーブルがオーバーライドするかを告げる。Section 35.4 [Syntax Properties], page 912 を参照のこと。

read-only
: ある文字がプロパティread-onlyをもつなら、その文字の変更は許可されない。これを行おうとするすべてのコマンドはtext-read-onlyエラーを受け取る。プロパティの値が文字列ならその文字列がエラーメッセージとして使用される。

  read-only 文字に隣接する箇所への挿入は、そこに通常のテキストの行うことが stickiness による read-onlyプロパティを継承するならエラーとなる。つまり stickiness を制御することにより read-only テキストに隣接する挿入の権限を制御することができる。Section 32.19.6 [Sticky Properties], page 827 を参照のこと。

  プロパティ変更はバッファー変更とみなされるので、特別なトリック (inhibit-read-onlyを非nilにバインドしてからプロパティを削除する) を知らないかぎり、read-onlyプロパティを取り除くことは不可能。Section 27.7 [Read Only Buffers], page 599 を参照のこと。

inhibit-read-only
: プロパティinhibit-read-onlyをもつ文字はたとえ読み取り専用バッファーでも編集できる。Section 27.7 [Read Only Buffers], page 599 を参照のこと。

invisible
: 非nilの invisibleプロパティにより、スクリーン上で文字を不可視にできる。詳細は Section 39.6 [Invisible Text], page 992 を参照のこと。

intangible
: 連続する文字のグループが非nilの等しい intangibleプロパティをもつなら、それらの文字の間にポイントを置くことは不可能。そのグループ内に前方へポイントの移動を試みると、ポイントは実際にはそのグループの終端に移動する。そのグループ内に後方へポイントの移動を試みると、ポイントは実際にはそのグループの先頭に移動する。

  連続する文字のグループが非nilの等しくない intangibleプロパティをもつなら、それらの文字は個別のグループに属して、各グループは上述のように別のグループとして扱われる。

  変数inhibit-point-motion-hooksが非nil (デフォルト) なら intangibleプロパティは無視される。

  注意せよ: このプロパティは非常に低レベルで処理されて、予想外の方法により多くのコードに影響する。そのため使用に際しては特別な注意を要する。誤った使用方法としては不可視のテキストに intangible プロパティを put するのが一般的な誤りであり、コマンドループは各コマンドの終わりに不可視テキストの外部へポイントを移動するだろうから、これは実際には必要ない。Section 21.6 [Adjusting Point], page 387 を参照のこと。

これらの理由によりこのプロパティは時代遅れであり、かわりに `cursor-intangible` プロパティを使用すること。

`cursor-intangible`
: マイナーモード `cursor-intangible-mode`がオンに切り替えられた際には、再表示の発生直前に非 `nil`の `cursor-intangible`プロパティをもつすべての位置からポイントが移動される。

  変数 `cursor-sensor-inhibit`が非 `nil`なら、`cursor-intangible`プロパティと `cursor-sensor-functions`プロパティ(以下参照) は無視される。

`field`
: 同じ `field`プロパティをもつ連続する文字はフィールドを構成する。`forward-word`や `beginning-of-line`を含むいくつかの移動関数はフィールド境界で移動を停止する。Section 32.19.9 [Fields], page 831 を参照のこと。

`cursor`
: カーソルは通常はカレントバッファー位置にあるオーバーレイ、およびテキストプロパティ文字列の先頭か終端に表示される。文字に非 `nil`の `cursor`テキストプロパティを与えることにより、それら文字列内の任意の望む文字にカーソルを置くことができる。加えて `cursor`プロパティの値が整数なら、それはカーソルがその文字上に表示されるようにオーバーレイまたは `display`プロパティが始まる位置から数えたバッファーの文字位置の数字を指定する。特にある文字の `cursor`プロパティの値が数字 *n*なら、カーソルは範囲 `[`*ovpos*`..`*ovpos*`+`*n*`)`内の任意のバッファー位置にあるその文字上に表示されるだろう。ここで *ovpos*は `overlay-start` (Section 39.9.1 [Managing Overlays], page 1000 を参照) により与えられるオーバーレイ開始位置、またはそのバッファー内で `display`プロパティが始まる位置である。

  言い換えると文字列の非 `nil`値の `cursor`プロパティをもつ文字はカーソルが表示される文字である。このプロパティの値はカーソルを表示するバッファーの位置を告げる。値が整数ならオーバーレイまたは `display`プロパティの始まりから *n*後ろの位置までの間にポイントがあるとき、カーソルはそこに表示される。値がそれ以外の非 `nil`ならポイントが `display`プロパティの先頭、または `overlay-start`の位置だけに表示される。

  バッファーのテキストを網羅するオーバーレイ文字列 (Section 39.9.2 [Overlay Properties], page 1002 を参照) や文字列であるような `display`プロパティがバッファーに多くある場合には、それらの文字列を走査する間にカーソルを置く箇所を Emacs に合図するために、`cursor`プロパティを使用するのはよいアイデアである。これは display やオーバーレイ文字列に “カバー” された何らかのバッファー位置にポイントがある際に、Lisp プログラムやユーザーがカーソルを配置したい箇所でディスプレイエンジンと直接通信する。

`pointer`
: これはそのテキストやイメージ上にマウスポインターがあるときの特定のマウスシェイプを指定する。利用できるポインターシェイプについては Section 29.19 [Pointer Shape], page 746 を参照のこと。

`line-spacing`
: 改行は改行で終わるディスプレイ行の高さを制御するテキストプロパティやオーバーレイプロパティ`line-spacing`をもつことができる。このプロパティ値はデフォルトのフレーム行スペーシングと、バッファーローカル変数 `line-spacing`をオーバーライドする。Section 39.11 [Line Height], page 1010 を参照のこと。

`line-height`
: 改行は改行で終わるディスプレイ行のトータル高さを制御するテキストプロパティ、またはオーバーレイプロパティ`line-height`をもつことができる。Section 39.11 [Line Height], page 1010 を参照のこと。

`wrap-prefix`
: テキストが`wrap-prefix`プロパティをもつなら、それが定義するプレフィクスはテキストラッピング (text wrapping: テキスト折り返し) に由来するすべての継続行の先頭に表示時に追加されるだろう (行が切り詰められた場合には wrap-prefix が使用されることはない)。これは文字列、イメージ (Section 39.16.4 [Other Display Specs], page 1046 を参照)、あるいはディスプレイプロパティ`:width`や`:align-to` (Section 39.16.2 [Specified Space], page 1044 を参照) により指定された空白文字範囲かもしれない。

  wrap-prefix はバッファーローカル変数`wrap-prefix`を使用して、バッファー全体にも指定され得る (が`wrap-prefix`テキストプロパティは`wrap-prefix`変数の値より優先される)。Section 39.3 [Truncation], page 982 を参照のこと。

`line-prefix`
: テキストが`line-prefix`プロパティをもつなら、それが定義するプレフィクスは表示時にすべての非継続行の先頭に追加されるだろう。これは文字列、イメージ (Section 39.16.4 [Other Display Specs], page 1046 を参照)、あるいはディスプレイプロパティ`:width`や`:align-to` (Section 39.16.2 [Specified Space], page 1044 を参照) により指定された空白文字範囲かもしれない。

  line-prefix はバッファーローカル変数`line-prefix`を使用して、バッファー全体にも指定され得る (が`line-prefix`テキストプロパティは`line-prefix`変数の値より優先される)。Section 39.3 [Truncation], page 982 を参照のこと。

`modification-hooks`
: ある文字がプロパティ`modification-hooks`をもつなら、その値は関数のリストであること。その文字の変更により、実際の変更前にそれらの関数すべてが呼び出される。それぞれの関数は、変更されようとするバッファー部分の先頭と終端という 2 つの引数を受け取る。特定の modification フック関数が単一のプリミティブにより変更されつつある複数の文字に出現する場合は、その関数が呼び出される回数を予測することはできない。さらに挿入は既存の文字を変更しないので、このフックは文字の削除、他の文字への置換、またはそれらのテキストプロパティ変更時のみ実行されるだろう。

  他の同類フックとは異なり、Emacs はこれらの関数を呼び出し時に`inhibit-modification-hooks`を非`nil`にバインドしない。関数がバッファーを変更するようなら、バッファー変更による変更フックの実行を防ぐために、この変数を非`nil`にバインドすることを考慮する必要がある。そうでなければ再帰呼び出しに備えなければならない。Section 32.32 [Change Hooks], page 850 を参照のこと。

  オーバーレイも`modification-hooks`プロパティをサポートするが詳細は若干異なる (Section 39.9.2 [Overlay Properties], page 1002 を参照)。

`insert-in-front-hooks`
`insert-behind-hooks`
: あるバッファーへの挿入操作は後続文字の`insert-in-front-hooks`プロパティ、および先行文字の`insert-behind-hooks`プロパティにリストされる関数の呼び出しも行う。これらの関数は挿入されるテキストの先頭と終端という 2 つの引数を受け取る。関数は優先される実際の挿入が行われた後に呼び出される。

これらの関数を呼び出す際には inhibit-modification-hooksは非 nilにバインドされる。関数がバッファーを変更する場合には、これらの変更にたいして変更フックが実行されるように、inhibit-modification-hooksを nilにバインドしたいと思うかもしれない。しかしこれを行うことによって、あなたの変更フックが再帰的に呼び出されるかもしれないので、確実にそれに備えること。

バッファー内のテキスト変更時にに呼び出される他のフックについては Section 32.32 [Change Hooks], page 850 も参照のこと。

`point-entered`
`point-left`

スペシャルプロパティpoint-enteredと point-leftはポイント移動をリポートするフック関数を記録する。ポイントを移動するたびに Emacs は以下の 2 つのプロパティ値を比較する:

- 古い位置の後の文字の point-leftプロパティ。
- 新しい位置の後の文字の point-enteredプロパティ。

これらの 2 つの値が異なる場合には、(nilでなければ) 古いポイント値と新しいポイント値という 2 つの引数とともにそれらそれぞれ呼び出される。

同じ比較は古い位置と新しい位置の前の文字にたいしても行われる。この結果として 2 つの point-left関数 (同じ関数かもしれない)、および/または 2 つの point-entered関数 (同じ関数かもしれない) が実行される可能性がある。ある場合においては、まずすべての point-left関数が呼び出されて、その後にすべての point-entered関数が呼び出される。

さまざまなバッファー位置にたいして、そこにポイントを移動することなく文字を調べるために char-afterを使用することができる。実際のポイント値変更だけがこれらのフック関数を呼び出す。

変数 inhibit-point-motion-hooksはデフォルトでは point-leftと point-enteredのフック実行を抑制する。[Inhibit point motion hooks], page 826 を参照のこと。

これらのプロパティは時代遅れであり、かわりに cursor-sensor-functionsを使用してほしい。

`cursor-sensor-functions`

このスペシャルプロパティはカーソル移動に反応する関数リストを記録する。このリスト内の各関数は影響を受けるウィンドウ、既知のカーソルの以前の位置、このプロパティをもつテキストにカーソルが入ったか離れたかに依存するシンボル enteredか leftという 3 つの受け取って再表示の直前に呼び出される。関数はマイナーモード cursor-sensor-modeがオンのときのみ呼び出される。

変数 cursor-sensor-inhibitが非 nilなら cursor-sensor-functionsプロパティは無視される。

`composition`

このテキストプロパティは文字シーケンスをコンポーネントから構成される単一グリフ (single glyph) として表示するために使用される。しかしこのプロパティの値自身は完全に Emacs の内部的なものであり、たとえば put-text-propertyなどで直接操作しないこと。

`minibuffer-message`
: このテキストプロパティは、アクティブミニバッファーの一時的なメッセージを表示する場所を指定する。具体的にはこのプロパティをもつミニバッファーテキストの最初の文字が、その前に表示する一時的なメッセージを所有する。デフォルトではミニバッファーテキスト終端に一時的なメッセージが表示される。このテキストプロパティは`set-message-function`のデフォルト値の関数が使用する (Section 39.4.1 [Displaying Messages], page 983 を参照)。

`inhibit-point-motion-hooks` [Variable]
: この時代遅れの変数が非`nil`のときは、`point-left`と`point-entered`のフックは実行されず`intangible`プロパティは効果をもたない。この変数はグローバルにセットせず`let`でバインドすること。この変数の影響を受けるプロパティは時代遅れなので、それらを効果的に無効にするためにデフォルト値は`t`。

`show-help-function` [Variable]
: この変数が非`nil`なら、それはヘルプ文字列を表示するために呼び出される関数を指定する。これらは`help-echo`プロパティ、メニューヘルプ文字列 (Section 22.17.1.1 [Simple Menu Items], page 448 と Section 22.17.1.2 [Extended Menu Items], page 448 を参照)、ツールバーヘルプ文字列 (Section 22.17.6 [Tool Bar], page 455 を参照) かもしれない。指定された関数は、ヘルプ文字列の最初の文字が非`nil`の`help-echo-inhibit-substitution`をもっていなければ、表示するためのヘルプ文字列 (関数に与えられる前に`substitute-command-keys`を通じて渡される) を単一の引数として呼び出される。Section 24.3 [Keys in Documentation], page 522 を参照のこと。`show-help-function`を使用するモードの例は、Tooltip モード (Section “Tooltips” in *The GNU Emacs Manual* を参照) のコード例を参照のこと。

`face-filters-always-match` [Variable]
: この変数が非`nil`なら、特定の条件が合致された際のみ適用される属性を指定するフェイスフィルターは常にマッチするとみなされる。

32.19.5 フォーマットされたテキストのプロパティ

以下のテキストプロパティはフィルコマンドの挙動に影響を与えます。これらはフォーマットされたテキストを表すために使用されます。Section 32.11 [Filling], page 797 と Section 32.12 [Margins], page 800 を参照してください。

`hard`
: 改行文字がこのプロパティをもつならそれは “hard” 改行。フィルコマンドは hard 改行を変更せずそれらを横断して単語を移動しない。しかしこのプロパティはマイナーモード`use-hard-newlines`が有効な場合のみ影響を与える。Section “Hard and Soft Newlines” in *The GNU Emacs Manual* を参照のこと。

`right-margin`
: このプロパティはその部分のテキストのフィルにたいして余分な右マージンを指定する。

`left-margin`
: このプロパティはその部分のテキストのフィルにたいして余分な左マージンを指定する。

`justification`
: このプロパティはその部分のテキストのフィルにたいして位置揃え (justification) のスタイルを指定する。

32.19.6 テキストプロパティの粘着性

ユーザーがそれらをタイプした際にはバッファーに挿入される自己挿入文字 (Section 32.5 [Commands for Insertion], page 783 を参照) は、通常は先行する文字と同じプロパティをもちます。これはプロパティの継承 *(inheritance)* と呼ばれます。

対照的に Lisp プログラムは継承の有無に関わらず挿入を行うことができ、それは挿入プリミティブの選択に依存します。`insert`のような通常のテキスト挿入関数は何もプロパティを継承しません。これらは挿入される文字列と正確に同じプロパティをもち、それ以外のプロパティはもちません。これはたとえば kill リング外部にたいしてのように、あるコンテキストから他のコンテキストにテキストをコピーするプログラムにたいして適正です。継承つきで挿入を行うためには、このセクションで説明するスペシャルプリミティブを使用します。自己挿入文字はこれらのプリミティブを使用するのでプロパティを継承します。

継承つきで挿入を行う際に、何のプロパティがどこから継承されるかは *sticky(*スティッキー、粘着する*)* に依存します。ある文字の後への挿入における、それらの文字のプロパティ継承は *rear-sticky(*後方スティッキー*)* です。ある文字の前への挿入における、それらの文字ノプロパティ継承は *front-sticky(*前方スティッキー*)* です。これら両側の sticky が同じプロパティにたいして異なる sticky 値をもつ場合には、前の文字の値が優先します。

デフォルトではテキストプロパティは front-sticky ではなく rear-sticky です。したがってデフォルトでは、すべてのプロパティは前の文字から継承して、後の文字からは何も継承しません。

さまざまなテキストプロパティの stickiness(スティッキネス、スティッキー性、粘着性、粘着度)は、2 つのテキストプロパティ`front-sticky`と `rear-nonsticky`、および変数 `text-property-default-nonsticky`で制御できます。与えられたプロパティにたいして異なるデフォルトを指定するためにこの変数を使用できます。テキストの任意の特定部分に特定の sticky や非 sticky プロパティを指定するために、これら 2 つのテキストプロパティを使用できます。

ある文字の `front-sticky`プロパティが `t`なら、その文字のすべてのプロパティは front-sticky です。`front-sticky`プロパティがリストなら、その文字の sticky なプロパティは名前がそのリスト内にあるプロパティです。たとえばある文字が値が `(face read-only)`であるような `front-sticky` プロパティをもつなら、その文字の前への挿入ではその文字の `face`プロパティと `read-only`プロパティは継承できますが他のプロパティは継承できません。

`rear-nonsticky`は逆の方法で機能します。ほとんどのプロパティはデフォルトで rear-sticky であり、`rear-nonsticky`プロパティはどのプロパティが *rear-sticky* ではないかを告げます。ある文字の `rear-nosticky`プロパティが `t`なら、その文字のすべてのプロパティは rear-sticky ではありません。`rear-nosticky`プロパティがリストなら、その文字の sticky なプロパティは名前がそのリスト内にないプロパティです。

`text-property-default-nonsticky` [Variable]

この変数はさまざまなテキストプロパティのデフォルトの rear-stickiness を定義する alist。各要素は `(`*`property . nonstickiness`*`)`という形式をもち、これは特定のテキストプロパティ *property*の stickiness を定義する。

*nonstickiness*が非 `nil`なら、それはプロパティ *property*がデフォルトで rear-nonsticky であることを意味する。すべてのプロパティはデフォルトでは front-nonsticky なので、これにより *property*は両方向にたいしてデフォルトで nonsticky になる。

テキストプロパティ`front-sticky`と `rear-nonsticky`が使用された際には、`text-property-default-nonsticky`内で指定されたデフォルトの *nonstickiness*より優先される。

以下はプロパティ継承つきでテキストを挿入する関数です:

`insert-and-inherit` **&rest** *strings* [Function]
関数 `insert`と同じように文字列 *strings*を挿入するが、隣接するテキストからすべての sticky なプロパティを継承する。

`insert-before-markers-and-inherit` **&rest** *strings* [Function]
関数 `insert-before-markers`と同じように文字列 *strings*を挿入するが、隣接するテキストからすべての sticky なプロパティを継承する。

継承を行わない通常の挿入関数については Section 32.4 [Insertion], page 781 を参照してください。

32.19.7 テキストプロパティの lazy な計算

バッファー内のすべてのテキストにたいしてテキストプロパティを計算するかわりに、何かがテキスト範囲に依存している場合にはテキストプロパティを計算するようにアレンジできます。

プロパティとともにバッファーからテキストを抽出するプリミティブは `buffer-substring`です。この関数はプロパティを調べる前にアブノーマルフック `buffer-access-fontify-functions`を実行します。

`buffer-access-fontify-functions` [Variable]
この変数はテキストプロパティ計算用の関数のリストを保持する。`buffer-substring`がバッファーの一部のテキストとテキストプロパティをコピーする前にこのリスト内の関数すべてを呼び出す。各関数はアクセスされるバッファー範囲を指定する 2 つの引数を受け取る (バッファーは常にカレントバッファー)。

関数 `buffer-substring-no-properties`はいずれにせよテキストプロパティを無視するので、これらの関数を呼び出さない。

同じバッファー部分にたいして複数回フック関数が呼び出されるのを防ぐために変数 `buffer-access-fontified-property`を使用できる。

`buffer-access-fontified-property` [Variable]
この変数の値が非 `nil`なら、それはテキストプロパティ名として使用されるシンボル。そのテキストプロパティにたいする非 `nil`値は、その文字にたいする他のテキストプロパティはすでに計算済みであることを意味する。

`buffer-substring`にたいして指定された範囲内のすべての文字、このプロパティにたいする値として非 `nil`をもつなら、`buffer-substring`は `buffer-access-fontify-functions`の関数を呼び出さない。それらの文字がすでに正しいテキストプロパティをもつとみなして、それらがすでに所有するプロパティを単にコピーする。

`buffer-access-fontify-functions`の関数にこのプロパティ、同様に他のプロパティを処理対象の文字に追加させることがこの機能の通常の用途である。この方法では同じテキストにたいして、それらの関数が何度も呼び出されるのを防ぐことができる。

32.19.8 クリック可能なテキストの定義

クリック可能テキスト *(clickable text)* とは何らかの結果を生成するためにマウスやキーボードコマンドを通じてクリックできるテキストです。多くのメジャーモードがテキスト的なハイパーリンク、略してリンク *(link)* を実装するためにクリック可能テキストを使用しています。

リンクの挿入や操作を行うもっとも簡単な方法は `button`パッケージの使用です。Section 39.19 [Buttons], page 1068 を参照してください。このセクションではテキストプロパティを使用してバッ

ファー内に手作業でクリック可能テキストをセットアップする方法を説明します。簡略にするためにクリック可能テキストをリンクと呼ぶことにします。

リンクの実装には、(1) リンク上にマウスが移動した際にクリック可能であることを示し、(2) そのリンク上の RETか *mouse-2*で何かを行うようにして、(3) そのリンクが mouse-1-click-follows-linkにしたがうよう follow-linkをセットアップする、という 3 つのステップが含まれます。

クリック可能なことを示すためには、そのリンクのテキストに mouse-faceプロパティを追加します。すると Emacs はそれ以降マウスがその上に移動した際にリンクをハイライトするでしょう。加えて help-echoテキストプロパティを使用してツールチップかエコーエリアメッセージを定義するべきです。Section 32.19.4 [Special Properties], page 820 を参照してください。たとえば以下は Dired がファイル名がクリック可能なことを示す方法です:

```
(if (dired-move-to-filename)
    (add-text-properties
      (point)
      (save-excursion
        (dired-move-to-end-of-filename)
        (point))
      '(mouse-face highlight
        help-echo "mouse-2: visit this file in other window")))
```

リンクをクリック可能にするためには、RETと *mouse-2*を望むアクションを行うコマンドにバインドします。各コマンドはリンク上から呼び出されたかチェックして、それに応じて動作するべきです。たとえば Dired メジャーモードのキーマップは *mouse-2*を以下のコマンドにバインドします:

```
(defun dired-mouse-find-file-other-window (event)
  "In Dired, visit the file or directory name you click on."
  (interactive "e")
  (let ((window (posn-window (event-end event)))
        (pos (posn-point (event-end event)))
        file)
    (if (not (windowp window))
        (error "No file chosen"))
    (with-current-buffer (window-buffer window)
      (goto-char pos)
      (setq file (dired-get-file-for-visit)))
    (if (file-directory-p file)
        (or (and (cdr dired-subdir-alist)
                 (dired-goto-subdir file))
            (progn
              (select-window window)
              (dired-other-window file)))
      (select-window window)
      (find-file-other-window (file-name-sans-versions file t)))))
```

このコマンドはクリックがどこで発生したかを判断するために関数 posn-windowと posn-point、visit するファイルの判断に関数 dired-get-file-for-visitを使用します。

マウスコマンドをメジャーモードキーマップ内でバインドするかわりに、keymapプロパティ(Section 32.19.4 [Special Properties], page 820 を参照) を使用してリンクテキスト内でバインドできます。たとえば:

```
(let ((map (make-sparse-keymap)))
  (define-key map [mouse-2] 'operate-this-button)
  (put-text-property link-start link-end 'keymap map))
```

この手法により異なるリンクに異なるコマンドを簡単に定義できます。さらにそのバッファー内の残りのテキストにたいしては、RETと *mouse-2*のグローバル定義を利用可能なまま残すことができます。

リンク上でのクリックにたいするEmacsの基本コマンドは*mouse-2*です。しかし他のグラフィカルなアプリケーションとの互換性のために、ユーザーがマウスを動かさずに素早くリンクをクリックするという条件の下で、Emacsはリンク上での*mouse-1*クリックも認識します。この振る舞いはユーザーオプション`mouse-1-click-follows-link`により制御されます。Section “Mouse References” in *The GNU Emacs Manual* を参照してください。

`mouse-1-click-follows-link`にしたがうようにリンクをセットアップするには、(1) そのテキストに`follow-link`テキストプロパティまたはオーバーレイプロパティを適用する、または (2)`follow-link`イベントをキーマップ (`keymap`テキストプロパティを通じたメジャーモードキーマップまたはローカルキーマップ) にバインドするかの、いずれかを行わなければなりません。`follow-link`プロパティの値、または`follow-link`イベントにたいするバインディングはリンクアクションにたいするコンディション (condition) として機能します。この条件はEmacsにたいして2つのことを告げます。それは*mouse-1*のクリックがそのリンクの内側で発生したとみなすべき状況、そして*mouse-1*のクリックを何に変換するかを告げるアクションコード (action code) を計算する方法です。そのリンクのアクション条件は以下のうちの1つです:

`mouse-face`
: コンディションがシンボル`mouse-face`の場合には、その位置に非`nil`の`mouse-face`プロパティがあればそれはリンク内側の位置。アクションコードは常に`t`。

  以下はInfoモードが`mouse-1`を処理する例:

```
(define-key Info-mode-map [follow-link] 'mouse-face)
```

関数
: コンディションが関数*func*の場合には、(*func pos*)が非`nil`に評価されれば、位置*pos*はリンクの内側。*func*がリターンする値はアクションコードとして機能する。

  以下はpcvsがファイル名の上でのみ*mouse-1*によるリンクのフォローを有効にする方法の例:

```
(define-key map [follow-link]
  (lambda (pos)
    (eq (get-char-property pos 'face) 'cvs-filename-face)))
```

その他
: コンディション値がそれ以外の場合には、その位置はリンク内側であり、そのコンディション自体がアクションコード。(バッファー全体に適用されないように) リンクテキストのテキストプロパティかオーバーレイプロパティを通じてコンディションを適用するときのみ、この類のコンディションを指定すべきなのは明確である。

アクションコードは*mouse-1*がリンクをフォローする方法を告げます:

文字列かベクター
: アクションコードが文字列かベクターなら、*mouse-1*イベントは文字列かベクターの最初の要素に変換される。つまり*mouse-1*クリックのアクションはその文字、またはシンボルのローカルかグローバルのバインディング。したがってアクションコードが`"foo"`なら、*mouse-1*は*f*、`[foo]`なら*mouse-1*は`foo`に変換される。

その他
: その他の非`nil`のアクションコードでは、*mouse-1*イベントは同じ位置の*mouse-2*イベントに変換される。

`define-button-type`で定義されるボタンをアクティブにするように*mouse-1*を定義するには、そのボタンに`follow-link`プロパティを与えます。このプロパティの値は上述したリンクのアクションコンディションであるべきです。Section 39.19 [Buttons], page 1068 を参照のこと。たとえば以下はHelpモードが*Mouse-1*を処理する例です。

```
(define-button-type 'help-xref
  'follow-link t
```

```
  'action #'help-button-action)
```

`define-widget`で定義されたウィジェットに *mouse-1*を定義するには、そのウィジェットに`:follow-link`プロパティを与えます。このプロパティの値は、上述したようなリンクのアクションコンディションであるべきです。たとえば以下は`mouse-1`クリックが`RET`に変換されるように`link`ウィジェットを指定する方法の例です:

```
(define-widget 'link 'item
  "An embedded link."
  :button-prefix 'widget-link-prefix
  :button-suffix 'widget-link-suffix
  :follow-link "\C-m"
  :help-echo "Follow the link."
  :format "%[%t%]")
```

`mouse-on-link-p` *pos* [Function]

この関数はカレントバッファー内の位置 *pos*がリンク上なら非`nil`をリターンする。*pos*は`event-start`がリターンするようなマウスイベント位置でもよい (Section 21.7.13 [Accessing Mouse], page 400 を参照)。

32.19.9 フィールドの定義と使用

フィールドとはバッファー内にある連続する文字範囲であり、`field`プロパティ(テキストプロパティかオーバーレイプロパティ) に同じ値 (`eq`で比較) をもつことにより識別されます。このセクションではフィールドの操作に利用できるスペシャル関数を説明します。

フィールドはバッファー位置 *pos*で指定します。各フィールドはバッファー位置の範囲を含むと考えて、指定した位置はその位置を含むフィールドを表します。

*pos*の前後の文字は同じフィールドに属し、どのフィールドが *pos*を含むかという疑問はありません。それらの文字が属するフィールドがそのフィールドです。*pos*がフィールド境界のときは、それがどのフィールドに属すかは、取り囲む 2 つの文字の`field`プロパティの stickiness に依存します (Section 32.19.6 [Sticky Properties], page 827 を参照)。*pos*に挿入されたテキストからプロパティが継承されたフィールドが *pos*を含むフィールドです。

*pos*に新たに挿入されたテキストが、いずれの側からも`field`プロパティを継承しない異常なケースがあります。これは前の文字の`field`プロパティが rear-sticky でなく、後の文字の`field`プロパティが front-sticky でもない場合に発生します。このケースでは *pos*は前後のフィールドいずれにも属しません。フィールド関数はそれを、開始と終了が *pos*であるような空フィールドに属するものとして扱います。

以下のすべての関数では、*pos*が省略か`nil`ならポイントの値がデフォルトとして使用されます。ナローイング (narrowing) が効力をもつ場合には、*pos*はアクセス可能部分にあるはずです。Section 30.4 [Narrowing], page 766 を参照してください。

`field-beginning` **&optional** *pos escape-from-edge limit* [Function]

この関数は *pos*で指定されたフィールドの先頭をリターンする。

*pos*が自身のフィールド先頭にあり、かつ *escape-from-edge*が非`nil`なら、*pos*周辺の`field`プロパティの stickiness に関わらず、リターン値は常に *pos*が終端であるような、前にあるフィールドの先頭になる。

*limit*が非`nil`なら、それはバッファーの位置。そのフィールドの先頭が *limit*より前なら、かわりに *limit*がリターンされるだろう。

`field-end` **&optional** *pos escape-from-edge limit* [Function]

この関数は *pos*で指定されるフィールドの終端をリターンする。

*pos*が自身のフィールド終端にあり、かつ *escape-from-edge*が非 `nil`なら、*pos*周辺の `field` プロパティの stickiness に関わらず、リターン値は常に *pos*が先頭であるような後のフィールドの終端になる。

*limit*が非 `nil`なら、それはバッファーの位置である。そのフィールドの終端が *limit*より後なら、かわりに *limit*がリターンされるだろう。

`field-string` **&optional** *pos* [Function]
この関数は *pos*で指定されるフィールドのコンテンツを文字列としてリターンする。

`field-string-no-properties` **&optional** *pos* [Function]
この関数は *pos*で指定されるフィールドのコンテンツを、テキストプロパティを無視して文字列としてリターンする。

`delete-field` **&optional** *pos* [Function]
この関数は *pos*で指定されるフィールドのテキストを削除する。

`constrain-to-field` *new-pos old-pos* **&optional** *escape-from-edge only-in-line inhibit-capture-property* [Function]
この関数は *new-pos*を *old-pos*が属するフィールドに “拘束 (constrain)” する。言い換えると、これは *old-pos*と同じフィールド内で *new-pos*にもっとも近い位置をリターンする。

*new-pos*が `nil`なら、`constrain-to-field`はかわりにポイントの値を使用してポイントをリターンすることに加えて、その位置にポイントを移動する。

*old-pos*が 2 つのフィールドの境界なら、許容できる最後の位置は引数 *escape-from-edge*に依存する。*escape-from-edge*が `nil`なら、*new-pos*は新たに文字が *old-pos*が挿入されたときに継承するであろう値と、`field`プロパティが等しいフィールドでなければならない。*escape-from-edge*が非 `nil`なら *new-pos*は隣接する 2 つのフィールド内のどこでも構わない。さらに 2 つのフィールドが特別な値 `boundary`により他のフィールドで分割されている場合には、このスペシャルフィールド内のすべてのポイントも境界上とみなされる。

引数なしの `C-a`コマンドのように、特別な種類の位置に後方へ移動して一度そこに留まるには、おそらく *escape-from-edge*にたいして `nil`を指定するべきであろう。フィールドをチェックする他の移動コマンドにたいしては、おそらく `t`を渡すべきである。

オプション引数 *only-in-line*が非 `nil`、かつ *new-pos*を通常の方法により拘束することにより異なる行へ移動するような場合には、*new-pos*は非拘束でリターンされる。これは `next-line`や `beginning-of-line`のような行単位の移動コマンドで、それらのコマンドが正しい行へ移動できる場合のみフィールド境界を尊重するようにするために用いられる。

オプション引数 *inhibit-capture-property*が非 `nil`、かつ *old-pos*がその名前の非 `nil`のプロパティをもつ場合には、すべてのフィールド境界は無視される。

変数 `inhibit-field-text-motion`を非`nil`値にバインドすることにより、`constrain-to-field`にすべてのフィールド境界を無視 (何者にも拘束されることがない) させることができる。

32.19.10 なぜテキストプロパティはインターバルではないのか

ユーザーにテキスト内のインターバル (訳注: 原文のインターバルは IT 用語としては時間や距離などの間隔を示す用語として用いれることが多いと思いますが、ここでは『範囲』を示す言葉として用いられているようです。他の箇所で『範囲』と訳した range 等と異なる機能なので、ここではそのまま『インターバル』としました) を指定させて、そのインターバルにプロパティを追加するために、バッファー内のテキストへの属性の追加をサポートするエディターがいくつかあります。それらのエディ

ターはユーザーやプログラマーが個別にインターバルの開始と終了を決定することを許可します。わたしたちはテキスト変更に関連する特定の逆説的振る舞いを避けるために、故意に異なる種類のインターフェイスを Emacs Lisp 内に提供しました。

複数のインターバルに細分化することが実際に意味をもつなら、それは特定のプロパティをもつ単一のインターバルのバッファーと、同じテキストをもち両方が同じプロパティをもつ 2 つのインターバルに分割されたバッファーを区別できることを意味します。

インターバルを 1 つだけもつバッファーがあり、その一部を kill することを考えてみてください。そのそのバッファーに残されるのは 1 つのインターバルであり、kill リング (と undo リスト) 内のコピーは別個のインターバルになります。その kill されたテキストを yank で戻すと、同じプロパティをもつ 2 つのインターバルを得ることになります。したがって編集では 1 つのインターバルと 2 つのインターバルの違いは保たれません。

テキスト挿入時に 2 つのインターバルを結合することにより、この問題に “対応” したとします。これはそのバッファーが元々単一のインターバルだったなら上手く機能します。しかしかわりに同じプロパティをもつ隣接する 2 つのインターバルがあり、そのうちの 1 つのインターバルからテキストを kill して yank で戻すことを考えてみてください。あるケースを解決する同じインターバル結合機能が、他のケースにおいては問題を引き起こすのです。この yank 後にインターバルはただ 1 つとなります。繰り返します、編集では 1 つのインターバルと 2 つのインターバルの違いは保たれないのです。

インターバルの間の境界上へのテキスト挿入においても満足できる回答が存在しないような問題が発生します。

しかし “バッファーにあるテキスト位置または文字列位置のプロパティは何か?” という形式の問にたいして、編集が一貫した振る舞いをするようアレンジするのは簡単です。そこでわたしたちはこれらが合理的な唯一の問いであると判断したのです。わたしたちはインターバルの開始と終了の場所を問うような実装をしませんでした。

実際には明白にインターバル境界であるような箇所では、通常はテキストプロパティ検索関数を使用できます。可能であるならインターバルは常に結合されるとみなすことにより、それらがインターバル境界を探すと考えることができます。Section 32.19.3 [Property Search], page 817 を参照してください。

Emacs はプレゼンテーション機能として明示的なインターバルも提供します。Section 39.9 [Overlays], page 999 を参照してください。

32.20 文字コードの置き換え

以下の関数は文字コードにもとづいて指定されたリージョン内の文字を置き換えます。

`subst-char-in-region` *start end old-char new-char* **&optional** *noundo* [Function]

この関数は *start* と *end* で定義されるカレントバッファーのリージョン内に出現する文字 *old-char* を *new-char* に置き換える。

noundo が非 `nil` なら `subst-char-in-region` は undo 用に変更を記録せず、バッファーを変更済みとマークしない。これは古い機能である選択的ディスプレイ (Section 39.7 [Selective Display], page 994 を参照) にとって有用だった。

`subst-char-in-region` はポイントを移動せず `nil` をリターンする。

```
---------- Buffer: foo ----------
This is the contents of the buffer before.
---------- Buffer: foo ----------
```

```
(subst-char-in-region 1 20 ?i ?X)
     ⇒ nil

---------- Buffer: foo ----------
ThXs Xs the contents of the buffer before.
---------- Buffer: foo ----------
```

`translate-region` *start end table* [Command]

この関数はバッファー内の位置 *start*と *end*の間の文字にたいして、変換テーブル (translation table) を適用する。

変換テーブル *table*は文字列か文字テーブル。`(aref table ochar)`は *ochar*に対応した変換後の文字を与える。*table*が文字列なら、*table*の長さより大きいコードの文字はこの変更により変更されない。

`translate-region`のリターン値は、その変換により実際に変更された文字数。変換テーブル内でその文字自身にマップされる文字は勘定に入らない。

32.21 レジスター

レジスター (register) とは、Emacs 内の編集においてさまざまな異なる種類の値を保持できる一種の変数です。レジスターはそれぞれ 1 文字で命名されます。すべての ASCII文字、およびそれらのメタ修飾された変種 (ただし *C-g*は例外) をレジスターの命名に使用できます。したがって利用可能なレジスター数は 255 になります。Emacs Lisp ではレジスターは自身の名前となるその文字により指定されます。

`register-alist` [Variable]

この変数は要素が `(name . contents)`という形式の alist。使用中の Emacs レジスターごとに通常は 1 つの要素が存在する。

オブジェクト *name*はレジスターを識別する文字 (整数)。

レジスターの *contents*には、いくつかのタイプがある:

数字　数字はそれ自身を意味する。`insert-register`はレジスター内の数字を探して 10 進数に変換する。

マーカー　マーカーはジャンプ先のバッファー位置を表す。

文字列　文字列の場合はレジスター内に保存されたテキスト。

矩形 (rectangle)
: 矩形は文字列のリストを表す。

`(window-configuration position)`
: これは 1 つのフレームにリストアされるウィンドウ構成、およびカレントバッファー内のジャンプ先の位置を表す。

`(frame-configuration position)`
: これはリストア用のフレーム構成とカレントバッファー内のジャンプ先の位置。

(file *filename*)
: これは visit するファイルを表し、この値にジャンプすることによりファイル *filename* を visit する。

(file-query *filename position*)
これは visit するファイルとファイル内の位置を表す。この値にジャンプすることによりファイル *filename*を visit してバッファー位置 *position*に移動する。このタイプの位置をリストアすると、まずユーザーにたいして確認を求める。

このセクションの関数は特に明記しない限り予期せぬ値をリターンします。

`get-register` *reg* [Function]
この関数はレジスター *reg*のコンテンツ、コンテンツがなければ `nil`をリターンする。

`set-register` *reg value* [Function]
この関数はレジスター *reg*のコンテンツに *value*をセットする。レジスターには任意の値をセットできるが、その他のレジスター関数は特定のデータ型を期待する。リターン値は *value*。

`view-register` *reg* [Command]
このコマンドはレジスター *reg*に何が含まれているかを表示する。

`insert-register` *reg* **&optional** *beforep* [Command]
このコマンドはカレントバッファーにレジスター *reg*のコンテンツを挿入する。

このコマンドは通常は挿入したテキストの前にポイント、後にマークを配置する。しかしオプションの第 2 引数 *beforep*が非 `nil`ならマークを前、ポイントを後に配置する。インタラクティブな呼び出しでは、プレフィクス引数を与えることにより 2 つ目の引数 *beforep*に `nil`を渡すことができる。

このコマンドはインタラクティブに呼び出された際には、デフォルトではテキストの後にポイントを配置して、プレフィクス引数を与えるとこの反対の振る舞いを行う。

レジスターに矩形が含まれる場合には、その矩形はポイントの左上隅に挿入される。これはそのテキストがカレント行と、その下に続く行に挿入されることを意味する。

レジスターが保存されたテキスト (文字列) または矩形 (リスク) 以外の何かを含む場合には、現在のところは役に立つようなことは起きない。これは将来変更されるかもしれない。

`register-read-with-preview` *prompt* [Function]
この関数は *prompt*、およびもしかしたら既存レジスターとそのコンテンツをプレビューしてレジスターの名前を読み取ってレジスター名をリターンする。このプレビューはユーザーオプション `register-preview-delay`と `register-alist`がいずれも非 `nil`なら、`register-preview-delay`で指定された遅延の後に一時ウィンドウ内に表示される。このプレビューはユーザーが (たとえばヘルプ文字のタイプにより) ヘルプを要求した場合にも表示される。レジスター名を読み取るンタラクティブな関数には、この関数の使用を推奨する。

32.22 テキストの交換

以下の関数はテキストの一部を置き換えるために使用できます:

`transpose-regions` *start1 end1 start2 end2* **&optional** *leave-markers* [Function]
この関数はバッファーの重複しない 2 つの部分を交換する (重複する場合にはエラーをシグナルする)。引数 *start1*と *end1*は一方の部分の両端、引数 *start2*と *end2*はもう一方の部分の両端を指定する。

`transpose-regions`は通常は置き換えたテキストにともないマーカーを再配置する。以前は 2 つの置き換えたテキストのうちの一方の部分に位置していたマーカーは、その部分とともに

移動されるので、それを挟む 2 つの文字の新たな位置の間に留まることになる。しかし *leave-markers*が非 `nil`なら、`transpose-regions`はこれを行わず、すべてのマーカーを再配置せずに残す。

32.23 バッファーテキストの置換

以下の関数を使用して、あるバッファーのテキストを他のバッファーのテキストで置き換えることができます:

`replace-buffer-contents` *source* **&optional** *max-secs max-costs* [Command]

この関数はバファー *source*のアクセス可能範囲でカレントバッファーのアクセス可能範囲を置き換える。*source*はバッファーオブジェクトかバッファー名のいずれか。`replace-buffer-contents`が成功するとカレントバッファーのアクセス可能範囲のテキストは、バッファー *source* のアクセス可能範囲のテキストと等しくなる。

この関数はバファー *source*のアクセス可能範囲でカこの関数はカレントバッファーのポイント、マーカー、テキストプロパティ、オーバーレイをそのまま維持しようと試みる。この挙動が好都合であるような潜在的なケースは外部コードをフォーマットするプログラムだろう。これらは通常は再フォーマットしたテキストを一時的なバッファーかファイルに書き込んで、`delete-region`や `insert-buffer-substring`を使用することによりそれらのプロパティを削除する。しかし後者の組み合わせのほうが通常は高速である (Section 32.6 [Deletion], page 784 と Section 32.4 [Insertion], page 781 を参照)。

これが機能するためには `replace-buffer-contents`が元バッファーと *source*のコンテンツを比較する必要があり、これはバッファーが巨大で多数の差異が存在する場合にはコストがかかる処理となる。`replace-buffer-contents`の実行時間を制限するために、2 つのオプション引数がある。

*max-secs*は秒単位のハードリミットを定義する。これが与えられて超過した場合には、`delete-region`および `insert-buffer-substring`にフォールバックする。

*max-costs*は差分計算の品質を定義する。このリミットを実際のコストが超過したら、次善だがより高速な発見的手法を使用する。デフォルト値は 1000000。

`replace-buffer-contents`は非破壊的な置換ができれば t をリターンする。それ以外の場合には *max-secs*を超過したら nil をリターンする。

`replace-region-contents` *beg end replace-fn* **&optional** *max-secs max-costs* [Function]

この関数は与えられた *replace-fn*を使用して、*beg*と *end*の間のリージョンを置換する。関数 *replace-fn*はカレントバッファーを指定されたリージョンにナローして実行される。リージョンを置換する文字列またはバッファーのいずれかをリターンすること。

置換は上述の `replace-buffer-contents` (引数 *max-secs*、*max-costs*、リターン値についても記述あり) を使用して行われる。

注意: 置換物が文字列なら一時バッファーに配置されるので `replace-buffer-contents`が処理することができる。したがってすでにバッファーに置換物がある場合には、`buffer-substring`の類を使用して文字列に変換することは無意味である。

32.24 圧縮されたデータの処理

`auto-compression-mode`が有効なときは、Emacs は圧縮されたファイルを visit する際に自動的に解凍して、それを変更して保存する際は自動的に再圧縮します。Section “Compressed Files” in *The GNU Emacs Manual* を参照してください。

上記の機能は外部の実行可能ファイル (例: `gzip`) を呼び出すことにより機能します。zlib ライブラリーを使用したビルトインの解凍サポートつきで Emacs をコンパイルすることもでき、これは外部プログラムの実行に比べて高速です。

`zlib-available-p` [Function]
この関数はビルトイン zlib 解凍が利用可能なら非 `nil`をリターンする。

`zlib-decompress-region` *start end* **&optional** *allow-partial* [Function]
この関数はビルトインの zlib 解凍を使用して *start*と *end*の間のリージョンを解凍する。このリージョンには gzip か zlib で圧縮されたデータが含まれていなければならない。この関数は成功したらリージョンのコンテンツを解凍されたデータに置き換える。*allow-partial*が `nil`か省略の場合に失敗すると、この関数はリージョンを変更せずに `nil`をリターンする。それ以外の場合には解凍されなかったバイト数をリターンして、正常に解凍されたデータが何であれ、それによりリージョンのテキストを置き換える。この関数はユニバイトバッファーでのみ呼び出すことができる。

32.25 Base 64 エンコーディング

Base64 コードは 8 ビットシーケンスをより長い ASCIIグラフィック文字シーケンスにエンコードするために email 内で使用されます。これはインターネット RFC2045 および RFC4648 でも定義されます [1]。このセクションでは、このコードへの変換および逆変換を行う関数について説明します。

`base64-encode-region` *beg end* **&optional** *no-line-break* [Command]
この関数は *beg*から *end*のリージョンを Base64 コードに変換する。これはエンコードされたテキストの長さをリターンする。リージョン内の文字がマルチバイトならエラーをシグナルする (マルチバイトバッファーではリージョンには `ascii`、`eight-bit-control`、`eight-bit-graphic`の文字以外は含まれてはならない)。

この関数は通常は行が長くなりすぎるのを防ぐために、エンコードされたテキストに改行を挿入する。しかしオプション引数 *no-line-break*が非 `nil`なら、これらの改行は追加されず出力は長い単一の行となる。

`base64url-encode-region` *beg end* **&optional** *no-pad* [Command]
この関数は `base64-encode-region`と同様だが RFC4648 にしたがい Base64 エンコーディングの URL バリアントを実装する。エンコードされたテキストに改行を挿入しないので、出力は 1 行だけの長い行となる。

オプション引数 *no-pad*は非 `nil`なら、この関数はパディング (`=`) を生成しない。

`base64-encode-string` *string* **&optional** *no-line-break* [Function]
この関数は文字列 *string*を Base64 コードに変換する。これはエンコードされたテキストを含む文字列をリターンする。`base64-encode-region`と同じように文字列内の文字がマルチバイトならエラーをシグナルする。

この関数は通常は行が長くなりすぎるのを防ぐためにエンコードされたテキストに改行を挿入する。しかしオプション引数 *no-line-break*が非 `nil`なら、これらの改行は追加されず結果となる文字列は長い単一の行となる。

[1] RFC(*Request for Comments* の略) とは標準を記述するナンバーが付与されたインターネット情報提供ドキュメントです。RFC は通常は自身が先駆的に活動する技術エキスパートによって記述され、伝統として現実的で経験主導で記述されます。

`base64url-encode-string` *string* **&optional** *no-pad* [Function]
`base64-encode-string`と同様だが Base64 の URL バリアントを生成する。エンコードされたテキストに改行を挿入しないので、結果は 1 行だけの長い行となる。

オプション引数 *no-pad*は非`nil`なら、この関数はパディングを生成しない。

`base64-decode-region` *beg end* **&optional** *base64url* [Command]
この関数は *beg*から *end*のリージョンの Base64 コードを対応するデコードされたテキストに変換する。これはデコードされたテキストの長さをリターンする。

デコード関数はエンコード済みテキスト内の改行文字を無視する。

オプション引数 *base64url*が非`nil`ならパディングはオプションであり、Base64 エンコーディングの URL バリアントが使用される。

`base64-decode-string` *string* **&optional** *base64url* [Function]
この関数は文字列 *string*を、Base64 コードから対応するデコード済みテキストに変換する。これはデコード済みテキストを含むユニバイトをリターンする。

デコード関数はエンコード済みテキスト内の改行文字を無視する。

オプション引数 *base64url*が非`nil`ならパディングはオプションであり、Base64 エンコーディングの URL バリアントが使用される。

32.26 チェックサムとハッシュ

Emacs には暗号化ハッシュ*(cryptographic hashes)* 計算用のビルトインのサポートがあります。暗号化ハッシュ、またはチェックサム *(checksum)* とはデータ断片にたいするデジタルな指紋 (fingerprint) であり、そのデータが変更されていないかチェックするために使用できます。

Emacs は MD5、SHA-1、SHA-2、SHA-224、SHA-256、SHA-384、SHA-512 のような一般的な暗号化ハッシュアルゴリズムをサポートします。これらのアルゴリズムのうち MD5 はもっとも古く、ネットワーク越しに転送されたメッセージの整合性をチェックするために一般的にはメッセージダイジェスト *(message digests)* 内で使用されています。MD5 と SHA-1 は '衝突耐性 (collision resistant) をもたない (同じ MD5 または SHA-1 のハッシュをもつ異なるデータ片を故意にデザインすることが可能) ので、セキュリティに関連することに使用するべきではありません。セキュリティーに関するアプリケーションでは SHA-2 (`sha256`や `sha512`) のような他のハッシュタイプを使用するべきです。

`secure-hash-algorithms` [Function]
この関数は `secure-hash`が使用可能なアルゴリズムを表すシンボルのリストをリターンする。

`secure-hash` *algorithm object* **&optional** *start end binary* [Function]
この関数は *object*にたいするハッシュをリターンする。引数 *algorithm*はどのハッシュを計算するかを示すシンボルで `md5`、`sha1`、`sha224`、`sha256`、`sha384`、`sha512`のいずれか。引数 *object*はバッファーまたは文字列であること。

オプション引数 *start*と *end*は、メッセージダイジェストを計算する *object*部分を指定する文字位置。これらが`nil`か省略なら、*object*全体にたいしてハッシュを計算する。

引数 *binary*が省略か`nil`なら、通常の Lisp 文字列としてハッシュのテキスト形式 *(text form)* をリターンする。*binary*が非`nil`なら、ユニバイト文字列に格納されたバイトシーケンスとしてハッシュのバイナリー形式 *(binary form)* をリターンする。

この関数は *object*のテキストの内部表現 (Section 33.1 [Text Representations], page 852 を参照) からハッシュを直接計算しない。かわりにコーディングシステム (Section 33.10 [Coding

Systems], page 864 を参照) を使用してテキストをエンコードして、そのエンコード済みテキストからハッシュを計算する。*object*がバッファーなら使用されているコーディングが、そのテキストをファイルに書き込むためのデフォルトとして選択される。*object*が文字列ならユーザーの好むコーディングシステムが使用される (Section “Recognize Coding” in *GNU Emacs Manual* を参照)。

`md5` *object* **&optional** *start end coding-system noerror* [Function]

この関数は MD5 ハッシュをリターンする。これはほとんどの目的において、*algorithm*引数に `md5`を指定して `secure-hash`を呼び出すのと等価であり半ば時代遅れである。引数の *object*、*start*、*end*は `secure-hash`のときと同じ意味をもつ。

*coding-system*が非 `nil`なら、それはテキストをエンコードするために使用するコーディングシステムを指定する。省略または `nil`なら、`secure-hash`と同様にデフォルトコーディングシステムが使用される。

`md5`は通常は指定や選択されたコーディングシステムを使用してテキストをエンコードできなければエラーをシグナルする。しかし *noerror*が非 `nil`なら、かわりに黙って `raw-text`コーディングシステムを使用する。

`buffer-hash` **&optional** *buffer-or-name* [Function]

*buffer-or-name*のハッシュをリターンする。`nil`の場合のデフォルトはカレントバッファー。この関数は `secure-hash`とは対照的にコーディングシステムとは無関係にバッファーの内部表現にもとづいてハッシュを計算する。したがって同一の Emacs 上で実行中の 2 つのバッファーを比較する際にのみ有用であり、異なるバージョンの Emacs 間で同じハッシュをリターンする保証はない。これは巨大なバッファーにたいして `secure-hash`より幾分効果的であり、`secure-hash`ほど多くのメモリーを割り当てないはずである。

32.27 GnuTLS 暗号化

GnuTLS とともにコンパイルされていれば、Emacs はビルトインの暗号化サポートを提供します。GnuTLS の API 用語にしたがいダイジェスト (digests)、MAC(メッセージ認証符号)、共通鍵暗号 (symmetric ciphers)、認証付き暗号 (AEAD ciphers) のツールが利用できます。

ここで使用する IV(Initialization Vector: 初期化ベクトル) のような暗号化にある程度親しんでいる必要のある用語は詳細には定義しません。GnuTLS ライブラリーの用語や構造を理解する助けとなる特定のドキュメントについては `https://www.gnutls.org/`を参照してください。

32.27.1 GnuTLS 暗号化入力のフォーマット

GnuTLS 暗号化関数への入力は Emacs Lisp のプリミティブかリストのいずれかにより、複数の方法で指定できます。

現在のところリストの形式は `md5`と `secure-hash`が動作する方法に似ています。

バッファー　入力として単にバッファーを渡すとバッファー全体が使用されることを意味する。

文字列　文字列は入力として直接使用される。(他のほとんどの Emacs Lisp 関数とは異なり) 関数の処理後に機密データ漏洩の機会を減少させるために関数が文字列を変更するかもしれない。

(*buffer-or-string start end coding-system noerror*)

これは上述のようにバッファーか文字列を指定するが、オプションで *start*と *end*で範囲を指定できる。

加えて必要ならオプションで *coding-system*を指定できる。

最後のオプションのアイテム *noerror*は指定もしくは選択されたコーディングシステムを使用してテキストをエンコードできない際の通常のエラーをオーバーライドする。*noerror*が非 `nil`なら関数は暗黙に `raw-text`コーディングシステムをかわりに使用する。

`(iv-auto` *`length`*`)`

これは GnuTLS の `GNUTLS_RND_NONCE`ジェネレーターを使用して指定された長さの IV(Initialization Vector: 初期化ベクトル) を生成する。これにより IV が予測不可能になり同一セッション内での再利用の可能性が低くなる。IV の実際の値は後述の関数がリターンする。

32.27.2 GnuTLS 暗号化関数

`gnutls-digests` [Function]

この関数は GnuTLS ダイジェストアルゴリズムの alist をリターンする。

各エントリーはアルゴリズムを表すキーとアルゴリズムの内部的な詳細を表す plist をもつ。plist は結果となるダイジェストのバイトサイズを示す`:type gnutls-digest-algorithm`、および`:digest-algorithm-length 64`のキーももつだろう。

GnuTLS MAC とダイジェストアルゴリズムの間には類似した名前が存在するが、これらは内部的には別物であり混在させるべきではない。

`gnutls-hash-digest` *digest-method input* [Function]

*digest-method*は `gnutls-digests`由来の完全な plist、単なるシンボルキー、またはシンボル名の文字列でもよい。

*input*はバッファー、文字列、もしくは他の方法 (Section 32.27.1 [Format of GnuTLS Cryptography Inputs], page 839 を参照) でも指定できる。

この関数はエラー時には `nil`、*digest-method*か *input*が無効なら Lisp エラーをシグナルする。成功時にはバイナリー文字列 (出力) と使用される IV のリストをリターンする。

`gnutls-macs` [Function]

この関数は GnuTLS MAC アルゴリズムの alist をリターンする。

各エントリーはアルゴリズムを表すキーとアルゴリズムの内部的な詳細を表す plist をもつ。この plist は結果となるハッシュ、キー、ナンスのバイトサイズを示すキー`:type gnutls-mac-algorithm`、`:mac-algorithm-length`、`:mac-algorithm-keysize`、`:mac-algorithm-noncesize`

現在のところナンスは使用されておらず、いくつかの MAC でのみサポートされる。

GnuTLS MAC とダイジェストアルゴリズムの間には類似した名前が存在するが、これらは内部的には別物であり混在させるべきではない。

`gnutls-hash-mac` *hash-method key input* [Function]

*hash-method*は `gnutls-macs`由来の完全な plist、単なるシンボルキー、またはシンボル名の文字列でもよい。

*key*はバッファー、文字列、あるいは他の方法 (Section 32.27.1 [Format of GnuTLS Cryptography Inputs], page 839 を参照) でも指定できる。*key*が文字列なら使用後に消去される。

*input*はバッファー、文字列、もしくは他の方法 (Section 32.27.1 [Format of GnuTLS Cryptography Inputs], page 839 を参照) でも指定できる。

この関数はエラー時には `nil`、*hash-method*や *key*、*input*が無効なら Lisp エラーをシグナルする。

成功時にはバイナリー文字列 (出力) と使用される IV をリターンする。

`gnutls-ciphers` [Function]

この関数は GnuTLS 暗号のリストをリターンする。

各エントリーは暗号を表すキーとそのアルゴリズムに関する内部的な詳細を表す plist をもつ。この plist は`:type gnutls-symmetric-cipher`、および AEAD 機能を示すために `nil`か `t`にセットされたキー`:cipher-aead-capable`、さらにタグ、結果データのブロックサイズ、キー、IV のバイトサイズを示すキー`:cipher-tagsize`、`:cipher-blocksize`、`:cipher-keysize :cipher-ivsize`ももつ。

`gnutls-symmetric-encrypt` *cipher key iv input* **&optional** *aead_auth* [Function]

*cipher*は `gnutls-ciphers`由来の完全な plist、単なるシンボルキー、またはシンボル名の文字列でもよい。

*key*はバッファー、文字列、あるいは他の方法 (Section 32.27.1 [Format of GnuTLS Cryptography Inputs], page 839 を参照) でも指定できる。*key*が文字列なら使用後に消去される。

iv、*input*、およびオプションの *aead_auth*はバッファー、文字列、または他の方法 (Section 32.27.1 [Format of GnuTLS Cryptography Inputs], page 839 を参照) でも指定できる。

*aead_auth*は AEAD 暗号、すなわち plist が`:cipher-aead-capable t`をもつ暗号でのみチェックされて、他では無視される。

この関数エラー時には `nil`、*cipher*や *key*、*iv*や *input*が無効だったり、AEAD 暗号で *aead_auth*が指定されてそれが無効な場合には Lisp エラーをシグナルする。

成功時にはバイナリー文字列 (出力) と使用される IV をリターンする。

`gnutls-symmetric-decrypt` *cipher key iv input* **&optional** *aead_auth* [Function]

*cipher*は `gnutls-ciphers`由来の完全な plist、単なるシンボルキー、またはシンボル名の文字列でもよい。

*key*はバッファー、文字列、あるいは他の方法 (Section 32.27.1 [Format of GnuTLS Cryptography Inputs], page 839 を参照) でも指定できる。*key*が文字列なら使用後に消去される。

iv、*input*、およびオプションの *aead_auth*はバッファー、文字列、または他の方法 (Section 32.27.1 [Format of GnuTLS Cryptography Inputs], page 839 を参照) でも指定できる。

*aead_auth*は AEAD 暗号、すなわち plist が`:cipher-aead-capable t`をもつ暗号でのみチェックされて、他では無視される。

この関数は解読エラー時には `nil`、*cipher*や *key*、*iv*や *input*が無効だったり、AEAD 暗号で指定された *aead_auth*が無効な場合には Lisp エラーをシグナルする。

成功時にはバイナリー文字列 (出力) と使用される IV をリターンする。

32.28 HTML と XML の解析

ビルトインの libxml2 サポートつきで Emacs をコンパイルできます。

`libxml-available-p` [Function]

この関数はビルトイン libxml2 サポートが利用可能なら非 `nil`をリターンする。

libxml2 サポートが利用可能なら、HTML や XML のテキストを Lisp オブジェクトツリーにパースするために以下の関数を利用できます。

`libxml-parse-html-region` *start end* **&optional** *base-url discard-comments* [Function]

この関数は *start*と *end*の間のテキストを HTML としてパースして、HTML パースツリー *(parse tree)* を表すリストをリターンする。これは構文誤りにたいして強力に対処することにより、現実世界の HTML の処理を試みる。

オプション引数 *base-url*が非 `nil`なら、それはリンク内に出現する相対 URL にたいするベース URL を指定する文字列であること。

オプション引数 *discard-comments*が非 `nil`なら、すべてのトップレベルのコメントを破棄する (この引数は時代遅れであり Emacs の将来のバージョンで削除されるだろう。コメントの削除にはパース関数の呼び出し前にデータにユーティリティ関数 `xml-remove-comments`を使用すること)。

パースツリー内では各 HTML ノードは 1 つ目の要素がノード名を表すシンボル、2 つ目の要素がノード属性の alist、残りの要素はサブノードであるようなリストにより表される。

以下の例でこれを示す。以下の (不正な)HTML ドキュメントを与えると:

```
<html><head></head><body width=101><div class=thing>Foo<div>Yes
```

`libxml-parse-html-region`呼び出しにより以下の DOM (document object model) がリターンされる:

```
(html nil
 (head nil)
 (body ((width . "101"))
  (div ((class . "thing"))
   "Foo"
   (div nil
    "Yes"))))
```

`shr-insert-document` *dom* [Function]

この関数は *dom*内のパース済み HTML をカレントバッファー内に描画する。引数 *dom*は `libxml-parse-html-region`で生成されるようなリストであること。この関数はたとえば *The Emacs Web Wowser Manual* により使用される。

`libxml-parse-xml-region` *start end* **&optional** *base-url discard-comments* [Function]

この関数は `libxml-parse-html-region`と同様だが、HTML ではなく XML(構文についてより厳格) としてテキストをパースする点が異なる。

32.28.1 ドキュメントオブジェクトモデル

`libxml-parse-html-region` (およびその他の XMLパース関数) がリターンする DOMはツリー構造です。このツリー構造ではそれぞれのノードがノード名 (タグ *(tag)* と呼ばれる) をもち、オプションで key/value 値からなる属性 *(attribute)* リスト、その後に子ノード *(child nodes)* が続きます。子ノードは文字列か DOMオブジェクトのいずれかです。

```
(body ((width . "101"))
 (div ((class . "thing"))
  "Foo"
```

```
(div nil
 "Yes")))
```

dom-node *tag* **&optional** *attributes* **&rest** *children* [Function]

この関数はタイプ *tag*の DOMノードを作成する。もし *attributes*が与えられたら、それは key/value ペアのリストであること。もし *children*が与えられたら、それは DOMノードであること。

この構造を処理するために以下の関数を使用できます。それぞれの関数は DOMノードかノードのリストを受け取ります。後者の場合には、そのリストの最初のノードだけが使用されます。

シンプルなアクセサー

dom-tag *node*
: ノードのタグ (“ノード名” とも呼ばれる) をリターンする。

dom-attr *node attribute*
: ノードの属性の値をリターンする。以下は一般的な使用例：

```
(dom-attr img 'href)
=> "https://fsf.org/logo.png"
```

dom-children *node*
: ノードのすべての子をリターンする。

dom-non-text-children *node*
: ノードのすべての非文字列の子をリターンする。

dom-attributes *node*
: ノードの属性の key/value ペアのリストをリターンする。

dom-text *node*
: ノードのすべてのテキスト的な要素を連結された文字列としてリターンする。

dom-texts *node*
: ノードのすべてのテキスト的な要素、およびノードのすべての子のテキスト的な要素を、連結された文字列として再帰的にリターンする。この関数はテキスト的な要素の間に挿入するオプションのセパレーターも受け取る。

dom-parent *dom node*
: *dom*内での *node*の親をリターンする。

dom-remove *dom node*
: *dom*から *node*を削除する。

以下は DOMを変更するための関数です。

dom-set-attribute *node attribute value*
: ノードの *attribute*に *value*をセットする。

dom-append-child *node child*
: *node*の最後の子として *child*を追加する。

dom-add-child-before *node child before*
: *node*の子リストのノード *before*の前に *child*を追加する。*before*が `nil`なら、*child*が最初の子になる。

`dom-set-attributes` *node attributes*
: ノードのすべての属性を新たな key/value リストに置き換える。

以下はDOM内の要素を検索する関数です。これらはマッチしたノードのリストをリターンします。

`dom-by-tag` *dom tag*
: *dom*内のタイプが *tag*のすべてのノードをリターンする。典型的な使用例は:

```
(dom-by-tag dom 'td)
=> '((td ...) (td ...) (td ...))
```

`dom-by-class` *dom match*
: *dom*内のクラス名が正規表現 *match*にマッチするすべてのノードをリターンする。

`dom-by-style` *dom style*
: *dom*内のスタイルが正規表現 *match*にマッチするすべてのノードをリターンする。

`dom-by-id` *dom style*
: *dom*内の ID が正規表現 *match*にマッチするすべてのノードをリターンする。

`dom-search` *dom predicate*
: *predicate*が非 `nil`値をリターンした *dom*内のすべてのノードをリターンする。*predicate*はテストされるノードをパラメーターとして呼び出される。

`dom-strings` *dom*
: *dom*内のすべての文字列をリターンする。

ユーティリティ関数:

`dom-pp` *dom* `&optional` *remove-empty*
: ポイント位置の *dom*にたいしてプリティプリント (pp: 優雅なプリント) を行う。*remove-empty*なら空白文字だけを含むテキスト的ノードはプリントしない。

32.29 JSON 値の解析と生成

JSON (*JavaScript Object Notation*) のサポートつきで Emacs をコンパイルした場合には、Lisp オブジェクトと JSON 値との間で変換を行う関数がいくつか提供されます。任意の JSON 値を Lisp オブジェクトに変換できますが、その逆は成り立ちません。具体的には:

- JSON は `true`、`null`、`false`という 3 つのキーワードを使用する。`true`はシンボル `t`を表す。デフォルトでは残り 2 つのキーワードはそれぞれシンボル`:null`、`:false`で表される。
- JSON には浮動小数点数しかない。これらは Lisp 整数と Lisp 浮動小数点数の両方を表すことができる。
- JSON 文字列は常に UTF-8 でエンコードされた Unicode 文字列になる。Lisp 文字列は非 Unicode 文字を含むことができる。
- JSON のシーケンス型は配列のみ。JSON 配列は Lisp ベクターを使用して表される。
- JSON のマップ型はオブジェクトのみ。JSON オブジェクトは Lisp のハッシュテーブル、alist、plists を使用して表される。alist や plist が同じキーで複数の要素を含む際には、Emacs は `assq`の挙動にしたがいシリアライズに最初の要素だけを使用する。

alist と plist の両方で有効な `nil`は、空の JSON オブジェクト`{}`を表すことに注意してください。`null`や `false`、空の配列は JSON ではすべて異なる値です。

JSON で何らかの Lisp オブジェクトを表現できなければ、シリアライゼーション関数はタイプ `wrong-type-argument`のエラーをシグナルします。パース関数も以下のエラーをシグナルする可能性があります:

`json-end-of-file`
: 入力テキストの早すぎる終端に遭遇した際にシグナルされる。

`json-trailing-content`
: パース済みの最初の JSON オブジェクトの後で予期せぬ入力に遭遇した際にシグナルされる。

`json-parse-error`
: 無効な JSON 構文に遭遇した際にシグナルされる。

トップレベルの値 (配列とオブジェクト) だけが JSON にシリアライズできます。これらのトップレベル値内のサブオブジェクトは任意のタイプにすることができます。同様にパース関数はベクター、ハッシュテーブル、alist、plist だけをリターンすます。

`json-serialize` *object* **&rest** *args* [Function]
: この関数は *object*の JSON 表現を含む Lisp 文字列を新たにリターンする。引数 *args*はキーワード/引数のペアからなるリスト。以下のキーワードが可能:

  `:null-object`
  : 値は JSON キーワードの `null`を表すために使用する Lisp オブジェクトを決定する。デフォルトはシンボル`:null`。

  `:false-object`
  : 値は JSON キーワードの `false`を表すために使用する Lisp オブジェクトを決定する。デフォルトはシンボル`:false`。

`json-insert` *object* **&rest** *args* [Function]
: このコマンドはカレントバッファーのポイントの前に *object*の JSON 表現を挿入する。引数 *args*は `json-parse-string`の場合と同様に扱われる。

`json-parse-string` *string* **&rest** *args* [Function]
: この関数は *string* (Lisp 文字列でなければならない) 内の JSON 値をパースする。*string*に有効な JSON オブジェクトが含まれていなければ、この関数は `json-parse-error`エラーをシグナルする。

  引数 *args*はキーワード/引数のペアのリスト。以下のキーワードが許されている:

  `:object-type`
  : 値は JSON オブジェクトのキーと値のマッピングを表現するために使用する Lisp オブジェクトを決定する。文字列をキーとするハッシュテーブル `hash-table` (デフォルト)、シンボルをキーとする alist を使用する `alist`、キーワードシンボルをキーとする plist を使用する `plist`のいずれかが可能。

  `:array-type`
  : 値は JSON 配列の表現に使用する Lisp オブジェクトを決定する。Lisp 配列を使用する `array` (デフォルト)、またはリストを使用する `list`のいずれかが可能。

  `:null-object`
  : 値は JSON キーワードの `null`を表すために使用する Lisp オブジェクトを決定する。デフォルトはシンボル`:null`。

`:false-object`
値は JSON キーワードの `false`を表すために使用する Lisp オブジェクトを決定する。デフォルトはシンボル`:false`。

`json-parse-buffer` **&rest** *args* [Function]
この関数はカレントバッファーのポイント位置の文字列から、次の JSON 値を読み取る。値に有効な JSON オブジェクトが含まれていれば値の直後にポイントを移動、それ以外なら`json-parse-error`エラーをシグナルしてポイントは移動しない。引数 *args*は`json-parse-string`の場合と同様に解釈される。

32.30 JSONRPC による対話

`jsonrpc`ライブラリーは`https://www.jsonrpc.org/`に記載された JSONRPC仕様を実装します。JSONRPC はその名前が示すように、Lisp との間で相互に変換可能な JSONオブジェクトを中心に設計された、遠隔手続呼出 *(Remote Procedure Call)* のための汎用プロトコルです。

32.30.1 概観

spec (`https://www.jsonrpc.org/`) から引用すると JSONRPC は、"同一プロセス、ソケットや http、多くのさまざまなメッセージパッシング環境において使用可能という概念においてトランスポート非依存"です。

この非依存性をモデル化するために、`jsonrpc`ライブラリーはリモートの JSON のエンドポイントへの接続の表現に`jsonrpc-connection`クラスのオブジェクトを使用します (Emacs のオブジェクトシステムの詳細は *EIEIO* を参照)。これはオブジェクト指向の現代的な用語では “抽象的 (abstract)” なクラス、すなわち有用な接続オブジェクトの実クラスは常に`jsonrpc-connection`のサブクラスになります。それにも関わらず、`jsonrpc-connection`クラスを中心に 2 つの API を個別に定義できます。

1. JSONRPC アプリケーション構築用のユーザーインターフェース

   このシナリオでは JSONRPC アプリケーションは`jsonrpc-connection`の具象サブクラスを選択して、`make-instance`を使用することによりサブクラスのオブジェクト作成を行う。JSONRPC アプリケーションはリモートのエンドポイントとの対話を開始するために、関数`jsonrpc-notify`、`jsonrpc-request`、および/または`jsonrpc-async-request`にこのオブジェクトを渡す。一般的には非同期に到達するリモートで開始された対話を処理するために、インスタンス化には`:request-dispatcher`と`:notification-dispatcher`の初期化引数 (initarg) を含める必要があり、これらはいずれも接続オブジェクト、リモートで呼び出された JSONRPC メソッドを命名するシンボル、JSONRPC の `params`オブジェクトという 3 つの引数を受け取る関数である。

   `:request-dispatcher`として渡された関数は、ローカルのエンドポイントからのリプライ (この場合では構築するプログラム) を除いた、リモートのエンドポイントのリクエストを処理する役目を担う。その関数の内部では局所的なリターン (通常のリターン) と非局所的なリターン (エラーによるリターン) が起こり得る。局所的なリターン値は JSON としてシリアライズ可能な Lisp オブジェクトでなければならない (Section 32.29 [Parsing JSON], page 844 を参照)。これは成功レスポンスを決定するとともに、オブジェクトは JSONRPC の `result`オブジェクトとしてサーバーにフォワードされる。関数`jsonrpc-error`の呼び出しにより達成される非局所的なリターンは、エラーレスポンスをサーバーに送信する。JSONRPC の `error`に付随する詳細には、`jsonrpc-error`に渡されるものすべてが含まれる。他のタイプの予期せぬエラーからトリガーされた非局所的なリターンでも、(`debug-on-error`をセットしていなければ) エラー

レスポンスを送信して、この場合には Lisp デバッガが呼び出される。Section 18.1.1 [Error Debugging], page 296 を参照のこと。

2. JSONRPC トランスポート実装構築用の継承インターフェース

   このシナリオでは基盤として異なるトランスポートストラテジーを実装するために `jsonrpc-connection`をサブクラス化する (サブクラス化する方法についての詳細は Section "Inheritance" in `eieio`を参照)。その後にアプリケーション構築インターフェースのユーザーは、(`make-instance`関数を使用して) その具象クラスのオブジェクトをインスタンス化して、そのストラテジーを使用して JSONRPC エンドポイントに接続できる。

   この API には必須部分とオプション部分がある。

   ユーザーが JSONRPC コンタクト (通知やリクエスト) を開始したり、エンドポイントにリプライできるようにするためには、そのサブクラスは `jsonrpc-connection-send`メソッドを実装しなければならない。

   同様に 3 種類のリモートコンタクト (リクエスト、通知、ローカルリクエストへの応答) を処理するために、トランスポート実装はワイヤー (wire: 通信ライン) 上への新たな JSONRPC メッセージの通知後に、(その`"`ワイヤー`"`が何であれ) 関数 `jsonrpc-connection-receive`が呼び出されるように計らわなければならない。

   最後にオプジョンとして `jsonrpc-shutdown`メソッドと `jsonrpc-running-p`メソッドの概念をトランスポートに適用する場合には、`jsonrpc-connection`サブクラスはそれらのメソッドを実装する必要がある。それらを行う場合には、ワイヤー上でメッセージをリッスン (listen) するために使用するシステムリソース (プロセス、タイマー等) は `jsonrpc-shutdown`で解放する必要がある (それらのリソースは `jsonrpc-running-p`が非 nil の間だけ必要なはずなので)。

32.30.2 プロセスベースの JSONRPC 接続

`jsonrpc`ライブラリーには利便性のために、ローカルサブプロセス (標準入力と標準出力を使用) や TCP ホスト (ソケットを使用)、または Emacs のプロセスオブジェクトが表現可能な他のリモートのエンドポイント (Chapter 38 [Processes], page 934 を参照) と対話可能なビルトインの `jsonrpc-process-connection`トランスポート実装が付属しています。

このトランスポートを使用することにより JSONRPC メッセージはワイヤー上にプレーンテキストとしてエンコードされて、"Content-Length" のように何らかの基本的な HTTP スタイルのエンベロープヘッダーが前置されます。

この JSONRPC 最上層のトランスポートスキームを使用したアプリケーションの例は、Language Server Protocol (`https://microsoft.github.io/language-server-protocol/specification`) を参照してください。

`:request-dispatcher`と`:notification-dispatcher`という必須の初期化引数 (initarg) に加えて、`jsonrpc-process-connection`クラスのユーザーは `make-instance`へのキーワード/値ペアとして以下の初期化引数を渡す必要があります:

`:process` 値は生きたプロセスオブジェクト、またはそのようなオブジェクトを生成する引数のない関数でなければならない。プロセスオブジェクトを渡された場合には、そのオブジェクトには事前に確立された接続が含まれていることが期待される。それ以外の場合には、オブジェクトの作成直後に関数が呼び出される。

`:on-shutdown`
値は `jsonrpc-process-connection`オブジェクトを単一の引数とする関数でなければならない。この関数は背後にあるプロセスオブジェクトの削除 (`jsonrpc-shutdown`による故意の削除、または何らかの外部要因による予期せぬ削除) の後に呼び出される。

32.30.3 JSONRPC の JSON オブジェクトフォーマット

JSONRPC の JSON と Lisp の plist(Section 5.9 [Property Lists], page 92 を参照) は交換することができます。JSON 互換の plist をディスパッチャ関数に渡したり、同様に JSON 互換の plist を `jsonrpc-notify`、`jsonrpc-request`、`jsonrpc-async-request`に渡すことができます。

plist 処理を容易にするために、このライブラリーは `cl-lib`ライブラリー (*Common Lisp Extensions for GNU Emacs Lisp* を参照) の積極的に使用しており、そのクライアントにたいしてもこれを同じように提案します (強制ではない)。以下の例のように JSON オブジェクトの非構造化用 lambda の作成にはマクロ `jsonrpc-lambda`を使用できます:

```
(jsonrpc-async-request
 myproc :frobnicate `(:foo "trix")
 :success-fn (jsonrpc-lambda (&key bar baz &allow-other-keys)
               (message "Server replied back with %s and %s!"
                        bar baz))
 :error-fn (jsonrpc-lambda (&key code message _data)
             (message "Sadly, server reports %s: %s"
                      code message)))
```

32.30.4 遅延された JSONRPC リクエスト

多くの RPC状況下において、対話中の 2 つのエンドポイント間での同期は、RPC アプリケーションを正しくデザインするために問題となります。同期が必要な際にはリクエスト (ブロックする)、不必要なら通知で十分です。しかしこれらのエンドポイントのいずれかで Emacs が動作している際にはリモートエンドポイントの状態に不確実性が依然として残っているので、(タイマーやプロセスに関連する) 非同期イベントがトリガーされる可能性があります。さらにこれらのイベントへの対応では、イベント固有の性質により、同期の要求が限定されるかもしれません。

`jsonrpc-request`や `jsonrpc-async-request`にたいするキーワード引数`:deferred`は特定のリクエストに同期が必要なことを呼び出し側が示せるようにして、リクエストの実際の発行は何らかの条件が満足されるまで遅延できるようにデザインされています。あるリクエストへの`:deferred` 指定はリクエストが遅延されるのではなく、遅延される可能性があることを意味します。リクエストが即座に送信されなければ、エンドポイントにたいして他のメッセージの受信や送信を行う際のように、通信中の特定タイミングで `jsonrpc`はリクエストを送る新たな試みを行います。

リクエストを送信するすべての試みの前にアプリケーション固有の条件がチェックされます。`jsonrpc`ライブラリーがこれらの条件を知ることはできないので、それらを指定するためにプログラムはジェネリック関数 `jsonrpc-connection-ready-p`を使用できます (Section 13.8 [Generic Functions], page 222 を参照)。この関数のデフォルトメソッドは `t`をリターンしますが、これをオーバーライドして渡された引数 (`jsonrpc-connection`オブジェクト。Section 32.30.1 [JSONRPC Overview], page 846 を参照) とキーワード引数`:deferred`として渡されたすべて値にもとづいて `nil`をリターンするメソッドを追加できます。

32.31 グループのアトミックな変更

データベース用語においてのアトミック *(atomic:* 原子的、不可分*)* な変更とは、全体として成功か失敗をすることはできるが、部分的にはできない個別の変更のことです。Lisp プログラムは単一もしくは複数のバッファーにたいする一連の変更をアトミック変更グループ *(atomic change group)* にすることができます。これはその一連の変更全体がそれらのバッファーに適用されるか、またはエラーの場合は何も適用されないかの、いずれかであることを意味します。

すでにカレントであるような単一のバッファーにたいしてこれを行うには、以下のように単に変更を行うコードの周囲に atomic-change-groupの呼び出しを記述します:

```
(atomic-change-group
  (insert foo)
  (delete-region x y))
```

atomic-change-groupの body 内部でエラー (またはその他の非ローカル exit) が発生した場合には、その body の実行の間にそのバッファーでのすべての変更が行われなかったことになります。この類の変更グループは他のバッファーには影響を与えず、それらのバッファーにたいする変更はそのまま残されます。

さまざまなバッファー内で行った変更から 1 つのアトミックグループを構成する等、より複雑な何かを必要とする場合には、atomic-change-groupが使用する、より低レベルな関数を直接呼び出さなければなりません。

prepare-change-group **&optional** *buffer* [Function]
この関数は *buffer* (デフォルトはカレントバッファー) にたいする変更グループをセットアップする。これはその変更グループを表す handle をリターンする。変更グループを activate したり、その後でそれを完了するためにはこの handle を使用しなければならない。

変更グループを使用するためには、それを *activate(*アクティブ化*)* しなければなりません。これは *buffer*のテキストを変更する前に行わなければなりません。

activate-change-group *handle* [Function]
これは *handle*が指定する変更グループを active にする。

変更グループを activate した後には、そのバッファー内で行ったすべての変更は変更グループの一部となります。そのバッファー内で目論んでいたすべての変更を行ったら、変更グループを *finish(*完了*)* しなければなりません。すべての変更を受け入れる (確定する) か、すべてをキャンセルするという 2 つの方法により、これを行うことができます。

accept-change-group *handle* [Function]
この関数は *handle*により指定される変更グループ内のすべての変更にたいして、finalize することにより変更を受け入れる。

cancel-change-group *handle* [Function]
この関数は *handle*により指定される変更グループ内のすべての変更をキャンセルして undo する。

グループが常に確実に finish されるようにするために、コードでは unwind-protectを使用するべきです。activate-change-groupの呼び出しは、実行直後にユーザーが *C-g*をタイプする場合に備えて unwind-protect内部にあるべきです (これが prepare-change-groupと activate-change-groupが別関数となっている 1 つの理由。なぜなら通常は unwind-protect開始前に prepare-change-groupを呼び出すであろうから)。グループを一度 finish したら、その handle を再度使用してはなりません。特に同じ変更グループを 2 回 finish しないでください。

複数バッファー変更グループ (multibuffer change group) を作成するためには、カバーしたいバッファーそれぞれで prepare-change-groupを一度呼び出してから、以下のようにリターン値を結合するために nconcを使用してください:

```
(nconc (prepare-change-group buffer-1)
       (prepare-change-group buffer-2))
```

その後は 1 回の activate-change-group呼び出しで複数変更グループをアクティブにして、1 回の accept-change-groupか cancel-change-group呼び出しでそれを finish してください。

同一バッファーにたいするネストされた複数の変更グループ使用は、あなたが期待するであろう通りに機能します。同一バッファーにたいするネストされていない変更グループの使用により Emacs が混乱した状態になるので、これが発生しないようにしてください。与えられた何らかのバッファーにたいして最初に開始した変更グループは最後に finish する変更グループです。

32.32 フックの変更

以下のフック変数によりバッファー (これらをバッファーローカルにした場合には特定のバッファー) での変更にたいして、通知を受け取るようにアレンジすることができます。テキストの特定部分にたいする変更の検出方法については Section 32.19.4 [Special Properties], page 820 も参照してください。

これらのフック内で使用する関数は、もしそれらが正規表現を使用して何かを行う場合にはマッチしたデータの保存とリストアを行うべきです。さもないとそれらが呼び出す編集処理に奇妙な方法で干渉するでしょう。

before-change-functions [Variable]
: この変数は Emacs がバッファー変更を行おうとする際に呼び出す関数のリストを保持する。各関数は変更されようとするリージョンの先頭と終端を整数で表す 2 つの引数を受け取る。変更されようとするバッファーは関数の呼び出しの際には常にカレントバッファーである。

after-change-functions [Variable]
: この変数は Emacs がバッファー変更を行った後に呼び出す関数のリストを保持する。各関数は正に変更されたリージョンの先頭と終端、およびその変更前に存在したテキストの長さという 3 つの引数を受け取る。これら 3 つの変数は、すべて整数。変更されたバッファーは関数の呼び出しの際には常にカレントバッファーである。

  古いテキストの長さは、変更される前のテキストでのテキストの前後のバッファー位置の差で与えられる。変更されたテキストでは、その長さは単に最初の 2 つの引数の差で与えられる。

これらの関数は*Messages*バッファーへのメッセージの出力では呼び出されず、特定の処理用に Emacs が作成する内部的なバッファーのような Lisp プログラムからは可視であるべきではないバッファーへの変更でも呼び出されません。

バッファーを変更するプリミティブのほとんどは、バランスされたカッコ内でのそれぞれの変更にたいして before-change-functionsと after-change-functionsを 1 回呼び出し、これらのフックにたいする引数は行われた変更を正確に区切ります。しかしフック関数は常にこのように行われると信頼すべきではありません。なぜなら複雑なプリミティブのいくつかは変更を行う前に before-change-functionsを呼び出してから、プリミティブが行なった個別の変更の数にもとづいて after-change-functionsを 0 回以上呼び出すからです。これが発生した場合には、before-change-functions の引数は個別の変更が行われたリージョンを囲むでしょうが、そのようなリージョンが最小である必要はなく、連続した after-change-functions呼び出しそれぞれにたいする引数は変更されたテキスト部分を正確に区切るでしょう。一般的には、before-change か after-change のいずれかのフックを使用して、両方は使用しないことを推奨します。

combine-after-change-calls *body...* [Macro]
: このマクロは普通に *body*を実行するが、もしそれが安全なように見えるなら一連の複数の変更にたいして正に一度、after-change 関数を呼び出すようにアレンジする。

そのバッファーの同じ領域内でプログラムが複数のテキスト変更を行う場合には、その部分のプログラムの周囲でマクロ `combine-after-change-calls`を使用することにより、after-change フック使用中の実行がかなり高速になり得る。after-change フックが最終的に呼び出される際には、その引数は `combine-after-change-calls`の body 内で行われたすべての変更にたいして含むバッファーの範囲を指定する。

警告: フォーム `combine-after-change-calls`の body 内で `after-change-functions` の値を変更してはならない。

警告: 組み合わされた変更がバッファーの広い範囲に点在してに出現する場合でも、これは依然として機能するが推奨できない。なぜならこれは、ある変更フック関数を非効率的な挙動へと導くかもしれないからである。

`combine-change-calls` *beg end body...* [Macro]

これは通常のように *body*を実行するが、`before-change-functions`および `after-change-functions`の呼び出しをトリガーしないすべてのバッファー変更を除く。かわりに *beg*と *end*で囲まれるリージョンにたいしてこれらのフックそれぞれを 1 回呼び出し、*body*が変更するサイズを反映したパラメーターを `after-change-functions`に与える。

このマクロの結果は *body*のリターンした結果。

このマクロはある関数がバッファーにたいして繰り返し多数の変更を行う可能性があり、このマクロ以外では個別のバッファー変更ごとにそれらの変更フックを実行するために実行に長時間を要する際に有用。Emacs 自身は、たとえばコマンド `comment-region`や `uncomment-region` の中でこのマクロを使用している。

警告: *body*内で `before-change-functions`や `after-change-function`の値を変更してはならない。

警告: *beg*と *end*で指定したリージョン外部でのバッファー変更は何も行ってはならない。

`first-change-hook` [Variable]

この変数は以前は未変更の状態だったバッファーが変更された際は常に実行されるノーマルフック。

`inhibit-modification-hooks` [Variable]

この変数が非 `nil`ならすべての変更フックは無効。それらは何も実行されない。これはこのセクションで説明したすべてのフック変数、同様に特定のスペシャルテキストプロパティ(Section 32.19.4 [Special Properties], page 820 を参照) とオーバーレイプロパティ (Section 39.9.2 [Overlay Properties], page 1002 を参照) にアタッチされたフックに影響を与える。

これらの同一フック変数上の関数の実行の間、バッファー変更によるデフォルトの変更フックが他の変更フック実行中に実行されないように、この変数は非 `nil`にバインドされる。それ自体が変更フックから実行される特定のコード断片内で変更フックを実行したければ、`inhibit-modification-hooks`を `nil`にローカルに再バインドすること。しかしこれを行うことで変更フックが再帰的に呼び出されるかもしれないのでそれに備えること (たとえばフックが何も行わないようにいくつかの変数をバインドする)。

バッファーのテキストコンテンツに永続的な変更をもたらさない変更 (たとえばフェイス変更や一時的な変更) だけにこの変数をバインドすることを推奨する。一連の変更の間は変更フックを遅延させる必要がある (通常は性能上な理由による) なら、かわりに `combine-change-calls` や `combine-after-change-calls`を使用すること。

33 非 ASCII文字

このチャプターは文字に関する特別な問題と、それらが文字列やバッファーに格納される方法について網羅しています。

33.1 テキストの表現方法

Emacs のバッファーと文字列は、既知のスクリプトで記述されたほとんどすべてのテキストをユーザーがタイプしたり表示できるように、多種多様な言語の広大な文字レパートリーをサポートします。

多種多様な文字やスクリプトをサポートするために、Emacs は *Unicode* 標準 *(Unicode Standard)* に厳密にしたがいます。Unicode 標準はすべての文字それぞれにたいして、コードポイント *(codepoint)* と呼ばれる一意な番号を割り当てています。コードポイントの範囲は Unicode、または Unicode コード空間 *(codespace)* により定義され、範囲は `0..#x10FFFF`(16 進表記、範囲両端を含む) です。Emacs はこれを範囲`#x110000..#x3FFFFF`のコードポイント範囲に拡張します。この範囲は Unicode として統一されていない文字や、文字として解釈できない *8* ビット *raw* バイト *(raw 8-bit bytes)* を表すために使用します。したがって Emacs 内の文字コードポイントは 22 ビットの整数になります。

メモリー節約のために、Emacs はバッファーや文字列内のテキスト文字にたいするコードポイントである 22 ビットの整数を固定長で保持しません。かわりに Emacs は文字の内部表現として可変長を使用します。これはそのコードポイントの値に応じて、各文字を 5 ビットから 8 ビットのバイトシーケンスとして格納するものです [1]。たとえばすべての ASCII文字は 1 バイト、Latin-1 文字は 2 バイトといった具合です。わたしたちはこれをテキストのマルチバイト *(multibyte)* 表現と呼んでいます。

Emacs 外部では ISO-8859-1、GB-2312、Big-5 等のような多種の異なるエンコーディングで文字を表すことができます。Emacs はバッファーや文字列へのテキスト読み込み時、およびディスク上のファイルへのテキスト書き込みや他プロセスへの引き渡し時に、これらの外部エンコーディングと内部表現の間で適切な変換を行います。

Emacs がエンコード済みテキストや非テキストデータをバッファーや文字列に保持したり操作する必要がある場合も時折あります。たとえば Emacs がファイルを visit する際には、まずそのファイルのテキストをそのままバッファーに読み込んで、その後にのみそれを内部表現に変換します。この変換前にバッファーに保持されいるのはエンコード済みテキストです。

Emacs に関する限り、エンコードされたテキストは実際のテキストではなく 8 ビット raw バイトです。エンコード済みテキストを保持するバッファーや文字列は、Emacs がそれらを個々のバイトシーケンスとして扱うことから、ユニバイト *(unibyte)* のバッファー (文字列) と呼んでいます。Emacs は通常はユニバイトのバッファーや文字列を`\237`のような 8 進コードで表示します。エンコード済みテキストやバイナリー非テキストデータを処理する場合を除いて、ユニバイトバッファーとユニバイト文字列は決して使用しないよう推奨します。

バッファーでは変数 `enable-multibyte-characters`のバッファーローカルな値が使用する表現を指定します。文字列での表現は文字列構築時に判断して、それを文字列内に記録します。

`enable-multibyte-characters` [Variable]

この変数はカレントバッファーのテキスト表現を指定する。非 `nil`ならバッファーはマルチバイトテキスト、それ以外ならエンコード済みユニバイトテキスト、またはバイナリー非テキストデータが含れる。

---

[1] この内部表現は任意の Unicode コードポイントを表すための、*UTF-8* と呼ばれる Unicode 標準によるエンコーディングの 1 つにもとづいたものですが、8 ビット raw バイトおよび Unicode に統一されていない文字を使用する追加のコードポイントを表現するために Emacs は UTF-8 を拡張しています。

この変数は直接セットできない。バッファーの表現の変更には、かわりに関数 `set-buffer-multibyte`を使用すること。

`position-bytes` *position* [Function]
バッファー位置は文字単位で測られる。この関数はカレントバッファー内のバッファー位置を、それに対応するバイト位置でリターンする。これはバッファー先頭を 1 としてバイト単位で増加方向に数えられる。*position*が範囲外なら値は `nil`。

`byte-to-position` *byte-position* [Function]
カレントバッファー内で与えられた *byte-position*に対応するバッファー位置を文字単位でリターンする。*byte-position*が範囲外なら値は `nil`。マルチバイトバッファーでは *byte-position* の任意の値が文字境界上になく、1 文字として表現されたマルチバイトシーケンス内にあるかもしれない。この場合には関数はその文字のマルチバイトシーケンスが *byte-position*を含むようなバッファー位置をリターンする。言い換えるとこの値は同じ文字に属するすべてのバイト位置にたいして変化しない。

以下の 2 つの関数はバッファーに visit されているファイル内でのバイトオフセットとバッファー位置を Lisp プログラムがマッピングする際に有用です。

`bufferpos-to-filepos` *position* **&optional** *quality coding-system* [Function]
この関数は `position-bytes`と似ているがカレントバッファー内でのバイト位置ではなく、バッファー内の *position*により与えられる文字に対応するカレントバッファーのファイル先頭からのオフセットをリターンする点が異なる。この変換にはバッファーのファイル内でテキストがエンコードされる方法を知ることが要求される。これが *coding-system*引数の存在意義であり、デフォルトは `buffer-file-coding-system`の値。オプション引数 *quality*は結果のあるべき正確さを指定する。これは以下いずれかであること:

`exact` 正確な結果でなければならない。関数は高価で低速になり得るバッファーの大きな範囲のエンコードとデコードを要するかもしれない。

`approximate`
近似的な値が可能。関数は高価な処理を回避して不正確な結果をリターンするかもしれない。

`nil` 正確な結果に高価な処理を要するなら、関数は近似値ではなく `nil`をリターンするだろう。これは引数が省略された場合のデフォルト。

`filepos-to-bufferpos` *byte* **&optional** *quality coding-system* [Function]
この関数は *byte* (ファイル先頭からの 0 基準のバイトオフセット) が指定するファイル位置に対応するバッファー位置をリターンする。この関数は `bufferpos-to-filepos`が行う変換と逆の処理を行う。オプション引数 *quality*と *coding-system*のもつ意味と値は `bufferpos-to-filepos`の場合と同様。

`multibyte-string-p` *string* [Function]
*string*がマルチバイト文字列なら `t`、それ以外は `nil`をリターンする。この関数は *string*が文字列以外でも `nil`をリターンする。

`string-bytes` *string* [Function]
この関数は *string*内のバイト数をリターンする。*string*がマルチバイト文字列なら、これは `(length` ***string***`)`より大きいかもしれない。

`unibyte-string` **&rest** *bytes* [Function]
この関数は引数 *bytes*をすべて結合して、その結果をユニバイト文字列で作成する。

33.2 マルチバイト文字の無効化

デフォルトでは Emacs はマルチバイトモードで開始されます。Emacs はマルチバイトシーケンスを使用して非 ASCII文字を表現する内部エンコーディングを使用することにより、バッファーや文字列のコンテンツを格納します。マルチバイトモードでは、サポートされるすべての言語とスクリプトを使用できます。

非常に特別な状況下においては、特定のバッファーでマルチバイト文字のサポートを無効にしたいときがあるかもしれません。あるバッファーにおいてマルチバイト文字が無効になっているときには、それをユニバイトモード *(unibyte mode)* と呼びます。ユニバイトモードではバッファー内の各文字は 0 から 255(8 進の 0377) の範囲の文字コードをもちます。0 から 127(8 進の 0177) は ASCII文字、128 から 255(8 進の 0377) は非 ASCII文字を表します。

特定のファイルをユニバイト表現で編集するためには、`find-file-literally`を使用してファイルを visit します。Section 25.1.1 [Visiting Functions], page 530 を参照してください。マルチバイトバッファーをファイルに保存してバッファーを kill した後に、再びそのファイルを `find-file-literally`で visit することによりマルチバイトバッファーをユニバイトに変換できます。かわりに *C-x RET c*(`universal-coding-system-argument`) を使用して、ファイルを visit または保存するコーディングシステムとして‘`raw-text`’を指定することもできます。Section “Specifying a Coding System for File Text” in *GNU Emacs Manual* を参照してください。`find-file-literally`とは異なり、‘`raw-text`’としてファイルを visit してもフォーマット変換、解凍、自動的なモード選択は無効になりません。

バッファーローカル変数 `enable-multibyte-characters`はマルチバイトバッファーなら非`nil`、ユニバイトバッファーなら `nil`です。マルチバイトバッファーかどうかはモードラインにも示されます。グラフィカルなディスプレイでのマルチバイトバッファーには文字セットを示すモードライン部分と、そのバッファーがマルチバイトであること (とそれ以外の事項) を告げるツールチップがあります。ユニバイトバッファーでは文字セットのインジケーターはありません。したがって (グラフィカルなディスプレイ使用時の) ユニバイトバッファーでは入力メソッドを使用していなければ、visit しているファイルの行末変換 (コロン、バックスラッシュ等) の標識の前には通常は何も標識がありません。

特定のバッファーでマルチバイトサポートをオフに切り替えるには、そのバッファー内でコマンド `toggle-enable-multibyte-characters`を呼び出してください。

33.3 テキスト表現の変換

Emacs はユニバイトテキストをマルチバイトに変換できます。マルチバイトテキストに含まれるのが ASCIIと 8 ビット raw バイトだけという条件つきでマルチバイトテキストからユニバイトへの変換もできます。一般的にこれらの変換はバッファーへのテキスト挿入時、または複数の文字列を 1 つの文字列に合成してテキストに put するときに発生します。文字列のコンテンツを明示的にいずれかの表現に変換することもできます。

Emacs はそのテキストの構成にもとづいて文字列の表現を選択します。一般的なルールではユニバイトテキストが他のマルチバイトテキストと組み合わされていればマルチバイト表現のほうがより一般的であり、ユニバイトテキストのすべての文字を保有できるのでユニバイトテキストをマルチバイトテキストに変換します。

バッファーへのテキスト挿入時に Emacs はそのバッファーの `enable-multibyte-characters`の指定にしたがってテキストをそのバッファーの表現に変換します。特にユニバイトバッファーにマルチバイトテキストを挿入する際には、たとえ一般的にはマルチバイトテキスト内のすべての文字を保持することはできなくても Emacs はテキストをユニバイトに変換します。バッファーコンテンツをマルチバイトに変換するという自然な代替方法は、そのバッファーの表現が自動的にオーバーライドできないユーザーによる選択にもとづく表現であるため許容されません。

ユニバイトテキストからマルチバイトテキストへの変換では ASCII文字は未変更のまま残されて、128 から 255 のコードをもつバイトが 8 ビット raw バイトのマルチバイト表現に変換されます。

マルチバイトテキストからユニバイトテキストへの変換では、すべての ASCIIと 8 ビット文字が、それらの 1 バイト形式に変換されますが、各文字のコードポイントの下位 8 ビット以外は破棄されるために非 ASCII文字の情報は失われます。ユニバイトテキストからマルチバイトテキストに変換してそれをユニバイトに戻せば、元のユニバイトテキストが再生成されます。

以下の 2 つの関数は引数 *string*、またはテキストプロパティをもたない新たに作成された文字列のいずれかをリターンします。

`string-to-multibyte` *string* [Function]
この関数は *string*と同じ文字シーケンスを含むマルチバイト文字列をリターンする。*string* がマルチバイト文字列なら未変更のままそれがリターンされる。この関数は *string*が ASCII 文字と 8 ビット raw バイトだけを含むと仮定する。後者は`#x3FFF80`から`#x3FFFFF`(両端を含む) に対応する 8 ビット raw バイトのマルチバイト表現に変換される (Section 33.1 [Text Representations], page 852 を参照)。

`string-to-unibyte` *string* [Function]
この関数は *string*と同じ文字シーケンスを含むユニバイト文字列をリターンする。*string*に非 ASCII文字が含まれる場合にはエラーをシグナルする。*string*がユニバイト文字列なら未変更のままそれがリターンされる。ASCII文字と 8 ビット文字だけを含む *string*引数にたいしてのみこの関数を使用すること。

`byte-to-string` *byte* [Function]
この関数は文字データ *byte*の単一バイトを含むユニバイト文字列をリターンする。*byte*が 0 から 255 までの整数でなければ、エラーをシグナルする。

`multibyte-char-to-unibyte` *char* [Function]
これはマルチバイト文字 *char*をユニバイト文字に変換してその文字をリターンする。*char*が ASCIIと 8 ビットのいずれでもなければこの関数は −1 をリターンする。

`unibyte-char-to-multibyte` *char* [Function]
これは *char*が ASCIIか 8 ビット raw バイトのいずれかであると仮定してユニバイト文字 ASCII をマルチバイト文字に変換する。

33.4 表現の選択

既存のバッファーや文字列がユニバイトの際に、それらをマルチバイトとして調べたり、その逆を行うことが有用なときがあります。

`set-buffer-multibyte` *multibyte* [Function]
カレントバッファーの表現タイプをセットする。*multibyte*が非 `nil`ならバッファーはマルチバイト、`nil`ならユニバイト。

この関数はバイトシーケンスとして認識時にはバッファーを未変更のままとする。結果として文字として認識時にはコンテンツを変更できる。たとえばマルチバイト表現では 1 文字として扱われる 3 バイトのシーケンスは、ユニバイト表現では 3 文字として数えられるだろう。例外は raw バイトを表す 8 ビット文字。これらはユニバイトバッファーでは 1 バイトで表現されるが、バッファーをマルチバイトにセットした際は 2 バイトのシーケンスに変換されて、その逆の変換も行われる。

この関数はどの表現が使用されているかを記録するために `enable-multibyte-characters` をセットする。これは以前の同じテキストをカバーするように、バッファー内のさまざまなデータ (オーバーレイ、テキストプロパティ、マーカーを含む) を調整する。

ナローイングはマルチバイト文字シーケンス中間で発生するかもしれないので、この関数はバッファーがナローイングされている場合はエラーをシグナルする。

そのバッファーがインダイレクトバッファー (indirect buffer: 間接バッファー) の場合にもエラーをシグナルする。インダイレクトバッファーは常にベースバッファー (base buffer: 基底バッファー) の表現を継承する。

`string-as-unibyte` *string* [Function]

*string*がすでにユニバイト文字列なら、この関数は *string*自身をリターンする。それ以外は *string*と同じバイトだが、それぞれの文字を個別の文字としてとして扱って新たな文字列をリターンする (値は *string*より多くの文字をもつかもしれない)。例外として raw バイトを表す 8 ビット文字は、それぞれ単一のバイトに変換される。新たに作成された文字列にテキストプロパティは含まれない。

`string-as-multibyte` *string* [Function]

*string*がすでにマルチバイト文字列なら、この関数は *string*自身をリターンする。それ以外は *string*と同じバイトだが、それぞれのマルチバイトシーケンスを 1 つの文字としてとして扱って新たな文字列をリターンする。これは値が *string*より少ない文字をもつかもしれないことを意味する。*string*内のバイトシーケンスが単一文字のマルチバイト表現として無効なら、そのシーケンスないの各バイトは 8 ビット raw バイトとして扱われる。新たに作成された文字列にはテキストプロパティは含まれない

33.5 文字コード

ユニバイトやマルチバイトによるテキスト表現は異なる文字コードを使用します。ユニバイト表現にたいして有効な文字コードの範囲は 0 から`#xFF`(255) でこれは 1 バイト範囲に収まる値です。マルチバイト表現にたいして有効な文字コードの範囲は 0 から`#x3FFFFF`です。このコード空間では値 0 から`#x7F`(127) が ASCII文字用、値`#x80`(128) から`#x3FFF7F`(4194175) が非 ASCII文字用になります。

Emacs の文字コードは、Unicode 標準のスーパーセット (superset: 上位集合) です。値 0 から`#x10FFFF`(1114111) は同じコードポイントの Unicode 文字に対応します。値`#x110000`(1114112) から`#x3FFF7F`(4194175) は Unicode に統一されていない文字、値`#x3FFF80`(4194176) から`#x3FFFFF`(4194303) は 8 ビット raw バイトを表します。

`characterp` *charcode* [Function]

これは *charcode*が有効な文字なら `t`、それ以外は `nil`をリターンする。

```
(characterp 65)
     ⇒ t
(characterp 4194303)
     ⇒ t
(characterp 4194304)
     ⇒ nil
```

`max-char` [Function]

この関数は有効な文字コードポイントが保有し得る最大の値をリターンする。

```
(characterp (max-char))
     ⇒ t
```

```
(characterp (1+ (max-char)))
     ⇒ nil
```

`char-from-name` *string* **&optional** *ignore-case* [Function]

この関数は Unicode 名が *string*であるような文字をリターンする。*ignore-case*が非 `nil`なら *string*の case(大文字小文字) は無視する。*string*が文字の名前でなければ、この関数は `nil`をリターンする。

```
;; U+03A3
(= (char-from-name "GREEK CAPITAL LETTER SIGMA") #x03A3)
     ⇒ t
```

`get-byte` **&optional** *pos string* [Function]

この関数はカレントバッファー内の文字位置 *pos*にあるバイトをリターンする。カレントバッファーがユニバイトなら、その位置のバイトをそのままリターンする。バッファーがマルチバイトなら、8 ビット raw バイトは 8 ビットコードに変換される一方で、ASCII文字のバ値は文字コードポイントと同じになる。この関数は *pos*にある文字が非 ASCIIならエラーをシグナルする。

オプション引数 *string*はカレントバッファーのかわりに文字列からバイト値を得ることを意味する。

33.6 文字のプロパティ

文字プロパティ*(character property* とは、その文字の振る舞いとテキストが処理や表示される間にどのように処理されるべきかを指定する名前つきの文字属性です。したがって文字プロパティはその文字の意味を指定するための重要な一部です。

全体として Emacs は自身の文字プロパティ実装において Unicode 標準にしたがいます。特に Emacs は Unicode Character Property Model (`https://www.unicode.org/reports/tr23/`) をサポートしており、Emacs 文字プロパティデータベースは Unicode 文字データベース (UCD: Unicode Character Database) から派生したものです。Unicode 文字プロパティとその意味についての詳細な説明は Character Properties chapter of the Unicode Standard (`https://www.unicode.org/versions/Unicode12.1.0/ch04.pdf`) を参照してください。このセクションではあなたがすでに Unicode 標準の該当する章に親しんでいて、その知識を Emacs Lisp プログラムに適用したいものと仮定します。

Emacs では各プロパティは名前をもつシンボルであり、そのシンボルは利用可能な値セットをもち、値の型はプロパティに依存します。ある文字が特定のプロパティをもたなければ、その値は `nil`になります。一般的なルールとして Emacs での文字プロパティ名は対応する Unicode プロパティ名を小文字にして、文字 '`_`'をダッシュ文字 '`-`'で置き換えることにより生成されます。たとえば`Canonical_Combining_Class`は `canonical-combining-class`となります。しかし簡単に使用できるように名前を短くすることもあります。

UCDによりいくつかのコードポイントは未割り当て *(unassigned)* のまま残されており、それらに対応する文字はありません。Unicode 標準は、そのようなコードポイントのプロパティにたいしてデフォルト値を定義しています。それらについては以下の各プロパティごとに注記することにします。

以下は Emacs が関知するすべての文字プロパティにたいする値タイプの完全なリストです:

`name` Unicode プロパティ`Name`に対応する。値はラテン大文字の A から Z、数字、スペース、ハイフン '`-`'の文字から構成される文字列。未割り当てのコードポイントにたいする値は `nil`。

`general-category`
: Unicode プロパティ`General_Category`に対応する。値はその文字の分類をアルファベット 2 文字に略したものを名前としてもつようなシンボル。未割り当てのコードポイントにたいする値は `Cn`。

`canonical-combining-class`
: Unicode プロパティ`Canonical_Combining_Class`に対応する。値は整数。未割り当てのコードポイントにたいする値は 0。

`bidi-class`
: Unicode プロパティ`Bidi_Class`に対応する。値はその文字の Unicode 方向タイプ *(directional type)* が名前であるようなシンボル。Emacs は表示のために双方向テキストを並び替える際にこのプロパティを使用する (Section 39.26 [Bidirectional Display], page 1085 を参照)。未割り当てのコードポイントにたいする値はそのコードポイントが属するコードブロックに依存する。未割り当てのコードポイントのほとんどは `L`(強い左方向) だが、`AL` (Arabic letter: アラビア文字) や `R` (強い右方向) を受け取るコースポイントもいくつかある。

`decomposition`
: Unicode プロパティの `Decomposition_Type`と `Decomposition_Value`に対応する。値は最初の要素が`small`のような互換性のあるフォーマットタグ (compatibility formatting tag) であるかもしれないリスト [2]。分割シーケンス (compatibility decomposition sequence) をもたない文字、および未割り当てのコードポイントにたいする値はその文字自身が唯一のメンバーであるようなリスト。

`decimal-digit-value`
: `Numeric_Type`が‘`Decimal`’であるような文字 Unicode プロパティ`Numeric_Value`に対応する。値は整数、その文字が 10 進値をもたなければ `nil`。未割り当てのコードポイントにたいする値は、NaNまたは “not a number(数字ではない)” を意味する `nil`。

`digit-value`
: `Numeric_Type`が‘`Digit`’であるような文字の Unicode プロパティ`Numeric_Value`に対応する。値は整数。このような文字には互換性のある添字や上付き数字が含まれ、値は対応する数字。何も数値をもたない文字および未割り当てのコードポイントにたいする値は NaNを意味する `nil`。

`numeric-value`
: `Numeric_Type`が‘`Numeric`’であるような文字の Unicode プロパティ`Numeric_Value`に対応する。このプロパティの値は数字。このプロパティをもつ文字の例には分数、添字、上付き数字、ローマ数字、通貨分数 (訳注: 原文は “currency numerators” でベンガル語の分数値用の歴史的な記号を指すと思われる)、丸数字が含まれる。たとえば文字 U+2155 (VULGAR FRACTION ONE FIFTH: (訳注) スラッシュで分子と分母を区切った表記による 5 分の 1 のこと) にたいするこのプロパティの値は `0.2`。数値をもたない文字と未割り当てのコードポイントにたいする値は NaNを意味する `nil`。

`mirrored`
: Unicode プロパティ`Bidi_Mirrored`に対応する。このプロパティの値は `Y`か `N`いずれかのシンボル。未割り当てのコードポイントにたいする値は `N`。

[2] Unicode 仕様ではこれらのタグ名を ‘`<..>`’カッコ内に記述しますが Emacs でのタグ名にはカッコは含まれません。Unicode での ‘`<small>`’指定は Emacs では ‘`small`’となります。

`mirroring`
: Unicode プロパティ`Bidi_Mirroring_Glyph`に対応する。このプロパティの値は、そのグリフ (glyph) がその文字のグリフの鏡像 (mirror image) を表すような文字、定義済みの鏡像グリフがなければ`nil`。`mirrored`プロパティが`N`であるようなすべての文字の`mirroring`プロパティは`nil`。しかし`mirrored`プロパティが`Y`の文字でも、鏡像をもつ適切な文字がないという理由により`mirroring`が`nil`の文字もある。Emacs は適切な際は鏡像を表示するためにこのプロパティを使用する (Section 39.26 [Bidirectional Display], page 1085 を参照)。未割り当てのコードポイントにたいする値は`nil`。

`paired-bracket`
: Unicode プロパティ`Bidi_Paired_Bracket`に対応する。このプロパティの値は文字の *paired bracket(カッコのペア)* のコードポイント、その文字が bracket 文字でなければ`nil`。これは Unicode 双方向アルゴリズム (Unicode Bidirectional Algorithm) によりカッコのペアとして扱われる文字間のマッピングを確立する。Emacs は丸カッコ (parentheses) や角カッコ (braces)、およびその類の文字を再配置する方法を決定する際にこのプロパティを使用する (Section 39.26 [Bidirectional Display], page 1085 を参照)。

`bracket-type`
: Unicode の`Bidi_Paired_Bracket_Type`プロパティに対応する。`paired-bracket`プロパティが非`nil`の文字にたいするこのプロパティは`o` (開カッコ文字) か`c` (閉カッコ文字) を表すシンボルのいずれか。`paired-bracket`プロパティが`nil`の文字にたいする値は`n` (None: なし)。`paired-bracket`と同じようにこのプロパティは双方向ディスプレーにより使用される。

`old-name`
: Unicode プロパティ`Unicode_1_Name`に対応する。値は文字列。未割り当てのコードポイント、およびこのプロパティにたいする値をもたない文字では値は`nil`。

`iso-10646-comment`
: Unicode プロパティ`ISO_Comment`に対応する。値は文字列か`nil`。未割り当てのコードポイントにたいする値は`nil`。

`uppercase`
: Unicode プロパティ`Simple_Uppercase_Mapping`に対応する。このプロパティの値は単一の文字。未割り当てのコードポイントの値は`nil`であり、これはその文字自身を意味する。

`lowercase`
: Unicode プロパティ`Simple_Lowercase_Mapping`に対応する。このプロパティの値は単一の文字。未割り当てのコードポイントの値は`nil`であり、これはその文字自身を意味する。

`titlecase`
: Unicode プロパティ`Simple_Titlecase_Mapping`に対応する。タイトルケース *(title case)* とは単語の最初の文字を大文字にする必要がある際に使用される文字の特別な形式のこと。このプロパティの値は単一の文字。未割り当てのコードポイントにたいする値は`nil`であり、これはその文字自身を意味する。

`special-uppercase`
: Unicode の言語やコンテキストに依存しない特別な大文字 case ルールに対応する。このプロパティの値は文字列 (空も可)。たとえば U+00DF LATIN SMALL LETTER SHARP

S にたいするマッピングは"SS"。特別なマッピングのない文字にたいする値は nil (かわりに uppercaseプロパティの照会が必要なことを意味する)。

special-lowercase
: Unicode の言語やコンテキストに依存しない特別な小文字 case ルールに対応する。このプロパティの値は文字列 (空も可)。たとえば U+0130 LATIN CAPITAL LETTER I WITH DOT ABOVE にたいするマッピングは"SS"。特別なマッピングのない文字にたいする値は nil (かわりに lowercaseプロパティの照会が必要なことを意味する)。

special-titlecase
: Unicode の無条件の特別なタイトル case ルールに対応する。このプロパティの値は文字列 (空も可)。たとえば U+FB01 LATIN SMALL LIGATURE FI にたいするマッピングは"Fi"。特別なマッピングのない文字にたいする値は nil (かわりに titlecaseプロパティの照会が必要なことを意味する)。

get-char-code-property *char propname* [Function]

この関数は *char*のプロパティ *propname*の値をリターンする。

```
(get-char-code-property ?\s 'general-category)
     ⇒ Zs
(get-char-code-property ?1 'general-category)
     ⇒ Nd
;; U+2084
(get-char-code-property ?\N{SUBSCRIPT FOUR}
                        'digit-value)
     ⇒ 4
;; U+2155
(get-char-code-property ?\N{VULGAR FRACTION ONE FIFTH}
                        'numeric-value)
     ⇒ 0.2
;; U+2163
(get-char-code-property ?\N{ROMAN NUMERAL FOUR}
                        'numeric-value)
     ⇒ 4
(get-char-code-property ?\( 'paired-bracket)
     ⇒ 41  ;; closing parenthesis
(get-char-code-property ?\) 'bracket-type)
     ⇒ c
```

char-code-property-description *prop value* [Function]

この関数はプロパティ *prop*の *value*の説明文字列 (description string)、*value*が説明をもたなければ nilをリターンする。

```
(char-code-property-description 'general-category 'Zs)
     ⇒ "Separator, Space"
(char-code-property-description 'general-category 'Nd)
     ⇒ "Number, Decimal Digit"
(char-code-property-description 'numeric-value '1/5)
     ⇒ nil
```

`put-char-code-property` *char propname value* [Function]
この関数は文字 *char*のプロパティ *propname*の値として *value*を格納する。

`unicode-category-table` [Variable]
この変数の値は、それぞれの文字にたいしてその Unicode プロパティ`General_Category`をシンボルとして指定する文字テーブル (Section 6.6 [Char-Tables], page 110 を参照)。

`char-script-table` [Variable]
この変数の値は、それぞれの文字がシンボルを指定するような文字テーブル。シンボルの名前は Unicode コードスペースからスクリプト固有ブロックへの Unicode 標準分類にしたがうような、その文字が属するスクリプト。この文字テーブルは余分のスロットを 1 つもち、値はすべてのスクリプトシンボルのリスト。

`char-width-table` [Variable]
この変数の値は、それぞれの文字がスクリーン上で占めるであろう幅を列単位で指定する文字テーブル。

`printable-chars` [Variable]
この変数の値は、それぞれの文字にたいしてそれがプリント可能かどうかを指定する文字テーブル。すなわち `(aref printable-chars char)`を評価した結果が `t`ならプリント可、`nil`なら不可。

33.7 文字セット

Emacs の文字セット *(character set*、もしくは *charset)* とは、それぞれの文字が数字のコードポイントに割り当てられれた文字セットのことです (Unicode 標準ではこれを符号化文字集合 *(coded character set)* と呼ぶ)。Emacs の各文字セットはシンボルであるような名前をもちます。1 つの文字が任意の数の異なる文字セットに属することができますが、各文字セット内で異なるコードポイントをもつのが一般的でしょう。文字セットの例には `ascii`、`iso-8859-1`、`greek-iso8859-7`、`windows-1255`が含まれます。文字セット内で文字に割り当てられるコードポイントは、Emacs 内のバッファーや文字列内で使用されるコードポイントとは通常は異なります。

Emacs は特別な文字セットをいくつか定義しています。文字セット `unicode`は Emacs コードポイントが `0..#x10FFFF`の範囲のすべての文字セットを含みます。文字セット `emacs`はすべての ASCII、および非 ASCII文字を含みます。最後に `eight-bit`文字セットは 8 ビット raw バイトを含みます。テキスト内で raw バイトを見つけたときに Emacs はこれを使用します。

`charsetp` *object* [Function]
*object*は文字セットを命名するシンボルなら `t`、それ以外は `nil`をリターンする。

`charset-list` [Variable]
値はすべての定義済み文字セットの名前のリスト。

`charset-priority-list` **&optional** *highestp* [Function]
この関数はすべての定義済み文字セットの優先順にソートされたリストをリターンする。*highestp*が非 `nil`なら、この関数はもっとも優先度の高い文字セット 1 つをリターンする。

`set-charset-priority` **&rest** *charsets* [Function]
この関数は *charsets*をもっとも高い優先度の文字セットにする。

`char-charset` *character* **&optional** *restriction* [Function]
この関数は *character*が属する文字セットで、もっとも優先度の高い文字セットの名前をリターンする。ただし ASCII文字は例外であり、この関数は常に `ascii`をリターンする。

*restriction*が非 `nil`なら、それは検索する文字セットのリストであること。かわりにコーディングシステムも指定でき、その場合にはそのコーディングシステムによりサポートされている必要がある (Section 33.10 [Coding Systems], page 864 を参照)。

`charset-plist` *charset* [Function]
この関数は文字セット *charset*のプロパティをリターンする。たとえ *charset*がシンボルだったとしても、これはそのシンボルのプロパティリストと同じではない。文字セットプロパティにはドキュメント文字列、短い名前等、その文字セットに関する重要な情報が含まれる。

`put-charset-property` *charset propname value* [Function]
この関数は *charset*のプロパティ *propname*に与えられた *value*をセットする。

`get-charset-property` *charset propname* [Function]
この関数は *charset*のプロパティ *propname*の値をリターンする。

`list-charset-chars` *charset* [Command]
このコマンドは文字セット *charset*内の文字のリストを表示する。

Emacs は文字の内部的な表現と、その文字の特定の文字セット内でのコードポイントを相互に変換することができます。以下はこれらをサポートするための関数です。

`decode-char` *charset code-point* [Function]
この関数は *charset*内で *code-point*に割り当てられた文字を Emacs の対応する文字にデコードしてリターンする。そのコードポイントの文字が *charset*に含まれなければ値は `nil`。

後方互換性のために *code-point*が Lisp の fixnum (Section 3.1 [Integer Basics], page 37 を参照) に収まらなければ、コンスセル `(high . low)`として指定できる。ここで *low*は値の下位 16 ビット、*high*は高位 16 ビット。この使用方法は時代遅れである。

`encode-char` *char charset* [Function]
この関数は *charset*内で文字 *char*に割り当てられたコードポイントをリターンする。*charset*が *char*にたいするコードポイントをもたなければ値は `nil`。

以下の関数は文字セット内の文字の一部、またはすべてにたいして特定の関数を適用するのに有用です。

`map-charset-chars` *function charset* **&optional** *arg from-code to-code* [Function]
*charset*内の文字にたいして *function*を呼び出す。*function*は 2 つの引数で呼び出される。1 つ目はコンスセル `(from . to)`であり、*from*と *to*は文字セット内に含まれる文字の範囲。*arg*は 2 つ目の引数として *function*に渡される。

デフォルトでは *function*に渡されるコードポイントの範囲には *charset*内のすべての文字が含まれるが、オプション引数 *from-code*と *to-code*により、それは *charset*の 2 つのコードポイント間にある文字範囲に制限される。*from-code*か *to-code*のいずれかが `nil`の場合のデフォルトは、*charset*のコードポイントの最初か最後。

33.8 文字セットのスキャン

特定の文字がどの文字セットに属するか調べられると便利なときがあります。これの用途の 1 つは、どのコーディングシステム (Section 33.10 [Coding Systems], page 864 を参照) が問題となっているテキストすべてを表現可能か判断することです。他にもそのテキストを表示するフォントの判断があります。

`charset-after` **&optional** *pos* [Function]
この関数は、カレントバッファー内の位置 *pos*にある文字を含む、もっとも高い優先度の文字セットをリターンする。*pos*が省略または `nil`の場合のデフォルトはポイントのカレント値。*pos*が範囲外なら値は `nil`。

`find-charset-region` *beg end* **&optional** *translation* [Function]
この関数はカレントバッファー内の位置 *beg*から *end*の間の文字を含む、もっとも優先度の高い文字セットのリストをリターンする。

オプション引数 *translation*はテキストのスキャンに使用するための変換テーブルを指定する (Section 33.9 [Translation of Characters], page 863 を参照)。これが非 `nil`ならリージョン内の各文字はそのテーブルを通じて変換され、リターンされる値にはバッファーの実際の文字ではなく変換された文字が記述される。

`find-charset-string` *string* **&optional** *translation* [Function]
この関数は *string*内の文字を含む、もっとも優先度の高い文字セットのリストをリターンする。これは `find-charset-region`と似ているが、カレントバッファーの一部ではなく *string*のコンテンツに適用される点が異なる。

33.9 文字の変換

変換テーブル *(translation table)* とは文字から文字へのマッピングを指定する文字テーブルです (Section 6.6 [Char-Tables], page 110 を参照)。これらのテーブルはエンコーディング、デコーディング、および他の用途にも使用されます。独自に変換テーブルを指定するコーディングシステムもいくつかあります。他のすべてのコーディングシステムに適用されるデフォルトの変換テーブルも存在します。

変換テーブルには余分のスロット (extra slots) が 2 つあります。1 つ目のスロットは `nil`、または逆の変換を処理する変換テーブルです。2 つ目のスロットは変換する文字シーケンスを照合する際の最大文字数です (以下の `make-translation-table-from-alist`の説明を参照)。

`make-translation-table` **&rest** *translations* [Function]
この関数は引数 *translations*にもとづいて変換テーブルをリターンする。*translations*の各要素は `(`*`from . to`*`)`という形式のリストであること。これは *from*から *to*への文字の変換を指示する。

各引数内の引数とフォームは順に処理され、もし前のフォームですでに *to*がたとえば *to-alt*に変換されていれば *from*も *to-alt*に変換される。

デコードを行う間、その変換テーブルの変換は通常のデコーディングの結果の文字に適用されます。あるコーディングシステムがプロパティ `:decode-translation-table`をもつなら、それは使用する変換テーブル、または順に適用するべき変換テーブルのリストを指定します (これはコーディングシステムの名前であるようなシンボルのプロパティではなく、`coding-system-get`がリターンするようなコーディングシステムのプロパティ。Section 33.10.1 [Basic Concepts of Coding Systems],

page 865 を参照)。最後にもし standard-translation-table-for-decodeが非 nilなら、結果となる文字はそのテーブルにより変換されます。

エンコードを行う間は、その変換テーブルの変換はバッファー内の文字に適用されて、変換結果は実際にエンコードされます。あるコーディングシステムがプロパティ:encode-translation-tableをもつならそれは使用する変換テーブル、または順に適用するべき変換テーブルのリストを指定します。加えてもし変数 standard-translation-table-for-encodeが非 nilなら、それは変換結果にたいして使用するべき変換テーブルを指定します。

standard-translation-table-for-decode [Variable]
これはデコード用のデフォルトの変換テーブル。あるコーディングシステムが独自に変換テーブルを指定する場合には、この変数の値が非 nilなら、それら独自のテーブルを適用後にこの変数の変換テーブルが適用される。

standard-translation-table-for-encode [Variable]
これはエンコード用のデフォルトの変換テーブル。あるコーディングシステムが独自に変換テーブルを指定する場合には、この変数の値が非 nilならそれら独自のテーブル適用後にこの変数の変換テーブルが適用される。

translation-table-for-input [Variable]
自己挿入文字は挿入前にこの変換テーブルを通じて変換が行われる。検索コマンドもバッファー内の内容とより信頼性のある比較ができるようにこのテーブルを通じて入力を変換する。

この変数はセット時に自動的にバッファーローカルになる。

make-translation-table-from-vector *vec* [Function]
この関数はバイト (値は 0 から#xFF) から文字にマップする 256 要素の配列であるような、*vec*から作成した変換テーブルをリターンする。未変換のバイトにたいする要素は nilかもしれない。リターンされるテーブルは余分な 1 つ目のスロットにそのマッピングを保持する変換テーブル、2 つ目の余分なスロットに値 1をもつ。

この関数は各バイトを特定の文字にマップするようなプライベートなコーディングシステムを簡単に作成する手段を提供する。define-coding-systemの *props*引数のプロパティ:decode-translation-tableと:encode-translation-tableに、リターンされるテーブルと逆変換テーブルを指定できる。

make-translation-table-from-alist *alist* [Function]
この関数は make-translation-tableと似ているが、シンプルな 1 対 1 のマッピングを行う変換テーブルではなく、より複雑な変換テーブルをリターンする。*alist*の各要素は (*from* . *to*)という形式をもち、ここで *from*および *to*は文字または文字シーケンスを指定するベクター。*from*が文字なら、その文字は *to*(文字か文字シーケンス) に変換される。*from*が文字のベクターならそのシーケンスは *to*に変換される。リターンされるテーブルは 1 つ目の余分なスロットに逆のマッピングを行う変換テーブル、2 つ目の余分なスロットには文字シーケンス *from*すべての最大長をもつ。

33.10 コーディングシステム

Emacs がファイルにたいして読み書きをしたりサブプロセスとテキストの送受信を行う際には、通常は特定のコーディングシステム *(coding system)* の指定にしたがって文字コード変換や行末変換を行います。

コーディングシステムの定義は難解な問題であり、ここには記述しません。

33.10.1 コーディングシステムの基本概念

文字コード変換 *(character code conversion)* により、Emacs 内部で使用される文字の内部表現と他のエンコーディングの間で変換が行われます。Emacs は多くの異なるエンコーディングをサポートしており、それらは双方向に変換が可能です。たとえば Latin 1、Latin 2、Latin 3、Latin 4、Latin 5、およびいくつかの ISO 2022 の変種等のようなエンコーディングにたいしてテキストを双方向に変換できます。あるケースにおいては同じ文字にたいして Emacs は複数のエンコーディング候補をサポートします。たとえばキリル (ロシア語) のアルファベットにたいしては ISO、Alternativnyj、KOI8 のように 3 つにコーディングシステムが存在します。

コーディングシステムはそれぞれ特定の文字コード変換セットを指定しますが、`undecided`というコーディングシステムは特別です。これはそれぞれのファイルにたいして、そのファイルのデータにもとづいて発見的に選択が行われるように選択を未指定のままにします。コーディングシステム `prefer-utf-8`は `undecided`と似ていますが、可能なら `utf-8`を優先的に選択します。

一般的にコーディングシステムは可逆的な同一性を保証しません。あるコーディングシステムを使用してバイトシーケンスをデコードしてから、同じコーディングシステムで結果テキストをエンコードしても、異なるバイトシーケンスが生成される可能性があります。しかしデコードされたオリジナルのバイトシーケンスとなることを保証するコーディングシステムもいくつかあります。以下にいくつかの例を挙げます:

iso-8859-1、utf-8、big5、shift_jis、euc-jp

バッファーテキストのエンコードと結果のデコードでもオリジナルテキストの再生成に失敗する可能性があります。たとえばその文字をサポートしないコーディングシステムで文字をエンコードした場合の結果は予測できず、したがって同じコーディングシステムを使用してそれをデコードしても異なるテキストが生成されるでしょう。現在のところ Emacs は未サポート文字のエンコーディングによる結果をエラーとして報告できません。

行末変換 *(end of line conversion:* 改行変換*)* はファイル内の行末を表すために、さまざまなシステム上で使用される 3 つの異なる慣例を扱います。GNU や Unix システムで使用される Unix の慣例では LF 文字 (linefeed 文字、改行とも呼ばれる) が使用されます。MS-Windows や MS-DOS システムで使用される DOS の慣例では行末に CR 文字 (carriage-return 文字、復帰文字とも呼ばれる) と LF 文字が使用されます。Mac の慣例では CR 文字だけが使用されます (これはクラシックな Mac OS で使用されていた慣例)。

`latin-1`のようなベースコーディングシステム *(base coding systems:* 基本コーディングシステム*)* では、データにもとづいて選択されるように行末変換は未指定となっています。`latin-1-unix`、`latin-1-dos`、`latin-1-mac`のようなバリアントコーディングシステム *(variant coding systems:* 変種コーディングシステム*)* では行末変換を明示的に指定します。ほとんどのベースコーディングシステムは '`-unix`'、'`-dos`'、'`-mac`'を追加した 3 つの対応する形式の変種をもちます。

`raw-text`は文字コード変換を抑制して、このコーディングシステムで visit されたバッファーがユニバイトバッファーとなる点において特殊なコーディングシステムです。歴史的な理由によりこのコーディングシステムによりユニバイトとマルチバイト両方のテキストを保存できます。マルチバイトテキストのエンコードに `raw-text`を使用した際には 1 文字コード変換を行います。8 ビット文字は 1 バイトの外部表現に変換されます。`raw-text`は通常のようにデータにより判断できるように行末変換を指定せず、通常のように行末変換を指定する 3 つの変種をもちます。

`no-conversion` (とエイリアスの `binary`) は `raw-text-unix`と等価です。これは文字コードおよび行末にたいする変換をいずれも指定しません。

`utf-8-emacs`はデータが Emacs の内部エンコーディング (Section 33.1 [Text Representations], page 852 を参照) で表されることを指定するコーディングシステムです。コード変換が何も発生しない

点で raw-textと似ていますが、結果がマルチバイトデータである点が異なります。emacs-internal という名前は utf-8-emacs-unixにたいするエイリアスです (そのため 3 種類すべての行末変換をデコードする utf-8-emacsと異なり行末変換を強制しない)。

coding-system-get *coding-system property* [Function]
この関数はコーディングシステム *coding-system*の指定されたプロパティをリターンする。コーディングシステムのプロパティのほとんどは内部的な目的のために存在するが、:mime-charset については有用と思うかもしれない。このプロパティの値はそのコーディングシステムが読み書きできる文字コードにたいして MIME 内で使用される名前。以下は例:

```
(coding-system-get 'iso-latin-1 :mime-charset)
     ⇒ iso-8859-1
(coding-system-get 'iso-2022-cn :mime-charset)
     ⇒ iso-2022-cn
(coding-system-get 'cyrillic-koi8 :mime-charset)
     ⇒ koi8-r
```

:mime-charsetプロパティの値はそのコーディングシステムにたいするエイリアスとしても定義されている。

coding-system-aliases *coding-system* [Function]
この関数は *coding-system*のエイリアスのリストをリターンする。

33.10.2 エンコーディングと I/O

コーディングシステムの主な目的はファイルの読み込みと書き込みへの使用です。関数 insert-file-contentsはファイルデータのデコードにコーディングシステムを使用して、write-regionはバッファーコンテンツのエンコードにコーディングシステムを使用します。

使用するコーディングシステムは明示的 (Section 33.10.6 [Specifying Coding Systems], page 874 を参照)、またはデフォルトメカニズム (Section 33.10.5 [Default Coding Systems], page 871 を参照) を使用により暗黙的に指定できます。しかしこれらの手法は何を行うかを完全には指定しないかもしれません。たとえば、これらはデータから文字コード変換を行わない undecided のようなコーディングシステムを選択するかもしれません。このような場合、I/O 処理はコーディングシステム選択により処理を完了します。後でどのコーディングシステムが選択されたか調べたいことが頻繁にあるでしょう。

buffer-file-coding-system [Variable]
このバッファーローカル変数はバッファーの保存、および write-regionによるバッファー部分のファイルへの書き出しに使用されるコーディングシステムを記録する。書き込まれるテキストがこの変数で指定されたコーディングシステムを使用して安全にエンコードできない場合には、これらの操作は関数 select-safe-coding-systemを呼び出すことにより代替となるエンコーディングを選択する (Section 33.10.4 [User-Chosen Coding Systems], page 870 を参照)。異なるエンコーディングの選択がユーザーによるコーディングシステムの指定を要するなら、buffer-file-coding-systemは新たに選択されたコーディングシステムに更新される。

buffer-file-coding-systemはサブプロセスへのテキスト送信に影響しない。

save-buffer-coding-system [Variable]
この変数は、(buffer-file-coding-systemをオーバーライドして) バッファーを保存するためのコーディングシステムを指定する。これは write-regionには使用されないことに注意。

あるコマンドがバッファーを保存するために `buffer-file-coding-system` (または `save-buffer-coding-system`) の使用を開始して、そのコーディングシステムがバッファー内の実際のテキストを処理できなければ、(`select-safe-coding-system`を呼び出すことにより) そのコマンドは他のコーディングシステムの選択をユーザーに求める。これが発生した後はコマンドはユーザー指定のコーディングシステムを表すために `buffer-file-coding-system`の更新も行う。

`last-coding-system-used` [Variable]

ファイルやサブプロセスにたいする I/O 操作は、使用したコーディングシステムの名前をこの変数にセットする。明示的にエンコードとデコードを行う関数 (Section 33.10.7 [Explicit Encoding], page 876 を参照) もこの変数をセットする。

警告: サブプロセス出力の受信によりこの変数がセットされるため、この変数は Emacs が wait している際は常に変更され得る。したがって興味対象となる値を格納する関数呼び出し後は、間を空けずにその値をコピーすること。

変数 `selection-coding-system`はウィンドウシステムにたいして選択 (selection) をエンコードする方法を指定します。Section 29.20 [Window System Selections], page 747 を参照してください。

`file-name-coding-system` [Variable]

変数 `file-name-coding-system`はファイル名のエンコーディングに使用するコーディングシステムを指定する。Emacs は、すべてのファイル操作にたいして、ファイル名のエンコードにそのコーディングシステムを使用する。`file-name-coding-system`が `nil`なら Emacs は選択された言語環境 (language environment) により決定されたデフォルトのコーディングシステムを使用する。デフォルト言語環境ではファイル名に含まれるすべての非 ASCII文字は特別にエンコードされない。これらは Emacs の内部表現を使用してファイルシステム内で表される。

警告: Emacs のセッション中に `file-name-coding-system` (または言語環境) を変更した場合には、以前のコーディングシステムを使用してエンコードされた名前をもつファイルを visit していると、新たなコーディングシステムでは異なるように扱われるので問題が発生し得る。これらの visit されたファイル名でこれらのバッファーの保存を試みると、保存で間違ったファイル名が使用されたりエラーとなるかもしれない。そのような問題が発生したら、そのバッファーにたいして新たなファイル名を指定するために *C*-x *C*-w`を使用すること。

Windows 2000 以降では Emacs は OS に渡すファイル名にデフォルトで Unicode API を使用するため、`file-name-coding-system`の値は大部分が無視される。Lisp レベルでファイル名のエンコードやデコードを必要とする Lisp アプリケーションは、`system-type`が `windows-nt`のときは `utf-8`をコーディングシステムに使用すること。UTF-8 でエンコードされたファイル名から、OS と対話するために適したエンコーディングへの変換は Emacs により内部的に処理される。

33.10.3 Lisp でのコーディングシステム

以下はコーディングシステムと連携する Lisp 機能です:

`coding-system-list` **&optional** *base-only* [Function]

この関数はすべてのコーディングシステムの名前 (シンボル) をリターンする。*base-only*が非 `nil`なら、値にはベースコーディングシステムだけが含まれる。それ以外ならエイリアス、およびバリアントコーディングシステムも同様に含まれる。

`coding-system-p` *object* [Function]
この関数は *object*がコーディングシステムの名前なら `t`、または `nil`をリターンする。

`check-coding-system` *coding-system* [Function]
この関数は *coding-system*の有効性をチェックする。有効なら *coding-system*をリターンする。*coding-system*が `nil`なら、この関数は `nil`をリターンする。それ以外の値にたいしては `error-symbol`が `coding-system-error`であるようなエラーをシグナルする (Section 11.7.3.1 [Signaling Errors], page 165 を参照)。

`coding-system-eol-type` *coding-system* [Function]
この関数は行末 (*eol* とも言う) を *coding-system*で使用されるタイプに変換する。*coding-system*が特定の eol 変換を指定する場合にはリターン値は 0、1、2 のいずれかであり、それらは順に `unix`、`dos`、`mac`を意味する。*coding-system*が明示的に eol 変換を指定しなければ、リターン値は以下のようにそれぞれが可能な eol 変換タイプをもつようなコーディングシステムのベクター:

```
(coding-system-eol-type 'latin-1)
     ⇒ [latin-1-unix latin-1-dos latin-1-mac]
```

この関数がベクターをリターンしたら、Emacs はテキストのエンコードやデコードプロセスの一部として使用する eol 変換を決定するだろう。デコードではテキストの行末フォーマットは自動検知され、eol 変換はそれに適合するようセットされる (DOS スタイルの CRLF フォーマットは暗黙で eol 変換に `dos`をセットする)。エンコードにたいしては適切なデフォルトコーディングシステム (`buffer-file-coding-system`にたいする `buffer-file-coding-system`のデフォルト値)、または背景にあるプラットフォームにたいして適切なデフォルト eol 変換が採用される。

`coding-system-change-eol-conversion` *coding-system eol-type* [Function]
この関数は *coding-system*と似ているが `eol-type`で指定された eol 変換の異なるコーディングシステムをリターンする。*eol-type*は `unix`、`dos`、`mac`、または `nil`であること。これが `nil`ならリターンされるコーディングシステムは、データの eol 変換により決定される。

*eol-type*は `unix`、`dos`、`mac`を意味する 0、1、2 でもよい。

`coding-system-change-text-conversion` *eol-coding text-coding* [Function]
この関数は *eol-coding*の行末変換と、*text-coding*のテキスト変換を使用するコーディングシステムをリターンする。*text-coding*が `nil`ならこれは `undecided`、または *eol-coding*に対応するバリアントの 1 つをリターンする。

`find-coding-systems-region` *from to* [Function]
この関数は *from*と *to*の間のテキストのエンコードに使用可能なコーディングシステムのリストをリターンする。このリスト内のすべてのリストは、そのテキスト範囲内にあるすべてのマルチバイト文字を安全にエンコードできる。

そのテキストがマルチバイト文字を含まれなければ、この関数はリスト `(undecided)`をリターンする。

`find-coding-systems-string` *string* [Function]
この関数は *string*のテキストのエンコードに使用可能な、コーディングシステムのリストをリターンする。このリスト内のすべてのリストは *string*にあるすべてのマルチバイト文字を安全にエンコードできる。そのテキストがマルチバイト文字を含まれなければ、この関数はリスト `(undecided)`をリターンする。

`find-coding-systems-for-charsets` *charsets* [Function]
この関数はリスト *charsets*内のすべての文字セットのエンコードに使用可能なコーディングシステムのリストをリターンする。

`check-coding-systems-region` *start end coding-system-list* [Function]
この関数はリスト `coding-system-list`内のコーディングシステムが *start*と *end*の間のリージョン内にあるすべての文字をエンコード可能かどうかをチェックする。このリスト内のすべてのコーディングシステムが指定されたテキストをエンコード可能なら、この関数は `nil`をリターンする。ある文字をエンコードできないコーディングシステムがある場合には、各要素が `(`*`coding-system1 pos1 pos2 ...`*`)`という形式の alist が値となる。これは *coding-system1* がバッファーの位置 *pos1*、*pos2*、. . . にある文字をエンコードできないことを意味する。

*start*は文字列かもしれず、その場合には *end*は無視されてリターン値はバッファー位置のかわりに文字列のインデックスを参照することになる。

`detect-coding-region` *start end* **&optional** *highest* [Function]
この関数は *start*から *end*のテキストのデコードに適したコーディングシステムを選択する。このテキストはバイトシーケンス、すなわちユニバイトテキスト、ASCIIのみのマルチバイトテキスト、8 ビット文字のシーケンスであること (Section 33.10.7 [Explicit Encoding], page 876 を参照)。

この関数は通常はスキャンしたテキストのデコーディングを処理可能なコーディングシステムのリストをリターンする。これらのコーディングシステムは優先度降順でリストされる。しかし *highest*が非 `nil`なら、リターン値はもっとも高い優先度のコーディングシステムただ 1 つとなる。

リージョンに ISO-2022 の `ESC`のような ISO-2022 制御文字を除いて ASCII文字だけが含まれる場合には値は `undecided`、`(undecided)`、またはテキストから推論可能なら eol 変換を指定するバリアントとなる。

リージョンに null バイトが含まれる場合には、あるコーディングシステムによりエンコードされたテキストがリージョン内に含まれる場合でも値は `no-conversion`となる。

`detect-coding-string` *string* **&optional** *highest* [Function]
この関数は `detect-coding-region`と似ているがバッファー内のバイトのかわりに *string*のコンテンツを処理する点が異なる。

`inhibit-nul-byte-detection` [Variable]
この変数が非 `nil`値をもつなら、リージョンや文字列のエンコーディング検出時に null バイトを無視する。これにより Index ノードをもつ Info ファイルのような null バイトを含むテキストのエンコーディングを正しく検出できる。

`inhibit-iso-escape-detection` [Variable]
この変数が非 `nil`値をもつなら、リージョンや文字列のエンコーディング検出時に ISO-2022 エスケープシーケンスを無視する。結果としてこれまでいくつかの ISO-2022 エンコーディングにおいてエンコード済みと検出されていたテキストがなくなり、バッファー内ですべてのエスケープシーケンスが可視になる。警告**:** この変数の使用には特に注意を払うこと。なぜなら Emacs ディストリビューション内で多くのファイルが ISO-2022 エンコーディングを使用するからである。

`coding-system-charset-list` *coding-system* [Function]

この関数は *coding-system*がサポートする文字セット (Section 33.7 [Character Sets], page 861 を参照) のリストをリターンする。リストすべき文字セットを非常に多くサポートするいくつかのコーディングシステムでは特別な値がリストされる:

- *coding-system*がすべての Emacs 文字をサポートするなら値は `(emacs)`。
- *coding-system*がすべての Unicode 文字をサポートするなら値は `(unicode)`。
- *coding-system*がすべての ISO-2022 文字をサポートするなら値は `iso-2022`。
- *coding-system*が Emacs バージョン 21(Unicode サポートの内部的な実装以前) で使用される内部的コーディングシステム内のすべての文字をサポートするなら値は `emacs-mule`。

サブプロセスへの入出力に使用されるコーディングシステムのチェックやセットの方法については [Process Information], page 949、特に関数 `process-coding-system`や `set-process-coding-system`の説明を参照してください。

33.10.4 ユーザーが選択したコーディングシステム

`select-safe-coding-system` *from to* **&optional** *default-coding-system accept-default-p file* [Function]

この関数は指定されたテキストをエンコードするために、必要ならユーザーに選択を求めてコーディングシステムを選択する。指定されるテキストは通常はカレントバッファーの *from*と *to* の間のテキスト。*from*が文字列なら、その文字列がエンコードするテキストを指定して、*to*は無視される。

指定されたテキストに raw バイト (Section 33.1 [Text Representations], page 852 を参照) が含まれる場合には、`select-safe-coding-system`はそのエンコーディングに `raw-text` を提案する。

*default-coding-system*が非 `nil`なら、それは試行すべき最初のコーディングシステムである。それがテキストを処理できるなら、`select-safe-coding-system`はそのコーディングシステムをリターンする。これはコーディングシステムのリストの可能性もある。その場合にはこの関数はそれらを 1 つずつ試みる。それらをすべて試した後に、(`undecided`以外なら) カレントバッファーの `buffer-file-coding-system`の値、次に `buffer-file-coding-system`のデフォルト値、最後にユーザーがもっとも好むコーディングシステム (コマンド `prefer-coding-system`でセットできる最優先されるコーディングシステム) を試みる (Section “Recognizing Coding Systems” in *The GNU Emacs Manual* を参照)。

これらのうちいずれかのコーディングシステムが指定されたテキストすべてを安全にエンコード可能なら、`select-safe-coding-system`はそれを選択およびリターンする。それ以外ならコーディングシステムのリストからすべてのテキストをエンコードできるコーディングシステムの選択をユーザーに求めてユーザーの選択をリターンする。

*default-coding-system*は、最初の要素が `t`で他の要素がコーディングシステムであるようなリストかもしれない。その場合にはリスト内にテキストを処理できるコーディングシステムがなければ、`select-safe-coding-system`は上述した 3 つの代替えいずれを試みることなく即座にユーザーに問い合わせる。これはリスト内のコーディングシステムだけをチェックするのに手軽。

オプション引数 *accept-default-p*はユーザーとの対話なしで選択されたコーディングシステムを許容するかどうかを決定する。これが省略か `nil`なら、そのような暗黙の選択は常に許容される。非 `nil`なら関数であること。`select-safe-coding-system`は選択するコーディングシステムのベースとなるコーディングシステムを単一の引数としてその関数を呼び出す。関数

が nilをリターンしたら select-safe-coding-systemは黙って選択されたコーディングシステムを拒絶して、可能な候補リストからコーディングシステムの選択をユーザーに求める。

変数 select-safe-coding-system-accept-defaultf-pが非 nilなら、それは 1 つの引数をとる関数であること。これは *accept-default-p*引数に与えられた値をオーバーライドすることにより *accept-default-p*のかわりに使用される。

最後のステップとして選択されたコーディングシステムをリターンする前に、select-safe-coding-systemはもしリージョンのコンテンツがファイルから読み込まれたものだったとしたなら選択されたであろうコーディングシステムと、そのコーディングシステムが一致するかどうかをチェックする (異なるならその後の再 visit と編集でファイル内のデータ汚染が起こり得る)。select-safe-coding-systemは通常はこの目的のためのファイルとして buffer-file-nameを使用するが、*file*が非 nilならかわりにそのファイルを使用する (これは write-regionや類似の関数に関連し得る)。明らかな不一致が検出された場合には select-safe-coding-systemはそのコーディングシステムを選択する前にユーザーに問い合わせる。

select-safe-coding-system-function [Variable]
この変数は出力処理がテキストを安全にエンコードできないときに、テキストをエンコードするための正しいコーディングシステムの選択をユーザーに求めるために呼び出される関数。この変数のデフォルト値は select-safe-coding-system。write-regionのようにテキストをファイルに書き込んだり、process-send-regionのように別プロセスにテキストを送信する Emacs プリミティブは、coding-system-for-writeが nilにバインドされていれば、通常はこの変数の値を呼び出す (Section 33.10.6 [Specifying Coding Systems], page 874 を参照)。

以下の 2 つの関数は補完つきでユーザーにコーディングシステムの選択を求めるために使用できます。Section 20.6 [Completion], page 350 を参照してください。

read-coding-system *prompt* **&optional** *default* [Function]
この関数は文字列 *prompt*をプロンプトにミニバッファーを使用してコーディングシステムを読み取り、そのコーディングシステムの名前をシンボルとしてリターンする。*default*はユーザーの入力が空の場合にリターンするべきコーディングシステムを指定する。これはシンボルか文字列であること。

read-non-nil-coding-system *prompt* [Function]
この関数は文字列 *prompt*をプロンプトにミニバッファーを使用してコーディングシステムを読み取り、そのコーディングシステムの名前をシンボルとしてリターンする。ユーザーが空の入力を試みると再度ユーザーに問い合わせを行う。Section 33.10 [Coding Systems], page 864 を参照のこと。

33.10.5 デフォルトのコーディングシステム

このセクションでは特定のファイルや特定のサブプロセス実行時のデフォルトコーディングシステムを指定する変数、およびそれらへアクセスするための I/O 処理が使用する関数について説明します。

これらの変数は希望するデフォルトにそれらすべてを一度セットして、その後は再びそれを変更しないというアイデアにもとづいています。Lisp プログラム内の特定の処理で特定のコーディングシステムを指定するために、これらの変数を変更しないでください。かわりに coding-system-for-readや coding-system-for-writeを使用して、それらをオーバーライドしてください (Section 33.10.6 [Specifying Coding Systems], page 874 を参照)。

`auto-coding-regexp-alist` [User Option]
この変数はテキストパターンと対応するコーディングシステムの alist。要素はそれぞれ (*regexp* . *coding-system*) という形式をもつ。冒頭の数キロバイトが *regexp*にマッチするファイルは、そのコンテンツをバッファーに読み込む際に *coding-system*によりデコードされる。この alist 内のセッティングはファイル内の `coding:`タグ、および `file-coding-system-alist` (以下参照) の内容より優先される。デフォルト値は、Emacs が自動的に Babyl フォーマットのメールファイルを認識してコード変換なしでそれらを読み取れるようにセットされている。

`file-coding-system-alist` [User Option]
この変数は特定のファイルの読み書きに使用するコーディングシステムを指定する alist。要素はそれぞれ (*pattern* . *coding*) という形式をもち、*pattern*は特定のファイル名にマッチする正規表現。この要素は *pattern*にマッチするファイル名に適用される。

要素の CDR となる *coding*はコーディングシステム、2 つのコーディングシステムを含むコンスセル、または関数名 (関数定義をもつシンボル) であること。*coding*がコーディングシステムなら、そのコーディングシステムはファイルの読み込みと書き込みの両方で使用される。*coding* が 2 つのコーディングシステムを含むコンスセルなら、CAR はデコード用のコーディングシステム、CDR はエンコード用のコーディングシステムを指定する。

*coding*が関数名なら、それは `find-operation-coding-system`に渡されたすべての引数からなるリストを唯一の引数とする関数であること。これはコーディングシステム、または 2 つのコーディングシステムを含むコンスセルをリターンしなければならない。この値は上記と同じ意味をもつ。

coding (または上記関数のリターン値) が `undecided`なら通常のコード検出が行われる。

`auto-coding-alist` [User Option]
この変数は特定のファイルの読み書きに使用するコーディングシステムを指定する alist。この変数の形式は `file-coding-system-alist`の形式と似ているが、後者と異なるのはこの変数がファイル内の `coding:`タグより優先されること。

`process-coding-system-alist` [Variable]
この変数は何のプログラムがサブプロセス内で実行中かによって、そのサブプロセスにたいしてどのコーディングシステムを使用するかを指定する alist。これは `file-coding-system-alist`と同じように機能するが、*pattern*がそのサブプロセスを開始するために使用されたプログラム名にたいしてマッチされる点が異なる。コーディングシステム、または alist 内で指定されたコーディングシステムは、そのサブプロセスへの I/O に使用されるコーディングシステムの初期化に使用されるが、`set-process-coding-system`を使用して後から他のコーディングシステムを指定できる。

警告: データからコーディングシステムを判断する `undecided`のようなコーディングシステムは、非同期のサブプロセスでは完全な信頼性をもって機能はしない。これは Emacs が非同期サブプロセスの出力を到着によりバッチ処理するためである。そのコーディングシステムが文字コード変換や行末変換を未指定にしておくと、Emacs は一度に 1 バッチから正しい変換の検出を試みなければならず、これは常に機能するとは限らない。

したがって非同期サブプロセスでは可能なら文字コード変換と行末変換の両方を判断するコーディングシステム、つまり `undecided`や `latin-1`ではなく `latin-1-unix`のようなコーディングシステムを使用すること。

`network-coding-system-alist` [Variable]
この変数はネットワークストリームに使用するコーディングシステムを指定する alist。これは `file-coding-system-alist`と同じように機能するが、要素内の *pattern*がポート番号、ま

たは正規表現かもしれない点が異なる。正規表現ならそのネットワークストリームのオープンに使用されたネットワークサービス名にたいしてマッチされる。

`default-process-coding-system` [Variable]
この変数は他に何を行うか指定されていない際に、サブプロセス (とネットワークストリーム) への入出力に使用するコーディングシステムを指定する。

値は (*input-coding* . *output-coding*) という形式のコンスセルであること。ここで *input-coding* はサブプロセスからの入力、*output-coding* はサブプロセスへの出力に適用される。

`auto-coding-functions` [User Option]
この変数はファイルのデコードされていないコンテンツにもとづいて、ファイルにたいするコーディングシステムの判断を試みる関数のリストを保持する。

このリスト内の各関数はカレントバッファー内のテキストを調べるように、ただしいかなる方法にせよそれを変更しないよう記述されるべきである。そのバッファーはファイルの一部であるデコードされていない Unicode テキストを含むだろう。各関数はポイントを始点に何文字を調べる可を告げる唯一の引数 *size* をとること。そのファイルにたいするコーディングシステムの決定に関数が成功したら、そのコーディングシステムをリターンすること。それ以外は `nil` をリターンするべきである。

このリスト内の関数はファイルが visit される際に Emacs がファイルのコンテンツのデコードをしようとする場合、および/またはそのファイルのバッファーを保存しようとする際に Emacs がファイルのコンテンツのエンコード方法を決定しようと場合に呼び出されるかもしれない。

ファイルに ‘`coding:`’タグがある場合にはそれが優先されるので、これらの関数が呼び出されることはないだろう。

`find-auto-coding` *filename size* [Function]
この関数は *filename* に適するコーディングシステムの判定を試みる。これは上記で説明した変数により指定されたルールのいずれかにマッチするまで、それらの変数を順に使用してファイルを visit するバッファーを調べる。そして (*coding* . *source*) という形式のコンスセルをリターンする。ここで *coding* は使用するコーディングシステム、*source* は `auto-coding-alist`、`auto-coding-regexp-alist`、`:coding`、`auto-coding-functions` のいずれかであるようなシンボルであり、マッチングルールとして提供されるルールを示す。値 `:coding` はファイル内の `coding:` タグによりコーディングシステムが指定されたことを意味する (Section “coding tag” in *The GNU Emacs Manual* を参照)。マッチングルールを調べる順序は `auto-coding-alist`、`auto-coding-regexp-alist`、`coding:`、`auto-coding-functions` の順。マッチングルールが見つからなければこの関数は `nil` をリターンする。

2 つ目の引数 *size* はポイントの後のテキストの文字単位のサイズ。この関数はポイントの後の *size* 文字のテキストだけを調べる。`coding:` タグが置かれる箇所としてはファイルの先頭 2 行が想定される箇所の 1 つなので、通常はバッファーの先頭位置でこの関数を呼び出すこと。その場合には *size* はそのバッファーのサイズであること。

`set-auto-coding` *filename size* [Function]
この関数はファイル *filename* に適するコーディングシステムをリターンする。これはコーディングシステムを探すために `find-auto-coding` を使用する。コーディングシステムを決定できなかったら、この関数は `nil` をリターンする。引数 *size* の意味は `find-auto-coding` と同様。

`find-operation-coding-system` *operation* **&rest** *arguments* [Function]
この関数は *operation* を *arguments* で行う際に、(デフォルトで) 使用するコーディングシステムをリターンする。値は以下の形式:

(*decoding-system* . *encoding-system*)

1 つ目の要素 *decoding-system*はデコード (*operation*がデコードを行う場合)、*encoding-system*はエンコード (*operation*がエンコードを行う場合) に使用するコーディングシステム。

引数 *operation*はシンボルで write-region、start-process、call-process、call-process-region、insert-file-contents、open-network-streamのいずれかであること。これらは文字コード変換と行末変換を行うことができる Emacs の I/O プリミティブの名前である。

残りの引数は対応する I/O プリミティブに与えられる引数と同じであること。そのプリミティブに応じてこれらの引数のうち 1 つがターゲットとして選択される。たとえば *operation*がファイル I/O ならファイル名を指定する引数がターゲット。サブプロセス用のプリミティブではプロセス名がターゲット。open-network-streamではサービス名またはポート番号がターゲット。

*operation*に応じてこの関数は file-coding-system-alist、process-coding-system-alist、network-coding-system-alistの中からターゲットを探す。この alist 内でターゲットが見つかったら find-operation-coding-systemは alist 内の association(連想: キーと連想値からなるコンスセル)、それ以外は nilをリターンする。

*operation*が insert-file-contentsならターゲットに対応する引数は (*filename* . *buffer*)という形式のコンスセルだろう。この場合には *filename*は file-coding-system-alist内で照合されるファイル名であり、*buffer*はそのファイルの (デコードされていない) コンテンツを含むバッファー。file-coding-system-alistがこのファイルにたいして呼び出す関数を指定していて、かつ (通常行われるように) ファイルのコンテンツを調べる必要があるならファイルを読み込むかわりに *buffer*のコンテンツを調べること。

33.10.6 単一の操作にたいするコーディングシステムの指定

変数 coding-system-for-readおよび/または coding-system-for-writeをバインドすることにより、特定の操作にたいしてコーディングシステムを指定できます。

coding-system-for-read [Variable]

この変数が非 nilなら、それはファイルの読み込みや同期サブプロセスプロセスからの入力にたいして使用するコーディングシステムを指定する。

これは非同期サブプロセスやネットワークストリームにも適用されるが方法は異なる。サブプロセス開始時やネットワークストリームオープン時の coding-system-for-readの値は、サブプロセスやネットワークストリームにたいして入力のデコードメソッドを指定する。そのサブプロセスやネットワークストリームにたいして、オーバーライドされるまでそれが使用され続ける。

特定の I/O 操作にたいして letでバインドするのがこの変数の正しい使い方である。この変数のグローバル値は常に nilであり、他の値にグローバルにセットするべきではない。以下はこの変数の正しい使用例:

```
;; 文字コード変換なしでファイルを読み込む
(let ((coding-system-for-read 'no-conversion))
  (insert-file-contents filename))
```

この変数の値が非 nilのときは file-coding-system-alist、process-coding-system-alist、network-coding-system-alistを含む、入力にたいして使用するコーディングシステムを指定するすべてのメソッドよりこの変数が優先される。

`coding-system-for-write` [Variable]
これは `coding-system-for-read`と同じように機能するが、入力ではなく出力に適用される点が異なる。これはファイルへの書き込み、同様にサブプロセスやネットワークストリームへの出力の送信にも適用される。これは Emacs がサブプロセスを呼び出す際のコマンドライン引数のエンコーディングにも適用される。

単一の操作が `call-process-region`や `start-process`のように入力と出力の両方を行う際には、`coding-system-for-read`と `coding-system-for-write`の両方がそれに影響する。

`coding-system-require-warning` [Variable]
`coding-system-for-write`に非 `nil`値をバインドすることにより、`select-safe-coding-system-function`が指定する関数の呼び出しによる出力プリミティブを抑制する (Section 33.10.4 [User-Chosen Coding Systems], page 870 を参照)。これは *C-x RET c* (`universal-coding-system-argument`) が `coding-system-for-write`をバインドすることにより機能して、かつ Emacs はユーザーの選択にしたがう必要があるからである。Lisp プログラムが書き込むテキストのエンコーディングに安全ではないかもしれない値を `coding-system-for-write`にバインドする場合には、`coding-system-require-warning`にも非 `nil`値をバインドできる。これはたとえ `coding-system-for-write`が非 `nil`でも `select-safe-coding-system-function`の値の呼び出しによる出力プリミティブにエンコードのチェックを強制する。または指定されたエンコーディングを使用する前に、明示的に `select-safe-coding-system`を呼び出すこと。

`inhibit-eol-conversion` [User Option]
この変数が非 `nil`なら、どのコーディングシステムが指定されたかに関わらず行末変換は何も行われない。これは Emacs すべての I/O やサブプロセスにたいするプリミティブ、および明示的なエンコード関数 (Section 33.10.7 [Explicit Encoding], page 876 を参照) とデコード関数に適用される。

ある操作にたいして固定された 1 つのコーディングシステムではなく複数のコーディングシステムを選択する必要があることがあります。Emacs では使用するコーディングシステムにたいして優先順位を指定できます。これは `find-coding-systems-region`(Section 33.10.3 [Lisp and Coding Systems], page 867 を参照) のような関数によりリターンされるコーディングシステムのリストのソート順に影響します。

`coding-system-priority-list` **&optional** *highestp* [Function]
この関数はコーディングシステムのカレント優先順にコーディングシステムのリストをリターンする。オプション引数 *highestp*が非 `nil`なら、それはもっとも高い優先度のコーディングシステムだけをリターンすることを意味する。

`set-coding-system-priority` **&rest** *coding-systems* [Function]
この関数はコーディングシステムの優先リストの先頭に *coding-systems*を配置して、それらを他のコーディングシステムすべてより高い優先度とする。

`with-coding-priority` *coding-systems* **&rest** *body* [Macro]
このマクロは *coding-systems*をコーディングシステム優先リスト先頭に配置して、`progn` (Section 11.1 [Sequencing], page 145 を参照) が行うように *body*を実行する。*coding-systems* は *body*実行中に選択するコーディングシステムのリストであること。

33.10.7 明示的なエンコードとデコード

Emacs 内外へテキストを転送するすべての操作は、そのテキストをエンコードまたはデコードする能力をもっています。このセクション内の関数を使用してテキストの明示的なエンコードやデコードを行うことができます。

エンコード結果やデコーディングへの入力は通常のテキストではありません。これらは理論的には一連のバイト値から構成されており、すなわち一連の ASCII文字と 8 ビット文字から構成されます。ユニバイトのバッファーや文字列では、これらの文字は 0 から#xFF(255) の範囲のコードをもちます。マルチバイトのバッファーや文字列では 8 ビット文字は#xFF より大きい文字コードをもちますが (Section 33.1 [Text Representations], page 852 を参照)、そのようなテキストのエンコードやデコードの際に Emacs は透過的にそれらを単一バイト値に変換します。

コンテンツを明示的にデコードできるようにバイトシーケンスとしてバッファーにファイルを読み込むには、`insert-file-contents-literally` (Section 25.3 [Reading from Files], page 537 を参照) を使用するのが通常の方法です。あるいは `find-file-noselect`でファイルを visit する際には、引数 *rawfile*に非 `nil`を指定することもできます。これらのメソッドの結果はユニバイトバッファーになります。

テキストを明示的にエンコードした結果であるバイトシーケンスは、たとえばそれを `write-region` (Section 25.4 [Writing to Files], page 538 を参照) で書き込み、`coding-system-for-write`を `no-conversion`にバインドすることによりエンコードを抑制する等、それをファイルまたはプロセスへコピーするのが通常の使い方です。

以下はエンコードやデコードを明示的に行う関数です。エンコード関数とはバイトシーケンスを生成し、デコード関数とはバイトシーケンスを操作する関数のことを意味します。これらの関数はすべてテキストプロパティを破棄します。これらは自身が使用したコーディングシステムを、正確に `last-coding-system-used`にセットすることも行います。

`encode-coding-region` *start end coding-system* **&optional** *destination* [Command]

このコマンドは *start*から *end*のテキストをコーディングシステム *coding-system*でエンコードする。バッファー内の元テキストは通常はエンコードされたテキストで置き換えられるが、オプション引数 *destination*でそれを変更できる。*destination*がバッファーなら、エンコードされたテキストはそのバッファーのポイントの後に挿入される (ポイントは移動しない)。`t`ならこのコマンドはエンコードされたテキストを挿入せずにユニバイトとしてリターンする。

エンコードされたテキストが何らかのバッファーに挿入された場合には、このコマンドはエンコードされたテキストの長さをリターンする。

エンコードされた結果は理論的にはバイトシーケンスだが、バッファーが以前マルチバイトだったならマルチバイトのまま留まり、すべての 8 ビットのバイトはマルチバイト表現に変換される (Section 33.1 [Text Representations], page 852 を参照)。

期待しない結果となる恐れがあるので、テキストをエンコードする際には *coding-system*に `undecided`を使用してはならない。*coding-system*にたいして自明な適値が存在しなければ適切なエンコードを提案させるために、かわりに `select-safe-coding-system`を使用すること (Section 33.10.4 [User-Chosen Coding Systems], page 870 を参照)。

`encode-coding-string` *string coding-system* **&optional** *nocopy buffer* [Function]

この関数はコーディングシステム *coding-system*で *string*内のテキストをエンコードする。これはエンコードされたテキストを含む新たな文字列をリターンするが、*nocopy*が非 `nil`の場

合には、それが些細なエンコード処理ならこの関数は *string*自身をリターンする。エンコード結果はユニバイト文字列。

`decode-coding-region` *start end coding-system* **&optional** *destination* [Command]

このコマンドはコーディングシステム *coding-system*で、*start*から *end*のテキストをデコードする。明示的なデコードを使いやすくするためにデコード前のテキストはバイトシーケンス値であるべきだが、マルチバイトとユニバイトのバッファーいずれでも許すようになっている (マルチバイトバッファーの場合 raw バイト値は 8 ビット文字で表現されていること)。デコードされたテキストにより通常はバッファー内の元のテキストは置き換えられるが、オプション引数 *destination*はそれを変更する。*destination*がバッファーなら、デコードされたテキストはそのバッファーのポイントの後に挿入される (ポイントは移動しない)。これが `t`ならこのコマンドはデコードされたテキストを挿入せずにマルチバイト文字列としてリターンする。

デコードしたテキストを何らかのバッファーに挿入すると、このコマンドはデコード済みテキストの長さをリターンする。バッファーがユニバイトバッファー (Section 33.4 [Selecting a Representation], page 855 を参照) なら、デコード済みテキストの内部表現 (Section 33.1 [Text Representations], page 852 を参照) が個別のバイトとしてバッファーに挿入される。

このコマンドはデコードされたテキストにテキストプロパティ`charset`を put する。このプロパティの値は元のテキストのデコードに使用された文字セットを示す。

`decode-coding-string` *string coding-system* **&optional** *nocopy buffer* [Function]

この関数は *coding-system*で *string*内のテキストをデコードする。これはデコードされたテキストを含む新たな文字列をリターンするが、*nocopy*が非 `nil`の場合には、それが些細なデコード処理なら *string*自体をリターンするかもしれない。明示的なデコードを使いやすくするために、*string*のコンテンツはバイトシーケンス値をもつユニバイト文字列であるべきだが、マルチバイト文字列も許すようになっている (マルチバイト形式で 8 ビットバイトを含むと仮定する)。

オプション引数 *buffer*がバッファーを指定する場合には、デコードされたテキストはバッファー内のポイントの後に挿入される (ポイントは移動しない)。この場合にはリターン値はデコードされたテキストの長さとなる。バッファーがユニバイトバッファーなら、デコード済みテキストの内部表現が個別のバイトとしてバッファーに挿入される。

この関数はデコードされたテキストにテキストプロパティ`charset`を put する。このプロパティの値は元のテキストのデコードに使用された文字セットを示す。

```
(decode-coding-string "Gr\374ss Gott" 'latin-1)
     ⇒ #("Gr^^c3^^bcss Gott" 0 9 (charset iso-8859-1))
```

`decode-coding-inserted-region` *from to filename* **&optional** *visit beg end replace* [Function]

この関数は *from*から *to*のテキストを、あたかも与えられた残りの引数で `insert-file-contents`を使用してファイル *filename*から読み込んだかのようにデコードする。

デコードせずにファイルからテキストを読み込んだ後で、やはりデコードすることを決心したときに使用するのがこの関数の通常の使い方である。テキストを削除して再度読み込むかわりに、この関数を呼び出せばデコードして読み込むことができる。

33.10.8 端末 I/O のエンコーディング

Emacs はキーボード入力のデコード、および端末出力のエンコードにコーディングシステムを使用できます。これは Latin-1 のような特定のエンコーディングを使用したテキストの送信や表示を行う端

末にとって有用です。端末 I/O をエンコードまたはデコードする際には、Emacs は `last-coding-system-used`をセットしません。

`keyboard-coding-system` **&optional** *terminal* [Function]
この関数は *terminal*からのキーボード入力をデコードするために使用するコーディングシステムをリターンする。`no-conversion`という値は何のデコーディングも行われていないことを意味する。*terminal*が省略または `nil`なら、それは選択されたフレームの端末を意味する。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`set-keyboard-coding-system` *coding-system* **&optional** *terminal* [Command]
このコマンドは *terminal*からのキーボード入力のデコードに使用するコーディングシステムとして *coding-system*を指定する。*coding-system*が `nil`なら、キーボード入力をデコードしないことを意味する。*terminal*がフレームなら、それはそのフレームの端末を意味する。`nil`ならそれはカレントで選択されたフレームの端末を意味する。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`terminal-coding-system` **&optional** *terminal* [Function]
この関数は *terminal*からの端末出力のエンコードに使用中のコーディングシステムをリターンする。`no-conversion`という値は何のデコーディングも行われていないことを意味する。*terminal*がフレームならそれはそのフレームの端末を意味する。`nil`ならそれはカレントで選択されたフレームの端末を意味する。

`set-terminal-coding-system` *coding-system* **&optional** *terminal* [Command]
この関数は *terminal*からの端末出力のエンコードに使用するためのコーディングシステムとして *coding-system*を指定する。*coding-system*が `nil`なら端末出力をエンコードしないことを意味する。*terminal*がフレームならそれはそのフレームの端末を意味する。`nil`ならそれはカレントで選択されたフレームの端末を意味する。

33.11 入力メソッド

入力メソッド *(input methods)* はキーボードから非 ASCII文字を簡単に入力する手段を提供します。プログラムが読み取ることを意図して非 ASCII文字とエンコーディングを相互に変換するコーディングシステムとは異なり、入力メソッドはヒューマンフレンドリーなコマンドを提供します (テキストを入力するためにユーザーが入力メソッドを使う方法については Section “Input Methods” in *The GNU Emacs Manual* を参照)。入力メソッドの定義方法はまだこのマニュアルにはありませんが、ここではそれらの使い方について説明します。

現在のところ入力メソッドは文字列で名前をもっていますが、将来的には入力メソッド名としてシンボルも利用可能になるかもしれません。

`current-input-method` [Variable]
この変数はカレントバッファーで現在アクティブな、入力メソッドの名前を保持する (方法に関わらずセット時には各バッファーで自動的にローカルになる)。バッファーで現在アクティブな入力メソッドがなければ値は `nil`。

`default-input-method` [User Option]
この変数は入力メソッドを選択するコマンドにたいしてデフォルトの入力メソッドを保持する。`current-input-method`と異なり、この変数は通常はグローバルである。

`set-input-method` *input-method* [Command]
このコマンドはカレントバッファーで入力メソッド *input-method*をアクティブにする。同様に `default-input-method`に *input-method*のセットも行う。*input-method*が `nil`なら、このコマンドはカレントバッファーで入力メソッドを非アクティブにする。

`read-input-method-name` *prompt* **&optional** *default inhibit-null* [Function]
この関数はプロンプト *prompt*とともにミニバッファーで入力メソッドの名前を読み取る。*default*が非 `nil`の場合には、ユーザーの入力が空ならそれがデフォルトとしてリターンされる。しかし *inhibit-null*が非 `nil`なら空の入力はエラーをシグナルする。

リターン値は文字列。

`input-method-alist` [Variable]
この変数はサポートされているすべての入力メソッドを定義する。各要素は 1 つの入力メソッドを定義して、それぞれ以下の形式をもつ:

```
(input-method language-env activate-func
 title description args...)
```

ここで *input-method*はメソッド名の文字列、*language-env*はこの入力メソッドが推奨される言語環境の名前の文字列 (これはドキュメントとしての目的のみの役割を果たす)。

*activate-func*はこのメソッドをアクティブにするために呼び出す関数、もしあれば *args*は *activate-func*に渡す引数。つまり *activate-func*の引数は *input-method*と *args*。

*title*は、その入力メソッドがアクティブな間にモードライン内に表示するための文字列、*description*はそのメソッドを説明して、それが何に適するかを説明する文字列。

入力メソッドのための基本的インターフェースは変数 `input-method-function`です。Section 21.8.2 [Reading One Event], page 405 と Section 21.8.4 [Invoking the Input Method], page 409 を参照してください。

33.12 locale

POSIX では、言語に関連する機能において使用する言語を制御するために locale という概念があります。以下の Emacs 変数は Emacs がこれらの機能と相互作用する方法を制御します。

`locale-coding-system` [Variable]
この変数は標準出力とエラーストリームへのバッチ出力の送信、`format-time-string`にたいする format 引数のエンコーディング、`format-time-string`のリターン値のデコーディングに際してシステムエラーメッセージ (および X ウィンドウシステムに限りキーボード入力) をデコーディングするコーディングシステムを指定する。

`system-messages-locale` [Variable]
この変数はシステムエラーメッセージを生成するために使用する locale を指定する。locale 変更によりメッセージが異なる言語になったり異なる表記になり得る。この変数が `nil`なら通常の POSIX 方式のように locale は環境変数により指定される。

`system-time-locale` [Variable]
この変数はタイムバリューをフォーマットするために使用する locale を指定する。locale 変更により異なる慣習によりメッセージが表示され得る。この変数が `nil`なら通常の POSIX 方式のように locale は環境変数により指定される。

`locale-info` *item* [Function]

この変数は、もし利用可能ならカレント POSIX locale にたいする locale データ *item*をリターンする。*item*は以下のシンボルのいずれかであること:

`codeset` 文字列として文字セットをリターンする (locale アイテムの `CODESET`)。

`days` 曜日名からなる 7 要素のベクターをリターンする (locale アイテムの `DAY_1`から `DAY_7`)。

`months` 月の名前からなる 12 要素のベクターをリターンする (locale アイテムの `MON_1` から `MON_12`)。

`paper` `(`*`width height`*`)` という 2 つの整数のリストで、デフォルト用紙サイズを mm 単位でリターンする (locale アイテム`_NL_PAPER_WIDTH`と`_NL_PAPER_HEIGHT`)。

システムが要求された情報を提供できなかったり、*item*が上記いずれのシンボルでもなければ値は `nil`。リターン値内のすべての文字列は `locale-coding-system`を使用してデコードされる。locale と locale アイテムについての詳細な情報は Section “Locales” in *The GNU Libc Manual* を参照のこと。

34 検索とマッチング

GNU Emacs はバッファーから指定されたテキストを検索するために 2 つの手段を提供します。それは文字列の正確一致検索 (exact string search) と正規表現検索 (regular expression search) です。正規表現検索の後で、マッチしたテキストが正規表現全体にマッチしたのか、それとも正規表現のさまざまな部分に一致したかを判断するためにマッチデータ *(match data)* を調べることができます。

'`skip-chars...`'関連の関数もある種の検索を行います。Section 30.2.7 [Skipping Characters], page 764 を参照してください。文字プロパティ内の変更の検索はSection 32.19.3 [Property Search], page 817 を参照してください。

34.1 文字列の検索

バッファー内のテキストを検索するためのプリミティブ関数が存在します。これらはプログラム内での使用を意図したものですがインタラクティブに呼び出すこともできます。これらをインタラクティブに呼び出すと検索文字列の入力を求めて、引数 *limit*と *noerror*は `nil`、*repeat*は 1 になります。インタラクティブ検索に関するより詳細な情報は Section "Searching and Replacement" in *The GNU Emacs Manual* を参照してください。

以下の検索関数はバッファーがマルチバイトバッファーならマルチバイト、ユニバイトバッファーならユニバイトに検索文字列を変換します。Section 33.1 [Text Representations], page 852 を参照してください。

`search-forward` *string* **&optional** *limit noerror count* [Command]

この関数は *string*にたいする正確なマッチをポイントから前方に検索する。成功したら見つかったマッチの終端にポイントをセットしてポイントの新たな値をリターンする。マッチが見つからない場合の値と副作用は *noerror* (以下参照) に依存する。

以下の例ではポイントは最初は行の先頭にある。その後の `(search-forward "fox")`によってポイントは'`fox`'の最後の文字の後に移動する:

```
---------- Buffer: foo ----------
⋆The quick brown fox jumped over the lazy dog.
---------- Buffer: foo ----------

(search-forward "fox")
     ⇒ 20

---------- Buffer: foo ----------
The quick brown fox⋆ jumped over the lazy dog.
---------- Buffer: foo ----------
```

引数 *limit*は検索の境界を指定するもので、それはカレントバッファー内の位置であること。その位置を超えるようなマッチは受け入れられない。*limit*が省略または `nil`の場合のデフォルトは、そのバッファーのアクセス可能範囲の終端。

検索失敗時に何が起こるかは *noerror*の値に依存する。*noerror*が `nil`なら `search-failed`はエラーをシグナルする。*noerror*が `t`なら `search-forward`は `nil`をリターンして何も行わない。*noerror*が `nil`と `t`いずれでもなければ、`search-forward`はポイントを境界上限に移動して `nil`をリターンする。

引数 *noerror*はマッチに失敗した有効な検索だけに影響する。無効な引数は *noerror*とは無関係にエラーとなる。

*count*が正の数 *n*なら、それは繰り返し回数の役目をもつ。検索は *n*回繰り返され、前回のマッチの終端から毎回検索が開始される。これらの連続する検索が成功した場合、関数は成功となりポイントを移動して新たな値をリターンする。それ以外は検索失敗となり、上述したように結果は *noerror*の値に依存する。*count*が負の数 $-n$なら、それは逆方向 (後方) への検索の繰り返し回数 *n*としての役目をもつ。

`search-backward` *string* **&optional** *limit noerror count* [Command]
この関数はポイントから後方に *string*を検索する。これは `search-forward`と似ているが、前方ではなく後方に検索する点が異なる。後方への検索ではポイントはマッチの先頭に残される。

`word-search-forward` *string* **&optional** *limit noerror count* [Command]
この関数はポイントから前方に *string*にたいする単語 (word) のマッチを検索する。マッチが見つかったら見つかったマッチの終端にポイントをセットしてポイントの新たな値をリターンする。

単語マッチは *string*を単語のシーケンスとみなし、それらを分割する句読点は無視する。これはバッファーから同じ単語シーケンスを検索する。単語はそれぞれバッファー内で明確に区別されていなければならない (単語 '`ball`'の検索は単語 '`balls`'にマッチしない) が、句読点やスペース等の細部は無視される ('`ball boy`'を検索すると '`ball. Boy!`'にマッチする)。

以下の例ではポイントは最初バッファー先頭にある。検索によりポイントは '`y`'と '`!`'の間に残される。

```
---------- Buffer: foo ----------
⋆He said "Please!  Find
the ball boy!"
---------- Buffer: foo ----------

(word-search-forward "Please find the ball, boy.")
     ⇒ 39

---------- Buffer: foo ----------
He said "Please!  Find
the ball boy⋆!"
---------- Buffer: foo ----------
```

*limit*が非 `nil`なら、それはカレントバッファー内の位置であること。これはその検索の境界上限を指定する。見つかったマッチはその位置を超えてはならない。

*noerror*が `nil`なら `word-search-forward`はエラーをシグナルする。*noerror*が `t`なら、エラーをシグナルするかわりに `nil`をリターンする。*noerror*が `nil`と `t`いずれでもなければ、ポイントを *limit*(またはバッファーのアクセス可能範囲の終端) に移動して `nil`をリターンする。

*count*が正の数なら、それは連続して検索する回数を指定する。ポイントは最後のマッチの終端に配置される。*count*が負の数なら、逆方向に検索してポイントは最後のマッチの先頭に配置される。

`word-search-forward`および関連する関数は、*string*から句読点を無視した正規表現に変換するために、内部的には関数 `word-search-regexp`を使用する。

`word-search-forward-lax` *string* **&optional** *limit noerror count* [Command]
このコマンドは `word-search-forward`と同じだが、*string*が空白で開始か終了していなければ、*string*の先頭か終端が単語境界にマッチする必要がない点が異なる。たとえば '`ball boy`' の検索は '`ball boyee`'にはマッチするが、'`balls boy`'にはマッチしない。

`word-search-backward` *string* **&optional** *limit noerror count* [Command]
この関数はポイントから後方へ*string*にマッチする単語を検索する。この関数は`word-search-forward`と同様だが、後方に検索して通常はマッチの先頭にポイントを残す点が異なる。

`word-search-backward-lax` *string* **&optional** *limit noerror count* [Command]
このコマンドは`word-search-backward`と同じだが、文字列が空白で開始か終了していなければ、*string*の先頭か終端が単語境界にマッチする必要がない点が異なる。

34.2 検索と大文字小文字

デフォルトの Emacs 検索では検索するテキストの case(大文字と小文字) は無視されます。検索対象に‘`FOO`’を指定すると、‘`Foo`’や‘`foo`’もマッチとみなされます。これは正規表現にも適用されます。つまり‘`[aB]`’は‘`a`’、‘`A`’、‘`b`’、‘`B`’にもマッチするでしょう。

この機能が望ましくなければ変数`case-fold-search`に`nil`をセットしてください。その場合にはすべての文字は case を含めて正確にマッチしなければなりません。これはバッファーローカル変数です。この変数の変更はカレントバッファーだけに影響を与えます (Section 12.11.1 [Intro to Buffer-Local], page 190 を参照)。かわりにデフォルト値を変更することもできます。Lisp コードでは`let`を使用して`case-fold-search`を望む値にバインドするほうが、より一般的でしょう。

ユーザーレベルのインクリメンタル検索機能では case の区別が異なることに注意してください。検索文字列に含まれるのが小文字だけなら検索は case を無視しますが、検索文字列に 1 つ以上の大文字が含まれれば検索は case を区別するようになります。しかし Lisp コード内で使用される検索関数では、これは何も行いません。Section “Incremental Search” in *The GNU Emacs Manual* を参照してください。

`case-fold-search` [User Option]
このバッファーローカル変数は検索が case を無視するべきかどうかを決定する。この変数が`nil`なら検索は case を無視しない。それ以外 (とデフォルト) では case を無視する。

`case-replace` [User Option]
この変数は高レベルの置換関数が case を保持するべきかどうかを決定する。この変数が`nil`なら、それは置換テキストをそのまま使用することを意味する。非`nil`値は置換されるテキストに応じて、置換テキストの case を変換することを意味する。

この変数は関数`replace-match`の引数として渡すことにより使用される。Section 34.6.1 [Replacing Match], page 897 を参照のこと。

34.3 正規表現

正規表現 *(regular expression)*、略して *regexp* は文字列の (もしかしたら無限の) セットを表すパターンのことです。regexp にたいするマッチの検索はとても強力な処理です。このセクションでは regexp の記述方法、それ以降のセクションではそれらを検索する方法を示します。

正規表現を対話的に開発するために *`M-x re-builder`*コマンドを使用できます。このコマンドは別のバッファーに即座に視覚的なフィードバックを表示することにより、正規表現を作成するための便利なインターフェースを提供します。regexp 編集とともにターゲットとなるバッファーのすべてのマッチがハイライトされます。カッコで括られた regexp の部分式 (sub-expression) は別のフェイスで表示され、非常に複雑な regexp を簡単に検証することが可能になります。

34.3.1 正規表現の構文

正規表現は少数の文字が特別な構成要素であり、残りは通常の文字であるような構文をもちます。通常の文字はその文字自身だけにマッチするシンプルな正規表現です。特別な文字は '.'、'*'、'+'、'?'、'['、'^'、'$'、および '\'です。将来に新たなスペシャル文字が定義されることはないでしょう。文字候補で終わる場合には ']'はスペシャル文字です。文字候補の間では '-'はスペシャル文字です。'[:'と対応する ':]'は文字候補内の文字クラスです。正規表現内に出現する他の文字は '\'が前置されていない限り通常の文字です。

たとえば 'f'はスペシャル文字ではなく通常文字なので、'f'は文字列 'f'にマッチして他の文字にはマッチしない正規表現です (これは文字列 'fg'にはマッチしないが、その文字列の部分にマッチする)。同様に 'o'は 'o'だけにマッチします。

任意の 2 つの正規表現 aと bを結合することができます。結合した結果は文字列の先頭からある長さの文字列が aにマッチして、残りの文字列が bにマッチするような文字列にマッチする正規表現になります。

単純な例として文字列 'fo'だけにマッチする正規表現の構成要素 'fo'を取得するために正規表現 'f'と 'o'を結合できます。

34.3.1.1 正規表現内の特殊文字

以下は正規表現内で特別な文字のリストです:

'.' (Period)
: これは改行を除く 1 文字にマッチするスペシャル文字。結合を使用して 'a.b'のような正規表現を作成できる。これは 'a'で始まり 'b'で終わる 3 文字の文字列にマッチする。

'*'
: これはそれ自身が構成要素ではない。これは前置された正規表現を可能な限り繰り返したものにマッチすることを意味する後置演算子である。したがって 'o*'は任意の個数の 'o'にマッチする ('o'を含まない場合にもマッチする)。

  '*'は常に前置された表現の最小の表現に適用される。つまり 'fo*'は 'o'の繰り返しであり 'fo'の繰り返しではない。これは 'f'、'fo'、'foo'、... にマッチする。

  マッチを行う処理は構成要素 '*'をマッチングにより即座に見つけ得る回数分処理して、その後にパターンの残りを継続する。これが失敗したら残りのパターンのマッチが可能になるかもしれないという期待のもとに、'*'の変更された構成のうちいくつかのマッチを破棄することでバックトラッキングが発生する。たとえば文字列 'caaar'にたいして 'ca*ar'をマッチングすると、'a*'はまず 3 つすべての 'a'へのマッチを試みる。しかし残りのパターンは 'ar'であり、マッチ対象に残されているのは 'r'だけなので試みは失敗する。'a*'にたいする次の代替策は、2 つの 'a'だけへのマッチである。この選択では残りの regexp のマッチは成功する。

  警告: ネストされた繰り返し処理は、それらが曖昧なマッチとなるような場合には無期限な長時間の実行となり得る。たとえば文字列 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxz'にたいして正規表現 '\(x+y*\)*a'のマッチを試みると、それが最終的に失敗するまでに数時間を要する可能性がある。Emacs はその試みのいずれも機能しないと結論する前に、'x'のグループ化のそれぞれを試みなければならない。さらに悪いことに '\(x*\)*'は無数の方法で null 文字列にマッチ可能なので無限ループを引き起こす。これらの問題を避けるにはネストされた繰り返しがバックトラッキングでの組み合わせ爆発 (combinatorial explosion) が発生しないことを確実にするために注意深くチェックすること。

‘+’ これは‘*’のような後置演算子だが前置された表現に少なくとも 1 回マッチしなければならない点が異なる。たとえば‘ca+r’は文字列‘car’や‘caaaar’にマッチするが文字列‘cr’にはマッチせず、その一方で‘ca*r’はこれら 3 つすべての文字列にマッチする。

‘?’ これは‘*’のような後置演算子だが前置された表現に 1 回、またはマッチしないかのいずれかでなければならない点が異なる。例えば‘ca?r’は‘car’と‘cr’にマッチするが他にはマッチしない。

‘*?’, ‘+?’, ‘??’

演算子‘*’、‘+’、‘?’には非欲張り *(non-greedy)* な変種が存在する。これらの演算子が可能な最長の部分文字列 (含まれる表現全体へのマッチと等しい) とマッチするのにたいして、非欲張りな変種は可能な最短の部分文字列 (含まれる表現全体と等しい) にマッチする。

たとえば正規表現‘c[ad]*a’を文字列‘cdaaada’に適用すると文字列全体にマッチするが、正規表現‘c[ad]*?a’を同じ文字列に適用すると‘cda’だけにマッチする (ここでマッチが許された表現全体にたいする‘[ad]*?’の可能な最短マッチは‘d’)。

‘[...]’ これは‘[’で始まり‘]’で終端される文字候補 *(character alternative)*。もっとも単純なケースでは、この 2 つのカッコ (brackets) の間にある文字が、この文字候補がマッチ可能な文字。

したがって‘[ad]’は 1 つの‘a’と 1 つの‘d’の両方にマッチし、‘[ad]*’は‘a’と‘d’だけで構成された任意の文字列 (空文字列を含む) にマッチする。つまり‘c[ad]*r’は‘cr’、‘car’、‘cdr’、‘caddaar’等にマッチする。

開始文字と終了文字の間に‘-’を記述することにより文字候補内に文字範囲を含めることができる。つまり‘[a-z]’は小文字の ASCIIアルファベット文字にマッチする。範囲は‘[a-z$%.]’のように個別の文字と自由に組み合わせることができる。これは任意の ASCII小文字アルファベットと‘$’、‘%’、またはピリオドとマッチする。しかし 1 つの範囲の終端文字が別の範囲の開始文字ではないこと。たとえば‘[a-m-z]’は使用しないこと。

文字候補には名前付き文字クラスも指定できる (Section 34.3.1.2 [Char Classes], page 887 を参照)。これは POSIX の機能。たとえば‘[[:ascii:]]’は任意の ASCII 文字にマッチする。文字クラスの使用は、そのクラス内すべての文字を記述するのと等しい。しかし異なる文字数千を含むクラスもあるので後者は実際は実現不可能。文字クラス範囲の上側や下側の境界に出現するべきではない。

文字候補の内部では、通常の regexp スペシャル文字ではスペシャルではない。完全に異なる文字セット‘]’、‘-’、‘^’がスペシャルになる。文字候補に‘]’を含めるには、それを先頭に配置する。‘^’を含めるには、それを先頭以外の場所に配置する。‘-’を含めるには、それを最後に配置する。したがって‘[]^-]’は、これら 3 つのスペシャル文字すべてにマッチする。ここでは‘\’はスペシャルではないので、これら 3 つの文字のエスケープに‘\’は使用できない。

以下の範囲にたいする側面は Emacs 固有であり、POSIX はこの振る舞いを許容はするが必須ではなく、Emacs 以外のプログラムは異なる振る舞いをするかもしれない。

1. case-fold-searchが非 nilなら‘[a-z]’は大文字にもマッチする。
2. 範囲は locale の照合順の影響を受けない。範囲は常にその範囲の境界間に存在するコードポイントを文字セットで表現されるので、たとえ C や POSIX の locale 外部でも‘[a-z]’がマッチするのは ASCII 文字のみ。

3. 範囲の下側境界が上側境界より大きければ範囲は空であり何の文字も表現しない。したがって‘[z-a]’は常にマッチに失敗するし、‘[^z-a]’は改行を含む任意の文字にマッチする。ただし逆転した範囲は typo でないことを明確にするために、常に文字‘z’から文字‘a’にすること。たとえば‘[+-*/]’は意図した 4 つの文字ではなく、‘/’だけにマッチするので避けること。

ある種の文字候補は、たとえそれらが Emacs 内において明確に定義された意味をもっているとしても最良のスタイルとならない。これらには以下が含まれる:

1. ほとんどすべての文字を範囲の境界にできるとはいえ、文字コードテーブルを記憶している人はほとんどいないので、ASCII 文字や数字の自然な順序を守るほうがよいスタイルである。たとえば‘[.-9]’は‘[./0-9]’、‘[`-~]’は‘[`a-z{|}~]’より明確さに劣る。ここでは Unicode の文字エスケープが助けとなる。たとえばほとんどのプログラマーにとっては‘[^^e0^^b8^^81-^^e0^^b8^^ba^^e0^^b8^^bf-^^e0^^b9^^9b]’より‘[\\u0E01-\\u0E3A\\u0E3F-\\u0E5B]’のほうが明確だろう。
2. 文字候補に重複を含めることができたとしても、それを避けるほうがよいスタイルである。たとえば‘[XYa-yYb-zX]’は‘[XYa-z]’より明確さに劣る。
3. 範囲を単に 1 文字、2 文字、あるいは 3 文字で表せたとしても、文字をリストするほうがシンプルである。たとえば‘[a-a0]’は‘[a0]’、‘[i-j]’は‘[ij]’、‘[i-k]’は‘[ijk]’より明確さに劣る。
4. たとえ文字候補の先頭や範囲の上側境界として‘-’を配置できるとしても、文字候補の最後に‘-’そのものを配置するほうがよいスタイルである。たとえば‘[-a-z]’が有効であっても‘[a-z-]’のほうがよいスタイルであり、‘[*--]’が有効だとしても‘[*+,-]’のほうが明確である。

‘[^ ...]’ ‘[^’は補完文字候補 *(complemented character alternative)* を開始する。これは指定された以外の任意の文字とマッチする。つまり‘[^a-z0-9A-Z]’は ASCII 文字と数字以外の、すべての文字にマッチする。

‘^’は文字クラス内では先頭に記述されない限り特別ではない。‘^’に続く文字は、あたかもそれが先頭にあるかのように扱われる (言い換えると‘-’や‘]’はここでは特別ではない)。

マッチしない文字の 1 つとして改行が記述されていなければ、補完文字候補は改行にマッチできる。これは grepのようなプログラム内での regexp の扱いとは対照的である。

文字候補のように名前付き文字クラスを指定できる。たとえば‘[^[:ascii:]]’は任意の非 ASCII文字にマッチする。Section 34.3.1.2 [Char Classes], page 887 を参照のこと。

‘^’ バッファーのマッチングの際には‘^’は空文字列、ただしマッチ対象のテキスト内にある行の先頭 (またはバッファーのアクセス可能範囲の先頭) だけにマッチする。それ以外のマッチはすべて失敗する。つまり‘^foo’は行の先頭に出現する‘foo’にマッチする。

バッファーではなく文字列とマッチする際には、‘^’は文字列の先頭か改行文字の後にマッチする。

歴史的な互換性という理由により‘^’は正規表現の先頭、または‘\(’、‘\(?:’、‘\|’の後でのみ使用できる。

‘$’ これは‘^’と似ているが、行の終端 (またはバッファーのアクセス可能範囲の終端) だけにマッチする。つまり‘x+$’は行末にある 1 つ以上の‘x’からなる文字列にマッチする。

バッファーではなく文字列とマッチする際には、‘$’は文字列の終端か改行文字の前にマッチする。

歴史的な互換性という理由により‘$’は正規表現の先頭、または‘\(’、‘\(?:’、‘\|’の前でのみ使用できる。

‘\’ これはスペシャル文字 (‘\’を含む) のクォートと、追加のスペシャル文字の導入という 2 つの機能をもつ。

‘\’はスペシャル文字をクォートするので‘\$’は‘$’、‘\[’は‘[’だけにマッチする正規表現のようになる。

‘\’は Lisp 文字列 (Section 2.4.8 [String Type], page 19 を参照) の入力構文 (read syntax) 内でも特別な意味をもち、‘\’でクォートしなければならないことに注意。たとえば文字‘\’にマッチする正規表現は‘\\’。文字‘\\’を含む Lisp 文字列を記述するには、別の‘\\’で‘\\’をクォートすることを Lisp 構文は要求する。したがって‘\’にマッチする正規表現にたいする入力構文は"\\\\"となる。

注意してください**:** 歴史的な互換性のために、スペシャル文字はそれらがもつ特別な意味が意味を成さないコンテキスト内にある場合には通常の文字として扱われます。たとえば‘*foo’は‘*’が作用可能な前置された表現がないので、通常の‘*’として扱われます。この挙動に依存するのは悪い習慣です。どこにそれが出現しようとスペシャル文字はすべてクォートしてください。

文字候補内で‘\’は何ら特別ではないので、‘-’や‘]’の特別な意味を取り除くことは決してありません。特別な意味をもたないような場合でも、これらの文字をクォートするべきではありません。バックスラッシュ以外の任意の 1 文字にマッチする‘[^\]’ (Lisp 文字列構文では"[^\\]") 内でのように、これらの文字が特別な意味をもつ箇所では、これらの文字にバックスラッシュを前置することには正当性があるので、何もそれほど明解にはしないでしょう。

実際には正規表現内に出現する‘]’は文字候補に近接しており、それ故そのほとんどがスペシャル文字です。しかしリテラルの‘[’と‘]’の複雑なパターンにたいしてマッチを試みることも時にはあるかもしれません。そのような状況では文字候補を囲う角カッコがどれなのかを判断するために、regexp を最初から注意深く解析することが必要なときもあるかもしれません。たとえば‘[^][]’は補完文字候補‘[^][]’ (角カッコ以外の任意の 1 文字とマッチする) と、その後のリテラルの‘]’により構成されます。

厳密には regexp 先頭の‘[’は特別で、‘]’は特別ではないというのがルールです。これはクォートされていない最初の‘[’で終わり、その後は文字候補になります。(文字クラス開始を除き)‘[’はもはや特別ではありませんが、‘]’は直後にスペシャル文字‘[’があるか、その‘[’の後に‘^’がある場合を除いて特別です。これは文字クラス終了ではない次のスペシャル文字‘]’まで続きます。これは文字候補を終了させて、通常の正規表現の構文をリストアします。クォートされていない‘[’は再び特別となり、‘]’は特別ではなくなります。

34.3.1.2 文字クラス

以下は文字候補内で使用できるクラスと意味のテーブルです。クラス名を囲む‘[’と‘]’の文字は名前の一部なので、これらのクラスを使用する正規表現では 1 つ余分にカッコ (brackets) が必要になります。たとえば 1 つ以上のアルファベットか数字のシーケンスにマッチする正規表現は、‘[:alnum:]+’ではなく‘[[:alnum:]]+’です。

‘[:ascii:]’
これは任意の ASCII文字 (コード 0 – 127) にマッチする。

‘[:alnum:]’
: これは任意の英数字にマッチする。マルチバイト文字では、アルファベット文字か数字であることを示すUnicodeプロパティ‘general-category’ (Section 33.6 [Character Properties], page 857を参照) をもつ文字にマッチする。

‘[:alpha:]’
: これは任意のアルファベットにマッチする。マルチバイト文字では、アルファベット文字であることを示すUnicodeプロパティ‘general-category’ (Section 33.6 [Character Properties], page 857を参照) をもつ文字にマッチする。

‘[:blank:]’
: これはUnicode Technical Standard #18のAnnex Cで定義される水平の空白文字 (horizontal whitespace) にマッチする。特にスペース、タブ、およびスペース区切りであることをUnicodeの‘general-category’プロパティ(Section 33.6 [Character Properties], page 857を参照) が示す他の文字にマッチする。

‘[:cntrl:]’
: これはコードが1から31の範囲にあるすべての文字にマッチする。

‘[:digit:]’
: これは‘0’から‘9’までにマッチする。つまり‘[-+[:digit:]]’は‘+’と‘-’、同様に任意の数にマッチする。

‘[:graph:]’
: これはUnicodeの‘general-category’プロパティで示されるようなグラフィック文字 (空白文字、ASCIIと非ASCIIの制御文字、サロゲートコードポイント、Unicodeで未割り当てのコードポイントを除くすべて) にマッチする (Section 33.6 [Character Properties], page 857を参照)。

‘[:lower:]’
: これはカレントのcaseテーブル (Section 4.10 [Case Tables], page 69を参照) で小文字と判断される文字すべてにマッチする。case-fold-searchが非nilなら大文字にもマッチする。

‘[:multibyte:]’
: これは任意のマルチバイト文字にマッチする (Section 33.1 [Text Representations], page 852を参照)。

‘[:nonascii:]’
: これは非ASCII文字にマッチする。

‘[:print:]’
: これは任意のプリント文字 (空白文字か‘[:graph:]’でマッチされるグラフィック文字のいずれか) にマッチする。

‘[:punct:]’
: これは任意の句読点文字 (punctuation character) にマッチする (現在のところマルチバイト文字では単語構文以外のすべてにマッチする)。

‘[:space:]’
: これは空白文字構文 (Section 35.2.1 [Syntax Class Table], page 907を参照) をもつ任意の文字にマッチする。

‘[:unibyte:]’
これは任意のユニバイト文字 (Section 33.1 [Text Representations], page 852 を参照) にマッチする。

‘[:upper:]’
これはカレントの case テーブル (Section 4.10 [Case Tables], page 69 を参照) で大文字と判断される文字すべてにマッチする。case-fold-searchが非 nilならこれは小文字にもマッチする。

‘[:word:]’
これは単語構文 (Section 35.2.1 [Syntax Class Table], page 907 を参照) をもつ任意の文字にマッチする。

‘[:xdigit:]’
これは 16 進数の数字 ‘0’から ‘9’、‘a’から ‘f’と ‘A’から ‘F’にマッチする。

34.3.1.3 正規表現内のバッククラッシュ構文

ほとんどの場合では、‘\’の後の任意の文字はその文字だけにマッチします。しかし例外もいくつかあります。‘\’で始まる特定のシーケンスには、特別な意味をもつものがあります。以下は特別な ‘\’構成要素のテーブルです。

‘\|’ これは選択肢を指定する。2 つの正規表現 aと b、その間にある ‘\|’により、aか bのいずれかにマッチする表現が形成される。

つまり ‘foo\|bar’は、‘foo’か ‘bar’のいずれかにマッチして他の文字列にはマッチしない。

‘\|’は周囲の適用可能な最大の表現に適用される。‘\|’を取り囲む ‘\(... \)’でグループ化することによりグループ化の効力を制限できる。

複数の ‘\|’の処理するための完全なバックトラッキング互換が必要なら、POSIX 正規表現関数を使用すること (Section 34.5 [POSIX Regexps], page 897 を参照)。

‘\{m\}’ これは前のパターンを正確に m回繰り返す後置演算子。つまり ‘x\{5\}’は文字列 ‘xxxxx’にマッチして、それ以外にはマッチしない。‘c[ad]\{3\}r’は ‘caaar’、‘cdddr’、‘cadar’等にマッチする。

‘\{m,n\}’ これは最小で m回、最大で n回繰り返すより一般的な後置演算子。m省略時の最小は 0、n省略時の最大は存在しない。いずれの形式でも mや nが指定された場合には、$2^{16}-1$ より大きくなることはない。

たとえば‘c[ad]\{1,2\}r’は文字列‘car’、‘cdr’、‘caar’、‘cadr’、‘cdar’、‘cddr’にマッチして、それ以外にはマッチしない。
‘\{0,1\}’や ‘\{,1\}’は ‘?’と同じ。
‘\{0,\}’や ‘\{,\}’は ‘*’と同じ。
‘\{1,\}’は ‘+’と同じ。

‘\(... \)’
これは以下の 3 つの目的を果たす役目をもつグループ化構成要素:

1. 他の操作のために一連の ‘\|’選択肢を囲う。つまり正規表現 ‘\(foo\|bar\)x’は、‘foox’か ‘barx’のいずれかにマッチする。
2. 後置演算子 ‘*’、‘+’、‘?’による複雑な表現を囲う。つまり ‘ba\(na\)*’は ‘ba’、‘bana’、‘banana’、‘bananana’、... 等の任意の数 (0 以上) の文字列 ‘na’にマッチする。

3. ‘*digit*’ (以下参照) による将来の参照にたいして、マッチする部分文字列を記録する。

この最後の目的はカッコによるグループ化というアイデアによるものではない。これは同じ構成要素 ‘\(... \)’にたいする 2 つ目の目的に割当てられた別の機能だが、実際のところ 2 つの意味は衝突しない。しかし稀に衝突が発生することがあり、それが内気 (shy) なグループの導入をもたらした。

‘\(?: ... \)’
これは内気なグループ *(shy group)* の構成要素。内気なグループは通常のグループの最初の 2 つの役目 (他の演算子のネスト制御) を果たすが、これは番号を取得せず ‘*digit*’ でその値を後方参照できない。内気なグループは通常の非内気なグループを変更することなく自動的に追加できるので、機械的に正規表現を構築するのに特に適している。

内気なグループ化は非キャプチャリング *(non-capturing)*、番号なしグループ *(unnumbered groups)* とも呼ばれる。

‘\(?*num*: ... \)’
これは明示的番号付きグループ *(explicitly numbered group)* の構成要素。通常のグループ化では位置をもとに番号が暗黙で取得されるが、これが不便な場合もあるだろう。この構成要素により特定のグループに番号を強制できる。番号の付与に特別な制限はなく、複数のグループに同じ番号を付与でき、その場合は最後の 1 つ (もっとも右のマッチ) がマッチとして採用される。暗黙に番号付けされたグループは、常に前のグループより大きい最小の整数となる番号を取得する。

‘*digit*’ これはグループ構成要素 (‘\(... \)’) の *digit*番目にマッチしたテキストと同じテキストにマッチする。

言い換えると最後のグループの後に、マッチ処理はそのグループによりマッチされたテキストの開始と終了を記憶する。その正規表現の先の箇所で ‘\’とその後に *digit*を使用すれば、それが何であれ同じテキストにマッチさせることができる。

検索やマッチングを行う関数に渡される正規表現全体の中で、最初の 9 つのグループ化構成要素にマッチする文字列には、その正規表現内で開カッコが出現する順に 1 から 9 までの番号が割り当てられる。したがって ‘\1’から ‘\9’までを使用して、対応するグループ化構成要素によりマッチされたテキストを参照できる。

たとえば ‘\(.*\)\1’は、一方がもう一方と等しいような 2 つの文字列から構成される、改行を含まない任意の文字列にマッチする。‘\(.*\)’は前半分にマッチし、これは何でもよいが、それに続く ‘\1’はそれと同じテキストに正確にマッチしなければならない。

構成要素 ‘\(... \)’が 2 回以上マッチする場合 (これはたとえば後に ‘*’をしたがえるとき発生し得る) には最後のマッチだけが記録される。

正規表現内の特定のグループ化構成要素がマッチしなかった場合には、たとえばそれが使用されない選択肢内にあったり、回数が 0 回の繰り返しの内部にあるなら、それに対応する ‘*digit*’構文は何にもマッチしない。作為的な例を用いると ‘\(foo\(b*\)\|lose\)\2’は ‘lose’にマッチできない。外側のグループ内の 2 つ目の選択肢がマッチするが、‘\2’が未定義となり何にたいしてもマッチできない。しかし ‘foobb’にたいしては、1 つ目の選択肢が ‘foob’にマッチして、‘\2’が ‘b’にマッチするのでマッチが可能になる。

‘\w’ これは任意の単語構成文字にマッチする。エディターの構文テーブルが、どの文字が単語構成文字かを決定する。Chapter 35 [Syntax Tables], page 906 を参照のこと。

‘\W’ これは任意の非単語構成文字にマッチする。

‘\s*code*’ これは構文が *code*であるような任意の文字にマッチする。ここで *code*は、構文コードを表す文字。‘w’は単語構成要素、‘-’は空白文字、‘(’は開カッコ、... 等。空白文字構文を表すには、‘-’かスペース文字のいずれかを使用する。構文コードとそれらを意味する文字のリストは Section 35.2.1 [Syntax Class Table], page 907 を参照のこと。

‘\S*code*’ これは構文が *code*でないような任意の文字にマッチする。

‘\c*c*’ これはカテゴリーが *c*であるような任意の文字にマッチする。ここで *c*はカテゴリーを表す文字。つまり標準カテゴリーテーブルで ‘c’は Chinese(中国語)、‘g’は Greek(ギリシャ語) の文字となる。*M-x describe-categories RET*で現在定義済みの全カテゴリーのリストを確認できる。`define-category`関数を使用すれば、標準カテゴリーに加えてカテゴリーを独自に定義することもできる (Section 35.8 [Categories], page 918 を参照)。

‘\C*c*’ これはカテゴリーが *c*ではない任意の文字にマッチする。

以下は空文字列にマッチ (つまり文字を何も消費しない) しますが、マッチするかどうかはコンテキストに依存するような正規表現を構築します。これらすべてにたいして、そのバッファーのアクセス可能範囲の先頭と終端は、あたかもそのバッファーの実際の先頭と終端のように扱われます。

‘\`’ これは空文字列、ただしバッファー先頭またはマッチ対象の文字列の先頭だけにマッチする。

‘\'’ これは空文字列、ただしバッファー終端またはマッチ対象の文字列の終端だけにマッチする。

‘\=’ これは空文字列、ただしポイント位置だけにマッチする (この構成要素はマッチ対象が文字列なら定義されない)。

‘\b’ これは空文字列、ただし単語の先頭だけにマッチする。つまり ‘\bfoo\b’は個別の単語として出現する ‘foo’だけにマッチする。‘\bballs?\b’は、個別の単語として ‘ball’か ‘balls’にマッチする。

‘\b’は、隣接するテキストが何であるかと無関係に、バッファー (か文字列) の先頭または終端にマッチする。

‘\B’ これは空文字列、単語の先頭や終端、またはバッファー (か文字列) の先頭や終端以外にマッチする。

‘\<’ これは空文字列、ただし単語の先頭だけにマッチする。‘\<’は後に単語構成文字が続く場合のみバッファー (か文字列) の先頭にマッチする。

‘\>’ これは空文字列、ただし単語の終端だけにマッチする。‘\<’はコンテンツが単語構成文字で終わる場合のみバッファー (か文字列) の終端にマッチする。

‘_<’ これは空文字列、ただしシンボルの先頭だけにマッチする。シンボルとは 1 つ以上の単語かシンボル構成文字のシーケンス。‘_<’は後にシンボル構成文字が続く場合のみバッファー (か文字列) の先頭にマッチする。

‘_>’ これは空文字列、ただし単語の終端だけにマッチする。‘_>’はコンテンツがシンボル構成文字で終わる場合のみバッファー (か文字列) の終端にマッチする。

すべての文字列が、有効な正規表現な訳ではありません。たとえば終端の ‘]’がない文字選択肢の内側で終わる文字列は無効であり、単一の ‘\’で終わる文字列も同様です。いずれかの検索関数にたいして無効な正規表現が渡されると `invalid-regexp`エラーがシグナルされます。

34.3.2 正規表現の複雑な例

以下は後続の空白文字とともにセンテンスの終わりを認識するために、以前の Emacs で使用されていた複雑な正規表現の例です (現在の Emacs は関数 `sentence-end`により構築される、同様のより複雑な regexp を使用する。Section 34.8 [Standard Regexps], page 905 を参照)。

以下ではまず、(スペースとタブ文字を区別するために)Lisp 構文の文字列として regexp を示して、それを評価した結果を示します。文字列定数の開始と終了はダブルクォーテーションです。'`\"`'は文字列の一部としてのダブルクォーテーション、'`\\`'は文字列の一部としてのバックスラッシュ、'`\t`'はタブ、'`\n`'は改行を意味します。

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\|  \\)[ \t\n]*"
     ⇒ "[.?!][]\"')}]*\\($\\| $\\|  \\|  \\)[
]*"
```

改行とタブは、それら自身として出力されます。

この正規表現は連続する 4 つのパートを含み、以下のように解読できます:

`[.?!]` この正規表現の 1 つ目のパートはピリオド、疑問符、感嘆符の 3 つのうちいずれか 1 つにマッチする文字選択肢。マッチはこれら 3 つの文字のいずれかで開始されなければならない (これは旧正規表現と Emacs が使用する新たなデフォルト regexp が異なる 1 つのポイントである。新たな値は後続の空白文字なしでセンテンスを終端する、いくつかの非 ASCII文字を許容する)。

`[]\"')}]*`
: パターンの 2 つ目のパートは任意の 0 個以上の閉カッコとクォーテーションマークであり、その後にピリオド、疑問符、感嘆符があるかもしれない。`\"`は文字列内でのダブルクォーテーションマークにたいする Lisp 構文。最後の '`*`'は直前の正規表現 (この場合は文字選択肢) の 0 回以上の繰り返しを示す。

`\\($\\| $\\|\t\\|  \\)`
: パターンの 3 つ目のパートはセンテンスの後の空白文字、すなわち行の終端 (スペースがあっても可)、タブ、または 2 つのスペースにマッチする。2 連バックスラッシュはカッコと垂直バーを正規表現構文としてマークする。すなわちカッコはグループを句切り、垂直バーは選択肢を区別する。ダラー記号は行の終端へのマッチに使用される。

`[ \t\n]*` 最後にパターンの最終パートはセンテンスを終端させるために必要とされる以上の、余分な空白文字にマッチする。

34.3.3 正規表現の関数

以下の関数は正規表現を扱います。

`regexp-quote` *string* [Function]

この関数は *string*だけに正確にマッチするような正規表現をリターンする。`looking-at`内でこの正規表現を使用すると、そのバッファー内の次の文字が *string*のときだけ成功するだろう。検索関数でのこの正規表現の使用は、検索されるテキストが *string*を含むなら成功するだろう。Section 34.4 [Regexp Search], page 894 を参照のこと。

これにより、その正規表現を求める関数呼び出し時に正確な文字列マッチや検索を要求できる。

```
(regexp-quote "^The cat$")
     ⇒ "\\^The cat\\$"
```

正規表現として記述されたコンテキストにおいて、正確な文字列マッチを結合することが`regexp-quote`の 1 つの使い方である。たとえば以下は空白文で囲まれた *string*の値であるような文字列を検索する:

```
(re-search-forward
 (concat "\\s-" (regexp-quote string) "\\s-"))
```

スペシャル文字を何も含まなければ、リターンされる文字列は *string*自身となるだろう。

`regexp-opt` *strings* **&optional** *paren* [Function]

この関数はリスト *strings*の文字列だけにマッチする効果的な正規表現をリターンする。これはマッチングや検索を可能な限り高速にする必要があるとき、たとえば Font Lock モードで有用である [1]。

*strings*が空リストなら、リターン値は何にもマッチすることはない regexp となる。

オプション引数 *paren*には以下のいずれかを指定できる:

文字列 結果となる regexp の前に *paren*、後に '`\)`'が付加される。たとえば '`"\\(?1:"`' を使用すると番号付きのグループを明示的に生成する。

`words` 結果となる regexp は '`\<\(`'と '`\)\>`'で括られる。

`symbols` 結果となる regexp は '`\_<\(`'と '`\)\_>`'で括られる (これはプログラミング言語のキーワードの類のマッチング時にしばしば適している)。

非 `nil` 結果となる regexp は '`\\(`'と '`\\)`'で括られる。

`nil` 後に付加する後置演算子が式全体に適用されるために必要なら、結果となる regexp は '`\(?:`'と '`\)`'で括られる。

リターンされる regexp は可能な最長文字列に常にマッチする方法の順序に整列される。

再整列されるまで`regexp-opt`の結果 regexp はその単純化されたバージョンと等価だが、通常はより効果的である。

```
(defun simplified-regexp-opt (strings &optional paren)
 (let ((parens
        (cond
         ((stringp paren)       (cons paren "\\)"))
         ((eq paren 'words)    '("\\<\\(" . "\\)\\>"))
         ((eq paren 'symbols) '("\\_<\\(" . "\\)\\_>"))
         ((null paren)          '("\\(?:" . "\\)"))
         (t                       '("\\(" . "\\)")))))
   (concat (car parens)
           (mapconcat 'regexp-quote strings "\\|")
           (cdr parens))))
```

`regexp-opt-depth` *regexp* [Function]

この関数は *regexp*内のグループ化された構成要素 (カッコで囲まれた正規表現) の総数をリターンする。これには内気なグループは含まれない (Section 34.3.1.3 [Regexp Backslash], page 889 を参照)。

[1] `regexp-opt`の結果が絶対的にもっとも効率的であるという保証はないことに注意してください。手作業でチューニングした正規表現のほうがわずかに効率的なこともありますが、これに努力する価値はほとんどないでしょう。

`regexp-opt-charset` *chars* [Function]

この関数は文字リスト *chars*内の文字にマッチする正規表現をリターンする。

```
(regexp-opt-charset '(?a ?b ?c ?d ?e))
     ⇒ "[a-e]"
```

`regexp-unmatchable` [Variable]

この変数はすべての文字列にマッチしないことが保証された regexp を含む。これは実際に何かとマッチするようにセットされ得る変数用のデフォルト値として特に有用。

34.4 正規表現の検索

GNU Emacs ではインクリメンタルと非インクリメンタルの両方で正規表現 (Section 34.3.1 [Syntax of Regexps], page 884 を参照) にたいする次のマッチを検索できます。インクリメンタル検索コマンドについては Section “Regular Expression Search” in *The GNU Emacs Manual* を参照してください。ここではプログラム内で有用な検索関数だけを説明します。重要な関数は `re-search-forward` です。

これらの検索関数はバッファーがマルチバイトならルチバイト、ユニバイトならユニバイトに正規表現を変換します。Section 33.1 [Text Representations], page 852 を参照してください。

`re-search-forward` *regexp* **&optional** *limit noerror count* [Command]

この関数はカレントバッファー内で、正規表現 *regexp*にマッチするテキスト文字列を前方へ検索する。この関数は *regexp*にマッチしない任意の量のテキストをスキップして、見つかった最初のマッチの終端にポイントを残す。これはポイントの新たな値をリターンする。

引数 *limit*は検索の境界を指定し、それはカレントバッファー内の位置であること。*limit*が非`nil`ならカレントバッファー内の位置でなければならない。これは検索の上限位置を指定するその位置を超えるようなマッチは、受け入れられない。*limit*が省略または`nil`の場合のデフォルトは、そのバッファーのアクセス可能範囲の終端である。

検索失敗時に `re-search-forward`が何を行うかは *noerror*の値に依存する。

`nil` `search-failed`エラーをシグナルする。

`t` 何もせず `nil`をリターンする。

その他 ポイントを *limit* (またはバッファーのアクセス可能範囲の終端) に移動して `nil`をリターンする。

引数 *noerror*はマッチに失敗した有効な検索だけに影響する。無効な引数は *noerror*とは無関係にエラーとなる。

*count*が正の数 *n*なら、それは繰り返し回数の役目をもつ。検索は *n*回繰り返され、前回のマッチの終端から毎回検索が開始される。これらの連続する検索が成功した場合、関数は成功となりポイントを移動して新たな値をリターンする。それ以外は検索失敗となり、上述したように結果は *noerror*の値に依存する。*count*が負の数 $-n$なら、それは逆方向 (後方) への検索の繰り返し回数 *n*としての役目をもつ。

以下の例ではポイントは最初は‘`T`’の前にある。この検索を評価することにより、その行の終端 (‘`hat`’の ‘`t`’と改行の間) にポイントは移動する。

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------
```

```
(re-search-forward "[a-z]+" nil t 5)
     ⇒ 27

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

`re-search-backward` *regexp* **&optional** *limit noerror count* [Command]

この関数はカレントバッファー内で正規表現 *regexp*にマッチするテキスト文字列を後方へ検索して、見つかった最初のマッチの先頭にポイントを残す。

この関数は `re-search-forward`と似ているが単なるミラーイメージ (mirror-image: 鏡像) ではない。`re-search-forward`は先頭が開始ポイントと可能な限り近いマッチを探す。`re-search-backward`が完全なミラーイメージなら終端が可能な限り近いマッチを探すだろう。しかし実際には先頭が可能な限り近い (かつ開始ポイントの前で終わる) マッチを探す。これは与えられた位置にたいする正規表現マッチングが常に正規表現の先頭から終端に機能して、指定された開始位置から開始されることが理由。

`re-search-forward`の真のミラーイメージには、正規表現を終端から先頭へマッチする特別な機能が要求されるだろう。それを実装することによる問題と比較して、値する価値はない。

`string-match` *regexp string* **&optional** *start* [Function]

この関数は *string*内で正規表現 *regexp*にたいする最初のマッチの開始位置のインデックスをリターンする。*string*内のそのインデックスから検索が開始される。

たとえば、

```
(string-match
 "quick" "The quick brown fox jumped quickly.")
     ⇒ 4
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
```

文字列の最初の文字のインデックスは 1、2 文字目は 2、... となる。

この関数がマッチを見つけたら、そのマッチの先の最初の文字のインデックスは `(match-end 0)`で利用できる。Section 34.6 [Match Data], page 897 を参照のこと。

```
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27

(match-end 0)
     ⇒ 32
```

`string-match-p` *regexp string* **&optional** *start* [Function]

この述語関数は `string-match`と同じことを行うが、マッチデータの変更を避ける。

`looking-at` *regexp* [Function]

この関数はカレントバッファー内のポイント直後のテキストが正規表現 *regexp*にマッチするかどうかを判断する。“直後” の正確な意味は、その検索が “固定” されていて、ポイントの後の

最初の文字からマッチが開始する場合のみ成功するということ。成功なら結果は t、それ以外は nil。

この関数はポイントを移動しないがマッチデータは更新する。Section 34.6 [Match Data], page 897 を参照のこと。マッチデータを変更せずにテストする必要があるなら、以下で説明する looking-at-pを使用すること。

以下の例ではポイントは 'T'の直前にある。それ以外の場所にあれば結果は nilになるだろう。

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-at "The cat in the hat$")
     ⇒ t
```

looking-back *regexp limit* **&optional** *greedy* [Function]

この関数はポイントの直前の (ポイントで終わる) テキストが *regexp*とマッチしたら t、それ以外は nilをリターンする。

正規表現マッチングは前方だけに機能するので、ポイントで終わるマッチをポイントから後方へ検索するように実装された。長い距離を検索する必要がある場合には、これは極めて低速になり得る。非 nil値を *limit*を指定してその前を検索しないよう告げることにより、検索に要する時間を制限できる。この場合には、マッチデータは *limit*かその後で始まらなければならない。以下は例:

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-back "read \"" 3)
     ⇒ t
(looking-back "read \"" 4)
     ⇒ nil
```

*greedy*が非 nilなら、この関数は可能な限り後方へマッチを拡張して、前方の 1 文字が *regexp* がマッチの一部とならなければ停止する。マッチが拡張されたときには、マッチ開始位置が *limit* の前にあっても許される。

一般的に looking-backは低速なので、可能な限り使用を避けることを推奨する。この理由により looking-back-pの追加は計画されていない。

looking-at-p *regexp* [Function]

この述語関数は looking-atと同様に機能するがマッチデータを更新しない。

search-spaces-regexp [Variable]

この変数が非 nilなら、それは空白文字を検索する方法を告げる正規表現であること。この場合には検索される正規表現内のすべてのスペース属は、この正規表現を使用することを意味する。しかし '[...]'、'*''+'、'?'のような構文要素内のスペースは search-spaces-regexp の影響を受けない。

この変数はすべての正規表現検索とマッチ構文要素に影響するので、コードの可能な限り狭い範囲にたいして一時的にバインドすること。

34.5 POSIX 正規表現の検索

通常の正規表現関数は、'\|'や繰り返しの構文要素を処理するために必要なときだけバックトラッキングを行いますが、何らかのマッチが見つかるまでの間だけこれを継続します。そして成功した後に見つかった最初のマッチを報告します。

このセクションでは正規表現にたいしてPOSIX標準で指定された完全なバックトラッキングを処理する他の検索関数を説明します。これらはPOSIXが要求する最長マッチを報告できるようにすべての可能なマッチを試みて、すべてのマッチが見つかるまでバックトラッキングを継続します。これは非常に低速なので、本当に最長マッチが必要なときだけこれらの関数を使用してください。

POSIXの検索とマッチ関数は、非欲張りな繰り返し演算子 (Section 34.3.1.1 [Regexp Special], page 884 を参照) を正しくサポートしません。これはPOSIXのバックトラッキングが非欲張りな繰り返しのセマンチックと競合するからです。

`posix-search-forward` *regexp* **&optional** *limit noerror count* [Command]
: これは`re-search-forward`と似ているが、正規表現マッチングにたいしてPOSIX標準が指定する完全なバックトラッキングを行う点が異なる。

`posix-search-backward` *regexp* **&optional** *limit noerror count* [Command]
: これは`re-search-backward`と似ているが、正規表現マッチングにたいしてPOSIX標準が指定する完全なバックトラッキングを行う点が異なる。

`posix-looking-at` *regexp* [Function]
: これは`looking-at`と似ているが、正規表現マッチングにたいしてPOSIX標準が指定する完全なバックトラッキングを行う点が異なる。

`posix-string-match` *regexp string* **&optional** *start* [Function]
: これは`string-match`と似ているが、正規表現にたいしてPOSIX標準が指定する完全なバックトラッキングを行う点が異なる。

34.6 マッチデータ

Emacsは検索の間に見つかったテキスト片の開始と終了の位置を追跡します。これはマッチデータ *(match data)* と呼ばれます。このマッチデータのおかげで、メールメッセージ内のデータのような複雑なパターンを検索した後に、そのパターンの制御下でマッチ部分を抽出できるのです。

マッチデータには通常はもっとも最近の検索だけが記述されるので、後で参照したい検索とそのマッチデータの使用の間に誤って別の検索を行わないように注意しなければなりません。誤って別の検索を避けるのが不可能な場合には、マッチデータの上書きを防ぐために前後でマッチデータの保存とリストアを行わなければなりません。

上書きを行わないと明記されていない限り、すべての関数は上書きを許されていることに注意してください。結果としてバックグラウンド (Section 40.11 [Timers], page 1114 と Section 40.12 [Idle Timers], page 1116 を参照) で暗黙に実行される関数は、おそらく明示的にマッチデータの保存とリストアを行うべきでしょう。

34.6.1 マッチしたテキストの置換

以下の関数は、最後の検索でマッチされたテキストのすべて、または一部を置換します。これはマッチデータにより機能します。

`replace-match` *replacement* **&optional** *fixedcase literal string subexp* [Function]

この関数はバッファーや文字列にたいして置換処理を行う。

あるバッファーで最後の検索を行った場合には、*string*引数を省略または `nil`を指定すること。また最後に検索を行ったバッファーがカレントバッファーであることを確認すること。その場合には、この関数はマッチしたテキストを *replacement*で置換することにより、そのバッファーを編集する。これは置換したテキスト終端にポイントを残す。

文字列にたいして最後の検索を行った場合には、同じ文字列が *string*に渡される。その場合には、この関数はマッチしたテキストが *replacement*に置き換えられた新たなテキストをリターンする。

*fixedcase*が非 `nil`なら `replace-match`は大文字小文字を変更せずに置換テキストを使用して、それ以外は置換されるテキストが capitalize(先頭が大文字) されているかどうかに応じて、置換テキストを変換する。元のテキストがすべて大文字なら置換テキストを大文字に変換する。元のテキストの単語すべてが capitalize されていたら置換テキストのすべての単語を capitalize する。すべての単語が 1 文字かつ大文字なら、それらはすべて大文字の単語ではなく capitalize された単語として扱われる。

*literal*が非 `nil`なら *replacement*はそのまま挿入されるが、必要に応じて case の変更だけが行われる。これが `nil`(デフォルト) なら文字 '`\`'は特別に扱われる。*replacement*内に '`\`'が出現した場合には、それは以下のシーケンスのいずれかの一部でなければならない:

'`\&`' これは置換されるテキスト全体を意味する。

'`\`*n*' (*n*は数字)
これは元の regexp の *n*番目の部分式にマッチするテキストを意味する。この部分式とは '`\(...\)`'の内部にグループかされた式のこと。*n*番目のマッチがなければ空文字列が代用される。

'`\\`' これは置換テキスト内で単一の '`\`'を意味する。

'`\?`' これはそれ自身を意味する (`replace-regexp`と関連するコマンドの互換用。Section "Regexp Replace" in *The GNU Emacs Manual* を参照)。

これら以外の '`\`'に続く文字はエラーをシグナルする。

'`\&`'や '`\n`'により行われる代替えは、もしあれば case 変換の後に発生する。したがって代替えする文字列は決して case 変換されない。

*subexp*が非 `nil`なら、それは全体のマッチではなくマッチされた regexp の部分式番号 *subexp* だけを置換することを指定する。たとえば '`foo \(ba*r\)`'のマッチング後に `replace-match` を呼び出すと、*subexp*が 1 なら '`\(ba*r\)`'にマッチしたテキストだけを置換することを意味する。

`match-substitute-replacement` *replacement* **&optional** *fixedcase literal string subexp* [Function]

この関数は `replace-match`によりバッファーに挿入されるであろうテキストをリターンするがバッファーを変更しない。これは '`\n`'や '`\&`'のような構文要素をマッチしたグループで置き換えた実際の結果をユーザーに示したいとき有用。引数 *replacement*、およびオプションの *fixedcase*、*literal*、*string*、*subexp*は `replace-match`のときと同じ意味をもつ。

34.6.2 単純なマッチデータへのアクセス

このセクションでは最後の検索やマッチング操作で、それが成功した場合に何がマッチされたのかを調べるために、マッチデータを使用する方法について説明します。

マッチしたテキスト全体または正規表現のカッコで括られた特定の部分式にたいして問い合わせることができます。以下の関数では、*count*によりどの部分式かを指定できます。*count*が 0 ならマッチ全体、*count*が正なら望む部分式を指定します。

正規表現での部分式とは、エスケープされたカッコ '`\(...\)`'でグループ化された表現だったことを思い出してください。*count*番目の部分式は正規表現全体の先頭から '`\(`'を数えることで見つけられます。最初の部分式が 1、2 つ目が 2、... となります。正規表現だけが部分式をもつことができ、単純な文字列検索の後で利用できるのはマッチ全体の情報だけです。

成功したすべての検索はマッチデータをセットします。したがって検索後は別の検索を行うかもしれない関数を呼び出す前に、検索の直後にマッチデータを問い合わせるべきです。別の検索を呼び出すかもしれない関数の前後で、かわりにマッチデータの保存とリストアすることもできます (Section 34.6.4 [Saving Match Data], page 901 を参照)。または `string-match-p`のようなマッチデータを変更しないと明示されている関数を使用してください。

検索が成功しようと失敗しようとマッチデータは変更されます。現在はこのように実装されていますが、これは将来変更されるかもしれません。失敗した後のマッチデータを信用しないでください。

`match-string` *count* **&optional** *in-string* [Function]

この関数は最後の検索やマッチ処理でマッチしたテキストを文字列としてリターンする。これは *count*が 0 ならテキスト全体、*count*が正なら *count*番目のカッコで括られた部分式に対応する部分だけをリターンする。

そのような最後の処理が文字列にたいする `string-match`呼び出しなら、引数 *in-string*には同じ文字列を渡すこと。バッファーの検索やマッチの後は、*in-string*を省略するか `nil`を渡すこと。しかし最後に検索やマッチを行ったバッファーが、`match-string`呼び出し時にカレントバッファーであることを確認すること。このアドバイスにしたがわなければ誤った結果となるだろう。

*count*が範囲外、'`\|`'選択肢内部の部分式が使用されない、または 0 回の繰り返しなら値は `nil`。

`match-string-no-properties` *count* **&optional** *in-string* [Function]

この関数は `match-string`と似ているが結果がテキストプロパティをもたない点が異なる。

`match-beginning` *count* [Function]

最後の正規表現検索がマッチを見つけたら、この関数はマッチしたテキストか部分式の開始位置をリターンする。

*count*が 0 なら値はマッチ全体の開始位置。それ以外なら *count*は正規表現内の部分式を指定するので、この関数の値はその部分式にたいするマッチの開始位置。

使用されない、あるいは 0 回の繰り返しであるような '`\|`'選択肢内部の部分式にたいしての値は `nil`。

`match-end` *count* [Function]

この関数は `match-beginning`と似ているがマッチの開始ではなく終了位置である点が異なる。

以下はマッチデータを使用する例です。コメントの数字はテキスト内での位置を示しています:

```
(string-match "\\(qu\\)\\(ick\\)"
              "The quick fox jumped quickly.")
              ;0123456789
    ⇒ 4
```

```
(match-string 0 "The quick fox jumped quickly.")
     ⇒ "quick"
(match-string 1 "The quick fox jumped quickly.")
     ⇒ "qu"
(match-string 2 "The quick fox jumped quickly.")
     ⇒ "ick"

(match-beginning 1)       ; 'qu'にたいするマッチ先頭の
     ⇒ 4                  ;   インデックスは 4

(match-beginning 2)       ; 'ick'にたいするマッチ先頭の
     ⇒ 6                  ;   インデックスは 6

(match-end 1)             ; 'qu'にたいするマッチ終端の
     ⇒ 6                  ;   インデックスは 6

(match-end 2)             ; 'ick'にたいするマッチ終端の
     ⇒ 9                  ;   インデックスは 9
```

別の例を以下に示します。ポイントは最初は行の先頭にあります。検索の後はポイントはスペースと単語 'in'の間にあります。マッチ全体の先頭はバッファーの 9 つ目の文字 'T'、1 つ目の部分式にたいするマッチの先頭は 13 番目の文字 'c'です。

```
(list
  (re-search-forward "The \\(cat \\)")
  (match-beginning 0)
  (match-beginning 1))
    ⇒ (17 9 13)

---------- Buffer: foo ----------
I read "The cat ⋆in the hat comes back" twice.
        ^   ^
        9  13
---------- Buffer: foo ----------
```

(この場合にはリターンされるインデックスはバッファー位置であり、バッファーの 1 つ目の文字を 1 と数える。)

34.6.3 マッチデータ全体へのアクセス

関数 match-dataと set-match-dataは、マッチデータ全体にたいして一度に読み取り、または書き込みを行います。

match-data **&optional** *integers reuse reseat* [Function]

この関数は最後の検索によりマッチしたテキストのすべての情報を記録する位置 (マーカーか整数) をリターンする。要素 0 は正規表現全体にたいするマッチの先頭の位置。要素 1 はその正規表現にたいするマッチの終端の位置。次の 2 つの要素は 1 つ目の部分式にたいするマッチの先頭と終了、... となる。一般的に要素番号 $2n$ は (match-beginning *n*)、要素番号 $2n + 1$ は (match-end *n*)に対応する。

すべての要素は通常はマーカーか nilだが、もし *integers*が非 nilならマーカーのかわりに整数を使用することを意味する (この場合にはマッチデータの完全なリストアを容易にするために、リストの最後の要素としてバッファー自身が追加される)。最後の検索が string-matchにより文字列にたいして行われた場合には、マーカーは文字列の内部をポイントできないので常に整数が使用される。

*reuse*が非 nilなら、それはリストであること。この場合には、match-dataはマッチデータを*reuse*内に格納する。つまり *reuse*は破壊的に変更される。*reuse*が正しい長さである必要はない。特定のマッチデータにたいして長さが十分でなければリストは拡張される。*reuse*が長過ぎる場合には、長さはそのままで使用しない要素に nilがセットされる。この機能にはガベージコレクションの必要頻度を減らす目的がある。

*reseat*が非 nilなら、*reuse*リスト内のすべてのマーカーは存在しない場所を指すよう再設定される。

他の場合と同じように検索関数とその検索のマッチデータへのアクセスを意図する match-data呼び出しの間に介入するような検索があってはならない。

```
(match-data)
     ⇒  (#<marker at 9 in foo>
         #<marker at 17 in foo>
         #<marker at 13 in foo>
         #<marker at 17 in foo>)
```

set-match-data *match-list* **&optional** *reseat* [Function]

この関数は *match-list*の要素からマッチデータをセットする。*match-list*は前の match-data呼び出しの値であるようなリストであること (正確には同じフォーマットなら他のものでも機能するだろう)。

*match-list*が存在しないバッファーを参照する場合でもエラーとはならない。これは無意味だが害のない方法でマッチデータをセットする。

*reseat*が非 nilなら、リスト *match-list*内のすべてのマーカーは存在しない場所を指すよう再設定される。

store-match-dataは set-match-dataの半ば時代遅れなエイリアス。

34.6.4 マッチデータの保存とリストア

以前に行った検索にたいするマッチデータを後で使用するために保護する必要があるなら、検索を行うかもしれない関数の呼び出し時に呼び出しの前後でマッチデータの保存とリストアを行う必要があるでしょう。以下はマッチデータ保存に失敗した場合に発生する問題を示す例です:

```
(re-search-forward "The \\(cat \\)")
     ⇒ 48
(foo)                   ; fooが他の検索を行うと
(match-end 0)
     ⇒ 61               ; 結果は期待する 48 と異なる!
```

save-match-dataでマッチデータの保存とリストアができます:

save-match-data *body...* [Macro]

このマクロは *body*を実行して、その前後のマッチデータの保存とリストアを行う。リターン値は *body*内の最後のフォームの値。

`set-match-data`と`match-data`を一緒に使用して、`save-match-data`の効果を模倣することができます。以下はその方法です:

```
(let ((data (match-data)))
  (unwind-protect
      ...   ; 元のマッチデータを変更しても OK
    (set-match-data data)))
```

プロセスフィルター関数 (Section 38.9.2 [Filter Functions], page 955 を参照)、およびプロセスセンチネル (Section 38.10 [Sentinels], page 959 を参照) の実行時には、Emacs が自動的にマッチデータの保存とリストアを行います。

34.7 検索と置換

バッファーのある部分で regexp にたいするすべてのマッチを見つけてそれらを置換したい場合には、以下のように`re-search-forward`と`replace-match`を使用して明示的なループを記述するのが最良の方法です:

```
(while (re-search-forward "foo[ \t]+bar" nil t)
  (replace-match "foobar"))
```

`replace-match`の説明は Section 34.6.1 [Replacing the Text that Matched], page 897 を参照してください。

しかし文字列内のマッチの置換、特に置換を効果的に行いたい場合には、より複雑になります。そのために Emacs はこれを行うための関数を提供します。

`replace-regexp-in-string` *regexp rep string* **&optional** *fixedcase literal subexp start* [Function]

この関数は *string*をコピーして *regexp*にたいするマッチを検索、それらを *rep*に置き換える。これは変更されたコピーをリターンする。*start*が非`nil`ならマッチにたいする検索は *string*内のそのインデックスから開始されて、リターン値には *string*の最初の *start*文字は含まれない。置き換え後の文字列全体を取得するには、*string*の最初の *start*文字とリターン値を結合すること。

この関数は置換を行うためにオプション引数 *fixedcase*、*literal*、*subexp*を渡して`replace-match`を使用する。

*rep*は文字列のかわりに関数でもよい。この場合には`replace-regexp-in-string`はそれぞれのマッチにたいして、そのテキストを単一の引数として *rep*を呼び出す。これは *rep*がリターンする値を収集して、それを置換文字列として`replace-match`に渡す。この時点でのマッチデータは *string*の部分文字列にたいする *regexp*のマッチ結果。

`query-replace`の行に関するコマンドを記述したい場合には、`perform-replace`を使用してこれを行うことができます。

`perform-replace` *from-string replacements query-flag regexp-flag delimited-flag* **&optional** *repeat-count map start end backward region-noncontiguous-p* [Function]

これは`query-replace`および関連するコマンドの根幹となる関数である。これは位置 *start*と *end*の間にあるテキスト内に出現する *from-string*の一部またはすべてを置換する。*start*が`nil` (または省略) ならかわりにポイントを、*end*にはそのバッファーのアクセス可能範囲の終端が使用される (オピション引数 *backward*が非`nil`なら検索は *end*から後方に開始される)。

*query-flag*が nilならすべてのマッチを置換する。それ以外なら、それぞれにたいしてユーザーにたいして何をすべきか問い合わせる。

*regexp-flag*が非 nilなら *from-string*は正規表現、それ以外はリテラルとしてマッチしなければならない。*delimited-flag*が非 nilなら単語境界に囲まれた置換だけが考慮される。

引数 *replacements*はマッチを何で置き換えるかを指定する。文字列ならその文字列を使用する。サイクル順に使用される文字列リストでもよい。

*replacements*がコンスセル (*function* . *data*) なら、置換テキストを取得するためにそれぞれのマッチ後に *function*を呼び出すことを意味する。この関数は *data*とすでに置換された個数という、2 つの引数で呼び出される。

*repeat-count*が非 nilなら、それは整数であること。その場合にはサイクルを次に進める前に、*replacements*リスト内の各文字列を何度使用するかを指定する。

*from-string*が大文字アルファベットを含む場合には、perform-replaceは case-fold-searchを nilにバインドして大文字小文字を変換せずに *replacements*を使用する。

キーマップ query-replace-mapは通常は問い合わせにたいして可能なユーザー応答を定義する。引数 *map*が非 nilなら、それは query-replace-mapのかわりに使用するキーマップを指定する。

*region-noncontiguous-p*が非 nilなら、*start*と *end*の間のリージョンは非連続部分から構成されることを意味する。これのもっとも一般的な例は部分が開業文字で区切られた矩形リージョンである。

この関数は *from-string*の次のマッチを検索するために 2 つの関数のうちいずれか 1 つを使用する。これらの関数は 2 つの変数 replace-re-search-functionと replace-search-functionにより指定される。引数 *regexp-flag*が非 nilなら前者、nilなら後者が呼び出される。

query-replace-map [Variable]

この変数は perform-replaceにたいする有効なユーザー応答を定義するスペシャルキーマップを保持して、コマンドは y-or-n-pや map-y-or-n-pと同様にそれを使用する。このマップは 2 つの点において普通のマップと異なる。

- キーバインディングはコマンドではなく、このマップを使用する関数にとって意味のある単なるシンボルであること。
- プレフィクスキーはサポートされない。各キーバインディングは単一イベントキーシーケンスでなければならない。この関数は入力を取得するために単一イベントを読み取って、それを “手動” で照合するので read-key-sequenceを使用しないからである。

query-replace-mapにたいして意味をもつバインディングがあります。それらのうちいくつかは query-replaceとその同族にたいしてのみ意味をもちます。

act 判断している対象にたいしてアクションを起こす (言い換えると “yes”)。

skip この問いにたいしてアクションを起こさない (言い換えると “no”)。

exit この問いにたいして “no” を答えて、さらに一連の問いすべてにたいして “no” が応答されたとみなして問い合わせをあきらめる。

exit-prefix

exitと似ているが、unread-command-eventsにたいして押下されたキーを追加する (Section 21.8.6 [Event Input Misc], page 410 を参照)。

`act-and-exit`
: この問いにたいして “yes” を答えて、さらに一連の問いすべてにたいして後続の問いに “no” が応答されるとみなして問い合わせをあきらめる。

`act-and-show`
: この問いに “yes” を答えるが、結果を表示してまだ次の問いへ進まない。

`automatic`
: これ以上のユーザーとの対話を行わず、この問いと後続の問いにたいして “yes” を答える。

`backup` 前に問い合わせた以前の場所に戻る。

`undo` 最後の置換をアンドゥして置換が行われた位置に戻る。

`undo-all` すべての置換をアンドゥして最初に置換が行われた位置に戻る。

`edit` この問いに対処するために、通常とられるアクションのかわりに再帰編集にエンターする。

`edit-replacement`
: ミニバッファー内で、この問いにたいする置換を編集する。

`delete-and-edit`
: 検討中のテキストを削除して、それを置換するために再帰編集にエンターする。

`recenter`
`scroll-up`
`scroll-down`
`scroll-other-window`
`scroll-other-window-down`
: 指定されたウィンドウスクロール操作を行って同じ問いを再度尋ねる。この問いには `y-or-n-p`と関連する関数だけが使用される。

`quit` 即座に quit を行う。この問いには `y-or-n-p`と関連する関数だけが使用される。

`help` ヘルプを表示して再度尋ねる。

`multi-query-replace-map` [Variable]

この変数はマルチバッファー置換で有用な追加キーバインディングを提供することにより `query-replace-map`を拡張するキーマップを保持する。追加されるバインディングは以下のとおり:

`automatic-all`
: 残りすべてのバッファーにたいして、それ以上の対話をせずその問いと後続のすべての問いに “yes” を答える。

`exit-current`
: この問いに “no” を答えてカレントバッファーにたいする一連の問いすべてをあきらめる。そしてシーケンス内の次のバッファーへ問いを継続する。

`replace-search-function` [Variable]

この変数は置換する次の文字列を検索するために `perform-replace`が呼び出す関数を指定する。デフォルト値は `search-forward`。それ以外の値の場合には `search-forward`の最初の 3 つの引数を引数とする関数を指定すること (Section 34.1 [String Search], page 881 を参照)。

`replace-re-search-function` [Variable]
この変数は置換する次の regexp を検索するために `perform-replace`が呼び出す関数を指定する。デフォルト値は `re-search-forward`。それ以外の値の場合には `re-search-forward`の最初の 3 つの引数を引数とする関数を指定すること (Section 34.4 [Regexp Search], page 894 を参照)。

34.8 編集で使用される標準的な正規表現

このセクションでは、編集において特定の目的のために使用される正規表現を保持するいくつかの変数を説明します。

`page-delimiter` [User Option]
これはページを分割する行開始を記述する正規表現。デフォルト値は`"^\014"` (`"^^L"`または `"^\C-l"`)。これはフォームフィード文字 (改頁文字) で始まる行とマッチする。

以下の 2 つの正規表現が、常に行頭からマッチが始まる正規表現とみなすべきではありません。これらを '`^`'にマッチするアンカーとして使用するべきではありません。ほとんどの場合では、パラグラフコマンドは行頭にたいしてのみマッチのチェックを行うので、これは '`^`'が不要であることを意味します。非 0 の左マージンが存在する場合には、これらは左マージンの後から始まるマッチに適用されます。その場合には、'`^`'は不適切でしょう。しかし左マージンを決して使用しないモードでは '`^`'は無害でしょう。

`paragraph-separate` [User Option]
これはパラグラフを分割する行の開始を認識する正規表現 (これを変更する場合は `paragraph-start`も変更する必要があるかもしれない)。デフォルト値は`"[ \t\f]*$"`であり、これは (左マージン以降) すべてがスペース、タブ、フォームフィードで構成される行とマッチする。

`paragraph-start` [User Option]
これはパラグラフを開始または分割する行の開始を認識する正規表現。デフォルト値は `"\f\\|[ \t]*$"`であり、これは (左マージン以降) すべてが空白文字で構成される行やフォームフィードで始まる行とマッチする。

`sentence-end` [User Option]
非 `nil`なら、以降に続く空白文字を含めてセンテンスの終わりを記述する正規表現であること (これとは無関係にパラグラフ境界もセンテンスを終了させる)。

値が `nil` (デフォルト) なら、関数 `sentence-end`が regexp を構築する。センテンス終端の認識に使用する regexp を得るために常に関数 `sentence-end`を使用するべきなのはこれが理由。

`sentence-end` [Function]
この関数は変数 `sentence-end`が非 `nil`ならその値をリターンする。それ以外なら変数 `sentence-end-double-space` ([Definition of sentence-end-double-space], page 799 を参照)、`sentence-end-without-period`、`sentence-end-without-space`にもとづくデフォルト値をリターンする。

35 構文テーブル

構文テーブル *(syntax table)* はバッファー内のそれぞれの文字にたいして構文的な役割を指定します。単語、シンボル、その他の構文要素の開始と終了の判定にこれを使用できます。この情報は Font Lock モード (Section 23.6 [Font Lock Mode], page 496 を参照) や、種々の複雑な移動コマンド (Section 30.2 [Motion], page 756 を参照) を含む多くの Emacs 機能により使用されます。

35.1 構文テーブルの概念

構文テーブルは、それぞれの文字の構文クラス *(syntax class)* やその他の構文的プロパティを照合するために使用できるデータ構造です。構文テーブルはテキストを横断したスキャンや移動のために Lisp プログラムから使用されます。

構文テーブルは内部的には文字テーブルです (Section 6.6 [Char-Tables], page 110 を参照)。インデックス *c*の要素はコード *c*の文字を記述します。値は該当する文字の構文を指定するコンスセルです。詳細は Section 35.7 [Syntax Table Internals], page 917 を参照してください。しかし構文テーブルの内容を変更や確認するために `aset`や `aref`を使用するかわりに、通常は高レベルな関数 `char-syntax`や `modify-syntax-entry`を使用するべきです。これらについては Section 35.3 [Syntax Table Functions], page 910 で説明します。

`syntax-table-p` *object* [Function]
この関数は *object*が構文テーブルなら `t`をリターンする。

バッファーはそれぞれ自身のメジャーモードをもち、それぞれのメジャーモードはさまざまな文字の構文クラスにたいして独自の考えをもっています。たとえば Lis モードでは文字 '`;`'はコメントの開始ですが、C モードでは命令文の終端になります。これらのバリエーションをサポートするために、構文テーブルはそれぞれのバッファーにたいしてローカルです。一般的に各メジャーモードは自身の構文テーブルをもち、そのモードを使用するすべてのバッファーにそれがインストールされます。たとえば変数 `emacs-lisp-mode-syntax-table`は Emacs の Lisp モードが使用する構文テーブル、`c-mode-syntax-table`は C モードが使用する構文テーブルを保持します。あるメジャーモードの構文テーブルを変更すると、そのモードのバッファー、およびその後でそのモードに置かれるすべてのバッファーの構文も同様に変更されます。複数の類似するモードが 1 つの構文テーブルを共有することがときおりあります。構文テーブルをセットアップする方法の例は Section 23.2.9 [Example Major Modes], page 478 を参照してください。

別の構文テーブルから構文テールを継承 *(inherit)* できます。これを親構文テーブル *(parent syntax table)* と呼びます。構文テーブルは、ある文字にたいして構文クラス "inherit" を与えることにより、構文クラスを未指定にしておくことができます。そのような文字は親構文テーブルが指定する構文クラスを取得します (Section 35.2.1 [Syntax Class Table], page 907 を参照)。Emacs は標準構文テーブル *(standard syntax table)* を定義します。これはデフォルトとなる親構文テーブルであり、Fundamental モードが使用する構文テーブルでもあります。

`standard-syntax-table` [Function]
この関数は標準構文テーブルをリターンする。これは Fundamental モードが使用する構文テーブルである。

Emacs Lisp リーダーは変更不可な独自のビルトイン構文ルールをもつので、構文テーブルは使用しません (いくつかの Lisp システムはリード構文を再定義する手段を提供するが、わたしたちは単純化のためこの機能を Emacs Lisp 外部に留める決定をした)。

35.2 構文記述子

構文クラス *(syntax class)* の文字は、その文字の構文的な役割を記述します。各構文テーブルは、それぞれの文字の構文クラスを指定します。ある構文テーブルでの文字のクラスと、別のテーブルにおけるその文字のクラスとの間に関連性がある必要はありません。

構文テーブルはそれぞれニーモニック文字 (mnemonic character) により選別され、クラスを指定する必要がある際にはそのクラスの名前としての役割を果たします。この指定子文字 (designator character) は通常はそのクラスに割当てられることが多々あります。しかしその指定子としての意味は不変であり、その文字がカレントでもつ構文とは独立しています。つまりカレント構文テーブルにおいて実際に文字 '\'が構文をもつかどうかに関係なく、指定子文字としての '\'は常にエスケープ文字 (escape character) を意味します。

構文記述子 *(syntax descriptor)* とは文字の構文クラスと、その他の構文的なプロパティを記述する Lisp 文字列です。ある文字の構文を変更したい際には、関数 `modify-syntax-entry`を呼び出して引数に構文記述子を渡すことにより行います (Section 35.3 [Syntax Table Functions], page 910 を参照)。

構文記述子の 1 つ目の文字は構文クラスの指定子文字でなければなりません。2 つ目の文字がもしあれば、マッチング文字を指定します (Lisp では '('にたいするマッチング文字は ')')。スペースはマッチング文字が存在しないことを指定します。その後に続く文字は追加の構文プロパティを指定します (Section 35.2.2 [Syntax Flags], page 909 を参照)。

マッチング文字やフラグが必要なければ、(構文クラスを指定する)1 つの文字だけで十分です。

たとえば C モードでの文字 '*'の構文記述子は`". 23"` (区切り記号、マッチング文字用スロットは未使用、コメント開始記号の 2 つ目の文字、コメント終了記号の 1 つ目の文字)、'/'にたいするエントリーは '`. 14`' (区切り記号、マッチング文字用スロットは未使用、コメント開始記号の 1 つ目の文字、コメント終了記号の 2 つ目の文字) です。

Emacs は低レベルでの構文クラスを記述するために使用される *raw* 構文記述子 *(raw syntax descriptors)* も定義しています。Section 35.7 [Syntax Table Internals], page 917 を参照してください。

35.2.1 構文クラスのテーブル

以下は構文クラス、それらの指定子となる文字と意味、および使用例を示すテーブルです。

空白文字: ' 'か '`-`'
: シンボルや単語を区別する文字。空白文字は通常は他の構文的な意義をもたず、複数の空白文字は構文的には単一の空白文字と等しい。スペース、タブ、フォームフィードは、ほとんどすべてのメジャーモードにおいて空白文字にクラス分けされる。

  この構文クラスは ' ' か '`-`'により指定できる。両指定子は等価。

単語構成文字: '`w`'
: 人間の言語における単語の一部。これらは通常はプログラム内において変数やコマンドの名前として使用される。すべての大文字と小文字、および数字は通常は単語構成文字。

シンボル構成文字: '`_`'
: 単語構成文字とともに変数やコマンドの名前で使用される追加の文字。例としては Lisp モードの文字 '`$&*+-_<>`'が含まれ、これらはたとえ英単語の一部ではないとしてもシンボルの名前の一部となり得る。標準 C ではシンボル内において非単語構成文字で有効な文字はアンダースコア ('`_`') のみ。

区切り文字: ‘.’

人間の言語において句読点として使用される文字、またはプログラミング言語でシンボルを別のシンボルと区別するために使用される文字。Emacs Lisp モードのようないくつかのプログラミング言語のモードでは、単語構成文字およびシンボル構成文字のいずれでもないいくつかの文字はすべて他の用途をもつので、このクラスの文字をもたない。C モードのような他のプログラミング言語のモードでは演算子にたいして区切り文字構文が使用される。

開カッコ文字: ‘(’
閉カッコ文字: ‘)’

文や式を囲うために異なるペアとして使用される文字。そのようなグループ化は開カッコで開始され、閉カッコで終了する。開カッコ文字はそれぞれ特定の閉カッコ文字にマッチして、その逆も成り立つ。Emacs は通常は閉カッコ挿入時にマッチする開カッコを示す。Section 39.21 [Blinking], page 1077 を参照のこと。

人間の言語や C のコードでは、カッコのペアは ‘()’、‘[]’、‘{}’。Emacs Lisp ではリストとベクターにたいする区切り文字 (‘()’と ‘[]’) はカッコ文字としてクラス分けされる。

文字列クォート: ‘"’

文字列定数を区切るために使用される文字。文字列の先頭と終端に同じ文字列クォート文字が出現する。このようなクォート文字列はネストされない。

Emacs のパース機能は文字列を単一のトークンとみなす。文字列内ではその文字の通常の構文的な意味は抑制される。

Lisp モードはダブルクォーテーション (‘"’) と垂直バー (‘|’) とう 2 つの文字列クォート文字をもつ。Emacs Lisp では ‘|’は使用しないが Common Lisp では使用される。C も文字列にたいするダブルクォート文字、および文字定数にたいするシングルアポストロフィ (‘'’) という 2 つのクォート文字をもつ。

人間用のテキストには文字列クォート文字がない。そのクォーテーション内の別の文字の通常の構文的プロパティを、クォーテーションマークがオフに切り替えることを、わたしたちは望まない。

エスケープ構文文字: ‘\’

文字列や文字定数内で使用されるようなエスケープシーケンスで始まる文字。C と Lisp の両方で文字 ‘\’はこのクラスに属する (C では文字列内でのみ使用されるが、C コード中を通じてこのように扱っても問題ないことがわかった)。

`words-include-escapes`が非 `nil`なら、このクラスの文字は単語の一部とみなされる。Section 30.2.2 [Word Motion], page 757 を参照のこと。

文字クォート: ‘/’

その文字の通常の構文的な意義を失うように、後続の文字をクォートするために使用される文字。これは直後に続く文字だけに影響する点がエスケープ文字と異なる。

`words-include-escapes`が非 `nil`なら、このクラスの文字は単語の一部とみなされる。Section 30.2.2 [Word Motion], page 757 を参照のこと。

このクラスは TEX モードのバックスラッシュにたいして使用される。

区切りペアー: ‘$’

文字列クォート文字と似ているが、この区切りの間にある文字の構文的なプロパティは抑制されない点が異なる。現在のところ TEX モードだけが区切りペアを使用する (‘$’により math モードに出入りする)。

式プレフィクス: ‘'’
: 式に隣接して出現した場合には式の一部とみなされる構文的演算子にたいして使用される文字。Lisp モードではアポストロフィー ‘'’ (クォートに使用)、カンマ ‘,’ (マクロに使用)、‘#’ (特定のデータ型にたいするリード構文として使用) が、これらの文字に含まれる。

コメント開始文字: ‘<’
コメント終了文字: ‘>’
: さまざまな言語においてコメントを区切るために使用する文字。人間用のテキストはコメント文字をもたない。Lisp ではセミコロン (‘;’) がコメントの開始、改行かフォームフィードで終了する。

標準構文の継承: ‘@’
: この構文クラスは特定の構文を指定しない。これはその文字の構文を探すために標準構文テーブルを照合するよう告げる。

汎用コメント区切り: ‘!’
: 特殊なコメントを開始または終了させる文字。任意の汎用コメント区切りは任意の汎用コメント区切りにマッチするが、コメント開始とコメント終了はマッチできない。汎用コメント区切りは汎用コメント区切り同士としかマッチできない。

  この構文クラスは主として `syntax-table`テキストプロパティ(Section 35.4 [Syntax Properties], page 912 を参照) とともに使用することを意図している。任意の文字範囲の最初と最後の文字にたいして、それらが汎用コメント区切りであることを示す`syntax-table`プロパティを付与することにより、その範囲がコメントを形成するとマークすることができる。

汎用文字列区切り: ‘|’
: 文字列を開始や終了させる文字。任意の汎用文字列区切りは任意の汎用文字列区切りにマッチするが、通常の文字列クォート文字とはマッチできない。

  この構文クラスは主として `syntax-table`テキストプロパティ(Section 35.4 [Syntax Properties], page 912 を参照) とともに使用することを意図している。任意の文字範囲の最初と最後の文字にたいして、それらが汎用文字列区切りであることを示す`syntax-table`プロパティを付与することにより、その範囲が文字列定数を形成するとマークすることができる。

35.2.2 構文フラグ

構文テーブル内の文字全体にたいして構文クラスに加えてフラグを指定できます。利用できる 8 つのフラグがあり、それらは文字 ‘1’、‘2’、‘3’、‘4’、‘b’、‘c’、‘n’、‘p’で表されます。

‘p’を除くすべてのフラグはコメント区切りを記述するために使用されます。数字のフラグは 2 文字から構成されるコメント区切りにたいして使用されます。これらは文字の文字クラスに関連付けられた構文的プロパティに加えて、その文字も同様にコメントシーケンスの一部となれることを示します。C モードでは区切り文字であり、かつコメントシーケンス開始 (‘/*’) の 2 文字目であり、かつコメントシーケンス終了 (‘*/’) の 1 文字目である ‘*’のような文字のためにフラグとクラスは互いに独立しています。フラグ ‘b’、‘c’、‘n’は対応するコメント区切りを限定するために使用されます。

以下は文字 cにたいして利用できるフラグと意味を示すテーブルです:

- ‘1’は cが 2 文字からなるコメント開始シーケンスの開始であることを意味する。
- ‘2’は cがそのようなシーケンスの 2 文字目であることを意味する。
- ‘3’は cが 2 文字からなるコメント終了シーケンスの開始であることを意味する。

- '4'は *c*がそのようなシーケンスの 2 文字目であることを意味する。
- 'b'は *c*が代替えのコメントスタイル "b" に属するコメント区切りであることを意味する。このフラグは 2 文字のコメント開始では 2 文字目、2 文字のコメント終了では 1 文字目にたいしてのみ意味をもつ。
- 'c'は *c*が代替えのコメントスタイル "c" に属するコメント区切りであることを意味する。2 文字からなるコメント区切りにたいしては、そのいずれかが 'c'であればスタイル "c" となる。
- コメント区切り文字での 'n'は、この種のコメントがネスト可能であることを指定する。このようなコメント内では、同じスタイルのコメントだけが認識される。2 文字からなるコメント区切りにたいしては、そのいずれかが 'n'であればネスト可能となる。

  Emacs は任意の構文テーブル 1 つにたいして同時に複数のコメントスタイルをサポートする。コメントスタイルはフラグ 'b'、'c'、'n'の組み合わせで 8 個の異なるコメントスタイルが可能で、コメントスタイルはそれぞれフラグセットにより命名される。コメント区切りはそれぞれスタイルをもち、同じスタイルのコメント区切りとのみマッチできる。つまりコメントがスタイル "bn" のコメント開始シーケンスで開始されるなら、そのコメントは次のスタイル "bn" のコメント終了シーケンスにマッチするまで拡張されるだろう。フラグセットが 'b'と 'c'のいずれでもなければ、結果となるスタイルは "a" スタイルと呼ばれる。

  C++にたいして適切なコメント構文は以下のようになる:

  '/' '124'

  '*' '23b'

  newline '>'

  これは 4 つのコメント区切りシーケンスを定義する:

  '/*' これは 2 文字目の '*'が 'b'フラグをもつので、"b" スタイルのコメント開始シーケンス。

  '//' これは 2 文字目の '/'が 'b'フラグをもたないので、"a" スタイルのコメント開始シーケンス。

  '*/' これは 1 文字目の '*'が 'b'フラグをもつので、"b" スタイルのコメント終了シーケンス。

  newline これは改行文字が 'b'フラグをもたないので、"a" スタイルのコメント終了シーケンス。
- 'p'は Lisp 構文にたいして追加のプレフィクス文字を識別する。これらが式の間に出現した際には空白文字として扱われる。これらが式の内部に出現したときは、それらの通常の構文クラスに応じて処理される。

  関数 `backward-prefix-chars`はこれらの文字、同様にメインの構文クラスがプレフィクスであるような文字 ('') を超えて後方に移動する。Section 35.5 [Motion and Syntax], page 913 を参照のこと。

35.3 構文テーブルの関数

このセクションでは構文テーブルの作成、アクセス、変更を行う関数を説明します。

`make-syntax-table` **&optional** *table* [Function]

この関数は新たに構文テーブルを作成する。*table*が非`nil`なら新たな構文テーブルの親は *table*、それ以外なら標準構文テーブルが親になる。

新たな構文テーブルでは最初はすべての文字に構文クラス “inherit”(‘@’) が与えられて、それらの構文は親テーブルから継承される (Section 35.2.1 [Syntax Class Table], page 907 を参照)。

`copy-syntax-table` **&optional** *table* [Function]

この関数は *table*のコピーを構築してそれをリターンする。*table*が省略または `nil`なら標準構文テーブルのコピーをリターンする。それ以外の場合には、*table*が構文テーブルでなければエラーをシグナルする。

`modify-syntax-entry` *char syntax-descriptor* **&optional** *table* [Command]

この関数は *syntax-descriptor*に応じて *char*の構文エントリーをセットする。*char*は文字、または `(`*min* `.` *max*`)`という形式のコンスセルでなければならない。後者の場合には、この関数は *min*と *max* (両端を含む) の間のすべての文字にたいして構文エントリーをセットする。

構文は *table* (デフォルトはカレントバッファーの構文テーブル) にたいしてのみ変更されて、他のすべての構文テーブルにたいしては変更されない。

引数 *syntax-descriptor*は構文記述子、すなわち 1 文字目が構文クラス指定子、2 文字目以降がオプションでマッチング文字と構文フラグを指定する文字列。Section 35.2 [Syntax Descriptors], page 907 を参照のこと。*syntax-descriptor*が有効な構文記述子でなければエラーがシグナルされる。

この関数は常に `nil`をリターンする。この文字にたいするテーブル内の古い構文情報は破棄される。

例:

```
;; 空白文字クラスのスペースを put する
(modify-syntax-entry ?\s " ")
     ⇒ nil

;; '$'を開カッコ文字にして、
;;    '^'を対応する閉カッコにする
(modify-syntax-entry ?$ "(^")
     ⇒ nil

;; '^'を閉カッコ文字にして
;;    '$'を対応する開カッコにする
(modify-syntax-entry ?^ ")$")
     ⇒ nil

;; '/'を区切り文字で
;;    コメント開始シーケンス 1 文字目、
;;    かつコメント終了シーケンス 2 文字目とする
;;    これは C モードで使用される
(modify-syntax-entry ?/ ". 14")
     ⇒ nil
```

`char-syntax` *character* [Function]

この関数は指定子文字 (Section 35.2.1 [Syntax Class Table], page 907 を参照) の表現で *character*の構文クラスをリターンする。これはクラスだけをリターンして、マッチング文字や構文フラグはリターンしない。

以下の例は C モードにたいして適用する (`char-syntax`がリターンする文字を確認しやすいように `string`を使用する)。

```
;; スペース文字は空白文字構文クラスをもつ
(string (char-syntax ?\s))
     ⇒ " "

;; スラッシュ文字は区切り文字構文をもつ。
;; コメント開始やコメント終了シーケンスの一部でもある場合、
;; char-syntax呼び出しはこれを明らかにしないことに注意。
(string (char-syntax ?/))
     ⇒ "."

;; 開カッコ文字は開カッコ構文をもつ。
;; これがマッチング文字 ')'をもつことは
;; char-syntax呼び出しでは自明ではないことに注意。
(string (char-syntax ?\())
     ⇒ "("
```

`set-syntax-table` *table* [Function]
この関数はカレントバッファーの構文テーブルを *table*にする。これは *table*をリターンする。

`syntax-table` [Function]
この関数はカレント構文テーブル (カレントバッファーのテーブル) をリターンする。

`describe-syntax` **&optional** *buffer* [Command]
このコマンドは *buffer* (デフォルトはカレントバッファー) の構文テーブルのコンテンツを help バッファーに表示する。

`with-syntax-table` *table body...* [Macro]
このマクロは *table*をカレント構文テーブルとして使用して *body*を実行する。これは古いカレント構文テーブルのリストア後に *body*の最後のフォームの値をリターンする。

各バッファーは独自にカレント構文テーブルをもつので、マクロはこれを入念に行うべきだろう。`with-syntax-table`はマクロの実行開始時には、そのときカレントのバッファーが何であれカレント構文テーブルを一時的に変更する。他のバッファーは影響を受けない。

35.4 構文プロパティ

ある言語の構文を指定するのに構文テーブルが十分に柔軟でないときは、バッファー内に出現する特定の文字にたいしてテキストプロパティ`syntax-table`を適用することにより構文テーブルをオーバーライドできます。テキストプロパティを適用する方法については Section 32.19 [Text Properties], page 813 を参照してください。

以下はテキストプロパティ`syntax-table`の有効な値です:

syntax-table
プロパティの値が構文テーブルなら、根底となるテキスト文字の構文を決定するカレントバッファーの構文テーブルのかわりにそのテーブルが使用される。

(*syntax-code . matching-char*)
この形式のコンスセルは根底となるテキスト文字の構文クラスを直接指定する raw 構文テーブル (Section 35.7 [Syntax Table Internals], page 917 を参照)。

`nil` このプロパティが `nil`なら、その文字の構文はカレント構文テーブルにより通常の方法で決定される。

`parse-sexp-lookup-properties` [Variable]
これが非 `nil`なら、`forward-sexp`のような構文をスキャンする関数は `syntax-table`テキストプロパティを考慮し、それ以外ならカレント構文テーブルだけを使用する。

`syntax-propertize-function` [Variable]
この変数が非 `nil`なら特定のテキスト範囲にたいして `syntax-table`プロパティを適用する関数を格納すること。これはモードに適した方法で `syntax-table`プロパティを適用する関数をインストールするようにメジャーモードで使用されることを意図している。

この関数は `syntax-ppss` (Section 35.6.2 [Position Parse], page 915 を参照)、および構文フォント表示化 (Section 23.6.8 [Syntactic Font Lock], page 505 を参照) の間に Font Lock モードにより呼び出される。これは作用すべきテキスト部分の開始 *start*と終了 *end*という 2 つの引数で呼び出される。これは *end*の前の任意の位置で `syntax-ppss`を呼び出すことが許されている。しかし `syntax-ppss-flush-cache`を呼び出すべきではなく、そのためある位置で `syntax-ppss`を呼び出して後からバッファー内の前の位置を変更することは許されていない。

`syntax-propertize-extend-region-functions` [Variable]
このアブノーマルフックは `syntax-propertize-function`呼び出しに先立ち構文解析コードにより実行される。これは `syntax-propertize-function`に渡すために安全なバッファーの開始と終了の位置を見つける助けをする役割をもつ。たとえばメジャーモードは複数行の構文構成を識別して、境界が複数行の中間にならないようにこのフックに関数を追加できる。

このフック内の各関数は引数 *start*と *end*を受け取ること。これは 2 つのバッファー位置を調整するコンスセル (`new-start . new-end`)、調整が必要なければ `nil`をリターンするべきである。フック関数はそれらすべてが `nil`をリターンするまで順番に繰り返し実行される。

35.5 モーションと構文

このセクションでは、特定の構文クラスをもつ文字間を横断して移動する関数を説明します。

`skip-syntax-forward` *syntaxes* **&optional** *limit* [Function]
この関数は *syntaxes*で指定された構文クラス (構文クラスの文字列) をもつ文字を横断してポイントを前方に移動する。バッファー終端か、(与えられた場合は) 位置 *limit*に到達、もしくはスキップしない文字に達した際に停止する。

*syntaxes*が '`^`'で始まる場合には、この関数は構文が *syntaxes*ではない文字をスキップする。

リターン値は移動した距離を表す非負の整数。

`skip-syntax-backward` *syntaxes* **&optional** *limit* [Function]
この関数は *syntaxes*で指定された構文クラスをもつ文字を横断してポイントを後方に移動する。バッファー先頭か、(与えられた場合は) 位置 *limit*に到達、もしくはスキップしない文字に達した際に停止する。

*syntaxes*が '`^`'で始まる場合には、この関数は構文が *syntaxes*ではない文字をスキップする。

リターン値は移動した距離を表す 0 以下の整数。

`backward-prefix-chars` [Function]
この関数は式プレフィクス構文の任意個数の文字を横断して後方にポイントを移動する。これには式プレフィクス構文クラスとフラグ '`p`'の文字の両方が含まれる。

35.6 式のパース

このセクションではバランスのとれた式の解析やスキャンを行う関数を説明します。たとえこれらの関数が Lisp 以外の言語にたいして作用可能であったとしても、Lisp 用語にしたがってそのような式のことを *sexps* という用語で参照することにします。基本的に sexp はバランスのとれたカッコによるグループ化、または文字列、シンボル (構文が単語構成要素かシンボル構成要素である文字シーケンス) のいずれかです。しかし式プレフィクス構文 (Section 35.2.1 [Syntax Class Table], page 907 を参照) の文字は、それらが sexp に隣接する場合には sexp の一部として扱われます。

構文テーブルは文字の解釈を制御するので、これらの関数は Lisp モードでの Lisp 式、C モードでの C の式にたいして使用できます。バランスのとれた式にたいして有用な高レベル関数については Section 30.2.6 [List Motion], page 762 を参照してください。

ある文字の構文はパーサー自身の状態の記述ではなくパーサー状態の変更方法を制御します。たとえば文字列区切り文字は in-string と in-code の間でパーサー状態をトグルしますが、文字の構文が直接文字列内部にあるかどうかを告げることはありません。たとえば (15 は汎用文字列区切りの構文コードであることに注意)、

```
(put-text-property 1 9 'syntax-table '(15 . nil))
```

これは Emacs にたいしてカレントバッファーの最初の 8 文字が文字列であることを告げますが、それらはすべて文字列区切りです。結果として Emacs はそれらを連続する 4 つの空文字列定数として扱います。

35.6.1 パースにもとづくモーションコマンド

このセクションでは式のパースにもとづいて処理を行うシンプルなポイント移動関数を説明します。

`scan-lists` *from count depth* [Function]

この関数は位置 *from*からバランスのとれたカッコのグループを前方に *count*個スキャンする。これはスキャンが停止した位置をリターンする。*count*が負ならスキャンは後方に移動する。

*depth*が非 0 なら開始位置のカッコのネスト深さを *depth*として扱う。スキャナーはネスト深さが 0 になるまで繰り返して *count*回、前方か後方に移動する。そのため正の *depth*は開始位置からカッコを *depth*レベル抜け出して移動する効果があり、負の *depth*はカッコが *depth*レベル深くなるよう移動する効果をもつ。

`parse-sexp-ignore-comments`が非 `nil`ならスキャンはコメントを無視する。

*count*個のカッコのグループをスキャンする前にスキャンがバッファーのアクセス可能範囲の先頭か終端に達した場合には、そのポイントのネスト深さが 0 なら値 `nil`をリターンする。ネスト深さが非 0 なら `scan-error`エラーをシグナルする。

`scan-sexps` *from count* [Function]

この関数は位置 *from*から *count*個の sexp を前方にスキャンする。これはスキャンが停止した位置をリターンする。*count*が負ならスキャンは後方へ移動する。

`parse-sexp-ignore-comments`が非 `nil`ならスキャンはコメントを無視する。

カッコのグループの中間でバッファー (のアクセス可能範囲) の先頭か終端に達したらエラーをシグナルする。*count*個を消費する前にカッコのグループの間でバッファーの先頭か終端に達したら `nil`をリターンする。

`forward-comment` *count* [Function]

この関数は *count*個の完全なコメント (すなわち、もしあれば開始区切りと終了区切りを含む)、および途中で遭遇する任意の空白文字を横断してポイントを前方に移動する。*count*が負なら

後方に移動する。コメントまたは空白文字以外のものに遭遇したら停止して停止位置にポイントを残す。これには、(たとえば) 前方に移動してコメント開始を調べる際にコメント終了を探すことも含まれる。この関数は指定された個数の完全なコメントを横断して移動した後にも即座に停止する。空白以外のものがコメント間に存在せずに期待どおり *count*個のコメントが見つかったら `t`、それ以外は `nil`をリターンする。

この関数はコメントを横断する際に、それが文字列内に埋め込まれているかどうか区別できない。それらがコメントのように見えればコメントとして扱われる。

ポイントの後のすべてのコメントと空白文字を飛び越して移動するには `(forward-comment (buffer-size))`を使用する。バッファー内のコメント数は `(buffer-size)`を超えることはできないので、これは引数としての使用に適している。

35.6.2 ある位置のパース状態を調べる

インデントのような構文分析にとっては、与えられたバッファー位置に応じた構文状態の計算が有用なことが多々あります。それを手軽に行うのが以下の関数です。

`syntax-ppss` **&optional** *pos* [Function]

この関数はパーサーがバッファーの可視範囲の先頭から開始して位置 *pos*で停止するだろうというパーサー状態をリターンする。パーサー状態の説明は次のセクションを参照のこと。

リターン値はバッファーの可視範囲の先頭から *pos*までパースするために低レベル関数 `parse-partial-sexp` (Section 35.6.4 [Low-Level Parsing], page 916 を参照) を呼び出した場合と同じようになる。しかし `syntax-ppss`は計算速度向上のためにキャッシュを使用する。この最適化のために、リターンされるパーサー状態のうち 2 つ目の値 (前の完全な部分式) と 6 つ目の値 (最小のカッコ深さ) は意味をもたない。

この関数は `syntax-ppss-flush-cache` (以下参照) にたいして、`before-change-functions` (Section 32.32 [Change Hooks], page 850 を参照) にバッファーローカルなエントリーを追加するという副作用をもつ。このエントリーはバッファー変更にたいしてキャッシュの一貫性を保つ。とはいえ `before-change-functions`が一時的に let でバインドされている間に `syntax-ppss`が呼び出された場合、または `inhibit-modification-hooks`使用時のようにバッファーがフックを実行せずに変更される場合にはキャッシュは更新されないかもしれない。そのような場合には明示的に `syntax-ppss-flush-cache`を呼び出す必要がある。

`syntax-ppss-flush-cache` *beg* **&rest** *ignored-args* [Function]

この関数は `syntax-ppss`が使用するキャッシュを位置 *beg*からフラッシュする。残りの引数 *ignored-args*は無視される。`before-change-functions` (Section 32.32 [Change Hooks], page 850 を参照) のような関数で直接使用できるように、この関数はそれらの引数を受け入れる。

35.6.3 パーサー状態

パーサー状態 *(parser state)* とは `parse-partial-sexp` (Section 35.6.4 [Low-Level Parsing], page 916 を参照) を使用してバッファー内の指定された開始位置と終了位置の間のテキストをパースした後の構文パーサーの状態を記述する (現在のところは)11 要素のリストです。`syntax-ppss`のようなパース関数も値としてパーサー状態をリターンします。 `parse-partial-sexp` can はパースを再開するために引数としてパーサー状態を受け取ります。

以下はパーサー状態の要素の意味です:

0. 0 から数えたカッコの深さ。警告**:** パーサーの開始位置と終了位置の間に開カッコより多くの閉カッコがあれば負になることもある。

1. 停止位置を含む最内のカッコグループの開始文字位置。なければ `nil`。
2. 最後の終端された完全な部分式の開始文字位置。なければ `nil`。
3. 文字列内部なら非 `nil`。より正確には文字列を終端させるであろう文字、または汎用文字列区切りが終端すべきような場合には `t`。
4. ネスト不可なコメント (または任意のコメントスタイル。Section 35.2.2 [Syntax Flags], page 909 を参照) の内部なら `t`、ネスト可なコメントの内部ならコメントのネストレベル。
5. 終了位置がクォート文字直後なら `t`。
6. 当該スキャン中に遭遇した最小のカッコ深さ。
7. アクティブなコメントの種類。コメント以外、またはスタイル 'a'のコメント内なら `nil`、スタイル 'b'のコメントなら 1、スタイル 'c'のコメントなら 2、汎用コメント区切り文字で終端されるべきコメントなら `syntax-table`。
8. 文字列やコメントの開始位置。コメント内部ならコメントが始まる位置。文字列内部なら文字列が始まる位置。文字列やコメントの外部ならこの要素は `nil`。
9. もっとも外側のカッコから始まる開きカッコのカレント位置のリスト。
10. スキャンされた最後のバッファー位置が (潜在的に)2 文字構文 (コメント区切りやエスケープされた文字とクォートされた文字のペア) の最初の文字ならその位置の *syntax-code* (Section 35.7 [Syntax Table Internals], page 917 を参照)、それ以外なら `nil`。

パース継続のための引数として `parse-partial-sexp`に渡す場合には要素 1、2、6 は無視されます。要素 9 と 10 は主にパーサーコードにより内部的に使用されます。

以下の関数を使用することにより追加でパーサー状態からいくつかの有用な情報を利用できます:

`syntax-ppss-toplevel-pos` *state* [Function]
この関数はパーサー状態 *state*から文法構造上トップレベルでのパースでのスキャンした最後の位置をリターンする。"トップレベル" とはすべてのカッコ、コメント、文字列の外部であることを意味する。

*state*がトップレベルの位置に到達したパースを表す場合には値は `nil`。

`syntax-ppss-context` *state* [Function]
*state*をリターンするスキャン終了位置が文字列内にあれば `string`、コメント内にあれば `comment`をリターンする。

35.6.4 低レベルのパース

式パーサーを使用するもっとも基本的な方法は特定の状態で与えられた位置からパースを開始して、指定した位置でパースを終了するよう指示する方法です。

`parse-partial-sexp` *start limit* **&optional** *target-depth stop-before state stop-comment* [Function]
この関数はカレントバッファー内の sexp を、*start*から開始して *limit*を超えてスキャンしないようパースを行う。これは位置 *limit*、または以下に記述する特定の条件に適合したら停止してパースが停止した位置にポイントをセットする。これはポイントが停止した位置でのパースの状態を記述するパーサー状態をリターンする。

3 つ目の引数 *target-depth*が非 `nil`の場合には、カッコの深さが *target-depth*と等しくなったらパースを停止する。この深さは 0、または *state*内で与えられる深さなら何であれそこから開始される。

4 つ目の引数 *stop-before*が非 `nil`の場合には、sexp 開始となる任意の文字に到達したときにパースは停止する。*stop-comment*が非 `nil`ならネストされていないコメントの開始の後にパースは停止する。*stop-comment*がシンボル `syntax-table`ならネストされていないコメントか文字列の開始の後、またはネストされていないコメントか文字列の終了のいずれか先に到達した方でパースは停止する。

*state*が `nil`なら、*start*は関数定義先頭のようなカッコ構造のトップレベルであるとみなされる。かわりにこの構造の中間でパースを再開したいと思うかもしれない。これを行うにはパースの初期状態を記述する *state*引数を提供しなければならない。前の `parse-partial-sexp`呼び出しでリターンされた値で、これをうまく行うことができるだろう。

35.6.5 パースを制御するためのパラメーター

`multibyte-syntax-as-symbol` [Variable]
この変数が非 `nil`なら構文テーブルがそれらについて何と言っているかに関わらず、`scan-sexps`はすべての非 ASCII文字をシンボル構成要素として扱う (とはいえ依然として `syntax-table`テキストプロパティは構文をオーバーラードできる)。

`parse-sexp-ignore-comments` [User Option]
この値が非 `nil`ならこのセクション内の関数、および `forward-sexp`、`scan-lists`、`scan-sexps`はコメントを空白文字として扱う。

`parse-partial-sexp`の振る舞いも `parse-sexp-lookup-properties`の影響を受けます (Section 35.4 [Syntax Properties], page 912 を参照)。

`comment-end-can-be-escaped` [Variable]
このバッファーローカル変数が非 `nil`なら、通常ならコメントを終端するような単一の文字は、エスケープ時にはコメントを終端しない。これは C と C++のモードにおいて '\'でエスケープされた改行により、'//'で開始される行コメントを次行に継続させるために使用される。

1 つ、または複数のコメントを横断して前方や後方に移動するには `forward-comment`を使用できます。

35.7 構文テーブルの内部

構文テーブルは文字テーブル (Section 6.6 [Char-Tables], page 110 を参照) として実装されていますが、ほとんどの Lisp プログラムが直接それらの要素に作用することはありません。構文テーブルは構文データとして構文記述子を格納しません (Section 35.2 [Syntax Descriptors], page 907 を参照)。それらは内部的なフォーマットを使用しており、それについてはこのセクションで説明します。この内部的フォーマットは構文プロパティとして割り当てることもできます (Section 35.4 [Syntax Properties], page 912 を参照)。

構文テーブル内の各要素は *raw* 構文記述子 *(raw syntax descriptor)* という `(`*`syntax-code`*` . `*`matching-char`*`)`という形式のコンスセルです。*syntax-code*は下記のテーブルに応じて構文クラスと構文フラグをエンコードする整数です。*matching-char*が非 `nil`なら、それはマッチング文字 (構文記述子内の 2 つ目の文字と同様) を指定します。

raw 構文記述子の文字を取得するには `(aref (syntax-table) ch)`のように `aref` (Section 6.3 [Array Functions], page 107 を参照) を使用してください。

以下はさまざまな構文クラスに対応する構文コードです。

Code	*Class*	*Code*	*Class*

0	空白文字	8	区切り文字ペアー
1	句読点	9	エスケープ
2	単語	10	文字クォート
3	シンボル	11	コメント開始
4	開カッコ	12	コメント終了
5	閉カッコ	13	継承
6	式プレフィクス	14	汎用コメント
7	文字列クォート	15	汎用文字列

たとえば標準構文テーブルでは‘(’にたいするエントリーは `(4 . 41)`、41 は‘)’の文字コードです。

構文フラグは最下位ビットから 16 ビット目より始まる高位ビットにエンコードされます。以下のテーブルは対応する各構文フラグにたいして 2 のべき乗を与えます。

Prefix	*Flag*	*Prefix*	*Flag*
‘1’	`(ash 1 16)`	‘p’	`(ash 1 20)`
‘2’	`(ash 1 17)`	‘b’	`(ash 1 21)`
‘3’	`(ash 1 18)`	‘n’	`(ash 1 22)`
‘4’	`(ash 1 19)`	‘c’	`(ash 1 23)`

`string-to-syntax` *desc* [Function]

与えられた構文記述子 *desc*(文字列) にたいして、この関数は対応する raw 構文記述子をリターンする。

`syntax-after` *pos* [Function]

この関数はバッファー内の位置 *pos*の後の文字にたいして、構文テーブルと同様に構文プロパティも考慮した raw 構文記述子をリターンする。*pos*がバッファーのアクセス可能範囲 (Section 30.4 [Narrowing], page 766 を参照) の外部ならリターン値は `nil`。

`syntax-class` *syntax* [Function]

この関数は raw 構文記述子 *syntax*にたいする構文コードをリターンする。より正確にはこれは raw 構文記述子の *syntax-code*要素から構文フラグを記録する高位 16 ビットをマスクして、その結果の整数をリターンする。

*syntax*が `nil`ならリターン値は `nil`。これは以下の式

```
(syntax-class (syntax-after pos))
```

は `pos`がバッファーのアクセス可能範囲外部なら、エラーを throw したり不正なコードをリターンすることなく `nil`に評価されるため。

35.8 カテゴリー

カテゴリー *(categories)* は構文的に文字をクラス分けする別の手段を提供します。必要に応じて複数のカテゴリーを定義して、それぞれの文字に独立して 1 つ以上のカテゴリーを割り当てることができます。構文クラスと異なりカテゴリーは互いに排他ではありません。1 つの文字が複数のカテゴリーに属すのは普通のことです。

バッファーはそれぞれカテゴリーテーブル *(category table)* をもっています。これはどのカテゴリーが定義されていて、各カテゴリーにどの文字が属すかを記録しています。カテゴリーテールは自身のカテゴリーを定義しますが、標準カテゴリーはすべてのモードで利用可能なので、これらは通常は標準カテゴリーテーブルをコピーすることにより初期化されます。

カテゴリーはそれぞれ‘ ’から‘~’の範囲の ASCIIプリント文字による名前をもちます。`define-category`で定義する際にはカテゴリーの名前を指定します。

カテゴリーテーブルは実際には文字テーブルです (Section 6.6 [Char-Tables], page 110 を参照)。カテゴリーテーブルのインデックス *c*の要素は、文字 *c*が属するカテゴリーを示すカテゴリーセット *(category set)* というブールベクターです。このカテゴリーセット内で、もしインデックス *cat*の要素が `t`なら *cat*はそのセットのメンバーであり、その文字 *c*はカテゴリー *cat*に属することを意味します。

以下の 3 つの関数のオプション引数 *table*のデフォルトは、カレントバッファーのカテゴリーテーブルです。

`define-category` *char docstring* **&optional** *table* [Function]

この関数はカテゴリーテーブル *table*にたいして名前が *char*、ドキュメントが *docstring*であるような新たなカテゴリーを定義する。

以下では R2L(right-to-left: 右から左) への強い方向性 (directionality) をもつ文字 (Section 39.26 [Bidirectional Display], page 1085 を参照) にたいするカテゴリーを新たに定義して、それを特別なカテゴリーテーブル内で使用する例を示す。文字の方向性に関する情報を取得するために、コード例では Unicode プロパティ‘`bidi-class`’ (Section 33.6 [Character Properties], page 857 を参照) を使用する。

```
(defvar special-category-table-for-bidi
  ;;      空の category-table を作成
  (let ((category-table (make-category-table))
        ;; Create a char-table which gives the 'bidi-class' Unicode
        ;; 各文字のプロパティ
        (uniprop-table
         (unicode-property-table-internal 'bidi-class)))
    (define-category ?R "Characters of bidi-class R, AL, or RLO"
                     category-table)
    ;; Unicode の'bidi-class'プロパティが
    ;; R、AL、RLO であるような (R2L の方向性をもつ) 文字の
    ;; カテゴリーエントリーを変更する
    (map-char-table
     (lambda (key val)
         (if (memq val '(R AL RLO))
             (modify-category-entry key ?R category-table)))
     uniprop-table)
    category-table))
```

`category-docstring` *category* **&optional** *table* [Function]

この関数はカテゴリーテーブル *table*内のカテゴリー *category*のドキュメント文字列をリターンする。

```
(category-docstring ?a)
     ⇒ "ASCII"
(category-docstring ?l)
     ⇒ "Latin"
```

`get-unused-category` **&optional** *table* [Function]

この関数は *table*内で現在のところ未定義なカテゴリーの名前 (文字) をリターンする。*table*内で利用可能なカテゴリーがすべて使用済みなら `nil`をリターンする。

`category-table` [Function]
この関数はカレントバッファーのカテゴリーテーブルをリターンする。

`category-table-p` *object* [Function]
この関数は *object*がカテゴリーテーブルなら `t`、それ以外は `nil`をリターンする。

`standard-category-table` [Function]
この関数は標準カテゴリーテーブルをリターンする。

`copy-category-table` **&optional** *table* [Function]
この関数は *table*のコピーを構築してリターンする。*table*が与えられない (または `nil`) なら、標準カテゴリーテーブルのコピーをリターンする。それ以外の場合には、もし *table*がカテゴリーテーブルでなければエラーをシグナルする。

`set-category-table` *table* [Function]
この関数は *table*をカレントバッファーのカテゴリーテーブルにする。リターン値は *table*。

`make-category-table` [Function]
これは空のカテゴリーテーブルを作成してリターンする。空のカテゴリーテーブルでは、どのカテゴリーも割り当てられておらず何らかのカテゴリーに属する文字もない。

`make-category-set` *categories* [Function]
この関数は初期内容が文字列 *categories*にリストされるカテゴリーであるような、新たなカテゴリーセット (ブールベクター) をリターンする。*categories*の要素はカテゴリー名であること。新たなカテゴリーセットはそれらのカテゴリーにたいして `t`、それ以外のすべてのカテゴリーにたいして `nil`をもつ。

```
(make-category-set "al")
     ⇒ #&128"\0\0\0\0\0\0\0\0\0\0\0\0\2\20\0\0"
```

`char-category-set` *char* [Function]
この関数はカレントバッファーのカテゴリーテーブル内で、文字 *char*にたいするカテゴリーセットをリターンする。これは文字 *char*が属するカテゴリーを記録するブールベクター。関数 `char-category-set`はカテゴリーテーブル内にある同じブールベクターをリターンするのでメモリーの割り当ては行わない。

```
(char-category-set ?a)
     ⇒ #&128"\0\0\0\0\0\0\0\0\0\0\0\0\2\20\0\0"
```

`category-set-mnemonics` *category-set* [Function]
この関数はカテゴリーセット *category-set*を、そのセットのメンバーのカテゴリーを指定する文字を含む文字列に変換する。

```
(category-set-mnemonics (char-category-set ?a))
     ⇒ "al"
```

`modify-category-entry` *char category* **&optional** *table reset* [Function]
この関数はカテゴリーテーブル *table* (デフォルトはカレントバッファーのカテゴリーテーブル) 内の *char*のカテゴリーセットを変更する。*char*には文字、または `(min . max)`という形式のコンスセルを指定できる。後者の場合には、この関数は *min*と *max*の間 (両端を含む) の範囲にあるすべての文字のカテゴリーセットを変更する。

これは通常はカテゴリーセットに *category*を追加することにより変更を行う。しかし *reset*が非 `nil`なら、かわりに *category*を削除する。

`describe-categories` **&optional** *buffer-or-name* [Command]

この関数はカレントカテゴリーテーブル内のカテゴリー仕様を説明する。これはその説明をバッファーに挿入してから、そのバッファーを表示する。*buffer-or-name*が非`nil`なら、かわりにそのバッファーのカテゴリーテーブルを説明する。

36 abbrev と abbrev 展開

略語 (abbreviation または *abbrev* は、より長い文字列へと展開される文字列です。ユーザーは abbrev 文字列を挿入して、それを探して自動的に abbrev の展開形に置換できます。これによりタイプ量を節約できます。

カレントで効果をもつ abbrevs のセットは *abbrev* テーブル *(abbrev table)* 内に記録されます。バッファーはそれぞれローカルに abbrev テーブルをもちますが、通常は同一のメジャーモードにあるすべてのバッファーが 1 つの abbrev テーブルを共有します。グローバル abbrev テーブルも存在します。通常は両者が使用されます。

abbrev テーブルは obarray として表されます。obarrays についての情報は Section 9.3 [Creating Symbols], page 126 を参照してください。abbrev はそれぞれ obarray 内のシンボルとして表現されます。そのシンボルの名前が abbrev であり、値が展開形になります。シンボルの関数定義は展開を行うフック関数です (Section 36.2 [Defining Abbrevs], page 923 を参照)。またシンボルのプロパティセルには使用回数やその abbrev が展開された回数を含む、さまざまな追加プロパティが含まれます (Section 36.6 [Abbrev Properties], page 927 を参照)。

システム *abbrev(system abbrevs)* と呼ばれる特定の abbrev は、ユーザーではなくメジャーモードにより定義されます。システム abbrev は非 `nil`の`:system`プロパティにより識別されます (Section 36.6 [Abbrev Properties], page 927 を参照)。abbrev が abbrev ファイルに保存される際には、システム abbrev は省略されます。Section 36.3 [Abbrev Files], page 924 を参照してください。

abbrev に使用されるシンボルは通常の obarray に intern されないので、Lisp 式の読み取り結果として現れることは決してありません。実際のところ通常は abbrev を扱うコードを除いて、それらが使用されることはありません。したがってそれらを非標準的な方法で使用しても安全なのです。

マイナーモードである Abbrev モードが有効な場合には、バッファーローカル変数 `abbrev-mode` は非 `nil`となり、そのバッファー内で abbrev は自動的に展開されます。abbrev 用のユーザーレベルのコマンドについては Section “Abbrev Mode” in *The GNU Emacs Manual* を参照してください。

36.1 abbrev テーブル

このセクションでは abbrev テーブルの作成と操作を行う方法について説明します。

`make-abbrev-table` **&optional** *props* [Function]
: この関数は空の abbrev テーブル (シンボルを含まない obarray) を作成してリターンする。これは 0 で充填されたベクター。*props*は新たなテーブルに適用されるプロパティリスト (Section 36.7 [Abbrev Table Properties], page 928 を参照)。

`abbrev-table-p` *object* [Function]
: この関数は *object*が abbrev テーブルなら非 `nil`をリターンする。

`clear-abbrev-table` *abbrev-table* [Function]
: この関数は *abbrev-table*内の abbrev をすべて未定義として空のまま残す。

`copy-abbrev-table` *abbrev-table* [Function]
: この関数は *abbrev-table*のコピー (同じ abbrev 定義を含む新たな abbrev テーブル) をリターンする。これは名前、値、関数だけをコピーしてプロパティリストは何もコピーしない。

`define-abbrev-table` *tabname definitions* **&optional** *docstring* **&rest** *props* [Function]

この関数は abbrev テーブル名 (値が abbrev テーブルであるような変数) として *tabname* (シンボル) を定義する。これはそのテーブル内に *definitions*に応じて、abbrev を定義する。*definitions*は `(abbrevname expansion [hook] [props...])`という形式の要素をもつリスト。これらの要素は引数として `define-abbrev`に渡される。

オプション文字列 *docstring*は変数 *tabname*のドキュメント文字列。プロパティリスト *props* は abbrev テーブルに適用される (Section 36.7 [Abbrev Table Properties], page 928 を参照)。

同一の *tabname*にたいしてこの関数が複数回呼び出されれると、元のコンテンツ全体を上書きせずに後続の呼び出しは *definitions*内の定義を *tabname*に追加する (後続の呼び出しでは *definitions*内で明示的に再定義または未定義にした場合のみ abbrev を上書きできる)。

`abbrev-table-name-list` [Variable]

これは値が abbrev テーブルであるようなシンボルのリスト。`define-abbrev-table`はこのリストに新たな abbrev テーブル名を追加する。

`insert-abbrev-table-description` *name* **&optional** *human* [Function]

この関数はポイントの前に名前が *name*の abbrev テーブルの説明を挿入する。引数 *name*は値が abbrev テーブルであるようなシンボル。

*human*が非 `nil`なら人間向けの説明になる。システム abbrev はそのようにリストされて識別される。それ以外なら説明は Lisp 式 (カレントで定義されているように *name*を定義するがシステム abbrev としては定義しないような `define-abbrev-table`呼び出し) となる (*name* を使用するモードまたはパッケージはそれらを個別に *name*に追加すると想定されている)。

36.2 abbrev の定義

`define-abbrev`は abbrev テーブル内に abbrev を定義するための基本的な低レベル関数です。

メジャーモードがシステム abbrev を定義する際には、`:system`プロパティに `t`を指定して `define-abbrev`を呼び出すべきです。すべての保存された非システム abbrev は起動時 (何らかのメジャーモードがロードされる前) にリストアされることに注意してください。したがってメジャーモードは最初にそのモードがロードされた際には、それらのモードの abbrev テーブルが空であると仮定するべきではありません。

`define-abbrev` *abbrev-table name expansion* **&optional** *hook* **&rest** *props* [Function]

この関数は *abbrev-table*内に *name*という名前で *expansion*に展開されて、*hook*を呼び出す abbrev をプロパティ *props* (Section 36.6 [Abbrev Properties], page 927 を参照) とともに定義する。リターン値は *name*。ここでは *props*内の`:system`プロパティは特別に扱われる。このプロパティが値 `force`をもつなら、たとえ同じ名前の非システム abbrev でも既存の定義を上書きするだろう。

*name*は文字列であること。引数 *expansion*は通常は望む展開形 (文字列) であり、`nil`ならその abbrev を未定義とする。これが文字列または `nil`以外の何かなら、その abbrev は *hook*を実行することにより単に展開される。

引数 *hook*は関数または `nil`であること。*hook*が非 `nil`なら abbrev が *expansion*に置換された後に引数なしでそれが呼び出される。*hook*呼び出しの際にはポイントは *expansion*の終端に配置される。

*hook*が `no-self-insert`プロパティが非 `nil`であるような非 `nil`のシンボルなら、*hook*は展開をトリガーするような自己挿入入力文字を挿入できるかどうかを明示的に制御できる。この場合には、*hook*が非 `nil`をリターンしたらその文字の挿入を抑止する。対照的に *hook*が `nil`をリターンしたら、あたかも実際には展開が行われなかったかのように `expand-abbrev` (または `abbrev-insert`) も `nil`をリターンする。

`define-abbrev`は実際に abbrev を変更した場合には、通常は変数 `abbrevs-changed`に `t`をセットする。これはいくつかのコマンドが abbrev の保存を提案するためである。いずれにせよシステム abbrev は保存されないので、システム abbrev にたいしてこれは行われない。

`only-global-abbrevs` [User Option]
この変数が非 `nil`なら、それはユーザーがグローバル abbrev のみの使用を計画していることを意味する。これはモード固有の abbrev を定義するコマンドにたいして、かわりにグローバル abbrev を定義するよう指示する。この変数はこのセクション内の関数の振る舞いを変更しない。それは呼び出し側により検証される。

36.3 ファイルへの abbrev の保存

abbrev 定義が保存されたファイルは実際には Lisp コードのファイルです。abbrev は同じコンテンツの同じ abbrev テーブルを定義する Lisp プログラムの形式で保存されます。したがってそのファイルは `load`によってロードすることができます (Section 16.1 [How Programs Do Loading], page 269 を参照)。しかしより簡便なインターフェースとして関数 `quietly-read-abbrev-file`が提供されています。Emacs は起動時に自動的にこの関数を呼び出します。

`save-some-buffers`のようなユーザーレベルの機能は、ここで説明する変数の制御下で自動的に abbrev をファイルに保存できます。

`abbrev-file-name` [User Option]
これは abbrev の読み込みと保存にたいするデフォルトのファイル名。デフォルトでは Emacs は`~/.emacs.d/abbrev_defs`を探して、見つからなければ`~/.abbrev_defs`を探して、いずれにもファイルが存在しなければ`~/.emacs.d/abbrev_defs`を作成する。

`quietly-read-abbrev-file` **&optional** *filename* [Function]
この関数は以前に `write-abbrev-file`で書き込まれた *filename*という名前のファイルから abbrev の定義を読み込む。*filename*が省略または `nil`なら `abbrev-file-name`内で指定されているファイルが使用される。

関数の名前が暗示するようにこの関数は何のメッセージも表示しない。

`save-abbrevs` [User Option]
`save-abbrevs`にたいする非 `nil`値はファイル保存時に、(もし何か変更されていれば)Emacs が abbrev の保存を提案するべきであることを意味する。値が `silently`なら Emacs はユーザーに尋ねることなく abbrev を保存する。`abbrev-file-name`は abbrev を保存するファイルを指定する。デフォルト値は `t`。

`abbrevs-changed` [Variable]
この変数は abbrev(システム abbrev を除く) の定義や変更によりセットされる。さまざまな Emacs コマンドにとって、これはユーザーに abbrev の保存を提案するためのフラグとしての役目をもつ。

`write-abbrev-file` **&optional** *filename* [Command]
`abbrev-table-name-list`内にリストされたすべての abbrev テーブルにたいして、ロード時に同じ abbrev を定義するであろう Lisp プログラム形式で、すべての abbrev 定義 (システム abbrev を除く) をファイル *filename*内に保存する。保存すべき abbrev がないテーブルは省略する。*filename*が `nil`なら `abbrev-file-name`が使用される。この関数は `nil`をリターンする。

36.4 略語の照会と展開

abbrev は通常は `self-insert-command`を含む特定の interactive なコマンドにより展開されます。このセクションではそのようなコマンドの記述に使用されるサブルーチン、並びに通信のために使用される変数について説明します。

`abbrev-symbol` *abbrev* **&optional** *table* [Function]
この関数は *abbrev*という名前の abbrev を表すシンボルをリターンする。その abbrev が定義されていなければ `nil`をリターンする。オプションの 2 つ目の引数 *table*はそれを照合するための abbrev テーブル。*table*が `nil`ならこの関数はまずカレントバッファーのローカル abbrev テーブル、次にグローバル abbrev テーブルを試みる。

`abbrev-expansion` *abbrev* **&optional** *table* [Function]
この関数は *abbrev*が展開されるであろう文字列 (カレントバッファーにたいして使用される abbrev テーブルで定義される文字列) をリターンする。これは *abbrev*が有効な abbrev でなければ `nil`をリターンする。オプション引数 *table*は `abbrev-symbol`の場合と同じように使用する abbrev テーブルを指定する。

`expand-abbrev` [Command]
このコマンドは、(もしあれば) ポイントの前の abbrev を展開する。ポイントが abbrev の後になければこのコマンドは何もしない。展開を行うためにこれは変数 `abbrev-expand-function`の値となっている関数を引数なしで呼び出して、何であれその関数がリターンしたものをリターンする。

デフォルトの展開関数は展開を行ったら abbrev のシンボル、それ以外は `nil`をリターンする。その abbrev シンボルが `no-self-insert`プロパティが非 `nil`のシンボルであるようなフック関数をもち、そのフック関数が値として `nil`をリターンした場合には、たとえ展開が行われたとしてもデフォルト展開関数は `nil`をリターンする。

`abbrev-insert` *abbrev* **&optional** *name start end* [Function]
この関数は `start`と `end`の間のテキストを置換することにより `abbrev`の abbrev 展開形を挿入する。`start`が省略された場合のデフォルトはポイント。`name`が非 `nil`なら、それはこの abbrev が見つかった名前 (文字列) であること。これは展開形の capitalization を調整するかどうかを判断するために使用される。この関数は abbrev の挿入に成功したら `abbrev`、それ以外は `nil`をリターンする。

`abbrev-prefix-mark` **&optional** *arg* [Command]
このコマンドはポイントのカレント位置を abbrev の開始としてマークする。`expand-abbrev`の次回呼び出しでは、通常のように以前の単語ではなく、ここからポイント (その時点での位置) にあるテキストが展開するべき abbrev として使用される。

このコマンドは、まず *arg*が `nil`ならポイントの前の任意の abbrev を展開する (インタラクティブな呼び出しでは *arg*はプレフィクス引数)。それから展開する次の abbrev の開始を示すためにポイントの前にハイフンを挿入する。実際の展開ではハイフンは削除される。

`abbrev-all-caps` [User Option]
これが非 `nil`にセットされているときは、すべて大文字で入力された abbrev はすべて大文字を使用して展開される。それ以外ならすべて大文字で入力された abbrev は、展開形の単語ごとに capitalize して展開される。

`abbrev-start-location` [Variable]
この変数の値は次に abbrev を展開する開始位置として `expand-abbrev`に使用されるバッファー位置。値は `nil`も可能であり、それはかわりにポイントの前の単語を使用することを意味する。`abbrev-start-location`は `expand-abbrev`の呼び出しごとに毎回 `nil`にセットされる。この変数は `abbrev-prefix-mark`からもセットされる。

`abbrev-start-location-buffer` [Variable]
この変数の値は `abbrev-start-location`がセットされたバッファー。他のバッファーで abbrev 展開を試みることにより `abbrev-start-location`はクリアーされる。この変数は `abbrev-prefix-mark`によりセットされる。

`last-abbrev` [Variable]
これは直近の abbrev 展開の `abbrev-symbol`。これは `unexpand-abbrev`コマンド (Section "Expanding Abbrevs" in *The GNU Emacs Manual* を参照) のために `expand-abbrev`により残された情報である。

`last-abbrev-location` [Variable]
これは直近の.abbrev 展開の場所。これには `unexpand-abbrev`コマンドのために `expand-abbrev`により残された情報が含まれる。

`last-abbrev-text` [Variable]
これは直近の abbrev 展開の正確な展開形を、(もしあれば) 大文字小文字変換した後のテキストである。その abbrev がすでに非展開されていれば値は `nil`。これには `unexpand-abbrev`コマンドのために `expand-abbrev`ga 残 sita 情報が含まれる。

`abbrev-expand-function` [Variable]
この変数の値は展開を行うために `expand-abbrev`が引数なしで呼び出すであろう関数。この関数では展開を行う前後に行いたいことを行うことができる。展開が行われた場合にはその abbrev シンボルをリターンすること。

以下のサンプルコードでは `abbrev-expand-function`のシンプルな使い方を示します。このサンプルでは `foo-mode`が '`#`'で始まる行がコメントであるような特定のファイルを編集するためのモードであるとします。それらコメント行にたいしては Text モードの abbrev の使用が望ましく、その他すべての行にたいしては正規のローカル abbrev テーブル `foo-mode-abbrev-table`が適しています。`local-abbrev-table`と `text-mode-abbrev-table`の定義については、Section 36.5 [Standard Abbrev Tables], page 927 を参照してください。`add-function`についての詳細は Section 13.11 [Advising Functions], page 227 を参照してください。

```
(defun foo-mode-abbrev-expand-function (expand)
  (if (not (save-excursion (forward-line 0) (eq (char-after) ?#)))
      ;; 通常の展開を行う
      (funcall expand)
    ;; コメント内は text-mode の abbrev を使用
    (let ((local-abbrev-table text-mode-abbrev-table))
      (funcall expand))))

(add-hook 'foo-mode-hook
```

```
(lambda ()
  (add-function :around (local 'abbrev-expand-function)
                #'foo-mode-abbrev-expand-function)))
```

36.5 標準 abbrev テーブル

以下は Emacs の事前ロードされるメジャーモード用の abbrev テーブルを保持する変数のリストです。

`global-abbrev-table` [Variable]
: これはモードに非依存な abbrev 用の abbrev テーブル。この中で定義される abbrev はすべてのバッファーに適用される。各バッファーはローカル abbrev テーブルももつかもしれず、それの abbrev 定義はグローバルテーブル内の abbrev 定義より優先される。

`local-abbrev-table` [Variable]
: このバッファーローカル変数の値はカレントバッファーの (モード固有な)abbrev テーブルである。これはそのようなテーブルのリストでもあり得る。

`abbrev-minor-mode-table-alist` [Variable]
: この変数の値は `(`*`mode`*` . `*`abbrev-table`*`)`という形式のリスト。ここで *mode*は変数の名前。その変数が非 `nil`にバインドされていれば *abbrev-table*はアクティブ、それ以外なら無視される。*abbrev-table*は abbrev テーブルのリストでもあり得る。

`fundamental-mode-abbrev-table` [Variable]
: これは Fundamental モードで使用されるローカル abbrev テーブル。言い換えるとこれは Fundamental モードにあるすべてのバッファーのローカル abbrev テーブルである。

`text-mode-abbrev-table` [Variable]
: これは Text モードで使用されるローカル abbrev テーブル。

`lisp-mode-abbrev-table` [Variable]
: これは Lisp モードで使用されるローカル abbrev テーブルであり、Emacs Lisp モードで使用されるローカル abbrev テーブルの親テーブル。Section 36.7 [Abbrev Table Properties], page 928 を参照のこと。

36.6 abbrev プロパティー

abbrev はプロパティをもち、それらのいくつかは abbrev の働きに影響します。これらのプロパティを `define-abbrev`の引数として提供して以下の関数で操作できます:

`abbrev-put` *abbrev prop val* [Function]
: *abbrev*のプロパティ*prop*に値 *val*をセットする。

`abbrev-get` *abbrev prop* [Function]
: *abbrev*のプロパティ*prop*、その abbrev がそのようなプロパティをもたなければ `nil`をリターンする。

以下のプロパティには特別な意味があります:

`:count` このプロパティはその abbrev が展開された回数を計数する。明示的にセットしなければ `define-abbrev`により 0 に初期化される。

`:system` 非 `nil`ならこのプロパティはシステム abbrev としてその abbrev をマスクする。そのような abbrev は保存されない (Section 36.3 [Abbrev Files], page 924 を参照)。

`:enable-function`
: 非`nil`の場合には、その abbrev が使用されるべきでなければ`nil`、それ以外なら`t`をリターンするような引数なしの関数であること。

`:case-fixed`
: 非`nil`なら、このプロパティはその abbrev の case(大文字小文字) には意味があり、同じパターンに capitalize されたテキストだけにマッチすべきことを示す。これは展開の capitalization を変更するコードも無効にする。

36.7 abbrev テーブルのプロパティー

abbrev と同じように abbrev テーブルもプロパティをもち、それらのいくつかは abbrev テーブルの働きに影響を与えます。これらのプロパティを`define-abbrev-table`の引数として提供して、それらを関数で操作できます:

`abbrev-table-put` *table prop val* [Function]
: abbrev テーブル *table*のプロパティ*prop*に値 *val*をセットする。

`abbrev-table-get` *table prop* [Function]
: abbrev テーブルのプロパティ*prop*、abbrev テーブル *table*がそのようなをプロパティもたなければ`nil`をリターンする。

以下のプロパティには特別な意味があります:

`:enable-function`
: abbrev プロパティ`:enable-function`と似ているが、そのテーブル内のすべての abbrev に適用される点が異なる。これはポイントの前の abbrev を探すことを試みる前にも使用されるので abbrev テーブルを動的に変更することが可能。

`:case-fixed`
: これは abbrev プロパティ`:case-fixed`と似ているが、そのテーブル内のすべての abbrev に適用される点が異なる。

`:regexp` 非`nil`なら、このプロパティはそのテーブルを照合する前にポイント前の abbrev 名を抽出するための方法を示す正規表現。その正規表現がポイントの前にマッチしたときは、その abbrev 名は submatch の 1 と期待される。このプロパティが`nil`ならデフォルトは`backward-word`と`forward-word`を使用して abbrev の名前を探す。このプロパティにより単語構文以外の文字を含む名前の abbrev が使用できる。

`:parents` このプロパティは他の abbrev を継承したテーブルのリストを保持する。

`:abbrev-table-modiff`
: このプロパティはそのテーブルに abbrev が追加される度に増分されるカウンターを保持する。

37 スレッド

Emacs Lisp はスレッド *(thread)* と呼ばれる限定的な並行性 (concurrency) の形式を提供します。Emacs の与えられたインターフェース内のすべてのスレッドは同じメモリーを共有します。Emacs Lisp 内の並行性は “概ね協調的 (mostly cooperative)” であり、これは Emacs がスレッド間の実効を明確に定義された時間にだけ切り替えることを意味しています。しかし Emacs でのスレッドのサポートは将来よりきめの細かい並行性が可能な方法でデザインされており、正しいプログラムは協調的なスレッドに依存するべきではありません。

現在のところスレッドの切り替えはキーボード入力や非同期プロセスからの出力の待機中 (つまり `accept-process-output`の間)、ミューテックスのロックや `thread-join`のようなスレッドに関連するブロッキング処理の間での、`thread-yield`を介した明示的な要求時に発生します。

Emacs Lisp はスレッドの作成や制御、さらにスレッドの同期に有用なミューテックスや条件変数の作成や制御を行うプリミティブを提供します。

グローバル変数は Emacs Lisp のすべてのスレッドの間で共有されますがローカル変数 (`let`によるダイナミックなバインドはローカル) は異なります。スレッドはそれぞれ自身のカレントバッファー (Section 27.2 [Current Buffer], page 590 を参照) と自身のマッチデータ (Section 34.6 [Match Data], page 897 を参照) を所有します。

`let`によるバイディングは Emacs Lisp 実装により特別に処理されることに注意してください。`let`の使う以外にこの unwind(巻き戻し) と rewind(巻き戻すための巻き取り) の振る舞いを複製する方法はありません。たとえば `unwind-protect`を使用して `let`の独自実装で記述しても変数値をスレッド固有にアレンジすることはできません。

レキシカルバインディング (Section 12.10 [Variable Scoping], page 185 を参照) の場合には他の Emacs Lisp オブジェクト類似するものはクロージ (closure) です。クロージャ内のバインディングはクロージャを呼び出したすべてのスレッド間で共有されます。

37.1 基本的なスレッド関数

スレッドを作成したり待機することができます。スレッドを直接 exit することはできませんがカンレントスレッドは暗黙に exit でき、他のスレッドはシグナルを受け取ることができます。

`make-thread` *function* **&optional** *name* [Function]

*function*を呼び出す新たなスレッドの実行を作成する。*function*のリターン時にスレッドは exit する。

新たなスレッドは効力をもつローカル変数のバインディングが何もない状態で作成される。新たなスレッドのカレントバッファーはカレントスレッドから継承される。

*name*でスレッドに名前を与えることができる。これはデバッグと情報的な用途だけに使用される名前であり、Emacs にとって意味はない。*name*を与える場合には文字列でなければならない。

この関数は新たなスレッドをリターンする。

`threadp` *object* [Function]

この関数は *object*が Emacs スレッドを表すなら `t`、それ以外は `nil`をリターンする。

`thread-join` *thread* [Function]

*thread*が exit するかカレントスレッドがシグナルされるまでブロックする。これは *thread*関数の結果をリターンする。*thread*が exit 済みなら即座にリターンする。

`thread-signal` *thread error-symbol data* [Function]
`signal` (Section 11.7.3.1 [Signaling Errors], page 165 を参照) と同様だがシグナルはスレッド *thread*に送信される。*thread*がカレントスレッドなら、即座に `signal`を呼び出す。それ以外なら *thread*がカレントになり次第、シグナルを受信する。`mutex-lock`、`condition-wait`、`thread-join`の呼び出しにより *thread*がブロックされていたら `thread-signal`がロックを解除する。

*thread*がメインスレッドならシグナルは伝播されない。かわりにメインスレッド内のメッセージとして表れる。

`thread-yield` [Function]
実行可能な次のスレッドに実行を譲り渡す。

`thread-name` *thread* [Function]
`make-thread`で指定された *thread*の名前をリターンする。

`thread-live-p` *thread* [Function]
*thread*が生きていれば `t`、それ以外は `nil`をリターンする。スレッドは自身の関数がまだ実行中なら生きている。

`thread--blocker` *thread* [Function]
*thread*が待機中のオブジェクトをリターンする。これは主にデバッグを糸した関数であり、それを示すために “2 重ハイフン” の名前を付与してある。

*thread*が `thread-join`内でブロックされていたら待機対象のスレッドをリターンする。

*thread*が `mutex-lock`内でブロックされていたらミューテックスをリターンする。

*thread*が `condition-wait`内でブロックされていたら条件変数をリターンする。

それ以外なら `nil`をリターンする。

`current-thread` [Function]
カレントスレッドをリターンする。

`all-threads` [Function]
すべての生きたスレッドオブジェクトのリストをリターンする。呼び出しそれぞれにたいして新たなリストをリターンする。

`main-thread` [Variable]
この変数は Emacs 実行中のメインスレッド、スレッドサポートなしで Emacs コンパイル時には `nil`を保持する。

スレッドが実行したコードがハンドルされないエラーをシグナルすると、そのスレッドは exit します。それ以外のスレッドは以下の関数を使用してスレッドの exit を誘発したエラーフォームにアクセスできます。

`thread-last-error` **&optional** *cleanup* [Function]
この関数はエラーによりスレッドが exit した際に記録された最後のエラーフォームをリターンする。異常終了 (abnormal exit) した各スレッドが、以前にスレッドエラーで格納されたフォームを新たな値に上書きするので、アクセスできるのは最後のフォームのみ。*cleanup*が非 `nil`なら格納されたフォームを `nil`にリセットする。

37.2 ミューテックス

ミューテックス *(mutex)* は排他的なロックです。任意のタイミングにおいて 0、または 1 つのスレッドがミューテックスを所有できます。スレッドがミューテックスの取得を試みたときに別スレッドがすでにそのミューテックスを所有していたら、ミューテックスが利用可能になるまで取得を試みたスレッドはブロックされます。

Emacs Lisp のミューテックスは再帰的 *(recursive)* と呼ばれるタイプです。これはスレッドがミューテックスを何回も再帰的に所有できることを意味します。ミューテックスは所有された回数を保持していて、それら所有のそれぞれがリリースとペアになっていなければなりません。スレッドによるミューテックスの最後のリリースによりミューテックスは他のスレッドが潜在的に所有可能である非所有な状態へとリバートされます。

`mutexp` *object* [Function]
: この関数は *object*が Emacs のミューテックスを表していれば `t`、それ以外は `nil`をリターンする。

`make-mutex` **&optional** *name* [Function]
: 新たなミューテックスを作成してリターンする。*name*が指定されたらミューテックスの名前として与えられる (文字列でなければならない)。これはデバッグにたいする用途のみの名前であり Emacs にとって意味はない。

`mutex-name` *mutex* [Function]
: `make-mutex`で指定された *mutex*の名前をリターンする。

`mutex-lock` *mutex* [Function]
: これはスレッドが *mutex*を所有するか、スレッドが `thread-signal`の使用によりシグナルされるまでブロックする。スレッドが *mutex*をすでに所有していたら単にリターンする。

`mutex-unlock` *mutex* [Function]
: *mutex*をリリースする。そのスレッドが *mutex*を所有していなければエラーをシグナルする。

`with-mutex` *mutex body...* [Macro]
: このマクロはもっともシンプルかつ安全にミューテックスを保持しつつフォームを評価する方法である。これは *mutex*を取得して *body*を呼び出してから *mutex*をリリースする。*body*の結果をリターンする。

37.3 条件変数

条件変数 *(condition variable)* は何らかのイベントが発生するまでスレッドをブロックするための手段です。スレッドは別のスレッドが条件を通知した際に目覚めるように条件変数を待機できます。

条件変数はミューテックス、および概念的に何らかの条件に関連付けられます。正しく操作するために、ミューテックスを所有してから待機スレッドはループして、条件のテストを行い条件変数を待機しなければなりません。たとえば:

```
(with-mutex mutex
  (while (not global-variable)
    (condition-wait cond-var)))
```

ミューテックスはアトミック操作のために、ループは堅牢性のためのものです。これは偽の通知があるかもしれないからです。

同様にミューテックスは条件の通知前に所持されていなければなりません。ミューテックスを所有して条件に関連する変更を行い、それを通知するのが典型的かつ最良なアプローチです:

```
(with-mutex mutex
  (setq global-variable (some-computation))
  (condition-notify cond-var))
```

`make-condition-variable` *mutex* **&optional** *name* [Function]
*mutex*に関連付けられた条件変数を作成する。*name*が指定されたら条件変数の名前として与えられる (文字列でなければならない)。これはデバッグにたいする用途のみの名前であり Emacs にとって意味はない。

`condition-variable-p` *object* [Function]
この関数は *object*が条件変数を表していれば `t`、それ以外は `nil`をリターンする。

`condition-wait` *cond* [Function]
他のスレッドによる条件変数 *cond*の通知を待機する。この関数は条件変数の通知、または `thread-signal`の使用によりシグナルが送信されるまでブロックする。

ミューテックスに関連付けられた条件を所持しない `condition-wait`の呼び出しはエラーとなる。

`condition-wait`は待機中に関連付けられたミューテックスをリリースする。これは別スレッドによる条件通知するためのミューテックス所有を可能にする。

`condition-notify` *cond* **&optional** *all* [Function]
*cond*を通知する。呼び出し前に *cond*に関連付けられるミューテックスを所有しなければならない。通常は `condition-notify`により単一の待機中スレッドが目覚めさせられる。しかし *all*が非 `nil`なら *cond*を待機中のすべてのスレッドに通知される。

`condition-notify`は関連付けられたミューテックスを待機中はリリースする。これによりスレッドが条件を待機するためにミューテックスを所有することが可能になる。

`condition-name` *cond* [Function]
`make-condition-variable`に渡された *cond*の名前をリターンする。

`condition-mutex` *cond* [Function]
*cond*に関連付けられたミューテックスをリターンする。関連付けられたミューテックスは変更できないことに注意。

37.4 スレッドリスト

`list-threads`コマンドはカレントでアクティブなすべてのスレッドをリストします。その結果となるバッファー内では、それぞれのスレッドは `make-thread` (Section 37.1 [Basic Thread Functions], page 929 を参照) に渡された名前、または名前つきで作成されていなければ一意な内部識別子により識別される。バッファーの作成時 (または最終更新時) の各スレッドの状態、さらにその時点でスレッドがブロックされていたらブロックしているオブジェクトが表示されます。

`thread-list-refresh-seconds` [Variable]
`*Threads*`バッファーは自動的に毎秒 2 回更新される。この変数をカスタマイズしてリフレッシュ間隔を加減できる。

以下はレッドリストバッファーで利用できるコマンドです:

`b` ポイント位置のスレッドのバックトレースを表示する。これは `b`を押下した時点でスレッドが生成またはブロックされたコードを表示する。このバックトレースはスナップショットであることに注意。それ以降にスレッドが実行を再開して異なる状態になったり exit しているかもしれない。

バックトレースバッファーは自動的に更新されないので、スレッドのバックトレースバッファーでは、更新されたバックトレースを取得するために `g`を使用できる。バックトレースおよびそれらに機能する他のコマンドの説明は Section 18.1.7 [Backtraces], page 301 を参照のこと。

`s` ポイント位置のスレッドにシグナルを送信する。`s`の後に `q`をタイプすると quit、`e`をタイプするとエラーをシグナルする。スレッドはシグナルを処理するように実装されているかもしれないが、デフォルトではすべてのシグナルにたいして exit する。したがってターゲットとなるスレッドの再開方法を理解している場合のみこのコマンドを使用すること (必要なスレッドが kill されると Emacs セッションは不正に振る舞うかもしれない)。

`g` スレッドのリストと状態を更新する。

38 プロセス

オペレーティングシステムの用語ではプロセス *(process)* とはプログラムを実行できるスペースのことです。Emacs はプロセス内で実行されます。Emacs Lisp プログラムは別のプログラムをそれら自身のプロセス内で呼び出すことができます。これらは親プロセス *(parent process)* である Emacs プロセスのサブプロセス *(subprocesses)*、または子プロセス *(child processes)* と呼ばれます。

Emacs のサブプロセスは同期 *(synchronous)* か非同期 *(asynchronous)* であり、それはそれらが作成された方法に依存します。同期サブプロセスを作成した際には、Lisp プログラムは実行を継続する前にそのサブプロセスの終了を待機します。非同期サブプロセスを作成したときには、それを Lisp プログラムと並行して実行できます。この種のサブプロセスは Emacs では Lisp オブジェクトとして表現され、そのオブジェクトも “プロセス” と呼ばれています。Lisp プログラムはサブプロセスとのやり取りやサブプロセスの制御のためにこのオブジェクトを使用できます。たとえばシグナル送信、ステータス情報の取得、プロセス出力の受信やプロセスへ入力を送信することができます。

プログラムを実行するプロセスに加えて、Lisp プログラムは同一または他のマシン上で実行中のデバイスやプロセスにたいして、いくつかのタイプの接続をオープンできます。サポートされる接続タイプはネットワーク接続の TCP と UDP、シリアルポート接続、およびパイプ接続です。そのような接続はそれぞれプロセスオブジェクトとしても表現されます。

`processp` *object* [Function]
: この関数は、*object*が Emacs のプロセスオブジェクトを表すなら `t`、それ以外は `nil`をリターンする。プロセスオブジェクトはプログラム実行中のサブプロセスやサポートされた任意のタイプの接続を表すことができる。

カレント Emacs セッションのサブプロセスに加えて、そのマシン上で実行中の他のプロセスにアクセスすることもできます。Section 38.12 [System Processes], page 961 を参照してください。

38.1 サブプロセスを作成する関数

内部でプログラムを実行するサブプロセスを作成するために 3 つのプリミティブが存在します。それらの 1 つは `make-process`であり、これは非同期プロセスを作成してプロセスオブジェクトをリターンします (Section 38.4 [Asynchronous Processes], page 941 を参照)。他の 2 つは `call-process` と `call-process-region`です。これらは同期プロセスを作成してプロセスオブジェクとをリターンしません (Section 38.3 [Synchronous Processes], page 937 を参照)。特定のタイプのプロセスを実行するために、これらのプリミティブを利用するさまざまな高レベル関数が存在します。

同期プロセスと非同期プロセスについては、以降のセクションで説明します。この 3 つの関数はすべて類似した様式で呼び出されるので、ここではそれらに共通の引数について説明します。

すべての場合において、関数は実行するプログラムを指定します。ファイルが見つからなかったり実行できなければエラーがシグナルされます。ファイル名が相対的なら、検索するディレクトリーのリストは変数 `exec-path`に格納されています。Emacs は起動の際に環境変数 `PATH`の値にもとづいて `exec-path`を初期化します。`exec-path`内では標準的なファイル名構成要素 ‘`~`’、‘`.`’、‘`..`’は通常どおりに解釈されますが、環境変数の置換 (‘`$HOME`’等) は認識されません。それらの置換を行うには `substitute-in-file-name`を使用してください (Section 25.9.4 [File Name Expansion], page 559 を参照)。このリスト内で `nil`は `default-directory`を参照します。

プログラムの実行では指定された名前にサフィックスの追加を試みることもできます:

`exec-suffixes` [User Option]
この変数は指定されたプログラムファイル名への追加を試みるためのサフィックス (文字列) のリスト。指定されたとおりの名前を試みたいならリストに`""`を含めること。デフォルト値はシステム依存。

注意してください: 引数 *program*にはプログラムのファイル名だけが含まれて、コマンドライン引数を含めることはできない。これらを提供するために以下で説明する別の引数 *args*を使用しなければならない。

サブプロセス作成関数にはそれぞれ *buffer-or-name*引数があります。これはプログラムの出力の行き先を指定します。これはバッファーかバッファー名であるべきです。バッファー名の場合には、もしそのバッファーがまだ作成されていなければバッファーを作成します。`nil`を指定することもでき、その場合にはカスタム製のフィルター関数が出力を処理するのでなければ出力を破棄するよう指示します (Section 38.9.2 [Filter Functions], page 955 と Chapter 19 [Read and Print], page 331 を参照)。通常は出力がランダムに混在してしまうために、同一バッファーに複数プロセスの出力を送信するのは避けるべきです。同期プロセスにたいしてはバッファーのかわりにファイルに出力を送信できます (したがって対応する引数はより適切な *destination*という名前で呼ばれる)。デフォルトでは標準出力と標準エラーの両ストリームの行き先 (destination) は同じだが、3 つのプリミティブはすべてオプションで標準エラーストリームに別の行き先を指定できる。

これら 3 つのサブプロセス作成関数は、すべて実行するプロセスにコマンドライン引数を指定できます。`call-process`と `call-process-region`では、これらは`&rest`形式の引数 *args*で与えられます。`make-process`では実行するプログラムとコマンドライン引数はいずれも文字列のリストとして指定されます。コマンドライン引数はすべて文字列でなければならず、それらは別個の引数文字列としてプログラムに与えられます。文字列は指定されたプログラムに直接渡されるので、ワイルドカード文字やその他の shell 構文はこれらの文字列内では特別な意味をもちません。

サブプロセスはその環境を Emacs から継承しますが、`process-environment`でそれをオーバーラードするよう指定することができます。Section 40.3 [System Environment], page 1100 を参照してください。サブプロセスは自身のカレントディレクトリーを `default-directory`の値から取得します。

`exec-directory` [Variable]
この変数の値は GNU Emacs とともに配布されて、Emacs により呼び出されることを意図したプログラムを含むディレクトリーの名前 (文字列)。プログラム `movemail`はそのようなプログラムの例であり、Rmail は inbox から新しいメールを読み込むためにこのプログラムを使用する。

`exec-path` [User Option]
この変数の値はサブプロセス内で実行するためのプログラムを検索するためのディレクトリーのリスト。要素はそれぞれディレクトリーの名前 (文字列)、または`nil`のいずれか。`nil`はデフォルトディレクトリー (`default-directory`の値) を意味する。この検索の詳細は Section 25.6.6 [Locating Files], page 549 を参照のこと。

`exec-path`の値は、*program*引数が絶対ファイル名でないときに `call-process`と `start-process`により使用される。

一般的には `exec-path`を直接変更するべきではない。かわりに Emacs 起動前に環境変数 `PATH`が適切にセットされているか確認すること。`PATH`とは独立に `exec-path`の変更を試みると混乱した結果へと導かれ得る。

`exec-path` [Function]
この関数は変数 `exec-path`の拡張である。`default-directory`がリモートパスを示す場合には、この関数は対応するリモートホスト上でプログラム検索に使用するデジレクトリーのリストをリターンする。`default-directory`がローカルにあれば、この関数は単に変数 `exec-path`の値をリターンする。

38.2 shell 引数

Lisp プログラムが shell を実行して、ユーザーが指定したファイル名を含むコマンドを与える必要がある場合が時折あります。これらのプログラムは任意の有効なファイル名をサポート可能であるはずです。しかし shell は特定の文字を特別に扱い、それらの文字がファイル名に含まれていると shell を混乱させるでしょう。これらの文字を処理するためには関数 `shell-quote-argument`を使用します。

`shell-quote-argument` *argument* [Function]
この関数は実際のコンテンツが *argument*であるような引数を表す文字列を shell の構文でリターンする。リターン値を shell コマンドに結合して実行のためにそれを shell に渡すことにより、信頼性をもって機能するはずである。

この関数が正確に何を行うかはオペレーティングシステムに依存する。この関数はそのシステムの標準 shell の構文で機能するようデザインされている。非標準の shell を使用する場合には、この関数を再定義する必要があるだろう。Section 40.22 [Security Considerations], page 1131 を参照のこと。

```
;; この例は GNU および Unix システムでの挙動を示す
(shell-quote-argument "foo > bar")
     ⇒ "foo\\ \\>\\ bar"

;; この例は MS-DOS および MS-Windows での挙動を示す
(shell-quote-argument "foo > bar")
     ⇒ "\"foo > bar\""
```

以下は `shell-quote-argument`を使用して shell コマンドを構築する例:

```
(concat "diff -u "
        (shell-quote-argument oldfile)
        " "
        (shell-quote-argument newfile))
```

以下の 2 つの関数はコマンドライン引数の文字列のリストを単一の文字列に結合したり、単一の文字列を個別のコマンドライン引数のリストへ分割するために有用です。これらの関数は主にミニバッファーでのユーザー入力である Lisp 文字列を `make-process`、`call-process`や `start-process`に渡す文字列引数のリストへ変換したり、そのような引数のリストをミニバッファーやエコーエリアに表示するための Lisp 文字列に変換することを意図しています。(`call-process-shell-command`を使用して)shell が呼び出される場合には、引数を依然として `shell-quote-argument`で保護する必要があることに注意。`combine-and-quote-strings`は shell の評価から特殊文字を保護することを意図していない。

`split-string-and-unquote` *string* **&optional** *separators* [Function]
この関数は `split-string`(Section 4.3 [Creating Strings], page 52 を参照) が行うように、正規表現 *separators*にたいするマッチで *string*を部分文字列に分割する。さらに加えてその部分文字列からクォートを削除する。それから部分文字列のリストを作成してリターンする。

*separators*が省略または`nil`の場合のデフォルトは`"\\s-+"`であり、これは空白文字構文 (Section 35.2.1 [Syntax Class Table], page 907 を参照) をもつ 1 つ以上の文字にマッチする正規表現である。

この関数は 2 つのタイプのクォートをサポートする。1 つは文字列全体をダブルクォートで囲う`"..."`のようなクォートで、もう 1 つはバックスラッシュ'\'によるエスケープで文字を個別にクォートするタイプである。後者は Lisp 文字列内でも使用されるので、この関数はそれらも同様に扱うことができる。

`combine-and-quote-strings` *list-of-strings* **&optional** *separator* [Function]

この関数は *list-of-strings*の各文字を必要に応じてクォートして単一の文字列に結合する。これはさらに各文字ペアーの間に *separator*文字列も挿入する。*separator*が省略または`nil`の場合のデフォルトは`" "`。リターン値はその結果の文字列。

*list-of-strings*内のクォートを要する文字列には、部分文字列として *separator*を含むものが該当する。文字列のクォートはそれをダブルクォートで`"..."`のように囲う。もっとも単純な例では、たとえば個別のコマンドライン引数からコマンドをコンス (cons) する場合には、埋め込まれたブランクを含む文字列はそれぞれクォートされるだろう。

38.3 同期プロセスの作成

同期プロセス *(synchronous process)* の作成後、Emacs は継続する前にそのプロセスの終了を待機します。GNU や Unix[1] での Dired の起動が例です。プロセスは同期的なので、Emacs がそれにたいして何か行おうと試みる前にディレクトリーのリスト全体がバッファーに到着します。

同期サブプロセス終了を Emacs が待機する間に、ユーザーは *C-g*をタイプすることで quit が可能です。最初の *C-g*は`SIGINT`シグナルによりサブプロセスの kill を試みます。しかしこれは quit する前に実際にそのサブプロセスが終了されるまで待機します。その間にユーザーがさらに *C-g*をタイプするとそれは`SIGKILL`で即座にサブプロセスを kill して quit します (別プロセスにたいする kill が機能しない MS-DOS を除く)。Section 21.11 [Quitting], page 413 を参照してください。

同期サブプロセス関数はプロセスがどのように終了したかの識別をリターンします。

同期サブプロセスからの出力はファイルからのテキスト読み込みと同じように、一般的にはコーディングシステムを使用してデコードされます。`call-process-region`によりサブプロセスに送信された入力は、ファイルへのテキスト書き込みと同じようにコーディングシステムを使用してエンコードされます。Section 33.10 [Coding Systems], page 864 を参照してください。

`call-process` *program* **&optional** *infile destination display* **&rest** *args* [Function]

この関数は *program*を呼び出して完了するまで待機する。

サブプロセスのカレント作業ディレクトリー (CWD: current working directory) はカレントバッファーの`default-directory`がローカル (`unhandled-file-name-directory`により判断される) ならその値、それ以外は`"~"`。リモートディレクトリーでプロセスを実行したければ`process-file`を使用すること。

新たなプロセスの標準入力は *infile*が非`nil`ならファイル *infile*、それ以外なら null デバイス。引数 *destination*はプロセスの出力をどこに送るかを指定する。以下は可能な値:

バッファー そのバッファーのポイントの前に出力を挿入する。これにはプロセスの標準出力ストリームと標準エラーストリームの両方が含まれる。

[1] 他のシステムでは Emacs は`ls`の Lisp エミュレーションを使用します。Section 25.10 [Contents of Directories], page 565 を参照してください。

バッファー名 (文字列)
: その名前のバッファーのポイントの前に出力を挿入する。

`t`
: カレントバッファーのポイントの前に出力を挿入する。

`nil`
: 出力を破棄する。

0
: 出力を破棄してサブプロセス完了を待機せずに即座に `nil`をリターンする。

  この場合にはプロセスは Emacs と並列に実行可能なので真に同期的ではない。しかしこの関数リターン後は本質的にはすみやかに Emacs がサブプロセスを終了するという点から、これを同期的と考えることができる。

  MS-DOS は非同期サブプロセスをサポートせずこのオプションは機能しない。

(`:file` *file-name*)
: 指定されたファイルに出力を送信して、ファイルが既に存在すれば上書きする。

(*real-destination error-destination*)
: 標準出力ストリームを標準エラーストリームと分けて保持する。通常の出力は *real-destination*の指定にしたがって扱い、エラー出力は *error-destination*にしたがって処分する。*error-destination*が `nil`ならエラー出力の破棄、`t`なら通常の出力と混合することを意味して、文字列ならそれはエラー出力をリダイレクトするファイルの名前である。

  エラー出力先に直接バッファーを指定することはできない。ただしエラー出力を一時ファイルに送信して、サブプロセス終了時にそのファイルをバッファーに挿入すればこれを達成できる。

*display*が非 `nil`なら、`call-process`は出力の挿入にしたがってバッファーを再表示する (しかし出力のデコードに選択されたコーディングシステムが実データからエンコーディングを推論することを意味する `undecided`なら、非 ASCIIに一度遭遇すると再表示が継続不能になることがある。これを修正するのが困難な根本的理由が存在する。Section 38.9 [Output from Processes], page 953 を参照)。

それ以外なら関数 `call-process`は再表示を行わずに、通常のイベントに由来する Emacs の再表示時だけスクリーン上で結果が可視になります。

残りの引数 *args*はそのプログラムにたいしてコマンドライン引数を指定する文字列です。文字列はそれぞれ別個の引数として *program*に渡されます。

(待機するよう告げた場合には) `call-process`がリターンする値はプロセスが終了した理由を示します。この数字はそのサブプロセスの exit ステータスであり 0 が成功、それ以外のすべての値は失敗を意味します。シグナルによりそのプロセスが終了された場合には、`call-process`はそれを記述する文字列をリターンします。`call-process`に待機しないように指示した場合には `nil`をリターンします。

以下の例ではカレントバッファーは '`foo`'です。

```
(call-process "pwd" nil t)
     ⇒ 0

---------- Buffer: foo ----------
/home/lewis/manual
---------- Buffer: foo ----------
```

```
(call-process "grep" nil "bar" nil "lewis" "/etc/passwd")
     ⇒ 0

---------- Buffer: bar ----------
lewis:x:1001:1001:Bil Lewis,,,,:/home/lewis:/bin/bash

---------- Buffer: bar ----------
```

以下は `call-process`の使用例であり、このような使用例は `insert-directory`関数の定義内で見つけることができます:

```
(call-process insert-directory-program nil t nil switches
              (if full-directory-p
                  (concat (file-name-as-directory file) ".")
                file))
```

`process-file` *program* **&optional** *infile buffer display* **&rest** *args* [Function]

この関数は別プロセス内でファイルを同期的に処理する。これは `call-process`と似ているが、サブプロセスのカレントワーキングディレクトリーを指定する変数 `default-directory`の値にもとづいて、ファイル名ハンドラーを呼び出すかもしれない。

引数は `call-process`の場合とほとんど同様の方法で処理されるが以下の違いがある:

引数 *infile*、*buffer*、*display*のすべての組み合わせと形式をサポートしないファイル名ハンドラーがあるかもしれない。たとえば実際に渡された値とは無関係に、*display*が `nil`であるかのように振る舞うファイル名ハンドラーがいくつかある。他の例としては *buffer*引数で標準出力とエラー出力を分離するのをサポートしないかもしれないファイル名ハンドラーがいくつか存在する。

ファイル名ハンドラーが呼び出されると、1 つ目の引数 *program*にもとづいて実行するプログラムを決定する。たとえばリモートファイルにたいするハンドラーが呼び出されたと考えてみよ。その場合にはプログラムの検索に使用されるパスは `exec-path`とは異なるかもしれない。

2 つ目の引数 *infile*はファイル名ハンドラーを呼び出すかもしれない。そのファイル名ハンドラーは、`process-file`関数自身にたいして選択されたハンドラーと異なるかもしれない (たとえば `default-directory`がリモートホスト上にあり *infile*は別のリモートホスト上の場合があり得る。もしくは `default-directory`は普通だが *infile*はリモートホスト上にあるかもしれない).

*buffer*が `(`*real-destination error-destination*`)`という形式のリストであり、かつ *error-destination*がファイルの名前なら *infile*と同じ注意が適用される。

残りの引数 (*args*) はそのままプロセスに渡される。Emacs は *args*内で与えられたファイル名の処理に関与しない。混乱を避けるためには *args*内で絶対ファイル名を使用しないのが最善であり、`default-directory`からの相対ファイル名ですべてのファイルを指定するほうがよいだろう。そのような相対ファイル名の構築には関数 `file-relative-name`が有用。かわりにリモートホスト視点から見た絶対ファイル名を取得するために `file-local-name`も使用できる (Section 25.12 [Magic File Names], page 569 を参照)。

`process-file-side-effects` [Variable]

この変数は `process-file`呼び出しがリモートファイルを変更するかどうかを示す。

この変数はデフォルトでは `process-file`呼び出しがリモートホスト上の任意のファイルを潜在的に変更し得ることを意味する `t`に常にセットされる。`nil`にセットされた際には、リモートファイル属性のキャッシュにしたがうことによりファイル名ハンドラーの挙動を最適化できる可能性がある。

この変数は決して `setq`ではなく、常に let バインディングでのみ変更すること。

`call-process-region` *start end program* **&optional** *delete destination display* **&rest** *args* [Function]

この関数は *start*から *end*のテキストを、実行中のプロセス *program*に標準入力として送信する。これは *delete*が非`nil`なら送信したテキストを削除する。これは出力をカレントバッファーの入力箇所に挿入するために、*destination*を`t`に指定している際に有用。

引数 *destination*と *display*はサブロセスからの出力にたいして何を行うか、および出力の到着にともない表示を更新するかどうかを制御する。詳細は上述の`call-process`の説明を参照のこと。*destination*が整数の 0 なら`call-process-region`は出力を破棄して、サブプロセス完了を待機せずに即座に`nil`をリターンする (これは非同期サブプロセスがサポートされる場合、つまり MS-DOS 以外でのみ機能する)。

残りの引数 *args*はそのプログラムにたいしてコマンドライン引数を指定する文字列です。

`call-process-region`のリターン値は`call-process`の場合と同様。待機せずにリターンするよう指示した場合には`nil`、数字か文字列ならそれはサブプロセスが終了した方法を表す。

以下の例ではバッファー '`foo`'内の最初の 5 文字 (単語 '`input`') を標準入力として、`call-process-region`を使用して`cat`ユーティリティを実行する。`cat`は自身の標準入力を標準出力へコピーする。引数 *destination*が`t`なので出力はカレントバッファーに挿入される。

```
---------- Buffer: foo ----------
input*
---------- Buffer: foo ----------

(call-process-region 1 6 "cat" nil t)
     ⇒ 0

---------- Buffer: foo ----------
inputinput*
---------- Buffer: foo ----------
```

たとえば`shell-command-on-region`コマンドは、以下のような方法で`call-shell-region`を使用する:

```
(call-shell-region
 start end
 command                ; shell コマンド
 nil                    ; region を削除しない
 buffer)                ; 出力を bufferに出力
```

`call-process-shell-command` *command* **&optional** *infile destination display* [Function]

この関数は shell コマンド *command*を非同期に実行する。他の引数は`call-process`の場合と同様に処理される。古い呼び出し規約は *display*の後に任意個数の追加引数を許容して、これは *command*に結合される。これはまだサポートされるものの使用しないことを強く推奨する。

`process-file-shell-command` *command* **&optional** *infile destination display* [Function]

この関数は`call-process-shell-command`と同様だが内部的に`process-file`を使用する点が異なる。`default-directory`に依存して *command*はリモートホスト上でも実行可能。古い呼び出し規約は *display*の後に任意個数の追加引数を許容して、これは *command*に結合される。これはまだサポートされるものの使用しないことを強く推奨する。

`call-shell-region` *start end command* **&optional** *delete destination* [Function]

この関数は *start*と *end*の間のテキストを、*command*を実行するシェルの標準入力として送信する。これはプロセスがシェルであるような`call-process-region`と類似している。引

数 delete、destination、およびリターン値は call-process-regionと同様。この関数は追加の引数を受け付けないことに注意。

shell-command-to-string *command* [Function]
この関数は shell コマンドとして *command* (文字列) を実行してコマンドの出力を文字列としてリターンする。

process-lines *program* **&rest** *args* [Function]
この関数は *program*を実行して完了を待機して、出力を文字列のリストとしてリターンする。リスト内の各文字列はプログラムのテキスト出力の 1 つの行を保持する。各行の EOL 文字 (行末文字) は取り除かれる。*program*の後の引数 *args*はそのプログラム実行に際して、コマンドライン引数を指定する文字列。

*program*が非 0 の exit ステータスで exit すると、この関数はエラーをシグナルする。

この関数は call-processを呼び出すことにより機能して、プログラムの出力は call-processの場合と同じ方法でデコードされる。

38.4 非同期プロセスの作成

このセクションでは非同期プロセス *(asynchronous process)* を作成する方法について説明します。非同期プロセスは作成後に Emacs と並列して実行され、Emacs は以降のセクション (Section 38.7 [Input to Processes], page 950 と Section 38.9 [Output from Processes], page 953 を参照) で説明する関数を使用してプロセスとコミュニケーションができます。プロセスコミュニケーションは部分的に非同期なだけであることに注意してください。Emacs はこれらの関数を呼び出したときだけプロセスとのデータを送受信できます。

非同期プロセスは *pty(pseudo-terminal:* 疑似端末*)*、または *pipe* のいずれかを通じて制御されます。pty か pipe の選択はデフォルトでは変数 process-connection-type (以下参照) の値にもとづいてプロセス作成時に行われます。Shell モードのように利用可能ならユーザーから可視なプロセスには、プロセスと子プロセス間でジョブ制御 (*C-c*、*C-z*等) が可能であり、インタラクティブなプログラムでは pty を端末デバイスとして扱いますが pipe はそのような機能をサポートしないので pty が通常は好まれます。しかし内部的な目的のために Lisp プログラムが使用する (サブプロセスとユーザーの相互作用が要求されない) サブプロセスでは、サブプロセスと Lisp プログラム間で大量データのやり取りが要求される場合には、pipe がより効率的なので pipe の使用が最良な場合がままあります。さらに多くのシステムでは pty の合計数に制限があり、それを浪費するのは得策ではありません。

make-process **&rest** *args* [Function]
この関数は非同期サブプロセスを開始するための基本的な低レベルなプリミティブである。これはサブプロセスを表すプロセスオブジェクトをリターンする。以下で説明するより高レベルな start-processと比較すると、この関数はキーワード引数を受け取り、より柔軟であり、単独の呼び出しでプロセスフィルターやセンチネルを指定できる。

引数 *args*は keyword/argument ペアのリスト。キーワードの省略は値 nilでそれを指定することと常に等価。以下は意味のあるキーワード:

:name *name*
プロセス名として文字列 *name*を使用する。その名前のプロセスがすでに存在すれば、(‘<1>’、... の追加により) 一意となるように *name*を修正する。

:buffer *buffer*
プロセスバッファーとして *buffer*を使用する。値が nilなら、そのサブプロセスには何のバッファーも関連付けられない。

:command *command*
: プロセスのコマドラインとして *command*を使用する。値はプログラムの実行可能ファイル名で始まり、後にプログラムの引数として与える文字列が続くリストであること。リストの最初の要素が`nil`なら、Emacs は新たな擬似端末 (pty) を作成して、実際には何もプログラムを実行せずに入出力を *buffer*に関連付ける。この場合には残りのリスト要素は無視される。

:coding *coding*
: *coding*がシンボルなら、それはその接続にたいする読み取りと書き込みの両方で使用するコーディングシステムを指定する。*coding*がコンスセル (`decoding . encoding`) なら読み取りに *decoding*、書き込みに *encoding*が使用される。プログラムに書き込むデータのエンコーディングに使用されるコーディングシステムは、コマンドライン引数のエンコーディングにも使用される (しかしプログラム自身にたいしてファイル名を別のファイル名にエンコードすることはない。Section 33.10.2 [Encoding and I/O], page 866 を参照)。

  *coding*が`nil`なら、デフォルトのコーディングシステム検出ルールを適用する。Section 33.10.5 [Default Coding Systems], page 871 を参照のこと。

:connection-type *type*
: サブプロセスとの対話に使用するデバイスタイプを初期化する。値には pty を使用する`pty`、pipe を使用する`pipe`、または変数`process-connection-type`の値から継承されるデフォルトを使用する`nil`を指定できる。このパラメーターと`process-connection-type`の値は`:stderr`パラメーターに非`nil`が指定されている場合には無視される。その場合にはタイプは常に`pipe`になる。このパラメーターは pty が利用できないシステム (MS-Windows) でも無視されて無条件で pipe が使用される。

:noquery *query-flag*
: プロセス query フラグを *query-flag*に初期化する。Section 38.11 [Query Before Exit], page 961 を参照のこと。

:stop *stopped*
: *stopped*が与えられた場合には`nil`でなければならない。非`nil`値の使用はすべてエラーとなる。それ以外の場合には`:stop`は無視される。これは pipe プロセスのような他のプロセスタイプへの互換性のために維持されている。非同期サブプロセスが stopped 状態で開始されることはあり得ない。

:filter *filter*
: プロセスフィルターを *filter*に初期化する。未指定ならデフォルトフィルターが提供されるが、これは後からオーバーライドできる。Section 38.9.2 [Filter Functions], page 955 を参照のこと。

:sentinel *sentinel*
: プロセスセンチネルを *sentinel*に初期化する。未指定ならデフォルトセンチネルが使用されるが、これは後からオーバーライドできる。Section 38.10 [Sentinels], page 959 を参照のこと。

:stderr *stderr*
: プロセスの標準エラーに *stderr*を割り当てる。値が非`nil`ならバッファー、または以下で説明する`make-pipe-process`で作成された pipe のいずれかであるこ

と。*stderr*が `nil`なら標準エラーを標準出力と合成して、両者を *buffer*か *filter* に送信する。

*stderr*がバッファーなら Emacs は pipe プロセス、標準エラープロセス *(standard error process)* を作成する。このプロセスはデフォルトフィルター (Section 38.9.2 [Filter Functions], page 955 を参照)、センチネル (Section 38.10 [Sentinels], page 959 を参照)、コーディングシステム (Section 33.10.5 [Default Coding Systems], page 871 を参照) をもつ。その一方で、自身の query-on-exit フラグとして *query-flag*を使用する (Section 38.11 [Query Before Exit], page 961 を参照)。このプロセスは*stderr*バッファーに関連づけられて、そこに出力 (メインプロセスの標準エラー) を送信する (Section 38.9.1 [Process Buffers], page 953 を参照)。

*stderr*が pipe プロセスなら、Emacs はそれを新たなプロセス用の標準エラープロセスとして使用する。

:file-handler *file-handler*

*file-handler*が非 `nil`なら、カレントバッファーの `default-directory`にたいするファイル名ハンドラーを探して、プロセスを作成するためにそのファイル名ハンドラーを呼び出す。そのようなハンドラーがなければ、*file-handler*が `nil`であるかのように処理する。

実際の接続情報で修正されたオリジナルの引数リストは `process-contact`を通じて利用できる。

サブプロセスのカレント作業ディレクトリー (CWD: current working directory) はカレントバッファーの `default-directory`がローカル (`unhandled-file-name-directory`により判断される) ならその値、それ以外は~。リモートディレクトリーでプロセスを実行したければ、`make-process`に`:file-handler t`を渡せばよい。この場合には、カレントのワーキングディレクトリー (CWD) は `default-directory`の (`file-local-name`で決定される) ローカル部分となる。

ファイル名ハンドラーの実装に依存して、結果となるプロセスオブジェクトへの *filter*や *sentinel* の適用が不可能かもしれない。`:stderr`引数は pipe プロセスではあり得ないので、ファイル名ハンドラーはこれにたいして pipe プロセスをサポートしない。`:stderr`引数としてバッファーは許されており、バッファーのコンテンツは pipe プロセスを使用することなく表示される。Section 38.9.2 [Filter Functions], page 955 および Section 38.9.4 [Accepting Output], page 957 を参照のこと。

いくつかのファイルハンドラーは `make-process`をサポートしないかもしれない。そのような場合には、この関数は何も行わずに `nil`をリターンする。

`make-pipe-process` **&rest** *args* [Function]

この関数は子プロセスにアタッチ可能な双方向の pipe を作成する。これは `make-process`の `:stderr`キーワードと併用することで有用。この関数はプロセスオブジェクトをリターンする。

引数 *args*は keyword/argument ペアのリスト。キーワードの省略はそのキーワードに値 `nil`を指定することと常に等価。

以下は意味のあるキーワード。

:name *name*

プロセス名として文字列 *name*を使用する。`make-process`の場合のように、一意にするために必要に応じて変更され得る。

:buffer *buffer*
: プロセスバッファーとして *buffer*を使用する。

:coding *coding*
: *coding*がシンボルなら、それはその接続にたいする読み取りと書き込みの両方で使用するコーディングシステムを指定する。*coding*がコンスセル (`decoding . encoding`) なら読み取りに *decoding*、書き込みに *encoding*が使用される。

  *coding*が `nil`なら、デフォルトのコーディングシステム検出ルールを適用する。Section 33.10.5 [Default Coding Systems], page 871 を参照のこと。

:noquery *query-flag*
: プロセス query フラグを *query-flag*に初期化する。Section 38.11 [Query Before Exit], page 961 を参照のこと。

:stop *stopped*
: *stopped*が非 `nil`なら停止状態でプロセスを開始する。停止状態では pipe プロセスは入力データを受け取らないが出力データは送信できる。停止状態は `stop-process`でセットして `continue-process`でクリアーされる (Section 38.8 [Signals to Processes], page 951 を参照)。

:filter *filter*
: プロセスフィルターを *filter*に初期化する。未指定ならデフォルトフィルターが提供されるが後で変更できる。Section 38.9.2 [Filter Functions], page 955 を参照のこと。

:sentinel *sentinel*
: プロセスセンチネルを *sentinel*に初期化する。未指定ならデフォルトセンチネルが使用されるが後で変更できる。Section 38.10 [Sentinels], page 959 を参照のこと。

実際の接続情報で修正されたオリジナルの引数リストは `process-contact`を通じて利用できる。

`start-process` *name buffer-or-name program* **&rest** *args* [Function]

この関数は `call-process`の類似したインターフェースを提供する、`make-process`周辺の高レベルのラッパー。これは新たに非同期サブプロセスを作成して、指定された *program*の実行をその内部で開始する。これは Lisp で新たなサブプロセスを意味するプロセスオブジェクトをリターンする。引数 *name*はプロセスオブジェクトの名前を指定する。`make-process`の場合のように、一意な名前となるように必要に応じて修正する。バッファー *buffer-or-name*はそのプロセスに関連付けるバッファー。

*program*が `nil`なら Emacs は疑似端末 (pty) を新たにオープンして、サブプロセスを新たに作成することなく pty の入力と出力を *buffer-or-name*に関連付ける。この場合には残りの引数 *args*は無視される。

残りの *args*はサブプロセスにコマンドライン引数を指定する文字列。

以下の例では 1 つ目のプロセスを開始して 100 秒間実行 (というよりは sleep) される。その間に 2 つ目のプロセスを開始して、一意性を保つために '`my-process<1>`'という名前が与えられる。これは 1 つ目のプロセスが終了する前にバッファー '`foo`'の最後にディレクトリーのリストを挿入する。その後に 2 つ目のプロセスは終了して、その旨のメッセージがバッファーに

挿入される。さらに遅れて 1 つ目のプロセスが終了して、バッファーに別のメッセージが挿入される。

```
(start-process "my-process" "foo" "sleep" "100")
     ⇒ #<process my-process>

(start-process "my-process" "foo" "ls" "-l" "/bin")
     ⇒ #<process my-process<1>>

---------- Buffer: foo ----------
total 8336
-rwxr-xr-x 1 root root 971384 Mar 30 10:14 bash
-rwxr-xr-x 1 root root 146920 Jul  5  2011 bsd-csh
...
-rwxr-xr-x 1 root root 696880 Feb 28 15:55 zsh4

Process my-process<1> finished

Process my-process finished
---------- Buffer: foo ----------
```

`start-file-process` *name buffer-or-name program* **&rest** *args* [Function]

`start-process`と同じようにこの関数は非同期サブプロセスを開始して、その内部で *program* を実行してそのプロセスオブジェクトをリターンする。

`start-process`との違いは、この関数が `default-directory`の値にもとづいてファイル名ハンドラーを呼び出すかもしれないという点である。このハンドラーはローカルホスト上、あるいは `default-directory`に応じたリモートホスト上で *program*を実行すること。後者の場合には、`default-directory`のローカル部分はそのプロセスのワーキングディレクトリーになる。

この関数は *program*や *args*の残りにたいしてファイル名ハンドラーの呼び出しを試みない。*program*は *args*のいずれかがリモートファイル構文 (Section 25.12 [Magic File Names], page 569 を参照) を使用する場合には、`file-local-name`を通じて実行することにより、それらの名前を `default-directory`に相対的な名前やリモートホスト上でローカルにファイルを識別する名前に変換しなければならないことが理由。

そのファイル名ハンドラーの実装によっては、リターン結果のプロセスオブジェクトに `process-filter`や `process-sentinel`を適用することができないかもしれない。Section 38.9.2 [Filter Functions], page 955 と Section 38.10 [Sentinels], page 959 を参照のこと。

いくつかのファイル名ハンドラーは `start-file-process`をサポートしないかもしれない (たとえば `ange-ftp-hook-function`関数)。そのような場合には、この関数は何も行わずに `nil`をリターンする。

`start-process-shell-command` *name buffer-or-name command* [Function]

この関数は `start-process`と同様だが、指定された *command*の実行に shell を使用する点が異なる。引数 *command*は shell コマンド文字列。変数 `shell-file-name`はどの shell を使用するかを指定する。

`make-process`や `start-process`でプログラムを実行せずに shell を通じて実行することの要点は、引数内のワイルドカード展開のような shell 機能を利用可能にするためである。そのためにはコマンド内に任意のユーザー指定引数を含めるなら、任意の特別な shell 文字が shell での特別な意味をもたないように、まず `shell-quote-argument`でそれらをクォートするべ

きである。Section 38.2 [Shell Arguments], page 936 を参照のこと。ユーザー入力にもとづいたコマンド実行時には当然セキュリティ上の影響も考慮するべきである。

`start-file-process-shell-command` *name buffer-or-name command* [Function]

この関数は `start-process-shell-command`と似ているが、内部的に `start-file-process`を使用する点が異なる。これにより `default-directory`に応じてリモートホスト上でも *command*を実行できる。

`process-connection-type` [Variable]

この変数は非同期サブプロセスと対話するために使用するデバイスタイプを制御する。これが非 `nil`の場合には利用可能なら pty、それ以外なら pipe が使用される。

`process-connection-type`の値は `make-process`や `start-process`の呼び出し時に効果を発揮する。そのためにこれらの関数の呼び出し前後でこの変数をバインドすることにより、サブプロセスとやり取りする方法を指定できる。

この変数の値は非 `nil`値の `:stderr`パラメーターで `make-process`が呼び出された際には無視される。この場合には Emacs は pipe を使用してプロセスと対話する。pty が利用不能 (MS-Windows) な場合にも無視される。

```
(let ((process-connection-type nil))  ; pipe を使用
  (start-process ...))
```

与えられたサブプロセスが実際には pipe と pty のどちらを取得したかを判断するには関数 `process-tty-name`を使用する (Section 38.6 [Process Information], page 947 を参照)。

38.5 プロセスの削除

プロセス削除 *(deleting a process)* とは Emacs をサブプロセスから即座に切断することです。プロセスは終了後に自動的に削除されますが即座に削除される必要はありません。任意のタイミングで明示的にプロセスを削除できます。終了したプロセスが自動的に削除される前に明示的に削除しても害はありません。実行中のプロセスの削除はプロセス (もしあれば子プロセスにも) を終了するためにシグナルを送信してプロセスセンチネルを呼び出します。Section 38.10 [Sentinels], page 959 を参照してください。

プロセスが削除される際、そのプロセスオブジェクト自体はそれを参照する別の Lisp オブジェクトが存在する限り継続し続けます。プロセスオブジェクトに作用するすべての Lisp プリミティブはプロセスの削除を受け入れますが、I/O を行ったりシグナルを送信するプリミティブはエラーを報告するでしょう。プロセスマークは通常はプロセスからの出力がバッファーに挿入される箇所となる、以前と同じ箇所をポイントし続けます。

`delete-exited-processes` [User Option]

この変数は、(`exit`呼び出しやシグナルにより) 終了したプロセスの自動的な削除を制御する。これが `nil`ならユーザーが `list-processes`を実行するまでプロセスは存在し続けて、それ以外なら exit 後に即座に削除される。

`delete-process` *process* [Function]

この関数はプロセスがプログラムを実行していたら `SIGKILL`シグナルで kill することによりプロセスを削除する。引数はプロセス、プロセスの名前、バッファー、バッファーの名前かもしれない (バッファーやバッファー名なら `get-buffer-process`がリターンするプロセスを意味する)。実行中のプロセスに `delete-process`を呼び出すことによりプロセスを終了してプロセス状態を更新して即座にセンチネルを実行する。そのプロセスがすでに終了していれば、

delete-process呼び出しはプロセス状態、または (遅かれ早かれ発生するであろう) プロセスセンチネルの実行に影響を与えない。

プロセスオブジェクトがネットワーク接続、シリアル接続、pipe 接続を表す場合には状態はclosed、それ以外ならそのプロセスがexit 済みでなければsignalに変更される。Section 38.6 [Process Information], page 947 を参照のこと。

```
(delete-process "*shell*")
     ⇒ nil
```

38.6 プロセスの情報

プロセスの状態に関する情報をリターンする関数がいくつかあり。

`list-processes` **&optional** *query-only buffer* [Command]

このコマンドは、すべての生きたプロセスのリストを表示する。加えてこれは最後に、状態が'Exited'か'Signaled'だったすべてのプロセスを削除する。このコマンドはnilをリターンする。

プロセスはメジャーモードがProcess Menu モードであるような、*Process List*という名前のバッファーに表示される (オプション引数 *buffer*で他の名前を指定していない場合)。

*query-only*が非nilなら、query フラグが非nilのプロセスだけをリストする。Section 38.11 [Query Before Exit], page 961 を参照のこと。

`process-list` [Function]

この関数は削除されていないすべてのプロセスのリストをリターンする。

```
(process-list)
     ⇒ (#<process display-time> #<process shell>)
```

`get-process` *name* [Function]

この関数は *name* (文字列) というプロセス、存在しなければnilをリターンする。引数 *name* はプロセスオブジェクトでもよく、この場合にはそれがリターンされる。

```
(get-process "shell")
     ⇒ #<process shell>
```

`process-command` *process* [Function]

この関数は *process*を開始するために実行されたコマンドをリターンする。これは文字列のリストで 1 つ目の文字列は実行されたプログラム、残りの文字列はそのプログラムに与えられた引数。ネットワーク接続、シリアル接続、pipe 接続にたいしてはnil (プロセスは実行中) かt (プロセスは停止中) のいずれか。

```
(process-command (get-process "shell"))
     ⇒ ("bash" "-i")
```

`process-contact` *process* **&optional** *key no-block* [Function]

この関数はネットワーク接続、シリアル接続、pipe 接続がセットアップされた方法に関する情報をリターンする。*key*がnilならネットワーク接続には (*hostname service*)、シリアル接続には (*port speed*)、pipe 接続にはtをリターンする。普通の子プロセスにたいしては、この関数は *key*がnilで呼び出されると常にtをリターンする。

*key*がtなら値はその接続、サーバー、シリアルポート、または pipe についての完全な状態情報、すなわち make-network-process、make-serial-process、または make-pipe-process内で指定されるキーワードと値のリストとなる。ただしいくつかの値については、指定した値のかわりにカレント状態を表す値となる。

ネットワークプロセスにたいしては以下の値が含まれる (完全なリストは `make-network-process`を参照):

`:buffer` 値にはプロセスのバッファーが割り当てられる。

`:filter` 値にはプロセスのフィルター関数が割り当てられる。Section 38.9.2 [Filter Functions], page 955 を参照のこと。

`:sentinel`
値にはプロセスのセンチネル関数が割り当てられる。Section 38.10 [Sentinels], page 959 を参照のこと。

`:remote` 接続にたいしては内部的なフォーマットによるリモートピアーのアドレス。

`:local` 内部的なフォーマットによるローカルアドレス。

`:service` この値はサーバーでは *service*に `t`を指定すると実際のポート番号。

`make-network-process`内で明示的に指定されていなくても`:local`と`:remote`は値に含まれる。

シリアル接続については `make-serial-process`、キーのリストは `serial-process-configure`を参照のこと。pipe 接続については `make-pipe-process`を参照のこと。

*key*がキーワードなら、この関数はそのキーワードに対応する値をリターンする。

*process*がまだ完全にセットアップされていない非ブロッキングネットワークストリームなら、この関数はセットアップされるまでブロックする。オプションの *no-block*パラメーターが与えられると、この関数はブロックせずに `nil`をリターンする。

`process-id` *process* [Function]
この関数は *process*の PIDをリターンする。これは同じコンピューター上でカレント時に実行中の他のすべてのプロセスからプロセス *process*を区別するための整数。プロセスの PIDはプロセスの開始時にオペレーティングシステムのカーネルにより選択されて、そのプロセスが存在する限り定数として保たれる。この関数はネットワーク接続、シリアル接続、pipe 接続には `nil`をリターンする。

`process-name` *process* [Function]
この関数は *process*の名前を文字列としてリターンする。

`process-status` *process-name* [Function]
この関数は *process-name*の状態を文字列でリターンする。引数 *process-name*はプロセス、バッファー、またはプロセス名 (文字列) でなければならない。

実際のサブプセスにたいして可能な値は:

`run` 実行中のプロセス。

`stop` 停止しているが継続可能なプロセス。

`exit` exit したプロセス。

`signal` 致命的なシグナルを受信したプロセス。

`open` オープンされたネットワーク接続、シリアル接続、または pipe 接続。

`closed` クローズされたネットワーク接続、シリアル接続、または pipe 接続。一度クローズされた接続は、たとえ同じ場所にたいして新たな接続をオープンすることができたとしても再度オープンすることはできない。

`connect` 完了を待つ非ブロッキング接続。

`failed` 完了に失敗した非ブロッキング接続。

`listen` listen 中のネットワークサーバー。

`nil` *process-name*が既存のプロセス名でない場合。

```
(process-status (get-buffer "*shell*"))
     ⇒ run
```

ネットワーク接続、シリアル接続、pipe 接続にたいして `process-status`は `open`、`stop`、または `closed`いずれかのシンボルをリターンする。`closed`は相手側が接続をクローズしたか、あるいは Emacs が `delete-process`を行なったことを意味する。値 `stop`はその接続で `stop-process`が呼び出されたことを意味する。

`process-live-p` *process* [Function]

この関数は *process*がアクティブなら、非 `nil`をリターンする。状態が `run`、`open`、`listen`、`connect`、`stop`のプロセスはアクティブとみなされる。

`process-type` *process* [Function]

この関数はネットワーク接続やサーバーにたいしては `network`、シリアルポート接続にたいしては `serial`、pipe 接続にたいしては `pipe`、プログラム実行用に作成されたサブプロセスにたいしては `real`というシンボルをリターンする。

`process-exit-status` *process* [Function]

この関数は *process*の exit ステータス、またはプロセスを kill したシグナル番号をリターンする (いずれであるかの判定には `process-status`の結果を使用)。*process*がまだ終了していなければ値は 0。すでに close されたネットワーク接続、シリアル接続、pipe 接続についての値は接続の close が正常か異常かによって 0 か 256 のいずれかとなる。

`process-tty-name` *process* [Function]

この関数は *process*が Emacs との対話に使用する端末名、pty ではなく pipe を使用する場合には `nil`をリターンする (Section 38.4 [Asynchronous Processes], page 941 の `process-connection-type`を参照)。*process*がリモートホスト上で実行中のプログラムを表す場合には、プロセスの `remote-tty`プロパティとしてリモートホスト上でそのプログラムに使用される端末名が提供される。*process*がネットワーク接続、シリアル接続、pipe 接続を表す場合には値は `nil`。

`process-coding-system` *process* [Function]

この関数は *process*からの出力のデコードに使用するコーディングシステムと、*process*への入力のエンコードに使用するコーディングシステムを記述するコンスセル (*decode* . *encode*) をリターンする (Section 33.10 [Coding Systems], page 864 を参照)。

`set-process-coding-system` *process* **&optional** *decoding-system* *encoding-system* [Function]

この関数は *process*にたいする後続の入出力に使用するコーディングシステムを指定する。これはサブプロセスの出力のデコードに *decoding-system*、入力のエンコードに *encoding-system* を使用する。

すべてのプロセスには、そのプロセスに関連するさまざまな値を格納するために使用できるプロパティリストもあります。

`process-get` *process propname* [Function]
この関数は *process*のプロパティ *propname*の値をリターンする。

`process-put` *process propname value* [Function]
この関数は *process*のプロパティ *propname*の値に *value*をセットする。

`process-plist` *process* [Function]
この関数は *process*のプロセス plist をリターンする。

`set-process-plist` *process plist* [Function]
この関数は *process*のプロセス plist に *plist*をセットする。

38.7 プロセスへの入力の送信

非同期サブプロセスは Emacs により入力が送信されたときに入力を受信して、それはこのセクション内の関数で行われます。これを行うには入力を送信するプロセスと送信するための入力データを指定しなければなりません。サブプロセスがプログラムを実行していたら、データはプログラムの標準入力として出現します。接続にたいしては、データは接続されたデバイスかプログラムに送信されます。

オペレーティングシステムには pty のバッファーされた入力にたいして制限をもつものがいくつかあります。それらのシステムでは、Emacs は他の文字列の間に定期的かつ強制的に EOFを送信します。ほとんどのプログラムにたいして、これらの EOFは無害です。

サブプロセスの入力はテキストをファイルに書き込むときと同じように、通常はサブプロセスが受信する前、コーディングシステムを使用してエンコードされます。どのコーディングシステムを使用するかを指定するには `set-process-coding-system`を使用できます (Section 38.6 [Process Information], page 947 を参照)。それ以外の場合には、非 `nil`なら `coding-system-for-write` がコーディングシステムとなり、さもなくばデフォルトのメカニズムがコーディングシステムを決定します (Section 33.10.5 [Default Coding Systems], page 871 を参照)。

入力バッファーが一杯のために、システムがプロセスからの入力を受け取ることができないことがあります。これが発生したときには、送信関数はしばらく待機してサブプロセスの出力を受け取り、再度送信を試みます。これは保留となっている更なる入力を読み取り、バッファーに空きを作る機会をサブプロセスに与えます。これはフィルター (現在実行中のものを含む)、センチネル、タイマーの実行も可能にするのでコードを記述する際はそれを考慮してください。

以下の関数では *process*引数はプロセス、プロセス名、またはバッファー、バッファー名 (`get-buffer-process`で取得されるプロセス)、`nil`はカレントバッファーのプロセスを意味します。

`process-send-string` *process string* [Function]
この関数は *string*のコンテンツを標準入力として *process*に送信する。たとえばファイルをリストする Shell バッファーを作成するには:

```
(process-send-string "shell<1>" "ls\n")
    ⇒ nil
```

`process-send-region` *process start end* [Function]
この関数は *start*と *end*で定義されるリージョンのテキストを標準入力として *process*に送信する。

*start*と *end*が、カレントバッファー内の位置を示す整数かマーカーでなければエラーがシグナルされる (いずれかの大小は重要ではない)。

`process-send-eof` **&optional** *process* [Function]

この関数は *process*が入力内の EOF (end-of-file) を見ることを可能にする。EOFはすべての送信済みテキストの後になる。この関数は *process*をリターンする。

```
(process-send-eof "shell")
     ⇒ "shell"
```

`process-running-child-p` **&optional** *process* [Function]

この関数は *process*が接続ではない実際のサブプロセスであり、端末の制御を自身の子プロセスに与えたかどうかを示す。これが真なら関数は *process*のフォアグラウンドプロセスグループの数値 ID、これが真ではないと Emacs が判断すれば `nil`をリターンする。これが真かどうかを Emacs が判断できなければ値は `t`。*process*がネットワーク接続、シリアル接続、pipe 接続、またはサブプロセスが非アクティブなら関数はエラーをシグナルする。

38.8 プロセスへのシグナルの送信

サブプロセスへのシグナル送信 *(sending a signal)* はプロセス活動に割り込む手段の 1 つです。異なる複数のシグナルがあり、それぞれが独自に意味をもちます。シグナルのセットとそれらの意味はオペレーティングシステムにより定義されます。たとえばシグナル `SIGINT`はユーザーが *C-c*をタイプしたか、それに類似する何かが発生したことを意味します。

各シグナルはサブプロセスに標準的な効果をもちます。ほとんどのシグナルはサブプロセスを kill しますが、かわりに実行を停止 (や再開) するものもいくつかあります。ほとんどのシグナルはオプションでプログラムでハンドル (処理) することができます。プログラムがそのシグナルをハンドルする場合には、その影響についてわたしたちは一般的には何も言うことはできません。

このセクション内の関数を呼び出すことにより明示的にシグナルを送信できます。Emacs も特定のタイミングで自動的にシグナルを送信します。バッファーの kill により、それに関連するプロセスには `SIGHUP`シグナル、Emacs の kill により残されたすべてのプロセスに `SIGHUP`シグナルが送信されます (`SIGHUP`は通常はユーザーが “hung up the phone”、電話を切った、つまり接続を断ったことを示す)。

シグナル送信関数はそれぞれ *process*と *current-group*いう 2 つのオプション引数を受け取ります。

引数 *process*はプロセス、プロセス名、バッファー、バッファー名、または `nil`のいずれかでなければなりません。バッファーやバッファー名は `get-buffer-process`を通じて得られるプロセスを意味します。`nil`はカレントバッファーに関連付けられたプロセスを意味します。`stop-process` and `continue-process`を除いて、*process*がプロセスを識別しない、あるいはネットワーク接続、シリアル接続、pipe 接続を表す場合にはエラーがシグナルされます。

引数 *current-group*は、Emacs のサブプロセスとしてジョブ制御 shell(job-control shell) を実行中の場合に異なる処理を行うためのフラグです。これが非 `nil`なら、そのシグナルは Emacs がサブプロセスとの対話に使用する端末のカレントプロセスグループに送信されます。そのプロセスがジョブ制御 shell なら、これはその shell のカレントの sub ジョブになります。*current-group*が `nil`なら、そのシグナルは Emacs 自身のサブプロセスのプロセスグループに送信されます。そのプロセスがジョブ制御 shell なら、それは shell 自身になります。*current-group*が `lambda`なら、端末を所有するもののそれ自身は shell でない場合にはプロセスグループにシグナルを送信します。

サブプロセスとの対話に pipe が使用されている際には、オペレーティングシステムが pipe の区別をサポートしないのでフラグ *current-group*に効果はありません。同じ理由により pipe が使用されていればジョブ制御 shell は機能しないでしょう。Section 38.4 [Asynchronous Processes], page 941 の `process-connection-type`を参照してください。

`interrupt-process` **&optional** *process current-group* [Function]
この関数はシグナル `SIGINT`を送信することによりプロセス *process*に割り込む。Emacs 外部では interrupt character(割り込み文字。いくつかのシステムでは通常は `C-c`、それ以外のシステムでは `DEL`) をタイプすることによりシグナルが送信される。引数 *current-group*が非 `nil` のときは、Emacs がサブプロセスと対話する端末上で `C-c`がタイプされたと考えることができる。

`kill-process` **&optional** *process current-group* [Function]
この関数はシグナル `SIGKILL`を送信することにより、プロセス *process*を kill する。このシグナルは即座にサブプロセスを kill してサブプロセスでハンドルすることはできない。

`quit-process` **&optional** *process current-group* [Function]
この関数はプロセス *process*にシグナル `SIGQUIT`を送信する。これは Emacs 外部では quit character(通常は `C-\`) により送信されるシグナル。

`stop-process` **&optional** *process current-group* [Function]
この関数は指定した *process*を停止する。それがプログラムを実行中の実際のサブプロセスなら、そのサブプロセスにシグナル `SIGTSTP`を送信する。*process*がネットワーク接続、シリアル接続、pipe 接続を表す場合には、この関数はその接続から到達するデータのハンドリングを抑制する。ネットワークサーバーでは、これは新たな接続を accept しないことを意味する。通常の実行の再開には `continue-process`を使用すること。

ジョブ制御をもつシステム上の Emacs 外部では stop character(通常は `C-z`) が `SIGTSTP`シグナルを送信する。*current-group*が非 `nil`なら、この関数をサブプロセスとの対話に Emacs が使用する端末上で `C-z`がタイプされたと考えることができる。

`continue-process` **&optional** *process current-group* [Function]
この関数はプロセス *process*の実行を再開する。それがプログラムを実行中の実際のサブプロセスなら、そのサブプロセスにシグナル `SIGCONT`を送信する。この関数は *process*が以前に停止されたとみなす。*process*がネットワーク接続、シリアル接続、pipe 接続を表す場合には、この関数はその接続から到達するデータのハンドリングを再開する。シリアル接続ではプロセス停止中に到達したデータは失われるかもしれない。

`signal-process` *process signal* [Command]
この関数はプロセス *process*にシグナルを送信する。引数 *signal*はどのシグナルを送信するかを指定する。これは整数、または名前がシグナルであるようなシンボルであること。

*process*引数にはシステムプロセス ID (整数) を指定できる。これにより Emacs の子プロセス以外のプロセスにシグナルを送信できる。Section 38.12 [System Processes], page 961 を参照のこと。

非ローカルな非同期プロセスへのシグナル送信が必要になることがあります。これは独自の `interrupt-process`実装を記述することにより可能です。それからその関数を `interrupt-process-functions`に追加する必要があります。

`interrupt-process-functions` [Variable]
この変数は `interrupt-process`用に呼び出される関数のリスト。関数の引数は `interrupt-process`にたいする引数と同じ。これらの関数はいずれかが非 `nil`をリターンするまでリスト順に呼び出される。このリスト上で常に最後になるデフォルトの関数は `internal-default-interrupt-process`。

これは Tramp が `interrupt-process`を実装するメカニズムである。

38.9 プロセスからの出力の受信

非同期サブプロセスが自身の標準出力に書き込んだ出力はフィルター関数 *(filter function)* と呼ばれる関数に渡されます。デフォルトのフィルター関数は単に出力をバッファーに挿入します。このバッファーをプロセスに関連付けられたバッファーと呼びます (Section 38.9.1 [Process Buffers], page 953 を参照)。プロセスがバッファーをもたなければデフォルトフィルターは出力を破棄します。

サブプロセスが標準エラーストリームに書き込む場合には、デフォルトではエラー出力もフィルター関数に渡されて処理されます。Emacs がサブプロセスとの対話に疑似端末 (pty: pseudo-TTY) を使用する場合には、疑似端末は出力チャンネルを 1 つしかもてないので、サブプロセスの標準出力ストリームと標準エラーストリームを区別するのは不可能です。それらのストリームの出力を区別して保ちたい場合は、たとえば `start-process-shell-command`や類似コマンドを通じて適当な shell コマンドを使用することにより、いずれか 1 つをファイルにリダイレクトする必要があります。

エラー出力先を標準出力から区別するためには、かわりに非 `nil`の`:stderr`パラメーターで `make-process` (Section 38.4 [Asynchronous Processes], page 941 を参照) を呼び出すことができます。この場合には、サブプロセスとの対話に Emacs は pipe を使用することになります。

サブプロセス終了時に Emacs は保留中の出力を読み取って、その後そのサブプロセスからの出力の読み取りを停止します。したがってそのサブプロセスに生きた子プロセスがあり、まだ出力を生成するような場合には、Emacs はその出力を受け取らないでしょう。

サブプロセスからの出力は Emacs が待機している間、端末入力読み取り時 (関数 `waiting-for-user-input-p`、Section 21.10 [Waiting], page 412 の `sit-for`と `sleep-for`、Section 38.9.4 [Accepting Output], page 957 の `accept-process-output`、およびプロセスへのデータ送信関数 (Section 38.7 [Input to Processes], page 950 を参照) のみ到着可能です。これは並列プログラミングで普遍的に悩みの種であるタイミングエラーの問題を最小化します。たとえば安全にプロセスを作成して、その後でのみプロセスのバッファーやフィルター関数を指定できます。その間にあるコードが待機するプリミティブを何も呼び出さなければ、完了するまで到着可能な出力はありません。

`process-adaptive-read-buffering` [Variable]

いくつかのシステムでは Emacs がサブプロセスの出力を読み取る際に出力データを非常に小さいブロックで読み取るために、結果として潜在的に非常に貧弱なパフォーマンスとなることがある。この挙動は変数 `process-adaptive-read-buffering`を非 `nil`値 (デフォルト) にセットして拡張することにより改善し得る。これにより、そのようなプロセスからの読み取りを自動的に遅延して、Emacs が読み取りを試みる前に出力がより多く生成されるようになる。

38.9.1 プロセスのバッファー

プロセスは関連付けられたバッファー *(associated buffer)* をもつことができます (通常はもつ)。これは普通の Emacs バッファーであり、2 つの目的のために使用されます。1 つはプロセスからの出力の格納、もう 1 つはプロセスを kill する時期を判断するためです。通常の習慣では任意の与えられたバッファーにたいして関連付けられるプロセスは 1 つだけなので、処理対象のプロセスを識別するためにそのバッファーを使用することもできます。プロセス使用の多くはプロセスに送信する入力を編集するためにもこのバッファーを使用しますが、これは Emacs Lisp の組み込みではありません。

デフォルトでは、プロセスの出力は関連付けられたバッファーに挿入されます (カスタムフィルター関数の定義により変更可能。Section 38.9.2 [Filter Functions], page 955 を参照)。出力を挿入する位置は `process-mark`により決定されます。これは正に挿入されたテキストの終端にポイントを更新します。通常 (常にではない) は `process-mark`はバッファーの終端になります。

プロセスに関連付けられたバッファーを kill することによりプロセスも kill されます。そのプロセスの `process-query-on-exit-flag`が非 `nil`なら、Emacs はまず確認を求めます

(Section 38.11 [Query Before Exit], page 961 を参照)。この確認は関数 `process-kill-buffer-query-function`により行われて、これは `kill-buffer-query-functions`から実行されます (Section 27.10 [Killing Buffers], page 603 を参照)。

`process-buffer` *process* [Function]

この関数は指定された *process*の関連付けられたバッファーをリターンする。

```
(process-buffer (get-process "shell"))
     ⇒ #<buffer *shell*>
```

`process-mark` *process* [Function]

この関数は *process*にたいするプロセスマーカーをリターンする。これはプロセスからの出力をどこに挿入するかを示すマーカー。

*process*がバッファーをもたなければ、`process-mark`は存在しない場所を指すマーカーをリターンする。

デフォルトのフィルター関数はプロセス出力の挿入場所の決定にこのマーカーを使用して、挿入したテキストの後にポイントを更新する。連続するバッチ出力が連続して挿入されるのはこれが理由。

カスタムフィルター関数はこのマーカーを通常は同じ方式で使用すること。`process-mark`を使用するフィルター関数の例は [Process Filter Example], page 956 を参照のこと。

ユーザーにプロセスバッファー内でプロセスに送信するための入力を期待する際には、プロセスマーカーは以前の出力から新たな入力を区別する。

`set-process-buffer` *process buffer* [Function]

この関数は *process*に関連付けられたバッファーに *buffer*をセットする。*buffer*が `nil`ならプロセスはバッファーに関連付けられない。

`get-buffer-process` *buffer-or-name* [Function]

この関数は *buffer-or-name*で指定されるバッファーに関連付けられた、削除されていないプロセスをリターンする。そのバッファーに複数のプロセスが関連付けられている場合には、この関数はいずれか 1 つ (現在のところもっとも最近作成されたプロセスだがこれを期待しないこと) を選択する。プロセスの削除 (`delete-process`を参照) により、そのプロセスはこの関数がリターンするプロセスとしては不適格となる。

同一のバッファーに複数のプロセスを関連付けるのは、通常は悪いアイデアである。

```
(get-buffer-process "*shell*")
     ⇒ #<process shell>
```

プロセスのバッファーを kill することにより、`SIGHUP`シグナルでサブプロセスを kill してプロセスを削除する (Section 38.8 [Signals to Processes], page 951 を参照)。

プロセスのバッファーがウィンドウに表示されている場合には、プロセスが出力をスクリーンのサイズに適応させるのと同様に、Lisp プログラムでウィンドウのサイズにプロセス出力を適応させるようにプロセスに指示したいと思うでしょう。以下の関数によりプロセスにたいしてこの種の情報をやり取りできます。しかしすべてのシステムが基礎となる機能をサポートする訳ではないので、コマンドライン引数や環境変数を通じたフォールバックを提供するのが最良です。

`set-process-window-size` *process height width* [Function]

*process*にたいして、その論理ウィンドウサイズが文字単位で *width*と *height*のサイズであることを告げる。関数がこの情報をプロセスとやり取りすることに成功したら `t`、それ以外は `nil`をリターンする。

プロセスに関連付けられたバッファーを表示するウィンドウがサイズを変更された際には、影響を受けるプロセスはその変更にたいして通知される必要があります。デフォルトではウィンドウ構成 (window configuration) が変更されると、ウィンドウにバッファーが表示されている各プロセスにかわり、プロセスのバッファーを表示するすべてのウィンドウのうち最小のサイズのウィンドウを引数として、Emacs が自動的に `set-process-window-size`を呼び出します。これはバッファーが少なくとも 1 つのウィンドウに表示されているプセスそれぞれにたいして、変数`window-adjust-process-window-size-function`の値である関数を呼び出すように指定する`window-configuration-change-hook` (Section 28.28 [Window Hooks], page 691 を参照) を通じて機能します。この変数をセットすることにより、この振る舞いをカスマイズできます。

`window-adjust-process-window-size-function` [User Option]

この変数の値はプロセスとプロセスのバッファーを表示するウィンドウのリストという 2 つの引数を受け取る関数であること。その関数が呼び出される際には、そのプロセスのバッファーがカレントバッファーとなる。関数は `set-process-window-size`の呼び出しを通じて渡される論理プロセスウィンドウ (logical process window) を記述するコンスセル (*`width . height`*) をリターンすること。関数は `nil`をリターンすることもでき、Emacs はこの場合にはそのプロセスにたいして `set-process-window-size`を呼び出さない。

この変数にたいして Emacs は 2 つの事前定義された値を提供する。1 つは `window-adjust-process-window-size-smallest`であり、これはプロセスのバッファーを表示するウィンドウのすべてのサイズから最小のサイズもう 1 つの `window-adjust-process-window-size-largest`は最大のサイズをリターンする。より複雑な方式には独自の関数を記述すること。

この変数はバッファーローカルにできる。

プロセスが`adjust-window-size-function`プロパティ(Section 38.6 [Process Information], page 947 を参照) をもつ場合には、その値は`window-adjust-process-window-size-function`のグローバル値とバッファーロール値をオーバーライドします。

38.9.2 プロセスのフィルター関数

プロセスのフィルター関数 *(filter function)* は、関連付けられたプロセスからの標準出力を受信します。そのプロセスのすべての出力はそのフィルターに渡されます。デフォルトのフィルターは単にプロセスバッファーに直接出力します。

デフォルトではプロセス用のエラー出力がもしあれば、プロセス作成時にプロセスの標準エラーストリームが標準出力から分離されていなければフィルター関数に渡されます。Emacs は特定の関数の呼び出し中のみフィルター関数を呼び出します。Section 38.9 [Output from Processes], page 953 を参照してください。フィルターによりこれらの関数のいずれかが呼び出されると、フィルターが再帰的に呼び出されるかもしれないことに注意してください。

フィルター関数は関連付けられたプロセス、およびそのプロセスから正に受信した出力である文字列という 2 つの引数を受け取らなければなりません。関数はその後に出力にたいして何であれ自由に行うことができます。

quit は通常はフィルター関数内では抑制されます。さもないとコマンドレベルでの *`C-g`*のタイプ、またはユーザーコマンドの quit は予測できません。フィルター関数内部での quit を許可したければ`inhibit-quit`を `nil`にバインドしてください。ほとんどの場合において、これを行う正しい方法はマクロ `with-local-quit`です。Section 21.11 [Quitting], page 413 を参照してください。

フィルター関数の実行中にエラーが発生すると、フィルター開始時に実行中だったプログラムが何であれ実行を停止しないように自動的に catch されます。しかし `debug-on-error`が非 `nil`ならエ

ラーは catch されません。これにより Lisp デバッガーを使用したフィルター関数のデバッグが可能になります。Section 18.1 [Debugger], page 296 を参照してください。

多くのフィルター関数は時折 (または常に)、デフォルトフィルターの動作を真似てプロセスのバッファーにその出力を挿入します。そのようなフィルター関数は確実にカレントバッファーの保存と、(もし異なるなら) 出力を挿入する前に正しいバッファーを選択して、その後に元のバッファーをリストアする必要があります。またそのバッファーがまだ生きているか、プロセスマーカーを更新しているか、そしていくつかのケースにおいてはポイントの値を更新しているかもチェックするべきです。以下はこれらを行う方法です:

```
(defun ordinary-insertion-filter (proc string)
  (when (buffer-live-p (process-buffer proc))
    (with-current-buffer (process-buffer proc)
      (let ((moving (= (point) (process-mark proc))))
        (save-excursion
          ;; テキストを挿入してプロセスマーカーを進める
          (goto-char (process-mark proc))
          (insert string)
          (set-marker (process-mark proc) (point)))
        (if moving (goto-char (process-mark proc)))))))
```

新たなテキスト到着時にフィルターが強制的にプロセスバッファーを可視にするために `with-current-buffer`構成の直前に以下のような行を挿入できます:

```
(display-buffer (process-buffer proc))
```

以前のポイント位置と関係なく新たな出力の終端にポイント位置を強制するためには、例から変数 `moving`を削除して無条件で `goto-char`を呼び出してください。これはウィンドウポイントの移動では必要ないことに注意してください。デフォルトのフィルターは実際にはウィンドウポイントを含むすべてのマーカーを移動する `insert-before-markers`を使用します。これは無関係のマーカーを移動するかもしれないので、一般的にはウィンドウポイントを明示的に移動するか、挿入タイプを `t` (Section 28.19 [Window Point], page 671 を参照) にセットしたほうがよいでしょう。

フィルター関数の実行中には、Emacs が自動的にマッチデータの保存とリストアを行うことに注意してください。Section 34.6 [Match Data], page 897 を参照してください。

フィルターへの出力は任意のサイズの chunk で到着する可能性があります。同じ出力を連続して 2 回生成するプログラムは一度に 200 文字を 1 回のバッチで送信して、次に 40 文字を 5 回のバッチで送信するかもしれません。フィルターが特定のテキスト文字列をサブプロセスの出力から探す場合には、それらの文字列が 2 回以上のバッチ出力を横断するケースに留意して処理してください。これを行うには受信したテキストを一時的なバッファーに挿入してから検索するのが 1 つの方法です。

`set-process-filter` *process filter* [Function]
この関数は *process*にフィルター関数 *filter*を与える。*filter*が `nil`なら、そのプロセスにたいしてプロセスバッファーにプロセス出力を挿入するデフォルトフィルターを与える。

`process-filter` *process* [Function]
この関数は *process*のフィルター関数をリターンする。

そのプロセスの出力を複数のフィルターに渡す必要がある場合には、既存のフィルターに新たなフィルターを組み合わせるために `add-function`を使用できる。Section 13.11 [Advising Functions], page 227 を参照のこと。

以下はフィルター関数の使用例:

```
(defun keep-output (process output)
   (setq kept (cons output kept)))
     ⇒ keep-output
```

```
(setq kept nil)
     ⇒ nil
(set-process-filter (get-process "shell") 'keep-output)
     ⇒ keep-output
(process-send-string "shell" "ls ~/other\n")
     ⇒ nil
kept
     ⇒ ("lewis@slug:$ "
"FINAL-W87-SHORT.MSS    backup.otl              kolstad.mss~
address.txt             backup.psf              kolstad.psf
backup.bib~             david.mss               resume-Dec-86.mss~
backup.err              david.psf               resume-Dec.psf
backup.mss              dland                   syllabus.mss
"
"#backups.mss#          backup.mss~             kolstad.mss
")
```

38.9.3 プロセス出力のデコード

Emacs が直接マルチバイトバッファーにプロセス出力を書き込む際には、プロセス出力のコーディングシステムに応じて出力をデコードします。コーディングシステムが `raw-text`か `no-conversion` なら Emacs は `string-to-multibyte`を使用してユニバイト出力をマルチバイトに変換して、その結果のマルチバイトテキストを挿入します。

どのコーディングシステムを使用するかは `set-process-coding-system`を使用して指定できます (Section 38.6 [Process Information], page 947 を参照)。それ以外では `coding-system-for-read`が非 `nil`ならそのコーディングシステム、`nil`ならデフォルトのメカニズムが使用されます (Section 33.10.5 [Default Coding Systems], page 871 を参照)。プロセスのテキスト出力に null バイトが含まれる場合には、Emacs はそれにたいしてデフォルトでは `no-conversion`を使用します。この挙動を制御する方法については Section 33.10.3 [Lisp and Coding Systems], page 867 を参照してください。

警告: データからコーディングシステムを判断する `undecided`のようなコーディングシステムは、非同期サブプロセスの出力にたいして完全な信頼性をもって機能しません。これは Emacs が到着に応じて非同期サブプロセスの出力をバッチで処理する必要があるからです。Emacs は 1 つのバッチが到着するたびに正しいコーディングシステムを検出しなければならずこれは常に機能するわけではありません。したがって可能であれば文字コード変換と EOL 変換の両方を決定するコーディングシステムつまり `latin-1-unix`、`undecided`、`latin-1`のようなコーディングシステムを指定してください。

Emacs がプロセスフィルター関数を呼び出す際には、そのプロセスのフィルターのコーディングシステムに応じて Emacs はプロセス出力をマルチバイト文字列、またはユニバイト文字列で提供します。Emacs はプロセス出力のコーディングシステムに応じて出力をデコードします。これは `binary` や `raw-text`のようなコーディングシステムを除いて、通常はマルチバイト文字列を生成します。

38.9.4 プロセスの出力を受け取る

非同期サブプロセスからの出力は、通常は Emacs が時間の経過や端末入力のような、ある種の外部イベントを待機する間だけ到着します。特定のポイントで出力の到着を明示的に許可したり、あるいはプロセスからの出力が到着するまで待機することでさえ、Lisp プログラムでは有用な場合が時折あります。

`accept-process-output` **&optional** *process seconds millisec just-this-one* [Function]

この関数はプロセスからの保留中の出力を Emacs が読み取ることを許す。この出力はプロセスのフィルター関数により与えられる。この関数は *process*が非 `nil`なら *process*から何らかの出力を受け取るか *process*が接続を close するまでリターンしない。

引数 *seconds*と *millisec*によりタイムアウトの長さを指定できる。前者は秒単位、後者はミリ秒単位でタイムアウトを指定する。この 2 つの秒数は、互いに足し合わせることによりタイムアウトを指定して、その秒数経過後はサブプロセスの出力が存在しなくてもリターンする。

*seconds*に浮動小数点数を指定することにより秒を少数点で指定できるので引数 *millisec*は時代遅れ (であり使用するべきではない)。*seconds*が 0 ならこの関数は保留中の出力が何であれ受け取り待機しない。

*process*がプロセスで引数 *just-this-one*が非 `nil`ならプロセスからの出力だけが処理され、そのプロセスからの出力を受信するかタイムアウトとなるまで他のプロセスの出力は停止される。*just-this-one*が整数ならタイマーの実行も抑制される。この機能は一般的には推奨されないが、音声合成のような特定のアプリケーションにとっては必要かもしれない。

関数 `accept-process-output`は *process*、*process*が `nil`なら何らかのプロセスからから出力を取得したら非 `nil`をリターンする。これは対応する接続にバッファーされたデータが含まれていれば、たとえプロセスの exit 後にも発生し得る。この関数はタイムアウトが発生したり出力の到着前に接続が close されると `nil`をリターンする。

プロセスからの接続にバッファーデータが含まれる場合には、プロセスの exit 後でも `accept-process-output`が非 `nil`をリターンするかもしれません。したがって、たとえ以下のようなループでも:

```
;; このループにはバグがある
(while (process-live-p process)
  (accept-process-output process))
```

これは *process*からすべての出力を読み取ることが頻繁にあり、接続にまだデータが含まれている間に `process-live-p`が `nil`をリターンすると競合条件をもつとともに何らかのデータが失われるかもしれません。以下のようなループを記述するほうがよいでしょう:

```
(while (accept-process-output process))
```

`make-process`に非 `nil`の *stderr*を渡すと、標準エラープロセスをもつことになる。Section 38.4 [Asynchronous Processes], page 941 を参照のこと。この場合にはメインプロセスからのプロセス出力の待機は、標準エラープロセスからの出力を待機しない。プロセスからすべての標準出力と標準エラーを確実に受け取るためには、以下のコードを使用する:

```
(while (accept-process-output process))
(while (accept-process-output stderr-process))
```

この方法でリモートホスト上で実行中のプロセスからの保留中の標準エラーを読み取ることはできません。

38.9.5 プロセスとスレッド

スレッドは比較的に新しく Emacs Lisp に追加されたものであり、ダイナミックバインドが `accept-process-output`と組み合わせて使用される方法のために、デフォルトではプロセスはそれを作成したスレッドにロックされます。プロセスがスレッドにロックされた場合には、プロセスの出力はそのスレッドだけが受け取ることができます。

Lisp プログラムはプロセスがロックされたスレッドがどれかを指定したり、あるいは Emacs にプロセスのアンロックを指示することができ、この場合にはプロセスの出力を任意のスレッドが受け取ることができます。与えられたプロセスから出力を待機できるのは一度に 1 つのスレッドだけです。1 つのスレッドが一度出力を待機すると、プロセスは `accept-process-output`か `sit-for`がリターンするまで一時的にロックされます。

スレッドが exit すると、それにロックされたすべてのプロセスがアンロックされます。

`process-thread` *process* [Function]
: *process*がロックされているスレッドをリターンする。*process*がロックされていなければ `nil` をリターンする。

`set-process-thread` *process thread* [Function]
: *process*をロックするスレッドを *thread*にセットする。*thread*は `nil`でもよく、この場合にはプロセスはアンロックされる。

38.10 センチネル: プロセス状態の変更の検知

プロセスセンチネル *(process sentinel:* プロセス番兵*)* とは、(Emacs により送信されたか、そのプロセス自身の動作が原因で送信された) プロセスを終了、停止、継続するシグナルを含む、何らかの理由により関連付けられたプロセスの状態が変化した際には常に呼び出される関数のことです。プロセスが exit する際にもプロセスセンチネルが呼び出されます。センチネルはイベントが発生したプロセスとイベントのタイプを記述する文字列という 2 つの引数を受け取ります。

プロセスにたいして何もセンチネル関数が指定されていなければ、プロセスのバッファーにプロセス名とイベントを記述する文字列とともにメッセージを挿入するデフォルトのセンチネル関数を使用します。

イベントを記述する文字列は以下のいずれかのような外見をもちます:

- `"finished\n"`.
- `"deleted\n"`.
- `"exited abnormally with code` *`exitcode`* `(core dumped)\n"`.
  “core dumped” の部分はオプションであり、プロセスがコアをダンプした場合のみ出現する。
- `"failed with code` *`fail-code`*`\n"`.
- `"`*`signal-description`* `(core dumped)\n"`.
  *signal-description*は `SIGKILL`にたいする`"killed"`のようなシステム依存の説明テキスト。
  “core dumped” の部分はオプションであり、プロセスがコアをダンプした場合のみ出現する。
- `"open from` *`host-name`*`\n"`.
- `"open\n"`.
- `"connection broken by remote peer\n"`.

センチネルは Emacs が (端末入力や時間経過、またはプロセス出力を) 待機している間だけ実行されます。これは他の Lisp プログラムの途中のランダムな箇所で実行されるセンチネルが原因となるタイミングエラーを無視します。プログラムはセンチネルが実行されるように、`sit-for`や `sleep-for`(Section 21.10 [Waiting], page 412 を参照)、または `accept-process-output`(Section 38.9.4 [Accepting Output], page 957 を参照) を呼び出すことにより待機することができます。Emacs はコマンドループが入力を読み取る際にもセンチネルの実行を許可します。`delete-process`は実行中のプログラムを終了させる際にセンチネルを呼び出します。

Emacs は 1 つのプロセスのセンチネル呼び出しの理由のために複数のキューを保持しません。これはカレント状態と変化があった事実だけを記録します。したがって非常に短い間隔で連続して状態に 2 つの変化があった場合には、一度だけセンチネルが呼び出されます。しかしプロセスの終了は常に正確に 1 回センチネルを実行するでしょう。これは終了後にプロセス状態が再び変更されることはないからです。

Emacs はプロセスセンチネル実行の前にプロセスからの出力をチェックします。プロセス終了によりセンチネルが一度実行されると、そのプロセスから更なる出力は到着しません。

プロセスのバッファーに出力を書き込むセンチネルは、そのバッファーがまだ生きているかチェックするべきです。死んだバッファーへの挿入を試みるとエラーになるでしょう。そのバッファーがすでに死んでいれば `(buffer-name (process-buffer` *process*`))`は `nil`をリターンします。

quit は通常はセンチネル内では抑制されます。さもないとコマンドレベルでの *C-g*のタイプ、またはユーザーコマンドの quit は予測できません。センチネル内部での quit を許可したければ `inhibit-quit`を `nil`にバインドしてください。ほとんどの場合において、これを行う正しい方法はマクロ `with-local-quit`です。Section 21.11 [Quitting], page 413 を参照してください。

センチネルの実行中にエラーが発生した場合には、センチネル開始時に実行中だったプログラムが何であれ実行を停止しないように自動的に catch されます。しかし `debug-on-error`が非 `nil`ならエラーは catch されません。これにより Lisp デバッガーを使用したセンチネルのデバッグが可能になります。Section 18.1 [Debugger], page 296 を参照してください。

センチネルの実行中にはセンチネルが再帰的に実行されないように、プロセスセンチネルは一時的に `nil`にセットされます。この理由によりセンチネルが新たにセンチネルを指定することはできません。

センチネル実行中には Emacs が自動的にマッチデータの保存とリストアを行うことに注意してください。Section 34.6 [Match Data], page 897 を参照してください。

`set-process-sentinel` *process sentinel* [Function]

この関数は *sentinel*を *process*に関連付ける。*sentinel*が `nil`なら、そのプロセスはプロセス状態変更時にプロセスのバッファーにメッセージを挿入するデフォルトのセンチネルをもつことになるだろう。

プロセスセンチネルの変更は即座に効果を発揮する。そのセンチネルは実行される予定だがまだ呼び出されておらず、かつ新たなセンチネルを指定した場合には、最終的なセンチネル呼び出しには新たなセンチネルが使用されるだろう。

```
(defun msg-me (process event)
   (princ
     (format "Process: %s had the event '%s'" process event)))
(set-process-sentinel (get-process "shell") 'msg-me)
     ⇒ msg-me
(kill-process (get-process "shell"))
     ⊣ Process: #<process shell> had the event 'killed'
     ⇒ #<process shell>
```

`process-sentinel` *process* [Function]

この関数は *process*のセンチネルをリターンする。

あるプロセス状態の変化を複数のセンチネルに渡す必要がある場合には、既存のセンチネルと新たなセンチネルを組み合わせるために `add-function`を使用できます。Section 13.11 [Advising Functions], page 227 を参照してください。

`waiting-for-user-input-p` [Function]
この関数はセンチネルやフィルター関数の実行中に、もし Emacs がセンチネルやフィルター関数呼び出し時にユーザーのキーボード入力を待機していたら非 `nil`、そうでなければ `nil`をリターンする。

38.11 exit 前の問い合わせ

Emacs が exit する際にはすべてのサブプロセスを終了します。プログラムを実行しているサブプロセスには `SIGHUP`を送信して、接続は単に close されます。それらのサブプロセスはさまざまな処理を行っているかもしれないので、Emacs は通常ユーザーにたいしてそれらを終了しても大丈夫かどうか確認を求めます。各プロセスは query(問い合わせ) のためのフラグをもち、これが非 `nil`なら Emacs はプロセスを kill して exit する前に確認を行うべきであることを示します。query フラグにたいするデフォルトは `t`で、これは問い合わせを行うことを意味しています。

`process-query-on-exit-flag` *process* [Function]
これは *process*の query フラグをリターンする。

`set-process-query-on-exit-flag` *process flag* [Function]
この関数は *process*の query フラグを *flag*にセットする。これは *flag*をリターンする。

以下は shell プロセス上で問い合わせを回避するために `set-process-query-on-exit-flag` を使用する例:

```
(set-process-query-on-exit-flag (get-process "shell") nil)
     ⇒ nil
```

`confirm-kill-processes` [User Option]
このユーザーオプションが `t` (デフォルト) にセットされていると、Emacs は exit に際してプロセスを kill する前に確認を求める。`nil`なら Emacs は確認なしでプロセスを kill する (すべてのプロセスの問い合わせフラグを無視する)。

38.12 別のプセスへのアクセス

カレント Emacs セッションのサブプロセスにたいするアクセスと操作に加えて、同一マシン上で実行中の他のプロセスにたいして Emacs Lisp プログラムがアクセスすることもできます。Emacs のサブプロセスと区別するために、わたしたちはこれらをシステムプロセス *(system processes)* と呼んでいます。

Emacs はシステムプロセスへのアクセス用のプリミティブをいくつか提供します。これらのプリミティブはすべてのプラットフォームではサポートされません。これらのプリミティブはサポートされないシステムでは `nil`をリターンします。

`list-system-processes` [Function]
この関数はそのシステム上で実行中のすべてのプロセスのリストをリターンする。各プロセスは PIDという OS から割り当てられた数値によるプロセス ID により識別され、同一時に同一マシン上で実行中の他のプロセスと区別される。

`process-attributes` *pid* [Function]
この関数はプロセス ID *pid*で指定されるプロセスにたいする属性の alist をリターンする。この alist 内の各属性は `(`*`key`*` . `*`value`*`)`という形式であり *key*は属性、*value*はその属性の値である。この関数がリターン可能なさまざまな属性にたいする *key*を以下にリストした。これら

すべての属性をすべてのプラットフォームがサポートする訳ではない。ある属性がサポートされていなければ、その連想値はリターンされる alist 内に出現しない。

`euid` そのプロセスを呼び出したユーザーの実効ユーザー ID(effective user ID)。対応する *value*は数値。プロセスがカレント Emacs セッションを実行したユーザーと同じなら値は `user-uid`がリターンする値と等しくなる (Section 40.4 [User Identification], page 1103 を参照)。

`user` そのプロセスの実効ユーザー ID に対応するユーザー名であるような文字列。

`egid` 実行ユーザー ID のグループ ID であるような数値。

`group` 実効ユーザーのグループ ID に対応するグループ名であるような文字列。

`comm` そのプロセス内で実効したコマンドの名前。これは通常は先行するディレクトリーを除いた実行可能ファイル名を指定する文字列。しかしいくつかの特別なシステムプロセスは、実行可能ファイルやプログラムに対応しない文字列を報告する可能性がある。

`state` そのプロセスの状態コード。これはそのプロセスのスケジューリング状態をエンコードする短い文字列。以下は頻繁に目にするコードのリスト:

- `"D"` 割り込み不可の sleep(通常は I/O による)
- `"R"` 実行中
- `"S"` 割り込み可能な sleep(何らかのイベント待ち)
- `"T"` ジョブ制御シグナルにより停止された
- `"Z"` zombie: 終了したが親プロセスに回収されていないプロセス

可能な状態の完全なリストは `ps`コマンドの man page を参照のこと。

`ppid` 親プロセスのプロセス ID であるような数値。

`pgrp` そのプロセスのプロセスグループ ID であるような数値。

`sess` そのプロセスのセッション ID。これはそのプロセスのセッションリーダー *(session leader)* のプロセス ID であるような数値。

`ttname` そのプロセスの制御端末の名前であるような文字列。これは Unix や GNU システムでは通常は`/dev/pts65`のような対応する端末デバイスのファイル名。

`tpgid` そのプロセスの端末を使用するフォアグラウンドプロセスグループのプロセスグループ ID であるような数値。

`minflt` そのプロセス開始以降に発生したマイナーなページフォルト数 (マイナーなページフォルトとはディスクからの読み込みを発生させないページフォルト)。

`majflt` そのプロセス開始以降に発生したメジャーなページフォルト数 (メジャーなページフォルトとはディスクからの読み込みを要し、それ故にマイナーページフォルトより高価なページフォルト)。

`cminflt`
`cmajflt` `minflt`や `majflt`と似ているが与えられたプロセスのすべての子プロセスのページフォルト数を含む。

`utime` 実行中のアプリケーションのコードにたいして、ユーザーコンテキスト内でプロセスに消費された時間。対応する *value*は Lisp のタイムスタンプ (Section 40.5 [Time of Day], page 1104 を参照)。

`stime` システムコールの処理にたいしてシステム (kernel) コンテキスト内でプロセスに消費された時間。対応する *value*は Lisp のタイムスタンプ。

`time` `utime`と `stime`の和。対応する *value*は Lisp のタイムスタンプ。

`cutime`
`cstime`
`ctime` `utime`や `stime`と同様だが与えられたプロセスのすべての子プロセスの時間が含まれる点が異なる。

`pri` そのプロセスの数値的な優先度。

`nice` そのプロセスの *nice* 値 *(nice value)* であるような数値 (小さい nice 値のプロセスがより優先的にスケジュールされる)。

`thcount` そのプロセス内のスレッド数。

`start` プロセスの開始時刻 (Lisp のタイムスタンプ)。

`etime` プロセスの開始からの経過時間 (Lisp のタイムスタンプ)。

`vsize` そのプロセスの仮想メモリーの KB 単位でのサイズ。

`rss` そのプロセスがマシンの物理メモリー内で占める常駐セット *(resident set)* の KB 単位でのサイズ。

`pcpu` プロセス開始以降に使用された CPU 時間のパーセンテージ。対応する *value*は 0 から 100 の間の浮動小数点数。

`pmem` マシンにインストールされた物理メモリー合計のうち、そのプロセスの常駐セットのパーセンテージ。値は 0 から 100 の間の浮動小数点数。

`args` そのプロセスが呼び出されたときのコマンドライン。これは個々のコマンドライン引数がブランクで区切られた文字列。引数に埋め込まれた空白文字はシステムに応じて適切にクォートされる。GNU や Unix ではバックスラッシュ文字によるエスケープ、Windows ではダブルクォート文字で囲まれる。つまりこのコマンドライン文字列は `shell-command`のようなプリミティブにより直接使用できる。

38.13 トランザクションキュー

トランザクションを用いてサブプロセスと対話するためにトランザクションキュー *(transaction queue)* を使用できます。まず `tq-create`を使用して指定したプロセスと対話するためのトランザクションキューを作成します。それからトランザクションを送信するために `tq-enqueue`を呼び出すことができます。

`tq-create` *process* [Function]
この関数は *process*と対話するトランザクションキューを作成してリターンする。引数 *process* はバイトストリームを送受信する能力をもつサブプロセスであること。これは子プロセス、または (おそらく別のマシン上の) サーバーへの TCP 接続かもしれない。

`tq-enqueue` *queue question regexp closure fn* **&optional** *delay-question* [Function]

この関数はキュー *queue*にトランザクションを送信する。キューの指定は対話するサブプロセスを指定する効果をもつ。

引数 *question*はトランザクションを開始するために発信するメッセージ。引数 *fn*は、それにたいする応答が返信された際に呼び出す関数。これは *closure*と受信した応答という 2 つの引数で呼び出される。

引数 *regexp*は応答全体の終端にマッチして、それより前にはマッチしない正規表現であること。これは `tq-enqueue`が応答の終わりを決定する方法である。

引数 *delay-question*が非 `nil`なら、そのプロセスが以前に発信したすべてのメッセージへの返信が完了するまでメッセージの送信を遅延する。これによりいくつかのプロセスにたいして、より信頼性のある結果が生成される。

`tq-close` *queue* [Function]

保留中のすべてのトランザクションの完了を待機して、トランザクションキュー *queue*をシャットダウンして、それから接続または子プロセスを終了する。

トランザクションキューはフィルター関数により実装されています。Section 38.9.2 [Filter Functions], page 955 を参照してください。

38.14 ネットワーク接続

Emacs Lisp プログラムは同一マシンまたは他のマシン上の別プロセスにたいしてストリーム (TCP) やデータグラム (UDP) のネットワーク接続 (Section 38.16 [Datagrams], page 967 を参照) をオープンできます。ネットワーク接続は Lisp によりサブプロセスと同様に処理されて、プロセスオブジェクトとして表されます。しかし対話を行うそのプロセスは Emacs の子プロセスではなく、プロセス IDをもたず、それを kill したりシグナルを送信することはできません。行うことができるのはデータの送信と受信だけです。`delete-process`は接続をクローズしますが、他方の端のプログラムを kill しません。そのプログラムは接続のクローズについて何を行うか決定しなければなりません。

ネットワークサーバーを作成することにより Lisp プログラムは接続を listen できます。ネットワークサーバーもある種のプロセスオブジェクトとして表されますが、ネットワーク接続とは異なりネットワークサーバーがデータ自体を転送することは決してありません。接続リクエストを受信したときは、それにたいして作成した接続を表す新たなネットワーク接続を作成します (そのネットワーク接続はサーバーからプロセス plist を含む特定の情報を継承する)。その後でネットワークサーバーは更なる接続リクエストの listen に戻ります。

ネットワーク接続とサーバーは、キーワード/引数のペアーで構成される引数リストで`make-network-process`を呼び出すことにより作成されます。たとえば`:server t`はサーバープロセス、`:type 'datagram`はデータグラム接続を作成します。詳細は Section 38.17 [Low-Level Network], page 968 を参照してください。以下で説明する `open-network-stream`を使用することもできます。

異なるプロセスのタイプを区別するために `process-type`関数はネットワーク接続とサーバーには `network`、シリアルポート接続は `serial`、pipe 接続には `pipe`、実際のサブプロセスには `real`をリターンします。

ネットワーク接続にたいして `process-status`関数は `open`、`closed`、`connect`、`stop`、または `failed`をリターンします。ネットワークサーバーにたいしては状態は常に `listen`になります。実際のサブプロセスにたいしては `stop`以外の値はリターンされません。Section 38.6 [Process Information], page 947 を参照してください。

`stop-process`と`continue-process`を呼び出すことにより、ネットワークプロセスの処理の停止と再開が可能です。サーバープロセスにたいする停止は新たな接続の受け付けないことを意味します (サーバー再開時は 5 つまでの接続リクエストがキューされる。これが OS による制限でなければこの制限は増やすことができる。Section 38.17.1 [Network Processes], page 968 の`make-network-process`の`:server`を参照)。ネットワークストリーム接続にたいしては、停止は入力の処理を行わないことを意味します (到着するすべての入力は接続の再開まで待つ)。データグラム接続にたいしては、いくらかのパケットはキューされますが入力は失われるかもしれません。ネットワーク接続またはサーバーが停止しているかどうかを判断するために、関数`process-command`を使用できます。これが非`nil`なら停止しています。

ビルトインや外部のサポートを使用することにより、Emacs は暗号化されたネットワーク接続を作成できます。ビルトインのサポートは GnuTLS トランスポート層セキュリティーライブラリーを使用します。the GnuTLS project page (`https://www.gnu.org/software/gnutls/`) を参照してください。GnuTLS サポートつきで Emacs をコンパイルした場合には関数`gnutls-available-p`が定義されて非`nil`をリターンします。詳細は Section “Overview” in *The Emacs-GnuTLS manual* を参照してください。外部のサポートの場合には`starttls.el`ライブラリーを使用します。これはシステム上に`gnutls-cli`のようなヘルパーユーティリティーのインストールを要します。`open-network-stream`関数は何であれ利用可能なサポートを使用して、暗号化接続作成の詳細を透過的に処理できます。

`open-network-stream` *name buffer host service* **&rest** *parameters* [Function]

この関数はオプションで暗号つきで TCP 接続をオープンして、その接続を表すプロセスオブジェクトをリターンする。

*name*引数はプロセスオブジェクトの名前を指定する。これは必要に応じて一意になるよう変更される。

*buffer*引数はその接続に関連付けるバッファー。その接続からの出力は出力を処理する独自のフィルター関数を指定していない場合には、*buffer*が`nil`ならその接続はバッファーに関連付けられない。

引数 *host*と *service*は接続先を指定する。*host*はホスト名 (文字列)、*service*は定義済みのネットワークサービス名 (文字列)、またはポート番号 (`80`のような整数か`\"80\"`のような整数文字列)。

残りの引数 *parameters*は主に暗号化された接続に関連するキーワード/引数のペアー:

`:nowait` *boolean*
: 非`nil`なら非同期接続を試みる。

`:type` *type*
: 接続のタイプ。オプションは以下のとおり:

  `plain` 通常の暗号化されていない接続。

  `tls`
  `ssl` TLS (Transport Layer Security) 接続。

  `nil`
  `network` plain 接続を開始してパラメーター ‘`:success`’と ‘`:capability-command`’が与えられたら、STARTTLSを通じて暗号化接続への更新を試みる。これが失敗したら暗号化されていない接続のまま留まる。

  `starttls` `nil`と同様だが STARTTLSが失敗したらその接続を切断する。

`shell` shell 接続。

`:always-query-capabilities` *boolean*
: 非`nil`なら、たとえ‘`plain`’な接続を行っているときでも常にサーバーの能力を問い合わせる。

`:capability-command` *capability-command*
: ホストの能力を問い合わせるためのコマンド文字列。

`:end-of-command` *regexp*
`:end-of-capability` *regexp*
: コマンドの終端、またはコマンド *capability-command*の終端にマッチする正規表現。前者は後者のデフォルト。

`:starttls-function` *function*
: 単一の引数 (*capability-command*にたいする応答) をとり`nil`、またはサポートされていればSTARTTLSをアクティブにするコマンドをリターンする関数。

`:success` *regexp*
: 成功したSTARTTLSネゴシェーションにマッチする正規表現。

`:use-starttls-if-possible` *boolean*
: 非`nil`なら、たとえEmacsがビルトインのTLSサポートをもっていなくても、日和見的 (opportunistic) にSTARTTLSアップグレードを行う。

`:warn-unless-encrypted` *boolean*
: 非`nil`、かつ`:return-value`も非`nil`なら、接続が暗号化されていない場合にはEmacsは警告するだろう。これはほとんどのユーザーがネットワークトラフィックが暗号化されていることを期待するであろうIMAPやその類のプロトコルにたいして有用。

`:client-certificate` *list-or-t*
: 証明書 (certificate) のキーと、証明書のファイル自身を命名する`(`*key-file cert-file*`)`という形式のリスト、またはこの情報にたいして`auth-source`を尋ねることを意味する`t`のいずれか (Section “auth-source” in *Emacs auth-source Library* を参照)。TLSやSTARTTLSにたいしてのみ使用される。`:client-certificate`を未指定時に、自動的な`auth-source`の問い合わせを有効にするには、`network-stream-use-client-certificates`を t にカスタマイズすればよい。

`:return-list` *cons-or-nil*
: この関数のリターン値。省略または`nil`ならプロセスオブジェクトをリターンする。それ以外なら`(`*process-object* `.` *plist*`)`という形式のコンスセルをリターンする。ここで *plist*は以下のキーワード:

  `:greeting` *string-or-nil*
  : 非`nil`ならホストからリターンされたgreeting(挨拶) 文字列。

  `:capabilities` *string-or-nil*
  : 非`nil`ならホストの能力 (capability) 文字列。

  `:type` *symbol*
  : 接続タイプであり、‘`plain`’か‘`tls`’のいずれか。

`:shell-command` *string-or-nil*
: 接続 `type`が `shell`なら、このパラメーターは接続の作成のために実行されるフォーマット仕様文字列として解釈される。利用可能な仕様はホスト名の '`%s`'とポート番号の '`%p`'。たとえばプレーン接続を作成する前にまず '`gateway`'に ssh で接続したければ、このパラメーターは '`ssh gateway nc %s %p`'のようになるかもしれない。

38.15 ネットワークサーバー

`:server t`で `make-network-process`を呼び出すことによりサーバーが作成されます (Section 38.17.1 [Network Processes], page 968 を参照)。そのサーバーはクライアントからの接続リクエストを listen するでしょう。クライアントの接続リクエストを accept(受け入れる) する際は以下のようなパラメーターで、それ自体がプロセスオブジェクトであるようなネットワーク接続を作成します。

- その接続のプロセス名はサーバープロセスの *name*とクライアント識別文字列を結合して構築される。IPv4 接続にたいするクライアント識別文字列はアドレスとポート番号を表す '`<a.b.c.d:p>`' のような文字列。それ以外なら '`<nnn>`'のようにカッコで囲まれた一意な数字。この数字はその Emacs セッション内のそれぞれの接続にたいして一意。
- サーバーが非デフォルトのフィルターをもつ場合には、その接続プロセスは別個にプロセスバッファーを取得しない。それ以外なら Emacs はその目的のために新たにバッファーを作成する。サーバーのバッファー名かプロセス名にクライアント識別文字列に結合したものがバッファー名になる。

  サーバーのプロセスバッファーの値が直接使用されることは決してないが、log 関数は接続のログを記録するためにそれを取得して、そこにテキストを挿入して使用することができる。
- 通信タイプ (communication type)、プロセスフィルター、およびセンチネルはそれぞれサーバーのものから継承される。サーバーが直接フィルターとセンチネルを使用することは決してない。それらの唯一の目的はサーバーへの接続を初期化することである。
- その接続のプロセスコンタクト情報は、クライアントのアドレス情報 (通常は IP アドレスとポート番号) に応じてセットされる。この情報は `process-contact`のキーワード `:host`、`:service`、`:remote`に関連付けられる。
- その接続のローカルアドレスは使用するポート番号に応じてセットアップされる。
- クライアントプロセスの plist はサーバーの plist からインストールされる。

38.16 データグラム

データグラム *(datagram)* 接続は、データストリームではなく個別のパッケージで対話します。`process-send`を呼び出すたびに 1 つのデータグラムパケット (Section 38.7 [Input to Processes], page 950 を参照) が送信されて、受信されたデータグラムごとに 1 回フィルター関数が呼び出されます。

データグラム接続は毎回同じリモートピア (remote peer) と対話する必要はありません。データグラム接続はデータグラムの送信先を指定するリモートピアアドレス *(remote peer address)* をもちます。フィルター関数にたいして受信されたデータグラムが渡されるたびに、そのデータグラムの送信元アドレスがピアアドレスにセットされます。このようにもしフィルター関数がデータグラムを送信したら、それは元の場所へ戻ることになります。`:remote`キーワードを使用してデータグラム接続を作成する際にはリモートピアアドレスを指定できます。`set-process-datagram-address`を呼び出すことにより後からそれを変更できます。

`process-datagram-address` *process* [Function]
: *process*がデータグラム接続かサーバーなら、この関数はそれのリモートピアアドレスをリターンする。

`set-process-datagram-address` *process address* [Function]
: *process*がデータグラム接続かサーバーなら、この関数はそのリモートピアアドレスに *address* をセットする。

38.17 低レベルのネットワークアクセス

`make-network-process`を使用することにより、`open-network-stream`より低レベルでの処理からネットワーク接続を作成することもできます。

38.17.1 `make-network-process`

ネットワーク接続やネットワークサーバーを作成する基本的な関数は`make-network-process`です。これは与えられた引数に応じて、これらの仕事のいずれかを行うことができます。

`make-network-process` **&rest** *args* [Function]
: この関数はネットワーク接続やサーバーを作成して、それを表すプロセスオブジェクトをリターンする。引数 *args*はキーワード/引数のペアからなるリスト。キーワードの省略は`:coding`、`:filter-multibyte`、`:reuseaddr`を除いて、常に値として`nil`を指定したのと同じことになる。重要なキーワードを以下に示す (ネットワークオプションに対応するキーワードを以降のセクションにリストする)。

  :name *name*
  : プロセス名として文字列 *name*を使用する。一意にするために必要に応じて変更され得る。

  :type *type* コミュニケーションのタイプを指定する。値`nil`はストリーム接続 (デフォルト)、`datagram`はデータグラム接続、`seqpacket`はシーケンスパケットストリーム (sequenced packet stream) による接続を指定する。接続およびサーバーの両方でこれらのタイプを指定できる。

  :server *server-flag*
  : *server-flag*が非`nil`ならサーバー、それ以外なら接続を作成する。ストリームタイプのサーバーでは *server-flag*はそのサーバーへの保留中の接続キューの長さを指定する整数を指定できる。キューのデフォルト長は 5。

  :host *host* 接続するホストを指定する。*host*はホスト名かインターネットアドレス (文字列)、またはローカルホストを指定するシンボル`local`。サーバーを *host*に指定する場合にはローカルホストにとって有効なアドレスを指定しなければならず、そのアドレスへはクライアント接続だけが許可されるようになる。`local`を使用する際にはデフォルトでは IPv4 を使用するが、`ipv6`の *family*を指定してこれをオーバーライドできる。すべてのインターフェースを listen するには IPv4 では‘`"0.0.0.0"`’、IPv6 では‘`"::"`’を指定する。いくつかのオペレーティングシステムでは‘`"::"`’を listen すると IPv4 も listen するので、その後で別個に IPv4 の listen を試みると結果はエラー `EADDRINUSE` (‘`"Address already in use"`’) となるだろう。

  :service *service*
  : *service*は接続先のポート番号、またはサーバーにたいしては listen するポート番号である。これは‘`"https"`’のようにポート番号に変換されるようなサービス名、

または直接ポート番号を指定する‘4430’のような整数か‘\"443\"’のような整数文字列であること。サーバーにたいしては tも指定でき、これは未使用のポート番号をシステムに選択させることを意味する。

:family *family*

*family*は接続のアドレス (またはプロトコル) のファミリーを指定する。nilは与えられた *host*と *service*にたいして自動的に適切なアドレスファミリーを決定する。localはUnixのsocketを指定して、この場合には *host*は無視される。ipv4と ipv6はそれぞれIPv4とIPv6の使用を指定する。

:use-external-socket *use-external-socket*

*use-external-socket*が非 nilなら割り当てられた socket のかわりに呼び出し時に Emacs に渡された socket を使用する。これは Emacs サーバーのコードにおいてオンデマンドの socket アクティベージョンを可能にするために使用される。Emacs が socket を渡されていなければ、このオプションは暗黙に無視される。

:local *local-address*

サーバープロセスでは *local-address*は listen するアドレスである。これは *family*、*host*、*service*をオーバーライドするので、これらを指定しないこともできる。

:remote *remote-address*

接続プロセスでは *remote-address*は接続先のアドレス。これは *family*、*host*、*service*をオーバーライドするので、これらを指定しないこともできる。

データグラムサーバーでは *remote-address*はリモートデータグラムアドレスの初期セッティングを指定する。

*local-address*と *remote-address*のフォーマットはアドレスファミリーに依存する:

- IPv4アドレスは4つの8ビット整数と1つの16ビット整数からなる5要素のベクター [*a b c d p*]で表され、それぞれ数値的なIPv4アドレス *a*.*b*.*c*.*d*、およびポート番号 *p*に対応する。
- IPv6アドレスは9要素の16ビット整数ベクター [*a b c d e f g h p*]で表され、それぞれ数値的なIPv 6アドレス *a*:*b*:*c*:*d*:*e*:*f*:*g*:*h*、およびポート番号 *p*に対応する。
- ローカルアドレスはローカルアドレススペース内でアドレスを指定する文字列として表される。
- 未サポートファミリー (unsupported family) のアドレスはコンスセル (*f* . *av*)で表される。ここで *f*はファミリー名、*av*はアドレスデータバイトごとに1つの要素を使用するソケットアドレスを指定するベクター。可搬性のあるコードでこのフォーマットを信頼してはならない。これは実装定義の定数、データサイズ、データ構造のアライメントに依存する可能性があるからだ。

:nowait *bool*

ストリーム接続にたいして *bool*が非 nilなら、その接続の完了を待機せずにリターンする。接続の成功や失敗時には、Emacsは"open" (成功時)、または"failed" (失敗時) にマッチするような第2引数によりセンチネル関数を呼び出すだろう。デフォルトでは wait せずに block するので、make-network-processはその接続が成功または失敗するまでリターンしない。

非同期 TLS 接続をセットアップする場合には:tls-parameterパラメーター (下記参照) も提供する必要があるだろう。

Emacs の機能に応じて:nowaitが非同期になる方法は異なる。非同期で実行可能 (または不可能) な 3 つの要素はドメイン名解決、socket のセットアップ、および (TLS 接続にいする)TLS ネゴシエーション。

プロセスオブジェクトと相互作用する多くの関数 (たとえばprocess-datagram-address) は有用な値をリターンする以前に、少なくとも socket を所有することに依存する。これらの関数は socket が望ましい状態に達するまでブロックされる。非同期 socket と相互作用するための推奨方法はプロセスにセンチネルを配置して、その状態が‘"run"’になるまで相互作用を試みないことである。この方法ではこれらの関数はブロックされない。

:tls-parameters
TLS 接続をオープンする際には最初の要素は TLS タイプ (gnutls-x509pkiか gnutls-anonのいずれか) であり、残りの要素は gnutls-bootが受容するキーワードリスト形式であること (このキーワードリストは gnutls-boot-parameters関数で取得できる)。それからホストへの接続が完了後に TLS 接続は確立される。

:stop *stopped*
*stopped*が非nilなら stopped(停止) の状態でネットワーク接続、またはサーバーを開始する。

:buffer *buffer*
プロセスバッファーとして *buffer*を使用する。

:coding *coding*
このプロセスにたいするコーディングシステムとして *coding*を使用する。接続からのデータのデコードおよび接続への送信データのエンコードに異なるコーディングシステムを指定するには、*coding*にたいして (*decoding* . *encoding*)と指定する。

このキーワードをまったく指定しないかった場合のデフォルトは、そのデータからコーディングシステムを判断する。

:noquery *query-flag*
プロセス query フラグを *query-flag*に初期化する。Section 38.11 [Query Before Exit], page 961 を参照のこと。

:filter *filter*
プロセスフィルターを *filter*に初期化する。

:filter-multibyte *multibyte*
*multibyte*が非nilならマルチバイト文字列、それ以外ならユニバイト文字列がプロセスフィルターに与えられるデフォルトは t。

:sentinel *sentinel*
プロセスセンチネルを *sentinel*に初期化する。

:log *log* サーバープロセスの log 関数を *log*に初期化する。サーバーがクライアントからネットワーク接続を accept するたびにその log 関数が呼び出される。log 関数に渡される引数は *server*、*connection*、*message*。ここで *server*はサーバープロセス、*connection*はその接続にたいする新たなプロセス、*message*は何が発生したかを説明する文字列。

:plist *plist*　プロセス plist を *plist*に初期化する。

実際の接続情報で修正されたオリジナルの引数リストは `process-contact`を通じて利用できる。

38.17.2 ネットワークのオプション

以下のネットワークオプションはネットワークプロセス作成時に指定できます。`:reuseaddr`を除き、`set-network-process-option`を使用してこれらのオプションを後からセットや変更することもできます。

サーバープロセスにたいしては、`make-network-process`で指定されたオプションはクライアントに継承されないので、子接続が作成されるたびに必要なオプションをセットする必要があるでしょう。

:bindtodevice *device-name*
: *device-name*がネットワークインターフェースを指定する空でない文字列なら、そのインターフェースで受信したパケットだけを処理する。*device-name*が `nil`(デフォルト)なら任意のインターフェースが受信したパケットを処理する。

このオプションの使用にたいして特別な特権を要求するシステムがいくつかあるかもしれない。

:broadcast *broadcast-flag*
: データグラムプロセスにたいして *broadcast-flag*が非 `nil`なら、そのプロセスはブロードキャストアドレスに送信されたデータグラムパケットを受信して、ブロードキャストアドレスにパケットを送信できるだろう。これはストリーム接続では無視される。

:dontroute *dontroute-flag*
: *dontroute-flag*が非 `nil`ならプロセスはローカルホストと同一ネットワーク上のホストだけに送信することができる。

:keepalive *keepalive-flag*
: ストリーム接続にたいして *keepalive-flag*が非 `nil`なら、低レベルの keep-alive メッセージの交換が有効になる。

:linger *linger-arg*
: *linger-arg*が非 `nil`なら、接続を削除 (`delete-process`を参照) する前にキューされたすべてのパケットの送信が成功するまで待機する。*linger-arg*が整数なら、接続クローズ前のキュー済みパケット送信のために待機する最大の秒数を指定する。デフォルトは `nil`で、これはプロセス削除時に未送信のキュー済みパケットを破棄することを意味する。

:oobinline *oobinline-flag*
: ストリーム接続にたいして *oobinline-flag*が非 `nil`なら、通常のデータストリーム内の帯域外 (out-of-band) データを受信して、それ以外なら帯域外データは破棄する。

:priority *priority*
: この接続で送信するパケットの優先順位を整数 *priority*にセットする。たとえばこの接続で送信する IP パケットの TOS(type of service) フィールドにセットする等、この数字の解釈はプロトコルに固有である。またそのネットワークインターフェース上で特定の出力キューを選択する等、これにはシステム依存の効果もある。

:reuseaddr *reuseaddr-flag*
: ストリームプロセスサーバーにたいして *reuseaddr-flag*が非 `nil` (デフォルト) なら、そのホスト上の別プロセスがそのポートですでに listen していなければ、このサーバー

は特定のポート番号 (`:service`を参照) を再使用できる。*reuseaddr-flag*が`nil`なら、(そのホスト上の任意のプロセスが) そのポートを最後に使用した後、そのポート上で新たなサーバーを作成するのが不可能となるような一定の期間が存在するかもしれない。

`set-network-process-option` *process option value* **&optional** *no-error* [Function]

この関数はネットワークプロセス *process*にたいしてネットワークオプションのセットや変更を行う。指定できるオプションは`make-network-process`と同様。*no-error*が非`nil`なら、*option*がサポートされないオプションの場合に、この関数はエラーをシグナルせずに`nil`をリターンする。この関数が成功裏に完了したら`t`をリターンする。

あるオプションのカレントのセッティングは`process-contact`関数を通じて利用できる。

38.17.3 ネットワーク機能の可用性のテスト

与えられネットワーク機能が利用可能かテストするためには以下のように`featurep`を使用します:

```
(featurep 'make-network-process '(keyword value))
```

このフォームの結果は`make-network-process`内で *keyword*に値 *value*を指定することが機能するなら`t`になります。以下はこの方法でテストできる *keyword*/*value*ペアーのいくつかです。

`(:nowait t)`
: 非ブロッキング接続がサポートされていれば非`nil`。

`(:type datagram)`
: データグラムがサポートされていれば非`nil`。

`(:family local)`
: ローカル socket(別名 “UNIX domain”) がサポートされていれば非`nil`。

`(:family ipv6)`
: IPv6 がサポートされていれば非`nil`。

`(:service t)`
: サーバーにたいしてシステムがポートを選択できれば非`nil`。

与えられたネットワークオプションが利用可能かテストするためには、以下のように`featurep`を使用します:

```
(featurep 'make-network-process 'keyword)
```

指定できる *keyword*の値は`:bindtodevice`等です。完全なリストは Section 38.17.2 [Network Options], page 971 を参照してください。このフォームは`make-network-process` (または`set-network-process-option`) が特定のネットワークオプションをサポートしていれば非`nil`をリターンします。

38.18 その他のネットワーク機能

以下の追加の関数はネットワーク接続の作成や操作に有用です。これらはいくつかのシステムでのみサポートされることに注意してください。

`network-interface-list` **&optional** *full family* [Function]

この関数は使用しているマシン上のネットワークインターフェースを記述するリストをリターンする。値は要素が`(ifname . address)`という形式をもつような alist。*ifname*はそのインターフェースを命名する文字列、*address*は`make-network-process`の引数 *local-address*お

よび *remote-address*の形式と同じ。デフォルトでは可能なら IPv4 と IPv6 の両方のアドレスをリターンする。

オプション引数 *full*が非 `nil`なら、かわりに (*ifname addr bcast netmask*) という形式の要素を 1 つ以上もつリストをリターンする。*ifname*はそのインターフェースを命名する一意ではない文字列。*addr*、*bcast*、*netmask*はそれぞれ IP アドレス、ブロードキャストアドレス、ネットワークマスクを詳述する整数のベクター。

シンボル `ipv4`または `ipv6`としてオプション引数 *family*を指定すると、*full*の値とは独立してリターンする情報をそれぞれ IPv4 または IPv6 に制限する。IPv6 サポートが利用不可な際に `ipv6`を指定するとエラーがシグナルされるだろう。

いくつか例を示す:

```
(network-interface-list) ⇒
(("vmnet8" .
  [172 16 76 1 0])
 ("vmnet1" .
  [172 16 206 1 0])
 ("lo0" .
  [65152 0 0 0 0 0 0 1 0])
 ("lo0" .
  [0 0 0 0 0 0 0 1 0])
 ("lo0" .
  [127 0 0 1 0]))
(network-interface-list t) ⇒
(("vmnet8"
  [172 16 76 1 0]
  [172 16 76 255 0]
  [255 255 255 0 0])
 ("vmnet1"
  [172 16 206 1 0]
  [172 16 206 255 0]
  [255 255 255 0 0])
 ("lo0"
  [65152 0 0 0 0 0 0 1 0]
  [65152 0 0 0 65535 65535 65535 65535 0]
  [65535 65535 65535 65535 0 0 0 0 0])
 ("lo0"
  [0 0 0 0 0 0 0 1 0]
  [0 0 0 0 0 0 0 1 0]
  [65535 65535 65535 65535 65535 65535 65535 65535 0])
 ("lo0"
  [127 0 0 1 0]
  [127 255 255 255 0]
  [255 0 0 0 0]))
```

`network-interface-info` *ifname* [Function]

この関数は *ifname*という名前のネットワークインターフェースに関する情報をリターンする。値は (*addr bcast netmask hwaddr flags*)という形式をもつリスト。

addr インターネットプロトコルアドレス。

bcast ブロードキャストアドレス。

netmask ネットワークマスク。

hwaddr レイヤー 2 アドレス (たとえばイーサネット MAC アドレス)。

flags そのインターフェースのカレントのフラグ。

この関数は IPv4 の情報だけをリターンすることに注意。

`format-network-address` *address* **&optional** *omit-port* [Function]

この関数はネットワークアドレスの Lisp 表現を文字列に変換する。

5 要素のベクター `[a b c d p]`は IPv4 アドレス *a.b.c.d*、およびポート番号 *p*を表す。`format-network-address`はこれを文字列`\"a.b.c.d:p\"`に変換する。

9 要素のベクター `[a b c d e f g h p]`はポート番号とともに IPv6 アドレスを表す。`format-network-address`はこれを文字列`"[a:b:c:d:e:f:g:h]:p"`に変換する。

このベクターにポート番号が含まれない、または *omit-port*が非 `nil`なら結果にサフィックス `:p`は含まれない。

`network-lookup-address-info` *name* **&optional** *family* [Function]

この関数は *name*でホスト名を照合するために使用される。この名前には ASCII 文字列のみを期待しており、さもなくばエラーをシグナルする。国際化されたホスト名を照合しなければ、最初に *name*にたいして `puny-encode-domain`を呼び出すこと。

成功時にはネットワークアドレスを表す Lisp のリスト、それ以外は `nil`をリターンする。

デフォルトでは IPv4 と IPv6 の両方の照合を試みる。オプション引数 *family*がこの挙動を制御する。これにシンボル `ipv4`または `ipv6`を指定すると、それぞれ IPv4 または IPv6 に照合を制限する。

38.19 シリアルポートとの対話

Emacs はシリアルポートと対話できます。インタラクティブな *M-x serial-term*の使用にたいしては端末ウィンドウをオープンして、Lisp プログラム `make-serial-process`にたいしてはプロセスオブジェクトを作成します。

シリアルポートはクローズと再オープンなしで実行時に設定することができます。関数 `serial-process-configure`によりスピード、バイトサイズ、およびその他のパラメーターを変更できます。`serial-term`で作成された端末ウィンドウではモードラインをクリックして設定を行うことができます。

シリアル接続はプロセスオブジェクトとして表されて、サブプロセスやネットワークプロセスと同様の方法で使用できます。これによりデータの送受信やシリアルポートの設定ができます。しかしシリアルプロセスオブジェクトにプロセス ID はありません。それにたいしてシグナルの送信はできずステータスコードは他のタイプのプロセスオブジェクトとは異なります。プロセスオブジェクトへの `delete-process`、またはプロセスバッファーにたいする `kill-buffer`は接続をクローズしますが、そのシリアルポートに接続されたデバイスに影響はありません。

関数 `process-type`はシリアルポート接続を表すプロセスオブジェクトにたいするシンボル `serial`をリターンします。

シリアルポートは GNU/Linux や Unix、そして MS Windows のシステムで利用できます。

`serial-term` *port speed* **&optional** *line-mode* [Command]
新たなバッファー内でシリアルポートにたいする端末エミュレーターを開始する。*port*は接続先のシリアルポートの名前。たとえばUnixではこれは`/dev/ttyS0`のようになるだろう。MS Windowsでは`COM1`や`\\.\COM10`のようになるかもしれない (Lisp文字列ではバックスラッシュは2重にすること)。

*speed*はビット毎秒でのシリアルポートのスピード。一般的な値は9600。そのバッファーはTermモードになる。このバッファーで使用するコマンドについてはSection "Term Mode" in *The GNU Emacs Manual* を参照のこと。モードラインメニューからスピードと設定を変更できる。*line-mode*が非`nil`なら`term-line-mode`、それ以外は`term-raw-mode`を使用する。

`make-serial-process` **&rest** *args* [Function]
この関数はプロセスとバッファーを作成する。引数はキーワード/引数ペアーで指定する。以下は意味のあるキーワードのリストで、最初の2つ (*port*と *speed*) は必須:

`:port` *port*
: これはシリアルポートの名前。UnixやGNUシステムでは`/dev/ttyS0`のようなファイル名、Windowsでは`COM1`、`COM9`より高位のポートでは`\\.\COM10`のようになるかもしれない (Lisp文字列ではバックスラッシュは2重にすること)。

`:speed` *speed*
: ビット毎秒でのシリアルポートのスピード。この関数は`serial-process-configure`を呼び出すことによりスピードを操作する。この関数の更なる詳細については以降のドキュメントを参照のこと。

`:name` *name*
: そのプロセスの名前。*name*が与えられなければ*port*がプロセス名の役目も同様に果たす。

`:buffer` *buffer*
: そのプロセスに関連付けられたバッファー。値はバッファー、またはそれがバッファーの名前であるような文字列かもしれない。出力を処理するために出力ストリームやフィルター関数を指定しなければ、プロセス出力はそのバッファーの終端に出力される。*buffer*が与えられなければ、そのプロセスバッファーの名前は`:name`キーワードから取得される。

`:coding` *coding*
: *coding*はこのプロセスにたいする読み書きに使用されるコーディングシステムを指定する。*coding*がコンス`(`*decoding* `.` *encoding*`)`なら読み取りに*decoding*、書き込みには*encoding*が使用される。指定されない場合のデフォルトはデータ自身から判断されるコーディングシステム。

`:noquery` *query-flag*
: プロセスqueryフラグを*query-flag*に初期化する。Section 38.11 [Query Before Exit], page 961 を参照のこと。未指定の場合のフラグのデフォルトは`nil`。

`:stop` *bool*
: *bool*が非`nil`ならstoppedの状態でプロセスを開始する。stopped状態ではシリアルプロセスは入力データを受け付けないが出力データの送信は可能。stopped状態のクリアーは`continue-process`、セットは`stop-process`で行う。

`:filter` *filter*
: プロセスフィルターとして *filter*をインストールする。

`:sentinel` *sentinel*
: プロセスセンチネルとして *sentinel*をインストールする。

`:plist` *plist*
: プロセスの初期 plist として *plist*をインストールする。

`:bytesize`
`:parity`
`:stopbits`
`:flowcontrol`
: これらは `make-serial-process`が呼び出す `serial-process-configure`により処理される。

後の設定により変更され得るオリジナルの引数リストは関数 `process-contact`を通じて利用可能。

以下は例:

```
(make-serial-process :port "/dev/ttyS0" :speed 9600)
```

`serial-process-configure` **&rest** *args* [Function]

この関数はシリアルポート接続を設定する。引数はキーワード/引数ペアーで指定する。与えられない属性はそのプロセスのカレントの設定 (関数 `process-contact`を通じて利用可能) から再初期化されるか、妥当なデフォルトにセットされる。以下の引数が定義されている:

`:process` *process*
`:name` *name*
`:buffer` *buffer*
`:port` *port*
: 設定するプロセスを識別するために、これらの引数のいずれかが与えられる。これらの引数が何も与えられなければカレントバッファーのプロセスが使用される。

`:speed` *speed*
: ビット毎秒、別名ボーレート *(baud rate)* によるシリアルポートのスピード。値には任意の数字が可能だが、ほとんどのシリアルポートは 1200 から 115200 の間の数少ない定義済みの値でのみ機能して、もっとも一般的な値は 9600。*speed*が `nil`なら、この関数は他のすべての引数を無視してそのポートを設定しない。これは接続を通じて送信された '`AT`'コマンドでのみ設定可能な、Bluetooth/シリアル変換アダプターのような特殊なシリアルポートで有用かもしれない。*speed*にたいする値 `nil`は `make-serial-process`か `serial-term`の呼び出しにより、すでにオープン済みの接続にたいしてのみ有効。

`:bytesize` *bytesize*
: ビット/バイトでの数値で 7 か 8 を指定できる。*bytesize*が与えられない、または `nil`の場合のデフォルトは 8。

`:parity` *parity*
: 値には `nil` (パリティなし)、シンボル `odd` (奇数パリティ)、シンボル `even` (偶数パリティ) を指定できる。*parity*が与えられない場合のデフォルトはパリティなし。

`:stopbits` *stopbits*
: 各バイトの送信を終了するために使用されるストップビットの数値。*stopbits*には 1 か 2 が可能。*stopbits*が与えられない、または`nil`の場合のデフォルトは 1。

`:flowcontrol` *flowcontrol*
: この接続にたいして使用するフロー制御のタイプで `nil` (フロー制御を使用しない)、シンボル `hw` (RTS/CTS ハードウェアフロー制御)、シンボル `sw` (XON/XOFF ソフトウェアフロー制御) のいずれか。*flowcontrol*が与えられない場合のデフォルトはフロー制御なし。

シリアルポートの初期設定のために`make-serial-process`は内部的に`serial-process-configure`を呼び出す。

38.20 バイト配列の pack と unpack

このセクションでは通常はバイナリーのネットワークプロトコル用のバイト配列を pack や unpack する方法を説明します。以下の関数はバイト配列と alist との間で相互に変換を行います。バイト配列はユニバイト文字列、または整数ベクターとして表現することができます。一方で alist はシンボルを固定サイズのオブジェクト、または再帰的な副 alist のいずれかに関連付けます。このセクションで参照する関数を使用するためには`bindat`ライブラリーをロードしてください。

バイト配列からネストされた alist への変換は逆方向への変換がシリアライズ化 *(serializing)* または *pack* 化 *(packing)* として呼ばれることから、非シリアル化【*deserializing)* または *unpack* 化 *(unpacking)* として知られています。

38.20.1 データレイアウトの記述

unpack と pack を制御するためにはデータレイアウト仕様 *(data layout specification)* を記述します。これは名前付きかつタイプ付けされたフィールド *(field)* を記述する特別なネスト化リストです。これは処理する各フィールドの長さ、およびそれを pack および unpack する方法を制御します。わたしたちは名前が '`-bindat-spec`'で終わる変数では bindat の仕様を遵守します。この類の変数名は自動的に risky(危険) だと認識されます。

フィールドのタイプ *(type)* はフィールドが表すオブジェクトのサイズ (バイト単位)、およびそれがマルチバイトフィールドならフィールがバイトオーダーされる方法を記述します。可能なオーダーはビッグエンディアン (big endian。ネットワークバイトオーダーとも呼ばれる)、およびリトルエンディアン (little endian) の 2 つです。たとえば数字`#x23cd` (10 進の 9165) のビッグエンディアンは`#x23 #xcd`の 2 バイト、リトルエンディアンは`#xcd #x23`になるでしょう。以下は可能なタイプの値です:

`u8`
`byte`
: 長さ 1 の符号なしタイプ。

`u16`
`word`
`short`
: 長さ 2 のネットワークバイトオーダーによる符号なし整数。

`u24`
: 長さ 3 のネットワークバイトオーダーによる符号なし整数。

`u32`
`dword`
`long`
: 長さ 4 のネットワークバイトオーダーによる符号なし整数。注意: これらの値は Emacs の整数の実装に制限されるだろう。

`u16r`
`u24r`
`u32r` それぞれ長さ 2、3、4 のリトルエンディアンオーダーによる符号なし整数。

`str` *len* 長さ *len*の文字列。

`strz` *len* 長さ *len*の固定長フィールド内の NUL 終端された文字列。

`vec` *len* [*type*]
タイプ *type* (デフォルトは byte) の *len*要素のベクター。*type*は上述した単純なタイプのいずれか、あるいは `(vec` *len* `[`*type*`])`という形式のリストによる別ベクターの指定。

`ip` インターネットアドレスを表す 4 つの byte のベクター。たとえば localhost は `[127 0 0 1]`。

`bits` *len* *len*バイト内のセットされたビット位置のリスト。バイトはビッグエンディアンでビット位置は `8 *` *len* $-$ `1`で始まり 0 で終わるよう番号が付与される。たとえば `bits 2`では、`#x28 #x1c`は `(2 3 4 11 13)`、`#x1c #x28`は `(3 5 10 11 12)`に unpack される。

`(eval` *form*`)`
*form*はフィールドが pack や unpack された瞬間に評価される Lisp 式。評価した結果は上記にリストしたタイプ使用のいずれかであること。

固定長フィールドでは長さ *len*がフィールド内のバイト数を指定する整数として与えられます。

フィールド長が固定でなければ通常は先行するフィールドの値に依存します。この場合には長さ *len*は後述の `bindat-get-field`のフォーマット指定によりフィールド名 *(field name)* を指定するリスト `(`*name* `...)`、または式 `(eval` *form*`)` (*form*はフィールド長を指定する整数に評価されること) のいずれかで与えることもできます。

フィールド仕様は一般的に `(`[*name*] *handler*`)`という形式をもち、*name*はオプションです。紛らわしくなるのでタイプ仕様 (上述) やハンドラー仕様 (後述) で意味をもつシンボルの名前は使用しないでください。*name*はシンボルまたは式 `(eval` *form*`)`でもよく、この場合には *form*はシンボルに評価される必要があります。

*handler*はそのフィールドが pack や unpack される方法を記述して、以下のいずれかを指定できます:

type タイプ仕様 *type*に応じてこのフィールドの unpack/pack を行う。

`eval` *form* 副作用のためだけに Lisp 式 *form*を評価する。フィールド名が指定されたら値はそのフィールド名にバインドされる。

`fill` *len* *len*バイトをスキップする。pack 化ではそれらを未変更のままとして、通常それらは 0 のままとなることを意味する。unpack 化ではそれらが無視されることを意味する。

`align` *len* *len*バイトの次の倍数にスキップする。

`struct` *spec-name*
副仕様 (sub-specification) として *spec-name*を処理する。これは別の構造体内にネストされる構造体を記述する。

`union` *form* `(`*tag spec*`)...`
Lisp 式 *form*を評価、それにマッチする最初の *tag*を探して、それに関連付けられたレイアウト仕様 *spec*を処理する。マッチングは以下の 3 つのいずれかで発生し得る:

- *tag*が `(eval` *expr*`)`という形式をもつ場合には、変数 `tag`を動的に *form*の値にバインドして *expr*を評価する。結果が非 `nil`ならマッチを示す。

- *tag*が *form*の値と `equal`ならマッチ。
- *tag*が `t`なら無条件にマッチ。

`repeat` *`count field-specs...`*
: *field-specs*を再帰的に順次処理した後に、最初のものから繰り返して、すべての仕様全体を *count*回処理する。*count*はフィールド長と同じフォーマットを使用して与えられる。`eval`フォームが使用された場合には 1 回だけ評価される。正しく処理されるためには、*field-specs*内の各仕様が名前を含まなければならない。

bindat 仕様内で仕様される `(eval` *`form`*`)`フォームでは、評価の間に *form*はこれらの動的にバインドされた変数へのアクセスと更新が可能である。

`last`
: 最後に処理されたフィールドの値。

`bindat-raw`
: バイト配列のデータ。

`bindat-idx`
: unpack 化/pack 化にたいする、(`bindat-raw`での) カレントインデックス。

`struct`
: これまでに unpack された構造化データ、または pack された構造体全体を含む alist。この構造体の特定のフィールドにアクセスするために `bindat-get-field`を使用できる。

`count`
`index`
: `repeat`ブロック内部では、これらは (*count*パラメーターで指定された) 繰り返しの最大回数、および (0 から数えた) カレント繰り返し回数を含む。`count`を 0 にセットすることにより、カレントの繰り返し終了後に最内繰り返しブロックを終了する。

38.20.2 バイトの unpack と pack を行う関数

以降のドキュメントでは *spec*はデータレイアウト仕様、`bindat-raw`はバイト配列、*struct*は unpack されたフィールドデータを表す alist を参照します。

`bindat-unpack` *spec bindat-raw* **&optional** *bindat-idx* [Function]
: この関数はユニバイト文字列、またはバイト配列`bindat-raw`のデータを *spec*に応じて unpack する。これは通常はバイト配列の先頭から unpack 化を開始するが、*bindat-idx*が非`nil`ならかわりに使用する 0 基準の開始位置を指定する。

  値はそれぞれの要素が unpack されたフィールドを記述する alist かネストされた alist。

`bindat-get-field` *struct* **&rest** *name* [Function]
: この関数はネストされた alist である *struct*からフィールドのデータを選択する。*struct*は通常は`bindat-unpack`がリターンしたもの。*name*が単一の引数に対応する場合にはトップレベルのフィールド値を抽出することを意味する。複数の *name*引数は副構造体を繰り返して照合することを指定する。整数の名前は配列のインデックスとして動作する。

  たとえば *name*が `(a b 2 c)`なら、それはフィールド `a`の副フィールド `b`の 3 番目の要素内のフィールド `c` (C では `struct.a.b[2].c`に相当) を意味する。

pack や unpack の処理をすることによりメモリー内でデータ構造が変化しても、そのデータの全フィールド長の合計バイト数であるトータル長 *(total length)* は保たれます。この値は一般的に仕様または alist 単独では固有ではありません。そのかわりこれら両方の情報がこの計算に役立ちます。同様に unpack される文字列や配列の長さは仕様の記述にしたがってデータのトータル長より長くなるかもしれません。

`bindat-length` *spec struct* [Function]

この関数は *struct*内のデータの *spec*に応じたトータル長をリターンする。

`bindat-pack` *spec struct* **&optional** *bindat-raw bindat-idx* [Function]

この関数は alist *struct*内のデータから *spec*に応じて pack されたバイト配列をリターンする。これは通常は先頭から充填された新たなバイト配列を作成する。しかし *bindat-raw*が非 `nil` なら、それは pack 先として事前に割り当てられたユニバイト文字列かベクターを指定する。*bindat-idx*が非 `nil`なら `bindat-raw`へ pack する開始オフセットを指定する。

事前に割り当てる際には out-of-range エラーを避けるために、`(length bindat-raw)`がトータル長またはそれ以上であることを確認すること。

`bindat-ip-to-string` *ip* [Function]

インターネットアドレスのベクター *ip*を通常のドット表記による文字列に変換する。

```
(bindat-ip-to-string [127 0 0 1])
     ⇒ "127.0.0.1"
```

39 Emacs のディスプレー表示

このチャプターでは Emacs によるユーザーへのプレゼンテーションとなる、表示に関連するいくつかの機能を説明します。

39.1 スクリーンのリフレッシュ

関数 `redraw-frame`は与えられたフレーム (Chapter 29 [Frames], page 696 を参照) のコンテンツ全体にたいしてクリアーと再描画を行います。これはスクリーンが壊れている (corrupted) 場合に有用です。

`redraw-frame` **&optional** *frame* [Function]
この関数はフレーム *frame*のクリアーと再表示を行う。*frame*が省略か `nil`なら選択されたフレームを再描画する。

更に強力なのは `redraw-display`です:

`redraw-display` [Command]
この関数はすべての可視なフレームのクリアーと再描画を行う。

Emacs ではユーザー入力は再描画より優先されます。入力が可能なときにこれらの関数を呼び出すと、これらはすぐに再描画はしませんが、要求された再描画はすべての入力処理後に行われます。

テキスト端末では通常は Emacs のサスペントと再開によりスクリーンのリフレッシュも行われます。Emacs のようなディスプレイ指向のプログラムと通常のシーケンシャル表示のプログラムで、コンテンツを区別して記録する端末エミュレーターがいくつかあります。そのような端末を使用する場合には、おそらく再開時の再表示を抑制したいでしょう。

`no-redraw-on-reenter` [User Option]
この変数は Emacs がサスペンドや再開された後にスクリーン全体を再描画するかどうかを制御する。非 `nil`なら再描画は不要、`nil`なら再描画が必要であることを意味する。デフォルトは `nil`。

39.2 強制的な再表示

Emacs は入力の待機時は常に再表示を試みます。以下の関数により実際に入力を待機することなく、Lisp コードの中から即座に再表示を試みることを要求できます。

`redisplay` **&optional** *force* [Function]
この関数は即座に再表示を試みる。オプション引数 *force*が非 `nil`の場合には、入力が保留中なら横取りされるかわりに強制的に再表示が行われる。

この関数は実際に再表示が試行されたなら `t`、それ以外は `nil`をリターンする。`t`という値は再表示の試行が完了したことを意味しない。新たに到着した入力に横取りされた可能性がある。

`redisplay`が即座に再表示を試みたとしても、Emacs がフレーム (複数可) のどの部分を再表示するか決定する方法が変更されるわけではありません。それとは対照的に以下の関数は特定のウィンドウを、(あたかもコンテンツが完全に変更されたかのように) 保留中の再表示処理に追加します。しかし再描画を即座には試みません。

`force-window-update` **&optional** *object* [Function]
この関数は Emacs が次に再表示を行う際にいくつか、あるいはすべてのウィンドウが更新されるよう強制する。*object*がウィンドウならそのウィンドウ、バッファーやバッファー名ならそのバッファーを表示するすべてのウィンドウ、`nil` (または省略) ならすべてのウィンドウが更新される。

この関数は即座に再表示を行わない。再表示は Emacs が入力を待機時、または関数`redisplay`呼び出し時に行われる。

`pre-redisplay-function` [Variable]
再表示の直前に実行される関数。これは再表示されるウィンドウセットを単一の引数として呼び出される。ウィンドウセットは選択されたウィンドウを意味する`nil`、すべてのウィンドウを意味する`t`を指定できる。

`pre-redisplay-functions` [Variable]
このフックは再表示の直前に実行される。これは再表示されようとするウィンドウそれぞれにたいして、そのウィンドウに表示されているバッファーを`current-buffer`にセットして 1 回呼び出される。

39.3 切り詰め

Emacs はテキスト行がウィンドウ右端を超過する際には、その行を継続 *(continue)* させる (次のスクリーン行へ wrap、すなわち折り返す) か、あるいはその行を切り詰め *(truncate)* て表示 (その行をスクリーン行の 1 行に制限) することができます。長いテキスト行を表示するために使用される追加のスクリーン行は継続 *(continuation)* 行と呼ばれます。継続はフィルとは異なります。継続はバッファーのコンテンツ内ではなくスクリーン上でのみ発生して、単語境界ではなく正確に右マージンで行をブレークします。Section 32.11 [Filling], page 797 を参照してください。

グラフィカルなディスプレイでは切り詰めと継続はウィンドウフリンジ内の小さな矢印イメージで示されます (Section 39.13 [Fringes], page 1034 を参照)。テキスト端末では切り詰めはそのウィンドウの最右列の '`$`'、折り返しは最右列の '`\`'で示されます (ディスプレイテーブルでこれを行うための代替え文字を指定できる。Section 39.22.2 [Display Tables], page 1079 を参照)。

`truncate-lines` [User Option]
このバッファーローカル変数が非`nil`ならウィンドウ右端を超過する行は切り詰められて、それ以外なら継続される。特別な例外として部分幅 *(partial-width)* ウィンドウ (フレーム全体の幅を占有しないウィンドウ) では変数`truncate-partial-width-windows`が優先される。

`truncate-partial-width-windows` [User Option]
この変数は部分幅 *(partial-width)* ウィンドウ内の行の切り詰めを制御する。部分幅ウィンドウとはフレーム全体の幅を占有しないウィンドウ (Section 28.6 [Splitting Windows], page 622 を参照)。値が`nil`なら行の切り詰めは変数`truncate-lines` (上記参照) により決定される。値が整数 *n*の場合には、部分幅ウィンドウの列数が *n*より小さければ`truncate-lines`の値とは無関係に行は切り詰められて、部分幅ウィンドウの列数が *n*以上なら行の切り詰めは`truncate-lines`により決定される。それ以外の非`nil`値では`truncate-lines`の値とは無関係にすべての部分幅ウィンドウで行は切り詰められる。

ウィンドウ内で水平スクロール (Section 28.23 [Horizontal Scrolling], page 680 を参照) を使用中は切り詰めが強制されます。

`wrap-prefix` [Variable]
: このバッファーローカル変数が非`nil`なら、それは Emacs が各継続行の先頭に表示する折り返しプレフィックス *(wrap prefix)* を定義する (行を切り詰めている場合には`wrap-prefix`は使用されない)。この値は文字列、またはイメージ (Section 39.16.4 [Other Display Specs], page 1046 を参照) やディスプレイプロパティ`:width`や`:align-to`で指定されるような伸長された空白文字を指定できる (Section 39.16.2 [Specified Space], page 1044 を参照)。値はテキストプロパティ`display`と同じ方法で解釈される。Section 39.16 [Display Property], page 1043 を参照のこと。

  折り返しプレフィックスはテキストプロパティかオーバーレイプロパティ`wrap-prefix`を使用することにより、テキストのリージョンにたいして指定することもできる。これは`wrap-prefix`変数より優先される。Section 32.19.4 [Special Properties], page 820 を参照のこと。

`line-prefix` [Variable]
: このバッファーローカル変数が非`nil`なら、それは Emacs がすべての非継続行の先頭に表示する行プレフィックス *(line prefix)* を定義する。この値は文字列、イメージ (Section 39.16.4 [Other Display Specs], page 1046 を参照)、またはディスプレイプロパティ`:width`や`:align-to`で指定されるような伸長された空白文字を指定できる (Section 39.16.2 [Specified Space], page 1044 を参照)。値はテキストプロパティ`display`と同じ方法で解釈される。Section 39.16 [Display Property], page 1043 を参照のこと。

  行プレフィックスはテキストプロパティまたはオーバーレイプロパティ`line-prefix`を使用することにより、テキストのリージョンにたいして指定することもできる。これは`line-prefix`変数より優先される。Section 32.19.4 [Special Properties], page 820 を参照のこと。

39.4 エコーエリア

エコーエリア *(echo area)* はエラーメッセージ (Section 11.7.3 [Errors], page 164) や`message`プリミティブで作成されたメッセージの表示、およびキーストロークをエコーするために使用されます。(アクティブ時には) ミニバッファーがスクリーン上のエコーエリアと同じ場所に表示されるという事実にも関わらずエコーエリアはミニバッファーと同じではありません。Section “The Minibuffer” in *The GNU Emacs Manual* を参照してください。

このセクションに記述された関数とは別に、出力ストリームとして`t`を指定することによりエコーエリアに Lisp オブジェクトをプリントできます。Section 19.4 [Output Streams], page 335 を参照してください。

39.4.1 エコーエリアへのメッセージの表示

このセクションではエコーエリア内にメッセージを表示する標準的な関数を説明します。

`message` *format-string* **&rest** *arguments* [Function]
: この関数はエコーエリア内にメッセージを表示する。`format-message`関数 (Section 4.7 [Formatting Strings], page 61 を参照) の場合と同じように *format-string*はフォーマット文字列、*arguments*はそのフォーマット仕様にたいするオブジェクトである。フォーマットされた結果文字列はエコーエリア内に表示される。それに`face`テキストプロパティが含まれる場合には指定されたフェイスにより表示される (Section 39.12 [Faces], page 1011 を参照)。この文字列は`*Messages*`バッファーにも追加されるがテキストプロパティは含まれない (Section 39.4.3 [Logging Messages], page 988 を参照)。

  フォーマット内のグレイヴアクセントとアポストロフィーは`"Missing ‘%s’"`から`"Missing ‘foo’"`のように、通常は対応する curved quote として結果内に変換される。この変換に影

響を与えたり抑制する方法については Section 24.4 [Text Quoting Style], page 524 を参照のこと。

バッチモードでは後に改行が付加されたメッセージが標準エラーストリームにプリントされる。

`inhibit-message`が非 `nil`のときはエコーエリアにはメッセージを何も表示せずに‘`*Messages*`’へのロギングだけとなる。

*format-string*が `nil`か空文字列なら、`message`はエコーエリアをクリアーする。エコーエリアが自動的に拡張されていたら、これにより通常のサイズに復元される。ミニバッファーがアクティブなら、これによりスクリーン上に即座にミニバッファーのコンテンツが復元される。

```
(message "Reverting `%s'..." (buffer-name))
 ⊣ Reverting ‘subr.el’...
⇒ "Reverting ‘subr.el’..."

---------- Echo Area ----------
Reverting ‘subr.el’...
---------- Echo Area ----------
```

エコーエリアやポップバッファー内に自動的にメッセージを表示するには、そのサイズに応じて `display-message-or-buffer` (以下参照) を使用する。

警告: 逐語的なメッセージとして独自の文字列を使用したければ、単に (`message` *`string`*) と記述してはならない。*string*に‘`%`’、‘`` ` ``’、‘`'`’が含まれていると望まぬ結果に再フォーマットされるかもしれない。かわりに (`message "%s"` *`string`*) を使用すること。

`set-message-function` [Variable]

この変数が非 `nil`なら、エコーエリア内に表示するためのメッセージテキストを単一の引数とする関数であること。その関数は `message`および関連する関数から呼び出されることになる。その関数が `nil`をリターンすると、通常どおりメッセージはエコーエリアに表示される。関数が文字列をリターンすると、その文字列が元メッセージのかわりにエコーエリアに表示される。その関数が他の非 `nil`値をリターンした場合にはメッセージが処理済みであることを意味するので、`message`はエコーエリアに何も表示しない。その関数で表示されたメッセージのクリアーに使用可能な `clear-message-function`も参照のこと。

デフォルト値はミニバッファーアクティブ時にミニバッファー終端にメッセージを表示する関数。しかしアクティブなミニバッファーに表示されるテキストの何らかの文字が `minibuffer-message`テキストプロパティ(Section 32.19.4 [Special Properties], page 820 を参照) をもつ場合には、メッセージはそのプロパティをもつ最初の文字の前に表示される。

`clear-message-function` [Variable]

この変数が非 `nil`の場合には、`message`および関連する関数は引数となるメッセージが `nil`か空文字列なら、それを引数なしで呼び出す。

この関数は通常はエコーエリアメッセージの表示後、次の入力イベントの到着時に呼び出される。これは `set-message-function`により指定されたカウンターパートとなる関数が表示したメッセージのクリアーを期待される関数である。

デフォルト値はアクティブなミニバッファーに表示されたメッセージをクリアーする関数。

`inhibit-message` [Variable]

この変数が非 `nil`なら、`message`および関連する関数はメッセージの表示にエコーエリアを使用しない。

`with-temp-message` *message* **&rest** *body* [Macro]
この構文は *body* 実行の間にエコーエリア内にメッセージを一時的に表示する。これは *message* を表示して *body* を実行して、それからエコーエリアの前のコンテンツをリストアするとともに body の最後のフォームの値をリターンする。

`message-or-box` *format-string* **&rest** *arguments* [Function]
この関数は `message` と同様にメッセージを表示するが、エコーエリアではなくダイアログボックスにメッセージを表示するかもしれない。この関数があるコマンド内からマウスを使用して呼び出されると— より正確には `last-nonmenu-event` (Section 21.5 [Command Loop Info], page 385 を参照) が `nil` かリストならメッセージの表示にダイアログボックスかポップアップメニュー、それ以外ならエコーエリアを使用する (これは `y-or-n-p` が同様の決定を行う際に使用する条件と同じ。Section 20.7 [Yes-or-No Queries], page 366 を参照)。

呼び出しの前後で `last-nonmenu-event` を適切な値にバインドすることによりエコーエリアでのマウスの使用を強制できる。

`message-box` *format-string* **&rest** *arguments* [Function]
この関数は `message` と同様にメッセージを表示するが、利用可能なら常にダイアログボックス (かポップアップメニュー) を使用する。端末がサポートしないためにダイアログボックスやポップアップメニューが使用できなければ、`message-box` は `message` と同様にエコーエリアを使用する。

`display-message-or-buffer` *message* **&optional** *buffer-name action frame* [Function]
この関数はメッセージ *message* を表示する。*message* には文字列かバッファーを指定できる。これが `max-mini-window-height` で定義されるエコーエリアの最大高さより小さければ、`message` を使用してエコーエリアに表示される。それ以外ならメッセージを表示するために `display-buffer` はポップアップバッファーを使用する。

エコーエリアに表示したメッセージ、またはポップアップバッファー使用時はその表示に使用したウィンドウをリターンする。

message が文字列ならオプション引数 *buffer-name* はポップアップバッファー使用時にメッセージ表示に使用するバッファー名 (デフォルトは `*Message*`)。*message* が文字列でエコーエリアに表示されていれば、いずれにせよコンテンツをバッファーに挿入するかどうかは指定されない。

オプション引数 *action* と *frame* は `display-buffer` の場合と同様に、バッファーが表示されている場合のみ使用される。

`current-message` [Function]
この関数はエコーエリア内にカレントで表示されているメッセージ、またはそれが存在しなければ `nil` をリターンする。

39.4.2 処理の進捗レポート

処理の完了まで暫く時間を要するかもしれない際には、進行状況についてユーザーに通知するべきです。これによりユーザーが残り時間を予測するとともに、Emacs が hung しているのではなく処理中であることを明確に確認できます。プログレスリポーター *(progress reporter:* 進行状況リポーター*)* を使用するのが、これを行う便利な方法です。

以下は何も有用なことを行わない実行可能な例です:

```
(let ((progress-reporter
```

```
      (make-progress-reporter "Collecting mana for Emacs..."
                              0  500)))
  (dotimes (k 500)
    (sit-for 0.01)
    (progress-reporter-update progress-reporter k))
  (progress-reporter-done progress-reporter))
```

`make-progress-reporter` *message* **&optional** *min-value max-value current-value min-change min-time* [Function]

この関数は以下に挙げる他の関数の引数として使用されることになるプログレスリポーターオブジェクトを作成してリターンする。これはプログレスリポーターを高速にするように、可能なかぎり多くのデータを事前に計算するというアイデアが元となっている。

この後にプログレスリポーターを使用する際には、進行状況のパーセンテージを後に付加して *message*が表示されるだろう。*message*は単なる文字列として扱われる。たとえばファイル名に依存させる必要があるなら、この関数の呼び出し前に `format-message`を使えばよい。

引数 *min-value*と *max-value*は処理の開始と終了を意味する数値であること。たとえばバッファーをスキャンする処理なら、これらをそれぞれ `point-min`と `point-max`にセットするべきだろう。*max-value*は *min-value*より大であること。

かわりに *min-value*と *max-value*を `nil`にセットすることができる。この場合にはプログレスリポーターは進行状況のパーセンテージを報告しない。かわりにプログレスリポーターを更新するたびに刻み (notch) を回転する “スピナー (spinner)” を表示する。

*min-value*と *max-value*が数値なら、進行状況の初期の数値を与える引数 *current-value*を与えることができる。省略時のデフォルトは *min-value*。

残りの引数はエコーエリアの更新レートを制御する。プログレスリポーターは次のメッセージを表示する前に、その処理が少なくとも *min-change*パーセントより多く完了するまで待機する。デフォルトは 1 パーセント。*min-time*は連続するプリントの間に空ける最小時間をミリ秒単位で指定する (いくつかのオペレーティングシステムではプログレスリポーターは秒の少数部をさまざまな精度で処理するかもしれない)。

この関数は `progress-reporter-update`を呼び出すので、最初のメッセージは即座にプリントされる。

`progress-reporter-update` *reporter* **&optional** *value* [Function]

この関数は操作の進行状況報告に関する主要な機能を担う。これは *reporter*のメッセージと、その後に *value*により決定された進行状況のパーセンテージを表示する。パーセンテージが 0、または引数 *min-change*と *min-time*に比べて十分 0 に近ければ出力は省略される。

*reporter*は `make-progress-reporter`呼び出しがリターンした結果でなければならない。*value*は処理のカレント状況を指定して、`make-progress-reporter`に渡された *min-value*と *max-value*の間 (両端を含む) でなければならない。たとえばバッファーのスキャンにおいては、*value*は `point`び呼び出し結果であるべきだろう。

オプション引数 *suffix*は、*reporter*のメインメッセージと進行状況テキストの後に表示する文字列。*reporter*が非数値のリポーターなら *value*は `nil`、または *suffix*のかわりに使用する文字列であること。

この関数は `make-progress-reporter`に渡された *min-change*と *min-time*にしたがい、毎回の呼び出しで新たなメッセージを出力しない。したがってこれは非常に高速であり、通常はこれを呼び出す回数を減らすことを試みるべきではない。結果として生じるオーバーヘッドは、あなたの努力をほぼ否定するだろう。

`progress-reporter-force-update` *reporter* **&optional** *value new-message suffix* [Function]

この関数は `progress-reporter-update` と同様だが、これは無条件にメッセージをエコーエリアにプリントする点が異なる。

reporter、*value*、*suffix* は `progress-reporter-update` の場合と同じ意味をもつ。オプションの *new-message* で *reporter* のメッセージを変更できる。この関数は常にエコーエリアを更新するので、そのような変更は即座にユーザーに示されるだろう。

`progress-reporter-done` *reporter* [Function]

この関数は処理の完了時に呼び出されること。これはエコーエリア内に単語 '`done`' を付加した *reporter* のメッセージを表示する。

`progress-reporter-update` に '`100%`' とプリントさせようとせずに、常にこの関数を呼び出すこと。まずこの関数は決してそれをプリントしないだろうし、これが発生しないために多くの正当な理由がある。次に '`done`' はより自明である。

`dotimes-with-progress-reporter` (*var count* [*result*]) [Macro]
reporter-or-message body...

これは `dotimes` と同じ方法で機能するが、上述の関数を使用してループ進行状況 (loop progress) の報告も行う便利なマクロである。これによりタイプ量を幾分節約できる。引数 *reporter-or-message* は文字列、またはプログレスリポーターオブジェクト。

以下の方法でこのマクロを使用することにより、このサブセクションの例を書き換えることができる:

```
(dotimes-with-progress-reporter
    (k 500)
    "Collecting some mana for Emacs..."
  (sit-for 0.01))
```

make-progress-reporter のオプション引数を指定したい場合には、*reporter-or-message* 引数としてリポーターオブジェクトを使用するのが便利。たとえば前出の例は以下のように書き換えられる:

```
(dotimes-with-progress-reporter
    (k 500)
    (make-progress-reporter "Collecting some mana for Emacs..." 0 500 0 1 1.5)
  (sit-for 0.01))
```

`dolist-with-progress-reporter` (*var count* [*result*]) [Macro]
reporter-or-message body...

これは `dolist` と同じ方法で機能するが、上述の関数を使用してループ進行状況 (loop progress) の報告も行う便利なマクロである。これによりタイプ量を幾分節約できる。`dotimes-with-progress-reporter` の場合のように、`reporter-or-message` はプログレスリポーターか文字列。このマクロにより、前出の例を以下のように書き換えられる:

```
(dolist-with-progress-reporter
    (k (number-sequence 0 500))
    "Collecting some mana for Emacs..."
  (sit-for 0.01))
```

39.4.3 *Messages*へのメッセージのロギング

エコーエリア内に表示されるほとんどすべてのメッセージは、ユーザーが後で参照できるように*Messages*バッファー内にも記録されます。これにはmessageにより出力されたメッセージも含まれます。デフォルトではこのバッファーは読み取り専用でメジャーモードmessages-buffer-modeを使用します。ユーザーによる*Messages*バッファーの kill を妨げるものは何もありませんが、次回のメッセージ表示でバッファーは再作成されます。*Messages*バッファーに直接アクセスする必要があり、それが確実に存在するようにしたい Lisp コードは、すべて関数messages-bufferを使用するべきです。

messages-buffer [Function]
この関数は*Messages*バッファーをリターンする。バッファーが存在しなければ作成してバッファーを messages-buffer-modeに切り替える。

message-log-max [User Option]
この変数は*Messages*バッファー内に保持するべき行数を指定する。値 tは保持すべき行数に制限がないことを意味して、値 nilはメッセージのロギングを完全に無効にする。以下はメッセージを表示して、それがロギングされることを防ぐ例:

```
(let (message-log-max)
  (message ...))
```

*Messages*にたいするユーザーの利便性を向上させるために、ロギング機能は連続する同じメッセージを結合します。さらに 2 つのケースのために連続する関連メッセージの結合も行います。2 つのケースとは応答を後にともなう質問 (question followed by answer)、および一連のプログレスメッセージ (series of progress messages) です。

応答を後にともなう質問 (question followed by an answer) とは、1 つ目が '*question*'、2 つ目が '*question*...*answer*'のように y-or-n-pが生成するような 2 つのメッセージをもつ応答です。1 つ目のメッセージは 2 つ目のメッセージ以上の追加情報を伝えないので、2 つ目のメッセージをロギングして 1 つ目のメッセージは破棄します。

一連のプログレスメッセージ (series of progress messages) は、make-progress-reporterが生成するような連続するメッセージをもちます。これらは '*base*...*how-far*'のような形式であり、*how-far*は毎回異なりますが *base*は常に同じです。このシリーズ内の各メッセージのロギングでは、そのメッセージが前のメッセージと連続していれば前のメッセージを破棄します。

関数 make-progress-reporterと y-or-n-pは、メッセージログ結合機能をアクティブにするために何ら特別なことを行う必要はありません。これは '...'で終わる共通のプレフィックスを共有する連続する 2 つのメッセージをログする際には常にこの処理を行います。

39.4.4 エコーエリアのカスタマイズ

以下の変数はエコーエリアが機能する方法の詳細を制御します。

cursor-in-echo-area [Variable]
この変数はエコーエリア内にメッセージ表示時にカーソルを表示する場所を制御する。これが非 nilならカーソルはメッセージの終端、それ以外ならカーソルはエコーエリア内ではなくポイント位置に表示される。

この値は通常は nil。Lisp プログラムは短時間の間、これを tにバインドする。

echo-area-clear-hook [Variable]
このノーマルフックは (message nil)、または別の何らかの理由によりエコーエリアが作成されると常に実行される。

`echo-keystrokes` [User Option]
: この変数はコマンド文字をエコーする前に、どれだけの時間を待機するかを決定する。この値は数字でなければならず、エコー前に待機する秒数を指定する。ユーザーが (`C-x`のような) プレフィックスキーをタイプしてから、継続してタイプを継続するのをこの秒数遅延した場合、エコーエリア内にそのプレフィックスキーがエコーされる (あるキーシーケンスで一度エコーが開始されると、同一のキーシーケンス内の後続するすべての文字は即座にエコーされる)。

  値が 0 ならコマンド入力はエコーされない。

`message-truncate-lines` [Variable]
: 長いメッセージの表示により、そのメッセージ全体を表示するために、通常はエコーエリアはリサイズされる。しかし変数 `message-truncate-lines`が非 `nil`なら、エコーエリアをリサイズせずエコーエリアに収まるようメッセージは切り詰められる。

ミニバッファーウィンドウのリサイズの最大高さを指定する変数 `max-mini-window-height`はエコーエリアにも適用される (エコーエリアは真にミニバッファーウィンドウの特殊な使い方である。Section 20.11 [Minibuffer Windows], page 371 を参照)。

39.5 警告のレポート

警告 *(warnings)* とはプログラムがユーザーにたいして問題の可能性を知らせるが、実行は継続するための機能です。

39.5.1 警告の基礎

すべての警告は、ユーザーに問題を説明するためのテキストのメッセージと、重大レベル *(severity level)* をもっています。重大レベルはシンボルです。以下は可能性のある重大レベルとその意味を、重大度の降順でリストしたものです:

`:emergency`
: 直ちに対処しなければ Emacs 処理が間もなく深刻に害される問題。

`:error` 本質的に悪いデータや状況のリポート。

`:warning` 本質的に悪くはないが、可能性のある問題を励起する恐れのあるデータや状況のリポート。

`:debug` デバッグ中なら有用かもしれない情報のリポート。

あなたのプログラムが無効な入力データに遭遇した際には、`error`呼び出しによる Lisp エラーのシグナルするか、または重大度`:error`の警告をリポートすることができます。Lisp エラーのシグナルはもっとも簡単に行えることですが、それはプログラムが処理を継続できないことを意味します。間違ったデータでも処理を継続するための方法を実装するためにトラブルを受け取めたい場合には、その問題をユーザーに知らせるために重大度`:error`の警告をリポートするのが正しい方法です。たとえば Emacs Lisp バイトコンパイラーはこの方法によりエラーを報告して、他の関数のコンパイルを継続できます (プログラムが Lisp エラーをシグナルして `condition-case`で handle したならユーザーがそのエラーを確認することはないだろう。これは警告としてリポートすることによりユーザーにメッセージを示すことができる)。

クラス分けのために警告にはそれぞれ警告タイプ *(warning type)* があります。このタイプはシンボルのリストです。最初のシンボルはそのプログラムのユーザーオプションとして使用するカスタムグループであるべきです。たとえばバイトコンパイラーの警告は警告タイプ `(bytecomp)`を使用します。もし望むなら、このリスト内で更にシンボルを使用することにより警告をサブカテゴリー化することもできます。

`display-warning` *type message* **&optional** *level buffer-name* [Function]
この関数はメッセージとして *message*、警告タイプとして *type*を使用して警告をリポートする。*level*は重大レベルであること。デフォルトは`:warning`。

*buffer-name*が非`nil`なら、それは警告をロギングするためのバッファー名を指定する。デフォルトは`*Warnings*`。

`lwarn` *type level message* **&rest** *args* [Function]
この関数は`*Warnings*`バッファー内のメッセージとして `(format-message` *message args...*`)`の値を使用して警告をリポートする。他の点ではこれは`display-warning`と同じ。

`warn` *message* **&rest** *args* [Function]
この関数はメッセージとして `(format-message` *message args...*`)`の値、タイプとして`(emacs)`、重大レベルとして`:warning`を使用して警告をリポートする。これは互換性のためだけに存在する。固有な警告タイプを指定するべきであり、この関数の使用は推奨しない。

39.5.2 警告のための変数

このセクション内で説明する変数をバインドすることにより、プログラムは警告が表示される方法をカスタマイズできます。

`warning-levels` [Variable]
このリストは警告の重大レベルの意味と重大度の順序を定義する。それぞれの要素は 1 つの重大レベルを定義して、それらを重大度の降順で配置した。

各要素は `(`*level string function*`)`という形式をもち、*level*はその要素が定義する重大レベル。*string*はそのレベルのテキストによる説明。*string*は警告タイプ情報の配置箇所の指定に '`%s`'を使用するか、さもなくばその情報を含まぬよう '`%s`'を省略できる。

オプションの *function*が非`nil`なら、これはユーザーの注目を得るために引数なしで呼び出される関数であること。

通常はこの変数の値を変更しないこと。

`warning-prefix-function` [Variable]
値が非`nil`なら、それは警告用にプレフィックスを生成する関数であること。プログラムはこの変数を適切な関数にバインドできる。`display-warning`は warnings バッファーがカレントの状態でこの関数を呼び出して、関数はそのバッファーにテキストを挿入できる。そのテキストが警告メッセージの先頭になる。

この関数は重大レベル、および`warning-levels`内でのその重大レベルのエントリーという 2 つの引数で呼び出される。これはエントリーとして使用するためのリストをリターンすること (この値は`warning-levels`の実際のメンバーである必要はない)。この値を構築することにより関数はその警告の重大レベルを変更したり、与えられた重大レベルにたいして異なる処理を指定することができる。

この変数の値が`nil`なら呼び出される関数は存在しない。

`warning-series` [Variable]
プログラムは次の警告がシリーズの開始であることを告げるために、この変数を`t`にバインドできる。複数の警告がシリーズを形成するということは、それぞれの警告にたいしてポイントが維持されるように移動して、最後の警告にポイントが表示されるのではなくシリーズの最初の警告にポイントを残すことを意味する。このシリーズは、そのローカルバインドが非バインドされて`warning-series`が再び`nil`になったときに終了する。

この値は関数定義をもつシンボルでもよい。これは次の警告により warnings バッファーがカレントの状態で、引数なしでその関数が呼び出されることを除き tと等価。この関数は警告シリーズのヘッダーの役目をもつであろうテキストを挿入できる。

あるシリーズが開始されると、その値は warnings バッファー内でシリーズ開始となるバッファー位置を指すマーカーとなる。

この変数の通常の値は nilで、これはそれぞれの警告を個別に処理することを意味する。

warning-fill-prefix [Variable]
この変数が非 nilなら、それは各警告テキストのフィルに使用するフィルプレフィックスを指定する。

warning-fill-column [Variable]
警告をフィルする列。

warning-type-format [Variable]
この変数は警告メッセージ内の警告タイプを表示するためのフォーマットを指定する。この方法でフォーマットされたタイプは、warning-levels内のエントリー内の文字列制御下にあるメッセージに含まれることになる。デフォルト値は" (%s)"。これを""にバインドすると警告タイプはまったく表示されなくなる。

39.5.3 警告のためのオプション

以下の変数は何が発生したときに Lisp プログラムが警告をリポートするかをユーザーが制御するために使用されます。

warning-minimum-level [User Option]
このユーザーオプションはユーザーにたいして即座に表示されるべき最小の重大レベルを指定する。デフォルトは:warningであり、これは:debug警告を除くすべての警告が即座に表示されることを意味する。

warning-minimum-log-level [User Option]
このユーザーオプションは warnings バッファー内にログされるべき最小の重大レベルを指定する。デフォルトは:warningであり、これは:debug警告を除くすべての警告がログされることを意味する。

warning-suppress-types [User Option]
このリストはユーザーにたいしてどの警告タイプを即座に表示するべきではないかを指定する。このリスト内の各要素はシンボルのリストであること。それの要素が警告タイプ内の最初の要素にマッチしたら警告は即座に表示されない。

warning-suppress-log-types [User Option]
このリストはユーザーにたいしてどの警告タイプが warnings バッファーにログされるべきではないかを指定する。このリスト内の各要素はシンボルのリストであること。それの要素が警告タイプ内の最初の数要素にマッチしたら警告はログされない。

39.5.4 遅延された警告

コマンド実行中には警告の表示を避けてコマンドの終わりでのみ警告を表示したいことがあるかもしれません。これは関数 delay-warningを使用して行うことができます。

`delay-warning` *type message* **&optional** *level buffer-name* [Function]
この関数は `display-warning` (Section 39.5.1 [Warning Basics], page 989 を参照) の遅延対応版であり、同じ引数で呼び出される。警告メッセージは `delayed-warnings-list`にキューイングされる。

`delayed-warnings-list` [Variable]
この変数の値はカレントのコマンド完了後に表示される警告のリスト。各要素は以下のようなリストでなければならない:

```
(type message [level [buffer-name]])
```

これらは`display-warning`の引数リストと同じ形式、同じ意味である。`post-command-hook` (Section 21.1 [Command Overview], page 375 を参照) の実行直後に、Emacs のコマンドループはこの変数で指定されたすべての警告を表示してから変数を `nil`にリセットする。

遅延警告メカニズムをよりカスタマイズする必要があるプログラムは変数 `delayed-warnings-hook`を変更することができます:

`delayed-warnings-hook` [Variable]
これは遅延警告を処理して表示するために、`post-command-hook`の後に Emacs コマンドループが実行するノーマルフック。

デフォルト値は 2 つの関数からなるリスト:

```
(collapse-delayed-warnings display-delayed-warnings)
```

関数 `collapse-delayed-warnings`は `delayed-warnings-list`から重複するエントリーを削除する。関数 `display-delayed-warnings`は `delayed-warnings-list`内の各要素にたいして順次`display-warning`を呼び出してから、`delayed-warnings-list`を `nil`にセットする。

39.6 不可視のテキスト

`invisible`プロパティにより、スクリーン上に表示されないように文字を不可視 *(invisible)* にすることができます。これはテキストプロパティ(Section 32.19 [Text Properties], page 813 を参照)、またはオーバーレイプロパティ(Section 39.9 [Overlays], page 999 を参照) のいずれかで行うことができます。カーソル移動もこれらの文字を部分的に無視します。あるコマンドの後に不可視テキスト範囲内にポイントがあることをコマンドループが検知した場合には、コマンドループはポイントをそのテキストの別サイドへ再配置します。

もっともシンプルなケースでは、非`nil`の `invisible`プロパティにより文字は不可視になります。これがデフォルトのケースであり、もし`buffer-invisibility-spec`のデフォルト値を変更したくない場合には、これが`invisible`プロパティを機能させる方法です。自身で`buffer-invisibility-spec`をセットする予定がなければ、`invisible`プロパティの値として通常は `t`を使用するべきです。

より一般的にはどの `invisible`の値がテキストを不可視にするかを制御するために変数 `buffer-invisibility-spec`を使用できます。テキストにたいして異なる `invisible`の値を与えることにより、事前に別のサブセットへテキストをクラス分けした後に `buffer-invisibility-spec`の値を変更して、さまざまなサブセットを可視や不可視にすることができます。

特にデータベース内のエントリーのリストを表示するプログラム内では、`buffer-invisibility-spec`による可視性の制御は有用です。これによりデータベース内の一部だけを閲覧するフィルターコマンドを簡便に実装することが可能になります。この変数をセットするのは非常に高速であり、バッファー内のすべてのテキストにたいしてプロパティが変更されたかスキャンするよりはるかに高速です。

`buffer-invisibility-spec` [Variable]

この変数はどの種類の `invisible`プロパティが実際に文字を不可視にするかを指定する。この変数はセットすることによりバッファーローカルになる。

`t` `invisible`プロパティが非 `nil`ならその文字は不可視になる。これがデフォルト。

リスト このリスト内の各要素は不可視性の条件を指定する。ある文字の `invisible`プロパティがこれらの条件のいずれかに適合したら、その文字は不可視になる。このリストは 2 種類の要素をもつことができる:

atom `invisible`プロパティの値が *atom*、または *atom*をメンバーにもつリストならその文字は不可視になる。比較は `eq`により行われる。

(*atom* . `t`)
`invisible`プロパティの値が *atom*、または *atom*をメンバーにもつリストならその文字は不可視になる。比較は `eq`により行われる。さらにそのような文字シーケンスは省略記号 (ellipsis) として表示される。

特に `buffer-invisibility-spec`への要素の追加と削除のために 2 つの関数が提供されています。

`add-to-invisibility-spec` *element* [Function]

この関数は、`buffer-invisibility-spec`に要素 *element*を追加する。`buffer-invisibility-spec`が `t`なら、これはリスト `(t)`に変更されて `invisible`プロパティが `t`のテキストは不可視のまま留まる。

`remove-from-invisibility-spec` *element* [Function]

この関数は `buffer-invisibility-spec`から要素 *element*を削除する。リスト内に *element* がなければ何も行わない。

`buffer-invisibility-spec`を使用するための規約として、メジャーモードは `buffer-invisibility-spec`の要素、および `invisible`プロパティの値として自身のモード名を使用することになっている。

```
;; 省略記号を表示したければ:
(add-to-invisibility-spec '(my-symbol . t))
;; 表示したくなければ:
(add-to-invisibility-spec 'my-symbol)

(overlay-put (make-overlay beginning end)
             'invisible 'my-symbol)

;; 不可視状態が終わったら:
(remove-from-invisibility-spec '(my-symbol . t))
;; または各々を:
(remove-from-invisibility-spec 'my-symbol)
```

以下の関数を使用することにより不可視性をチェックできます:

`invisible-p` *pos-or-prop* [Function]

この関数は *pos-or-prop*がマーカーか数字の場合には、その位置のテキストがカレントで不可視なら非 `nil`をリターンする。

*pos-or-prop*が別の類の Lisp オブジェクトなら、テキストプロパティかオーバーレイプロパティとして可能な値を意味すると解釈される。この場合には`buffer-invisibility-spec`のカレント値にもとづき、もしその値がテキストを不可視とするようならこの関数は非`nil`をリターンする。

この関数のリターン値はテキストがディスプレー上で完全に非表示なら`t`、省略記号 (ellipsis) で置き換えられていれば`nil`でも`t`でもない値となる。

テキストを操作したりポイントを移動する関数は、通常はそのテキストが不可視かどうかに注意を払わずに、可視と不可視のテキストを同様に処理します。`next-line`や`previous-line`のようなユーザーレベルの行移動関数は`line-move-ignore-invisible`が非`nil` (デフォルト) なら不可視な改行を無視します。これらの関数は不可視な改行がそのバッファーに存在しないかのように振る舞いますが、それはそう振る舞うように明示的にプログラムされているからです。

あるコマンドが不可視テキストの境界内側のポイントで終了した場合には、メイン編集ループはその不可視テキストの両端のうちのいずれかにポイントを再配置します。そのコマンドの移動関数の全体的な方向と同じになるように Emacs が再配置の方向を決定します。これに疑問がある場合には、挿入された文字が`invisible`プロパティを継承しないような位置を優先してください。加えてそのテキストが省略記号で置換されずに、コマンドが不可視テキスト内への移動のみを行う場合には、ポイントを 1 文字余計に移動して目に見えるようカーソルを移動することにより、そのコマンドの移動を反映するよう試みます。

したがってコマンドが (通常の stickiness をもつ) 不可視範囲に後方へとポイントを移動すると、Emacs はポイントをその範囲の先頭に後方へと移動します。コマンドが不可視範囲へ前方にポイントを移動した場合には、Emacs は不可視テキストの前にある最初の可視文字に前方へとポイントを移動して、その後さらに前方へ 1 文字余計に移動します。

これら不可視テキスト中間で終了するポイントにたいするこれらの調整 *(adjustments)* は、`disable-point-adjustment`を非`nil`にセットすることにより無効にできます。Section 21.6 [Adjusting Point], page 387 を参照してください。

インクリメンタル検索はマッチが不可視テキストを含む場合には、一時的および/または永続的に不可視オーバーレイを可視にすることができます。これを有効にするためには、そのオーバーレイが非`nil`の`isearch-open-invisible`プロパティをもつ必要があります。プロパティの値は、そのオーバーレイを引数として呼び出される関数であるべきです。その関数はオーバーレイを永続的に可視にする必要があります。これは検索からの exit 時にマッチがそのオーバーレイに重なるときに使用されます。

検索の間にそのようなオーバーレイの invisible、および intangible プロパティを一時的に変更することによりオーバーレイは一時的に可視にされます。特定のオーバーレイにたいして異なる方法でこれを行いたいなら、それを`isearch-open-invisible-temporary`プロパティ(関数) に与えてください。その関数は 2 つの引数により呼び出されます。1 つ目はそのオーバーレイ、2 つ目は`nil`ならオーバーレイを可視、`t`なら再び不可視にします。

39.7 選択的な表示

選択的表示 *(selective display)* とはスクリーン上で特定の行を隠蔽する関連する機能ペアーを指します。

1 つ目の変種は明示的な選択的表示であり、これは Lisp プログラム内で使用するようにデザインされています。これはテキスト変更により、どの行を隠すかを制御します。この種の隠蔽は現在では時代遅れであ推奨されていません。同じ効果を得るには、かわりに`invisible`プロパティ(Section 39.6 [Invisible Text], page 992 を参照) を使用する必要があります。

2 つ目の変種はインデントにもとづいて隠す行の選択を自動的に行います。この変種はユーザーレベルの機能としてデザインされています。

選択的表示を明示的に制御する方法では改行 (control-j) を復帰 (control-m) に置換します。これにより以前は行末に改行があった行は隠蔽されます。厳密に言うと改行だけが行を分離できるので、これはもはや一時的には行ではなく前の行の一部です。

選択的表示は編集コマンドに直接影響を与えません。たとえば *C-f* (`forward-char`) は隠蔽された行へと気軽にポイントを移動します。しかし復帰文字による改行文字の置換は、いくつかの編集コマンドに影響を与えます。たとえば `next-line`は改行だけを検索するために、隠蔽された行をスキップします。選択的表示を使用するモードは改行を考慮するコマンドを定義したり、テキストのどの部分を隠すか制御することもできます。

選択的表示されたバッファーをファイルに書き込む際には、control-m はすべて改行として出力されます。これはファイル内のテキストを読み取る際には、すべて問題なく隠蔽されずに表示されることを意味します。選択的表示は Emacs 内でのみ顕在する効果です。

`selective-display` [Variable]

このバッファーローカル変数は選択的表示を有効にする。これは行、または行の一部を隠すことができることを意味する。

- `selective-display`の値が `t`なら、文字 control-m が隠蔽されたテキストの開始をマークする。control-m と後続する行の残りは表示されない。これは明示的な選択的表示である。
- `selective-display`の値が正の整数なら、それより多くの列によるインデントで始まる行は表示されない。

バッファーの一部が隠蔽されている際には垂直移動コマンドはあたかもその部分を存在しないかのように処理して、1 回の `next-line`コマンドで任意の行数の隠蔽された行をスキップできる。しかし (`forward-char`のような) 文字移動コマンドは隠蔽された部分をスキップせずに、(注意すれば) 隠蔽された部分にたいしてテキストの挿入と削除が可能である。

以下の例では `selective-display`の値の変更によるバッファー `foo`の外観表示を示す。このバッファーのコンテンツは変更されない。

```
(setq selective-display nil)
     ⇒ nil

---------- Buffer: foo ----------
1 on this column
 2on this column
  3n this column
  3n this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

```
(setq selective-display 2)
     ⇒ 2

---------- Buffer: foo ----------
1 on this column
 2on this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

`selective-display-ellipses` [User Option]

このバッファーローカル変数が非`nil`なら、Emacs は隠蔽されたテキストを後にともなう行の終端に '`...`'を表示する。以下は前の例からの継続。

```
(setq selective-display-ellipses t)
     ⇒ t

---------- Buffer: foo ----------
1 on this column
 2on this column ...
 2on this column
1 on this column
---------- Buffer: foo ----------
```

省略記号 ('`...`') にたいして他のテキストを代替えするためにディスプレイテーブルを使用できる。Section 39.22.2 [Display Tables], page 1079 を参照のこと。

39.8 一時的な表示

一時的表示 (temporary display) は出力をバッファーに配置して編集用ではなく閲覧用としてユーザーに示すために Lisp プログラムにより使用されます。多くのヘルプコマンドはこの機能を使用します。

`with-output-to-temp-buffer` *buffer-name body...* [Macro]

この関数は *buffer-name*という名前のバッファー (必要なら最初に作成される) にプリントされた任意の出力が挿入されるようアレンジ、さらにバッファーを Help モードにして *body*内のフォームを実行する (類似する以下のフォーム`with-temp-buffer-window`を参照)。最後にそのバッファーはいずれかのウィンドウに表示されるが、そのウィンドウは選択されない。

*body*内のフォームが出力バッファーのメジャーモードを変更しないため、実行の最後においても依然として Help モードにあるなら、`with-output-to-temp-buffer`は最後にそのバッファーを読み取り専用するとともに、クリック可能なクロスリファレンスとなるように関数名と変数名のスキャンも行う。特にドキュメント文字列内のハイパーリンク上アイテムに関する詳細は [Tips for Documentation Strings], page 1169 を参照のこと。

文字列 *buffer-name*は一時的なバッファーを指定して、これはあらかじめ存在する必要はない。引数はバッファーではなく文字列でなければならない。そのバッファーは最初に消去されて (確認なし)、`with-output-to-temp-buffer`の exit 後は未変更 (unmodified) とマークされる。

`with-output-to-temp-buffer`は `standard-output`を一時的バッファーにバインドして *body*内のフォームを評価する。*body*内の Lisp 出力関数を使用した出力のデフォルト出力先は、そのバッファーになる (しかしスクリーン表示やエコーエリア内のメッセージは一般的な

世界の感覚では “出力” であるものの影響は受けない)。Section 19.5 [Output Functions], page 337 を参照のこと。

この構構文の振る舞いをカスタマイズするために利用できるフックがいくつかあり、それらは以下にリストしてある。

リターン値は *body*内の最後のフォームの値。

```
---------- Buffer: foo ----------
 This is the contents of foo.
---------- Buffer: foo ----------

(with-output-to-temp-buffer "foo"
    (print 20)
    (print standard-output))
⇒ #<buffer foo>

---------- Buffer: foo ----------

20

#<buffer foo>

---------- Buffer: foo ----------
```

`temp-buffer-show-function` [User Option]

この変数が非`nil`なら、`with-output-to-temp-buffer`はヘルプバッファーを表示する処理を行うためにその関数を呼び出す。この関数は表示すべきバッファーという 1 つの引数を受け取る。

`with-output-to-temp-buffer`が通常行うように、`save-selected-window`内部や選択されたウィンドウ内でバッファーか選択された状態で `temp-buffer-show-hook`を実行するのは、この関数にとってよいアイデアである。

`temp-buffer-setup-hook` [Variable]

このノーマルフックは *body*を評価する前に `with-output-to-temp-buffer`により実行される。フック実行時には一時的バッファーがカレントになる。このフックは通常はそのバッファーを Help モードにするための関数にセットアップされる。

`temp-buffer-show-hook` [Variable]

このノーマルフックは一時的バッファー表示後に `with-output-to-temp-buffer`により実行される。フック実行時には一時的バッファーがカレントになり、それが表示されているウィンドウが選択される。

`with-temp-buffer-window` *buffer-or-name action quit-function body...* [Macro]

このマクロは `with-output-to-temp-buffer`と類似している。`with-output-to-temp-buffer`構文と同様に、これはプリントされる任意の出力が *buffer-or-name*という名前のバッファーに挿入されるようにアレンジして *body*を実行して、そのバッファーをいずれかのウィンドウに表示する。しかし `with-output-to-temp-buffer`とは異なり、このマクロはそのバッファーを自動的に Help モードに切り替えない。

引数 *buffer-or-name*は一時的バッファーを指定する。これはバッファー (既存でなければならない)、または文字列を指定でき、文字列の場合には必要ならその名前のバッファーが作成される。そのバッファーは `with-temp-buffer-window`の exit 時には、未変更かつ読み取り専用とマークされる。

このマクロは `temp-buffer-show-function`を呼び出さない。かわりにそのバッファーを表示するために *action*引数を `display-buffer` (Section 28.13.1 [Choosing Window], page 640 を参照) に渡す。

引数 *quit-function*が指定されていなければ *body*内の最後のフォームの値がリターンされる。指定されている場合には、そのバッファーを表示するウィンドウと *body*の結果という 2 つの引数で呼び出される。その場合には、最終的なリターン値は何であれ *quit-function*がリターンした値となる。

このマクロは `with-output-to-temp-buffer`により実行される類似フックのかわりにノーマルフック `temp-buffer-window-setup-hook`と `temp-buffer-window-show-hook`を使用する。

次の 2 つの構文は `with-temp-buffer-window`とほとんど同じですが、説明している点が異なります：

`with-current-buffer-window` *buffer-or-name action quit-function* **&rest** *body* [Macro]

このマクロは `with-temp-buffer-window`と同様だが、*body*の実行に際して *buffer-or-name*で指定したバッファーをカレントにする点が異なる。

`with-displayed-buffer-window` *buffer-or-name action quit-function* **&rest** *body* [Macro]

このマクロは `with-temp-buffer-window`と同様だが、*body*の実行の前に *buffer-or-name*で指定したバッファーを表示する点が異なる。

一時バッファーを表示しているウィンドウは以下のモードを使用してそのバッファーにサイズを適合できます:

`temp-buffer-resize-mode` [User Option]

このマイナーモードが有効なときは、一時的バッファーを表示しているウィンドウはバッファーのコンテンツにフィットするように自動的にリサイズされる。

そのバッファーにたいして特別に作成されたウィンドウの場合のみウィンドウはリサイズされる。特に前に別のバッファーを表示していたウィンドウはリサイズされない。デフォルトではこのモードはリサイズに `fit-window-to-buffer`を使用する (Section 28.4 [Resizing Windows], page 617 を参照)。以下のオプション `temp-buffer-max-height`と `temp-buffer-max-width`をカスタマイズして他の関数を指定できる。

`temp-buffer-max-height` [User Option]

このオプションは `temp-buffer-resize-mode`が有効な際に一時的バッファーを表示するウィンドウの最大高さ (行数) を指定する。その種のバッファーの高さ選択のために呼び出す関数でもよい。これはバッファーを唯一の引数として受け取り、正の整数をリターンすること。関数の呼び出し時にはリサイズされるウィンドウが選択される。

`temp-buffer-max-width` [User Option]

このオプションは `temp-buffer-resize-mode`が有効な際に一時的バッファーを表示するウィンドウの最大幅 (列数) を指定する。その種のバッファーの高さ選択のために呼び出す関数でも

よい。これはバッファーを唯一の引数として受け取り、正の整数をリターンすること。関数の呼び出し時にはリサイズされるウィンドウが選択される。

以下の関数は一時的な表示にカレントバッファーを使用します:

`momentary-string-display` *string position* **&optional** *char message* [Function]
この関数はカレントバッファー内の *position*に *string*を瞬間表示 (momentarily display) する。これは undo リストやバッファーの変更状態 (modification status) に影響を与えない。

瞬間表示は次の入力イベントまで留まる。次の入力イベントが *char*なら `momentary-string-display`はそれを無視してリターンする。それ以外ならそのイベントは後続の入力として使用するためにバッファリングされる。つまり *char*とタイプすると表示からその文字列を単に削除して、(たとえば) *char*ではない `C-f`とタイプすると表示からその文字列を削除して、その後に (おそらく) ポイントを前方へ移動するだろう。引数 *char*のデフォルトはスペース。

`momentary-string-display`のリターン値に意味はない。

文字列 *string*がコントロール文字を含まなければ、`before-string`プロパティでオーバーレイを作成 (その後に削除) することで、同じことをより汎用的に行うことができる。Section 39.9.2 [Overlay Properties], page 1002 を参照のこと。

*message*が非 `nil`なら、バッファー内に *string*が表示されている間はエコーエリアにそれが表示される。`nil`の場合のデフォルトは、継続するためには *char*をタイプするように告げるメッセージ。

以下の例では最初はポイントは 2 行目の先頭に置かれている:

```
---------- Buffer: foo ----------
This is the contents of foo.
⋆Second line.
---------- Buffer: foo ----------

(momentary-string-display
  "**** Important Message! ****"
  (point) ?\r
  "Type RET when done reading")
⇒ t

---------- Buffer: foo ----------
This is the contents of foo.
**** Important Message! ****Second line.
---------- Buffer: foo ----------

---------- Echo Area ----------
Type RET when done reading
---------- Echo Area ----------
```

39.9 オーバーレイ

バッファーのテキストのスクリーン上の見栄えを変更するために、プレゼンテーション機能としてオーバーレイ *(overlay)* を使用できます。オーバーレイとは個々のバッファーに属するオブジェクトであり、指定された開始と終了をもっています。確認したりセットすることができるプロパティももっています。それらはオーバーレイをもつテキストの表示に影響を与えます。

オーバーレイの視覚的効果は、対応するテキストプロパティと同様です (Section 32.19 [Text Properties], page 813 を参照)。しかし実装が異なるために、オーバーレイは一般的にスケーラブルではありません (処理数に応じてバッファー内のオーバーレイ数に比例した時間を要する)。バッファー内の多数の部分の視覚的外観に効果を及ぼす必要がある場合にはテキストプロパティの使用を推奨します。

オーバーレイはその開始と終了を記録するためにマーカーを使用します。したがってバッファーのテキスト編集では、すべてのオーバーレイがそのテキストに留まるように開始と終了が調整されます。オーバーレイ作成時にはオーバーレイの先頭、または同様に終端にテキストが挿入された場合に、それがオーバーレイの内側 (または外側) になるべきなのかを指定できます。

39.9.1 オーバーレイの管理

このセクションではオーバーレイの作成、削除、移動、およびそれらのコンテンツを調べる関数を説明します。オーバーレイはバッファーのコンテンツの一部ではないので、その変更はバッファーの undo リストに記録されません。

`overlayp` *object* [Function]
この関数は *object*がオーバーレイなら `t`をリターンする。

`make-overlay` *start end* **&optional** *buffer front-advance rear-advance* [Function]
この関数は *buffer*に属する、*start*から *end*の範囲のオーバーレイを作成してリターンする。*start*と *end*はいずれもバッファーの位置を指定しなければならず、整数かマーカーを指定できる。*buffer*が省略されると、そのオーバーレイはカレントバッファーに作成される。

*start*と *end*が同一のバッファー位置を指定するオーバーレイは空 *(empty)* のオーバーレイとして知られる。*start*と *end*の間のテキストが削除されれば、非空のオーバーレイも空になり得る。これが発生したとき、デフォルトではオーバーレイは削除されないが、‘`evaporate`’プロパティを与えることにより削除されるようにできる (Section 39.9.2 [Overlay Properties], page 1002 を参照)。

引数 *front-advance*と *rear-advance*はそれぞれオーバーレイの開始と終了にたいするマーカーの挿入タイプを指定する。Section 31.5 [Marker Insertion Types], page 771 を参照のこと。どちらも `nil` (デフォルト) なら、そのオーバーレイは先頭に挿入された任意のテキストを含むように拡張されるが、終端に挿入されたテキストにたいしては拡張されない。*front-advance*が非 `nil`なら、オーバーレイの先頭に挿入されたテキストはオーバーレイから除外される。*rear-advance*が非 `nil`なら、オーバーレイの終端に挿入されたテキストはオーバーレイに含まれる。

`overlay-start` *overlay* [Function]
この関数は *overlay*が開始する位置を整数でリターンする。

`overlay-end` *overlay* [Function]
この関数は *overlay*が終了する位置を整数でリターンする。

`overlay-buffer` *overlay* [Function]
この関数は *overlay*が所属するバッファーをリターンする。*overlay*が削除されていたら `nil`をリターンする。

`delete-overlay` *overlay* [Function]
この関数は *overlay*を削除する。そのオーバーレイは Lisp オブジェクトとして存在し続けて、そのプロパティリストは変更されないがバッファーへの所属と表示にたいするすべての効果を失う。

削除済みオーバーレイが永続的に非接続という訳ではない。`move-overlay`を呼び出すことによりバッファー内の位置を与えることができる。

`move-overlay` *overlay start end* **&optional** *buffer* [Function]
この関数は *overlay*を *buffer*に移動して、その境界を *start*と *end*に配置する。*start*と *end*の引数はいずれもバッファーの位置を指定しなければならず、整数かマーカーを指定できる。

*buffer*が省略された場合、*overlay*はすでに関連付けられている同じバッファーに留まる。さらに *overlay*が削除されていたら、それをカレントバッファーに所属させる。

リターン値は *overlay*。

これはオーバーレイの両端位置を変更する唯一の有効な方法である。手作業でオーバーレイ内のマーカーの変更を試みてはならない。それにより他の重要なデータ構造の更新が失敗して、いくつかのオーバーレイが失われる可能性がある。

`remove-overlays` **&optional** *start end name value* [Function]
この関数はプロパティ *name*が値 *value*をもつような、*start*と *end*の間のすべてのオーバーレイを削除する。これによりオーバーレイの両端位置が変更されたり分割される可能がある。

*name*が省略か `nil`なら、それは指定されたリージョン内のすべてのオーバーレイを削除することを意味する。*start*および/または *end*が省略か `nil`なら、それぞれバッファーの先頭と終端を意味する。したがって `(remove-overlays)`はカレントバッファー内のすべてのオーバーレイを削除する。

`copy-overlay` *overlay* [Function]
この関数は *overlay*のコピーをリターンする。このコピーは *overlay*と同じ両端位置とプロパティをもつ。しかしオーバーレイの開始と終了にたいするマーカー挿入タイプはデフォルト値にセットされる (Section 31.5 [Marker Insertion Types], page 771 を参照)。

以下にいくつか例を示します:

```
;; オーバーレイの作成
(setq foo (make-overlay 1 10))
     ⇒ #<overlay from 1 to 10 in display-ja.texi>
(overlay-start foo)
     ⇒ 1
(overlay-end foo)
     ⇒ 10
(overlay-buffer foo)
     ⇒ #<buffer display-ja.texi>
;; 後でチェックできるようプロパティ付与
(overlay-put foo 'happy t)
     ⇒ t
;; プロパティが付与されたか検証
(overlay-get foo 'happy)
     ⇒ t
;; オーバーレイを移動
(move-overlay foo 5 20)
     ⇒ #<overlay from 5 to 20 in display-ja.texi>
(overlay-start foo)
     ⇒ 5
```

```
(overlay-end foo)
     ⇒ 20
;; オーバーレイを削除
(delete-overlay foo)
     ⇒ nil
;; 削除されたか検証
foo
     ⇒ #<overlay in no buffer>
;; 削除済みオーバーレイは位置をもたない
(overlay-start foo)
     ⇒ nil
(overlay-end foo)
     ⇒ nil
(overlay-buffer foo)
     ⇒ nil
;; オーバーレイの削除取り消し
(move-overlay foo 1 20)
     ⇒ #<overlay from 1 to 20 in display-ja.texi>
;; 結果の検証
(overlay-start foo)
     ⇒ 1
(overlay-end foo)
     ⇒ 20
(overlay-buffer foo)
     ⇒ #<buffer display-ja.texi>
;; オーバーレイの移動と削除では、オーバーレイのプロパティは変更されない
(overlay-get foo 'happy)
     ⇒ t
```

Emacs はそれぞれのバッファーのオーバーレイを任意の中心位置 (center position) で分割される 2 つのリストに格納します。一方のリストはバッファーの中心位置から後方に拡張されて、もう一方は中心位置から前方へと拡張されます。中心位置はバッファーの任意の位置をとることができます。

`overlay-recenter` *pos* [Function]

この関数はカレントバッファーのオーバーレイを位置 *pos*の周辺に再センタリングする。これにより位置 *pos*近傍のオーバーレイの照合は高速になるが、*pos*から離れた位置にたいしては低速になる。

バッファーを前方にスキャンしてオーバーレイを作成するループは、最初に `(overlay-recenter (point-max))`を行うことにより高速になる可能性があります。

39.9.2 オーバーレイのプロパティ

オーバーレイプロパティは文字が表示される方法をどちらのソースからも取得できるという点においてテキストプロパティと似ています。しかしほとんどの観点において両者は異なります。これらの比較は Section 32.19 [Text Properties], page 813 を参照してください。

テキストプロパティはそのテキストの一部として考えることができます。オーバーレイとそのプロパティは特にテキストの一部とはみなされません。したがってさまざまなバッファーや文字列の間でテキストをコピーすると、テキストプロパティは保持されますがオーバーレイを保持しようとは試み

ません。バッファーのテキストプロパティの変更はバッファーを変更済みとマークしますが、オーバーレイの移動やプロパティの変更は違います。テキストプロパティの変更とは異なり、オーバーレイプロパティの変更はバッファーの undo リストに記録されません。

複数のオーバーレイが同じ文字にたいしてプロパティ値を指定できるので、Emacs は各オーバーレイにたいして優先度の指定を促します。優先度の値はオーバーラップするオーバーレイのどちらが“勝つ” かを判断するために使用されます。

以下の関数はオーバーレイのプロパティの読み取りとセットを行います:

`overlay-get` *overlay prop* [Function]
この関数は *overlay*内に記録されたプロパティ *prop*の値をリターンする。そのプロパティにたいして *overlay*が何も値を記録していないが、シンボルであるような `category`プロパティをもつ場合には、そのシンボルの *prop*プロパティが使用される。それ以外なら値は `nil`。

`overlay-put` *overlay prop value* [Function]
この関数は *overlay*内に記録されたプロパティ *prop*の値に *value*をセットする。リターン値は *value*。

`overlay-properties` *overlay* [Function]
これは *overlay*のプロパティリストのコピーをリターンする。

与えられた文字にたいしてテキストプロパティとオーバーレイプロパティの両方をチェックする関数 `get-char-property`も参照してください。Section 32.19.1 [Examining Properties], page 813 を参照してください。

多くのオーバーレイプロパティには特別な意味があります。以下はそれらのテーブルです:

`priority` このプロパティの値はオーバーレイの優先度を決定する。優先度にたいして値を指定したければ `nil` (か 0)、または正の整数を使用すること。それ以外のすべての値にたいする動作は未定義。

2 つ以上のオーバーレイが同じ文字をカバーし、いずれもが同じプロパティを指定する場合には、優先度が重要になる。他より `priority`の値が大きいほうが他をオーバーライドする。(`face`プロパティにたいしては、より高い優先度のオーバーレイの値は他の値を完全にはオーバーライドしない。かわりに高い優先度の face 属性がより低い優先度の `face`プロパティの face 属性をオーバーライドする)。2 つのオーバーレイが同じ優先値をもち、一方がもう一方にネストされている場合には、内側のオーバーレイが外側のオーバーレイより優先される。いずれのオーバーレイも他方にネストされていなければ、どちらのオーバーレイが優先するかを仮定しないこと。

現在のところ、すべてのオーバーレイはテキストプロパティより優先される。

Emacs は内部的ないくつかのオーバーレイに非数値の優先値を使用するときがあるので、(自分が作成したものでなければ) オーバーレイの優先度にたいして数値演算を試みてはならない。特にリージョンの表示に使用されるオーバーレイは `(`*`primary . secondary`*`)` (*primary*は上述のように使用されて、*secondary*は *primary*とネストを考慮したオーバーレイ間の優先度の解決が失敗した際に使用される) という形式の優先値を使用する。しかしあなたにはこの実装の詳細にもとづいて Lisp プログラムをデザインしないようにアドバイスした。優先順にオーバーレイを配置する必要があるなら `overlays-at`の *sorted*引数を使用すること。Section 39.9.3 [Finding Overlays], page 1006 を参照のこと。

`window` `window`プロパティが非 `nil`ならオーバーレイはそのウィンドウだけに適用される。

`category` オーバーレイが `category`プロパティをもつなら、それをオーバーレイのカテゴリー *(category)* と呼ぶ。これはシンボルであること。そのシンボルのプロパティはオーバーレイのプロパティにたいしてデフォルトの役割を果たす。

`face` このプロパティはテキストの外観を制御する (Section 39.12 [Faces], page 1011 を参照)。プロパティの値は以下のいずれか:

- フェイス名 (シンボルか文字列)。
- anonymous フェイス: `(`*`keyword value ...`*`)`という形式のプロパティリストであり *keyword*はフェイス属性名、*value*はその属性の値。
- フェイスのリスト。リストの要素はそれぞれフェイス名か anonymous フェイスのいずれかであること。これはリストされた各フェイスの属性を集約するフェイスを指定する。このリスト内で先に出現するフェイスが、より高い優先度をもつ。
- `(foreground-color . `*`color-name`*`)`か `(background-color . `*`color-name`*`)`という形式のコンスセル。これは `(:foreground `*`color-name`*`)`や `(:background `*`color-name`*`)`と同じように、フォアグラウンドとバックグラウンドのカラーを指定する。この形式は後方互換性のためだけにサポートされており、使用は避けること。

`mouse-face`
このプロパティはマウスがオーバーレイ範囲内にあるときに、`face`のかわりに使用される。しかし Emacs はこのプロパティに由来するテキストのサイズを変更するようなフェイス属性 (`:height`、`:weight`、`:slant`) をすべて無視する。これらの属性はハイライトされていないテキストでは常に同一である。

`display` このプロパティはテキストが表示される方法を変更するさまざまな機能をアクティブにする。たとえばこれはテキストの外観を縦長 (taller) や横長 (shorter) にしたり、高く (higher) したり低く (lower) したり、イメージによる置き換えを行う。Section 39.16 [Display Property], page 1043 を参照のこと。

`help-echo`
あるオーバーレイが`help-echo`プロパティをもつなら、そのオーバーレイ内のテキスト上にマウスを移動した際に、Emacs はエコーエリアかツールチップウィンドウにヘルプ文字列を表示する。詳細は [Text help-echo], page 821 を参照のこと。

`field` 同じ `field`プロパティをもつ連続する文字はフィールド *(field)* を形成する。`forward-word`や `beginning-of-line`を含むいくつかの移動関数はフィールド境界で移動を停止する。Section 32.19.9 [Fields], page 831 を参照のこと。

`modification-hooks`
このプロパティの値はオーバーレイ内の任意の文字の変更、またはオーバーレイの厳密に内側にテキストが挿入された場合に呼び出される関数のリスト。

このフックの関数は各変更の前後両方で呼び出される。これらの関数が受け取った情報を保存して呼び出し間で記録を比較すれば、バッファー内のテキストでどのような変更が行われたかを正確に判断できる。

変更前に呼び出された際にはオーバーレイ、`nil`、変更されたテキスト範囲の開始と終了という 4 つの引数を各関数は受け取る。

変更後に呼び出された際にはオーバーレイ、`t`、変更されたテキスト範囲の開始と終了、およびその範囲により置き換えられた変更前のテキスト長という 5 つの引数を各関数は

受け取る (変更前の長さは挿入では 0、削除では削除された文字数であり、変更後の先頭と終端が等しくなる)。

これらの関数が呼び出される際には `inhibit-modification-hooks`が、非 `nil`にバインドされる。関数がバッファーを変更した場合には変更にたいして変更フックが実行されるように、`inhibit-modification-hooks`を `nil`にバインドしたいと思うかもしれない。しかしこれを行うことにより、あなた自身の変更フックが再帰的に呼び出されるかもしれないので、それに確実に備える必要がある。Section 32.32 [Change Hooks], page 850 を参照のこと。

テキストプロパティも `modification-hooks`プロパティをサポートするが詳細は幾分異なる (Section 32.19.4 [Special Properties], page 820 を参照)。

`insert-in-front-hooks`
: このプロパティの値はオーバーレイ先頭へのテキスト挿入前後に呼び出される関数のリスト。呼び出し方は `modification-hooks`の関数と同様。

`insert-behind-hooks`
: このプロパティの値はオーバーレイ終端へのテキスト挿入前後に呼び出される関数のリスト。呼び出し方は `modification-hooks`の関数と同様。

`invisible`
: `invisible`プロパティによりオーバーレイ内のテキストを不可視にできる。これはそのテキストがスクリーン上に表示されないことを意味する。詳細は Section 39.6 [Invisible Text], page 992 を下さいのこと。

`intangible`
: オーバーレイの `intangible`プロパティは正に `intangible`テキストプロパティと同様に機能する。これは時代遅れである。詳細は See Section 32.19.4 [Special Properties], page 820 を参照のこと。

`isearch-open-invisible`
: このプロパティはインクリメンタル検索にたいして最後のマッチがそのオーバーレイに重なる場合に、不可視なオーバーレイを永続的に可視にする方法を告げる。Section 39.6 [Invisible Text], page 992 を参照のこと。

`isearch-open-invisible-temporary`
: このプロパティはインクリメンタル検索にたいして、検索の間に不可視なオーバーレイを一時的に可視にする方法を告げる。Section 39.6 [Invisible Text], page 992 を参照のこと。

`before-string`
: このプロパティの値はオーバーレイ先頭に表示するために追加する文字列。この文字列はいかなる意味においてもバッファー内には出現せずにスクリーン上にのみ表れる。

`after-string`
: このプロパティの値はオーバーレイ終端に表示するために追加する文字列。この文字列はいかなる意味においてもバッファー内には出現せずにスクリーン上にのみ表れる。

`line-prefix`
: このプロパティは表示時にそれぞれの非継続行の後に追加するディスプレイ仕様 (display spec) を指定する。Section 39.3 [Truncation], page 982 を参照のこと。

`wrap-prefix`
: このプロパティは表示時にそれぞれの継続行の前に追加するディスプレイ仕様 (display spec) を指定する。Section 39.3 [Truncation], page 982 を参照のこと。

`evaporate`
: このプロパティが非`nil`の場合には、そのオーバーレイが空 (長さが 0) になったら自動的に削除される。空のオーバーレイ (Section 39.9.1 [Managing Overlays], page 1000 を参照) にたいして非`nil`の`evaporate`プロパティを与えた場合には即座に削除される。オーバーレイがこのプロパティをもたなければ、バッファーからオーバーレイの開始位置と終了位置の間のテキストが削除された際に削除されないことに注意。

`keymap`
: このプロパティが`nil`なら、そのテキスト範囲にたいしてキーマップを指定する。このキーマップはポイントの後の文字がそのオーバーレイ内にあるときに使用されて、他のほとんどのキーマップより優先される。Section 22.7 [Active Keymaps], page 429 を参照のこと。

`local-map`
: `local-map`プロパティは`keymap`プロパティと同様だが、既存のキーマップに付け加えるのではなくバッファーのローカルマップを置き換える点が異なる。これはそのキーマップがマイナーモードキーマップより低い優先度をもつことも意味する。

`keymap`と`local-map`プロパティは`before-string`、`after-string`、`display`プロパティにより表示された文字列には影響しません。これはポイントがその文字列上にない場合のマウスクリックや、その文字列に関する他のマウスイベントにのみ関係があります。その文字列に特別なマウスイベントをバインドするには、そのイベントを`keymap`か`local-map`プロパティに割り当てます。Section 32.19.4 [Special Properties], page 820 を参照してください。

39.9.3 オーバーレイにたいする検索

`overlays-at` *pos* **&optional** *sorted* [Function]
: この関数はカレントバッファー内の位置 *pos*にある文字をカバーするすべてオーバーレイのリストをリターンする。*sorted*が非`nil`ならリストは優先度降順、それ以外なら特定の順にはソートされない。オーバーレイが *pos*、またはそれより前から始まり、かつ *pos*の後で終わるなら位置 *pos*はオーバーレイに含まれる。

以下はポイント位置の文字にたいしてプロパティ *prop*を指定するオーバーレイのリストをリターンする Lisp 関数の使用例:

```
(defun find-overlays-specifying (prop)
  (let ((overlays (overlays-at (point)))
        found)
    (while overlays
      (let ((overlay (car overlays)))
        (if (overlay-get overlay prop)
            (setq found (cons overlay found))))
      (setq overlays (cdr overlays)))
    found))
```

`overlays-in` *beg end* [Function]
: この関数は *beg*から *end*のリージョンと重複 (overlap) するオーバーレイのリストをリターンする。オーバーレイがリージョン内の文字を少なくとも 1 つ含めば、オーバーレイとリージョンは重複している。空のオーバーレイ (Section 39.9.1 [Managing Overlays], page 1000 を参照) が *beg*にある場合、厳密には *beg*と *beg*の間、または *end*がバッファー終端の位置を意味するときは *end*にある場合には重複している。

`next-overlay-change` *pos* [Function]
この関数は *pos*の後にあるオーバーレイの開始か終了となるバッファー位置をリターンする。それが存在しなければ `(point-max)`をリターンする。

`previous-overlay-change` *pos* [Function]
この関数は *pos*の前にあるオーバーレイの開始か終了となるバッファー位置をリターンする。それが存在しなければ `(point-min)`をリターンする。

以下に例としてプリミティブ関数 `next-single-char-property-change` (Section 32.19.3 [Property Search], page 817 を参照) の単純化 (かつ非効率的) したバージョンを示します。これは位置 *pos*から前方へ与えられたプロパティ`prop`にたいして、オーバーレイプロパティまたはテキストプロパティのいずれかの値が変化した次の位置を検索します。

```
(defun next-single-char-property-change (position prop)
  (save-excursion
    (goto-char position)
    (let ((propval (get-char-property (point) prop)))
      (while (and (not (eobp))
                  (eq (get-char-property (point) prop) propval))
        (goto-char (min (next-overlay-change (point))
                        (next-single-property-change (point) prop)))))
    (point)))
```

39.10 表示されるテキストのサイズ

すべての文字が同じ幅をもつ訳ではありませんが、以下の関数により文字の幅をチェックできます。関連する関数については Section 32.17.1 [Primitive Indent], page 807 と Section 30.2.5 [Screen Lines], page 760 を参照してください。

`char-width` *char* [Function]
この関数は文字 *char*がカレントバッファーに表示された場合 (つまりそのバッファーのディスプレイテーブルがあれば考慮に入れる。Section 39.22.2 [Display Tables], page 1079 を参照) の幅を列数でリターンする。タブ文字の幅、通常は `tab-width` (Section 39.22.1 [Usual Display], page 1078 を参照)。

`string-width` *string* [Function]
この関数は文字列 *string*がカレントバッファーおよび選択されたウィンドウに表示された場合の幅を列数でリターンする。

`truncate-string-to-width` *string width* **&optional** *start-column padding ellipsis* [Function]
この関数は *string*の一部を列数 *width*にフィット新たな文字列としてリターンする。

*string*が *width*に満たなければ文字列終端が結果の終端となる。*string*内の 1 つの複数列文字が列 *width*を超えて跨がるようなら、その文字は結果に含まれない。つまり結果は *width*より短くなるかもしれないが超えることはできない。

オプション引数 *start-column*は開始列を指定する。これが非 `nil`なら、その文字列の最初の *start-column*列は値から省かれる。*string*内の 1 つの複数列文字が列 *start-column*を超えて跨がるようなら、その文字は結果に含まれない。

オプション引数 *padding*が非 `nil`なら、結果となる文字列の幅を正確に *width*列に拡張するためにパディング文字が追加される。結果文字列が *width*より短ければ結果文字列の終端にパ

ディング文字が使用される。*string*内の 1 つの複数列文字が列 *start-column*を跨ぐ場合は先頭にもパディング文字が使用される。

*ellipsis*が非 `nil`なら、それは *string*の表示幅が *ellipsis*の表示幅以下でなければ *width*を超えてしまう場合に、*string*の終端 (任意のパディングを含む) を置き換える文字列であること。*ellipsis*が非 `nil`かつ文字列以外なら、それは変数 `truncate-string-ellipsis`の値を意味する。

```
(truncate-string-to-width "\tab\t" 12 4)
     ⇒ "ab"
(truncate-string-to-width "\tab\t" 12 4 ?\s)
     ⇒ "    ab  "
```

以下の関数は与えられたウィンドウにあるテキストを表示したときのサイズをピクセル単位でリターンします。この関数はテキストを含むためにウィンドウを十分大きくするために `fit-window-to-buffer`と `fit-frame-to-buffer` (Section 28.4 [Resizing Windows], page 617 を参照) により使用されます。

`window-text-pixel-size` **&optional** *window from to x-limit y-limit mode-and-header-line* [Function]

この関数は *window*のバッファーのテキストサイズをピクセル単位でリターンする。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。リターン値は任意のテキスト行の最大ピクセル幅と、すべてのテキスト行の最大ピクセル高さのコンス。この関数はバッファーテキストの表示に要する *window*サイズを、Lisp プログラムが調整できるようにするために存在する。

オプション引数 *from*が非 `nil`なら、それは考慮すべき最初のテキスト位置を指定する。デフォルトはそのバッファーのアクセス可能な最小の位置。*from*が `t`なら、改行文字ではないアクセス可能な最小位置を意味する。オプション引数 *to*が非 `nil`なら、それは考慮すべき最後のテキスト位置を指定する。デフォルトはそのバッファーのアクセス可能な最大の位置。*to*が `t`なら、改行文字ではないアクセス可能な最大位置を意味する。

オプション引数 *x-limit*が非 `nil`なら、その位置を超えるテキストを指定するような最大 X 座標を指定する。したがってこれはこの関数がリターンし得る最大のピクセル幅でもある。*x-limit*が `nil`または省略なら、*window*の body(Section 28.3 [Window Sizes], page 613 を参照) のピクセル幅を使用することを意味する。このデフォルト値はウィンドウより永井切り詰められた行のテキストは無視されることを意味する。このデフォルト値はは呼び出し側が *window*の幅の変更を意図しない場合に有用。それ以外なら呼び出し側はここで想定される *window*の body の最大幅を指定すること。特に行の切り詰めが予想される場合に、それらの行のテキストを勘定に入れる必要があるなら、*x-limit*を大きな値にセットする必要がある。長い行の幅の計算にはいくらかの時間を要するかもしれないので、必要に応じてこの変数を小さくするのはよいアイデアである。これはいずれにせよ切り詰められるような長い行をバッファーが含む場合が特に該当する。

オプション引数 *y-limit*が非 `nil`なら、その値を超えるテキストは無視されるような最大 Y 座標を指定する。したがってこれは関数がリターンし得る最大のピクセル高さでもある。*y-limit*が nil か省略なら、*to*で指定したバッファー位置までのすべてのテキスト行を考慮することを意味する。大きなバッファーのピクセル高さの計算には多くの時間を要する可能性があるので、特に呼び出し側がバッファーのサイズを知らない場合におけるこの変数の指定は合理的である。

オプション引数 *mode-and-header-line*が `nil`または省略された場合には、リターン値に *window*のモードラインとヘッダーラインの高さを含めないことを意味する。これがシンボル

`mode-line`か`header-line`のいずれかなら、それらが存在する場合にはリターン値にそのラインの高さだけを含める。これが`t`なら存在する場合は両方の高さをリターン値に含める。

`window-text-pixel-size`はウィンドウ内に表示されているテキスト全体を扱い、個々の行サイズには留意しません。それは以下の関数が行います。

`window-lines-pixel-dimensions` **&optional** *window first last body inverse left* [Function]

この関数は指定した *window*に表示された各行のピクセルサイズを計算する。これは *window* のカレントグリフマトリクス (*window*にカレントで表示されている各バッファー文字のグリフを格納するマトリクス。Section 39.22.4 [Glyphs], page 1081 を参照) を調べることにより機能する。成功したら各行末文字の右下隅の X 座標と Y 座標を表すコンスペアのリストをリターンする。これらの座標は *window*の左上隅にある原点 (0, 0) からピクセル単位で計測される。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。

オプション引数 *first*が整数なら、リターンする *window*のグリフマトリクスの最初の行のインデックス (0 から開始) を示す。*window*にヘッダーラインがあればインデックス 0 の行はヘッダーラインになることに注意。*first*が`nil`なら考慮する最初の行はオプション引数 *body*の値で判断される。*body*が非`nil`なら、(もしあれば) ヘッダーラインをすべてスキップして *window* の body の最初の行から開始することを意味する。それ以外なら *window*のグリフマトリクスの最初の行 (ヘッダーラインかもしれない) から開始することを意味する。

オプション引数 *last*が整数なら、リターンする *window*のグリフマトリクスの最後の行のインデックスを示す。*last*が`nil`なら考慮する最初の行はオプション引数 *body*の値で判断される。*body*が非`nil`なら、*window*のモードラインを省略して *window*の body の最後の行を使用することを意味する。それ以外なら *window*のグリフマトリクスの最後の行 (モードラインかもしれない) を使用することを意味する。

オプション引数 *inverse*が`nil`なら、リターンされる任意の行にたいする Y ピクセル値が *window*の左エッジ (*body*が非`nil`なら左 body エッジ) から、その行の最後のグリフの右エッジまでのピクセル単位の距離を指定することを意味する。非`nil`の *inverse*は、リターンされる任意の行にたいする Y ピクセル値がその行の最後のグリフの右エッジから右エッジ (*body*が非`nil`なら右 body エッジ) までのピクセル単位の距離を指定することを意味する。これは各行末の未使用スペースの量を判断するために有用。

オプション引数 *left*が非`nil`なら各行左端文字の左下隅の X 座標と Y 座標をリターンすることを意味する。これは主に右から左にテキストを表示にたいして使用されるべき値である。

*left*が非`nil`で *inverse*が`nil`なら、リターンされる任意の行にたいする Y ピクセル値がその行の最後 (左端) のグリフの左エッジから、*window*の右エッジ (*body*が非`nil`なら右 body エッジ) までのピクセル単位の距離を指定することを意味する。*left*と *inverse*がいずれも非`nil`なら、リターンされる任意の行にたいする Y ピクセル値が *window*の左エッジ (*body*が非`nil`なら左 body エッジ) から、その行の最後 (左端) のグリフの左エッジまでのピクセル単位の距離を指定することを意味する。

この関数は *window*のカレントグリフマトリクスが最新でなければ`nil`をリターンする。これはたとえばコマンドの処理中のように Emacs が busy な際に通常は発生する。これは遅延が 0 秒であるようなアイドルタイマーからこの関数が実行された際に取得され得る値である。

`line-pixel-height` [Function]

この関数は選択されたウィンドウのポイント位置にある行の高さをピクセル単位でリターンする。値にはその行の行スペーシングが含まれる (Section 39.11 [Line Height], page 1010 を参照)。

行番号 (Section “Display Custom” in *The GNU Emacs Manual* を参照) とともにバッファーを表示している際には、行番号の表示に必要な幅が解ると便利なときがあります。以下はレイアウト計算用にこの情報を必要とする Lisp プログラムのための関数です。

`line-number-display-width` **&optional** *pixelwise* [Function]
この関数は選択されたウィンドウで行番号の表示に使用される幅をリターンする。オプション引数 *pixelwise*がシンボル `columns`なら、リターン値はフレームの正準列 (清純文字幅) にたいする浮動小数点数となる。*pixelwise*が `t`やそれ以外の非 `nil`値なら、値はピクセルで計測した整数となる。*pixelwise*が省略か `nil`なら、値は `line-number`フェイスにたいして定義されたフォントによる列数を表す整数となる。この場合には値には番号の表示の間隙を埋めるために使用する 2 列分は含まれない。選択されたウィンドウに行番号が表示されていなければ、*pixelwise*の値にかかわらず値は 0 になる。別のウィンドウにたいしてこの情報が必要なら `with-selected-window`を使用すること (Section 28.9 [Selecting Windows], page 631 を参照)。

39.11 行の高さ

各ディスプレイ行のトータル高さは、その行のコンテンツ高さにディスプレイ上部や下部にオプションで追加される垂直行スペーシングを加えて構成されます。

行のコンテンツ高さは、もしあれば最後の改行を含む、そのディスプレイ行の文字またはイメージの最大高さです (継続されるディスプレイ行には最後の改行が含まれない)。特にこれより大きい高さを指定しなければ、これがデフォルトの行高さになります (これは一般的には対応するフレームのデフォルトのフォント高さに等しい。Section 29.3.2 [Frame Font], page 707 を参照)。

より大きい行高さを明示的に指定するためにはディスプレイ行の絶対高さ、または垂直スペースを指定する複数の方法が存在します。しかし何を指定したかに関わらず、実際の行高さがデフォルトの高さより小さくなることはありません。

改行はその改行で終わるディスプレイ行のトータル高さを制御するテキストプロパティとオーバーレイプロパティ`line-height`をもつことができます。

プロパティの値が `t`なら改行文字はその行の表示高さにたいして効果をもたず、可視なコンテンツだけが高さを決定します。この場合には以下で説明する `line-spacing`プロパティも無視されます。これはイメージ間に追加のブランク領域をもたない、小さなイメージ (やイメージスライス) にたいして有用です。

プロパティの値が `(`*`height total`*`)`という形式のリストなら、これはディスプレイ行の下部に余分なスペースを追加します。最初に Emacs は、その行の上部の余分なスペースを制御するための高さ spec として、*height*を使用します。それから行のトータル高さを *total*にするために、行の下部に必要なスペースを追加します。この場合には、改行にたいする `line-spacing`プロパティのすべての値は無視されます。

他の種類のプロパティ値は高さ spec(height spec) です。これは行の高さを指定する数値に変換されます。高さ spec を記述するためには複数の方法があります。以下はそれらが数値に変換される方法です:

`integer` 高さ spec が正の整数なら高さの値はその整数。

`float` 高さ spec が浮動小数点数 *float*なら高さ数値はそのフレームのデフォルト行高さの *float* 倍。

`(face . ratio)`
: 高さ spec がこのフォーマットのコンスなら、高さ数値はフェイス *face*の高さの *ratio* 倍。*ratio*には任意の型の数値を指定でき、`nil`は 1 の ratio を意味する。*face*が `t`ならカレントフェイスを参照する。

`(nil . ratio)`
: 高さ spec がこのフォーマットのコンスなら高さ数値はその行のコンテンツ高さの *ratio*倍。

したがって任意の有効な種々の高さ spec によりピクセル単位で高さが決定されます。行のコンテンツ高さがこれより小さければ、Emacs は指定されたトータル高さになるように余分な垂直スペースを行の上部に追加します。

`line-height`プロパティを指定しない場合には、その行の高さは行のコンテンツ高さとに行スペーシングを追加して構成されます。Emacs の異なるさまざまな部分のテキストにたいして、行スペーシングを指定する複数の方法が存在します。

グラフィカルなディスプレイではフレームパラメーター`line-spacing` (Section 29.4.3.4 [Layout Parameters], page 719 を参照) を使用することにより、フレーム内のすべての行にたいして行スペーシングを指定できます。しかし `line-spacing`のデフォルト値が非 `nil`なら、それはそのフレームのフレームパラメーター `line-spacing`をオーバーライドします。整数は行の下部に配するピクセル数を指定します。浮動小数点数はフレームのデフォルト行高さに相対的にスペーシングを指定します。

バッファーローカル変数 `line-spacing`を通じて、バッファー内のすべての行の行スペーシングを指定できます。整数は行の下部に配するピクセル数を指定します。浮動小数点数はデフォルトフレーム行高さに相対的にスペーシングを指定します。これはそのフレームにたいして指定された行スペーシングをオーバーライドします。

最後に改行は改行で終わるディスプレイ行にたいしてデフォルトフレーム行スペーシングを広くできるテキストプロパティとオーバーレイプロパティ`line-spacing`、および変数 `line-spacing`をもつことができます。その値がバッファーやフレームのデフォルトより大きければ、その改行で終端されるディスプレイ行にはかわりにその値が使用されます。

種々の方法によりこれらのメカニズムは各行のスペーシングにたいする Lisp 値を指定します。値は高さ spec で、これは上述した Lisp 値に変換されます。しかしこの場合には高さ数値は行高さではなく行スペーシングを指定します。

テキスト端末では行スペーシングは変更できません。

39.12 フェイス

フェイス *(face)* とはフォント、フォアグラウンドカラー、バックグラウンドカラー、オプションのアンダーライン等のテキストを表示するためのグラフィカルな属性のコレクションのことです。フェイスは Emacs がバッファー内や、同様にモードラインのようなフレームの他の部分でテキストを表示する方法を制御します。

フェイスを表現する 1 つの方法として `(:foreground "red" :weight bold)`のような属性のプロパティリストがあります。このようなリストは *anonymous* フェイス *(anonymous face)* と呼ばれます。たとえば `face`テキストプロパティとして anonymous フェイスを割り当てることができ、Emacs は指定された属性でテキストを表示するでしょう。Section 32.19.4 [Special Properties], page 820 を参照してください。

より一般的にはフェイスはフェイス名 *(face name)* を通じて参照されます。これはフェイス属性のセットに関連付けられた Lisp シンボル [1] です。名前つきフェイスは `defface`マクロを使用して定

[1] 後方互換のため、フェイス名の指定に文字列も使用できます。これは同名の Lisp シンボルと等価です。

義できます (Section 39.12.2 [Defining Faces], page 1015 を参照)。Emacs にはいくつかの標準名前つきフェイスが同梱されています (Section 39.12.8 [Basic Faces], page 1024 を参照)。

Emacs のある部分では名前つきフェイスが要求されます (たとえば Section 39.12.3 [Attribute Functions], page 1018 に記す関数)。特に明記しないかぎり、名前つきフェイスの参照だけに用語フェイスを使用することにします。

`facep` *object* [Function]
この関数は *object*が名前つきフェイス (フェイス名の役目をもつ Lisp シンボルか文字列) なら非 `nil`、それ以外なら `nil`をリターンする。

39.12.1 フェイスの属性

フェイス属性 *(Face attributes)* は、フェイスの視覚的外観を決定します。以下はすべてのフェイス属性と、それらの可能な値と効果に関するテーブルです。

以下の値とは別に各フェイス属性は値 `unspecified`をもつことができます。この特殊な値はフェイスがその属性を直接指定しないことを意味します。`unspecified`属性は Emacs にかわりに親フェイス (以下の`:inherit`属性の記述を参照) を参照して、それに失敗したら基礎フェイス (Section 39.12.4 [Displaying Faces], page 1021 を参照) を参照することを指示します。`default`フェイスはすべての属性を指定しなければなりません。

これらの属性のいくつかは特定の種類のディスプレイにおいてのみ意味があります。ディスプレイが特定の属性を処理できなければ、その属性は無視されます。

`:family` フォントファミリー名 (文字列)。フォントファミリーに関する詳細は Section “Fonts” in *The GNU Emacs Manual* を参照のこと。関数 `font-family-list` (以下参照) は利用可能なファミリー名のリストをリターンする。

`:foundry` `:family`属性により指定されるフォントファミリーにたいするフォント *foundry* (文字列)。Section “Fonts” in *The GNU Emacs Manual* を参照のこと。

`:width` 相対的な文字幅。これはシンボル `ultra-condensed`、`extra-condensed`、`condensed`、`semi-condensed`、`normal`、`semi-expanded`、`expanded`、`extra-expanded`、`ultra-expanded`のいずれかであること。

`:height` フォントの高さ。もっともシンプルなケースでは 1/10 ポイントを単位とする整数。

値には基礎フェイス *(underlying face)* にたいして相対的に高さを指定する浮動小数点数、または関数も指定できる (Section 39.12.4 [Displaying Faces], page 1021 を参照)。浮動小数点数は基礎フェイスの高さをスケーリングする量を指定する。関数値は基礎フェイスの高さを単一の引数として呼び出されて、新たなフェイスの高さをリターンする。関数が整数を引数として渡された場合には整数をリターンしなければならない。

デフォルトフェイスの高さは整数を使用して指定しなければならない。浮動小数点数や関数は受け入れられない。

`:weight` フォントの weight。シンボル `ultra-bold`、`extra-bold`、`bold`、`semi-bold`、`normal`、`semi-light`、`light`、`extra-light`、`ultra-light` (太字から細字順) のいずれか。可変輝度テキストをサポートするテキスト端末では、normal より大な weight はより高輝度、小な weight はより低輝度で表示される。

`:slant` フォントの slant。シンボル `italic`、`oblique`、`normal`、`reverse-italic`、`reverse-oblique`のいずれか。可変輝度テキストをサポートするテキスト端末では slant されたテキストは half-bright で表示される。

`:foreground`
: フォアグラウンドカラー (文字列)。値にはシステム定義済みカラー、または 16 進カラー仕様を指定できる。Section 29.22 [Color Names], page 748 を参照のこと。白黒ディスプレイでは特定のグレー色調が点描パターンで実装されている。

`:distant-foreground`
: 代替えのフォアグラウンドカラー (文字列)。これは`:foreground`と似ているが、使用されるであろうフォアグラウンドカラーがバックグラウンドカラーに近いときのみフォアグラウンドカラーとして使用される点が異なる。これはたとえばテキストをマーク時 (リージョンフェイス) に有用。そのテキストがリージョンフェイスとして可視なフォアグラウンドをもつ場合には、そのフォアグラウンドが使用される。フォアグラウンドがリージョンフェイスのバックグラウンドに近ければ、テキストを可読にするために`:distant-foreground`が使用される。

`:background`
: バックグラウンドカラー (文字列)。値にはシステム定義済みカラー、または 16 進カラー仕様を指定できる。Section 29.22 [Color Names], page 748 を参照のこと。

`:underline`
: 文字にアンダーラインを引くべきか否か、およびその方法。`:underline`属性として可能な値は以下のとおり:

  `nil`
  : アンダーラインを引かない。

  `t`
  : そのフェイスのフォアグラウンドカラーでアンダーラインを引く。

  color
  : 文字列 *color*で指定されたカラーでアンダーラインを引く。

  (`:color` *color* `:style` *style*)
  : *color*は文字列、またはそのフェイスのフォアグラウンドカラーを意味するシンボル`foreground-color`。属性`:color`の省略はフェイスのフォアグラウンドカラーの使用を意味する。*style*は直線を意味する`line`、または波線を意味する`wave`いずれかのシンボルであること。属性`:style`の省略は直線を意味する。

`:overline`
: 文字にオーバーラインを引くべきか否か、およびそのカラー。値が`t`ならフェイスのフォアグラウンドカラーを使用してオーバーラインを引く。値が文字列ならそのカラーを使用してオーバーラインを引く。値`nil`はオーバーラインを引かないことを意味する。

`:strike-through`
: 文字に取り消し線を引くべきか否か、およびそのカラー。値は`:overline`で使用される値と同じ。

`:box`
: 文字周囲に枠 (box) を描画するか否か、そのカラー、枠線の幅、および 3D 外観。以下は`:box`の可能な値と意味:

  `nil`
  : 枠を描画しない。

  `t`
  : 幅 1 のフォアグラウンドカラーで枠線を描画する。

  color
  : 幅 1 のカラー *color*で枠線を描画する。

  (`:line-width` *width* `:color` *color* `:style` *style*)
  : この方法では枠のすべての形相を明示的に指定できる。値 *width*は描画する線の幅を指定する。デフォルトは 1。負の幅 $-n$は、上端と下端が基礎テ

キストのスペースを占有する線幅 *n*を意味しており、文字の高さの増加を避けることができる。

値 *color*は描画するカラーを指定する。シンプルな枠線ではフェイスのフォアグラウンドカラー、3D 枠線ではフェイスのバックグラウンドカラーがデフォルト。

値 *style*は 3D 枠線を描画するか否かを指定する。`released-button`なら押下された 3D ボタンのような外観、`pressed-button`なら押下されていない 3D ボタンのような外観、`nil`または省略された場合には 2D 枠線が使用される。

`:inverse-video`
文字が反転表示されて表示されるべきか否か。値は `t` (反転表示する) か `nil` (反転表示しない) のいずれか。

`:stipple` バックグラウンドの点描 (ビットマップ)。

値には文字列を指定できる。外部形式 X ビットマップデータを含むファイルの名前であること。ファイルは変数 `x-bitmap-file-path`にリストされるディレクトリー内で検索される。

かわりに `(`*`width height data`*`)`という形式のリストによりビットマップで直接値を指定できる。ここで *width*と *height*はピクセル単位によるサイズ、*data*は行単位でビットマップの raw ビットを含む文字列。各行は文字列内で連続する $(width + 7)/8$ バイトを占める (最善の結果を得るためにはユニバイト文字列であること)。これは各行が常に少なくとも 1 バイト全体を占めることを意味する。

値が `nil`なら点描パターンを使用しないことを意味する。

これは特定のグレー色調を処理するために自動的に使用されるので、通常は stipple 属性のセットは必要ない。

`:font` そのフェイスの表示に使用されるフォント。値はフォントオブジェクトかフォントセットであること。フォントオブジェクト、フォントスペース、フォントエンティティーに関する情報は Section 39.12.12 [Low-Level Font], page 1029、フォントセットに関する情報は Section 39.12.11 [Fontsets], page 1027 を参照のこと。

`set-face-attribute`や `set-face-font` (Section 39.12.3 [Attribute Functions], page 1018 を参照) を使用してこの属性を指定する際にはフォント spec、フォントエンティティー、または文字列を与えることもできる。Emacs はそのような値を適切なフォントオブジェクトに変換して、実際の属性値としてそのフォントオブジェクトを格納する。文字列を指定する場合には、その文字列のコンテンツはフォント名であること (Section “Fonts” in *The GNU Emacs Manual* を参照)。フォント名がワイルドカードを含む XLFD なら、Emacs はそれらのワイルドカードに最初にマッチするフォントを選択する。この属性の指定により`:family`、`:foundry`、`:width`、`:height`、`:weight`、`:slant`の属性値も変更される。

`:inherit` 属性を継承するフェイス名、またはフェイス名のリスト。継承フェイス由来の属性は基礎フェイスより高い優先度で、基礎フェイスの場合と同じような方法でマージされる (Section 39.12.4 [Displaying Faces], page 1021 を参照)。継承元のフェイスが `unspecified`なら Emacs は`:inherit`属性を決してマージしないので `nil`と同様に扱われる。フェイスのリストが使用された場合には、リスト内先頭側フェイスの属性が末尾側フェイスの属性をオーバーライドする。

`:extend` そのフェイスが行末を超えて拡張されるか、および行末とウィンドウのエッジの間の空スペースの表示に影響を与えるかどうか。値は行末とウィンドウのエッジの間の空スペースの表示にそのフェイスを使用するらら `t`、使用しなければ `nil`。Emacs が行末を超える空スペースの表示でいくつかのフォントをマージする際に、`:extend`が非 `nil`のフェイスだけがマージされる。デフォルトではこの属性がセットされているのは少数のフェイス、特に `region`だけである。この属性はテーマがフェイスに明示的な値を指定しない際に、値が `defface`によるフェイスの元定義から継承される点において他の属性とは異なる (Section 39.12.2 [Defining Faces], page 1015 を参照)。

`font-family-list` **&optional** *frame* [Function]
この関数は利用可能なフォントファミリー名のリストをリターンする。オプション引数 *frame* はそのテキストが表示されるフレームを指定する。これが `nil`なら選択されたフレームが使用される。

`underline-minimum-offset` [User Option]
この変数はアンダーラインが引かれたテキスト表示時に、ベースラインとアンダーライン間の最小距離をピクセル単位で指定する。

`x-bitmap-file-path` [User Option]
この変数は`:stipple`属性のビットマップファイルを検索するディレクトリーのリストを指定する。

`bitmap-spec-p` *object* [Function]
これは *object*、`:stipple` (上記参照) での使用に適した有効なビットマップ仕様なら `t`、それ以外なら `nil`をリターンする。

39.12.2 フェイスの定義

フェイスを定義する通常の方法は `defface`マクロを通じて定義する方法です。このマクロはフェイス名 (シンボル) をデフォルトのフェイス *spec(face spec)* と関連付けます。フェイス spec とは任意の与えられた端末上でフェイスがどの属性をもつべきかを指定する構文です。たとえばあるフェイス spec は高カラー端末ではあるフォアグラウンドカラーし、低カラー端末では異なるフォアグラウンドカラーを指定するかもしれません。

値がフェイス名であるような変数を作りたがる人がいます。ほとんどの場合には、これは必要ありません。通常の手順は `defface`でフェイスを定義して、その名前を直接使用することです。

(通常は `defface`により) 一度フェイスを定義したら、Emacs を再起動する以外にそのフェイスを安全に未定義にすることはできません。

`defface` *face spec doc* [*keyword value*]. . . [Macro]
このマクロは *spec*によりデフォルトフェイス spec が与えられるような名前つきフェイスとして *face*を宣言する。シンボル *face*はクォートせずに '`-face`'で終わらないこと (冗長かもしれない)。引数 *doc*はフェイスにたいするドキュメント文字列。追加の *keyword*引数は `defgroup`や `defcustom`の場合と同じ意味をもつ (Section 15.1 [Common Keywords], page 250 を参照)。

*face*がすでにデフォルトフェイス spec をもつ場合には、このマクロは何も行わない。

デフォルトフェイス spec は何もカスタマイゼーション (Chapter 15 [Customization], page 250 を参照) の効果がないときの *face*の外観を決定する。*face*が (Custom テーマや init ファイルから読み込んだカスタマイズにより) すでにカスタマイズ済みなら、その外観はデ

フォルトフェイス spec の *spec*をオーバーライドするカスタムフェイス spec により決定される。しかしその後カスタマイゼーションが削除されたら、*face*の外観は再びそのデフォルトフェイス spec により決定されるだろう。

例外として Emacs Lisp モードで *C-M-x* (`eval-defun`) から `defface`を評価した場合には、`eval-defun`の特別な機能により `defface`が何を指示するかをフェイスが正確に反映するように、そのフェイス上の任意のカスタムフェイスをオーバーライドする。

*spec*引数は異なる種別の端末上でそのフェイスがどのような外観で表示されるべきかを示すフェイス *spec*。これは各要素が以下の形式であるような alist であること

```
(display . plist)
```

*display*は端末のクラス (以下参照) を指定する。*plist*はそのような端末上でフェイスがどのような外観かを指定するフェイス属性とその値からなるプロパティリストであること。後方互換性のために `(display plist)`のように要素を記述することもできる。

*spec*の要素の *display*の部分は、その要素がマッチする端末を決定する。与えられた端末にたいして複数の要素がマッチした場合には、最初にマッチした要素がその端末にたいして使用される。*display*には以下の 3 つが可能:

`default` *spec*のこの要素はどの端末にもマッチしない。かわりにすべての端末に適用されるデフォルトを指定する。この要素が使用する場合には、*spec*の最初の要素でなければならない。この後の要素はこれらのデフォルトの一部、またはすべてをオーバーライドできる。

`t` *spec*のこの要素はすべての端末にマッチする。したがって *spec*の後続要素が使用されることはない。`t`は通常は *spec*の最後 (か唯一) の要素として使用される。

リスト *display*がリストなら各要素は `(characteristic value...)`という形式をもつこと。ここで *characteristic*は端末をクラス分けする方法、*value*は *display*に適用されるべき可能なクラス分類。*characteristic*に利用可能な値は:

`type` その端末が使用するウィンドウシステムの種類で `graphic` (任意のグラフィック対応ディスプレイ)、`x`、`pc` (MS-DOS コンソール)、`w32` (MS Windows 9X/NT/2K/XP)、または `tty` (グラフィック非対応ディスプレイ) のいずれか。Section 39.24 [Window Systems], page 1083 を参照のこと。

`class` その端末がサポートするカラーの種類であり `color`、`grayscale`か `mono`のいずれか。

`background`
バックグラウンドの種類であり `light`か `dark`のいずれか。

`min-colors`
その端末がサポートするべき最小カラー数を表す整数。端末の `display-color-cells`の値が少なくとも指定された整数ならその端末にマッチ。

`supports` その端末が *value*...で与えられたフェイス属性を表示可能か否か(Section 39.12.1 [Face Attributes], page 1012 を参照)。このテストがどのように行われるかについてのより正確な情報は [Display Face Attribute Testing], page 752 を参照のこと。

与えられた *characteristic*にたいして *display*の要素が複数の *value*を指定する場合には、いずれの値も許容され得る。*display*が複数の要素をもつ場合には、各要素は異なる *characteristic*を指定すること。その端末のそれぞれの characteristic は *display*内で指定された値のいずれか 1 つとマッチしなければならない。

たとえば以下は標準フェイス `highlight`の定義です:

```
(defface highlight
  '((((class color) (min-colors 88) (background light))
     :background "darkseagreen2")
    (((class color) (min-colors 88) (background dark))
     :background "darkolivegreen")
    (((class color) (min-colors 16) (background light))
     :background "darkseagreen2")
    (((class color) (min-colors 16) (background dark))
     :background "darkolivegreen")
    (((class color) (min-colors 8))
     :background "green" :foreground "black")
    (t :inverse-video t))
  "Basic face for highlighting."
  :group 'basic-faces)
```

内部的には Emacs はフェイスのシンボルプロパティ`face-defface-spec`内にそれぞれのフェイスのデフォルト spec を格納します (Section 9.4 [Symbol Properties], page 129 を参照)。`saved-face`プロパティはカスタマイゼーションバッファーを使用してユーザーが保存した任意のフェイス spec を格納します。`customized-face`プロパティはカレントセッションにたいしてカスタマイズされた保存されていないフェイス spec を格納します。そして `theme-face`プロパティはそのフェイスにたいするアクティブなカスタマイゼーションセッティングと、フェイス spec をもつ Custom テーマを関連付ける alist です。そのフェイスのドキュメント文字列は `face-documentation`プロパティ内に格納されます。

フェイスは通常は `defface`を使用して 1 回だけ宣言されて、その外観にたいするそれ以上の変更は Customize フレームワーク (Customize ユーザーインターフェースか `custom-set-faces`関数を通じて。Section 15.5 [Applying Customizations], page 266 を参照)、またはフェイスリマッピング (Section 39.12.5 [Face Remapping], page 1021 を参照) により行われます。Lisp から直接フェイス spec 変更を要する稀な状況では `face-spec-set`関数を使用できます。

`face-spec-set` *face spec* **&optional** *spec-type* [Function]

この関数は `face`にたいするフェイス spec として *spec*を適用する。*spec*は上述した `defface`にたいするフェイス spec であること。

この関数はもし *face*が既存のものでなければ有効なフェイス名として *face*を定義して、既存フレームのその属性の (再) 計算も行う。

オプション引数 *spec-type*はどの spec をセットするかを決定する。これが省略か `nil`、または `face-override-spec`なら、この関数はオーバーライド *spec(override spec)* をセットする。これは後述する *face*上の他のすべてのフェイス spec をオーバーライドする。これは Custom のコード外部からこの関数を呼び出す際に有用。*spec-type*が `customized-face`か `saved-face`なら、この関数はカスタマイズされた spec、または保存されたカスタム spec をセットする。`face-defface-spec`なら、この関数はデフォルトフェイス spec(`defface`によりセットされるものと同一) をセットする。`reset`なら、この関数は *face*からすべてのカスタマイゼーション

spec とオーバーライド spec をクリアーする (この場合には *spec*の値は無視される)。*spec-type* の他の任意の値がフェイス spec に及ぼす効果は内部的な使用のために予約済みだが、それでも後述するようにこの関数は *face*自体の定義を行い属性を再計算する。

39.12.3 フェイス属性のための関数

このセクションでは名前つきフェイスの属性に直接アクセスしたり変更する関数を説明します。

`face-attribute` *face attribute* **&optional** *frame inherit* [Function]

この関数は *frame*上の *face*にたいする属性 *attribute*の値をリターンする。

*frame*が省略か `nil`なら選択されたフレームを意味する (Section 29.10 [Input Focus], page 732 を参照)。*frame*が `t`なら、この関数は新たに作成されるフレームにたいして指定された属性の値をリターンする (これは下記の `set-face-attribute`を使用して何らかの値を指定していなければ通常は `unspecified`)。

*inherit*が `nil`なら *face*により定義される属性だけが考慮されるのでリターンされる値は `unspecified`、または相対的な値かもしれない。*inherit*が非 `nil`なら *face*の *attribute*の定義が、`:inherit`属性で指定されたフェイスとマージされる。しかしリターンされる値は依然として `unspecified`、または相対的な値かもしれない。*inherit*がフェイスかフェイスのリストなら、指定された絶対的な値になるまで結果はそのフェイス (1 つ以上) と更にマージされる。

リターン値が指定されていて、かつ絶対的であることを保証するためには *inherit*にたいして `default`の値を使用すること。(常に完全に指定される)`default`フェイスとマージすることにより、すべての未指定や相対的な値は解決されるだろう。

たとえば

```
(face-attribute 'bold :weight)
     ⇒ bold
```

`face-attribute-relative-p` *attribute value* [Function]

この関数は *value*がフェイス属性 *attribute*の値として使用された際に相対的なら非 `nil`をリターンする。これはフェイスリスト内の後続のフェイス、または継承した他のフェイスが由来となる任意の値で完全にオーバーライドするのではなく、それが変更されるであろうことを意味する。

すべての属性にたいして `unspecified`は相対的な値。`:height`にたいしては浮動小数点数と関数値も相対的である。

たとえば:

```
(face-attribute-relative-p :height 2.0)
     ⇒ t
```

`face-all-attributes` *face* **&optional** *frame* [Function]

この関数は *face*の属性の alist をリターンする。結果の要素は `(attr-name . attr-value)` という形式の名前/値ペアー。オプション引数 *frame*はリターンするべき *face*の定義をもつフレームを指定する。省略か `nil`ならリターン値には新たに作成されるフレームにたいする *face* のデフォルト属性が記述される。

`merge-face-attribute` *attribute value1 value2* [Function]

*value1*がフェイス属性 *attribute*にたいして相対的な値なら、基礎的な値 *value2*とマージしてリターンする。それ以外の場合には *value1*がフェイス属性 *attribute*にたいして絶対的な値なら *value1*を変更せずにリターンする。

Emacs は通常は各フレームのフェイス属性を自動的に計算するために、各フェイスのフェイス spec を使用します (Section 39.12.2 [Defining Faces], page 1015 を参照)。関数 `set-face-attribute` は特定またはすべてのフレームのフェイスに直接属性を割り当てることにより、この計算をオーバーライドできます。この関数は主として内部的な使用を意図したものです。

`set-face-attribute` *face frame* **&rest** *arguments* [Function]
この関数は *frame*にたいする *face*の 1 つ以上の属性をセットする。この方法で指定された属性は *face*に属するフェイス spec(1 つ以上) をオーバーライドする。

余分の引数 *arguments*はセットするべき属性と値を指定する。これらは (`:family`や `:underline`のような) 属性名と値が交互になるように構成されていること。つまり、

```
(set-face-attribute 'foo nil :weight 'bold :slant 'italic)
```

これは属性`:weight`を `bold`、. 属性`:slant`を `italic`にセットする。

*frame*が `t`ならこの関数は新たに作成されるフレームにたいするデフォルト属性をセットする。*frame*が `nil`ならこの関数はすべての既存フレーム、同様に新たに作成されるフレームにたいしてその属性をセットする。

以下のコマンドと関数は主として古いバージョンの Emacs にたいする互換性のために提供されます。これらは `set-face-attribute`を呼び出すことにより機能します。これらの *frame*引数にたいする値 `t`と `nil` (や省略) は `set-face-attribute`や `face-attribute`の場合と同様に処理されます。コマンドがインタラクティブに呼び出されるとミニバッファーを使用して引数を読み取ります。

`set-face-foreground` *face color* **&optional** *frame* [Command]
`set-face-background` *face color* **&optional** *frame* [Command]
これらはそれぞれ *face*の`:foreground`属性、または`:background`属性に *color*をセットする。

`set-face-stipple` *face pattern* **&optional** *frame* [Command]
これは *face*の`:stipple`属性に *pattern*をセットする。

`set-face-font` *face font* **&optional** *frame* [Command]
*face*のフォント関連の属性を *font* (文字列かフォントオブジェクト) の属性に変更する。*font*引数でアポートされるフォーマットについては [face-font-attribute], page 1014 を参照のこと。この関数はフェイスの`:font`属性、および間接的に`:family`、`:foundry`、`:width`、`:height`、`:weight`、`:slant`の属性もそのフォントで定義された値にセットする。*frame*が非 `nil`なら、指定したフレームの属性だけを変更する。

`set-face-bold` *face bold-p* **&optional** *frame* [Function]
これは *face*の`:weight`属性にたいして *bold-p*が `nil`なら *normal*、それ以外なら *bold*をセットする。

`set-face-italic` *face italic-p* **&optional** *frame* [Function]
これは *face*の`:slant`属性にたいして *italic-p*が `nil`なら *normal*、それ以外なら *italic*をセットする。

`set-face-underline` *face underline* **&optional** *frame* [Command]
これは *face*の`:underline`属性に *underline*をセットする。

`set-face-inverse-video` *face inverse-video-p* **&optional** *frame* [Command]
これは *face*の`:inverse-video`属性に *inverse-video-p*をセットする。

`invert-face` *face* **&optional** *frame* [Command]
これはフェイス *face*のフォアグラウンドカラーとバックグラウンドカラーを交換する。

`set-face-extend` *face extend* **&optional** *frame* [Command]
これは *face*の`:extend`属性に *extend*をセットする。

以下はフェイスの属性を調べる関数です。これらは主として古いバージョンの Emacs との互換性のために提供されます。これらにたいして *frame*を指定しなければ選択されたフレーム、`t`なら新たなフレームにたいするデフォルトデータを参照します。フェイスがその属性にたいして何の値も定義していなければ`unspecified`がリターンされます。*inherit*が`nil`ならそのフェイスにより直接定義された属性だけがリターンされます。*inherit*が非`nil`ならそのフェイスの`:inherit`属性により指定される任意のフェイス、*inherit*がフェイスまたはフェイスのリストなら指定された属性が見つかるまでそれらも考慮します。リターンされる値が常に指定された値であることを保証するためには *inherit*に値`default`を使用してください。

`face-font` *face* **&optional** *frame character* [Function]
この関数はフェイス *face*のフォント名をリターンする。

オプション引数 *frame*が指定されたら、そのフレームの *face*のフォント名をリターンする。*frame*が省略か`nil`なら選択されたフレームを使用する。後者の場合にはオプションの 3 つ目の引数 *character*が与えられると、*character*にたいして使用されるフォント名をリターンする。

`face-foreground` *face* **&optional** *frame inherit* [Function]
`face-background` *face* **&optional** *frame inherit* [Function]
これらの関数はそれぞれフェイス *face*のフォアグラウンドカラーまたはバックグラウンドカラーを文字列としてリターンする。カラーが未指定ならをリターンする。

`face-stipple` *face* **&optional** *frame inherit* [Function]
この関数はフェイス *face*のバックグラウンド点描パターンの名前、もしなければ`nil`をリターンする。

`face-bold-p` *face* **&optional** *frame inherit* [Function]
この関数は *face*の`:weight`属性が normal より bold 寄り (`semi-bold`、`bold`、`extra-bold`、`ultra-bold`のいずれか) なら非`nil`、それ以外なら`nil`をリターンする。

`face-italic-p` *face* **&optional** *frame inherit* [Function]
この関数は *face*の`:slant`属性が`italic`か`oblique`なら非`nil`、それ以外なら`nil`をリターンする。

`face-underline-p` *face* **&optional** *frame inherit* [Function]
この関数はフェイス *face*が非`nil`の`:underline`属性を指定すれば非`nil`をリターンする。

`face-inverse-video-p` *face* **&optional** *frame inherit* [Function]
この関数はフェイス *face*が非`nil`の`:inverse-video`属性を指定すれば非`nil`をリターンする。

`face-extend-p` *face* **&optional** *frame* [Function]
この関数はフェイス *face*が非`nil`の`:extend`属性を指定すれば非`nil`をリターンする。

39.12.4 フェイスの表示

Emacs が与えられたテキスト断片を表示する際には、そのテキストの視覚的外観は異なるソースから描画されるフェイスにより決定されるかもしれません。これら種々のソースが特定の文字にいたいして複数のフェイスを指定する場合には、Emacs はそれらのさまざまなフェイスの属性をマージします。以下に Emacs がフェイスをマージする順序を優先度順に記します:

- そのテキストが特別なグリフで構成される場合には、そのグリフは特定のフェイスを指定できる。Section 39.22.4 [Glyphs], page 1081 を参照のこと。
- アクティブなリージョンにテキストがある場合には、Emacs は `region`フェイスを使用してそれをハイライトする。Section “Standard Faces” in *The GNU Emacs Manual* を参照のこと。
- 非 `nil`の `face`属性をもつオーバーレイにテキストがある場合には、Emacs はそのプロパティにより指定されるフェイス (1 つ以上) を適用する。そのオーバーレイが `mouse-face`プロパティをもち、マウスがそのオーバーレイに十分に近ければ Emacs はかわりに `mouse-face`で指定されるフェイスかフェイス属性を適用する。Section 39.9.2 [Overlay Properties], page 1002 を参照のこと。

  1 つの文字を複数のオーバーレイがカバーする場合には、高優先度のオーバーレイが低優先度のオーバーレイをオーバーライドする。Section 39.9 [Overlays], page 999 を参照のこと。
- そのテキストが `face`や `mouse-face`プロパティを含む場合には、Emacs は指定されたフェイスやフェイス属性を適用する。Section 32.19.4 [Special Properties], page 820 を参照のこと (これは Font Lock モードのフェイス適用方法。Section 23.6 [Font Lock Mode], page 496 を参照)。
- そのテキストが選択されたウィンドウのモードラインにある場合には、Emacs は `mode-line`フェイスを適用する。選択されていないウィンドウのモードラインでは Emacs は `mode-line-inactive`フェイスを使用する。ヘッダーラインにたいしては Emacs は `header-line`フェイスを適用する。タブラインにたいしては、Emacs は `tab-line`フェイスを適用する。
- `before-string`や `after-string`プロパティを介したオーバーレイ文字列 (Section 39.9.2 [Overlay Properties], page 1002 を参照)、あるいはディスプレイ文字列 (Section 39.16.4 [Other Display Specs], page 1046 を参照) に由来するテキストであり、かつ文字列に `face`や `mouse-face`のプロパティが含まれない、またはそれらのプロパティが何らかのフェイス属性を未定義のままにしているが、バッファーのテキストがフェイスを定義するディスプレイプロパティやオーバーレイプロパティの影響を受ける場合には、Emacs は “基礎” にあるバッファーテキストのフェイスやフェイス属性を適用する。これはたとえオーバーレイ文字列やディスプレイ文字列がマージン内に表示されている場合も同様であることに注意 (Section 39.16.5 [Display Margins], page 1047 を参照)。
- 先行ステップの間に与えられた属性が指定されなければ、Emacs は `default`フェイスの属性を適用する。

各ステージにおいてフェイスが有効な`:inherit`属性をもつ場合には、Emacs は値 `unspecified`をもつすべての属性が、親フェイス (1 つ以上) 由来で描画に使用される対応する値をもつものとして扱います。Section 39.12.1 [Face Attributes], page 1012 を参照してください。親フェイスでも属性が unspecified のままかもしれないことに注意してください。その場合にはフェイスマージの次レベルでもその属性は unspecified のままです。

39.12.5 フェイスのリマップ

変数 `face-remapping-alist`はあるフェイスの外観のバッファーローカル、またはグローバルな変更にたいして使用されます。たとえばこれは `text-scale-adjust`コマンド (Section “Text Scale” in *The GNU Emacs Manual* を参照) の実装に使用されています。

`face-remapping-alist` [Variable]

この変数の値は要素が (*face . remapping*) という形式をもつ alist。これにより Emacs はフェイス *face*をもつ任意のテキストを、通常の *face*の定義ではなく *remapping*で表示する。

*remapping*にはテキストプロパティ`face`にたいして適切な任意のフェイス spec、すなわちフェイス (フェイス名か属性/値ペアーのプロパティリスト)、またはフェイスのリストのいずれかを指定できる。詳細は Section 32.19.4 [Special Properties], page 820 の `face`テキストプロパティの記述を参照のこと。*remapping*はリマップされるフェイスにたいる完全な仕様としての役目をもつ。これは通常の *face*を変更せずに置き換える。

`face-remapping-alist`がバッファーローカルなら、そのローカル値はそのバッファーだけに効果をもつ。(`:filtered` (`:window` *param val*) *spec*) を使用することにより特定のウィンドウだけに適用されるフェイスを含んだ `face-remapping-alist`では、そのフェイスはそのフィルター条件にマッチするだけに効果を及ぼす (Section 32.19.4 [Special Properties], page 820 を参照)。フェイスのフィルタリングを一時的にオフにするには、`face-filters-always-match`を非 `nil`値にバインドすれば、すべてのフェイスフィルターは任意のウィンドウにマッチする。

注意: フェイスのリマッピングは再帰的ではない。*remapping*が同じフェイス名 *face*を参照する場合には、直接または *remapping*内の他の何らかのフェイスの`:inherit`属性を通じて、その参照は *face*の通常の定義を使用する。たとえば`mode-line`フェイスが`face-remapping-alist`内の以下のエントリーでリマップされるなら:

```
(mode-line italic mode-line)
```

`mode-line`フェイスの新たな定義は`italic`フェイス、および (リマップされていない) 通常の`mode-line`フェイスの定義から継承される。

以下の関数は `face-remapping-alist`にたいする高レベルなインターフェースを実装します。ほとんどの Lisp コードはリマッピングが他の場所に適用されてしまうのを避けるために、`face-remapping-alist`を直接セットするのではなくこれらの関数を使用するべきです。これらの関数はバッファーローカルなリマッピングを意図しており、すべてが副作用として `face-remapping-alist`をバッファーローカルにします。これらは以下の形式の`face-remapping-alist`エントリーを管理します

```
(face relative-spec-1 relative-spec-2 ... base-spec)
```

上述したように *relative-spec-N*と *base-spec*はそれぞれフェイス名か属性/値ペアーのプロパティリストです。相対的リマッピング *(relative remapping)* エントリー *relative-spec-N*はそれぞれ関数`face-remap-add-relative`と`face-remap-remove-relative`により管理されます。これらはテキストサイズ変更のような単純な変更を意図しています。ベースリマッピング *(base remapping)* エントリー *base-spec*は最低の優先度をもち、関数`face-remap-set-base`と`face-remap-reset-base`により管理されます。これはメジャーモードが制御下のバッファーでフェイスをリマップするために用いることを意図しています。

`face-remap-add-relative` *face* **&rest** *specs* [Function]

この関数はカレントバッファー内のフェイス *face*にたいして、相対的リマッピングとして *specs*内にフェイス spec を追加する。残りの引数 *specs*はフェイス名のリスト、または属性/値ペアーのプロパティリストのいずれかの形式であること。

リターン値は cookie としての役目をもつ Lisp オブジェクト。後でそのリマッピングの削除を要する場合には、引数として `face-remap-remove-relative`にこのオブジェクトを渡すことができる。

```
;; 'escape-glyph'フェイスを'highlight'と'italic'
```

```
;; を組み合わせたフェイスにリマップする
(face-remap-add-relative 'escape-glyph 'highlight 'italic)

;; 'default'フェイスのサイズを 50%増加:
(face-remap-add-relative 'default :height 1.5)
```

`face-remap-remove-relative` *cookie* [Function]
この関数は以前 `face-remap-add-relative`で追加された相対的リマッピングを削除する。*cookie*はリマッピングが追加されたときに `face-remap-add-relative`がリターンした Lisp オブジェクトであること。

`face-remap-set-base` *face* **&rest** *specs* [Function]
この関数はカレントバッファー内の *face*のベースリマッピングを *specs*にセットする。*specs*が空なら `face-remap-reset-base`(以下参照) を呼び出したようにデフォルトベースリマッピングがリストアされる。これは単一の値 `nil`を含む *specs*とは異なることに注意。これは逆の結果をもたらす (*face*のグローバル定義は無視される)。

これはグローバルなフェイス定義を継承したデフォルトの *base-spec*を上書きするので、必要ならそのような継承を追加するのは呼び出し側の責任である。

`face-remap-reset-base` *face* [Function]
この関数は *face*のベースリマッピングに、*face*のグローバル定義から継承したデフォルト値にセットする。

39.12.6 フェイスを処理するための関数

以下はフェイスの作成や処理を行う追加の関数です。

`face-list` [Function]
この関数はすべての定義済みフェイス名のリストをリターンする。

`face-id` *face* [Function]
この関数はフェイス *face*のフェイス番号 *(face number)* をリターンする。これは Emacs 内部の低レベルでフィエスを一意に識別する番号。フェイス番号によるフェイスの参照を要するのは稀である。しかし `make-glyph-code`や `glyph-face` (Section 39.22.4 [Glyphs], page 1081 を参照) のようなグリフを操作する関数は内部的にフェイス番号にアクセスする。フェイス番号はフェイスシンボルの `face`プロパティの値として格納されることに注意。このフェイスプロパティにあなた自身が値をセットしないことを推奨する。

`face-documentation` *face* [Function]
この関数はフェイス *face*のドキュメント文字列、指定されていなければ `nil`をリターンする。

`face-equal` *face1 face2* **&optional** *frame* [Function]
これはフェイス *face1*とフェイス *face2*が表示にたいして同じ属性をもつなら `t`をリターンする。

`face-differs-from-default-p` *face* **&optional** *frame* [Function]
これはフェイス *face*の表示がデフォルトフェイスと異なるなら非 `nil`をリターンする。

フェイスエイリアス *(face alias)* はあるフェイスにたいして等価な名前を提供します。エイリアスシンボルの `face-alias`プロパティに対象となるフェイス名を与えることによってフェイスエイリアスを定義できます。以下の例では `mode-line`フェイスにたいするエイリアスとして `modeline`を作成します。

```
(put 'modeline 'face-alias 'mode-line)
```

`define-obsolete-face-alias` *obsolete-face current-face when* [Macro]
このマクロは *current-face*のエイリアスとして `obsolete-face`を定義するとともに、将来に削除されるかもしれないことを示すために obsolete(時代遅れ) とマークする。*when*は `obsolete-face`が obsolete になる時期を示す文字列であること (通常はバージョン番号文字列)。

39.12.7 フェイスの自動割り当て

以下のフックはバッファー内のテキストに自動的にフェイスを割り当てるために使用されます。これは Jit-Lock モードの実装の一部であり Font-Lock により使用されます。

`fontification-functions` [Variable]
この変数は再表示を行う直前に Emacs の再表示により呼び出される関数のリストを保持する。これらは Font Lock が有効でないときでも呼び出される。Font Lock モードが有効なら、この変数は通常は単一の関数 `jit-lock-function`だけを保持する。

関数はバッファー位置 *pos*を単一の引数としてリストされた順に呼び出される。これらはカレントバッファー内の *pos*で開始されるテキストにたいして集合的にフェイスの割り当てを試みること。

関数は `face`プロパティをセットすることにより割り当てるフェイスを記録すること。またフェイスを割り当てたすべてのテキストに非 `nil`の `fontified`プロパティも追加すること。このプロパティは再表示にたいして、そのテキストにたいしてそのフェイスがすでに割り当て済みであることを告げる。

*pos*の後の文字がすでに非 `nil`の `fontified`プロパティをもつがフォント表示化を要さない場合には、何も行わない関数を追加するのがおそらくよいアイデアである。ある関数が前の関数による割り当てをオーバーライドする場合には、実際に問題となるのは最後の関数終了後のプロパティである。

効率化のために通常は各呼び出しにおいて 400 から 600 前後の文字にフェイスを割り当てるように、これらの関数を記述することを推奨する。

39.12.8 基本的なフェイス

テキストにたいして Emacs Lisp プログラムが何らかのフェイス割り当てを要する場合には、完全に新たなフェイスを定義するより特定の既存フェイス、またはそれらを継承したフェイスを使用するほうがよいアイデアである場合がしばしばあります。Emacs に特定の外観を与えるために別のユーザーが基本フェイス (basic face) をカスタマイズしていても、この方法なら追加のカスタマイズなしでプログラムは適合することでしょう。

以下に Emacs が定義する基本フェイスのいくつかをリストしました。これらに加えて、ハイライトが Font Lock モードによりまだ処理されていなかったり、いくつかの Font Lock フェイスが使用されていなければ、構文的ハイライトのために Font Lock フェイスを使うようにしたいと思うかもしれません。Section 23.6.7 [Faces for Font Lock], page 504 を参照してください。

`default` 属性がすべて指定されたデフォルトフェイス。他のすべてのフェイスは暗にこのフェイスを継承する。未指定 (unspecified) な任意の属性は、このフェイスの属性をデフォルトとする (Section 39.12.1 [Face Attributes], page 1012 を参照)。

`bold`
`italic`
`bold-italic`
`underline`
`fixed-pitch`
`fixed-pitch-serif`
`variable-pitch`
: これらは名前に示されるような属性をもち (`bold`は bold の`:weight`属性をもつ)、それ以外のすべての属性は未指定 (そのために`default`により与えられる)。

`shadow`
: テキストの淡色表示 (dimmed out) 用。たとえばこれはミニバッファー内で無視されるファイル名部分に使用される (Section “Minibuffers for File Names” in *The GNU Emacs Manual* を参照)。

`link`
`link-visited`
: ユーザーを別のバッファーや位置へと送るクリック可能テキストボタン用。

`highlight`
: 一時的に強調するべきテキスト範囲用。たとえば一般的にカーソルのハイライトには`mouse-face`プロパティが割り当てられる (Section 32.19.4 [Special Properties], page 820 を参照)。

`match`
`isearch`
`lazy-highlight`
: それぞれ定型検索 (permanent search) のマッチ、インタラクティブ検索のマッチ、カレントのインタラクティブな検索のマッチ以外の lazy ハイライトにたいするテキスト用。

`error`
`warning`
`success`
: エラー、警告、成功に関するテキスト用。たとえば`*Compilation*`内のメッセージにたいして使用される。

39.12.9 フォントの選択

Emacs がグラフィカルなディスプレイ上で文字を描画可能になる前に、まずその文字にたいするフォント *(font)* を選択しなければなりません [2]。Section “Fonts” in *The GNU Emacs Manual* を参照してください。Emacs は通常はその文字に割り当てられたフェイス、特にフェイス属性`:family`、`:weight`、`:slant`、`:width`(Section 39.12.1 [Face Attributes], page 1012 を参照) にもとづいて自動的にフォントを選択します。フォントの選択は表示される文字にも依存します。表示できるのは文字セットが限定されているフォントもいくつかあります。利用可能なフォントがこの要件を完全に満たさなければ Emacs はもっとも近いフォント *(closest matching font)* を探します。このセクション内の変数は Emacs がこの選択を行う方法を制御します。

`face-font-family-alternatives` [User Option]
: ある family が指定されたが存在しなければ、この変数は試みるべき代替えのフォントファミリーを指定する。各要素は以下の形式をもつ:

```
(family alternate-families...)
```

[2] このコンテキストでは用語 *font* は Font Lock(Section 23.6 [Font Lock Mode], page 496 を参照) にたいして何も行いません。

*family*が指定されたが利用できなければ、Emacs は *alternate-families*で与えられるファミリーで存在するものが見つかるまで 1 つずつファミリーを試みる。

`face-font-selection-order` [User Option]
希望するすべてのフェイス属性 (`:width`、`:height`、`:weight`、`:slant`) に完全にマッチするフォントが存在しなければ、この変数はもっとも近いフォントの選択時に考慮すべきこれらの属性の順序を指定する。値はこれらの属性シンボルを重要度降順で含むリストであること。デフォルトは (`:width :height :weight :slant`)。

フォント選択はまずこのリスト内の最初の属性にたいして利用可能な最適マッチを探す。その後に、この方法で最適なフォントの中から 2 つ目の属性にたいして最適なマッチを検索、... のように選択を行う。

属性`:weight`と`:width`は`normal`を中心とする範囲のようなシンボリック値をもつ。より極端 (`normal`から離れた) なマッチは、より極端ではない (`normal`に近い) マッチより幾分優先される。これは可能なかぎり非 normal なフェイスが、normal なフェイスとは対照的になることを保証するようにデザインされている。

この変数が違いを生むケースの例はデフォルトフォントに等価なイタリックがない場合である。デフォルトの順では`italic`フェイスはデフォルトのフォントに類似した非イタリックのフォントを使用するだろう。しかし`:height`の前に`:slant`を配置すると、`italic`フェイスはたとえ height が同じでなくともイタリックフォントを使用するだろう。

`face-font-registry-alternatives` [User Option]
この変数は registry が指定されたがそれが存在しない場合に試みるべき代替えのフォントレジストリーを指定する。各要素は以下の形式をもつ:

```
(registry alternate-registries...)
```

*registry*が指定されたが利用できなければ、Emacs は *alternate-registries*内で存在するレジストリーが見つかるまで他のレジストリーを 1 つずつ試みる。

Emacs がスケーラブルフォントを使用するようにできますがデフォルトではそれらを使用しないようになっています。

`scalable-fonts-allowed` [User Option]
この変数はどのスケーラブルフォントを使用するかを制御する。値`nil` (デフォルト) はスケーラブルフォントを使用しないことを意味する。`t`はそのテキストにたいして適切と思われる任意のスケーラブルフォントを使用することを意味する。

それ以外なら値は正規表現のリストであること。その場合には名前がこのリスト内の正規表現にマッチする任意のスケーラブルフォントの使用が有効になる。たとえば、

```
(setq scalable-fonts-allowed '("iso10646-1$"))
```

これはレジストリーが`iso10646-1`のようなスケーラブルフォントの使用を可能にする。

`face-font-rescale-alist` [Variable]
この変数は特定のフォントにたいするスケーリングを指定する。値は以下の形式の要素をもつリストであること

```
(fontname-regexp . scale-factor)
```

使用しようとするフォントの名前が *fontname-regexp*にマッチする場合には、これはファクター *scale-factor*に対応した同様な大きさのフォントの選択を指示する。特定のフォントが提示する通常の height や width が大きい、または小さい場合にフォントサイズを正規化するためにこの機能を使用できるだろう。

39.12.10 フォントの照会

`x-list-fonts` *name* **&optional** *reference-face frame maximum width* [Function]

この関数は *name*にマッチする利用可能なフォント名のリストをリターンする。*name*は Fontconfig、GTK+、または XLFD のいずれかのフォーマットによるフォント名を含む文字列であること (Section "Fonts" in *The GNU Emacs Manual* を参照)。XLFD 文字列ではワイルドカード文字が使用できる。'`*`'文字は任意の部分文字列、'`?`'は任意の単一文字にマッチする。フォント名のマッチングでは case(大文字小文字) の違いは無視される。

オプション引数 *reference-face*と *frame*が指定された場合には、リターンされるリストにはその時点でフレーム *frame*上での *reference-face* (フェイス名) と同じサイズのフォントだけが含まれる。

オプション引数 *maximum*はリターンされるフォント数の制限をセットする。これが非 `nil`ならリターン値は最初にマッチした *maximum*個のフォントの後が切り捨てられる。*maximum*に小さい値を指定すれば、そのパターンに多くのフォントがマッチするような場合に関数をより高速にできる。

オプション引数 *width*は希望するフォントの幅を指定する。これが非 `nil`なら、この関数は文字の幅 (平均) が *reference-face*の *width*倍の幅であるようなフォントだけをリターンする。

`x-family-fonts` **&optional** *family frame* [Function]

この関数は *frame*上のファミリー *family*にたいして利用可能なフォントを記述するリストをリターンする。*family*が省略か `nil`ならこのリストはすべてのファミリーに適用されて、それはすなわち利用可能なすべてのフォントを含む。それ以外なら *family*は文字列であること。これにはワイルドカード '`?`'と '`*`'を含めることができる。

このリストは *frame*のあるディスプレイを記述する。*frame*が省略か `nil`なら、これは選択されたフレームのディスプレイに適用される (Section 29.10 [Input Focus], page 732 を参照)。

このリスト内の各要素は以下の形式のベクターであること:

```
[family width point-size weight slant
 fixed-p full registry-and-encoding]
```

最初の 5 つの要素はフェイス属性に対応する。あるフェイスにたいしてこれらの属性を指定した場合には、そのフォントが使用されるだろう。

最後の 3 つの要素は、そのフォントに関する追加の情報を与える。そのフォントが固定ピッチ (fixed-pitch) でなければ *fixed-p*は非 `nil`。*full*はそのフォントのフルネーム、*registry-and-encoding*はそのフォントのレジストリーとエンコーディングを与える。

39.12.11 フォントセット

フォントセット *(fontset)* とは、それぞれが文字コードの範囲に割り当てられるフォントのリストのことです。個々のフォントでは Emacs がサポートする文字の全範囲を表示できませんが、フォントセットであれば表示することができます。フォントのようにフォントセットは名前をもつことができ、フレームやフェイスにたいしてフォントを指定する際に、フォント名としてフォントセット名を使用できます。以下は Lisp プログラム制御下でのフォントセット定義に関する情報です。

`create-fontset-from-fontset-spec` *fontset-spec* **&optional** *style-variant-p noerror* [Function]

この関数は仕様文字列 *fontset-spec*に応じて新たなフォントセットを定義する。この文字列は以下のような形式であること:

```
fontpattern, [charset:font]...
```

カンマの前後の空白文字は無視される。

この文字列の最初の部分 *fontpattern*は、最後の 2 つのフィールドが '`fontset-alias`'であることを除外して標準 X フォント名形式をもつこと。

新たなフォントセットは long 名と short 名という 2 つの名前をもつ。long 名はそれ全体が *fontpattern*、short 名は '`fontset-alias`'。いずれの名前でもこのフォントセットを参照できる。同じ名前がすでに存在するフォントセットでは *noerror*が `nil`ならエラーがシグナルされ、*noerror*が非 `nil`ならこの関数は何も行わない。

オプション引数 *style-variant-p*が非 `nil`なら、そのフォントセットの bold、italic、および bold-italic も同様に作成するよう指示する。これらの変種フォントセットは short 名をもたず bold、および/または italic を示すように *fontpattern*を変更して作成した long 名だけをもつ。

仕様文字列はそのフォントセット内でどのフォントを使用するかも宣言する。詳細は以下を参照。

構文 '*charset*`:`*font*'はある特定の文字セットにたいして、(このフォントセット内の) どのフォントを使用するかを指定します。ここで *charset*は文字セットの名前、*font*はその文字セットにたいして使用するフォントです。仕様文字列内ではこの構文を任意の回数使用できます。

明示的に指定しなかった残りの文字セットにたいして、Emacs は *fontpattern*にもとづいてフォントを選択します。これは '`fontset-`*alias*'をその文字セットを命名する値に置き換えます。文字セット ASCIIにたいしては、'`fontset-`*alias*'は '`ISO8859-1`'に置き換えられます。

加えて後続の複数フィールドがワイルドカードなら、Emacs はそれらを 1 つのワイルドカードにまとめます。これは自動スケールフォント (auto-scaled fonts) の使用を防ぐためです。フォントを大きくスケーリングすることにより作成されたフォントは編集に使用できず、小さくスケーリングされたフォントは、それ自身のサイズがより小さいフォントを使用する (Emacs が行う方法) ほうがよいので有用ではありません。

つまり以下のような *fontpattern*なら

```
-*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24
```

ASCIIにたいするフォント spec は以下のようになるでしょう:

```
-*-fixed-medium-r-normal-*-24-*-ISO8859-1
```

また Chinese GB2312 文字にたいするフォント spec は以下のようになるでしょう:

```
-*-fixed-medium-r-normal-*-24-*-gb2312*-*
```

上記のフォント spec にマッチする Chinese フォントをもっていないかもしれません。ほとんどの X ディストリビューションには、*family*フィールドに '`song ti`'か '`fangsong ti`'をもつ Chinese フォントだけが含まれます。そのような場合には以下のように '`Fontset-`*n*'を指定できます:

```
Emacs.Fontset-0: -*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24,\
       chinese-gb2312:-*-*-medium-r-normal-*-24-*-gb2312*-*
```

この場合には Chinese GB2312 以外のすべての文にたいするフォント spec は *family*フィールドに '`fixed`'をもち、Chinese GB2312 にたいするフォント spec は *family*フィールドにワイルドカード '`*`'をもちます。

`set-fontset-font` *name character font-spec* **&optional** *frame add* [Function]

この関数は指定された *character*にたいして、*font-spec*のフォントマッチングを使用するように既存のフォントセット *name*を変更する。

*name*が `nil`ならこの関数は *frame*のフォントセット、*frame*が `nil`なら選択されたフレームのフォントセットを変更する。

*name*が `t`ならこの関数は short 名が '`fontset-default`'であるようなデフォルトフォントセットを変更する。

*character*には `(from . to)`のようなコンスを指定できる。ここで *from*と *to*は文字コードポイント。この場合には範囲 *from*から *to* (両端を含む) までのすべての文字にたいして *font-spec*を使用する。

*character*には文字セット (Section 33.7 [Character Sets], page 861 を参照) も指定できる。この場合には、その文字セット内のすべての文字にたいして *font-spec*を使用する。

*character*にはスクリプト名 (Section 33.6 [Character Properties], page 857 を参照) も指定できる。この場合にはスクリプトに属するすべての文字にたいして *font-spec*を使用する。

*character*には `nil`も指定できる。この場合にはフォント仕様が指定されていないすべての文字に *font-spec*を使用する。

*font-spec*は関数 `font-spec`が作成した font-spec オブジェクトかもしれない (Section 39.12.12 [Low-Level Font], page 1029 を参照)。

*font-spec*にはコンス `(family . registry)`を指定できる。ここで *family*はフォントのファミリー名 (先頭に foundry 名が含まれるかもしれない)、*registry*はフォントのレジストリー名 (末尾にエンコーディング名が含まれるかもしれない)。

*font-spec*にはフォント名 (文字列) も指定できる。

*font-spec*には、指定された *character*にたいするフォントが存在しないことを明示的に指定する `nil`を指定できる。これはたとえば Unicode の PUA(Private Use Area) 由来のようなグリフをもたない文字のフォントにたいするシステムワイドの高価な検索を避けるために有用。

オプション引数 *add*が非 `nil`なら以前セットされたフォント spec に *font-spec*を追加する方法を指定する。`prepend`なら *font-spec*は先頭、`append`なら *font-spec*は末尾に追加される。デフォルトでは *font-spec*は以前のセッティングをオーバーライドする。

たとえば以下は文字セット `japanese-jisx0208`に属するすえての文字にたいして、ファミリー名が '`Kochi Gothic`'であるようなフォントを使用するようにデフォルトフォントセットを変更する。

```
(set-fontset-font t 'japanese-jisx0208
                  (font-spec :family "Kochi Gothic"))
```

`char-displayable-p` *char* [Function]

この関数は Emacs が *char*を表示できるようなら `t`をリターンする。より正確には選択されたフレームのフォントセットが、*char*が属する文字セットを表示するためのフォントをもてば `t`をリターンする。

フォントセットは文字単位でフォントを指定できる。フォントセットがこれを行う場合には、この関数の値は正確ではないかもしれない。

39.12.12 低レベルのフォント表現

通常はフォントを直接扱う必要はありません。これを行う必要がある場合にはこのセクションでその方法を説明します。

Emacs Lisp ではフォントはフォントオブジェクト *(font objects)*、フォント *spec(font specs)*、フォントエンティティー *(font entities)* という 3 つの異なる Lisp オブジェクトを使用して表現されます。

`fontp` *object* **&optional** *type* [Function]

*object*がフォントオブジェクト、フォント spec、フォントエンティティーなら `t`、それ以外なら `nil`をリターンする。

オプション引数 *type*が非 `nil`なら、チェックする Lisp オブジェクトの正確なタイプを決定する。この場合には *type*は `font-object`、`font-spec`、`font-entity`のいずれかであること。

フォントオブジェクトは Emacs がオープンしたフォントを表します。Lisp でフォントオブジェクトは変更できませんが調べることはできます。

`font-at` *position* **&optional** *window string* [Function]
ウィンドウ *window*内の位置 *position*にある文字を表示するために使用されているフォントオブジェクトをリターンする。*window*が `nil`の場合のデフォルトは選択されたウィンドウ。*string*が `nil`なら *position*はカレントバッファー内の位置を指定する。それ以外なら *string*は文字列、*position*はその文字列内での位置を指定すること。

フォント spec はフォントを探すために使用できる仕様セットを含む Lisp オブジェクトです。フォント spec 内の仕様にたいして 1 つ以上のフォントがマッチすることができます。

`font-spec` **&rest** *arguments* [Function]
*arguments*内の仕様を使用して新たなフォント spec をリターンする。これは `property-value`のペアーであること。可能な仕様は以下のとおり:

`:name` XLFD、Fontconfig、GTK+いずれかのフォーマットによるフォント名 (文字列)。Section “Fonts” in *The GNU Emacs Manual* を参照のこと。

`:family`
`:foundry`
`:weight`
`:slant`
`:width` これらは同じ名前のフェイス属性と同じ意味をもつ。Section 39.12.1 [Face Attributes], page 1012 を参照のこと。`:family`と`:foundry`は文字列で、それ以外はシンボル。例の値のように`:slant`は `italic`、`:weight`は `bold`、`:width`は `normal`かもしれない。

`:size` フォントサイズ。非負の整数はピクセル単位、浮動小数点数ならポイントサイズを指定する。

`:adstyle` ‘`sans`’のような、そのフォントにたいするタイポグラフィックスタイル (typographic style) の追加情報。値は文字列かシンボルであること。

`:registry`
‘`iso8859-1`’のようなフォントの文字セットレジストリーとエンコーディング。値は文字列かシンボルであること。

`:script` そのフォントがサポートしなければならないスクリプト (シンボル)。

`:lang` そのフォントがサポートするべき言語。値は名前が 2 文字の ISO-639 言語名であるようなシンボルであること。X では値は (もし空でなければ) フォントの XLFD 名の “Additional Style” フィールドにたいしてマッチされる。MS-Windows ではその spec にマッチするフォントにはその言語にたいして必要なコードページのサポートが要求される。現在のところこのプロパティでは ‘`ja`’、‘`ko`’、‘`zh`’という CJK 言語の小セットだけがサポートされる。

`:otf` 複雑なテキストレイアウトを必要とするスクリプトをサポートする GNU/Linux 上の ‘`libotf`’のようなライブラリーとともに Emacs がコンパイルされているような場合には、そのフォントはそれらの OpenType 機能をサポートする OpenType フォントでなければならない。値は以下の形式のリストでなければならない

```
(script-tag langsys-tag gsub gpos)
```

ここで *script-tag* は OpenType スクリプトタグシンボル、*langsys-tag* は OpenType 言語システムタグシンボル (`nil`ならデフォルト言語システムを使用)、`gsub` は OpenType GSUB 機能タグシンボル (何も要求されなければ `nil`)、`gpos`は OpenType GPOS 機能タグシンボルのリスト (何も要求されなければ `nil`)。`gsub`や `gpos`がリストなら、そのリスト内の `nil`要素は、そのフォントが残りすべてのタグシンボルにマッチしてはならないことを意味する。`gpos`は省略可。

`font-put` *font-spec property value* [Function]
フォント spec *font-spec*内のプロパティ *property*に *value*をセットする。

フォントエンティティーはオープンする必要がないフォントへの参照です。フォントオブジェクトとフォント spec の中間的な性質をもちフォント spec とは異なり、フォントオブジェクトと同じように単一かつ特定のフォントを参照します。フォントオブジェクトとは異なりフォントエンティティーの作成では、そのフォントのコンテンツはコンピューターへのメモリーにロードされません。Emacs はスケーラブルフォントを参照するために単一のフォントエンティティーから複数の異なるサイズのフォントオブジェクトをオープンするかもしれません。

`find-font` *font-spec* **&optional** *frame* [Function]
この関数はフレーム *frame*上のフォント spec *font-spec*にもっともマッチするフォントエンティティーをリターンする。*frame*が `nil`の場合のデフォルトは選択されたフレーム。

`list-fonts` *font-spec* **&optional** *frame num prefer* [Function]
この関数はフォント spec *font-spec*にマッチするすべてのフォントエンティティーのリストをリターンする。

オプション引数 *frame*が非 `nil`なら、そのフォントが表示されるフレームを指定する。オプション引数 *num*が非 `nil`なら、それはリターンされるリストの最大長を指定する整数だること。オプション引数 *prefer*が非 `nil`なら、それはリターンされるリスト順を制御するために使用する別のフォント spec であること。リターンされるフォント spec はそのフォント spec にもっとも近い降順にソートされて格納される。

`:font`属性の値としてフォント spec、フォントエンティティー、フォント名文字列を渡して `set-face-attribute`を呼び出すと、Emacs は表示に利用可能なもっともマッチするフォントをオープンします。そしてそのフェイスにたいする`:font`属性の実際の値として、対応するフォントオブジェクトを格納します。

以下の関数はフォントに関する情報を取得するために使用できます。これらの関数の *font*引数にはフォントオブジェクト、フォントエンティティー、またはフォント spec を指定できます。

`font-get` *font property* [Function]
この関数は *font*にたいするフォントプロパティ *property*の値をリターンする。

*font*がフォント spec であり、そのフォント spec が *property*を指定しなければリターン値は `nil`。*font*がフォントオブジェクトかフォントエンティティーなら、*:script*プロパティにたいする値はそのフォントがサポートするスクリプトのリストかもしれない。

`font-face-attributes` *font* **&optional** *frame* [Function]
この関数は *font*に対応するフェイス属性のリストをリターンする。オプション引数 *frame*はフォントが表示されるフレームを指定する。これが `nil`なら選択されたフレームが使用される。リターン値は以下の形式

```
(:family family :height height :weight weight
   :slant slant :width width)
```

ここで *family*、*height*、*weight*、*slant*、*width*の値はフェイス属性の値。*font*により指定されない場合には、いくつかのキー/属性ペアーはこのリストから省略されるかもしれない。

`font-xlfd-name` *font* **&optional** *fold-wildcards* [Function]
この関数は *font*にマッチする XLFD((X Logical Font Descriptor)) を文字列としてリターンする。XLFD に関する情報は Section "Fonts" in *The GNU Emacs Manual* を参照のこと。その名前が XLFD(最大 255 文字を含むことが可能) にたいして長すぎれば、この関数は `nil`をリターンする。

オプション引数 *fold-wildcards*が非 `nil`なら連続するワイルドカードは 1 つにまとめられる。

以下の 2 つの関数はフォントに関して重要な情報をリターンします。

`font-info` *name* **&optional** *frame* [Function]
この関数は *frame*で使用されるような文字列 *name*で指定されたフォントに関する情報をリターンする。*frame*が省略か `nil`の場合のデフォルトは選択されたフレーム。

この関数は `[`*`opened-name full-name size height baseline-offset relative-compose default-ascent max-width ascent descent space-width average-width filename capability`*`]`という形式のベクターによる値をリターンする。以下はこのベクターの各コンポーネントの意味:

opened-name
: フォントのオープンに使用された名前 (文字列)。

full-name
: フォントの完全名 (文字列)。

size
: フォントのピクセルサイズ。

height
: フォント高さ (ピクセル単位)。

baseline-offset
: ASCIIベースラインからのピクセル単位のオフセット (上方が正)。

relative-compose
default-ascent
: 文字の組み合わせ (compose) の方式を制御する数値。

max-width
: フォントの最大のアドバンス幅。

ascent
descent
: このフォントのアセント (ascent) とディセント (descent)。これら 2 つの数値の合計は上述の *height*と等しくなること。

space-width
: そのフォントのスペース文字の幅 (ピクセル単位)。

average-width
: そのフォントの文字の平均幅。これが 0 なら Emacs は表示のテキストレイアウト計算時にかわりに *space-width*の値を使用する。

filename
: フォントのファイル名 (文字列)。フォントのバックエンドがフォントのファイル名を見つける手段を提供しなければ `nil`もあり得る。

capability 最初の要素がフォントタイプを表す `x`、`opentype`、`truetype`、`type1`、`pcf`、`bdf` のいずれかのシンボルであるようなリスト。OpenType フォントでは、フォントによりサポートされる機能 GSUB と GPOS の 2 つの要素が含まれる。これらの要素はそれぞれ `((`*`script`* `(`*`langsys feature ...`*`) ...) ...)`という形式のリストであり、ここで *script*は OpenType の script タグを表すシンボル、*langsys* は OpenType の langsys タグを表すシンボル (またはデフォルトの langsys を表す `nil`)、そよび *feature*はそれぞれ OpenType の feature タグを表す。

`query-font` *font-object* [Function]

この関数は *font-object*に関する情報をリターンする (これは引数としてフォント名を文字列で受け取る `font-info`とは対照的)。

この関数は `[`*`name filename pixel-size max-width ascent descent space-width average-width capability`*`]`という形式のベクターで値をリターンする。以下はこのベクターの各要素の意味:

name フォント名 (文字列)。

filename フォントのファイル名 (文字列)。フォントのバックエンドがフォントのファイル名を見つける手段を提供しなければ `nil`もあり得る。

pixel-size フォントをオープンするために使用されたフォントのピクセルサイズ。

max-width
フォントの最大のアドバンス幅。

ascent
descent このフォントのアセント (ascent) とディセント (descent)。これら 2 つの数値の合計はフォントの高さを与える。

space-width
そのフォントのスペース文字の幅 (ピクセル単位)。

average-width
そのフォントの文字の平均幅。これが 0 なら Emacs は表示のテキストレイアウト計算時にかわりに *space-width*の値を使用する。

capability 最初の要素がフォントタイプを表す `x`、`opentype`、`truetype`、`type1`、`pcf`、`bdf` のいずれかのシンボルであるようなリスト。OpenType フォントでは、フォントによりサポートされる機能 GSUB と GPOS の 2 つの要素が含まれる。これらの要素はそれぞれ `((`*`script`* `(`*`langsys feature ...`*`) ...) ...)`という形式のリストであり、ここで *script*は OpenType の script タグを表すシンボル、*langsys* は OpenType の langsys タグを表すシンボル (またはデフォルトの langsys を表す `nil`)、そよび *feature*はそれぞれ OpenType の feature タグを表す。

以下の 4 つの関数はさまざまなフェイスにより使用されるフォントに関するサイズ情報をリターンして、Lisp プログラム内でのさまざまなレイアウトの検討を可能にします。これらの関数は問い合わせられたフェイスがリマップされていたら、リマップされたフェイスに関する情報をリターンすることによりフェイスのシマップを考慮します。Section 39.12.5 [Face Remapping], page 1021 を参照してください。

`default-font-width` [Function]

カレントバッファーの選択されたフレームにたいして定義されたデフォルトフェイスで使用されるフォントの平均幅をピクセル単位でリターンする。

`default-font-height` [Function]
この関数は選択されたフレームにたいして定義されたカレントバッファーのデフォルトフェイスで使用されるフォントの高さをピクセル単位でリターンする。

`window-font-width` **&optional** *window face* [Function]
この関数は *window*内の *face*で使用されるフォントの平均幅をピクセル単位でリターンする。*window*には生きたウィンドウを指定しなければならない。`nil`か省略なら *window*のデフォルトは選択されたウィンドウ、*face*のデフォルトは *window*内のデフォルトフェイス。

`window-font-height` **&optional** *window face* [Function]
この関数は *window*内の *face*で使用されるフォントの高さをピクセル単位でリターンする。*window*には生きたウィンドウを指定しなければならない。`nil`か省略なら *window*のデフォルトは選択されたウィンドウ、*face*のデフォルトは *window*内のデフォルトフェイス。

39.13 フリンジ

グラフィカルなディスプレイでは Emacs は各ウィンドウに隣接してフリンジ *(fringes)* を描画します。これは切り詰め (truncation)、継続 (continuation)、水平スクロールを示すビットマップを表示できる側面の細い垂直ストリップです。

39.13.1 フリンジのサイズと位置

以下のバッファーローカル変数はバッファーを表示するウィンドウのフリンジの位置と幅を制御します。

`fringes-outside-margins` [Variable]
フリンジは通常はディスプレイマージンとウィンドウテキストの間に表示される。この値が非 `nil`ならフリンジはディスプレイマージンの外側に表示される。Section 39.16.5 [Display Margins], page 1047 を参照のこと。

`left-fringe-width` [Variable]
この変数が非 `nil`なら、それは左フリンジの幅をピクセル単位で指定する。値 `nil`はそのウィンドウのフレームの左フリンジ幅を使用することを意味する。

`right-fringe-width` [Variable]
この変数が非 `nil`なら、それは右フリンジの幅をピクセル単位で指定する。値 `nil`はそのウィンドウのフレームの右フリンジ幅を使用することを意味する。

これらの変数にたいして値を指定しないすべてのバッファーは、フレームパラメーター `left-fringe`および `right-fringe`で指定された値を使用します (Section 29.4.3.4 [Layout Parameters], page 719 を参照)。

上記の変数はサブルーチンとして `set-window-fringes`を呼び出す関数 `set-window-buffer` (Section 28.11 [Buffers and Windows], page 635 を参照) を通じて実際に効果をもちます。これらの変数のいずれかを変更しても影響を受ける各ウィンドウで `set-window-buffer`を呼び出さなければ、そのバッファーを表示する既存のウィンドウのフリンジ表示は更新されません。個別のウィンドウでのフリンジ表示を制御するために `set-window-fringes`を使用することもできます。

`set-window-fringes` *window left* **&optional** *right outside-margins persistent* [Function]
この関数はウィンドウ *window*のフリンジ幅をセットする。*window*が `nil`なら選択されたウィンドウが使用される。

引数 *left*は左フリンジ、同様に *right*は右フリンジにたいしてピクセル単位で幅を指定する。いずれかにたいする値 `nil`はデフォルトの幅を意味する。*outside-margins*が非 `nil`ならフリンジをディスプレイマージンの外側に表示することを指定する。

期待する幅のフリンジを収容するのに *window*が十分大きくなければ、*window*のフリンジは未変更のままになる。

ここで指定した値は、*window*にたいして *keep-margins*を `nil`か省略した `set-window-buffer` (Section 28.11 [Buffers and Windows], page 635 を参照) の呼び出しによって後刻オーバーライドされるかもしれない。しかしオプションの 5 つ目の引数 *persistent*が非 `nil`、かつ他の引数が成功裏に処理されたら、ここで指定した値は後続の `set-window-buffer`呼び出しに引き継がれる。これはミニバッファーウィンドウでフリンジを永続的にオフにするために使用できる。例は `set-window-scroll-bars` (Section 39.14 [Scroll Bars], page 1039 を参照) の説明を参照のこと。

`window-fringes` **&optional** *window* [Function]
この関数はウィンドウ *window*のフリンジに関する情報をリターンする。*window*が省略か `nil`なら選択されたウィンドウが使用される。値は `(`*`left-width right-width outside-margins persistent`*`)`という形式。

39.13.2 フリンジのインジケーター

フリンジインジケーター *(Fringe indicators)* は行の切り詰めや継続、バッファー境界などを示すウィンドウフリンジ内に表示される小さいアイコンのことです。

`indicate-empty-lines` [User Option]
これが非 `nil`なら Emacs はグラフィカルなディスプレイ上で、バッファー終端にある空行それぞれにたいしてフリンジ内に特別なグリフを表示する。Section 39.13 [Fringes], page 1034 を参照のこと。この変数はすべてのバッファーにおいて自動的にバッファーローカルになる。

`indicate-buffer-boundaries` [User Option]
このバッファーローカル変数はウィンドウフリンジ内でバッファー境界とウィンドウのスクロールを示す方法を制御する。

Emacs はバッファー境界 (そのバッファーの最初の行と最後の行) がスクリーン上に表示された際には、それを三角アイコン (angle icon) で示すことができる。加えてスクリーンより上にテキストが存在すれば上矢印 (up-arrow)、スクリーンの下にテキストが存在すれば下矢印 (down-arrow) をフリンジ内に表示してそれを示すことができる。

基本的な値として 3 つの値がある:

`nil` これらのフリンジアイコンを何も表示しない。

`left` 左フリンジに三角アイコンと矢印を表示する。

`right` 右フリンジに三角アイコンと矢印を表示する。

その他の非 alist
左フリンジに三角アイコンを表示して矢印を表示しない。

値がそれ以外ならどのフリンジインジケーターをどこに表示するかを指定する alist であること。alist の各要素は `(`*`indicator . position`*`)`のような形式をもつ。ここで *indicator*は `top`、`bottom`、`up`、`down`、または `t` (指定されていないすべてのアイコンをカバーする) のいずれかであり *position*は `left`、`right`、または `nil`のいずれか。

たとえば `((top . left) (t . right))` は左フリンジに top angle ビットマップを、右フリンジに bottom angle ビットマップと両 arrow ビットマップを配置する。左フリンジに angle ビットマップを表示して arrow ビットマップを表示しないようにするには `((top . left) (bottom . left))` を使用する。

`fringe-indicator-alist` [Variable]

このバッファーローカル変数は論理的ロジカルフリンジインジケーターから、ウィンドウフリンジ内に実際に表示されるビットマップへのマッピングを指定する。値は (*indicator* . *bitmaps*) のような要素をもつ alist。ここで *indicator* は論理的インジケータータイプ、*bitmaps* はそのインジケーターに使用するフリンジビットマップを指定する。

indicator はそれぞれ以下のシンボルのいずれかであること:

`truncation`、`continuation`。
: 行の切り詰めと継続に使用される。

`up`、`down`、`top`、`bottom`、`top-bottom`
: `indicate-buffer-boundaries` が非 `nil` の際に使用される。`up` と `down` は、そのウィンドウ端より上と下にあるバッファー境界を示す。`top` と `bottom` はバッファーの最上端と最下端のテキスト行を示す。`top-bottom` はバッファー内にテキスト行 1 行だけが存在することを示す。

`empty-line`
: `indicate-empty-lines` が非 `nil` の際に、バッファーの後の空行を示すために使用される。

`overlay-arrow`
: オーバーレイ矢印に使用される (Section 39.13.6 [Overlay Arrow], page 1039 を参照)。

各 *bitmaps* の値にはシンボルのリスト (*left right* [*left1 right1*]) を指定できる。シンボル *left* と *right* は特定のインジケーターにたいして左および/または右フリンジに表示するビットマップを指定する。*left1* と *right1* はインジケーター `bottom` と `top-bottom` に固有であり、最後の改行をもたない最後のテキスト行を示すために使用される。かわりに *bitmaps* に左フリンジと右フリンジの両方で使用される単一のシンボルを指定することもできる。

標準のビットマップシンボルのリストと自身で定義する方法については Section 39.13.4 [Fringe Bitmaps], page 1037 を参照のこと。加えて `nil` は空ビットマップ (表示されないインジケーター) を表す。

`fringe-indicator-alist` がバッファーローカルな値をもち、論理的インジケーターにたいしてビットマップが定義されていないかビットマップが `t` ならば、`fringe-indicator-alist` のデフォルト値から対応する値が使用される。

39.13.3 フリンジのカーソル Frin

ある行がウィンドウと正確に同じ幅なとき、2 行を使用するかわりに Emacs は右フリンジ内にカーソルを表示します。フリンジ内のカーソルを表すために使用されるビットマップの違いはカレントバッファーのカーソルタイプに依存します。

`overflow-newline-into-fringe` [User Option]

これが非 `nil` なら、ウィンドウと正確に同じ幅の (最後の改行文字に継続されない) 行は継続されない。ポイントが行端に達した際には、カーソルはかわりに右フリンジに表示される。

`fringe-cursor-alist` [Variable]

この変数は論理的カーソルタイプから、右フリンジ内に実際に表示されるフリンジビットマップへのマッピングを指定する。値は各要素が (*cursor-type* . *bitmap*)のような形式をもつような alist。ここで *bitmap*は使用するフリンジビットマップ、*cursor-type*は表示するカーソルタイプ。

*cursor-type*はそれぞれ `box`、`hollow`、`bar`、`hbar`、`hollow-small`のいずれかであること。最初の 4 つはフレームパラメーター `cursor-type`の場合と同じ意味をもつ (Section 29.4.3.9 [Cursor Parameters], page 725 を参照)。`hollow-small`タイプは特定のディスプレイ行にたいして通常の `hollow-rectangle`が高すぎる際に `hollow`のかわりに使用される。

*bitmap*はそれぞれ、その論理的カーソルタイプにたいして表示されるフリンジビットマップを指定するシンボルであること。詳細は次のサブセクションを参照のこと。

`fringe-cursor-alist`がバッファーローカルな値をもち、カーソルタイプにたいして定義されたビットマップが存在しなければ、`fringes-indicator-alist`のデフォルト値の対応する値が使用される。

39.13.4 フリンジのビットマップ

フリンジビットマップ *(fringe bitmaps)* は行の切り詰めや継続、バッファー境界、オーバーレイ矢印等にたいする論理的フリンジインジケーターを表現する実際のビットマップです。それぞれのビットマップはシンボルにより表されます。これらのシンボルは前のサブセクションで説明した変数 `fringe-indicator-alist`と `fringe-cursor-alist`から参照されます。

Lisp プログラムも行内に出現する文字の 1 つに `display`プロパティを使用することにより、左フリンジまたは右フリンジ内にビットマップを直接表示することができます。そのような表示指定は以下の形式をもちます

(*fringe bitmap* [*face*])

*fringe*は `left-fringe`か `right-fringe`いずれかのシンボル、*bitmap*は表示するビットマップを識別するシンボルです。オプションの *face*はビットマップの表示でフォアグラウンカラーおよびバックグラウンドカラーを使用するフェイスを命名します (*face*が指定しないカラーにたいしては `fringe`フェイスの属性を使用する)。*face*を省略すると、`fringe`が指定しないカラーに `default`フェイスの属性を使用することを意味します。`default`フェイスや `fringe`フェイスの属性に依存しない予測可能な結果を得るために、*face*は省略せず常に特定のフェイスを与えることを推奨します。特にビットマップを `fringe`フェイスで表示したければ、*face*として `fringe`を使用してください。

たとえば左フリンジに矢印を表示するためには、`warning`フェイスを使用して以下のようにできます:

```
(overlay-put
 (make-overlay (point) (point))
 'before-string (propertize
                 "x" 'display
                 `(left-fringe right-arrow warning)))
```

以下は Emacs が定義する標準的なフリンジビットマップと、(`fringe-indicator-alist`と `fringe-cursor-alist`を通じて)Emacs 内で現在それらが使用される方法のリストです。

`left-arrow`、`right-arrow`
: 切り詰められた行を示すために使用される。

`left-curly-arrow`、`right-curly-arrow`
: 継続された行を示すために使用される。

`right-triangle`、`left-triangle`
: 前者はオーバーレイ矢印により使用され、後者は使用されない。

`up-arrow`、`down-arrow`
`bottom-left-angle`、`bottom-right-angle`
`top-left-angle`、`top-right-angle`
`left-bracket`、`right-bracket`
`empty-line`
: バッファー境界を示すために使用される。

`filled-rectangle`、`hollow-rectangle`
`filled-square`、`hollow-square`
`vertical-bar`、`horizontal-bar`
: フリンジカーソルの異なるタイプにたいして使用される。

`exclamation-mark`、`question-mark`
: Emacs の中核機能では使用されない。

次のサブセクションではフリンジビットマップを独自に定義する方法を説明します。

`fringe-bitmaps-at-pos` **&optional** *pos window* [Function]
: この関数はウィンドウ *window*内の位置 *pos*を含むディスプレイ行のフリンジビットマップをリターンする。リターン値は (*left right ov*)という形式をもつ。ここで *left*は左フリンジ内のフリンジビットマップにたいするシンボル (ビットマップなしなら `nil`)、*right*は同様に右フリンジにたいして、*ov*が非 `nil`なら左フリンジにオーバーレイ矢印が存在することを意味する。

  *window*内で *pos*が可視でなければ値は `nil`。*window*が `nil`なら選択されたウィンドウを意味する。*pos*が `nil`なら *window*内のポイントの値を意味する。

39.13.5 フリンジビットマップのカスタマイズ

`define-fringe-bitmap` *bitmap bits* **&optional** *height width align* [Function]
: この関数はシンボル *bitmap*を新たなフリンジビットマップとして定義、またはその名前の既存のビットマップを置き換える。

  引数 *bits*は使用するイメージを指定する。これは各要素 (整数) が対応するビットマップの 1 行を指定する文字列か整数ベクターであること。整数の各ビットはそのビットマップの 1 ピクセル、低位ビットはそのビットマップの最右ピクセルに対応する (このビットオーダーは XBM イメージと逆順であることに注意。Section 39.17.3 [XBM Images], page 1053 を参照)。

  高さは通常は *bits*の長さ。しかし非 `nil`の *height*により異なる高さを指定できる。幅は通常は 8 だが非 `nil`の *width*により異なる幅を指定できる。width は 1 から 16 の整数でなければならない。

  引数 *align*はそのビットマップが使用される行範囲に相対的なビットマップの位置を指定する。デフォルトはそのビットマップの中央。指定できる値は `top`、`center`、`bottom`。

  *align*引数にはリスト (*align periodic*)も指定できて、*align*は上述のように解釈される。*periodic*が非 `nil`なら、それは `bits`内の行が指定される高さに達するのに十分な回数繰り返されるべきであることを指定する。

`destroy-fringe-bitmap` *bitmap* [Function]
: この関数は *bitmap*により識別されるフリンジビットマップを破棄する。*bitmap*が標準フリンジビットマップを識別する場合には、それを完全に消去するかわりに実際にはそのビットマップの標準定義をリストアする。

`set-fringe-bitmap-face` *bitmap* **&optional** *face* [Function]
これはフリンジビットマップ *bitmap*にたいするフェイスに *face*をセットする。*face*が `nil`なら `fringe`フェイスを選択する。ビットマップのフェイスはそれを描画するカラーを制御する。

*face*は `fringe`にマージされるため *face*は通常はフォアグラウンドカラーだけを指定すること。

39.13.6 オーバーレイ矢印

オーバーレイ矢印 *(overlay arrow)* は、バッファー内の特定の行にたいしてユーザーに注意を促すために有用です。たとえばデバッガーでのインターフェースに使用されるモードでは、オーバーレイ矢印は実行されているコード行を示します。この機能はオーバーレイ *(overlays)* にたいして何も行いません (Section 39.9 [Overlays], page 999 を参照)。

`overlay-arrow-string` [Variable]
この変数は特定の行にたいして注意を喚起するために表示する文字列、または矢印機能が使用されていなければ `nil`を保持する。グラフィカルなディスプレイではこの文字列のコンテンツは無視され、かわりにフリンジ領域からディスプレイ領域左側にグリフが表示される。

`overlay-arrow-position` [Variable]
この変数はオーバーレイ矢印を表示する箇所を示すマーカーを保持する。これは行の先頭となるポイントであること。非グラフィカルなディスプレイではその行の先頭に矢印テキストが表示され、矢印テキストが表示されないときに表示されるべきテキストがオーバーレイされる。その矢印は通常は短く行は普通はインデントで開始されるので、上書きが問題となることは通常はない。

オーバーレイ矢印の文字列は、そのバッファーの `overlay-arrow-position`の値がバッファー内を指せば与えられた任意のバッファーで表示される。したがって `overlay-arrow-position`のバッファーローカルなバインディングを作成することにより、複数のオーバーレイ矢印の表示が可能である。しかしこれを達成するためには、`overlay-arrow-variable-list`を使用するほうが通常はより明快。

`before-string`プロパティをもつオーバーレイを作成することにより同様のことを行うことができます。Section 39.9.2 [Overlay Properties], page 1002 を参照してください。

変数 `overlay-arrow-variable-list`を通じて複数のオーバーレイ矢印を定義できます。

`overlay-arrow-variable-list` [Variable]
この変数の値は、それぞれがオーバーレイ矢印の位置を指定する変数のリスト。変数 `overlay-arrow-position`はこのリスト上にあるために通常の意味をもつ。

このリスト上の各変数は対応するオーバーレイ矢印位置に表示するためのオーバーレイ矢印文字列を指定する `overlay-arrow-string`プロパティ(テキスト端末用)、およびフリンジビットマップを指定する `overlay-arrow-bitmap`プロパティ(グラフィカル端末用) をもつことができます。これらのプロパティがセットされていなければデフォルトのフリンジインジケーター `overlay-arrow-string`と `overlay-arrow`が使用されます。

39.14 スクロールバー

フレームパラメーター `vertical-scroll-bars`はそのフレーム内のウィンドウが垂直スクロールバーをもつべきかと、それらが左か右のいずれかに配置されるべきかを通常は制御します。フレームパラメーター `scroll-bar-width`はそれらの幅を指定します (`nil`はデフォルトを意味する)。

フレームパラメーター `horizontal-scroll-bars`はフレーム内のウィンドウが水平スクロールバーをもつかどうかを制御します。フレームパラメーター `scroll-bar-height`はそれらの高さを指定します (`nil`はデフォルトを意味する)。Section 29.4.3.4 [Layout Parameters], page 719 を参照のしてください。

水平スクロールバーはすべてのプラットフォームで利用可能ではありません。引数を受け取らない関数 `horizontal-scroll-bars-available-p`は、システム上で水平スクロールバーが利用可能なら非 `nil`をリターンします。

以下の 3 つの関数は引数として生きたフレームを受け取り、デフォルトは選択されたフレームです。

`frame-current-scroll-bars` **&optional** *frame* [Function]
この関数はフレーム *frame*のスクロールバーのタイプを報告する。値はコンスセル (*vertical-type* . *horizontal-type*)。ここで *vertical-type*は `left`、`right`、または `nil` (スクロールバーなしを意味する) のいずれか。*horizontal-type*は `bottom`か `nil` (水平スクロールバーなしを意味する) のいずれか。

`frame-scroll-bar-width` **&optional** *frame* [Function]
この関数はウィンドウ *window*にたいして垂直スクロールバーの幅をピクセル単位でリターンする。

`frame-scroll-bar-height` **&optional** *frame* [Function]
この関数は *frame*の水平スクロールバーの高さをピクセル単位でリターンする。

以下の関数を使用することにより、特定のウィンドウにたいするフレーム固有のセッティングをオーバーライドできます:

`set-window-scroll-bars` *window* **&optional** *width vertical-type height horizontal-type persistent* [Function]
この関数はウィンドウ *window*のスクロールバーの幅および/または高さ、およびタイプをセットする。*window*が `nil`なら選択されたウィンドウが使用される。

*width*はピクセル単位で垂直スクロールバーの幅を指定する (`nil`はそのフレームにたいして指定された幅の使用を意味する)。*vertical-type*は垂直スクロールバーをもつかどうか、もつ場合にはその位置を指定する。可能な値は `left`、`right`、`t` (フレームのデフォルトの使用を意味する)、垂直スクロールバーなしなら `nil`のいずれか。

*height*はピクセル単位で水平スクロールバーの高さを指定する (`nil`はそのフレームにたいして指定された高さの使用を意味する)。*horizontal-type*は水平スクロールバーをもつかどうかを指定する。可能な値は `bottom`、`t` (フレームのデフォルトの使用を意味する)、水平スクロールバーなしなら `nil`のいずれか。ミニウィンドウにたいして値 `t`は `nil`、すなわち水平スクロールバーを表示しないことを意味することに注意。ミニウィンドウで水平スクロールバーを表示するためには、明示的に `bottom`を指定する必要がある。

期待するサイズのスクロールバーを収容するのに *window*が十分大きくなければ、*window*のスクロールバーは未変更のままになる。

ここで指定した値は *window*にたいする引数 *keep-margins*が `nil`か省略した `set-window-buffer` (Section 28.11 [Buffers and Windows], page 635 を参照) の呼び出しにより後からオーバーライドされるかもしれない。しかしオプションの 1 つ目の引数 *persistent*が非 `nil`、かつ他の引数が成功裏に処理されたなら、ここで指定された値は後続の `set-window-buffer`呼び出しに引き継がれる。

`set-window-scroll-bars`および `set-window-fringes` (Section 39.13.1 [Fringe Size/Pos], page 1034 を参照) の *persistent*引数を使用すれば、以下のスニペットを早期 init ファイル (early init file) に追加することにより、すべてのミニバッファーウィンドウでスクロールバーおよび/またはフリンジを信頼性をもって永続的にオフにすることができます (Section 40.1.2 [Init File], page 1093 を参照)。

```
(add-hook 'after-make-frame-functions
          (lambda (frame)
            (set-window-scroll-bars
             (minibuffer-window frame) 0 nil 0 nil t)
            (set-window-fringes
             (minibuffer-window frame) 0 0 nil t)))
```

以下の 4 つの関数は引数として生きたウィンドウを受け取り、デフォルトは選択されたウィンドウです。

`window-scroll-bars` **&optional** *window* [Function]
この関数は `(`*`width columns vertical-type height lines horizontal-type persistent`*`)`という形式のリストをリターンする。

値 *width*は垂直スクロールバーの幅に指定された値 (`nil`もあり得る)。*columns*は垂直スクロールバーが実際に占有する列数 (丸められているかもしれない)。

値 *height*は水平スクロールバーの高さに指定された値 (`nil`もあり得る)。*lines*は水平スクロールバーが実際に占有する行数 (丸められているかもしれない)。

*persistent*の値は最後に成功裏に `set-window-scroll-bars`を呼び出した際に *window*に指定した値、そのような呼び出しがなければ `nil`。

`window-current-scroll-bars` **&optional** *window* [Function]
この関数はウィンドウ *window*にたいするスクロールバータイプを報告する。値はコンスセル (*vertical-type* . *horizontal-type*)。`window-scroll-bars`とは異なりフレームのデフォルトと `scroll-bar-mode`を考慮して実際に使用されているスクロールバータイプを報告する。

`window-scroll-bar-width` **&optional** *window* [Function]
この関数は *window*の垂直スクロールバーの幅をピクセル単位でリターンする。

`window-scroll-bar-height` **&optional** *window* [Function]
この関数は、*window*の水平スクロールバーの高さをピクセル単位でリターンする。

`set-window-scroll-bars`でウィンドウのスクロールバーのセッティングを指定しない場合には、表示されようとするバッファーのバッファーローカル変数 `vertical-scroll-bar`、`horizontal-scroll-bar`、`scroll-bar-width`、`scroll-bar-height`がウィンドウのスクロールバーを制御します。`set-window-buffer`はこれらの変数を調べる関数です。あるウィンドウですでに可視なバッファーでこれらを変更した場合には、すでに表示されているのと同じバッファーを指定して `set-window-buffer`を呼び出すことにより、そのウィンドウに新たな値を記録させることができます。

以下の変数をセット (セットにより自動的にバッファーローカルになる) することにより、特定のバッファーのスクロールバーの外観を制御できます。

`vertical-scroll-bar` [Variable]
この変数は垂直スクロールバーの配置を指定する。可能な値は `left`、`right`、そのフレームのデフォルトの使用を意味する `t`、スクロールバーなしの `nil`のいずれか。

`horizontal-scroll-bar` [Variable]
この変数は水平スクロールバーの配置を指定する。可能な値は `bottom`、そのフレームのデフォルトの使用を意味する `t`、スクロールバーなしの `nil`のいずれか。

`scroll-bar-width` [Variable]
この変数はそのバッファーの垂直スクロールバーをピクセル単位で量った幅を指定する。値 `nil` はフレームにより指定された値の使用を意味する。

`scroll-bar-height` [Variable]
この変数はそのバッファーの水平スクロールバーをピクセル単位で量った高さを指定する。値 `nil`はフレームにより指定された値の使用を意味する。

最後に変数 `scroll-bar-mode`と `horizontal-scroll-bar-mode`をカスタマイズすることにより、すべてのフレームでのスクロールバーの表示を切り替えることができます。

`scroll-bar-mode` [User Option]
この変数はすべてのフレームに垂直スクロールバーを配置するべきかと、その場所を制御する。可能な値は、スクロールバーなしの `nil`、左にスクロールバーを配置する `left`、右にスクロールバーを配置する `right`のいずれか。

`horizontal-scroll-bar-mode` [User Option]
この変数はすべてのフレームに水平スクロールバーを表示するかどうかを制御する。

39.15 ウィンドウディバイダー

ウィンドウディバイダーとはフレームのウィンドウ間に描画されるバーのことです。右 (right) ディバイダーはあるウィンドウと、その右に隣接する任意のウィンドウの間に描画されます。その幅 (厚さ) はフレームパラメーター `right-divider-width`で指定されます。下 (bottom) ディバイダーはあるウィンドウと、その下に隣接するウィンドウやエコーエリアとの間に描画されます。その幅はフレームパラメーター `bottom-divider-width`で指定されます。いずれの場合でも幅に 0 を指定すると、そのようなディバイダーを描画しないことを意味します。Section 29.4.3.4 [Layout Parameters], page 719 を参照してください。

技術的には右ディバイダーはそれの左にあるウィンドウに所属して、その幅がそのウィンドウのトータル幅に寄与することを意味します。下ディバイダーは上にあるウィンドウに所属して、その幅がそのウィンドウのトータル高さに寄与することを意味します。Section 28.3 [Window Sizes], page 613 を参照してください。あるウィンドウが右ディバイダーと左ディバイダーの両方をもつ場合には下ディバイダーが優勢になります。これは右ディバイダーが下ディバイダーの上で終端されるのに比べて、下ディバイダーはそのウィンドウの完全なトータル幅で描画されることを意味しています。

ディバイダーはマウスでドラッグできるのでマウスで隣接するウィンドウのサイズを調整するために有用です。これらはスクロールバーやモードラインが表示されていないときに隣接するウィンドウを視覚的に分離する役目もあります。以下の 3 つのフェイスによりディバイダーの外観をカスタマイズできます:

`window-divider`
ディバイダーの幅が 3 ピクセル未満のときは、このフェイスのフォアグラウンドカラーで塗りつぶしで描画される。これより広いディバイダーでは、最初と最後のピクセルを除いた内部にたいしてのみこのフェイスが使用される。

`window-divider-first-pixel`
: これは少なくとも幅が 3 ピクセルあるディバイダーの最初のピクセルを描画するために使用される。塗りつぶし (solid) の外観を得るためには `window-divider`フェイスに使用されるのと同じ値をセットすること。

`window-divider-last-pixel`
: これは少なくとも幅が 3 ピクセルあるディバイダーの最後のピクセルを描画するために使用される。塗りつぶし (solid) の外観を得るためには `window-divider`フェイスに使用されるのと同じ値をセットすること。

以下の 2 つの関数により特定のウィンドウのディバイダーのサイズを取得できます。

`window-right-divider-width` **&optional** *window* [Function]
: *window*の右ディバイダーの幅 (厚さ) をピクセル単位でリターンする。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。最右ウィンドウにたいするリターン値は常に 0。

`window-bottom-divider-width` **&optional** *window* [Function]
: *window*の下ディバイダーの幅 (厚さ) をピクセル単位でリターンする。*window*は生きたウィンドウでなければならずデフォルトは選択されたウィンドウ。ミニバッファーウィンドウやミニバッファーがないフレームの最下ウィンドウにたいするリターン値は常に 0。

39.16 `display`プロパティ

テキストプロパティ (またはオーバーレイプロパティ) の `display`はテキストへのイメージ挿入、およびテキスト表示のその他の事相を制御します。`display`プロパティの値はディスプレイ仕様、または複数のディスプレイ仕様を含むリストかベクターであるべきです。同じ `display`プロパティ値内のディスプレイ仕様は、一般的にはそれらがカバーするテキストにたいして並行して適用されます。

複数のソース (オーバーレイおよび/またはテキストプロパティ) が `display`プロパティにたいして値を指定しますが 1 つの値だけが効果をもち、それは `get-char-property`のルールにしたがいます。Section 32.19.1 [Examining Properties], page 813 を参照してください。

いくつかのディスプレイ仕様には、表示時に評価される Lisp フォームを含めることができます。これは特定の状況では安全ではないかもしれません (ディスプレイ仕様が何らかの外部のプログラムやエージェントにより生成されたとき等)。`('disable-eval` *spec*`)` のように特別なシンボル `disable-eval` で始まるリスト内にディスプレイ仕様をラップすることにより、他のすべてのディスプレイプロパティ機能をサポートしつつ、*spec*内の任意の Lisp 評価が無効になります。

このセクションの残りの部分では、複数の種類のディスプレイ仕様とそれらの意味を説明します。

39.16.1 テキストを置換するディスプレー仕様

ある種のディスプレイ仕様は、そのプロパティをもつテキストのかわりに表示する何かを指定します。これらは置換 *(replacing)* ディスプレイ仕様と呼ばれます。Emacs はユーザーにたいして、この方法で置換されたバッファーテキストの中間への対話的なポイント移動を許可しません。

ディスプレイ仕様のリストに 1 つ以上の置換ディスプレイ仕様が含まれる場合には、最初の置換ディスプレイ仕様が残りをオーバーライドします。置換ディスプレイ仕様は他のほとんどのディスプレイ仕様は置換を許容しないので、それらとは無関係です。

置換ディスプレイ仕様では、そのプロパティをもつテキストとは、`display`プロパティとして同一の Lisp オブジェクトをもつ連続したすべての文字を意味します。これらの文字は単一の単位として置

換されます。displayプロパティに異なる Lisp オブジェクト (eqではないオブジェクト) をもつ 2 つの文字は個別に処理されます。

以下はこの要点を示すための例です。文字列が置換ディスプレイ仕様としての役割をもち、指定された文字列のプロパティをもつテキストを置換します (Section 39.16.4 [Other Display Specs], page 1046 を参照)。以下の関数を考えてみてください:

```
(defun foo ()
  (dotimes (i 5)
    (let ((string (concat "A"))
          (start (+ i i (point-min))))
      (put-text-property start (1+ start) 'display string)
      (put-text-property start (+ 2 start) 'display string))))
```

この関数はバッファー内の最初の 10 文字それぞれにたいして文字列"A"であるような displayプロパティを与えますが、これらはすべて同じ文字列オブジェクトを取得しません。最初の 2 文字は同じ文字列オブジェクトなので 1 つの 'A'に置換されます。2 つの別々の put-text-property呼び出しでそのディスプレイプロパティが割り当てられたという事実は無関係です。同様に次の 2 文字は 2 つ目の文字列 (concatにより新たに作成された文字列オブジェクト) を取得するので 1 つの 'A'で置換されて、... となります。したがって 10 文字は 5 つの A で表示されます。

39.16.2 スペースの指定

指定された幅および/または高さのスペースを表示するためには (space . *props*)という形式のディスプレイ仕様を使用します。このプロパティを 1 つ以上の連続する文字に put することができます。これらすべての文字のかわりに指定された高さと幅のスペースが表示されます。以下はスペースのウェイトを指定するために *props*内で使用できるプロパティです:

`:width` *width*
: *width*が数字なら、それはスペースの幅が通常の文字幅の *width*倍であるべきかを指定する。*width*はピクセル幅 *(pixel width)* 仕様でも可 (Section 39.16.3 [Pixel Specification], page 1045 を参照)。

`:relative-width` *factor*
: 幅の広さは同じ displayプロパティをもつ連続する文字のグループ内の最初の文字から計算される必要があることを指定する。スペースの幅はその文字のピクセル幅に *factor* を乗じた幅である (テキストモード端末では文字の "ピクセル幅" は通常は 1 だが TAB 文字や 2 倍の幅をもつ CJK 文字では 1 以上になり得る)。

`:align-to` *hpos*
: スペースが *hpos*に達するほど十分に広くあるべきことを指定する。*hpos*が数字なら通常の文字幅の単位で量られる。*hpos*はピクセル幅 *(pixel width)* 仕様でも可 (Section 39.16.3 [Pixel Specification], page 1045 を参照)。

上記プロパティのいずれか 1 つだけを使用するべきです。以下のプロパティでスペースの高さも指定できます:

`:height` *height*
: スペースの高さを指定する。*height*が数字ならスペースの高さが通常の文字高さの *height* 倍であるべきことを指定する。*height*はピクセル高さ仕様 *(pixel height)* でも可 (Section 39.16.3 [Pixel Specification], page 1045 を参照)。

`:relative-height` *factor*
: このディスプレイ仕様をもつテキストの通常の高さに *factor*を乗じることによりスペースの高さを指定する。

`:ascent` *ascent*

> *ascent*の値が非負の 100 以下の数字ならスペースの高さの *ascent*パーセントをスペースのアセント (ascent: 上方)、すなわちベースラインより上の部分とみなす。ピクセルアセント *(pixel ascent)* 仕様によりアセントをピクセル単位で指定することも可 (Section 39.16.3 [Pixel Specification], page 1045 を参照)。

`:height`と`:relative-height`を両方同時に使用しないでください。

`:width`と`:align-to`プロパティは非グラフィック端末でサポートされますが、このセクションのその他のスペースプロパティはサポートされません。

スペースプロパティは双方向テキスト表示の並べ替えのためのパラグラフ区切りとして扱われます。詳細は Section 39.26 [Bidirectional Display], page 1085 を参照してください。

39.16.3 スペースにたいするピクセル指定

プロパティ`:width`、`:align-to`、`:height`、`:ascent`の値は再表示の間に評価される特別な種類の式です。その評価の結果はピクセルの絶対数として使用されます。

以下の式がサポートされています:

```
expr ::= num | (num) | unit | elem | pos | image | xwidget | form
num  ::= integer | float | symbol
unit ::= in | mm | cm | width | height
elem ::= left-fringe | right-fringe | left-margin | right-margin
       | scroll-bar | text
pos  ::= left | center | right
form ::= (num . expr) | (op expr ...)
op   ::= + | -
```

フォーム *num*はデフォルトフレームフォントの高さか幅、フォーム `(num)`は絶対ピクセル数を指定します。*num*がシンボル *symbol*なら、それのバッファーローカルな変数バインディングが使用されます。このバインディングには数字か上述の形式のコンスセル (他にもバッファーローカルなバインディングをもつシンボルが `car`であるような他のコンスセルも含む) が可能です。

単位 `in`、`mm`、`cm`はそれぞれインチ、ミリメートル、センチメートルごとのピクセル数を指定します。単位 `width`と `height`はそれぞれカレントフェイスのデフォルトの幅と高さに対応します。`(image . props)` という形式のイメージ仕様は、指定されたイメージの幅や高さに対応します (Section 39.17.2 [Image Descriptors], page 1049 を参照)。同様に `(xwidget . props)` という形式の xwidget 仕様は指定された xwidget び幅や高さを意味します。Section 39.18 [Xwidgets], page 1066 を参照してください。

要素 `left-fringe`、`right-fringe`、`left-margin`、`right-margin`、`scroll-bar`、`text`はそのウィンドウの対応する領域の幅を指定します。そのウィンドウで行番号 (Section 39.10 [Size of Displayed Text], page 1007 を参照) を表示している際にはテキストエリアの幅は行番号の表示に要するスクリーンスペースで減じられます。

位置 `left`、`center`、`right`はテキストエリアの左端、中央、右端から相対的に位置を指定するために`:align-to`とともに使用できます。ウィンドウで行番号を表示している際の `left`と `center`の位置は、行番号の表示に要するスクリーンスペースを考慮したオフセットになります。

(`text`を除いた) 上記ウィンドウ要素は与えられたエリアの左端から相対的に位置を指定するために`:align-to`とともに使用することもできます。(最初に出現するこれらシンボルのいずれかにより) 相対的位置にたいするベースオフセットが一度セットがされると、残りのシンボルは指定されたエリアの幅として解釈されます。たとえば左マージンの中央に位置揃えするには以下のようにします

```
:align-to (+ left-margin (0.5 . left-margin))
```

位置揃えにたいしてベースオフセットが何も指定されなければ、テキストエリア左端にたいして常に相対的になります。たとえばヘッダーライン内の '`:align-to 0`'はテキストエリアの最初のテキスト行に位置揃えします。ウィンドウで行番号を表示している際には、テキストは行番号表示の終了に使用されるスペースから開始するとみなされます。

(*num* . *expr*)という形式の値は、*num*と *expr*により生成される値を意味します。たとえば (2 . in)は 2 インチの幅、(0.5 . *image*)は指定された *image* (それのイメージ spec により与えられる必要がある) の幅 (または高さ) の半分を指定します。

フォーム (+ *expr* ...)は式の値を合計します。フォーム (- *expr* ...)は式の値を符号反転または減算します。

39.16.4 その他のディスプレー仕様

以下は `display`テキストプロパティ内で使用できる他のディスプレイ仕様です。

string このプロパティをもつテキストのかわりに *string*を表示する。

再帰的なディスプレイ仕様はサポートされない。つまり *string*の `display`プロパティがあっても使用されない。

(image . *image-props*)
この種のディスプレイ仕様はイメージディスクリプタである (Section 39.17.2 [Image Descriptors], page 1049 を参照)。ディスプレイ仕様として使用時には、そのディスプレイ仕様をもつテキストのかわりに表示するイメージを意味する。

(slice *x y width height*)
この仕様は `image`とともに、表示するイメージのスライス *(slice:* イメージの特定の領域*)* を指定する。要素 *y*と *x*はイメージ内での左上隅、*width*と *height*はそのスライスの幅と高さを指定する。整数はピクセル数、0.0 から 1.0 までの浮動小数点数はイメージ全体の幅や高さの割合を意味する。

((margin nil) *string*)
この形式のディスプレイ仕様は、このディスプレイ仕様をもつテキストのかわりにテキストと同じ位置に表示する *string*を意味する。これは単に *string*を使用するのと同じだが、マージン表示 (Section 39.16.5 [Display Margins], page 1047 を参照) の特殊なケースとして行われる点が異なる。

(left-fringe *bitmap* [*face*])
(right-fringe *bitmap* [*face*])
テキスト行の任意の文字がこのディスプレイ仕様をもつ場合には、その文字のかわりにその行の左や右のフリンジに表示する *bitmap*を指定する。オプションの *face*はビットマップ表示にカラーを使用するフェイスを指定する。詳細は Section 39.13.4 [Fringe Bitmaps], page 1037 を参照のこと。

(space-width *factor*)
このディスプレイ仕様は、この仕様をもつテキスト内のすべてのスペース文字に効果を及ぼす。これらすべてのスペースは通常の幅の *factor*倍の幅で表示される。要素 *factor*は整数か浮動小数点数であること。スペース以外の文字は影響を受けない。特にこれはタブ文字に影響を与えない。

(height *height*)
このディスプレイ仕様はテキストを高く (taller)、または低く (shorter) する。*height*には以下を指定できる:

`(+ n)` これは *n*ステップ大きいフォントの使用を意味する。ステップは利用可能なフォントのセットから定義される。利用可能なフォントとは、具体的には、このような場合でなければ、height を除いてそのテキストに指定されたすべての属性にマッチするフォント。適切なフォントの各サイズは別のステップとして利用可能とみなされる。*n*は整数であること。

`(- n)` これは *n*ステップ小さいフォントの使用を意味する。

factor (数値)
: 数値 *factor*はデフォルトフォントの *factor*倍高いフォントの使用を意味する。

function (シンボル)
: 高さを計算する関数。この関数はカレントの高さを引数として呼び出されて、使用する新たな高さをリターンすること。

form (上記以外)
: *height*の値が上記のいずれにもマッチしなければ、それはフォームである。Emacs は `height`をカレントで指定されたフォントの高さにバインドして新たな高さを取得するためにフォームを評価する。

`(raise` *factor*`)`
: この種のディスプレイ仕様は、その行のベースラインに相対的にテキストを上 (raise)、または下 (lower) に指定する。これは主に上付き文字と下付き文字を意図している。

*factor*は影響を受けるテキストの高さにたいする乗数として解釈される数値でなければならない。これが正なら上、負なら下に文字を表示することを意味する。

テキストが `raise`より前 (左) に指定された `height`ディスプレイ仕様をもつ場合には、テキストが上下される量はテキストの高さにもとづくので、上下されるピクセル数には `raise`が効果をもつ。したがって通常のテキスト高さより小さい上付きや下付きで表示したければ、`height`の前に `raise`を指定することを考慮するべきである。

任意のディスプレイ仕様にたいして条件を作成できます。これを行うには、`(when` *condition* `.` *spec*`)`という形式の別リスト内にパッケージします。この場合には、仕様 *spec*は *condition*が非 `nil`値に評価されたときだけ適用されます。この評価の間に `object`は条件つき `display`プロパティをもつ文字列、またはバッファーにバインドされます。`position`と `buffer-position`はそれぞれ `object`内の位置、および `display`プロパティが見つかったバッファー位置にバインドされます。`object`が文字列の際には両者の位置は異なるかもしれません。

*condition*はこのディスプレイ仕様が配置された箇所のテキストを再表示が調べる際だけ評価されるので、この機能は比較的安定した条件にもっとも適していることに注意してください (特定のバッファー位置それぞれにたいして評価ごとに同じ結果を取得する)。同じのテキスト位置にたいする結果が異なる、つまり結果がポイント位置に依存するようなら、この条件仕様はあなたの望むものではないかもしれません。なぜなら再表示は最後の表示サイクル以降に何かが変更されたといなす理由があるバッファーテキストの部分だけを調べるからです。

39.16.5 マージン内への表示

バッファーはその左側と右側にディスプレイマージン *(display margins)* と呼ばれるブランクエリアをもつことができます。それらのエリア内には通常はテキストが出現することはありませんが、`display`プロパティを使用してディスプレイマージン内に何かを配置することができます。現在のところマージン内のテキストやイメージをマウスセンシティブにする方法はありません。

マージン内に何かを表示するにはテキストの `display`プロパティのマージン表示仕様 (margin display specification) で指定します。これは配置したテキストが表示されないことを意味する置換表示仕様です。マージン表示は表示されますがそのテキストは表示されません。

マージン表示仕様とは (`(margin right-margin)` *spec*)や (`(margin left-margin)` *spec*) のようなものです。ここで *spec*はマージン内に何を表示するかを告げる別の表示仕様です。典型的にはこれは表示するテキスト文字列やイメージディスクリプタです。

特定のバッファーテキストに割り当てられたマージンに何かを表示するためには、そのテキストに `before-string`プロパティを付してコンテンツとしてマージン表示仕様を put します。

マージン内に表示する文字列がフェイスを指定しなければ、テキストエリア内に表示される文字列に準じたいくつかの規則と優先度によりフェイスが決定されることに注意してください (Section 39.12.4 [Displaying Faces], page 1021 を参照)。これが望ましくないマージンへのフェイスの “漏洩” をもたらすようなら、文字列が文字列用に明示的なフェイスを確実にもつようにしてください。

ディスプレイマージンが何かを表示可能になる前に、それらに非 0 の幅を与えなければなりません。これを行う通常の方法は以下の変数をセットする方法です:

`left-margin-width` [Variable]
: この変数は左マージンの幅を文字セル (別名は “列”) 単位で指定する。これ、すべてのバッファーでバッファーローカルである。値 `nil`は左マージンエリアなしを意味する。

`right-margin-width` [Variable]
: この変数は右マージンの幅を文字セル単位で指定する。これはすべてのバッファーでバッファーローカルである。値 `nil`は右マージンエリアなしを意味する。

これらの変数をセットしてもウィンドウには即座には反映されません。これらの変数はウィンドウ内に新たなバッファーを表示する際にチェックされます。したがって `set-window-buffer`を呼び出すことにより変更を反映することができます。これらの変数を左右のマージンのカレント幅の判定を試みるために使用してはいけません。かわりに関数 `window-margins`を使用してください。

マージン幅を即座にセットすることもできます。

`set-window-margins` *window left* **&optional** *right* [Function]
: この関数はウィンドウ *window*のマージン幅、文字セル単位で指定する。引数 *left*は左マージン、*right*は右マージン (デフォルトは `0`) を制御する。

  期待する幅のマージンを収容するのに *window*が十分大きくなければ、*window*のマージンは未変更のままになる。

  ここで指定した値は *window*にたいする引数 *keep-margins*が `nil`か省略した `set-window-buffer` (Section 28.11 [Buffers and Windows], page 635 を参照) の呼び出しにより後からオーバーライドされるかもしれない。

`window-margins` **&optional** *window* [Function]
: この関数は *window*の左マージンと右マージンの幅を (*`left . right`*) という形式のコンスセルでリターンする。2 つのマージンエリアのいずれか一方が存在しなければ幅は `nil`でリターンされる。2 つのマージンがどちらも存在しなければ、この関数は `(nil)`をリターンする。*window*が `nil`なら選択されたウィンドウが使用される。

39.17 イメージ

Emacs バッファー内にイメージを表示するためには最初にイメージディスクリプタを作成して、それを表示されるテキストの `display`プロパティ(Section 39.16 [Display Property], page 1043 を参照) 内のディスプレイ指定子として使用しなければなりません。

Emacs はグラフィカルな端末で実行時には、通常はイメージの表示が可能です。テキスト端末、イメージサポートを欠く特定のグラフィカル端末、またはイメージサポートなしでコンパイルされた Emacs ではイメージを表示できません。原則的にイメージが表示可能か判断するためには関数 `display-images-p`を使用できます (Section 29.25 [Display Feature Testing], page 751 を参照)。

39.17.1 イメージのフォーマット

Emacs はいくつかの異なるフォーマットのイメージを表示できます。これらのイメージフォーマットのいくつかは、特定のサポートライブラリーがインストールされている場合のみサポートされます。いくつかのプラットフォームでは Emacs はオンデマンドでサポートライブラリーをロードできます。そのような場合には、それらの動的ライブラリーにたいする既知の名前セットを変更するために変数 `dynamic-library-alist`を使用できます。Section 40.21 [Dynamic Libraries], page 1130 を参照してください。

サポートされるイメージフォーマット (と要求されるサポートライブラリー) には PBM と XBM(サポートライブラリーに依存せず常に利用可能)、XPM (`libXpm`)、GIF (`libgif`か`libungif`)、JPEG (`libjpeg`)、TIFF (`libtiff`)、PNG (`libpng`)、SVG (`librsvg`) が含まれます。

これらのイメージフォーマットはそれぞれイメージタイプシンボル *(image type symbol)* に関連付けられます。上記のフォーマットにたいするシンボルは順に `pbm`、`xbm`、`xpm`、`gif`、`jpeg`、`tiff`、`png`、`svg`になります。

さらに ImageMagick(`libMagickWand`) のサポートつきで Emacs をビルドした場合には、Emacs は ImageMagick が表示可能なイメージフォーマットを表示できます。Section 39.17.5 [ImageMagick Images], page 1054 を参照してください。ImageMagick を通じて表示されるすべてのイメージはタイプシンボル `imagemagick`をもちます。

`image-types` [Variable]

この変数はカレント構成で潜在的にサポートされるイメージフォーマットにたいするタイプシンボルのリストを含む。

“潜在的” とは Emacs がそのイメージタイプを知っていることを意味しており、実際に使用可能である必要はない (たとえば動的ライブラリーが利用できないせいかもしれない)。どのイメージタイプが実際に利用できるか知るためには `image-type-available-p`を使用すること。

`image-type-available-p` *type* [Function]

この関数はタイプ *type*のイメージのロードと表示が可能なら非 `nil`をリターンする。*type*はイメージタイプシンボルであること。

サポートライブラリーが静的にリンクされたイメージタイプにたいして、この関数は常に `t`をリターンする。サポートライブラリーが動的にロードされるイメージタイプにたいしてはライブラリーがロード可能なら `t`、それ以外なら `nil`をリターンする。

39.17.2 イメージのディスクリプタ

イメージディスクリプタ *(image descriptor)* とは、イメージにたいする基礎的なデータと表示する方法を指定するリストです。これは通常はオーバーレイプロパティかテキストプロパティ `display`(Section 39.16.4 [Other Display Specs], page 1046 を参照) の値を通じて使用されま

すが、バッファーにイメージを挿入する便利なヘルパー関数については Section 39.17.9 [Showing Images], page 1063 を参照してください。

イメージディスクリプタはそれぞれ `(image . props)` という形式をもちます。ここで *props* はキーワードシンボルと値のペアーからなるプロパティリストであり、少なくともそのイメージタイプを指定するペアー `:type type` を含みます。

以下はすべてのイメージタイプにたいして意味のあるプロパティのリストです (以降のサブセクションで説明するように特定のイメージタイプにたいしてのみ意味があるプロパティも存在する):

`:type` *type*
: イメージタイプ。すべてのイメージディスクリプタは。このプロパティを含まなければならない。

`:file` *file*
: これはファイル *file* からイメージをロードすることを意味する。*file* が絶対ファイル名でなければ、それは `data-directory` の `images` サブディレクトリーから相対的に展開される。これに失敗すると `x-bitmap-file-path` にリストされたディレクトリーから相対的に展開される (Section 39.12.1 [Face Attributes], page 1012 を参照)。

`:data` *data*
: これは raw イメージデータを指定する。すべてのイメージディスクリプタは `:data` か `:file` のいずれかをもたなければならないが両方もつことはできない。

  ほとんどのイメージタイプにたいして、`:data` プロパティの値はイメージデータを含む文字列であること。いくつかのイメージタイプは `:data` をサポートしない。それ以外のイメージタイプにたいしては `:data` 単独では不十分であり、`:data` とともに他のイメージプロパティを使用する必要がある。詳細は以下のサブセクションを参照のこと。

`:margin` *margin*
: これはイメージ周囲に余分なマージンとして何ピクセル追加するかを指定する。値 *margin* は非負の数値か、そのような数値のペアー `(x . y)` でなければならない。ペアーなら *x* は水平方向に追加するピクセル数、*y* は垂直方向に追加するピクセル数を指定する。`:margin` が指定されない場合のデフォルトは 0。

`:ascent` *ascent*
: これはイメージのアセント (ベースラインの上の部分) に使用するイメージの高さの分量を指定する。値 *ascent* は 0 から 100 の数値かシンボル `center` でなければならない。

  ascent が数値ならアセントに使用するイメージの高さのパーセンテージであること。

  ascent が `center` なら、イメージにたいしてテキストプロパティやオーバーレイプロパティにより指定される方法で、センターライン (そのイメージ位置にテキストを描画する際の垂直方向のセンターライン) の垂直方向中心にイメージが配置される。

  このプロパティが省略された場合のデフォルトは 50。

`:relief` *relief*
: これはイメージ周辺にシャドー矩形を追加する。値 *relief* はシャドーライン幅をピクセルで指定する。*relief* が負ならボタンを押下した状態、それ以外はボタンを押下していない状態のイメージでシャドーを描画する。

`:width` *width*, `:height` *height*
: キーワード `:width` と `:height` はイメージのスケーリングに使用される。いずれか一方のみが指定された場合には、アスペクト比を保つためにもう一方が算出される。両方が指定された場合にはアスペクト比は保たれないかもしれない。

`:max-width` *max-width*, `:max-height` *max-height*
: キーワード`:max-width`と`:max-height`は、イメージのサイズがこれらの値を超過した場合のスケーリングに使用される。`:width`がセットされた場合には`max-width`より優先されて、`:height`がセットされた場合には`max-height`より優先されるだろうが、それ以外ではこれらのキーワードを望むように混交できる。

  `:max-width`と`:height`が指定されていて`:width`が未指定なら、アスペクト比を維持することにより`:max-width`を超える幅が要求されるかもしれない。これが発生した場合には、スケーリングは`:max-width`を超過しないアスペクト比を維持できるように、小さい値を使用する。`:max-height`と`:width`が指定されていて`:height`が未指定のときも同様。たとえば 200x100 のイメージがあり`:width`を 400、`:max-height`を 150 に指定すると、アスペクト比を保持しつつ “max” のセッティングを超過しないようにイメージは最終的には 300x150 になる。このパラメーターの組み合わせは、“可能なかぎり大きく、ただし利用可能なディスプレイエリア以下でこのイメージを表示せよ” のように指示する簡便な手段である。

`:scale` *scale*
: これは数字であること。1 より大きい値は幅および高さを乗じたサイズの増加、小さい値はサイズの減少を意味する。たとえば値 0.25 はイメージをオリジナルの 1/4 のサイズにするだろう。このスケーリングにより`:max-width`や`:max-height`で指定されたイメージの元のサイズより大きくなる場合でも、結果サイズがこれら 2 つの値を超過することはない。`:scale`および`:height`/`:width`の両方が指定されたら、高さ/幅は指定されたスケーリング倍率に調整される。

`:rotation` *angle*
: ローテーション角度を度数 (degree) で指定する。イメージタイプが`imagemagick`でなければ、90°の倍数のみをサポート。値が正なら時計回り、負なら反時計回り。ローテーションはスケーリングとクロッピング (cropping: 切り取り、抜き出し) の後に行われる。

`:index` *frame*
: Section 39.17.10 [Multi-Frame Images], page 1065 を参照のこと。

`:conversion` *algorithm*
: これはイメージを表示する前に適用するべき変換アルゴリズムを指定する。値 *algorithm* は何のアルゴリズムかを指定する。

  `laplace`
  `emboss`
  : カラーの大きな差異を強調して小さな差異を不鮮明にするラプラスエッジ検出アルゴリズム (Laplace edge detection algorithm) を指定する。無効なボタンのイメージ表示に、これが役立つと考える人もいます。

  `(edge-detection :matrix` *matrix* `:color-adjust` *adjust*`)`
  : 一般的なエッジ検出アルゴリズムを指定する。*matrix*は数値からなる 9 要素のリストかベクターでなければならない。変換されたイメージ内の位置 x/y にあるピクセルは、その位置周辺にある元のピクセルから計算される。*matrix*は x/y に近接する各ピクセルにたいして、そのピクセルが変換先ピクセルに影響するファクター (factor: 要因) を指定する。以下のように要素 0 は $x-1/y-1$ にあるピクセルのファクター、要素 1 は $x/y-1$ にあるピクセルにたいするファクター、... を指定する。

$$\begin{pmatrix} x-1/y-1 & x/y-1 & x+1/y-1 \\ x-1/y & x/y & x+1/y \\ x-1/y+1 & x/y+1 & x+1/y+1 \end{pmatrix}$$

結果となるピクセルは周辺ピクセルの RGB 値を合計したカラーを指定されたファクターで乗じて、その合計をファクター絶対値の合計で除した色強度から計算される。

ラプラスエッジ検出は現在のところは以下のマトリクス

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

エンボスエッジ検出 (Emboss edge-detection) は以下のマトリクスを使用する

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & -2 \end{pmatrix}$$

`disabled` イメージが無効 (disabled) に見えるよう変換することを指定する。

`:mask` *mask*

*mask*が `heuristic`か `(heuristic` *bg*`)`なら、フレームのバックグラウンドがイメージ背後に見えるようにイメージのクリッピングマスクを構築する。*bg*が未指定か `t`なら、イメージ 4 隅に最頻するカラーをそのイメージのバックグラウンドカラーとみなしてバックグラウンドカラーを決定する。それ以外なら *bg*はイメージのバックグラウンドとみなすべきカラーを指定するリスト `(`*red green blue*`)`でなければならない。

*mask*が `nil`なら、イメージがマスクをもつ場合にはマスクを削除する。マスクを含むフォーマットのイメージは`:mask nil`を指定することにより削除される可能性がある。

`:pointer` *shape*

これはマウスポインターがそのイメージ上にある際のポインターシェイプを指定する。利用可能なポインターシェイプについては Section 29.19 [Pointer Shape], page 746 を参照のこと。

`:map` *map* これはイメージにホットスポット *(hot spots)* のイメージマップを関連付ける。

イメージマップは各要素が `(`*area id plist*`)`という形式をもつ alist。*area*には rectangle(矩形)、circle(円)、または polygon(ポリゴン、多角形) のいずれかを指定する。

rectangle は矩形エリアの左上隅と右下隅のピクセル座標を指定するコンス `(rect . ((`*x0 . y0*`) . (`*x1 . y1*`)))`。

circle は円の中心と半径を指定するコンス `(circle . ((`*x0 . y0*`) . `*r*`))`。*r*は整数か浮動小数点数。

polygon は各ペアーが多角形の 1 つの頂点を記述するコンス `(poly . [`*x0 y0 x1 y1 ...*`])`。

マウスポインターがホットスポット上にある際には、ホットスポットの *plist*が参照される。これが`help-echo`プロパティを含むならそのホットスポットのツールチップ、`pointer`プロパティを含む場合はマウスカーソルがホットスポット上にあるときのマウスカーソルのシェイプを指定する。利用可能なポインターシェイプについては Section 29.19 [Pointer Shape], page 746 を参照のこと。

マウスポインターがホットスポット上にあるときにマウスをクリックしたときのイベントは、ホットスポットの *id*とマウスイベントを組み合わせて構成される。たとえばホットスポットの *id*が `area4`なら `[area4 mouse-1]`。

`image-mask-p` *spec* **&optional** *frame* [Function]
この関数はイメージ *spec*がマスクビットマップをもつなら `t`をリターンする。*frame*はそのイメージが表示されるフレーム。*frame*が `nil`か省略された場合には選択されたフレームが使用される (Section 29.10 [Input Focus], page 732 を参照)。

`image-transforms-p` **&optional** *frame* [Function]
この関数は *frame*がイメージのスケーリングとローテーションをサポートすれば非 `nil`をリターンする。*frame*が `nil`か省略なら選択されたフレームの使用を意味する (Section 29.10 [Input Focus], page 732 を参照)。リターンされるリストには何のイメージ変換操作がサポートされるかを示すシンボルが含まれる:

`scale` *frame*は`:scale`、`:width`、`:height`、`:max-width`、`:max-height`のプロパティを通じたイメージスケーリングをサポートする。

`rotate90` *frame*はローテーション角度が 90の整数倍ならイメージローテーションをサポートする。

イメージ変換がサポートされていなければ`:rotation`、`:crop`、`:width`、`:height`、`:scale`、`:max-width`、`:max-height`は (利用可能なら)ImageMagick を通じた場合のみ使用可能 (Section 39.17.5 [ImageMagick Images], page 1054 を参照)。

39.17.3 XBM イメージ

XBM フォーマットを使用するにはイメージタイプとして `xbm`を指定します。このイメージフォーマットは外部ライブラリーを要求せず、このタイプのイメージは常にサポートされます。

`xbm`イメージタイプにたいして追加のイメージプロパティがサポートされます:

`:foreground` *foreground*
値 *foreground*はそのイメージのフォアグラウンドカラーを指定する文字列、またはデフォルトカラーを指定する `nil`であること。このカラーは XBM 内の 1 の各ピクセルに使用される。デフォルトはフレームのフォアグラウンドカラー。

`:background` *background*
値 *background*はそのイメージのバックグラウンドカラーを指定する文字列、またはデフォルトカラーを指定する `nil`であること。このカラーは XBM 内の 0 の各ピクセルに使用される。デフォルトはフレームのバックグラウンドカラー。

外部ファイルのかわりに Emacs 内のデータを指定して XBM イメージを指定するには以下の 3 つのプロパティを使用する:

`:data` *data*
値 *data*はイメージのコンテンツを指定する。*data*として使用できる 3 つのフォーマットが存在する:

- それぞれがイメージの 1 ラインを指定するような文字列ベクターか bool ベクター。`:height`と`:width`を指定する。
- 文字列なら XBM ファイルが含むのと同じバイトシーケンスを含む。この場合は`:height`と`:width`を指定してはならない。これらを省略することが、そのデータが XBM ファイルのフォーマットをもつことを示すからである。イメージの高さと幅はファイルのコンテンツにより指定される。
- イメージのビットを含む文字列か bool ベクター (終端の使用されない余分なビットを含むかもしれない)。少なくとも ***stride* * *height*** ビットを含むこと (*stride*は

イメージ幅以上の 8 の最小倍数)。この場合にはその文字列が XBM ファイル全体ではなく、単にビットだけを含むことを示すとともに、そのイメージのサイズを指定するために`:height`、`:width`、`:stride`を指定する必要がある。

`:width` *width*
: 値 *width*はピクセル単位でイメージの幅を指定する。

`:height` *height*
: 値 *height*はピクセル単位でイメージの高さを指定する。

`:stride` *stride*
: 各行に格納さたブールベクターのエントリー数 (*width*以上の 8 の最小倍数)。

39.17.4 XPM イメージ

XPM フォーマットを使用するにはイメージタイプに `xpm`を指定します。`xpm`イメージタイプでは追加のプロパティ`:color-symbols`にも意味があります。

`:color-symbols` *symbols*
: 値 *symbols*は要素が `(`*name* `.` *color*`)`という形式をもつような alist であること。各要素において *name*はイメージファイル内に出現するカラー名、*color*はそのカラー名の実際の表示に使用するカラーを指定する。

39.17.5 ImageMagick イメージ

ImageMagick のサポートつきの Emacs ビルドでは、多くくのイメージフォーマットをロードするために ImageMagick ライブラリーを使用できます (Section “File Conveniences” in *The GNU Emacs Manual* を参照)。ImageMagick を通じてロードしたイメージのイメージタイプシンボルは、基礎となる実際のイメージフォーマットとは無関係に `imagemagick`になります。

ImageMagick サポートをチェックするには以下を使用してください:

```
(image-type-available-p 'imagemagick)
```

`imagemagick-types` [Function]
: この関数はカレントの ImageMagick インストールによりサポートされるイメージファイル拡張子のリストをリターンする。リストの各要素は`.bmp`イメージは `BMP`のような、イメージタイプにたいして内部的な ImageMagick 名を表すシンボル。

`imagemagick-enabled-types` [User Option]
: この変数の値は Emacs が ImageMagick を使用してレンダリングを試みるかもしれない ImageMagick イメージタイプのリスト。リストの各要素は `imagemagick-types`がリターンするリスト内のシンボルのいずれか、または等価な文字列。もしくは値 `t`は ImageMagick にたいして利用できるすべてのイメージタイプを有効にする。この変数の値とは関係なく `imagemagick-types-inhibit` (以下参照) が優先される。

`imagemagick-types-inhibit` [User Option]
: この変数の値は `imagemagick-enabled-types`の値とは無関係に、ImageMagick を使用して決してレンダリングされることのない ImageMagick イメージタイプのリスト。値 `t`は ImageMagick を完全に無効にする。

`image-format-suffixes` [Variable]
: この変数はイメージタイプをファイル名拡張子にマッピングする alist。Emacs は ImageMagick ライブラリーにイメージのタイプに関するヒントを与えるために、この変数と`:format`イメー

ジプロパティ(以下参照) を組み合わせて使用する。各要素は (*type extension*)という形式をもち *type*はイメージの content-type を指定するシンボル、*extension*は関連付けられるファイル名拡張子を指定する文字列。

ImageMagick によりロードされたイメージは、追加で以下のイメージディスクリプタプロパティをサポートします:

`:background` *background*
: *background*が非 `nil`なら、カラーを指定する文字列であること。これはイメージが透明度をサポートする場合に、イメージのバックグラウンドカラーとして使用される。値が `nil`の場合のデフォルトはフレームのバックグラウンドカラー。

`:format` *type*
: 値 *type*は `image-format-suffixes`で見られるような、イメージのタイプを指定するシンボルであること。これはイメージが関連付けられたファイル名をもたない際に、イメージタイプを検出する助けとなるヒントを ImageMagick に提供する。

`:crop` *geometry*
: *geometry*の値は (*width height x y*)という形式のリストであること。*width*と*height*はクロップするイメージの幅と高さを指定する。*x*が正の数値なら元イメージの左エッジ、負の数値なら右エッジからクロップのオフセットを指定する。*y*が正の数値なら元イメージの上エッジ、負の数値なら下エッジからクロップのオフセットを指定する。*x*か *y*が `nil`か未指定ならクロップ領域は元イメージの中央になる。

  クロップ領域がイメージ外部、またはエッジとオーバーラップする場合には、イメージ外部の全領域を除外するように縮小される。これは *width*や *height*に大きな値を与えてイメージサイズを増加するために`:crop`を使用できないことを意味する。

  クロッピングはスケーリング後、ローテーション前に行われる。

39.17.6 SVG イメージ

SVG(Scalable Vector Graphics) はイメージを指定する XML フォーマットです。SVG サポートつきで Emacs がビルドされていれば、以下の `svg.el`ライブラリー由来の関数でこれらのイメージの作成や操作ができます。

`svg-create` *width height* **&rest** *args* [Function]
: 指定したサイズで新たに SVG イメージを作成する。*args*は plist 引数であり、以下を指定できる:

  `:stroke-width`
  : 作成するすべてのラインのデフォルト幅 (ピクセル単位)。

  `:stroke` 作成するすべてのラインのデフォルトのストロークカラー。

  この関数は *SVG* オブジェクト (SVG イメージを指定する Lisp データ構造) をリターンする。以下の関数はすべてこのブジェクトにたいして機能する。以下の関数の引数 *svg*はこのような SVG オブジェクトを指定する。

`svg-gradient` *svg id type stops* [Function]
: *svg*に識別子 *id*でグラデーションを作成する。*type*はグラデーションタイプで`linear`か`radial`のいずれかを指定する。*stops*はパーセント割合/カラーのペアからなるリスト。

  以下は最初の赤から 25%の緑、最後は青に至る線形グラデーションを作成する:

```
(svg-gradient svg "gradient1" 'linear
```

```
'((0 . "red") (25 . "green") (100 . "blue")))
```

作成 (および SVG オブジェクトに挿入) されたグラデーションは、後でシェイプを作成するすべての関数で使用できる。

以下の関数はすべてさまざまな属性のデフォルト値を変更するオプションのキーワードパラメーターを受け取ります。有効な属性には以下が含まれます:

`:stroke-width`
: ラインとソリッドシェイプ枠線の描画幅 (ピクセル単位)。

`:stroke-color`
: ラインとソリッドシェイプ枠線の描画カラー。

`:fill-color`
: ラインとソリッドシェイプに使用するカラー。

`:id`
: シェイプの識別子。

`:gradient`
: 与えられた場合には以前に定義されたグラデーションオブジェクトの識別子であること。

`:clip-path`
: クリックパスの識別子。

`svg-rectangle` *svg x y width height* **&rest** *args* [Function]
左上隅が位置 *x*/*y*、サイズが *width*/*height*の矩形を *svg*に追加する。

```
(svg-rectangle svg 100 100 500 500 :gradient "gradient1")
```

`svg-circle` *svg x y radius* **&rest** *args* [Function]
中央が位置 *x*/*y*、半径が *radius*の円を *svg*に追加する。

`svg-ellipse` *svg x y x-radius y-radius* **&rest** *args* [Function]
中央が位置 *x*/*y*、水平半径が *x-radius*、垂直半径が *y-radius*の楕円を *svg*に追加する。

`svg-line` *svg x1 y1 x2 y2* **&rest** *args* [Function]
始点が *x1*/*y1*、終点が *x2*/*y2*の線を *svg*に追加する。

`svg-polyline` *svg points* **&rest** *args* [Function]
points (X と Y の位置ペアのリスト) を通過する複数セグメントライン、いわゆる “ポリゴン (polyline)” を *svg*に追加する。

```
(svg-polyline svg '((200 . 100) (500 . 450) (80 . 100))
              :stroke-color "green")
```

`svg-polygon` *svg points* **&rest** *args* [Function]
ポリゴン外周の位置 X と Y のペアからなるリストであるような *points*により記述されるポリゴンを *svg*に追加する。

```
(svg-polygon svg '((100 . 100) (200 . 150) (150 . 90))
             :stroke-color "blue" :fill-color "red")
```

`svg-path` *svg commands* **&rest** *args* [Function]
*commands*に応じて *svg*にシェイプのアウトラインを追加する。[SVG Path Commands], page 1058 を参照のこと。

デフォルトでは絶対座標。最後 (または最初) の位置を基準に相対座標を指定するには、属性 *:relative*に `t`をセットする。この属性は関数や個々の commands にたいして指定できる。関数に指定された場合には、すべての commands はデフォルトで相対座標を使用する。個々の commands に絶対座標を使用させるなら、*:relative*に `nil`をセットする。

```
(svg-path svg
  '((moveto ((100 . 100)))
    (lineto ((200 . 0) (0 . 200) (-200 . 0)))
    (lineto ((100 . 100)) :relative nil))
  :stroke-color "blue"
  :fill-color "lightblue"
  :relative t)
```

`svg-text` *svg text* **&rest** *args* [Function]
指定した *text*を *svg*に追加する。

```
(svg-text
 svg "This is a text"
 :font-size "40"
 :font-weight "bold"
 :stroke "black"
 :fill "white"
 :font-family "impact"
 :letter-spacing "4pt"
 :x 300
 :y 400
 :stroke-width 1)
```

`svg-embed` *svg image image-type datap* **&rest** *args* [Function]
埋め込みの (ラスター) イメージを *svg*に追加する。*datap*が `nil`なら *image*はファイル名、それ以外なら *image*はイメージデータを raw バイトとして含む文字列であること。*image-type* は`"image/jpeg"`のような MIMEイメージタイプであること。

```
(svg-embed svg "~/rms.jpg" "image/jpeg" nil
           :width "100px" :height "100px"
           :x "50px" :y "75px")
```

`svg-clip-path` *svg* **&rest** *args* [Function]
*svg*にクリッピングパスを追加する。*:clip-path*プロパティを通じてシェイプに適用された場合には、クリッピング外部のシェイプ部分は描画されない。

```
(let ((clip-path (svg-clip-path svg :id "foo")))
  (svg-circle clip-path 200 200 175))
(svg-rectangle svg 50 50 300 300
               :fill-color "red"
               :clip-path "url(#foo)")
```

`svg-node` *svg tag* **&rest** *args* [Function]
カスタムノード *tag* を *svg* に追加する。

```
(svg-node svg
          'rect
          :width 300 :height 200 :x 50 :y 100 :fill-color "green")
```

`svg-remove` *svg id* [Function]
`svg` から識別子 `id` の要素を取り除く。

`svg-image` *svg* [Function]
最後に `svg-image` は引数として SVG を受け取り、`insert-image` のような関数での使用に適したイメージオブジェクトをリターンする。

以下は円のイメージを作成して挿入する完全な例です:

```
(let ((svg (svg-create 400 400 :stroke-width 10)))
  (svg-gradient svg "gradient1" 'linear '((0 . "red") (100 . "blue")))
  (svg-circle svg 200 200 100 :gradient "gradient1"
                :stroke-color "green")
  (insert-image (svg-image svg)))
```

SVG Path Commands

SVG パス *(SVG paths)* によりライン (lines: 線)、カーブ (curves: 曲線)、アーク (arcs: 円弧)、またはその他の基本的なシェイプを組み合わせて複雑なイメージを作成できます。以下に説明する関数で Lisp プログラムから SVG パスコマンドを呼び出すことができます。

`moveto` *points* [Command]
points の最初のポイントにペンを移動する。それ以降のポイントはラインで接続される。*points* は XY 座標ペアのリスト。後続の `moveto` コマンドは新たなサブパス *(subpath)* の開始を表す。

```
(svg-path svg '((moveto ((200 . 100) (100 . 200) (0 . 100))))
          :fill "white" :stroke "black")
```

`closepath` [Command]
サブパスの初期ポイントに接続することによりカレントのサブパスを終了する。ラインは接続に沿って描画される。

```
(svg-path svg '((moveto ((200 . 100) (100 . 200) (0 . 100)))
                (closepath)
                (moveto ((75 . 125) (100 . 150) (125 . 125)))
                (closepath))
          :fill "red" :stroke "black")
```

`lineto` *points* [Command]
カレントのポイントから *points* (XY 位置ペアのリスト) の最初の要素にラインを描画する。複数ポイントを指定するとポリライン (polyline: 折れ線) を描画する。

```
(svg-path svg '((moveto ((200 . 100)))
                (lineto ((100 . 200) (0 . 100))))
          :fill "yellow" :stroke "red")
```

`horizontal-lineto` *x-coordinates* [Command]

カレントポイントから *x-coordinates*の最初の要素へ水平ラインを描画する。通常は無意味だが複数座標の指定は可能。

```
(svg-path svg '((moveto ((100 . 200)))
                (horizontal-lineto (300)))
          :stroke "green")
```

`vertical-lineto` *y-coordinates* [Command]

垂直ラインを描画する。

```
(svg-path svg '((moveto ((200 . 100)))
                (vertical-lineto (300)))
          :stroke "green")
```

`curveto` *coordinate-sets* [Command]

*coordinate-sets*の最初の要素を使用して、カレントポイントからベジェ曲線 (cubic B^^c3^^a9zier curve) を描画する。複数の座標セットがあればポリベジェ(polybezier: 複数のベジェ曲線) を描画する。座標セットはそれぞれ `(x1 y1 x2 y2 x y)`という形式のリストであり、(*x*, *y*) はカーブの終点。(*x1*, *y1*) と (*x2*, *y2*) はそれぞれ先頭ポイントと終点ポイントを制御する。

```
(svg-path svg '((moveto ((100 . 100)))
                (curveto ((200 100 100 200 200 200)
                          (300 200 0 100 100 100))))
          :fill "transparent" :stroke "red")
```

`smooth-curveto` *coordinate-sets* [Command]

*coordinate-sets*の最初の要素を使用して、カレントポイントから三次ベジェ曲線 (cubic B^^c3^^a9zier curve) を描画する。複数の座標セットがあればポリベジェ(polybezier: 複数のベジェ曲線) を描画する。座標セットはそれぞれ `(x2 y2 x y)`という形式のリストであり、(*x*, *y*) はカーブの終点、(*x2*, *y2*) は対応するコントロールポイント。前のコマンドが `curveto`か `smooth-curveto`なら、そのコマンドの 2 つ目のコントロールポイントのカレントポイントから相対的なリフレクション (reflection)。それ以外なら最初のコントロールポイントはカレントポイントと一致する。

```
(svg-path svg '((moveto ((100 . 100)))
                (curveto ((200 100 100 200 200 200)))
                (smooth-curveto ((0 100 100 100))))
          :fill "transparent" :stroke "blue")
```

`quadratic-bezier-curveto` *coordinate-sets* [Command]

*coordinate-sets*の最初の要素を使用して、カレントポイントから二次ベジェ曲線 (quadratic B^^c3^^a9zier curve) を描画する。複数の座標セットがあればポリベジェ(polybezier: 複数のベジェ曲線) を描画する。座標セットはそれぞれ `(x1 y1 x y)`という形式のリストであり、(*x*, *y*) はカーブの終点、(*x1*, *y1*) はコントロールポイント。

```
(svg-path svg '((moveto ((200 . 100)))
                (quadratic-bezier-curveto ((300 100 300 200)))
                (quadratic-bezier-curveto ((300 300 200 300)))
                (quadratic-bezier-curveto ((100 300 100 200)))
                (quadratic-bezier-curveto ((100 100 200 100))))
          :fill "transparent" :stroke "pink")
```

`smooth-quadratic-bezier-curveto` *coordinate-sets* [Command]

*coordinate-sets*の最初の要素を使用して、カレントポイントから二次ベジェ曲線 (quadratic B^^c3^^a9zier curve) を描画する。複数の座標セットがあればポリベジェ(polybezier: 複数のベジェ曲線) を描画する。座標セットはそれぞれ `(x y)` という形式のリストであり、(x, y) はカーブの終点。前のコマンドが `quadratic-bezier-curveto`か `smooth-quadratic-bezier-curveto`なら、そのコマンドのコントロールポイントのカレントポイントから相対的なリフレクション (reflection)。それ以外なら最初のコントロールポイントはカレントポイントと一致する。

```
(svg-path svg '((moveto ((200 . 100)))
                (quadratic-bezier-curveto ((300 100 300 200)))
                (smooth-quadratic-bezier-curveto ((200 300)))
                (smooth-quadratic-bezier-curveto ((100 200)))
                (smooth-quadratic-bezier-curveto ((200 100))))
          :fill "transparent" :stroke "lightblue")
```

`elliptical-arc` *coordinate-sets* [Command]

*coordinate-sets*の最初の要素を使用して、カレントポイントから楕円弧 (elliptical arc) を描画する。複数の座標セットがあれば一連の楕円弧を描画する。座標セットはそれぞれ `(rx ry x y)` という形式のリストであり、(x, y) は楕円の終点、(rx, ry) は楕円の半径。リストに属性を追加できる:

`:x-axis-rotation`
: カレントの座標システムの X 軸から相対的にローテートされた楕円 X 軸の度数角度。

`:large-arc`
: `t`にセットすると、180 ̊以上のアークスイープ (arc sweep) を描画する。それ以外なら 180 ̊以下のアークスイープを描画する。

`:sweep` `t`にセットすると正の角度方向 *(positive angle direction)*、それ以外なら負の角度方向 *(negative angle direction)* にアークを描画する。

```
(svg-path svg '((moveto ((200 . 250)))
                (elliptical-arc ((75 75 200 350))))
          :fill "transparent" :stroke "red")
(svg-path svg '((moveto ((200 . 250)))
                (elliptical-arc ((75 75 200 350 :large-arc t))))
          :fill "transparent" :stroke "green")
(svg-path svg '((moveto ((200 . 250)))
                (elliptical-arc ((75 75 200 350 :sweep t))))
          :fill "transparent" :stroke "blue")
(svg-path svg '((moveto ((200 . 250)))
                (elliptical-arc ((75 75 200 350 :large-arc t
                                        :sweep t))))
          :fill "transparent" :stroke "gray")
(svg-path svg '((moveto ((160 . 100)))
                (elliptical-arc ((40 100 80 0)))
                (elliptical-arc ((40 100 -40 -70
                                        :x-axis-rotation -120)))
```

```
                (elliptical-arc ((40 100 -40 70
                                  :x-axis-rotation -240))))
          :stroke "pink" :fill "lightblue"
          :relative t)
```

39.17.7 その他のイメージタイプ

PBM イメージにはイメージタイプ `pbm`を指定します。カラー、グレースケール、およびモノクロのイメージがサポートされます。モノクロの PBM イメージにたいしては、2 つの追加イメージプロパティがサポートされます。

`:foreground` *foreground*
: 値 *foreground*はイメージのフォアグラウンドカラーを指定する文字列、またはデフォルトカラーなら `nil`であること。このカラーは PBM 内の 1 であるようなピクセルすべてに使用される。デフォルトはフレームのフォアグラウンドカラー。

`:background` *background*
: 値 *background*はイメージのバックグラウンドカラーを指定する文字列、またはデフォルトカラーなら `nil`であること。このカラーは PBM 内の 0 であるようなピクセルすべてに使用される。デフォルトはフレームのバックグラウンドカラー。

Emacs がサポート可能な残りのイメージタイプは以下のとおり:

GIF
: イメージタイプ `gif`。`:index`プロパティをサポートする。Section 39.17.10 [Multi-Frame Images], page 1065 を参照のこと。

JPEG
: イメージタイプ `jpeg`。

PNG
: イメージタイプ `png`。

TIFF
: イメージタイプ `tiff`。`:index`プロパティをサポートする。Section 39.17.10 [Multi-Frame Images], page 1065 を参照のこと。

39.17.8 イメージの定義

関数 `create-image`、`defimage`、`find-image`はイメージディスクリプタを作成するための便利な手段を提供します。

`create-image` *file-or-data* **&optional** *type data-p* **&rest** *props* [Function]
: この関数は *file-or-data*内のデータを使用するイメージディスクリプタを作成してリターンする。*file-or-data*はファイル名、またはイメージデータを含む文字列を指定できる。前者なら *data-p*は `nil`、後者なら非 `nil`であること。

  オプション引数 *type*はイメージタイプを指定するシンボル。*type*が省略か `nil`なら、`create-image`はファイル先頭の数バイトかファイル名からイメージタイプの判断を試みる。

  残りの引数 *props*は追加のイメージプロパティを指定する。たとえば、

  ```
  (create-image "foo.xpm" 'xpm nil :heuristic-mask t)
  ```

  この関数はそのタイプのイメージがサポートされていなければ `nil`、それ以外ならイメージディスクリプタをリターンする。

`defimage` *symbol specs* **&optional** *doc* [Macro]
: このマクロはイメージマクロとして *symbol*を定義する。引数 *specs*はイメージの表示方法を指定するリストである。3 つ目の引数 *doc*はオプションのドキュメント文字列。

*specs*内の各要素はプロパティリストの形式をもち、それぞれが少なくとも`:type`プロパティと、`:file`か`:data`いずれかのプロパティをもつこと。`:type`の値はイメージタイプを指定するシンボル、`:file`の値はイメージをロードするファイル、`:data`の値は実際のイメージデータを含む文字列であること。以下は例:

```
(defimage test-image
  ((:type xpm :file "~/test1.xpm")
   (:type xbm :file "~/test1.xbm")))
```

`defimage`はそれが使用可能か、つまりそのタイプがサポートされているかとファイルが存在するかを確認するために各要素を 1 つずつテストする。最初に使用可能な引数が *symbol*内に格納するイメージディスクリプタを作成するために使用される。

機能する候補がなければ *symbol*は`nil`として定義される。

`image-property` *image property* [Function]

*image*の *property*の値をリターンする。プロパティは`setf`を使用してセットできる。プロパティに`nil`をセットすることによりイメージからプロパティを削除できる。

`find-image` *specs* [Function]

この関数はイメージ仕様 *specs*のリストの 1 つを満足するイメージを探すための、便利な手段を提供する。

*specs*内の各仕様はイメージタイプに応じた内容のプロパティリストである。すべての仕様は少なくとも`:type` *type*、および`:file` *file*か`:data` *data*のいずれかのプロパティを含まなければならない。ここで *type*は`xbm`のようにイメージタイプを指定するシンボル、*file*はイメージをロードするファイル、*data*は実際のイメージデータを含む文字列。このリスト内で *type*がサポートされていて、かつ *file*が存在する最初の仕様が、リターンされるイメージ仕様の構築に使用される。満足する仕様がなければ`nil`がリターンされる。

イメージは`image-load-path`内で検索される。

`image-load-path` [User Option]

この変数の値はイメージファイルを検索する場所のリスト。要素が文字列か値が文字列であるような変数シンボルなら、その文字列が検索を行うディレクトリーの名前になる。値がリストであるような変数シンボルの場合、それは検索を行うディレクトリーのリストとなる。

デフォルトでは`data-directory`で指定されたディレクトリーのサブディレクトリー`images`、次に`data-directory`で指定されたディレクトリー、最後に`load-path`で指定されたディレクトリー内を検索する。サブディレクトリーは自動的には検索に含まれないので、イメージファイルをサブディレクトリー内に配置した場合には、サブディレクトリーを明示的に与える必要がある。たとえば`data-directory`内でイメージ`images/foo/bar.xpm`を見つけるには以下のようにそのイメージを指定すること:

```
(defimage foo-image '((:type xpm :file "foo/bar.xpm")))
```

`image-load-path-for-library` *library image* **&optional** *path* *no-error* [Function]

この関数は Lisp パッケージ *library*により使用されるイメージにたいして適切な検索パスをリターンする。

この関数はまず `image-load-path` (`data-directory/images`を除外) を使用し、次に`load-path`の後に *library*にとって適切なパス (ライブラリーファイル自身にたいする相対パス`../../etc/images`と`../etc/images`を含む) を補い、最後に `data-directory/images` から *image*を検索する。

それからこの関数は先頭に *image*が見つかったディレクトリー、その後に `load-path`の値が続くディレクトリーのリストをリターンする。*path*が与えられたら `load-path`のかわりに使用する。

*no-error*が非 `nil`、かつ適切なパスが見つからない場合にはエラーをシグナルしない。かわりに前記のディレクトリーリストをリターンするが、イメージのディレクトリーの箇所に `nil`が出現する点が異なる。

以下は `image-load-path-for-library`の使用例:

```
(defvar image-load-path) ; shush compiler
(let* ((load-path (image-load-path-for-library
                   "mh-e" "mh-logo.xpm"))
       (image-load-path (cons (car load-path)
                              image-load-path)))
  (mh-tool-bar-folder-buttons-init))
```

イメージは `image-scaling-factor`変数にもとづいて作成時に自動的にスケーリングされます。この値は浮動小数点数 (1 より大きい値はサイズの拡大、小さい値はサイズの縮小を意味する)、またはフォントのピクセルサイズにもとづいたスケーリング倍率で計算を行うシンボル `auto`のいずれかです。

39.17.9 イメージの表示

自分で `display`プロパティをセットアップしてイメージディスクリプタを使用できますが、このセクションの関数を使用するほうがより簡単です。

`insert-image` *image* **&optional** *string area slice* [Function]

この関数はカレントバッファーのポイント位置に *image*を挿入する。*image*はイメージディスクリプタであること。これは `create-image`によりリターンされた値、または `defimage`で定義されたシンボルの値を使用できる。引数 *string*はイメージを保持するためにバッファー内に配置するテキストを指定する。これが省略か `nil`なら、`insert-image`はデフォルトで`" "`を使用する。

引数 *area*はマージン内にイメージを置くかどうかを指定する。これが `left-margin`なら左マージンにイメージが表示され、`right-margin`なら右マージンを指定する。*area*が `nil`か省略なら、イメージはバッファーのテキスト内のポイント位置に表示される。

引数 *slice*は挿入するイメージのスライスを指定する。*slice*が `nil`か省略された場に合はイメージ全体が挿入される。それ以外では、*slice*がリスト `(x y width height)`なら *x*と *y*は位置、*width*と *height*は挿入するイメージの領域を指定する。整数値はピクセル単位。0.0 から 1.0 までの浮動小数点数はイメージ全体の幅や高さにたいする割合を指定する。

この関数は内部的にはバッファー内に *string*を挿入して、*image*を指定する `display`プロパティにそれを渡す。Section 39.16 [Display Property], page 1043 を参照のこと。

`insert-sliced-image` *image* **&optional** *string area rows cols* [Function]

この関数は `insert-image`と同様にカレントバッファー内に *image*を挿入するが、イメージを *rows*^^e2^^9c^^95*cols*の同一サイズのスライスに分割する点が異なる。

Emacs は各スライスを個別のイメージとして表示して、(巨大な) イメージを表示するバッファーのページングの際にイメージ全体を上下にジャンプするのではなく、より直感的な上下スクロールが可能になる。

`put-image` *image pos* **&optional** *string area* [Function]
この関数はカレントバッファー内の *pos*の前にイメージ *image*を配置する。引数 *pos*は整数かマーカーであること。これはイメージが表示されるべきバッファー位置を指定する。引数 *string*は代替として表示されるべきデフォルトのイメージを保持するテキストであること。

引数 *image*はイメージディスクリプタでなければならず、それは `create-image`がリターンされたか、あるいは `defimage`により格納されたイメージディスクリプタかもしれない。

引数 *area*はマージン内にイメージを置くかどうかを指定する。これが `left-margin`なら左マージンにイメージが表示され、`right-margin`なら右マージンを指定する。*area*が `nil`か省略なら、イメージはバッファーのテキスト内のポイント位置に表示される。

内部的には、この関数はオーバーレイを作成して、値がそのイメージであるような `display`プロパティをもつテキストを含む、`before-string`プロパティをそのオーバーレイに与えている (なんと!)。

`remove-images` *start end* **&optional** *buffer* [Function]
この関数は *buffer*の位置 *start*と *end*の間のイメージを削除する。*buffer*が省略か `nil`ならカレントバッファーからイメージを削除する。

これは `put-image`が行う方法で *buffer*に配置されたイメージだけを削除して、`insert-image`や他の方法で挿入されたイメージは削除しない。

`image-size` *spec* **&optional** *pixels frame* [Function]
この関数はペアー (`width . height`) としてイメージのサイズをリターンする。*spec*はイメージ spec。*pixels*が非 `nil`ならピクセル単位、それ以外なら *frame*のデフォルトの文字サイズ単位で量ったサイズをリターンする (Section 29.3.2 [Frame Font], page 707 を参照)。*frame*はイメージが表示されるフレーム。*frame*が `nil`または省略された場合には選択されたフレームを使用する (Section 29.10 [Input Focus], page 732 を参照)。

`max-image-size` [Variable]
この変数は Emacs がロードするイメージの最大サイズを定義するために使用される。Emacs はこの制限より大きいイメージのロード (と表示) を拒絶するだろう。

値が整数ならピクセル単位で量ったイメージの最大の高さと幅を直接指定する。浮動小数点数ならフレームの高さと幅にたいする比率として、イメージの最大の高さと幅を指定する。値が数値でなければイメージサイズにたいする明示的な制限は存在しない。

この変数の目的は意図せず Emacs に不当に大きなイメージがロードされるとを防ぐことである。これはイメージの初回ロード時だけ効果がある。イメージが一度イメージキャッシュに置かれると、その後に `max-image-size`の値が変更されても、そのイメージは常に表示可能である (Section 39.17.11 [Image Cache], page 1065 を参照)。

上記の挿入関数で挿入されたイメージは、表示されたイメージを横断するテキスト (またはオーバーレイ) のプロパティ内にインストールされたローカルキーマップも取得します。このキーマップは以下のコマンドを定義します。

`+` イメージサイズを拡大する (`image-increase-size`)。'4'というプレフィクス値は 40% のサイズ拡大を意味する。デフォルトは 20%。

`-` イメージサイズを縮小する (`image-increase-size`)。'4'というプレフィクス値は 40% のサイズ縮小を意味する。デフォルトは 20%。

`r` イメージを時計回りに 90 ゜ローテートする (`image-rotate`。プレフィスを与えるとかわりに反時計回りに 90 ゜ローテートすることを意味する。

`o` イメージをファイルに保存する (`image-save`)。

39.17.10 マルチフレームのイメージ

複数のイメージを含むことができるイメージファイルがいくつかあります。わたしたちはこのような場合には、イメージ内に複数の “フレーム” があると表現しています。現在のところ Emacs は GIF、TIFF、および DJVM のような特定の ImageMagick フォーマットにたいする複数フレームをサポートします。

フレームは複数のページを表現するため (通常はたとえばマルチフレーム TIFF のケースが該当)、あるいはアニメーションを作成するため (通常はマルチフレーム GIF ファイルのケースが該当) に使用できます。

マルチフレームイメージは、表示されるフレームを指定する整数値 (0 から数える) が値であるようなプロパティ `:index`をもっています。

`image-multi-frame-p` *image* [Function]
この関数は *image*が 2 つ以上のフレームを含めば非 `nil`をリターンする。実際のリターン値はコンス (`(`*nimages* `.` *delay*`)`であり *nimages*はフレーム数、*delay*はフレーム間の遅延秒数、イメージが遅延を指定しなければ `nil`。通常はアニメーションを意図されたイメージはフレームの遅延を指定して、複数ページとして扱われることを意図したイメージは指定しない。

`image-current-frame` *image* [Function]
この関数は *image*にたいして 0 から数えたカレントフレーム番号のインデックスをリターンする。

`image-show-frame` *image n* **&optional** *nocheck* [Function]
この関数は *image*をフレーム番号 *n*とスイッチする。*nocheck*が `nil`なら有効範囲外のフレーム番号を範囲終端に置き換える。*image*が指定された番号のフレームを含まなければイメージは中抜きの四角 (hollow box) で表示される。

`image-animate` *image* **&optional** *index limit* [Function]
この関数は *image*をアニメーション表示する。オプションの整数 *index*は開始するフレームを指定する (デフォルトは 0)。オプション引数 *limit*はアニメーションの長さを制御する。これが省略か `nil`ならアニメーション回数は 1 回、`t`なら永久にループ表示する。数値ならその秒数後にアニメーションは停止する。

Animation operates by means of a timer. Note that Emacs imposes a アニメーションはタイマーにより処理されます。Emacs は最小のフレーム遅延を 0.01 秒 (`image-minimum-frame-delay`の値) とすることに注意してください。そのイメージ自身が遅延を指定しなければ Emacs は `image-default-frame-delay`を使用します。

`image-animate-timer` *image* [Function]
この関数はもし存在すれば *image*のアニメーションに責任をもつタイマーをリターンする。

39.17.11 イメージキャッシュ

Emacs はイメージをより効果的に再表示できるようにイメージをキャッシュします。Emacs がイメージを表示する際には、既存のイメージ仕様が望む仕様と `equal`なイメージキャッシュを検索します。マッチが見つかったらイメージはキャッシュから表示され、それ以外ではイメージは通常のようにロードされます。

`image-flush` *spec* **&optional** *frame* [Function]
この関数はフレーム *frame*のイメージキャッシュから仕様 *spec*のイメージを削除する。イメージ仕様の比較には `equal`を使用する。*frame*が `nil`の場合のデフォルトは選択されたフレーム。*frame*が `t`ならイメージはすべての既存フレームでフラッシュされる。

Emacs の現実装では各グラフィカル端末はイメージキャッシュを処理して、それはその端末上のすべてのフレームにより共有される (Section 29.2 [Multiple Terminals], page 698 を参照)。つまりあるフレームでイメージをリフレッシュすると、同一端末上の他のすべてのフレームでもリフレッシュされる。

`image-flush`の 1 つの用途は Emacs にイメージファイルの変更を伝えることです。イメージ仕様が`:file`プロパティを含む場合には、そのイメージの初回表示時にファイルコンテンツにもとづいてイメージがキャッシュされます。たとえその後にファイルが変更されても、Emacs はそのイメージの古いバージョンを表示し続けます。`image-flush`を呼び出すことによりそのイメージはキャッシュからフラッシュされて、イメージの表示が次回必要になった際に Emacs にファイルの再読み込みを強制します。

`image-flush`の他の用途はメモリー節約です。Lisp プログラムで `image-cache-eviction-delay` (以下参照) より遥かに短い期間に多数の一時イメージを作成する場合には、Emacs が自動的に行うことを待たずに自身で使用されていないイメージのフラッシュを選択できます。

`clear-image-cache` **&optional** *filter* [Function]
この関数はイメージキャッシュ内に格納されたすべてのイメージを削除してイメージキャッシュをクリアーする。*filter*が省略か `nil`なら選択されたフレームにたいしてキャッシュをクリアーする。*filter*がフレームなら、そのフレームにたいしてキャッシュをクリアーする。*filter*が `t`なら、すべてのイメージキャッシュをクリアーする。それ以外なら *filter*はファイル名として解釈されて、すべてのイメージキャッシュからそのファイル名に関連付けられたすべてのイメージを削除する。

イメージキャッシュ内のイメージが指定された期間内に表示されなければ、Emacs はそれをキャッシュから削除して割り当てられたメモリーを解放します。

`image-cache-eviction-delay` [Variable]
この変数は表示されることなくイメージがキャッシュ内に残留できる秒数を指定する。あるイメージがこの秒数の間に表示されなければ、Emacs はそれをイメージキャッシュから削除する。

ある状況下では、もしキャッシュ内のイメージ数が大きくなり過ぎた場合には実際の立ち退き遅延 (eviction delay) はこれより短くなり得る。

値が `nil`なら明示的にキャッシュをクリアーした場合を除き、Emacs はキャッシュからイメージを削除しない。このモードはデバッグ時に有用かもしれない。

39.18 埋め込みネイティブウィジェット

必要なサポートライブラリーつきで Emacs がビルドされていて、かつグラフィカル端末上で実行されていれば、Emacs バッファー内に GTK+ WebKit ウィジェットのようなネイティブウィジェットを表示することができます。Emacs が埋め込みウィジェットを表示可能かテストするには、`xwidget-internal`機能が利用可能かどうかをチェックします (Section 16.7 [Named Features], page 279 を参照)。

Emacs バッファー内に埋め込みウィジェットを表示するためには、最初に xwidget オブジェクトを作成して、テキストプロパティまたはオーバーレイプロパティ`display`内のディスプレイ仕様としてそのオブジェクトを使用します (Section 39.16 [Display Property], page 1043 を参照)。

`make-xwidget` *type title width height arguments* **&optional** *buffer* [Function]
これは xwidget オブジェクトを作成してリターンする。*buffer*が省略か `nil`の場合のデフォルトはカレントバッファー。*buffer*が存在しないバッファーの名前を指定する場合には作成する。*type*は xwidget コンポーネントを識別するもので以下のいずれかが可能：

`webkit` WebKit コンポーネント。

引数 *width*と *height*はウィジェットのサイズをピクセル単位で指定して、*title*はそのタイトルを指定する文字列。

`xwidgetp` *object* [Function]
この関数は *object*が xwidget なら `t`、それ以外は `nil`をリターンする。

`xwidget-plist` *xwidget* [Function]
この関数は *xwidget*のプロパティリストをリターンする。

`set-xwidget-plist` *xwidget plist* [Function]
この関数は *plist*で与えられた新たなプロパティリストで *xwidget*のプロパティリストを置き換える。

`xwidget-buffer` *xwidget* [Function]
この関数は *xwidget*のバッファーをリターンする。

`get-buffer-xwidgets` *buffer* [Function]
この関数は *buffer*に関連付けられた xwidget オブジェクトのリストをリターンする。*buffer*はバッファーオブジェクトか既存のバッファー名 (文字列) を指定できる。*buffer*に xwidget が含まれなければ値は `nil`。

`xwidget-webkit-goto-uri` *xwidget uri* [Function]
この関数は与えられた *xwidget*内で指定した *uri*をブラウズ (browse: 閲覧) する。*uri*はファイルか URL を指定する文字列。

`xwidget-webkit-execute-script` *xwidget script* [Function]
この関数は *xwidget*で指定されるブラウザウィジェットに、`script`で指定する JavaScript を実行させる。

`xwidget-webkit-execute-script-rv` *xwidget script* **&optional** *default* [Function]
この関数は `xwidget-webkit-execute-script`と同様に指定した *script*を実行するが、スクリプトのリターン値も文字列としてリターンする。この関数は *script*が値をリターンしなければ *default*、*default*が省略されたら `nil`をリターンする。

`xwidget-webkit-get-title` *xwidget* [Function]
この関数は *xwidget*のタイトルを文字列としてリターンする。

`xwidget-resize` *xwidget width height* [Function]
この関数は指定した *xwidget*を *width*x*height*のサイズ (ピクセル単位) にリサイズする。

`xwidget-size-request` *xwidget* [Function]
この関数は *xwidget*のサイズを `(width height)`という形式のリストでリターンする。単位はピクセル。

`xwidget-info` *xwidget* [Function]
この関数は `[type title width height]` という形式のベクターで *xwidget*の属性をリターンする。属性は通常は xwidget の作成時に `make-xwidget`で決定される。

`set-xwidget-query-on-exit-flag` *xwidget flag* [Function]
この関数は Emacs が *xwidget*に関連付けられたバッファーの exit や kill の前にユーザーに確認を求めるようにアレンジすることを可能にする。*flag*が非 `nil`なら Emacs はユーザーに確認を求めて、それ以外なら確認を求めない。

`xwidget-query-on-exit-flag` *xwidget* [Function]
この関数は *xwidget*の query-on-exit フラグのカレントセッティングを `t`か `nil`のいずれかでリターンする。

39.19 ボタン

Button パッケージはマウスやキーボードコマンドでアクティブ化することができる、ボタン *(buttons)* の挿入と操作に関する関数を定義します。これらのボタンは典型的には種々のハイパーリンクに使用されます。

本質的にボタンとはバッファー内のテキスト範囲にアタッチされたテキストプロパティやオーバーレイのプロパティのセットです。これらのプロパティはボタンプロパティ *(button properties)* と呼ばれます。これらのプロパティのうちの 1 つはアクションプロパティ *(action property)* であり、これはユーザーがキーボードかマウスを使用してボタンを呼び出した際に呼び出される関数を指定します。アクション関数はボタンを調べ、必要に応じて他のプロパティを使用できます。

いくつかの機能面で Button パッケージと Widget パッケージは重複しています。Section “Introduction” in *The Emacs Widget Library* を参照してください。Button パッケージの利点は、より高速で小さくプログラムにたいしてよりシンプルであることです。ユーザーの観点からは、2 つのパッケージが提供するインターフェイスは非常に類似しています。

39.19.1 ボタンのプロパティ

ボタンはその外観と振る舞いを定義するプロパティの連想リスト (associated list) をもち、アプリケーションの特別な目的のために他の任意のプロパティを使用できます。以下のプロパティは Button パッケージにたいして特別な意味をもちます:

`action` ユーザーがボタンを呼び出した際に呼び出す関数であり、単一の引数 *button*を渡して呼び出される。デフォルトではこれは何も行わない `ignore`。

`mouse-action`
これは `action`と似ているが与えられた際には、(`RET`押下のかわりに) マウスクリックによりボタンが呼び出された場合 n `action`のかわりに使用される。与えられなければマウスクリックはかわりに `action`を使用する。

`face` このタイプのボタンが表示される方法を制御する Emacs フェイス。デフォルトは `button` フェイス。

`mouse-face`
ボタン上にマウスがある際の外観を制御する追加のフェイス (通常の button フェイスとマージされる)。デフォルトは Emacs の通常の `highlight`フェイス。

`keymap` そのボタンリージョン (button region) でアクティブなバインディングを定義するボタンのキーマップ。デフォルトは変数 `button-map`に格納された通常のボタンリージョンキーマップであり、これはボタン呼び出しにたいして `RET`と `mouse-2`を定義している。

`type` ボタンのタイプ。Section 39.19.2 [Button Types], page 1069 を参照のこと。

`help-echo`
Emacs のツールチップヘルプシステムが表示する文字列であり、デフォルトは`"mouse-2, RET: Push this button"`。かわりに表示される文字列をリターンする関数や文字列に評価されるフォーム、あるいは`nil`。詳細は [Text help-echo], page 821 を参照のこと。

関数なら *window*、*object*、*pos*という 3 つの引数で呼び出される。2 つ目の引数 *object* はプロパティをもつオーバーレイ (オーバーレイボタンにたいして)、あるいはボタンを含むバッファー (テキストプロパティボタンにたいして) のいずれか。その他の引数はスペシャルテキストプロパティ`help-echo`の場合と同じ意味をもつ。

`follow-link`
このボタンにたいして`mouse-1`クリックが振る舞う方法を定義する`follow-link`プロパティ。Section 32.19.8 [Clickable Text], page 828 を参照のこと。

`button` すべてのボタンは非`nil`の`button`プロパティをもち、これはボタンを含むテキストリージョンを探すのに有用かもしれない (標準的なボタン関数はこれを行う)。

ボタン内のテキストリージョンにたいして定義された他のプロパティも存在しますが、それらは典型的な用途にたいしては一般的には無関係でしょう。

39.19.2 ボタンのタイプ

すべてのボタンはボタンのプロパティにたいするデフォルト値を定義するボタンタイプ *(button type)* をもっています。ボタンタイプは、より汎用的なタイプから特化したタイプへと継承される階層構造で構成されており、特定のタスクにたいして特殊用途のボタンを簡単に定義できます。

`define-button-type` *name* **&rest** *properties* [Function]
name (シンボル) と呼ばれるボタンタイプを定義する。残りの引数は *property value*ペアーのシーケンスを形成する。これはそのタイプのボタンにたいするデフォルトのプロパティ値を指定する (ボタンのタイプはキーワード引数`:type`を使用してボタン作成時にそれを`type`プロパティに与えることによりセット可能)。

加えて *name*がデフォルトプロパティ値を継承するボタンタイプ指定するためにキーワード引数`:supertype`を使用できる。この継承は *name*の定義時のみ発生することに注意。その後に supertype に行われた変更は subtype には反映されない。

`define-button-type`を使用してボタンのデフォルトプロパティを定義するのは必須ではありません — 特定のタイプをもたないボタンはビルトインのボタンタイプ`button`を使用します — が推奨しません。これを行うことにより通常はコードがより明快かつ効果的になるからです。

39.19.3 ボタンの作成

ボタンはボタン固有の情報を保持するために、オーバーレイプロパティかテキストプロパティを使用してテキストのリージョンに関連付けられます。これらはすべてボタンのタイプ (デフォルトはビルトインのボタンタイプ`button`) から初期化されます。すべての Emacs テキストと同じようにボタンの外観は`face`プロパティにより制御されます。(ボタンタイプ`button`から継承された`face`プロパティを通じることにより) デフォルトでは典型的なウェブページリンクのようなシンプルなアンダーラインです。

簡便さのために 2 種類のボタン作成関数があります。1 つはバッファーの既存リージョンにボタンプロパティを追加する`make-...button`と呼ばれる関数、もう 1 つはボタンテキストを挿入する`insert-...button`と呼ばれる関数です。

すべてのボタン作成関数は`&rest`引数の *properties*を受け取ります。これはボタンに追加するプロパティを指定する *property value*ペアーのシーケンスである必要があります。Section 39.19.1 [Button Properties], page 1068 を参照してください。これに加えて他のプロパティの継承元となるボタンタイプの指定にキーワード引数`:type`を使用できます。Section 39.19.2 [Button Types], page 1069 を参照してください。作成の間に明示的に指定されなかったプロパティは、(そのタイプがそのようなプロパティを定義していれば) そのボタンのタイプから継承されます。

以下の関数はボタンプロパティを保持するためにオーバーレイを使用してボタンを追加します (Section 39.9 [Overlays], page 999 を参照)。

`make-button` *beg end* **&rest** *properties* [Function]
これはカレントバッファー内の *beg*から *end*にボタンを作成してリターンする。

`insert-button` *label* **&rest** *properties* [Function]
これはポイント位置にラベル *label*のボタンを挿入してリターンする。

以下の関数も同様ですが、ボタンプロパティを保持するためにテキストプロパティを使用します (Section 32.19 [Text Properties], page 813 を参照)。この種のボタンはバッファーにマーカーを追加しないので、非常に多数のボタンが存在してもバッファーでの編集が低速になることはありません。しかしそのテキストに既存の face テキストプロパティが存在する場合 (たとえば Font Lock モードにより割り当てられたフェイス) には、そのボタンのフェイスは可視にならないかもしれません。これらの関数はいずれも新たなボタンの開始位置をリターンします。

`make-text-button` *beg end* **&rest** *properties* [Function]
これはテキストプロパティを使用してカレントバッファー内の *beg*から *end*にボタンを作成する。

`insert-text-button` *label* **&rest** *properties* [Function]
これはテキストプロパティを使用してポイント位置にラベル *label*のボタンを挿入する。

39.19.4 ボタンの操作

ボタンのプロパティの取得やセットを行う関数が存在します。これらは何を行うかを判断するためにボタンが呼び出す関数からよく使用される関数です。

*button*パラメーターが指定された場合にはオーバーレイ (オーバーレイボタンの場合)、またはバッファー位置やマーカー (テキストプロパティボタンの場合) いずれかという、特定のボタンを参照するオブジェクトを意味します。そのようなオブジェクトはボタンが関数を呼び出す際に 1 つ目の引数として渡されます。

`button-start` *button* [Function]
*button*が開始される位置をリターンする。

`button-end` *button* [Function]
*button*が終了する位置をリターンする。

`button-get` *button prop* [Function]
ボタン *button*の *prop*という名前のプロパティを取得する。

`button-put` *button prop val* [Function]
*button*の *prop*プロパティに *val*をセットする。

`button-activate` *button* **&optional** *use-mouse-action* [Function]
*button*の `action`プロパティを呼び出す (単一の引数 *button*を渡してプロパティの値である関数を呼び出す)。*use-mouse-action*が非 `nil`なら、`action`のかわりにそのボタンの `mouse-action`プロパティの呼び出しを試みる。ボタンが `mouse-action`プロパティをもたなければ通常どおり `action`を使用する。*button*で `button-data`プロパティが与えられた場合には、`action`関数の引数として *button*のかわりに使用される。

`button-label` *button* [Function]
*button*のテキストラベルをリターンする。

`button-type` *button* [Function]
*button*のボタンタイプをリターンする。

`button-has-type-p` *button type* [Function]
*button*がボタンタイプ *type*、または *type*の subtype のいずれかをもつなら `t`をリターンする。

`button-at` *pos* [Function]
カレントバッファー内の位置 *pos*にあるボタン、または `nil`をリターンする。*pos*にあるボタンがテキストプロパティボタンならリターン値は *pos*を指すマーカー。

`button-type-put` *type prop val* [Function]
ボタンタイプ *type*の *prop*プロパティに *val*をセットする。

`button-type-get` *type prop* [Function]
ボタンタイプ *type*の *prop*という名前のプロパティを取得する。

`button-type-subtype-p` *type supertype* [Function]
ボタンタイプ *type*が *supertype*の subtype なら `t`をリターンする。

39.19.5 ボタンのためのバッファーコマンド

Emacs バッファー内にボタンの配置や操作を行うコマンドや関数が存在します。

`push-button`はユーザーが実際にボタンを押下 (push) するために使用するコマンドであり、そのボタンのオーバーレイプロパティかテキストプロパティを使用することにより、そのボタンの `RET`と `mouse-2`にデフォルトでバインドされます。ボタン自身の外部で有用な `forward-button`や `backward-button`のようなコマンドは、`button-buffer-map`に格納されたキーマップ内で追加で利用可能です。ボタンを使用するモードはそのキーマップの親キーマップとして `button-buffer-map`の使用を望むかもしれません。

ボタンが非 `nil`の `follow-link`プロパティをもち、かつ `mouse-1-click-follows-link`がセットされている場合には、素早い `mouse-1`クリックにより `push-button`コマンドもアクティブになるでしょう。Section 32.19.8 [Clickable Text], page 828 を参照してください。

`push-button` **&optional** *pos use-mouse-action* [Command]
位置 *pos*にあるボタンが指定するアクションを行う。*pos*はバッファー位置、またはマウスイベントのいずれか。*use-mouse-action*が非 `nil`、または *pos*がマウスイベントなら `action`のかわりにそのボタンの `mouse-action`プロパティの呼び出しを試みて、ボタンに `mouse-action`プロパティがなければ通常のように `action`を使用する。`push-button`がマウスイベントの結果としてインタラクティブに呼び出されたときはそのマウスイベントの位置、それ以外ではポイントの位置が *pos*のデフォルトになる。*pos*にボタンがなければ何もせずに `nil`をリターンして、それ以外なら `t`をリターンする。

`forward-button` *n* **&optional** *wrap display-message no-error* [Command]
次の *n*番目、*n*が負なら前の *n*番目のボタンに移動する。*n*が 0 ならポイント位置にある任意のボタンの開始に移動する。*wrap*が非 `nil`ならバッファーの先頭または終端を超えてもう一方の端へ移動を継続する。*display-message*が非 `nil`ならボタンの help-echo 文字列が表示される。非 `nil`の `skip`プロパティをもつボタンはすべてスキップされる。見つかったボタンをリターンするか、ボタンが見つからなければエラーをシグナルする。*no-error*が非 `nil`なら、エラーをシグナルするかわりに `nil`をリターンする。

`backward-button` *n* **&optional** *wrap display-message* [Command]
前の *n*番目、*n*が負なら次の *n*番目のボタンに移動する。*n*が 0 ならポイント位置にある任意のボタンの開始に移動する。*wrap*が非 `nil`ならバッファーの先頭または終端を超えて、もう一方の端へ移動を継続する。*display-message*が非 `nil`ならボタンの help-echo 文字列が表示される。非 `nil`の `skip`プロパティをもつボタンはすべてスキップされる。見つかったボタンをリターンするか、ボタンが見つからなければエラーをシグナルする。*no-error*が非 `nil`なら、エラーをシグナルするかわりに `nil`をリターンする。

`next-button` *pos* **&optional** *count-current* [Function]
`previous-button` *pos* **&optional** *count-current* [Function]
カレントバッファー内の位置 *pos*の次 (`next-button`の場合)、または前 (`previous-button`の場合) のボタンをリターンする。*count-current*が非 `nil`なら、次のボタンから検索を開始するかわりに *pos*にある任意のボタンを考慮する。

39.20 抽象的なディスプレー

Ewoc パッケージは Lisp オブジェクトの構造を表すバッファーテキストを構成して、その構造体の変更にしたがってテキストを更新します。これはデザインパラダイム “model–view–controller” 内の “view” コンポーネントと似ています。Ewoc は “Emacs's Widget for Object Collections(オブジェクトコレクション用 Emacs ウィジェット)” を意味します。

ewoc は特定の Lisp データを表現するバッファーテキストの構築に要される情報を組織化します。ewoc のバッファーテキストは順番に、まず固定された *header* テキスト、次に一連のデータ要素のテキスト記述 (あなたが指定する Lisp オブジェクト)、最後に固定された *footer* テキストという 3 つのパートをもっています。具体的には ewoc は以下の情報を含んでいます:

- そのテキストが生成されたバッファー。
- バッファー内でのそのテキストの開始位置。
- ヘッダー文字列とフッター文字列。
- 2 重リンクされたノード *(nodes)* のチェーン。各ノードは以下を含む:
  - データ要素 *(data element)*。単一の Lisp オブジェクト。
  - そのチェーン内で先行と後続のノードへのリンク。
- カレントバッファー内にデータ要素値のテキスト表現を挿入する責務をもつ *pretty-printer* 関数。

通常は `ewoc-create`により ewoc を定義して、その結果の ewoc 構造体内にノードを構築するために Ewoc パッケージ内の別の関数に渡してバッファー内に表示します。バッファー内でこれが一度表示されれば、他の関数はバッファー位置とノードの対応を判断したり、あるノードのテキスト表現から別のノードのテキスト表現への移動等を行います。Section 39.20.1 [Abstract Display Functions], page 1073 を参照してください。

ノードは変数が値を保持するのと同じ方法でデータ要素をカプセル化 *(encapsulate)* します。カプセル化は通常は ewoc へのノード追加の一部として発生します。以下のようにデータ要素値を取得して、その場所に新たな値を配置することができます:

```
(ewoc-data node)
⇒ value

(ewoc-set-data node new-value)
⇒ new-value
```

データ要素値として実際の値のコンテナーであるような Lisp オブジェクト (リストまたはベクター)、または他の構造体へのインデックスも使用できます。例 (Section 39.20.2 [Abstract Display Example], page 1075 を参照) では後者のアプローチを使用しています。

データが変更された際にはバッファー内のテキストを更新したいでしょう。`ewoc-refresh`呼び出しにより全ノード、`ewoc-invalidate`を使用して特定のノード、または `ewoc-map`を使用して述語を満足するすべてのノードを更新できます。あるいは `ewoc-delete`を使用して無効なノードを削除したり、その場所に新たなノードを追加できます。ewoc からのノード削除はバッファーからそれに関連付けられたテキスト記述も同様に削除します。

39.20.1 抽象ディスプレーの関数

このセクションでは、*ewoc*と *node*は上述 (Section 39.20 [Abstract Display], page 1072 を参照) の構造体を、*data*はデータ要素として使用される任意の Lisp オブジェクトを意味します。

`ewoc-create` *pretty-printer* **&optional** *header footer nosep* [Function]
これはノード (とデータ要素) をもたない新たな ewoc を構築してリターンする。*pretty-printer* は 1 つの引数を受け取る関数であること。この引数は当該 ewoc 内で使用を計画する類のデータ要素であり、`insert`を使用してポイント位置にそのテキスト記述を挿入する (Ewoc パッケージの内部的メカニズムと干渉するために `insert-before-markers`は決して使用しない)。

ヘッダー、フッター、およびすべてのノードのテキスト記述の後には、通常は自動的に改行が挿入される。*nosep*が非 `nil`なら改行は何も挿入されない。これは ewoc 全体を単一行に表示したり、これらのノードにたいして何も行わないように *pretty-printer*をアレンジすることによりノードを不可視にするために有用かもしれない。

ewoc は作成時にカレントだったバッファー内のテキストを保守するので、`ewoc-create`呼び出し前に意図するバッファーへ切り替えること。

`ewoc-buffer` *ewoc* [Function]
これは、*ewoc*がそのテキストを保守するバッファーをリターンする。

`ewoc-get-hf` *ewoc* [Function]
これは *ewoc*のヘッダーとフッターから作成されたコンスセル `(header . footer)`をリターンする。

`ewoc-set-hf` *ewoc header footer* [Function]
これは *ewoc*のヘッダーとフッターに文字列 *header*と *footer*をセットする。

`ewoc-enter-first` *ewoc data* [Function]
`ewoc-enter-last` *ewoc data* [Function]
これらはそれぞれ *data*を新たなノードにカプセル化して、それを *ewoc*のチェーンノードの先頭または終端に配置する。

`ewoc-enter-before` *ewoc node data* [Function]
`ewoc-enter-after` *ewoc node data* [Function]
これらはそれぞれ *data*を新たなノードにカプセル化して、それを *ewoc*の *node*の前または後に追加する。

`ewoc-prev` *ewoc node* [Function]
`ewoc-next` *ewoc node* [Function]
これらはそれぞれ *ewoc*内の *node*の前または次のノードをリターンする。

`ewoc-nth` *ewoc n* [Function]
これは *ewoc*内で 0 基準のインデックス *n*で見つかったノードをリターンする。負の *n*は終端から数えることを意味する。*n*が範囲外なら `ewoc-nth`は `nil`をリターンする。

`ewoc-data` *node* [Function]
これは *node*にカプセル化されたデータを抽出してリターンする。

`ewoc-set-data` *node data* [Function]
これは *node*にカプセル化されるデータとして *data*をセットする。

`ewoc-locate` *ewoc* **&optional** *pos guess* [Function]
これはポイント (指定された場合は *pos*) を含む *ewoc*内のノードを判断して、そのノードをリターンする。*ewoc*がノードをもたなければ、`nil`をリターンする。*pos*が最初のノードの前なら最初のノード、最後のノードの後なら最後のノードをリターンする。オプションの 3 つ目の引数 *guess*は、*pos*近傍にあると思われるノードであること。これは結果を変更しないが、関数の実行を高速にする。

`ewoc-location` *node* [Function]
これは *node*の開始位置をリターンする。

`ewoc-goto-prev` *ewoc arg* [Function]
`ewoc-goto-next` *ewoc arg* [Function]
これらはそれぞれ *ewoc*内の前または次の *arg*番目のノードにポイントを移動する。すでに最初のノードにポイントがある場合、または *ewoc*が空の場合には `ewoc-goto-prev`は移動しない。また `ewoc-goto-next`が最後のノードを超えて移動すると結果は `nil`。この特殊なケースを除き、これらの関数は移動先のノードをリターンする。

`ewoc-goto-node` *ewoc node* [Function]
これは *ewoc*内の *node*の開始にポイントを移動する。

`ewoc-refresh` *ewoc* [Function]
この関数は *ewoc*のテキストを再生成する。これはヘッダーとフッターの間のテキスト、すなわちすべてのデータ要素の表現を削除して、各ノードにたいして 1 つずつ順に pretty-printer 関数を呼び出すことによりすることにより機能する。

`ewoc-invalidate` *ewoc* **&rest** *nodes* [Function]
これは `ewoc-refresh`と似ているが、*ewoc*内のノードセット全体ではなく *nodes*だけを対象とする点が異なる。

`ewoc-delete` *ewoc* **&rest** *nodes* [Function]
これは *ewoc*から *nodes*内の各要素を削除する。

`ewoc-filter` *ewoc predicate* **&rest** *args* [Function]
これは *ewoc*内の各データ要素にたいして *predicate*を呼び出して、*predicate*が `nil`をリターンしたノードを削除する。任意の *args*を *predicate*に渡すことができる。

`ewoc-collect` *ewoc predicate* **&rest** *args* [Function]
これは *ewoc*内の各データ要素にたいして *predicate*を呼び出して、*predicate*が非 `nil`をリターンしたノードのリストをリターンする。リスト内の要素はバッファー内での順序になる。任意の *args*を *predicate*に渡すことができる。

`ewoc-map` *map-function ewoc* **&rest** *args* [Function]
これは *ewoc*内の各データ要素にたいして *map-function*を呼び出して、*map-function*が非 `nil`をリターンしたノードを更新する。任意の *args*を *map-function*に渡すことができる。

39.20.2 抽象ディスプレーの例

以下は 3 つの整数からなるベクターを表すバッファー内の領域であるカラー構成 *(color components)* 表示を ewoc パッケージ内の関数を使用して、さまざまな方法で実装するシンプルな例です。

```
(setq colorcomp-ewoc nil
      colorcomp-data nil
      colorcomp-mode-map nil
      colorcomp-labels ["Red" "Green" "Blue"])

(defun colorcomp-pp (data)
  (if data
      (let ((comp (aref colorcomp-data data)))
        (insert (aref colorcomp-labels data) "\t: #x"
                (format "%02X" comp) " "
                (make-string (ash comp -2) ?#) "\n"))
    (let ((cstr (format "#%02X%02X%02X"
                        (aref colorcomp-data 0)
                        (aref colorcomp-data 1)
                        (aref colorcomp-data 2)))
          (samp " (sample text) "))
      (insert "Color\t: "
              (propertize samp 'face
                          `(foreground-color . ,cstr))
              (propertize samp 'face
                          `(background-color . ,cstr))
              "\n"))))

(defun colorcomp (color)
  "新たなバッファー内で COLOR の編集を許可する。
そのバッファーは Color Components モードになる。"
  (interactive "sColor (name or #RGB or #RRGGBB): ")
  (when (string= "" color)
    (setq color "green"))
  (unless (color-values color)
    (error "No such color: %S" color))
```

```
  (switch-to-buffer
   (generate-new-buffer (format "originally: %s" color)))
  (kill-all-local-variables)
  (setq major-mode 'colorcomp-mode
        mode-name "Color Components")
  (use-local-map colorcomp-mode-map)
  (erase-buffer)
  (buffer-disable-undo)
  (let ((data (apply 'vector (mapcar (lambda (n) (ash n -8))
                                     (color-values color))))
        (ewoc (ewoc-create 'colorcomp-pp
                           "\nColor Components\n\n"
                           (substitute-command-keys
                            "\n\\{colorcomp-mode-map}"))))
    (set (make-local-variable 'colorcomp-data) data)
    (set (make-local-variable 'colorcomp-ewoc) ewoc)
    (ewoc-enter-last ewoc 0)
    (ewoc-enter-last ewoc 1)
    (ewoc-enter-last ewoc 2)
    (ewoc-enter-last ewoc nil)))
```

この例は colorcomp-dataの変更して選択プロセスを “完了” して、それらを互いに簡便に結ぶキーマップを定義することにより (言い換えると “model/view/controller” デザインパラダイムの controller 部分)、“color selection widget” への拡張が可能です。

```
(defun colorcomp-mod (index limit delta)
  (let ((cur (aref colorcomp-data index)))
    (unless (= limit cur)
      (aset colorcomp-data index (+ cur delta)))
    (ewoc-invalidate
     colorcomp-ewoc
     (ewoc-nth colorcomp-ewoc index)
     (ewoc-nth colorcomp-ewoc -1))))

(defun colorcomp-R-more () (interactive) (colorcomp-mod 0 255 1))
(defun colorcomp-G-more () (interactive) (colorcomp-mod 1 255 1))
(defun colorcomp-B-more () (interactive) (colorcomp-mod 2 255 1))
(defun colorcomp-R-less () (interactive) (colorcomp-mod 0 0 -1))
(defun colorcomp-G-less () (interactive) (colorcomp-mod 1 0 -1))
(defun colorcomp-B-less () (interactive) (colorcomp-mod 2 0 -1))

(defun colorcomp-copy-as-kill-and-exit ()
  "color components を kill リングにコピーしてバッファーを kill。
文字列は#RRGGBB(6 桁 16 進が付加されたハッシュ) にフォーマットされる。"
  (interactive)
  (kill-new (format "#%02X%02X%02X"
                    (aref colorcomp-data 0)
                    (aref colorcomp-data 1)
                    (aref colorcomp-data 2)))
  (kill-buffer nil))

(setq colorcomp-mode-map
      (let ((m (make-sparse-keymap)))
        (suppress-keymap m)
```

```
(define-key m "i" 'colorcomp-R-less)
(define-key m "o" 'colorcomp-R-more)
(define-key m "k" 'colorcomp-G-less)
(define-key m "l" 'colorcomp-G-more)
(define-key m "," 'colorcomp-B-less)
(define-key m "." 'colorcomp-B-more)
(define-key m " " 'colorcomp-copy-as-kill-and-exit)
m))
```

わたしたちが決して各ノード内のデータを変更していないことに注意してください。それらのデータは ewoc 作成時に `nil`、または実際のカラーコンポーネントであるベクター `colorcomp-data`にたいするインデックスに固定されています。

39.21 カッコの点滅

このセクションではユーザーが閉カッコを挿入した際に、マッチする開カッコを Emacs が示すメカニズムを説明します。

`blink-paren-function` [Variable]
: この変数の値は閉カッコ構文 (close parenthesis syntax) の文字が挿入された際に常に呼び出される関数 (引数なし) であること。`blink-paren-function`の値は `nil`も可能であり、この場合は何も行わない。

`blink-matching-paren` [User Option]
: この変数が `nil`なら `blink-matching-open`は何も行わない。

`blink-matching-paren-distance` [User Option]
: この変数はギブアップする前にマッチするカッコをスキャンする最大の距離を指定する。

`blink-matching-delay` [User Option]
: この変数はマッチするカッコを示し続ける秒数を指定する。分数の秒も良好な結果をもたらすことがあるが、デフォルトはすべてのシステムで機能する 1 である。

`blink-matching-open` [Command]
: この関数は `blink-paren-function`のデフォルト値である。この関数は閉カッコ構文の文字の後にポイントがあると仮定して、マッチする開カッコに瞬時適切な効果を適用する。その文字がまだスクリーン上になければ、エコーエリア内にその文字のコンテキストを表示する。長い遅延を避けるために、この関数は文字数 `blink-matching-paren-distance`より遠くを検索しない。

  以下はこの関数を明示的に呼び出す例。

```
(defun interactive-blink-matching-open ()
  "ポイント前のカッコによる sexp 開始を瞬時示す"
  (interactive)
  (let ((blink-matching-paren-distance
         (buffer-size))
        (blink-matching-paren t))
    (blink-matching-open)))
```

39.22 文字の表示

このセクションでは文字が Emacs により実際に表示される方法について説明します。文字は通常はグリフ *(glyph)* として表示されます。グリフとはスクリーン上で 1 文字の位置を占めるグラフィカル

なシンボルであり、その外観はその文字自身に対応します。たとえば文字 'a' (文字コード 97) は 'a' と表示されます。しかしいくつかの文字は特別な方法で表示されます。たとえば改頁文字 (文字コード 12) は通常は 2 つのグリフのシーケンス '^L'で表示されて、改行文字 (文字コード 10) は新たなスクリーン行を開始します。

ディスプレイテーブル *(display table)* を定義することにより、各文字が表示される方法を変更できます。これはそれぞれの文字をグリフのシーケンスにマップするテーブルです。Section 39.22.2 [Display Tables], page 1079 を参照してください。

39.22.1 通常の表示の慣習

以下は各文字コードの表示にたいする慣習です (ディスプレイテーブルが存在しなければこれらの慣習をオーバーライドできる)。

- コード 32 から 126 のプリント可能 *ASCII*文字 *(printable ASCII characters:* 数字、英文字、および *'#'*のようなシンボル*)* は文字通りそのまま表示される。
- タブ文字 (文字コード 9) は次のタブストップ列まで伸長された空白文字として表示される。Section "Text Display" in *The GNU Emacs Manual* を参照のこと。変数 `tab-width`はタブストップごとのスペース数を制御する (以下参照)。
- 改行文字 (文字コード 10) は特殊効果をもつ。これは先行する行を終端して新たな行を開始する。
- 非プリント可能 *ASCII*制御文字 *(ASCII control characters)* — 文字コード 0 から 31 と `DEL`文字 (文字コード 127) — は変数 `ctl-arrow`に応じて 2 つの方法のいずれかで表示される。この変数が非 `nil` (デフォルト) なら、たとえば `DEL`にたいしては '^?'のように、これらの文字は 1 つ目のグリフが '^' ('^'のかわりに使用する文字をディスプレイテーブルで指定できる) のような 2 つのグリフのシーケンスとして表示される。

  `ctl-arrow`が `nil`なら、これらの文字は 8 進エスケープとして表示される (以下参照)。

  このルールはバッファー内に復帰文字 (CR: carriage return、文字コード 13) があればそれにも適用される。しかし復帰文字は通常はバッファーテキスト内には存在しない。これらは行末変換 (end-of-line conversion) の一部として除去される (Section 33.10.1 [Coding System Basics], page 865 を参照)。
- *raw* バイト *(raw bytes)* とはコード 128 から 255 の非 ASCII文字である。これらの文字は *8* 進エスケープ *(octal escapes)* として表示される。これは 1 つ目が '\'にたいする ASCIIコードのグリフで、残りがその文字のコードを 8 進で表した数字である (ディスプレイテーブルで '\'のかわりに使用するグリフを指定できる)。
- 255 を超える非 ASCII文字は、端末がサポートしていればそのまま表示される。端末がサポートしない場合には、その文字はグリフなし *(glyphless)* と呼ばれて、通常はプレースホルダーグリフを使用して表示される。たとえばある文字にたいしてグラフィカル端末がフォントをもたなければ、Emacs は通常は 16 進文字コードを含むボックスを表示する。Section 39.22.5 [Glyphless Chars], page 1081 を参照のこと。

上記の表示慣習はたとえディスプレイテーブルがあっても、アクティブディスプレイテーブル内のエントリーが `nil`であるようなすべての文字にたいして適用されます。したがってディスプレイテーブルのセットアップ時に指定が必要なのは特別な振る舞いを望む文字だけです。

以下の変数はスクリーン上で特定の文字が表示される方法に影響します。これらはその文字が占める列数を変更するのでインデント関数にも影響を与えます。またモードラインが表示される方法にも影響があります。新たな値を使用してモードラインを強制的に再表示するには関数 `force-mode-line-update`を呼び出してください (Section 23.4 [Mode Line Format], page 485 を参照)。

`ctl-arrow` [User Option]
このバッファーローカル変数はコントロール文字が表示される方法を制御する。非`nil`なら‘`^A`’のようにカレットとその文字、`nil`なら‘`\001`’のようにバックスラッシュと 8 進 3 桁のように 8 進エスケープとして表示される。

`tab-width` [User Option]
このバッファーローカル変数の値は Emacs バッファー内でのタブ文字表示で使用するタブストップ間のスペース数。値は列単位でデフォルトは 8。この機能はコマンド`tab-to-tab-stop`で使用されるユーザー設定可能なタブストップとは完全に無関係であることに注意。Section 32.17.5 [Indent Tabs], page 811 を参照のこと。

39.22.2 ディスプレーテーブル

ディスプレイテーブルとはサブタイプとして`display-table`をもつ特殊用途の文字テーブル (Section 6.6 [Char-Tables], page 110 を参照) であり、文字の通常の表示慣習をオーバーライドするために使用されます。このセクションではディスプレイテーブルオブジェクトの作成と調査、および要素を割り当てる方法について説明します。

`make-display-table` [Function]
これはディスプレイテーブルを作成してリターンする。テーブルは初期状態ではすべての要素に`nil`をもつ。

ディスプレイテーブルの通常の要素は文字コードによりインデックス付けされます。インデックス *c*の要素はコード *c*の表示方法を示します。値は`nil` (これは通常の表示慣習に応じて文字 *c*を表示することを意味する。Section 39.22.1 [Usual Display], page 1078 を参照)、またはグリフコードのベクター (これらのグリフとして文字 *c*を表示することを意味する。Section 39.22.4 [Glyphs], page 1081 を参照) のいずれかです。

警告**:** 改行文字の表示を変更するためにディスプレイテーブルを使用すると、バッファー全体が 1 つの長い行として表示されるでしょう。

ディスプレイテーブルは特殊用途向け 6 つのエクストラスロット *(extra slots)* をもつこともできます。以下はそれらの意味についてのテーブルです。`nil`のスロットは以下で示すそのスロットにたいするデフォルトの使用を意味します。

0 切り詰められたスクリーン行終端のグリフ (デフォルトでは‘`$`’)。Section 39.22.4 [Glyphs], page 1081 を参照のこと。グラフィカルな端末ではフリンジを無効にしていなければ (Section “Window Fringes” in *the GNU Emacs Manual* を参照)、Emacs はデフォルトでは切り詰められたことをフリンジ内の矢印で示し、ディスプレイテーブルは使用しない。

1 継続行終端のグリフ (デフォルトは‘`\`’)。グラフィカルな端末ではフリンジを無効にしていなければ、デフォルトでは Emacs は継続ををフリンジ内の曲矢印で示し、ディスプレイテーブルは使用しない。

2 8 進文字コードとして表示される文字を示すグリフ (デフォルトは‘`\`’)。

3 コントロール文字を示す (デフォルトは‘`^`’)。

4 不可視行があることを示すグリフのベクター (デフォルトは‘`...`’)。Section 39.7 [Selective Display], page 994 を参照のこと。

5 横並びのウィンドウ間のボーダー描画に使用されるグリフ (デフォルトは‘`|`’)。Section 28.6 [Splitting Windows], page 622 を参照のこと。これは現在のところテキ

スト端末でのみ効果がある。グラフィカル端末では垂直スクロールバーがサポートされていて使用中ならスクロールバーが 2 つのウィンドウを分割する。垂直スクロールバーとディバイダー (Section 39.15 [Window Dividers], page 1042 を参照) がなければ、Emacs は境界を示すために細いラインを使用する。

たとえば以下は関数 `make-glyph-code`にたいして `ctl-arrow`に非 `nil`をセットして得られる効果を模倣するディスプレイテーブル (Section 39.22.4 [Glyphs], page 1081 を参照のこと) を構築する例です:

```
(setq disptab (make-display-table))
(dotimes (i 32)
  (or (= i ?\t)
      (= i ?\n)
      (aset disptab i
            (vector (make-glyph-code ?^ 'escape-glyph)
                    (make-glyph-code (+ i 64) 'escape-glyph)))))
(aset disptab 127
      (vector (make-glyph-code ?^ 'escape-glyph)
              (make-glyph-code ?? 'escape-glyph))))
```

`display-table-slot` *display-table slot* [Function]
この関数は *display-table*のエクストラスロット *slot*の値をリターンする。引数 *slot*には 0 から 5 の数字 (両端を含む)、またはスロット名 (シンボル) を指定できる。有効なシンボルは `truncation`、`wrap`、`escape`、`control`、`selective-display`、`vertical-border`。

`set-display-table-slot` *display-table slot value* [Function]
この関数は *display-table*のエクストラスロット *slot*に *value*を格納する。引数 *slot*には 0 から 5 の数字 (両端を含む)、またはスロット名 (シンボル) を指定できる。有効なシンボルは `truncation`、`wrap`、`escape`、`control`、`selective-display`、`vertical-border`。

`describe-display-table` *display-table* [Function]
この関数はヘルプバッファーにディスプレイテーブル *display-table*の説明を表示する。

`describe-current-display-table` [Command]
このコマンドはヘルプバッファーにカレントディスプレイテーブルの説明を表示する。

39.22.3 アクティブなディスプレーテーブル

ウィンドウはそれぞれディスプレイテーブルを指定でき、各バッファーもディスプレイテーブルを指定できます。もしウィンドウにディスプレイテーブルがあれば、それはバッファーのディスプレイテーブルより優先されます。ウィンドウとバッファーがいずれもディスプレイテーブルをもたなければ、Emacs は標準的なディスプレイテーブルの使用を試みます。標準ディスプレイテーブルが `nil`なら Emacs は通常の文字表示慣習を使用します (Section 39.22.1 [Usual Display], page 1078 を参照)。

ディスプレイテーブルはモードラインが表示される方法に影響を与えるので、新たなディスプレイテーブルを使用してモードラインを強制的に再表示するには `force-mode-line-update`を使用することに注意してください (Section 23.4 [Mode Line Format], page 485 を参照)。

`window-display-table` **&optional** *window* [Function]
この関数は *window*のディスプレイテーブル、ディスプレイテーブルがなければ `nil`をリターンする。*window*のデフォルトは選択されたウィンドウ。

`set-window-display-table` *window table* [Function]
この関数は *window*のディスプレイテーブルに *table*をセットする。引数 *table*はディスプレイテーブルか `nil`のいずれかであること。

`buffer-display-table` [Variable]
この変数はすべてのバッファーにおいて自動的にバッファーローカルになる。変数の値はバッファーのディスプレイテーブルを指定する。これが `nil`ならバッファーのディスプレイテーブルは存在しない。

`standard-display-table` [Variable]
この変数の値はウィンドウ内にバッファーを表示する際、ウィンドウディスプレイテーブルとバッファーディスプレイテーブルのいずれも定義されていないとき、または Emacs がテキストを標準出力やエラーストリームに出力しているときにに Emacs が使用する標準ディスプレイテーブル (standard display table)。デフォルトが通常は `nil`だとしても、curved quotes を表示できない端末でのインタラクティブなセッションでは、デフォルトで curved quotes を ASCII 近似文字にマップする。Section 24.4 [Text Quoting Style], page 524 を参照のこと。

`disp-table`ライブラリーでは、標準ディスプレイテーブルを変更するために、いくつかの関数を定義されています。

39.22.4 グリフ

グリフ *(glyph)* とはスクリーン上で 1 文字を占めるグラフィカルなシンボルです。各グリフは Lisp 内でグリフコード *(glyph code)* として表現されます。これは文字と、表示するフェイスをオプションで指定します (Section 39.12 [Faces], page 1011 を参照)。ディスプレイテーブル内でのエントリーとしての使用がグリフコードの主な用途です (Section 39.22.2 [Display Tables], page 1079 を参照)。以下の関数はグリフコードを操作するために使用されます:

`make-glyph-code` *char* **&optional** *face* [Function]
この関数は文字 *char*を表すグリフをフェイス *face*でリターンする。*face*が省略か `nil`ならグリフはデフォルトフェイスを使用して、その場合にはグリフコードは整数。*face*が非 `nil`ならグリフコードが整数オブジェクトである必要はない。

`glyph-char` *glyph* [Function]
この関数はグリフコード *glyph*の文字をリターンする。

`glyph-face` *glyph* [Function]
この関数はグリフコード *glyph*のフェイス、または *glyph*がデフォルトフェイスを使用する場合には `nil`をリターンする。

39.22.5 グリフなし文字の表示

グリフ無し文字 *(glyphless characters)* とは literal に表示されるのではなく特別な方法、すなわち 16 進コードを中に含むボックスとして表示される文字です。これらの文字にはグリフが無いと明示的に定義された文字や、利用可能なフォントがない文字 (グラフィカルなディスプレイ)、その端末のコーディングシステムではエンコードできない文字 (テキスト端末) が同様に含まれます。

`glyphless-char-display` [Variable]
この変数の値はグリフ無し文字と表示方法を定義する文字テーブル。エントリーはそれぞれ以下の表示メソッドのいずれかでなければならない:

`nil` 通常の方法でその文字を表示する。

`zero-width`
: その文字を表示しない。

`thin-space`
: グラフィカルな端末では幅が 1 ピクセル、テキスト端末では幅が 1 文字の狭いスペース。

`empty-box`
: 空のボックスを表示する。

`hex-code` その文字の Unicode コードポイントの 16 進表記を含むボックスを表示する。

ASCII文字列
: その文字列を含むボックスを表示する。文字列には最大で 6 文字が含まれること。

コンスセル (*graphical* . *text*)
: グラフィカルな端末では *graphical*、テキスト端末では *text*をで表示する。*graphical*と *text*はいずれも上述した表示メソッドのいずれかでなければならない。

`thin-space`、`empty-box`、`hex-code`、および ASCII文字列は `glyphless-char`フェイスで描画される。テキスト端末ではボックスは square brackets '`[]`'でエミュレートされる。

文字テーブルには利用可能なすべてのフォントでも表示できない、またはその端末のコーディングシステムでエンコードできないすべての文字の表示方法を定義する余分なスロットが 1 つある。その値は上述した表示メソッドのうち `zero-width`とコンスセル以外のいずれかでなければならない。

アクティブなディスプレイテーブル内に非 `nil`なエントリーをもつ文字では、そのディスプレイテーブルが効果をもつ。この場合には Emacs は `glyphless-char-display`をまったく参照しない。

`glyphless-char-display-control` [User Option]

このユーザーオプションは似かよった文字のグループにたいして `glyphless-char-display` をセットする便利な手段を提供する。Lisp コードからこの値を直接セットしてはならない。`glyphless-char-display`更新するカスタム関数`:set`を通じた場合のみ値は効果をもつ。

この値は要素が (*group* . *method*)であるような alist であること。ここで *group*は文字のグループを指定するシンボル、*method*はそれらを表示する方法を指定するシンボル。

*group*は以下のいずれかであること:

`c0-control`
: 改行文字とタブ文字を除く `U+0000`から `U+001F`までの ASCIIコントロール文字 (通常は '`^A`'のようなエスケープシーケンスとして表示される。Section "How Text Is Displayed" in *The GNU Emacs Manual* を参照)。

`c1-control`
: `U+0080`から `U+009F`までの非 ASCII、非プリント文字 (通常は '`\230`'のような 8 進エスケープシーケンスとして表示される)。

`format-control`
: U+200E LEFT-TO-RIGHT MARK のような Unicode General Category [Cf] の文字だが、U+00AD SOFT HYPHEN のようにグラフィックイメージをもつ文字を除く。

`no-font` 適切なフォントが存在しない、あるいはその端末のコーディングシステムではエンコードできない文字。

*method*シンボルは `zero-width`、`thin-space`、`empty-box`、`hex-code`のいずれかであること。これらは上述の `glyphless-char-display`での場合と同様の意味をもつ。

39.23 ビープ

このセクションではユーザーの注意を喚起するために、Emacs でベルを鳴らす方法を説明します。これを行う頻度は控え目にしてください。頻繁なベルは刺激過剰になる恐れがあります。同様にエラーのシグナル時に過度にビープ音を使用しないよう注意してください。

`ding` **&optional** *do-not-terminate* [Function]
この関数はビープ音を鳴らす、またはスクリーンをフラッシュする (後述の `visible-bell`を参照)。*do-not-terminate*が `nil`なら、この関数はカレントで実行中のキーボードマクロも終了する。

`beep` **&optional** *do-not-terminate* [Function]
これは `ding`のシノニム。

`visible-bell` [User Option]
この変数はベルを表すためにスクリーンをフラッシュすべきかどうかを決定する。非 `nil`ならフラッシュして、`nil`ならフラッシュしない。これはグラフィカルなディスプレイで効果的であり、テキスト端末ではその端末の Termcap エントリーが可視ベル (visible bell) '`vb`'の能力を定義する。

`ring-bell-function` [User Option]
これが非 `nil`なら Emacs がどのようにベルを鳴らすかを定義すること。値は引数なしの関数であること。これが非 `nil`なら `visible-bell`より優先される。

39.24 ウィンドウシステム

Emacs は複数のウィンドウシステムで機能しますが、特に X ウィンドウシステムにおいてもっとも機能します。Emacs と X はどちらも "ウィンドウ" を使用しますが異なる使い方をします。Emacs のフレームは X においては単一のウィンドウです。Emacs の個々のウィンドウについては、X はまったく関知しません。

`window-system` [Variable]
この端末ローカルな変数は、Emacs がフレームを表示するのに何のウィンドウシステムを使用しているかを示す。可能な値は、

`x` Emacs は X を使用してフレームを表示している。

`w32` Emacs はネイティブ MS-Windows GUI を使用してフレームを表示している。

`ns` Emacs は Nextstep インターフェイスを使用してフレームを表示している (GNUstep と macOS で使用されている)。

`pc` Emacs は MS-DOS のスクリーン直接書き込みを使用してフレームを表示している。

`nil` Emacs は文字ベース端末を使用してフレームを表示している。

`initial-window-system` [Variable]
この変数はスタートアップの間に Emacs が作成する最初のフレームにたいして使用される `window-system`の値を保持する (デーモンとして Emacs を呼び出し時には初期フレームを作成しないので、`w32`の MS-Windows を除き `initial-window-system`は `nil`。Section "Initial Options" in *The GNU Emacs Manual* を参照)。

`window-system` **&optional** *frame* [Function]
この関数は *frame*を表示するために使用されているウィンドウシステムを示す名前のシンボルをリターンする。この関数がリターンし得るシンボルのリストは変数 `window-system`の記述と同様。

テキスト端末とグラフィカルなディスプレイで異なる処理を行うコードを記述したいときは、`window-system`と `initial-window-system`を述語やブーリーンフラグ変数として使用しないでください。これは与えられたディスプレイタイプでの Emacs の能力指標として `window-system`が適していないからです。かわりに `display-graphic-p`、または Section 29.25 [Display Feature Testing], page 751 で説明しているその他の述語 `display-*-p`を使用してください。

39.25 ツールチップ

ツールチップ *(Tooltips)* はマウスポインターのカレント位置に関連するヘルプ的なヒント (別名 "tips") の表示に使用される特別なフレームです (Chapter 29 [Frames], page 696 を参照)。Emacs はテキストのアクティブ範囲 (Section 32.19.4 [Special Properties], page 820 を参照)、およびメニューアイテム (Section 22.17.1.2 [Extended Menu Items], page 448 を参照) やツールバーのボタン (Section 22.17.6 [Tool Bar], page 455 を参照) のような種々の UI 要素に関するヘルプ文字列の表示にツールチップを使用します。

`tooltip-mode` [Function]
Tooltip モードはツールチップの表示を有効にするマイナーモード。このモードをオフにするとツールチップはエコーエリアに表示される。テキストモード (別名 "TTY") のフレームでは、ツールチップは常にエコーエリアに表示される。

GTK+サポート付きで Emacs がビルドされた際にはデフォルトで GTK+関数を使用してツールチップを表示して、ツールチップの外観は GTK+のセッティングにより制御されます。GTK+ツールチップは変数 `x-gtk-use-system-tooltips`の値を `nil`に変更して無効にできます。このセクションの残りでは Emacs 自身が提供する非 GTK+ツールチップを制御する方法を説明します。

ツールチップは独自のパラメーターをもつツールチップフレームと呼ばれる特別なフレームに表示されます (Section 29.4 [Frame Parameters], page 712 を参照)。他のフレームとは異なり、ツールチップフレームのデフォルトパラメターは特別な変数に格納されています。

`tooltip-frame-parameters` [User Option]
このカスタマイズ可能なオプションはツールチップ表示に使用するデフォルトのフレームパラメーターを保持する。フォントとカラーに関するパラメーターは無視して、`tooltip`フェイスの対応する属性をかわりに使用する。`left`や `top`のパラメーターが含まれていれば、ツールチップを表示すべきフレームに相対的な絶対座標として使用する (Section "Tooltips" in *The GNU Emacs Manual* に記された変数を使用すればマウスに相対的なツールチップをカスタマイズできる)。`left`と `top`のパラメーターが与えられた場合にはマウスに相対的なオフセットをオーバーラードすることに注意。

`tooltip`フェイスはツールチップ内に表示されるテキストの見栄えを決定します。デフォルトのフレームフォントより一般的にはサイズの小さい可変ピッチフォントの使用が必要になります。

`tooltip-functions` [Variable]
これは Emacs がツールチップの表示を必要とする際に呼び出す関数のリストであるようなアブノーマルフック。関数はそれぞれ最後のマウス移動イベントである *event*を単一の引数として呼び出される。このリスト上の関数が実際にツールチップを表示するなら非 `nil`をリターンして、残りの関数は呼び出されない。この変数のデフォルト値は `tooltip-help-tips`という 1 つの関数。

`tooltip-functions`のリストに配置する関数を独自に記述する場合には、ツールチップの表示をトリガーしたマウスイベントのバッファーを知る必要があるかもしれません。以下はこの情報を提供する関数です。

`tooltip-event-buffer` *event* [Function]
この関数は *event*が発生したバッファーをリターンする。テキストがツールチップをトリガーしたバッファーを取得するために、これを `tooltip-functions`の関数の引数で呼び出す。イベントはバッファーではないところ (たとえばツールバー) で発生したかもしれず、そのような場合にはこの関数は `nil`をリターンする。

ツールチップ表示に関する他の側面は、いくつかのカスタマイズ可能なセッティングにより制御されます。Section “Tooltips” in *The GNU Emacs Manual* を参照してください。

39.26 双方向テキストの表示

Emacs はアラビア語、ペルシア語、ヘブライ語のような水平方向テキストの自然な表示順が R2L(right-to-left: 右から左) に実行されるようなスクリプトで記述されたテキストを表示できます。さらに L2R(right-to-left: 左から右) のテキストに埋め込まれた R2L スクリプト (例: プログラムソースファイル内のアラビア語やヘブライ語のコメント) は適宜右から左に R2L に表示される一方、ラテンスクリプト部や R2L テキストに埋め込まれた数字は L2R で表示されます。そのような L2R と R2L が混交されたテキストを、わたしたちは双方向テキスト *(bidirectional text)* と呼んでいます。このセクションでは双方向テキストの編集と表示にたいする機能とオプションについて説明します。

テキストはロジカルな順序 (または読込順)、すなわち人間が各文字を読み込むであろう順序でテキストを Emacs バッファーや文字列に格納します。R2L および双方向テキストでは、スクリーン上で文字が表示される順序 (ビジュアル順と呼ばれる) はロジカル順と同一ではありません。それら各文字のスクリーン位置は文字列やバッファー位置により単調に増加しません。この双方向の並べ替え *(bidirectional reordering)* を処理を行う際に、Emacs は Unicode 双方向アルゴリズム (UBA : Unicode Bidirectional Algorithm) にしたがいます (`http://www.unicode.org/reports/tr9/`)。Emacs は Unicode Standard v9.0 の要件に合致する UBAの “Full Bidirectionality(完全な双方向性)” を提供します。とはいえテキストがパラグラフの基本方向と逆方向なときに Emacs が継続行を表示する方法は、表示するテキストの再並び換えの前に行の折り返しを要求する UBA からは逸脱していることに注意してください。

`bidi-display-reordering` [Variable]
このバッファーローカル変数の値が非 `nil` (デフォルト) なら、Emacs は表示で双方向の並べ替えを行う。この並べ替えはバッファーテキスト、同様に文字列表示やバッファー内のテキストプロパティやオーバーレイプロパティ由来のオーバーレイ文字列に効果を及ぼす (Section 39.9.2 [Overlay Properties], page 1002 および Section 39.16 [Display Property], page 1043 を参照)。値が `nil`なら Emacs はバッファー内での双方向の並べ替えを行わない。

`bidi-display-reordering`のデフォルト値は、モードライン内に表示されるテキスト (Section 23.4 [Mode Line Format], page 485 を参照)、およびヘッダー行 (Section 23.4.7

[Header Lines], page 493 を参照) を含む、バッファーにより直接提供されない文字列の並べ替えを制御する。

たとえバッファーの `bidi-display-reordering`が非 `nil`でも、Emacs がユニバイトバッファーのテキストの並べ替えを行うことはありません。これはユニバイトバッファーに含まれるのが文字ではなく raw バイトであり、並べ替えに要する方向的なプロパティを欠くからです。したがってあるバッファーのテキストが並べ替えられるかどうかテストするには、`bidi-display-reordering`のテスト単独では不十分です。正しいテストは以下のようになります:

```
(if (and enable-multibyte-characters
         bidi-display-reordering)
    ;; 表示時にバッファーは並べ替えられるだろう
  )
```

とはいえ親バッファーが並べ替えられた際には、ユニバイト表示やオーバーレイ文字列は並べ替えられます。これは Emacs によりプレーン ASCII 文字列がユニバイト文字列に格納されるからです。ユニバイト表示やオーバーレイ文字列が非 ASCII 文字列を含むなら、それらの文字は L2R の方向をもつとみなされます。

テキストプロパティ`display`、値が文字列であるような `display`プロパティによるオーバーレイ、バッファーテキストを置換するその他任意のプロパティにカバーされたテキストは表示時の並べ替えの際には単一の単位として扱われます。つまりこれらのプロパティにカバーされたテキストの chunk 全体が一緒に並べ替えられます。さらにそのようなテキスト chunk 内の文字の双方向的なプロパティは無視されて、Emacs はあたかもそれらがオブジェクト置換文字 *(Object Replacement Character)* として知られる単一文字で置換されたかのように並べ替えます。これはテキスト範囲上に display プロパティを配置することにより、表示時に周辺テキストを並べ替える方法が変更され得ることを意味しています。このような予期せぬ効果を防ぐには、常に周辺テキストと等しい方向のテキストにたいしてそのようなプロパティを配置してください。

双方向テキストのパラグラフはそれぞれ、R2L か L2R いずれかの基本方向 *(base direction)* をもちます。L2R パラグラフはウィンドウの左マージンを先頭に表示され、そのテキストが右マージンに達したら切り詰めや継続されます。R2L パラグラフはウィンドウの右マージンを先頭に表示され、そのテキストが左マージンに達したら切り詰めや継続されます。

Emacs の UBA実装の目的におけるパラグラフの開始および終了の正確な位置は、以下の 2 つのローカル変数により決定されます (`paragraph-start`と `paragraph-separate`に効果はないことに注意)。デフォルトではこれらの変数は `nil`であり、パラグラフは空行 (改行を後に併なう 0 個以上の空白文字) で囲まれます。

`bidi-paragraph-start-re` [Variable]
この変数の値が非 `nil`ならパラグラフの開始か 2 つのパラグラフを分割する行にマッチする正規表現であること。この正規表現は常に改行の後にマッチするので、それをアンカーにする (`"^"` で開始する) のが最善である。

`bidi-paragraph-separate-re` [Variable]
この変数の値が非 `nil`なら 2 つのパラグラフを分割する行にマッチする正規表現であること。この正規表現は常に改行の後にマッチするので、それをアンカーにする (`"^"`で開始する) のが最善である。

これら 2 つの変数のいずれかを変更する場合には、整合性のあるパラグラフの記述を保証するために、通常は両方を変更するべきです。たとえば双方向の並べ替え目的のために各改行を新たなパラグラフの開始とするには両方の変数に`"^"`をセットしてください。

デフォルトでは Emacs はテキスト先頭を調べることにより各パラグラフの基本方向を判断します。基本方向の精細な決定手法は UBAにより指定されており、簡潔に言うとその明示にな方向生をもつそのパラグラフ内の最初の文字がパラグラフの基本方向を決定します。とはいえ、あるバッファーが自身のパラグラフにたいして特定の基本方向の強制を要する場合もあります。たとえばプログラムソースコードを含むバッファーは、すべてのパラグラフが L2R で表示されるよう強制されるべきでしょう。これを行うために以下の変数を使用できます:

`bidi-paragraph-direction` [User Option]

このバッファーローカル変数の値が `right-to-left`か `left-to-right`いずれかのシンボルなら、そのバッファー内のすべてのパラグラフがその指定された方向をもつとみなされる。その他すべての値は `nil` (デフォルト) と等価であり、それは各パラグラフの基本方向が内容により判断されることを意味する。

プログラムソースコードにたいするモードは、これを `left-to-right`にセットすること。Prog モードはデフォルトでこれを行うので、Prog モードから派生したモードは明示的にセットする必要はない (Section 23.2.5 [Basic Major Modes], page 472 を参照)。

`current-bidi-paragraph-direction` **&optional** *buffer* [Function]

この関数は *buffer*という名前のバッファーのポイント位置のパラグラフ方向をリターンする。リターンされる値は `left-to-right`か `right-to-left`いずれかのシンボルである。*buffer*が省略または `nil`の場合のデフォルトはカレントバッファー。変数 `bidi-paragraph-direction` のバッファーローカル値が非 `nil`なら、リターンされる値はその値と等しくなるだろう。それ以外ならリターンされる値は Emacs により動的に決定されたパラグラフの方向を反映する。`bidi-display-reordering`の値が `nil`のバッファー、同様にユニバイトバッファーにたいしては、この関数は常に `left-to-right`をリターンする。

バッファーのカレントのスクリーン位置にたいして、ビジュアル順に L2R か R2L いずれかの方向に厳密なポイント移動を要す場合があります。Emacs はこれを行うためのプリミティブを提供します。

`move-point-visually` *direction* [Function]

この関数は、カレントで選択されたウィンドウのバッファーにたいしてポイントを、スクリーン上ですぐ右か左のポイントへ移動する。*direction*が正ならスクリーン位置は右、それ以外ならスクリーン位置は左へ移動するだろう。周囲の双方向コンテキストに依存して、これは潜在的に多くのバッファーのポイントを移動し得ることに注意。スクリーン行終端で呼び出された場合には、この関数は *direction*に応じて適宜、次行か前行の右端か左端のスクリーン位置にポイントを移動する。

この関数は値として新たなバッファー位置をリターンする。

バッファー内で双方向の内容をもつ 2 つの文字列が並置されているときや、プログラムで 1 つのテキスト文字列に結合した場合には、双方向の並べ替えは以外かつ不快な効果を与える可能性があります。典型的な問題ケースは Buffer Menu モードや Rmail Summary モードのようにバッファーがスペースや区切り文字分割されたテキストのフィールドのシーケンスで構成されているときです。それはセパレーターとして使用されている区切り文字が弱い方向性をもち、周囲のテキストの方向を採用するためです。結果として双方向の内容のフィールドが後続する数値フィールドは、先行するフィールドへ左方向に表示され、期待したレイアウトを破壊してしまいます。この問題を回避するための方法がいくつかあります :

– 双方向の内容をもち得る各フィールド終端に、スペシャル文字 LEFT-TO-RIGHT MARK(略して LRM) の U+200E LEFT-TO-RIGHT MARK を付加する。後述の関数 `bidi-string-mark-left-to-right`は、この目的に手頃 (R2L パラグラフではかわりに

RIGHT-TO-LEFT MARK、略して RLMの U+200F RIGHT-TO-LEFT MARK を使用する)。これは UBA により推奨される解決策の 1 つである。

– フィールドセパレーターにタブ文字を含める。タブ文字は双方向の並べ替えにおいてセグメントセパレーター *(segment separator)* の役割を演じて、両側のテキストを個別に並べ替えさせる。

– `display`プロパティ、または `(space . PROPS)`という形式の値をもつオーバーレイ (Section 39.16.2 [Specified Space], page 1044 を参照) でフィールドを区切る。Emacs はこの display 仕様をパラグラフセパレーター *(paragraph separator)* として扱い両側のテキストを個別に並べ替える。

`bidi-string-mark-left-to-right` *string* [Function]

この関数は結果を安全に他の文字列に結合できるよう、あるいはこの文字列とスクリーン上で次行となる行に関連するレイアウトを乱すことなくバッファー内の他の文字列に並置できるよう、自身への引数 *string*を恐らく変更してリターンする。この関数がリターンする文字列が R2L パラグラフの一部として表示される文字列なら、それは常に後続するテキストの左に出現するだろう。この関数は自身の引数の文字を検証することにより機能して、もしそれらの文字のいずれかがディスプレイ上の並べ替えを発生し得るなら、この関数はその文字列に LRM文字を付加する。付加された LRM文字はテキストプロパティ`invisible`に `t`を与えることにより不可視にできる (Section 39.6 [Invisible Text], page 992 を参照)。

並べ替えアルゴリズムは`bidi-class`プロパティとして格納された文字の双方向プロパティを使用します (Section 33.6 [Character Properties], page 857 を参照)。Lisp プログラムは `put-char-code-property`関数を呼び出すことにより、これらのプロパティを変更できます。しかしこれを行うには UBAの完全な理解が要求されるので推奨しません。ある文字の双方向プロパティにたいする任意の変更はグローバルな効果をもちます。これらは Emacs のフレームのすべてのフレームとウィンドウに影響します。

同様に `mirroring`プロパティは並べ替えられたテキスト内の適切にミラーされた文字の表示に使用されます。Lisp プログラムはこのプロパティを変更することにより、ミラーされた表示に影響を与えることができます。繰り返しますがそのような変更は Emacs のすべての表示に影響を与えます。

スペシャル双方向制御文字 LEFT-TO-RIGHT OVERRIDE (LRO) と RIGHT-TO-LEFT OVERRIDE (RLO) をテキストに挿入することにより、文字の双方向プロパティをオーバーライドできます。RLOと改行か POP DIRECTIONAL FORMATTING (PDF) のいずれか先にある文字間のすべての文字は、それらが強い R2L であるかのように表示されます (反転して表示される)。同様に LROと PDFか改行の間のすべての文字は、それらがたとえ強い R2L であっても強い L2R であるかのように反転して表示されません。

これらのオーバーライドは、あるテキストを並び替えアルゴリズムの影響を受けずに、直接表示順を制御したいときに有用です。しかしこれらはフィッシング *(phishing)* として知られるような悪意のある用途にも使用されます。特にウェブ上の URL や email メッセージ内のリンクは真のリンク先はまったく異なるのに、ブラウザによる論理順で解釈される外観を認識不能に操作したり、何らかの著名で安全なリンク先に偽装される可能性があります。

Emacs はアプリケーションが使用するために、双方向プロパティで L2R 文字を R2L、またはその逆にするようにオーバーライドされたテキストのインスタンスを検知するプリミティブを提供します。

`bidi-find-overridden-directionality` *from to* **&optional** *object* [Function]

この関数は *object*で指定されたテキストの *from* (含む) と *to* (含まず) の間のテキストを調べて R2L の文字であるかのように表示が強制されている双方向プロパティの強い L2R 文字、L2R の文字であるかのように表示が強制されている強い R2L 文字の最初の位置をリターンする。指定されたテキストリージョンでそのような文字が見つからなければ`nil`をリターンする。

オプション引数 *object*は検索するテキストを指定して、デフォルトはカレントバッファー。*object* が非 `nil`なら別のバッファーや文字列、またはウィンドウかもしれない。文字列ならこの関数はその文字列を検索する。ウィンドウならこの関数はそのウィンドウが表示するバッファーを検索する。検査したいテキストをもつバッファーが何らかのウィンドウに表示されていれば、この関数にバッファーを渡すのではなくそのウィンドウの指定を推奨する。これはウィンドウ固有のオーバーレイにカバーされたバッファーのテキストでは関数の結果が変化し得るが、関数にウィンドウ固有のオーバーレイを正しく考慮するように指示するからである。

テキストが R2L 文字と L2R 文字の混交を含み、かつ双方向制御が別の場所にコピーされる際には、その視覚的外見は変化するかもしれず、コピー先の周辺テキストの視覚的外見にも影響するかもしれません。これは UBAで指定される双方向テキストの並び替えでは、コピーされるテキストとそれを取り囲む周辺テキストの両方が非自明かつコンテキスト依存の効果をもつからです。

コピーされるテキストとコピー先周辺のテキストの視覚的外見を Lisp プロパティが保証することが必要なときがあるかもしれません。この効果を達成するために Lisp プログラムは以下の関数を使用できます。

`buffer-substring-with-bidi-context` *start end* **&optional** *no-properties* [Function]

この関数は `buffer-substring` (Section 32.2 [Buffer Contents], page 778 を参照) と同様に機能するが、テキストが別の場所にコピーされる際に視覚的外見を保つために必要な双方向制御文字を前や後に付加する点が異なる。オプション引数 *no-properties*が非 `nil`なら、それはテキストのコピーからテキストプロパティを削除することを意味する。

40 オペレーティングシステムのインターフェース

これは Emacs の開始と終了、オペレーティングシステム内の値へのアクセス、端末の入力と出力に関するチャプターです。

関連する情報は Section E.1 [Building Emacs], page 1175 を参照してください。端末とスクリーンに関連するオペレーティングシステムの状態に関する追加情報は Chapter 39 [Display], page 981 を参照してください。

40.1 Emacs のスタートアップ

このセクションでは Emacs が開始時に何を行うか、およびそれらのアクションのカスタマイズ方法を説明します。

40.1.1 要約: スタートアップ時のアクション順序

Emacs は起動時に以下の処理を行います (`startup.el`内の `normal-top-level`を参照):

1. この `load-path`の各ディレクトリー内にある `subdirs.el`という名前のファイルを実行して `load-path`にサブディレクトリーを追加する。このファイルは通常はそのディレクトリー内にあるサブディレクトリーをこのリスト変数に追加して、それらを順次スキャンする。ファイル `subdirs.el`は通常は Emacs インストール時に自動的に作成される。
2. `load-path`のディレクトリー内で見つかった `leim-list.el`をすべてロードする。このファイルは入力メソッドの登録を意図している。この検索はユーザーが作成するかもしれない個人的な `leim-list.el`すべてにたいしてのみ行われる。標準的な Emacs ライブラリーを含むディレクトリーはスキップされる (これらは単一の `leim-list.el`だけに含まれるべきであり Emacs 実行形式にコンパイル済)。
3. 変数 `before-init-time`に `current-time`の値をセットする (Section 40.5 [Time of Day], page 1104 を参照)。これは `after-init-time`に `nil`をセットすることにより Emacs 初期化時に Lisp プログラムへの合図も行う。
4. `LANG`のような環境変数がそれを要するなら言語環境と端末のコーディングシステムをセットする。
5. コマンドライン引数にたいして基本的なパースをいくつか行う。
6. ユーザーの▼早期 init ファイルをロードする Section “Early Init File” in *The GNU Emacs Manual*(を参照)。これはオプション ‘`-q`’、‘`-Q`’、または ‘`--batch`’が指定されていたら行われない。‘`-u`’オプションが指定されたら Emacs はかわりにそのユーザーのホームディレクトリー内で init ファイルを探す。
7. インストール済みのオプションの Emacs Lisp パッケージをすべてアクティブ化するために関数 `package-activate-all`を呼び出す。Section 41.1 [Packaging Basics], page 1134 を参照のこと。しかし `package-enable-at-startup`が `nil`、または ‘`-q`’、‘`-Q`’、‘`--batch`’のいずれかのオプションで開始時には、Emacs はパッケージのアクティブ化をしない。後者のケースでパッケージをアクティブ化するには、(たとえば ‘`--funcall`’オプションを通じて) 明示的に `package-activate-all`を呼び出すこと。
8. batch モードで実行されていなければ変数 `initial-window-system`が指定するウィンドウシステムを初期化する (Section 39.24 [Window Systems], page 1083 を参照)。初期化関数 `window-system-initialization`はジェネリック関数 *generic function* であり、本当の実装はサポートされる各ウィンドウシステムごとに異なる (Section 13.8 [Generic Functions], page 222 を参照)。`initial-window-system`の値が *windowsystem*なら、ファイル `term/`*windowsystem*`-win.el`内で適切な初期化関数の実装が定義されている。このファイルはビルド時に Emacs 実行可能形式にコンパイルされているはずである。

9. ノーマルフック `before-init-hook`を実行する。
10. それが適切ならグラフィカルなフレームを作成する。グラフィカルなフレーム作成の一環として `initial-frame-alist`と `default-frame-alist` (Section 29.4.2 [Initial Parameters], page 713 を参照) により指定されたウィンドウシステム用の `window-system-initialization`関数を呼び出すことにより、そのウィンドウシステムのグラフィカルなフレームを初期化する。これは (非インタラクティブな)batch モードやデーモンモードでは行われない。
11. 初期フレームのフェイスを初期化して必要ならメニューバーとツールバーをセットする。グフィカルなフレームがサポートされていたら、たとえカレントフレームがグラフィカルでなくても、後でグラフィカルなフレームが作成されるかもしれないのでツールバーをセットアップする。
12. リスト `custom-delayed-init-variables`内のメンバーを再初期化するために `custom-reevaluate-setting`を使用する。これらのメンバーは、デフォルト値がビルド時ではなく実行時のコンテキストに依存する、すべての事前ロード済ユーザーオプションである。Section E.1 [Building Emacs], page 1175 を参照のこと。
13. 存在すればライブラリー `site-start`をロードする。これはオプション '`-Q`'か'`--no-site-file`'が指定された場合は行われない。
14. ユーザーの init ファイルをロードする (Section 40.1.2 [Init File], page 1093 を参照)。これはオプション '`-q`'、'`-Q`'、または '`--batch`'が指定されていたら行われない。'`-u`'オプションが指定されたら Emacs はかわりにそのユーザーのホームディレクトリー内で init ファイルを探す。
15. 存在すればライブラリー `default`をロードする。これは `inhibit-default-init`が非 `nil`、あるいはオプション '`-q`'、'`-Q`'、または '`--batch`'指定された場合には行われない。
16. もしファイルが存在して読み込み可能なら、`abbrev-file-name`で指定されるファイルからユーザーの abbrev をロードする (Section 36.3 [Abbrev Files], page 924 を参照)。オプション '`--batch`'が指定されていたら行われない。
17. 変数 `after-init-time`に `current-time`の値をセットする。この変数は事前に `nil`にセットされている。これをカレント時刻にセットすることが初期化フェーズが終わったことの合図となり、かつ `before-init-time`と共に用いることにより初期化に要した時間の計測手段を提供する。
18. ノーマルフック `after-init-hook`を実行する。
19. バッファー`*scratch*`が存在して、まだ (デフォルトであるべき)Fundamental モードなら `initial-major-mode`に応じたメジャーモードをセットする。
20. テキスト端末で開始された場合には、端末固有の Lisp ライブラリー (Section 40.1.3 [Terminal-Specific], page 1095 を参照) をロードしてフック `tty-setup-hook`を実行する。これは`--batch`モード、または `term-file-prefix`が `nil`なら実行されない。
21. `inhibit-startup-echo-area-message`で抑制していなければエコーエリアに初期メッセージを表示する。
22. これ以前に処理されていないコマンドラインオプションをすべて処理する。
23. オプション`--batch`が指定されていたら、ここで exit する。
24. `*scratch*`が存在して空ならばバッファーに `(substitute-command-keys initial-scratch-message)`を挿入する。
25. `initial-buffer-choice`が文字列ならその名前のファイル (かディレクトリー) を visit する。関数なら引数なしでその関数を呼び出して、それがリターンしたバッファーを選択する。コマンドライン引数として単一のファイルが与えられた場合にはファイルを visit して、そのバッファーを `initial-buffer-choice`のそばに表示する。複数のファイルが与えられた場合にはすべて

のファイルを visit して、`initial-buffer-choice`のそばに`*Buffer List*`バッファーを表示する。

26. `emacs-startup-hook`を実行する。
27. init ファイルの指定が何であれ、それに応じて選択されたフレームのパラメーターを変更する`frame-notice-user-settings`を呼び出す。
28. `window-setup-hook`を実行する。このフックと `emacs-startup-hook`の違いは前述したフレームパラメーターの変更後にこれが実行される点のみ。
29. copyleft と Emacs の基本的な使い方を含んだ特別なバッファースタートアップスクリーン *(startup screen)* を表示する。これは `inhibit-startup-screen`か `initial-buffer-choice`が非 `nil`、あるいはコマンドラインオプション '`--no-splash`'か '`-Q`'が指定されていたら行われない。
30. デーモンが要求された場合には`server-start`を呼び出す (POSIX システムではバックグラウンドのデーモンが要求された場合には制御端末からデタッチされる)。Section "Emacs Server" in *The GNU Emacs Manual* を参照のこと。
31. セッションマネージャーにより開始された場合には、以前のセッションの ID を引数として`emacs-session-restore`を呼び出す。Section 40.18 [Session Management], page 1122 を参照のこと。

以下のオプションはスタートアップシーケンスにおけるいくつかの側面に影響を与えます。

`inhibit-startup-screen` [User Option]

この変数が非`nil`ならスタートアップスクリーンを抑制する。この場合には Emacs は通常は`*scratch*`バッファーを表示する。しかし以下の `initial-buffer-choice`を参照されたい。

新しいユーザーが copyleft や Emacs の基本的な使い方に関する情報を入手するのを防げるので、新しいユーザーの init ファイル内や複数ユーザーに影響するような方法でこの変数をセットしてはならない。

`inhibit-startup-message`と `inhibit-splash-screen`はこの変数にたいするエイリアス。

`initial-buffer-choice` [User Option]

非`nil`ならこの変数はスタートアップ後にスタートアップスクリーンのかわりに Emacs が表示するファイルを指定する文字列であること。この変数が関数なら Emacs はその関数を呼び出して、その関数はその後に表示するバッファーをリターンしなければならない。値が `t`なら Emacs は`*scratch*`バッファーを表示する。

`inhibit-startup-echo-area-message` [User Option]

この変数はエコーエリアのスタートアップメッセージの表示を制御する。ユーザーの init ファイル内に以下の形式のテキストを追加することによりエコーエリアのスタートアップメッセージを抑制できる:

```
(setq inhibit-startup-echo-area-message
      "your-login-name")
```

Emacs はユーザーの init ファイル内で上記のような式を明示的にチェックする。ユーザーのログイン名は Lisp の文字列定数としてこの式内に記述されていなければならない。Customize インターフェイスを使用することもできる。他の方法で同じ値に `inhibit-startup-echo-area-message`をセットしてもスタートアップメッセージは抑制されない。この方法により望むならユーザー自身で簡単にメッセージを抑制できるが、単に自分用の ini ファイルを別のユーザーにコピーしてもメッセージは抑制されないだろう。

`initial-scratch-message` [User Option]
この変数が非`nil`なら Emacs スタートアップ時に`*scratch*`バッファーに挿入するドキュメントとして扱われる文字列であること。`nil`なら`*scratch*`バッファーは空になる。

以下のコマンドラインオプションはスタートアップシーケンスにおけるいくつかの側面に影響を与えます。Section “Initial Options” in *The GNU Emacs Manual* を参照してください。

`--no-splash`
スプラッシュスクリーンを表示しない。

`--batch` 対話的な端末なしで実行する。Section 40.17 [Batch Mode], page 1121 を参照のこと。

`--daemon`
`--bg-daemon`
`--fg-daemon`
表示の初期化を何も行わず単にサーバーを開始する (“バックグラウンド” のデーモンは自動的にバックグラウンドで実行される)。

`--no-init-file`
`-q` init ファイルと`default`ライブラリーをいずれもロードしない。

`--no-site-file`
`site-start`ライブラリーをロードしない。

`--quick`
`-Q` ‘`-q --no-site-file --no-splash`’と等価。

40.1.2 init ファイル

Emacs の開始時は通常はユーザーの *init* ファイル *(init file)* のロードを試みます。これはユーザーのホームディレクトリー内にある`.emacs`か`.emacs.el`という名前のファイル、あるいはホームディレクトリーの`.emacs.d`という名前のサブディレクトリー内にある`init.el`という名前のファイルのいずれかのファイルです。

コマンドラインスイッチ‘`-q`’、‘`-Q`’、‘`-u`’は init ファイルを探すべきか、およびどこで探すべきかを制御します。‘`-u` *user*’はそのユーザーではなく *user*の init ファイルのロードを指示しますが、‘`-q`’ (‘`-Q`’のほうが強力) は init ファイルをロードしないことを指示します。Section “Entering Emacs” in *The GNU Emacs Manual* を参照してください。いずれのオプションも指定されていなければユーザーのホームディレクリーから init ファイルを探すために、Emacs は環境変数`LOGNAME`、`USER` (ほとんどのシステム)、または`USERNAME` (MS システム) を使用します。この方法によりたとえ su していたとしても、依然として Emacs はそのユーザー自身の init ファイルをロードできるのです。これらの環境変数が存在していなくても Emacs はユーザー ID からユーザーのホームディレクトリーを探します。

Emacs は早期 *init* ファイル *(early init file)* と呼ばれる 2 つ目の init ファイルが存在すれば、それのロードも試みます。これは`~/.emacs.d`にある`early-init.el`という名前のファイルです。早期 init ファイルはスタートアッププロセスのより速いタイミングでロードされるために、通常の init ファイルのロード前に初期化される何かをカスタマイズするために使用できるのが早期 init ファイルと通常の init ファイルの違いです。たとえば *package-load-list*や *package-enable-at-startup*のような変数をセットしてパッケージシステムの初期化プロセスをカスタマイズできます。Section “Package Installation” in *The GNU Emacs Manual* を参照してください。

インストールした Emacs によっては`default.el`という Lisp ライブラリーのデフォルト *init* ファイル *(default init file)* が存在するかもしれません。Emacs はライブラリーの標準検索パスからこの

ファイルを探します (Section 16.1 [How Programs Do Loading], page 269 を参照)。このファイルは Emacs ディストリビューション由来ではありません。このファイルはローカルなカスタマイズを意図しています。デフォルト init ファイルが存在する場合には常にこのファイルが Emacs 開始時にロードされます。しかしユーザー自身の init ファイルが存在する場合にはそれが最初にロードされます。それにより `inhibit-default-init`が非 `nil`値にセットされた場合には、Emacs は後続する `default.el`ファイルのロードを行いません。batch モードまたは '`-q`' (または '`-Q`') を指定した場合には、Emacs は個人的な init ファイルトでデフォルト init ファイのいずれもロードしません。

サイトのカスタマイズのためのファイルは `site-start.el`です。Emacs はユーザーの init ファイルの前にこれをロードします。オプション '`--no-site-file`'により、このファイルのロードを抑制できます。

`site-run-file` [User Option]
この変数はユーザーの init ファイルの前にロードするサイト用のカスタマイズファイルを指定する。通常の値は`"site-start"`。実際に効果があるようにこれを変更するには、Emacs の dump 前に変更するのが唯一の方法である。

一般的に必要とされる`.emacs`ファイルのカスタマイズ方法については Section "Init File Examples" in *The GNU Emacs Manual* を参照のこと。

`inhibit-default-init` [User Option]
この変数が非 `nil`なら Emacs がデフォルトの初期化ライブラリーファイルをロードするのを防ぐ。デフォルト値は `nil`。

`before-init-hook` [Variable]
このノーマルフックはすべての init ファイル (`site-start.el`、ユーザーの init ファイル、および `default.el`) のロード直前に一度実行される (実際に効果があるようにこれを変更するには Emacs の dump 前に変更するのが唯一の方法)。

`after-init-hook` [Variable]
このノーマルフックはすべての init ファイル (`site-start.el`、ユーザーの init ファイル、および `default.el`) のロード直後、端末固有ライブラリーのロードとコマンドラインアクション引数の処理の前に一度実行される。

`emacs-startup-hook` [Variable]
このノーマルフックはコマンドライン引数の処理直後に一度実行される。batch モードでは Emacs はこのフックを実行しない。

`window-setup-hook` [Variable]
このノーマルフックは `emacs-startup-hook`と非常に類似している。このフックは若干遅れてフレームパラメーターのセット後に実行されるのが唯一の違い。Section 40.1.1 [Startup Summary], page 1090 を参照のこと。

`user-init-file` [Variable]
この変数はユーザーの init ファイルの絶対ファイル名を保持する。実際にロードされた init ファイルが`.emacs.elc`のようにコンパイル済なら、値はそれに対応するソースファイルを参照する。

`user-emacs-directory` [Variable]
この変数は Emacs のデフォルトディレクトリーの名前を保持する。`~/.emacs.d/`および `~/.emacs`が存在せず`${XDG_CONFIG_HOME-'~/.config'}/emacs/`が存在すればその

ディレクトリーがデフォルト、それ以外なら MS-DOS 以外のすべてのプラットフォームでは`~/.emacs.d/`がデフォルト。ここで`${XDG_CONFIG_HOME-'~/.config'}`は環境変数`XDG_CONFIG_HOME`がセットされていればその値、それ以外なら`~/.config`を意味する。Section “How Emacs Finds Your Init File” in *The GNU Emacs Manual* を参照のこと。

40.1.3 端末固有の初期化

端末タイプはそれぞれ、その端末のタイプで Emacs が実行時にロードする独自の Lisp ライブラリーをもつことができます。そのライブラリーの名前は変数`term-file-prefix`の値と端末タイプ (環境変数`TERM`により指定) を結合することにより構築されます。`term-file-prefix`は通常は値`"term/"`をもち変更は推奨しません。連想リスト`term-file-aliases`内に`TERM`にマッチするエントリーが存在する場合には、Emacs は`TERM`のかわりにんその連想値を使用します。Emacs は通常の方法、つまり`load-path`のディレクトリーから‘`.elc`’と‘`.el`’の拡張子のファイルを検索することにより、このファイルを探します。

端末固有ライブラリーの通常の役割は特殊キーにより Emacs が認識可能なシーケンスを送信可能にすることです。Termcap と Terminfo のエントリーがその端末のすべてのファンクションキーを指定していなければ、`input-decode-map`へのセットや追加も必要になるかもしれません。Section 40.13 [Terminal Input], page 1118 を参照してください。

端末タイプにハイフンとアンダースコアーが含まれて、その端末名に等しい名前のライブラリーが見つからないときには、Emacs はその端末名から最後のハイフンまたはアンダースコアー以降を取り除いて再試行します。このプロセスは Emacs がマッチするライブラリーを見つかるか、その名前にハイフンとアンダースコアーが含まれなくなる (つまりその端末固有ファイルが存在しない) まで繰り返されます。たとえば端末名が‘`xterm-256color`’で`term/xterm-256color.el`というライブラリーが存在しなければ Emacs は`term/xterm.el`のロードを試みます。必要なら端末タイプの完全な名称を見つかるために端末ライブラリーは`(getenv "TERM")`を評価できます。

init ファイルで変数`term-file-prefix`を`nil`にセットすることにより端末固有ライブラリーのロードを防ぐことができます。

`tty-setup-hook`を使用することにより、端末固有ライブラリーのいくつかのアクションのアレンジやオーバーライドもできます。これは新たなテキスト端末の初期化後に Emacs が実行するノーマルフックです。自身のライブラリーをもたない端末にたいして初期化を定義するために、このフックを使用することのできるでしょう。Section 23.1 [Hooks], page 461 を参照してください。

`term-file-prefix` [User Option]

この変数の値が非`nil`なら Emacs は以下のように端末固有初期化ファイルをロードする:

```
(load (concat term-file-prefix (getenv "TERM")))
```

端末初期化ファイルのロードを望まない場合には変数`term-file-prefix`に`nil`をセットできる。

MS-DOS では Emacs は環境変数`TERM`に‘`internal`’をセットする。

`term-file-aliases` [User Option]

この変数は端末タイプをエイリアスにマップする連想リスト。たとえば`("vt102" . "vt100")`という形式の要素は‘`vt102`’というタイプの端末を‘`vt100`’タイプの端末として扱うことを意味する。

`tty-setup-hook` [Variable]

この変数は新たなテキスト端末の初期化後に Emacs が実行するノーマルフック (これは非ウィンドウのモードでの Emacs 開始時と`emacsclient`の TTY 接続作成時に適用される)。(適用

可能なら) このフックはユーザーの init ファイルおよび端末固有 Lisp ファイルのロード後に実行されるので、そのファイルにより行われた定義を調整するためにフックを使用できる。

関連する機能については Section 40.1.2 [Init File], page 1093 を参照のこと。

40.1.4 コマンドライン引数

Emacs 開始時に種々のアクションをリクエストするためにコマンドライン引数を使用できます。Emacs を使う際にはログイン後に一度だけ起動して同一の Emacs セッション内ですべてを行うのが推奨される方法です (Section "Entering Emacs" in *The GNU Emacs Manual* を参照)。この理由によりコマンドライン引数を頻繁に使うことはないかもしれません。それでもセッションスクリプトから Emacs を呼び出すときや Emacs のデバッグ時にコマンドライン引数が有用になるかもしれません。このセクションでは Emacs がコマンドライン引数を処理する方法を説明します。

`command-line` [Function]

この関数は Emacs が呼び出された際のコマンドライン引数を解析、処理、そして (とりわけ) ユーザーの init ファイルをロードしてスタートアップメッセージを表示する。

`command-line-processed` [Variable]

この変数の値は一度コマンドラインが処理されると `t`になる。

`dump-emacs` (Section E.1 [Building Emacs], page 1175 を参照) を呼び出すことにより Emacs を再 dump する場合には、新たに dump された Emacs に新たなコマンドライン引数を処理させるために最初にこの変数に `nil`をセットしたいと思うかもしれない。

`command-switch-alist` [Variable]

この変数はユーザー定義のコマンドライン引数とそれに関連付けられたハンドラー関数の alist。デフォルトでは空だが望むなら要素を追加できる。

コマンドラインオプション *(command-line option)* は以下の形式をもつコマンドライン上の引数である:

```
-option
```

`command-switch-alist`の要素は以下のようになる:

```
(option . handler-function)
```

CAR の *option*は文字列でコマンドラインオプションの名前 (先頭のハイフンは含む)。*handler-function*は *option*を処理するために呼び出されて、単一の引数としてオプション名を受け取る。

このオプションはコマンドライン内で引数を併う場合がある。この場合には、*handler-function* は残りのコマンドライン引数すべてを変数 `command-line-args-left` (以下参照) で見つけることができる (コマンドライン引数のリスト全体は `command-line-args`)。

`command-switch-alist`の処理は *option*内の等号を特別扱いしないことに注意。つまりコマンドラインに`--name=value`のようなオプションがなければ、`car`が文字通り`--name=value` の `command-switch-alist`のメンバーでなければこのオプションにマッチしない。そのようなオプションをパースしたければ、かわりに `command-line-functions` (以下参照) を使う必要がある。

コマンドライン引数は`startup.el`ファイル内の `command-line-1`により解析される。Section "Command Line Arguments for Emacs Invocation" in *The GNU Emacs Manual* も参照のこと。

`command-line-args` [Variable]

この変数の値は Emacs に渡されたコマンドライン引数のリスト。

`command-line-args-left` [Variable]
この変数の値はまだ処理されていないコマンドライン引数のリスト。

`command-line-functions` [Variable]
この変数の値は認識されなかったコマンドライン引数を処理するための関数のリスト。次の引数が処理されてそれに特別な意味がないときは、その都度このリスト内の関数が非 `nil`をリターンするまでリスト内での出現順に呼び出される。

これらの関数は引数なしで呼び出される。関数はその時点で一時的にバインドされている変数 `argi`を通じて検討中のコマンドラインにアクセスできる。残りの引数 (カレントの引数含まず) は変数 `command-line-args-left`内にあり。

関数が `argi`内のその引数を認識して処理したときは引数を処理したと告げるために非 `nil`をリターンすること。後続の引数のいくつかを処理したときは `command-line-args-left`からそれらを削除してそれを示すことができる。

これらの関数すべてが `nil`をリターンした場合には引数は visit すべきファイル名として扱われる。

40.2 Emacs からの脱出

Emacs から抜け出すには 2 つの方法があります: 1 つ目は永遠に exit する Emacs ジョブの kill、2 つ目はサスペンドする方法でこれは後から Emacs プロセスに再エンターすることができます (もちろんグラフィカルな環境では Emacs で特に何もせず単に他のアプリケーションにスイッチして後で望むときに Emacs に戻れる)。

40.2.1 Emacs の kill

Emacs の kill とは Emacs プロセスの終了を意味します。端末から Emacs を開始した場合には、通常は親プロセスの制御が再開されます。Emacs を kill する低レベルなプリミティブは `kill-emacs`です。

`kill-emacs` **&optional** *exit-data* [Command]
このコマンドはフック `kill-emacs-hook`を呼び出してから Emacs プロセスを exit して kill する。

*exit-data*が整数なら Emacs プロセスの exit ステータスとして使用される (これは主に batch 処理で有用。Section 40.17 [Batch Mode], page 1121 を参照)。

*exit-data*が文字列なら内容は端末の入力バッファーに詰め込まれるので、shell(や何であれ次の入力を読み込むプログラム) が読み込むことができる。

*exit-data*が整数や文字列以外、または省略なら成功裏にプログラムが終了したことを示す exit ステータス (システム固有) の使用を意味する。

関数`kill-emacs`は通常はより高レベルなコマンド *C-x C-c* (`save-buffers-kill-terminal`) を通じて呼び出される。Section “Exiting” in *The GNU Emacs Manual* を参照のこと。これは Emacs がオペレーティングシステムのシグナル `SIGTERM`や `SIGHUP`を受け取った場合 (たとえば制御端末が切断されたとき) や、batch モードで実行中に `SIGINT`を受け取った場合 (Section 40.17 [Batch Mode], page 1121 を参照) にも自動的にこれが呼び出される。

`kill-emacs-hook` [Variable]
このノーマルフックは Emacs の kill の前に `kill-emacs`により実行される。

`kill-emacs`はユーザーとの対話が不可能な状況 (たとえば端末が切断されたとき) で呼び出されるかもしれないので、このフックの関数はユーザーとの対話を試みるべきではない。Emacs シャットダウン時にユーザーと対話したければ下記の`kill-emacs-query-functions`を使用すること。

Emacs を kill したときには保存されたファイルを除き Emacs プロセス内のすべての情報が失われます。うっかり Emacs を kill することで大量の作業が失われるので、`save-buffers-kill-terminal`コマンドは保存を要するバッファーがあったり実行中のサブプロセスがある場合には確認の問い合わせを行います。これはアブノーマルフック`kill-emacs-query-functions`も実行します。

`kill-emacs-query-functions` [User Option]
`save-buffers-kill-terminal`が Emacs を kill する際には標準の質問を尋ねた後、`kill-emacs`を呼び出す前にこのフック内の関数を呼び出す。関数は出現順に引数なしで呼び出される。関数はそれぞれ追加でユーザーから確認を求めることができる。それらのいずれかが`nil`をリターンすると`save-buffers-kill-emacs`は Emacs を kill せずに、このフック内の残りの関数は実行されない。直接`kill-emacs`を呼び出すとフックは実行されない。

40.2.2 Emacs のサスペンド

テキスト端末では *Emacs* のサスペンドができます。これは Emacs を一時的にストップして上位のプロセスに制御を返します。これは通常は shell です。これにより後で同じ Emacs プロセス内の同じバッファー、同じ kill リング、同じアンドゥヒストリー、... で編集を再開できます。Emacs を再開するには親 shell 内で適切なコマンド— 恐らくは`fg` — を使用します。

その Emacs セッションが開始された端末デバイス上でのみサスペンドは機能します。そのデバイスのことをセッションの制御端末 *(controlling terminal)* と呼びます。制御端末がグラフィカルな端末ならサスペンドは許されません。グラフィカルな端末では Emacs で特別なことをせずに単に別のアプリケーションにスイッチできるのでサスペンドは通常は関係ありません。

いくつかのオペレーティングシステム (`SIGTSTP`のないシステムや MS-DOS) ではジョブの停止はサポートされません。これらのシステムでの停止は Emacs のサブプロセスとして新たな shell を一時的に作成します。Emacs に戻るためには shell を exit すればよいでしょう。

`suspend-emacs` **&optional** *string* [Command]
この関数は Emacs を停止して上位のプロセスに制御を返す。上位プロセスが Emacs を再開する際には、Lisp での`suspend-emacs`の呼び出し元に`nil`をリターンする。

この関数はその Emacs セッションの制御端末上でのみ機能する。他の TTY デバイスの制御を放棄するには`suspend-tty`を使用する (下記参照)。その Emacs セッションが複数の端末を使用する場合には Emacs のサスペンド前に他のすべての端末からフレームを削除しなければならず、さもないとこの関数はエラーをシグナルする。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

*string*が非`nil`なら、その各文字は Emacs の上位 shell に端末入力として送信される。*string* 内の文字は上位 shell によりエコーされずに結果だけが表示される。

サスペンドする前に`suspend-emacs`はノーマルフック`suspend-hook`を実行する。ユーザーが Emacs 再開後に`suspend-emacs`はノーマルフック`suspend-resume-hook`を実行する。Section 23.1 [Hooks], page 461 を参照のこと。

再開後の次回再表示では変数`no-redraw-on-reenter`が`nil`ならスクリーン全体が再描画される。Section 39.1 [Refresh Screen], page 981 を参照のこと。

以下はこれらのフックの使用例:

```
(add-hook 'suspend-hook
          (lambda () (or (y-or-n-p "Really suspend? ")
                         (error "Suspend canceled"))))
(add-hook 'suspend-resume-hook (lambda () (message "Resumed!")
                                  (sit-for 2)))
```

`(suspend-emacs "pwd")`を評価すると以下を目にするだろう:

```
---------- Buffer: Minibuffer ----------
Really suspend? y
---------- Buffer: Minibuffer ----------

---------- Parent Shell ----------
bash$ /home/username
bash$ fg

---------- Echo Area ----------
Resumed!
```

Emacs サスペンド後に ‘`pwd`’がエコーされないことに注意。エコーはされないが shell により読み取られて実行されている。

`suspend-hook` [Variable]

この変数は Emacs がサスペンド前に実行するノーマルフック。

`suspend-resume-hook` [Variable]

この変数はサスペンド後の再開時に Emacs が実行するノーマルフック。

`suspend-tty` **&optional** *tty* [Function]

*tty*に Emacs が使用する端末デバイスを指定すると、この関数はそのデバイスを放棄して以前の状態にリストアする。そのデバイスを使用していたフレームは存在を続けるが更新はされず、Emacs はそれらのフレームから入力を読み取らない。*tty*には端末オブジェクト、フレーム (そのフレームの端末の意)、`nil` (選択されたフレームの端末の意) を指定できる。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

*tty*がサスペンド済みなら何も行わない。

この関数は端末オブジェクトを各関数への引数としてフック `suspend-tty-functions`を実行する。

`resume-tty` **&optional** *tty* [Function]

この関数は以前にサスペンドされたデバイス *tty*を再開する。ここで *tty*は `suspend-tty`に指定できる値と同じである。

この関数は端末デバイスの再オープンと再初期化を行い、その端末の選択されたフレームで端末を再描画する。それから端末ブジェクトを各関数への引数としてフック `resume-tty-functions`を実行する。

同じデバイスが別の Emacs 端末で使用済みなら、この関数はエラーをシグナルする。*tty*がサスペンドされていなければ何もしない。

`controlling-tty-p` **&optional** *tty* [Function]

この関数は *tty*がその Emacs セッションの制御端末なら非 `nil`をリターンする。*tty*には端末オブジェクト、フレーム (そのフレームの端末の意)、`nil` (選択されたフレームの端末の意) を指定できる。

`suspend-frame` [Command]
このコマンドはフレームをサスペンドする。GUI フレームでは `iconify-frame`を呼び出す (Section 29.11 [Visibility of Frames], page 736 を参照)。テキスト端末上のフレームでは、そのフレームが制御端末デバイス上で表示されていれば `suspend-emacs`、されていなければ `suspend-tty`のいずれかを呼び出す。

40.3 オペレーティングシステムの環境

Emacs はさまざまな変数を通じてオペレーティングシステム環境内の変数へのアクセスを提供します。これらの変数にはシステムの名前、ユーザーの UIDなどが含まれます。

`system-configuration` [Variable]
この変数はユーザーのシステムのハードウェアとソフトウェアにたいする GNU の標準コンフィグレーション名 (standard GNU configuration name) を保持する。たとえば 64 ビット GNU/Linux システムにたいする典型的な値は‘`"x86_64-unknown-linux-gnu"`’。

`system-type` [Variable]
この変数の値は Emacs 実行中のオペレーティングシステムのタイプを示すシンボル。可能な値は :

`aix` IBM の AIX。

`berkeley-unix`
Berkeley BSD とその変種。

`cygwin` MS-Windows で最上位の Posix レイヤーである Cygwin。

`darwin` Darwin (macOS).

`gnu` (HURD と Mach から構成される)GNU システム。

`gnu/linux`
GNU/Linux システム — すなわち Linux カーネルを使用する GNU システムの変種 (これらのシステムは人がしばしば “Linux” と呼ぶシステムだが実際には Linux は単なるカーネルであってシステム全体ではない)。

`gnu/kfreebsd`
FreeBSD カーネルによる (glibc ベースの)GNU システム。

`hpux` ヒューレット・パッカードの HPUX オペレーティングシステム。

`nacl` Google Native Client(NaCl) サンドボックスシステム。

`ms-dos` Microsoft の DOS。MS-DOS にたいする DJGPP でコンパイルされた Emacs は、たとえ MS-Windows 上で実行されていても `system-type`が `ms-dos`にバインドされる。

`usg-unix-v`
AT&T の Unix System V。

`windows-nt`
Microsoft の Windows NT、9X 以降。たとえば Windows 10 でも `system-type`の値は常に `windows-nt`。

わたしたちは絶対に必要になるまでは、より細分化するために新たなシンボルを追加したくありません。実際のところ将来的にはこれらの候補のいくつかを取り除きたいと思っています。`system-type`で許されているより細分化する必要がある場合には、たとえば正規表現にたいして`system-configuration`をテストできます。

`system-name` [Function]
この関数は実行中のマシン名を文字列としてリターンする。

`mail-host-address` [User Option]
この変数が非`nil`の場合には、email アドレスを生成するために`system-name`のかわりにこの変数が使用される。たとえばこれは`user-mail-address`のデフォルト値の構築時に使用される。Section 40.4 [User Identification], page 1103 を参照のこと。

`getenv` *var* **&optional** *frame* [Command]
この関数は環境変数 *var*の値を文字列としてリターンする。*var*は文字列であること。その環境内で *var*が未定義なら `getenv`は`nil`をリターンする。*var*がセットされているが null(訳注: 空文字列) なら‘`""`’をリターンする。Emacs 内では環境変数とそれらの値のリストは変数`process-environment`内に保持されている。

```
(getenv "USER")
     ⇒ "lewis"
```

shell コマンド`printenv`は環境変数のすべて、または一部をプリントする:

```
bash$ printenv
PATH=/usr/local/bin:/usr/bin:/bin
USER=lewis
TERM=xterm
SHELL=/bin/bash
HOME=/home/lewis
...
```

`setenv` *variable* **&optional** *value substitute* [Command]
このコマンドは *variable*という名前の環境変数の値に *value*をセットする。*variable*は文字列であること。内部的には Emacs Lisp は任意の文字列を扱える。しかし *variable*は通常は shell 識別子として有効、すなわちアルファベットかアンダースコアで始まり、アルファベットか数字またはアンダースコアのシーケンスであること。それ以外なら Emacs のサブプロセスが *variable*の値にアクセスを試みるとエラーが発生するかもしれない。*value*が省略か`nil`の場合 (またはプレフィクス引数とともにインタラクティブに呼び出された場合) には、`setenv`はその環境から *variable*を削除する。それ以外なら *variable*は文字列であること。

オプション引数 *substitute*が非`nil`なら、*value*内のすべての環境変数を展開するために Emacs は関数`substitute-env-vars`を呼び出す。

`setenv`は`process-environment`を変更することにより機能する。この変数を`let`でバインドするのも合理的プラクティスである。

`setenv`は *variable*の新たな値、または環境から *variable*が削除されていれば`nil`をリターンする。

`process-environment` [Variable]
この変数はそれぞれが 1 つの環境変数を記す文字列リスト。関数`getenv`と`setenv`はこの変数により機能する。

```
process-environment
⇒ ("PATH=/usr/local/bin:/usr/bin:/bin"
    "USER=lewis"
    "TERM=xterm"
    "SHELL=/bin/bash"
    "HOME=/home/lewis"
    ...)
```

`process-environment`に同じ環境変数を指定す複数の要素が含まれる場合には、それらの最初の要素が変数を指定して他は無視される。

`initial-environment` [Variable]
この変数は Emacs 開始時にその親プロセスから Emacs が継承した環境変数のリストを保持する。

`path-separator` [Variable]
この変数は、(環境変数で見つけた) 検索パス内でディレクトリーを区切る文字を示す文字列を保持する。値は Unix と GNU システムでは`":"`、MS システムでは`";"`。

`parse-colon-path` *path* [Function]
この関数は環境変数`PATH`の値のような検索パス文字列を引数に受け取り、セパレーターで分割してディレクトリーのリストをリターンする。このリスト内では`nil`はカレントディレクトリーを意味する。この関数の名前からはセパレーターは “コロン” となるが、実際に使用するのは`path-separator`の値。

```
(parse-colon-path ":/foo:/bar")
     ⇒ (nil "/foo/" "/bar/")
```

`invocation-name` [Variable]
この変数は Emacs が呼び出された時のプログラム名を保持する。値は文字列でありディレクトリー名は含まれない。

`invocation-directory` [Variable]
この変数は Emacs 実行可能形式が実行されたときに配置されていたディレクトリー名、ディレクトリーが判断できなければ`nil`をリターンする。

`installation-directory` [Variable]
非`nil`ならサブディレクトリー`lib-src`と`etc`を探すディレクトリーである。インストールされた Emacs なら通常は`nil`。Emacs が標準のインストール位置にそれらのディレクトリーを見つけられないものの、Emacs 実行可能形式を含むディレクトリー (たとえば`invocation-directory`) に何らかの関連があるディレクトリーで見つかることができたら非`nil`。

`load-average` **&optional** *use-float* [Function]
この関数はカレント、1 分、5 分、15 分のロードアベレージ (load averages: 平均負荷) をリストでリターンする。このロードアベレージはシステム上で実行を試みているプロセス数を示す。

デフォルトでは値はシステムロードアベレージを 100 倍にした整数だが、*use-float*が非`nil`なら 100 を乗ずることなくこれらの値は浮動小数点数としてリターンされる。

ロードアベレージ入手が不可能ならこの関数はエラーをシグナルする。いくつかのプラットフォームではロードアベレージへのアクセスにカーネル情報を読み取れるように、通常は推奨されない setuid か setgid した Emacs のインストールを要する。

1 分のロードアベレージは利用できるが、5 分と 15 分のアレージは利用できなければ、この関数は利用可能なアベレージを含んだ短縮されたリストをリターンする。

```
(load-average)
     ⇒ (169 48 36)
(load-average t)
     ⇒ (1.69 0.48 0.36)
```

shell コマンドの `uptime`はこれと類似する情報をリターンする。

`emacs-pid` [Function]

この関数は Emacs プロセスのプロセス IDを整数としてリターンする。

`tty-erase-char` [Variable]

この変数は Emacs 開始前にそのシステムの端末ドライバーで選択されていた erase 文字を保持する。

40.4 ユーザーの識別

`init-file-user` [Variable]

この変数は Emacs によりどのユーザーの init が使用されるべきか — なければ `nil`をリターンする。`""`はログイン時のオリジナルのユーザーをリターンする。この値は '`-q`'や '`-u` *user*' のようなコマンドラインオプションを反映する。

カスタマイズ関連のファイルや、他の類の短いユーザープロファイルをロードする Lisp パッケージは、それをどこで探すか判断するためにこの変数にしたがうこと。これらの Lisp パッケージはこの変数内で見つかったユーザー名のプロファイルをロードすること。`init-file-user`が `nil`なら '`-q`'、'`-Q`'、または '`-batch`'オプションが使用されたことを意味しており、その場合には Lisp パッケージはカスタマイズファイルやユーザープロファイルを何もロードするべきではない。

`user-mail-address` [User Option]

これは Emacs を使用中のユーザーの電子メールアドレスを保持する。

`user-login-name` **&optional** *uid* [Function]

この関数はユーザーのログイン名をリターンする。これはいずれかがセットされていれば環境変数 `LOGNAME`か `USER`を使用する。それ以外なら値は実 UIDではなく実効 UIDにもとづく。

uid (数字) を指定すると *uid*に対応するユーザー名、そのようなユーザーが存在しなければ `nil`が結果となる。

`user-real-login-name` [Function]

この関数は Emacs の実 UIDに対応するユーザー名をリターンする。これは実効 UID、および環境変数 `LOGNAME`と `USER`を無視する。

`user-full-name` **&optional** *uid* [Function]

この関数はログインユーザーの完全名、環境変数 `NAME`がセットされていればその値をリターンする。

Emacs プロセスのユーザー ID が既知のユーザーに不一致 (かつ与えられた `NAME`が未セット) なら結果は`"unknown"`。

*uid*が非 `nil`なら数字 (ユーザー ID) か文字列 (ログイン名) であること。その場合には `user-full-name`はそのユーザー名かログイン名に対応する完全名をリターンする。未定義のユーザー名かログイン名を指定すると `nil`をリターンする。

シンボル `user-login-name`、`user-real-login-name`、`user-full-name`は変数であると同時に関数でもあります。関数の場合には、その名前の変数と同じ値をリターンします。これらの変数を使えば対応する関数が何をリターンするべきかを告げることにより Emacs を騙すことができます。またフレームタイトルの構築においても、これらの関数は有用です (Section 29.6 [Frame Titles], page 730 を参照)。

`user-real-uid` [Function]
この関数はユーザーの実 UIDをリターンする。This function returns the real of the user.

`user-uid` [Function]
この関数はユーザーの実効 UIDをリターンする。

`group-gid` [Function]
この関数は Emacs プロセスの実効 GIDをリターンする。

`group-real-gid` [Function]
この関数は Emacs プロセスの実 GIDをリターンする。

`system-users` [Function]
この関数はシステム上のユーザー名をリストする文字列リストをリターンする。この情報を Emacs が取得できなければ`user-real-login-name`の値だけを含んだリストをリターンする。

`system-groups` [Function]
この関数はシステム上のグループ名をリストする文字列リストをリターンする。この情報を Emacs が取得できなければリターン値は `nil`。

`group-name` *gid* [Function]
この関数は数値のグループ ID *gid*に対応するグループ名、そのようなグループがなければ `nil` をリターンする。

40.5 時刻

このセクションではカレント時刻とタイムゾーンを決定する方法を説明します。

`current-time`や `file-attributes`のような多くの関数は秒をカウントする *Lisp* タイムスタンプ *(Lisp timestamp)* 値をリターンします。この値は 1970-01-01 00:00:00 UTC というエポック *(epoch)* からの経過秒数をカウントすることにより絶対時刻を表すことができます。

伝統的な Lisp タイムスタンプが整数のペアであったとしても、それらの形式は進化しており、プログラムは通常はカレントのデフォルト形式に依存するべきではありません。プログラムに特定のタイムスタンプ形式が必要なら、`time-convert`関数を使用して必要とする形式に変換できます。Section 40.7 [Time Conversion], page 1106 を参照してください。

現在のところ 3 つの Lisp タイムスタンプ形式があり、それぞれが秒数を表します:

- 整数。これがもっとも単純な形式だが、少数秒のタイムスタンプは表現できない。
- 整数のペア `(`*ticks* `.` *hz*`)` (*hz*は正)。これは *ticks*/*hz*秒を表し、*hz*が 1 なら単なる *ticks*と同時刻。*hz*にたいして一般的な値はナノ秒解像度クロック用の 1000000000[1]。

[1] 現在のところ *hz*は互換性に関する警告を回避するために、少なくとも 65536 である必要があります。以前のバージョンの Emacs は後方互換性の観点からそのようなタイムスタンプを異なるように扱うからです。これらの警告は Emacs の将来バージョンでは削除が計画されています。

- 4 つの整数からなるリスト (*high low micro pico*) (0≤*low*<65536、0≤*micro*<1000000、0≤*pico*<1000000)。これは次式を使用して秒数を表す: $high \times 2^{16} + low + micro \times 10^{-6} + pico \times 10^{-12}$。あるケースにおいては、この関数はデフォルトで *micro*および *pico*を 0 に省略した 2 つ、または 3 つの要素のリストをリターンするかもしれない。現在のすべてのマシンでは *pico*は 1000 の倍数だが、より高精度のクロックが利用可能になったら変更されるかもしれない。

関数の引数 (`current-time-string`の *time*引数) に、より一般的な *time* 値 *(time value)* のフォーマット (Lisp タイムスタンプ、カレント時刻にたいする `nil`、秒にたいする単精度浮動小数点数、欠落要素を 0 に切り詰めたタイムスタンプリスト (*high low micro*)) を許容します。

time 値は暦形式や他形式に相互に変換できます。これらの変換のいくつかは利用可能な time 値範囲を制限するオペレーティングシステム関数に依存しており、その制限を超えると‘`"Specified time is not representable"`’のようなエラーをシグナルします。たとえばあるシステムでは 1970 や 1901 より前の年、遠い将来の年をサポートしないかもしれません。`format-time-string`を使用して可読性のある文字列、`time-convert`を使用して Lisp タイムスタンプ、`decode-time`や`float-time`を使用して別の形式に time 値を変換できます。これらの関数については以降のセクションで説明します。

`current-time-string` **&optional** *time zone* [Function]

この関数はカレントの時刻と日付を可読形式の文字列でリターンする。この文字列の先頭部分には曜日、月、日付、時刻がこの順に含まれて、それらが可変長となることはない。これらのフィールドにたいして使用される文字数は常に同じとはいえ、年は正確に 4 桁とはかぎらず、いつかの将来に終端に追加情報が追加されるかもしれないので、`current-time-string`の出力からフィールドを抽出するより `format-time-string`を使うほうが通常は便利です。

引数 *time*が与えられたら、それはカレント時刻のかわりにフォーマットする時刻を指定する。オプション引数 *zone*のデフォルトはカレントのタイムゾーンルール。Section 40.6 [Time Zone Rules], page 1105 を参照のこと。time の範囲および zone の値はオペレーティングシステムが制限する。

```
(current-time-string)
     ⇒ "Fri Nov  1 15:59:49 2019"
```

`current-time` [Function]

この関数はカレント時刻を Lisp タイムスタンプとしてリターンする。Emacs の現在のリリースでタイムスタンプが (*high low micro pico*)という形式だとしても、Emacs の将来バージョンで変更が予定されている。タイムスタンプを他形式に変換するには`time-convert`関数を使用できる。Section 40.7 [Time Conversion], page 1106 を参照のこと。

`float-time` **&optional** *time* [Function]

この関数はエポックからの経過秒数を浮動小数点数としてリターンする。オプション引数 *time-value*が与えられた場合には、カレント時刻ではなく変換する時刻を指定する。

警告: 結果は浮動小数点数なので正確ではないかもしれない。正確なタイムスタンプが必要なら使用しないこと。たとえば典型的なシステムにおいては `(float-time '(1 . 10))`を‘`0.1`’と表示するが、これは 1/10 より若干大きい。

`time-to-seconds`はこの関数のエイリアス。

40.6 タイムゾーンのルール

デフォルトのタイムゾーンは環境変数`TZ`により判断されます。Section 40.3 [System Environment], page 1100 を参照してください。たとえば `(setenv \"TZ\" \"UTC0\")`とすれば万国標準時の使用

を Emacs に指示できます。その環境に TZがなければ、Emacs はプラットフォーム依存のデフォルト時刻であるシステムの実時間 (system wall clock time) を使用します。

サポートされる TZのセットはシステム依存です。GNU と他の多くのシステムは tzdata データベースをサポートします。これはたとえば‘"America/New_York"’はニューヨーク市周辺のタイムゾーンを夏時間ヒストリーを指定します。GNU と他の多くのシステムは POSIX スタイルの TZセッティングをサポートします。これはたとえば‘"EST+5EDT,M4.1.0/2,M10.5.0/2"’は 1987 から 2006 にニューヨークで使用されたルールを指定します。すべてのシステムは万国標準時 (Universal Time) を意味する‘"UTC0"’文字列をサポートします。

ローカル時刻にたいする変換関数は、変換のタイムゾーンと夏時間の履歴を指定するタイムゾーンルール)*time zone rule* をオプションとして受け取ります。タイムゾーンルールが省略か nilなら、変換にはデフォルトのタイムゾーン、tなら万国標準時、wallならシステムの実時間 (system wall clock time) が使用されます。これが文字列なら変換には TZにその文字列をセットしたのと等価なタイムゾーンルールが使用されます。(*offset abbr*) (*offset*は万国標準時刻より進んでいる秒数を与える整数で *abbr*は文字列) のようなリストなら、変換には与えられたオフセットと省略名の固定タイムゾーンが使用されます。整数 *offset*の場合には、POSIX 互換プラットフォームでは数値省略形の *abbr*、MS-Windows では未指定の *abbr*のとなるような (*offset abbr*) であるかのように扱われます。

`current-time-zone` **&optional** *time zone* [Function]

この関数はユーザーが居るタイムゾーンを記すリストをリターンする。

値は (*offset abbr*)という形式をもつ。ここで *offset*は万国標準時刻より進んでいる秒数 (グリニッジより東) を与える整数。負の値はグリニッジより西を意味する。2 つ目の要素 *abbr*はそのタイムゾーンのを与える省略名の文字列。たとえば‘"CST"’は中国標準時か米国中部標準時のタイムゾーン。夏時間の開始と終了時に、いずれの要素も変化し得る。ユーザーが季節時間調整を用いていないタイムゾーンを指定した場合には、値は時期を通して定数となる。

この値を計算するのに必要なすべての情報をオペレーティングシステムが提供しなければ、このリストの未知の要素は nilになる。

引数 *time*が与えられたら、それはカレント時刻のかわりに分析するべき time 値を指定する。オプション引数 *zone*のデフォルトはカレントのタイムゾーンルール。time の範囲および zone の値はオペレーティングシステムが制限する。

40.7 時刻の変換

以下の関数は time 値 (Section 40.5 [Time of Day], page 1104 を参照) を Lisp タイムスタンプ暦情報に変換したり逆の変換を行います。

32 ビットオペレーティングシステムの多くは、秒数コンポーネントに 32 ビット情報を含んだシステム時刻に制限されます。これらのシステムは通常は万国標準時の 1901-12-13 20:45:52 から 2038-01-19 03:14:07 までの時刻だけを処理します。しかし 64 ビット、およびいくつかの 32 ビットオペレーティングシステムは、より大きな秒数コンポーネント値をもち、より遠い過去や未来の時刻を表現できます。

暦変換関数は、たとえグレゴリオ暦導入前の日付にたいしても常にグレゴリオ暦を使用します。年は BC. 1 年から年数を数えて伝統的なグレゴリオ年が行うように 0 年をスキップしません。たとえば年数 −37 はグレゴリオ年の BC. 38 年を表します。

`time-convert` *time* **&optional** *form* [Function]

この関数は time 値を Lisp タイムスタンプに変換する。

オプションの *form*引数はリターンするタイムスタンプ形式を指定する。この関数は *form*がシンボル integerなら整数でカウントした秒数をリターンする。正の整数の *form*はクロック周

波数を指定する。その場合にはこの関数は `(ticks . form)`という整数ペアのタイムスタンプをリターンする [2]。*form*が `t`なら、この関数はタイムスタンプを適切に表現するような正の整数としてそれを扱う。たとえば *time*が nil でプラットフォームのタイムスタンプがナノ秒の解像度をもつ場合には 1000000000 としてそれを扱う。*form*が `list`なら、この関数は整数のリスト `(high low micro pico)`をリターンする。現在のところ *form*が省略か `nil`なら `list`のように動作するとしても、Emacs の将来バージョンで変更が予定されているので、呼び出し側がリスト形式のタイムスタンプを必要とする場合には明示的に `list`を渡すこと。

*time*が無限 (infinite) や NaN(not a number: 数字ではない) なら、この関数はエラーをシグナルする。それ以外の場合には、*time*を正確に表せなければ、負の無限大方向に切り詰めて変換する。*form*が `t`なら変換は常に正確なので切り詰めは発生せず、リターン値のクロック解像度が *time*の解像度より小さくなることはない。対照的に `float-time`は、たとえ結果が正確でなくても、任意の Lisp time 値をエラーをシグナルせずに変換できる。Section 40.5 [Time of Day], page 1104 を参照のこと。

この関数は効率化のために *time*と `eq`な値、あるいは *time*と構造を共有する値をリターンするかもしれない。

`(time-convert nil nil)`は `(current-time)`と等価だが後者は幾分速い。

```
(setq a (time-convert nil t))
⇒ (1564826753904873156 . 1000000000)
(time-convert a 100000)
⇒ (156482675390487 . 100000)
(time-convert a 'integer)
⇒ 1564826753
(time-convert a 'list)
⇒ (23877 23681 904873 156000)
```

`decode-time` **&optional** *time zone form* [Function]

この関数は time 値を暦情報に変換する。*time*を指定しなければカレント時刻をデコードする。同様に *zone*のデフォルトはカレントのタイムゾーンルール。Section 40.6 [Time Zone Rules], page 1105 を参照のこと。time の範囲および zone の値はオペレーティングシステムが制限する。

*form*引数はリターンされる *seconds*要素の形式を制御する (以下参照)。リターン値は以下のような 9 要素のリスト:

```
(seconds minutes hour day month year dow dst utcoff)
```

以下は各要素の意味:

seconds 以下で説明する形式による、分秒の秒。

minutes 0 から 59 までの整数で表した時を過ぎた時分秒の分。

hour 0 から 23 までの整数で表した時分秒の時。

day 1 から 31 までの整数で表した年月日の日。

month 1 から 12 までの整数で表した年月日の月。

year 通常は 1900 より大きい整数で表した年月日の年。

[2] 標準的な関数に与えることを意図した Lisp タイムスタンプをリターン値として期待するなら、*form*は現在のところ少なくとも 65536 であるような正の整数である必要があります。

dow 0 から 6 までの整数で表した曜日であり 0 は日曜日を意味する。

dst 夏時間が有効なら `t`、無効なら `nil`、その情報が利用できなければ −1。

utcoff 万国標準時からの秒数、すなわち東グリニッジの秒数を示す整数。

Lisp タイムスタンプの *seconds*要素は非負かつ 61 より小さいこと。これはは正の閏秒の間以外は 60 より小さくなる (オペレーティングシステムが閏秒をサポートする場合)。オプションの *form*引数が`t`なら、*seconds*は *time*と同じ精度を使用する。*form*が`integer`なら *seconds*を整数に切り捨てる。たとえば *time*がタイムスタンプ`(1566009571321 . 1000)` (閏秒がない通常のシステムでは 2019-08-17 02:39:31.321 UTC を表す) なら、`(decode-time` *time* `t t)`は`((31321 . 1000) 39 2 17 8 2019 6 nil 0)`だが`(decode-time` *time* `t 'integer)`は`(31 39 2 17 8 2019 6 nil 0)`をリターンする。*form*が省略か`nil`の場合のデフォルトは現在のところ`integer`だが、このデフォルトは Emacs の将来バージョンで変更されるかもしれないので、呼び出し側は特定の形式が必要なら *form*を指定すること。

Common Lisp に関する注意**:** Common Lisp では *dow*と *utcoff*の意味が異なり、*second*は 0 から 59(両端を含む) の整数である。

time 値の要素にアクセス (や変更) するためにはアクセッサとして`decoded-time-second`、`decoded-time-minute`、`decoded-time-hour`、`decoded-time-day`、`decoded-time-month`、`decoded-time-year`、`decoded-time-weekday`、`decoded-time-dst`、`decoded-time-zone`を使用できる。

たとえばデコード済み time の年を増加するには以下のようにする:

```
(setf (decoded-time-year decoded-time)
      (+ (decoded-time-year decoded-time) 4))
```

以下の関数も参照のこと。

`decoded-time-add` *time delta* [Function]

この関数はデコード済み time 構造を受け取り、*delta* (これもデコード済み time 構造) を追加する。*delta*内の`nil`の要素は無視する。

たとえば “来月同刻” を得るには以下のようにする:

```
(let ((time (decode-time nil nil t))
      (delta (make-decoded-time :month 2)))
  (encode-time (decoded-time-add time delta)))
```

この日付が存在しない (たとえば 1 月 31 日に実行した) 場合には、有効な日付を入手できるまで日付を戻る (2 月の 28 日か 29 日になるだろう)。

フィールドはもっとも意味をもつ順に追加されるので、上述のような調整が発生したら日時分秒の追加の前に行われる。

値を減算するために *delta*内に負の値を指定できる。

リターン値はデコード済み time 構造。

`make-decoded-time` **&key** *second minute hour day month year dst zone* [Function]

リターンされるデコード済み time 構造では、与えられたキーワードだけ充填されて、それ以外は`nil`のままとなる。たとえば “2ヵ月” を表す構造を得るには:

```
(make-decoded-time :month 2)
```

`encode-time` *time* **&rest** *obsolescent-arguments* [Function]
これは *time*を Lisp タイムスタンプに変換する。これは `decode-time`の逆の関数として機能する。

最初の引数は通常は `decode-time`の形式でデコード済み time を指定するリスト (*second minute hour day month year ignored dst zone*)なので、(`encode-time` (`decode-time ...`))は機能する。これらのリストメンバーの意味は `decode-time`の下のテーブルを参照のこと。

廃れた呼び出し規約として、この関数は 6 つ以上の引数を受け取ることができる。最初の 6 つの引数 *second*、*minute*、*hour*、*day*、*month*、*year*はデコード済み time のほとんどの要素を指定する。7 つ目以降の引数があれば、最後の引数は *zone*として使用されれるので、(`apply #'encode-time (decode-time ...)`)は機能する。この廃れた規約においては、*zone*のデフォルトはレントタイムゾーンルール (Section 40.6 [Time Zone Rules], page 1105 を参照) であり、*dst*は −1 であるかのように扱われる。

100 未満の年が特別に扱われることはない。これに 1900 や 2000 を超える年を意味させたい場合には、`encode-time`を呼び出す前に自身でこれらを修正しなければならない。time の範囲および zone の値はオペレーティングシステムが制限する。

`encode-time`関数は `decode-time`のラフな逆関数として動作する。たとえば以下のように後者の出力を前者に渡すことができる:

```
(encode-time (decode-time ...))
```

seconds、*minutes*、*hour*、*day*、*month*に範囲外の値を使用することにより単純な日付計算ができる。たとえば *day*が 0 なら与えられた *month*の前月末日になる。

40.8 時刻のパースとフォーマット

以下の関数は time 値と文字列内のテキストの変換と逆変換を行います。time 値には `nil`、整数、および Lisp タイムスタンプが含まれます (Section 40.5 [Time of Day], page 1104 を参照)。

`date-to-time` *string* [Function]
この関数は time 文字列 *string*をパースして対応する Lisp タイムスタンプをリターンする。引数 *string*は日時を表現する `parse-time-string` (以下参照) が認識する形式のいずれかであること。この関数は *string*に明示的なタイムゾーン情報が欠落していれば万国標準時 (Universal Time) を仮定する。time の範囲および zone の値はオペレーティングシステムが制限する。

`parse-time-string` *string* [Function]
この関数は time 文字列 *string*を以下の形式のリストにパースする:

```
(sec min hour day mon year dow dst tz)
```

このリストのフォーマットは `decode-time`が受け取るフォーマットと同じであり、詳細はそこに説明してある (Section 40.7 [Time Conversion], page 1106 を参照)。入力から判断できない要素はすべて `nil`にセットされる。引数 *string*は RFC 822(またはそれ以降) や ISO 8601 に類似した “Fri, 25 Mar 2016 16:24:56 +0100” や “1998-09-12T12:21:54-0200” のような文字列であることが必要だが、この関数は形式が若干不正な time 文字列のパースも同様に試みる。

`iso8601-parse` *string* [Function]
より厳格な (無効な入力にはエラーを出力する) 関数のかわりに、この関数を使用できる。これは ISO 8601 標準の変種をパースできるので、上述のフォーマットに加えて “1998W45-3”

(週番号) や “1998-245” (序数日) のような日付をパースできる。期間 (duration) のパースは `iso8601-parse-duration`、間隔 (interval) のパースには `iso8601-parse-interval` がある。これらの関数は最後の関数 (それらのうち開始、終了、期間の 3 つをリターンする) を除いて、すべてデコード済み time 構造をリターンする。

`format-time-string` *format-string* **&optional** *time zone* [Function]

この関数は *time* (省略か `nil`ならカレント時刻) を *format-string*に応じて文字列に変換する。この変換にはタイムゾーンルール *zone* (デフォルトはカレントのタイムゾーンルール) を使用する。Section 40.6 [Time Zone Rules], page 1105 を参照のこと。引数 *format-string*には、時刻を置換する ‘`%`’シーケンスを含めることができる。以下は ‘`%`’シーケンスは何を意味するかのテーブルである:

‘`%a`’ 曜日の短縮名を意味する。

‘`%A`’ 曜日の完全名を意味する。

‘`%b`’ 月の短縮名を意味する。

‘`%B`’ 月の完全名を意味する。

‘`%c`’ ‘`%x %X`’のシノニム。

‘`%C`’ これは世紀、つまり年を 100 で除して小数点以下を切り捨てる。デフォルトのフィールド幅は 2。

‘`%d`’ 0 パディングされた年月日の日。

‘`%D`’ ‘`%m/%d/%y`’のシノニム。

‘`%e`’ ブランクでパディングされた年月日の日。

‘`%F`’ これは ‘`%+4Y-%m-%d`’のような ISO 8601 日付フォーマットと似ているが、‘`+`’と ‘`4`’を任意のフラグとフィールド幅 (6 を減じた後) をオーバーライドする点が異なる。

‘`%g`’ これはその世紀内の ISO 週に対応する年を意味する。

‘`%G`’ これは ISO 週に対応する年を意味する。

‘`%h`’ ‘`%b`’のシノニム。

‘`%H`’ 時分秒の時 (00 から 23) を意味する。

‘`%I`’ 時分秒の時 (01 から 12) を意味する。

‘`%j`’ 年内の経過日 (001 から 366) を意味する。

‘`%k`’ ブランクでパディングされた時分秒の時 (0 から 23) を意味する。

‘`%l`’ ブランクでパディングされた時分秒の時 (1 から 12) を意味する。

‘`%m`’ 年月日の月 (01 から 12) を意味する。

‘`%M`’ 時分秒の分 (00 から 59) を意味する。

‘`%n`’ 改行を意味する。

‘`%N`’ ナノ秒 (000000000–999999999) を意味する。より少ない桁数を求める場合にはミリ秒は ‘`%3N`’、マイクロ秒は ‘`%6N`’を使用する。余分な桁は丸めずに切り捨てられる。

‘%p’ 必要に応じて ‘AM’か ‘PM’を意味する。

‘%q’ これはカレンダーの四半期 (1–4) を意味する (訳注: 会計年度で使用される 4 月 1 日を年度開始日とする四半期ではなく元旦を年度開始日とする四半期)。

‘%r’ ‘%I:%M:%S %p’のシノニム。

‘%R’ ‘%H:%M’のシノニム。

‘%s’ これはエポック以降の経過秒数を表す整数。

‘%S’ これは秒を意味する (00–59、閏年をサポートするプラットフォームでは 00–60)。

‘%t’ タブ文字を意味する。

‘%T’ ‘%H:%M:%S’のシノニム。

‘%u’ これは数字で表した曜日 (1 から 7) で、月曜日が 1。

‘%U’ 週の開始を日曜日とみなした年内の週 (01 から 52)。

‘%V’ これは ISO 8601 にたいする年内の週を意味する。

‘%w’ 数字で表した曜日 (0 から 6) で日曜日が 0。

‘%W’ これは週の開始を月曜日とみなした年内の週 (01 から 52)。

‘%x’ これは locale 固有の意味をもつ。デフォルト locale(C という名前の locale) では ‘%D’と等価。

‘%X’ これは locale 固有の意味をもつ。デフォルト locale(C という名前の locale) では ‘%T’と等価。

‘%y’ 世紀を含まない年 (00 から 99) を意味する。

‘%Y’ 世紀を併なう年を意味する。

‘%Z’ タイムゾーンの短縮形 (たとえば ‘EST’) を意味する。

‘%z’ これはタイムゾーンの数値オフセットを意味する。‘z’の前に 1 つ、2 つ、または 3 つのコロンを前置できる。単なる ‘%z’が ‘-0500’を意味する場合には ‘%:z’は ‘-05:00’、‘%::z’は ‘-05:00:00’を意味する。‘%:::z’は ‘%::z’と同様だが末尾の ‘:00’を無するので、この例では ‘-05’を意味する。

‘%%’ これは単独の ‘%’を意味する。

‘%’の直後には 1 つ以上のフラグ文字を記述してもよい。‘0’は 0、‘+’は 0 をパディングして 5 桁以上の非負の年の前に ‘+’を配置、‘_’は空白によるパディング、‘-’はパディングの抑制、‘^’は英大文字、‘#’は文字の case(大文字小文字) を反転させる。

これらの ‘%’シーケンスすべてにたいしてフィールド幅とパディングのタイプの指定でのきる。これは printfと同じように機能する。フィールド幅は ‘%’シーケンス内の任意のフラグの後に数字として記述するたとえば ‘%S’は分内で経過した秒数を指定するが、‘%03S’は 3 箇所の 0、‘%_3S’は 3 箇所にスペースをパディングすることを意味する。ただの ‘%3S’は 0 でパディングを行う。これは ‘%S’が通常において 2 箇所にパディングする方法だからである。

文字 ‘E’と ‘O’は、‘%’シーケンス内のすべてのフラグとフィールド幅の後に使用されたときは修飾子として作用する。‘E’は日付と時刻にカレント locale の “代替” バージョンの使用を指定する。たとえば日本の locale では、%Exでは日本の元号にもとづく日付フォーマットを得られるだろう。‘E’では ‘%Ec’、‘%EC’、‘%Ex’、‘%EX’、‘%Ey’、‘%EY’の使用が許されている。

‘O’は通常の 10 進数字 (訳注 : アラビア数字) ではなく、カレント locale の数字の代替表現を使用する。これは数字を出力する、ほとんどすべてのアルファベットで使用が許されている。

デバッグプログラム支援のために、認識されない ‘%’シーケンスはそれ自体を意味しており、そのまま出力される。将来の Emacs バージョンでは拡張として新たな ‘%’シーケンスが認識されるかもしれないので、プログラムはこの挙動に頼るべきではない。

この関数は処理のほとんどを行うために C ライブラリー関数 `strftime`を使用している (Section “Formatting Calendar Time” in *The GNU C Library Reference Manual* を参照)。その関数とやり取りするために最初に *time*と *zone*を内部形式に変換する。time の範囲および zone の値はオペレーティングシステムが制限する。この関数は *format-string*もエンコードする。`strftime`が結果文字列をリターンした後に同じコーディングシステムを使用して `format-time-string`はデコードを行う。

`format-seconds` *format-string seconds* [Function]

この関数は引数 *seconds*を *format-string*に応じた年、日、時、... の文字列に変換する。引数 *format-string*には変換を制御する ‘%’シーケンスを指定することができる。以下のテーブルは ‘%’の意味:

‘%y’
‘%Y’ 年間 365 日での年の整数。

‘%d’
‘%D’ 年月日の日。

‘%h’
‘%H’ 時分秒の時の整数。

‘%m’
‘%M’ 時分秒の分の整数。

‘%s’
‘%S’ 時分秒の秒の整数。

‘%z’ 非プリント制御フラグ。これを使用する際には他の指定はサイズ減少順、すなわち年、日、時刻、分、... のように与えなければならない。最初の非 0 変換に遭遇するまで ‘%z’の左側の結果文字列は生成されない。たとえば `emacs-uptime` (Section 40.9 [Processor Run Time], page 1112 を参照) で使用されるデフォルトフォーマットでは、秒数は常に生成されるが年、日、時、分はそれらが非 0 の場合のみ生成されるだろう。

‘%%’ リテラルの ‘%’を生成する。

大文字のフォーマットシーケンスは数字に加えて単位を生成するが、小文字フォーマットは数字だけを生成する。

‘%’に続けてフィールド幅を指定できる。指定したフ幅より短ければブランクでパディングされる。この幅の前にオプションでピリオドを指定すれば、かわりに 0 パディングを要求する。たとえば`"%.3Y"`は`"004 years"`を生成するだろう。

40.9 プロセッサーの実行時間

Emacs は Emacs プロセスにより使用された経過時間 (elapsed time) とプロセッサー時間 (processor time) の両方にたいして、それらをリターンする関数とプリミティブをいくつか提供します。

`emacs-uptime` **&optional** *format* [Command]
この関数は Emacs の *uptime* — この Emacs インスタンスが実行してから経過した実世界における稼動時間を表す文字列をリターンする。この文字列はオプション引数 *format*に応じて `format-seconds`によりフォーマットされる。利用できるフォーマット記述子については Section 40.8 [Time Parsing], page 1109 を参照のこと。*format*が `nil`か省略された場合のデフォルトは`"%Y, %D, %H, %M, %z%S"`。

インタラクティブに呼び出されるとエコーエリアに uptime をプリントする。

`get-internal-run-time` [Function]
この関数は Emacs が使用したプロセッサー実行時間を Lisp タイムスタンプとしてリターンする (Section 40.5 [Time of Day], page 1104 を参照)。

この関数がリターンする値には Emacs がプロセッサーを使用していない時間は含まれないこと、そして Emacs プロセスが複数のスレッドをもつ場合には、すべての Emacs スレッドにより使用されたプロセッサー時間の合計値がリターンされることに注意。

システムがプロセッサー実行時間を判断する方法を提供しなければ`get-internal-run-time`は `current-time`と同じ値をリターンする。

`emacs-init-time` [Command]
この関数は Emacs の初期化 (Section 40.1.1 [Startup Summary], page 1090 を参照) にかかった秒数を文字列としてリターンする。インタラクティブに呼び出された場合にはエコーエリアにプリントする。

40.10 時間の計算

以下の関数は time 値を使用して暦計算を行います (Section 40.5 [Time of Day], page 1104 を参照)。これらの関数で time 値を受け取るすべての引数では、他の time 値と同じように `nil`値はカレントシステム時刻、単一の数値はエポックからの経過秒数を意味します。

`time-less-p` *t1 t2* [Function]
これは time 値 *t1*が time 値 *t2*より小なら `t`をリターンする。引数のいずれかが NaN なら結果は `nil`。

`time-equal-p` *t1 t2* [Function]
これは *t1*と *t2*が等しい time 値なら `t`をリターンする。引数のいずれかが NaN なら結果は `nil`。

`time-subtract` *t1 t2* [Function]
これは 2 つの time 値の差 *t1* − *t2*を Lisptime 値としてリターンする。結果は正確であり、そのクロック解像度が 2 つの引数の解像度より劣ることはない。結果が無限または NaN の場合のみ浮動小数点数。経過秒の単位の差が必要なら、`time-convert`や `float-time`で変換できる。Section 40.7 [Time Conversion], page 1106 を参照のこと。

`time-add` *t1 t2* [Function]
これは`time-subtract`のようないくつかの変換ルールを使用して、2 つの time 値の和を time 値としてリターンする。ここで引数のうち 1 つはある時点での時刻ではなく時間差を表すこと (time 値は経過秒数という単一の数値であることがよくある)。以下はある time 値に秒数を加算する方法:

```
(time-add time seconds)
```

`time-to-days` *time-value* [Function]
この関数は AC. 1 年元日から *time-value*までの間の日数をデフォルトタイムゾーンとみなしてリターンする。time の範囲および zone の値はオペレーティングシステムが制限する。

`time-to-day-in-year` *time-value* [Function]
これは *time-value*に対応する年内の日数をデフォルトタイムゾーンとみなしてリターンする。time の範囲および zone の値はオペレーティングシステムが制限する。

`date-leap-year-p` *year* [Function]
この関数は *year*が閏年なら `t`をリターンする。

`date-days-in-month` *year month* [Function]
*year*の *month*の日数をリターンする。たとえば 2020 年 2 月は 29 日。

`date-ordinal-to-time` *year ordinal* [Function]
*year*の序数日 *ordinal*をデコード済み構造としてリターンする。たとえば 2004 年の 120 日目は 2004 年 4 月 29 日。

40.11 遅延実行のためのタイマー

将来の特定時刻や特定の長さのアイドル時間経過後に関数を呼び出すためにタイマー *(timer)* をセットアップできます。タイマーは次回の呼び出し時刻と呼び出す関数についての情報を格納したスペシャルオブジェクトです。

`timerp` *object* [Function]
この述語関数は `object`がタイマーなら非 `nil`をリターンする。

Emacs は Lisp プログラム内では、任意の時点ではタイマーを実行できません。サブプロセスからの出力が受け入れ可能なときだけ Emacs はタイマーを実行できます。つまり待機中や待機することが可能な `sit-for`や `read-event`のような特定のプリミティブ関数内部でのみタイマーを実行できます。したがって Emacs が busy ならタイマーの実行は遅延するかもしれません。しかし Emacs が idle なら実行される時刻は非常に正確になります。

quit により多くのタイマー関数が物事を不整合な状態に放置し得るので、ターマー関数呼び出し前に Emacs は `inhibit-quit`に `t`をバインドします。ほとんどのタイマー関数は多くの作業を行わないので、これは通常は問題にはなりません。しかし実際には実行に長時間を要する関数を呼び出すタイマーが問題となる恐れがあります。タイマー関数が quit を許容する必要があるなら `with-local-quit`を使用するべきです (Section 21.11 [Quitting], page 413 を参照)。たとえば外部プロセスから出力を受け取るためにタイマー関数が `accept-process-output`を呼び出す場合には、外部プロセスのハング時の *C-g*を確実に機能させるために、その呼び出しを `with-local-quit`内部にラップすべきです。

バッファー内容の変更のためにタイマー関数を呼び出すのは通常は悪いアイデアです。これを行うときには単一のアンドゥエントリーが巨大になるのを防ぐために、通常はバッファーの変更前後で `undo-boundary`を呼び出して、タイマーによる変更とユーザーのコマンドによる変更を分離するべきです。

タイマー関数は `sit-for`のような Emacs に待機を発生させるような関数 (Section 21.10 [Waiting], page 412 を参照) の呼び出しも避けるべきです。その待機中に別のタイマー (同じタイマーとう可能性さえある) が実行され得るので、これは予測不可能な効果を導く恐れがあります。特定時間の経過後に処理される必要があるタイマー関数は、新たなタイマーをスケジュールしてこれを行うことができます。

マッチデータを変更するかもしれない関数を呼び出すタイマー関数はマッチデータの保存とリストアをするべきです。Section 34.6.4 [Saving Match Data], page 901 を参照してください。

`run-at-time` *time repeat function* **&rest** *args* [Command]
これは時刻 *time*に引数 *args*で関数 *function*を呼び出すタイマーをセットアップする。*repeat*が数値 (整数か浮動小数点数) ならタイマーは *time*後の各 *repeat*秒ごとに再実行されるようスケジュールされる。*repeat*が `nil`ならタイマーは 1 回だけ実行される。

*time*には絶対時刻と相対時刻を指定できる。

絶対時刻は限定された種々フォーマットの文字列を使用して指定でき、すでに経過後の時刻であっても当日の時刻とみなされる。認識される形式は '*xxxx*'、'*x*`:`*xx*'、または '*xx*`:`*xx*' (軍用時間)、および '*xx*`am`'、'*xx*`AM`'、'*xx*`pm`'、'*xx*`PM`'、'*xx*`:`*xx*`am`'、'*xx*`:`*xx*`AM`'、'*xx*`:`*xx*`pm`'、'*xx*`:`*xx*`PM`'のいずれか。時と分の部分の区切りはコロンのかわりにピリオドも使用できる。

相対時刻は単位を付加した数字を文字列として指定する。たとえば:

'`1 min`' 現在時刻から 1 分後を表す。

'`1 min 5 sec`'
現在時刻から 65 秒後を表す。

'`1 min 2 sec 3 hour 4 day 5 week 6 fortnight 7 month 8 year`'
現在時刻から丁度 103ヵ月 123 日 10862 秒後を表す。

相対 time 値にたいして Emacs は月を正確に 30 日、年を正確に 365.25 とみなす。

有用なフォーマットのすべてが文字列という訳ではない。*time*が数字 (整数か浮動小数点数) なら秒で数えた相対時刻を指定する。`encode-time`の結果は *time*にたいする絶対時刻の指定にも使用できる。

ほとんどの場合には、*repeat*を最初に呼び出されている際には効果はなく *time*単独で時刻を指定する。例外が 1 つあり *time*が `t`ならエポックから *repeat*の倍数秒ごとに毎回そのタイマーが実行される。これは `display-time`のような関数にとって有用。

関数 `run-at-time`はスケジュール済みの将来の特定アクションを識別する time 値をリターンする。`cancel-timer`(以下参照) の呼び出しにこの値を使用できる。

`run-with-timer` *secs repeat function* **&rest** *args* [Command]
これは正に `run-at-time` と同じだが、これは遅延を秒で指定する際の使用を意図している (パラメーターは `run-at-time`の説明を参照。ただしこの関数は *time*を *secs*として渡している)。

タイマーのリピートは名目上は *repeat*秒ごとに毎回実行されますが、すべてのタイマー呼び出しは遅延する可能性があることを忘れないでください。1 つの繰り返しの遅延が次の繰り返しに影響を与えることはありません。たとえば 3 回分のスケジュール済みのタイマー繰り返しをカバーするほどの計算等により Emacs が busy でも、それらは待機を開始して連続してそのタイマー関数が 3 回呼び出されることになります (それらの間の別のタイマー呼び出しは想定していない)。最後の呼び出しから *n*秒より短くならずにタイマーを再実行したい場合には *repeat*引数を使用しないでください。タイマー関数は、かわりにそのタイマーを明示的に再スケジュールするべきです。

`timer-max-repeats` [User Option]
この変数の値は以前スケジュールされていた呼び出しが止むを得ずに遅延された際に、タイマー関数がリピートによりまとめて呼び出される最大の回数を指定する

`with-timeout` (*seconds timeout-forms...*) *body...* [Macro]
*body*を実行するが *seconds*秒後に実行を諦める。タイムアップ前に *body*が終了したら、`with-timeout`は *body*内の最後のフォームの値をリターンする。ただしタイムアウトにより *body*の実行が打ち切られた場合には、`with-timeout`は *timeout-forms*をすべて実行して最後のフォームの値をリターンする。

このマクロは *seconds*秒後に実行するタイマーをセットすることにより機能する。その時刻の前に *body*が終了したらそのタイマーを削除して、タイマーが実際に実行されたら *body*の実行を終了してから *timeout-forms*を実行する。

Lisp プログラムでは待機を行えるプリミティブをプログラムが呼び出している時のみタイマーを実行できるので、*body*が計算途中の間は`with-timeout`は実行を停止できない— そのプログラムがこれらのプリミティブのいずれかを呼び出したときのみ停止できる。そのため *body*で長時間の計算を行う場合ではなく、入力を待機する場合だけ`with-timeout`を使用すること。

あまりに長時間応答を待機するのを避けるために、関数`y-or-n-p-with-timeout`はタイマーを使用するシンプルな方法を提供します。Section 20.7 [Yes-or-No Queries], page 366 を参照してください。

`cancel-timer` *timer* [Function]
これは *timer*にたいして要求されたアクションをキャンセルする。ここで *timer*はタイマーであること。これは通常は以前に`run-at-time`か`run-with-idle-timer`がリターンしたものである。この関数はこれらの関数の 1 つの呼び出しの効果をキャンセルする。指定した時刻が到来しても特に何も起きないだろう。

`list-timers`コマンドはカレントでアクティブなすべてのタイマーをリストします。表示されたバッファーで利用できるコマンドは `c` (`timer-list-cancel`) の 1 つだけであり、これはポイントのある行のタイマーをキャンセルします。

40.12 アイドルタイマー

以下は Emacs の特定の期間アイドル時に実行するタイマーをセットアップする方法です。それらをセットアップする方法とは別にすればアイドルタイマーは通常のタイマーと同様に機能します。

`run-with-idle-timer` *secs repeat function* **&rest** *args* [Command]
Emacs の次回 *secs*秒間アイドル時に実行するタイマーをセットアップする。*secs*の値には数値、または`current-idle-time`がリターンするタイプの値を指定できる。

*repeat*が`nil`なら、Emacs が充分長い間アイドルになった初回の 1 回だけタイマーは実行される。これは大抵は *repeat*が非`nil`の場合であり、そのときは Emacs が *secs*秒間アイドルになったときに毎回そのタイマーが実行される。

関数`run-with-idle-timer`は`cancel-timer`呼び出し時に使用できるタイマー値をリターンする。

ユーザー入力の待機時に Emacs はアイドル *(idle)* となり、ユーザーが何らかの入力を与えるまでアイドルのままとなります。あるタイマーを 5 秒間のアイドルにセットすると、Emacs が最初に約 5 秒間アイドルになったときにタイマーが実行されます。たとえ *repeat*が非`nil`でも Emacs がアイドルであり続けるかぎりタイマーが再実行されることはありません。アイドル期間は増加を続けて再び 5 秒に減少することはないからです。

アイドル時に Emacs はガーベージコレクションや自動保存やサブプロセスからのデータ処理など、さまざまなことを行うことができます。しかしこれらの幕間劇がアイドルのクロックを 0 にリセットす

ることはないのでアイドルタイマーと干渉することはありません。600 秒にセットされたアイドルタイマーはたとえその 10 分間にサブプロセスの出力が何回到達しても、たとえガーベージコレクションや自動保存が行われてもユーザーコマンドが最後に終了してから 10 分経過後に実行されるでしょう。

ユーザーが入力を与えると Emacs は入力の実行の間は非アイドルになります。それから再びアイドルとなると、繰り返すようにセットアップされたすべてのアイドルタイマーは 1 つずつ異なる時刻に実行されるでしょう。

実行ごとに特定の量を処理するループを含んだり、`(input-pending-p)`が非 `nil`のときに exit するアイドルタイマー関数を記述しないでください。このアプローチはとても自然に見えますが 2 つの問題があります:

- すべてのプロセスの出力をブロックする (Emacs は待機時のみプロセス出力を受け入れるため)。
- その時刻の間に実行されるべきすべてのアイドルタイマーをブロックする。

同様に *secs*引数がカレントのアイドル期間以下となるような、別のアイドルタイマー (同じアイドルタイマーも含む) をセットアップするアイドルタイマー関数を記述しないでください。そのようなタイマーはほとんど即座に実行されて、Emacs が次回アイドルになるのを待機するかわりに再現なく継続して実行されるでしょう。以下で説明するようにカレントのアイドル期間を適切に増加させて再スケジュールするのが正しいアプローチです。

`current-idle-time` [Function]

この関数は Emacs がアイドルなら Emacs がアイドルとなった期間を `current-time`で使用するのと同じフォーマットでリターンする (Section 40.5 [Time of Day], page 1104 を参照)。

Emacs がアイドルでなければ `current-idle-time`は `nil`をリターンする。これは Emacs がアイドルかどうかテストする手軽な方法である。

`current-idle-time`の主な用途はアイドルタイマー関数を少し “休憩” したいときです。そのアイドルタイマー関数はさらに数秒アイドル後に、同じ関数を再呼び出しするために別のタイマーをセットアップできます。以下はその例です:

```
(defvar my-resume-timer nil
  "Timer for `my-timer-function' to reschedule itself, or nil.")

(defun my-timer-function ()
  ;; my-resume-timerアクティブの間にユーザーがコマンドをタイプ
  ;; したら、次回この関数はそれのメインアイドルタイマーから呼び出され
  ;; my-resume-timerを非アクティブにする
  (when my-resume-timer
    (cancel-timer my-resume-timer))
  ...do the work for a while...
  (when taking-a-break
    (setq my-resume-timer
          (run-with-idle-timer
            ;; カレント値より大きいアイドル
            ;; 期間 break-length を計算
            (time-add (current-idle-time) break-length)
            nil
            'my-timer-function))))
```

40.13 端末の入力

このセクションでは端末入力の記録や操作のための関数と変数を説明します。関連する関数については Chapter 39 [Display], page 981 を参照してください。

40.13.1 入力のモード

`set-input-mode` *interrupt flow meta* **&optional** *quit-char* [Function]

この関数はキーボード入力の読み取りにたいしてモードをセットする。Emacs は *interrupt*が非 `nil`なら入力割り込み、`nil`なら CBREAK モードを使用する。デフォルトのセッティングはシステムに依存する。いくつかのシステムでは指定に関わらずに常に CBREAK モードを使用する。

Emacs が X と直接通信する際にはこの引数を無視して、それが Emacs の知る通信手段であれば割り込みを使用する。

*flow*が非 `nil`なら、Emacs は端末への出力にたいして XON/XOFF フロー制御 (`C-q`と `C-s`) を使用する。これは CBREAK 以外では効果がない。

引数 *meta*は 127 より上の文字コード入力にたいするサポートを制御する。*meta*が `t`なら Emacs は 8 番目のビットがセットされた文字をメタ文字に変換する。*meta*が `nil`なら Emacs は 8 番目のビットを無視する。これは端末がそのビットをパリティビットとして使用する場合に必要となる。*meta*が `t`と `nil`のいずれでもなければ、Emacs は入力の 8 ビットすべてを変更せずに使用する。これは 8 ビット文字セットを使用する端末にたいして適している。

*quit-char*が非 `nil`なら quit に使用する文字を指定する。この文字は通常は `C-g`。Section 21.11 [Quitting], page 413 を参照のこと。

`current-input-mode`関数は Emacs がカレントで使用する入力モードのセッティングをリターンします。

`current-input-mode` [Function]

この関数はキーボード入力読み取りにたいするカレントのモードをリターンする。これは `set-input-mode`の引数に対応した `(interrupt flow meta quit)`という形式のリストをリターンする。

interrupt Emacs が割り込み駆動の入力 (interrupt-driven input) を使用時には非 `nil`。`nil`なら Emacs は CBREAK モードを使用している。

flow Emacs が端末出力に XON/XOFF フロー制御 (`C-q`と `C-s`) を使用していれば非 `nil`。この値は *interrupt*が `nil`のときのみ意味がある。

meta Emacs が入力文字の 8 番目のビットをメタ文字として扱う場合には `t`。`nil`は Emacs がすべての入力文字の 8 ビット目をクリアーすることを意味する。その他の値は Emacs が 8 ビットすべてを基本的な文字コードとして使用することを意味する。

quit カレントで Emacs が quit に使用する文字であり通常は `C-g`。

40.13.2 入力の記録

`recent-keys` **&optional** *include-cmds* [Function]

この関数はキーボードかマウスからの最後の入力イベント 300 個を含んだベクターをリターンする。その入力イベントがキーシーケンスに含まれるか否かに関わらずすべての入力イベント

が含まれる。つまりキーボードマクロにより生成されたイベントを含まない、最後の入力イベント 300 個を常に入手することになる (キーボードマクロは、デバッグにとってより興味深いとはいえないので除外されている。そのマクロを呼び出したイベントを確認するだけで充分であるはず)、

*include-cmds*が非 `nil`なら、結果ベクター内の完全なキーシーケンスが `(nil . COMMAND)`という形式の疑似イベントが差し込まれる。ここで *COMMAND*はそのキーシーケンスの開始 (Section 21.1 [Command Overview], page 375 を参照)。

`clear-this-command-keys` (Section 21.5 [Command Loop Info], page 385 を参照) を呼び出すと、その直後はこの関数は空のベクターをリターンする。

`open-dribble-file` *filename* [Command]
この関数は *filename*という名前の *dribble* ファイル *(dribble file)* をオープンする。dribble ファイルがオープンされたとき、キーボードとマウス (ただしキーボードマクロ由来は除く) からのそれぞれの入力イベントはそのファイルに書き込まれる。非文字イベントは '`<...>`'で囲まれたプリント表現で表される。(パスワードのような) 機密情報は dribble ファイルへの記録を終了させることに注意。

引数 `nil`でこの関数を呼び出すことによりファイルはクローズされる。

Section 40.14 [Terminal Output], page 1119 の `open-termscript`も参照のこと。

40.14 端末の出力

端末出力関数は出力をテキスト端末に送信したり、端末に送信した出力を追跡します。変数 `baud-rate` は Emacs が端末の出力スピードをどのように考慮すべきかを指示します。

`baud-rate` [User Option]
この変数は Emacs の認識する端末の出力スピード。この変数をセットしても実際のデータ転送スピードは変化しないが、この値はパディングのような計算に使用される。

これはテキスト端末でスクリーンの一部をスクロールしたり再描画すべきかどうかについての判定にも影響する。グラフィカルな端末での対応する機能については Section 39.2 [Forcing Redisplay], page 981 を参照のこと。

値の単位はボー (baud)。

ネットワークを介して実行中にネットワークの別の部分が違うボーレートで機能している場合には、Emacs がリターンする値はユーザーのローカル端末で使用される値と異なるかもしれません。いくつかのネットワークプロトコルはローカル端末のスピードでリモートマシンと対話するので、Emacs や他のプログラムは正しい値を得ることができますが相手側はそうではありません。Emacs が誤った値をもつ場合には最適よりも劣る判定をもたらします。この問題を訂正するためには `baud-rate`をセットします。

`send-string-to-terminal` *string* **&optional** *terminal* [Function]
この関数は *string*を変更せずに *terminal*へ送信する。*string*内のコントロール文字は端末依存の効果をもつ (端末上に非 ASCII テキストを表示する必要があるなら Section 33.10.7 [Explicit Encoding], page 876 に記述した関数のいずれかを使用してエンコードすること)。この関数はテキスト端末だけを操作する。*terminal*には端末オブジェクト、フレーム、または選択されたフレームの端末を意味する `nil`を指定できる。batch モードでは *terminal*が `nil`なら、*string* は `stdout`に送信される。

この関数の 1 つの用途はダウンロード可能なファンクションキー定義をもつ端末上でファンクションキーを定義することである。たとえば以下は (特定の端末で) ファンクションキー 4 を前方へ 4 文字移動 (そのコンピューターへ文字 *C-u C-f*を送信) するように定義する方法:

```
(send-string-to-terminal "\eF4\^U\^F")
     ⇒ nil
```

`open-termscript` *filename* [Command]

この関数は Emacs が端末へ送信したすべての文字を記録する *termscript* ファイル *(termscript file)* をオープンする。リターン値は `nil`。termscript ファイルは Emacs のスクリーン文字化け問題、不正な Termcap エントリーや、実際の Emacs バグより頻繁に発生する望ましくない端末オプションのセッティングの調査に有用。どの文字が実際に出力されるか確信できれば、それらの文字が使用中の Termcap 仕様に対応するかどうか確実に判断できる。

```
(open-termscript "../junk/termscript")
     ⇒ nil
```

引数 `nil`でこの関数を呼び出すことにより termscript ファイルはクローズされる。

Section 40.13.2 [Recording Input], page 1118 の `open-dribble-file`も参照のこと。

40.15 サウンドの出力

Emacs を使用してサウンドを再生するためには関数 `play-sound`を使用します。特定のシステムだけがサポートされています。実際に処理を行うことができないシステムで `play-sound`を呼び出すとエラーが発生します。

サウンドは RIFF-WAVE フォーマット (‘`.wav`’) か Sun Audio フォーマット (‘`.au`’) で格納されていなければなりません。

`play-sound` *sound* [Function]

この関数は指定されたサウンドを再生する。引数 *sound*は `(sound` *properties*`...)`という形式をもつ。ここで *properties*はキーワード (特定のシンボルが特別に認識される) とそれに対応する値で交互に構成されている。

以下のテーブルは現在のところ *sound*内で意味をもつキーワードとそれらの意味:

`:file` *file*
: これは再生するサウンドを含んだファイルを指定する。絶対ファイル名でなければディレクトリー `data-directory`にたいして展開される。

`:data` *data*
: これはファイルを参照する必要がないサウンドの再生を指定する。値 *data*はサウンドファイルと同じバイトを含む文字列であること。わたしたちはユニバイト文字列の使用を推奨する。

`:volume` *volume*
: これはサウンド再生での音の大きさを指定する。0 から 1 までの数値であること。どんな値であれ以前に指定されたボリュームがデフォルトとして使用される。

`:device` *device*
: これはサウンドを再生するシステムデバイスを文字列で指定する。デフォルトのデバイスはシステム依存。

実際にサウンドを再生する前に `play-sound`はリスト `play-sound-functions`内の関数を呼び出す。関数はそれぞれ 1 つの引数 *sound*で呼び出される。

`play-sound-file` *file* **&optional** *volume device* [Command]
この関数はオプションで *volume*と *device*を指定してサウンド *file*を再生する代替インターフェイス。

`play-sound-functions` [Variable]
リストの関数はサウンド再生前に呼び出される。関数はそれぞれサウンドを記述するプロパティリストを単一の引数として呼び出される。

40.16 X11 キーシンボルの処理

システム固有の X11 keysym(key symbol: キーシンボル) を定義するには変数 `system-key-alist` をセットします。

`system-key-alist` [Variable]
この変数の値はシステム固有の keysym それぞれにたいして 1 つの要素をもつような alist であること。要素はそれぞれ `(`*code* `.` *symbol*`)`という形式をもつ。ここで *code*は数字の keysym コード (ベンダー固有の -2^{28}), のビットは含まない)、*symbol*はそのファンクションキーの名前。

たとえば `(168 . mute-acute)`は数字コード -2^{28} + 168 のシステム固有キーを定義する (HP X サーバーで使用される)。

この alist から他の X サーバーの keysym を除外することは重要ではない。実際に使用中の X サーバーが使用する keysym が競合しないかぎり無害である。

この変数は常にカレント端末にたいしてローカルでありバッファーローカルにできない。Section 29.2 [Multiple Terminals], page 698 を参照のこと。

以下の変数をセットすれば Emacs が修飾キー Control、Meta、Alt、Hyper、Super にたいして何の keysym を使用するべきかを指定できます。

`x-ctrl-keysym` [Variable]
`x-alt-keysym` [Variable]
`x-meta-keysym` [Variable]
`x-hyper-keysym` [Variable]
`x-super-keysym` [Variable]
keysym の名前はそれぞれ修飾子 Control、Alt、Meta、Hyper、Super を意味する名前であること。たとえば以下は Meta 修飾キーと Alt 修飾キーを交換する方法:

```
(setq x-alt-keysym 'meta)
(setq x-meta-keysym 'alt)
```

40.17 batch モード

コマンドラインオプション '`-batch`'で Emacs を非対話的に実行できます。このモードでは Emacs は端末からコマンドを読み取りません。また終端モード (terminal modes) を変更せずに、消去可能なスクリーンへの出力も待ち受けません。これは Lisp プログラムの実行を指示して終了したら Emacs が終了するというアイデアです。これを行うには '`-l` *file*'により *file*という名前のライブラリーをロードするか、'`-f` *function*'により引数なしで *function*を呼び出す、または '`--eval=`*form*'で実行するプログラムを指定できます。

通常はエコーエリアに出力したりストリームとして `t`を指定する `message`や `prin1`等を使用した Lisp プログラムの出力は batch モードでは Emacs の標準記述子へと送られます (`prin1`や他のプリ

ント関数は標準記述子に書き込むが`message`は標準エラー記述子に書き込む)。同様に通常はミニバッファーから読み取られる入力は標準入力から読み取られます。つまり Emacs は非インタラクティブなアプリケーションプログラムのように振る舞います (コマンドのエコーのように Emacs が通常生成するエコーエリアへの出力はすべて抑制される)。

標準出力やエラー記述子に書き込まれる非 ASCII テキストは、`locale-coding-system`が非`nil`ならそれを使用してエンコードされます (Section 33.12 [Locales], page 879 を参照)。`coding-system-for-write`を他のコーディングシステムにバインドすればこれをオーバーライドできます (Section 33.10.7 [Explicit Encoding], page 876 を参照)。

`noninteractive` [Variable]

Emacs が batch モードで実行中ならこの変数は非`nil`。

batch モードで Emacs がエラーのシグナルにより exit すると、Emacs の exit ステータスは非 0 です:

```
$ emacs -Q --batch --eval '(error "foo")'; echo $?
foo
255
```

40.18 セッションマネージャー

Emacs はアプリケーションのサスペンドとリスタートに使用される X セッション管理プロトコル (XSMP: X Session Management Protocol) をサポートしています。X ウィンドウシステムではセッションマネージャー *(session manager)* と呼ばれるプログラムが実行中アプリケーション追跡の責を負います。X サーバーのシャットダウン時にセッションマネージャーはアプリケーションに状態を保存するか尋ねて、それらが応答するまでシャットダウンを遅延します。アプリケーションがそのシャットダウンをキャンセルすることもできます。

セッションマネージャーがサスペンドされたセッションをリスタートする際には、これらのアプリケーションにたいして保存された状態をリロードするように個別に指示します。これはリストアする保存済みセッションが何かを指定する特別なコマンドラインオプションを指定することにより行われます。これは Emacs では‘`--smid` *`session`*’という引数です。

`emacs-save-session-functions` [Variable]

Emacs は`emacs-save-session-functions`と呼ばれるフックを介した状態の保存をサポートする。セッションマネージャーがウィンドウシステムのシャットダウンを告げた際に Emacs はこのフックを実行する。これらの関数はカレントバッファーを一時バッファーにセットして引数なしで呼び出される。それぞれの関数はバッファーに Lisp コードを追加するために`insert`を使用できる。最後に Emacs はセッションファイル *(session file)* と呼ばれるファイル内にそのバッファーを保存する。

その後でセッションマネージャーが Emacs を再開する際に、Emacs はセッションファイルを自動的にロードする (Chapter 16 [Loading], page 269 を参照)。これはスタートアップ中に呼び出される`emacs-session-restore`という名前の関数により処理される。Section 40.1.1 [Startup Summary], page 1090 を参照のこと。

`emacs-save-session-functions`内の関数が非`nil`をリターンすると、Emacs はセッションマネージャーにシャットダウンのキャンセルを要求します。

以下はセッションマネージャにより Emacs がリストアされる際に単に`*scratch*`にテキストを挿入する例です。

```
(add-hook 'emacs-save-session-functions 'save-yourself-test)
```

```
(defun save-yourself-test ()
  (insert "(save-current-buffer
  (switch-to-buffer \"*scratch*\")
  (insert \"I am restored\"))")
  nil)
```

40.19 デスクトップ通知

Emacs は freedesktop.org の Desktop Notifications Specification をサポートするシステムと MS-Windows では通知 *(notifications)* を送ることができます。この機能を POSIX ホストで使用するには Emacs が D-Bus サポート付きでコンパイルされていて、`notifications`ライブラリーがロードされていなければなりません。Section “D-Bus” in *D-Bus integration in Emacs* を参照してください。D-Bus サポートが利用できるときには以下の関数がサポートされます。

`notifications-notify` **&rest** *params* [Function]

この関数は引数 *params*で指定された構成したパラメーターにより D-Bus を通じてデスクトップに通知を送信する。これらの引数は交互になったキーワードと値のペアーで構成されていること。以下はサポートされているキーワードと値:

`:bus` *bus* D-Bus のバス。この引数は`:session`以外のバスを使用する場合のみ必要。

`:title` *title*
通知のタイトル。

`:body` *text*
通知の body のテキスト。通知サーバーの実装に依存して‘`"<b>bold text</b>"`’のような HTML マークアップ、ハイパーリンク、イメージをテキストに含むことができる。HTML 特殊文字は‘`"Contact <postmaster@localhost>!"`’のようにエンコードしなければならない。

`:app-name` *name*
その通知を送信するアプリケーション名。デフォルトは`notifications-application-name`。

`:replaces-id` *id*
この通知が置換する通知の *id*。*id*は`notifications-notify`の以前の呼び出し結果でなければならない。

`:app-icon` *icon-file*
通知アイコンのファイル名。`nil`ならアイコンは表示されない。デフォルトは`notifications-application-icon`。

`:actions` (*key title key title ...*)
適用されるアクションのリスト。*key*と *title*はどちらも文字列。デフォルトのアクション (通常は通知クリックで呼び出される) は‘`"default"`’という名前であること。実装がそれを表示しないようにするには自由だが title は何でもよい。

`:timeout` *timeout*
timeout は通知が表示されてからその通知が自動的にクローズされるまでのミリ秒での時間。−1 なら通知の有効期限は通知サーバーのセッティングに依存して、通知のタイプにより異なるかもしれない。0 なら通知は失効しない。デフォルト値は −1。

`:urgency` *urgency*
: 緊急レベル。`low`、`normal`、`critical`のいずれか。

`:action-items`
: このキーワードが与えられるとアクションの *title*文字列はアイコン名として解釈される。

`:category` *category*
: 通知の種類の文字列。標準のカテゴリーのリストは Desktop Notifications Specification (`https://developer.gnome.org/notification-spec/#categories`) を参照のこと。

`:desktop-entry` *filename*
: これは‘`"emacs"`’のようにプログラムを呼び出すデスクトップファイル名の名前を指定する。

`:image-data (`*width height rowstride has-alpha bits channels data*`)`
: これはそれぞれ width、height、rowstride、および alpha channel、bits per sample、channels、image data の有無を記述する raw データのイメージフォーマット。

`:image-path` *path*
: これは URI(現在サポートされているのは URI スキーマは‘`file://`’のみ)、または‘`$XDG_DATA_DIRS/icons`’にある freedesktop.org 準拠のアイコンテーマ名のいずれかを表す。

`:sound-file` *filename*
: 通知ポップアップ時に再生するサウンドファイルのパス。

`:sound-name` *name*
: 通知ポップアップ時に再生する freedesktop.org サウンド命名仕様準拠のテーマに対応した‘`$XDG_DATA_DIRS/sounds`’にある名前付きサウンド。アイコン名と同様にサウンドにたいしてのみ。例としては‘`"message-new-instant"`’。

`:suppress-sound`
: それが可能ならサーバーにすべてのサウンドの再生を抑制させる。

`:resident`
: セットするとアクション呼び出し時にサーバーは通知を自動的に削除しない。ユーザーか送信者により明示的に削除されるまで通知はサーバー内に常駐し続ける。恐らくこのヒントはサーバーが`:persistence`能力をもつときのみ有用。

`:transient`
: セットするとサーバーはその通知を過渡的なものとして扱い、もしそれが永続的であるべきならサーバーの persistence 能力をバイパスする。

`:x` *position*
`:y` *position*
: その通知がポイントすべきスクリーン上の X と Y の座標を指定する。これらの引数は併せて使用しなければならない。

`:on-action` *function*
: アクション呼び出し時に呼び出す関数。通知 *id*とアクションの *key*は引数としてその関数に渡される。

`:on-close` *function*
: タイムアウトかユーザーにより通知がクローズされたときに呼び出す関数。通知 *id*とクローズ理由 *reason*は引数としてその関数に渡される。:

- 通知が失効した場合は `expired`。
- ユーザーが通知を却下したら `dismissed`。
- `notifications-close-notification`呼び出しにより通知がクローズされたら `close-notification`
- 通知サーバーが理由を提供しなかったら `undefined`。

通知サーバーがどのパラメーターを受け入れるかのチェックは `notifications-get-capabilities`を通じて行うことができる。

この関数は整数の通知 id をリターンする。この id は`notifications-close-notification`や別の `notifications-notify`呼び出しの`:replaces-id`引数で通知アイテムの操作に使用できる。たとえば:

```
(defun my-on-action-function (id key)
  (message "Message %d, key \"%s\" pressed" id key))
     ⇒ my-on-action-function

(defun my-on-close-function (id reason)
  (message "Message %d, closed due to \"%s\"" id reason))
     ⇒ my-on-close-function

(notifications-notify
 :title "Title"
 :body "This is <b>important</b>."
 :actions '("Confirm" "I agree" "Refuse" "I disagree")
 :on-action 'my-on-action-function
 :on-close 'my-on-close-function)
     ⇒ 22

A message window opens on the desktop.  Press ``I agree''.
     ⇒ Message 22, key "Confirm" pressed
        Message 22, closed due to "dismissed"
```

`notifications-close-notification` *id* **&optional** *bus* [Function]
: この関数は識別子 *id*の通知をクローズする。*bus*は D-Bus 接続を表す文字列でありデフォルトは`:session`。

`notifications-get-capabilities` **&optional** *bus* [Function]
: 通知サーバーの能力をシンボルのリストでリターンする。*bus*は D-Bus 接続を表す文字列でありデフォルトは`:session`。以下は期待できる能力:

`:actions` サーバーはユーザーにたいする指定されたアクションを提供する。

`:body` body のテキストをサポートする。

`:body-hyperlinks`
: サーバーは通知内のハイパーリンクをサポートする。

`:body-images`
サーバーは通知内のイメージをサポートする。

`:body-markup`
サーバーは通知内のマークアップをサポートする。

`:icon-multi`
サーバーは与えられたイメージ配列内のすべてのフレームのアニメーションを描画できる。

`:icon-static`
与えられたイメージ配列内の正確に 1 フレームの表示をサポートする。この値は、`:icon-multi`とは相互に排他。

`:persistence`
サーバーは通知の永続性をサポートする。

`:sound` サーバーは通知のサウンドをサポートする。

これらに加えてベンダー固有の能力は`:x-gnome-foo-cap`のように`:x-vendor`で始まる。

`notifications-get-server-information` **&optional** *bus* [Function]
通知サーバーの情報を文字列のリストでリターンする。*bus*は D-Bus 接続を表す文字列でありデフォルトは`:session`。リターンされるリストは (*name vendor version spec-version*)。

name サーバーのプロダクト名。

vendor ベンダー名。たとえば‘`"KDE"`’や‘`"GNOME"`’。

version サーバーのバージョン番号。

spec-version
サーバーが準拠する仕様のバージョン。

*spec_version*が`nil`ならサーバーは‘`"1.0"`’以前の仕様をサポートする。

Emacs が MS-Windows で GUI セッションとして実行時には、ネイティブのプリミティブを通じて D-Bus 通知の小サブセットをサポートします:

`w32-notification-notify` **&rest** *params* [Function]
この関数は *params*の指定にしたがって MS-Windows のトレー通知 (tray notification) を表示する。MS-Windows トレー通知はタスクバーのの通知エリア内のアイコンからのバルーン内に表示される。

値は以下で説明する `w32-notification-close`で通知の削除に使用できる一意な通知 ID。関数が失敗するとリターン値は`nil`。

引数 *params*は keyword/value ペアーで指定する。パラメーターはすべてオプションだが何もパラメーターを指定しなければ関数は何もせずに`nil`をリターンする。

は以下はサポートされるパラメーター:

`:icon` *icon*
システムトレーに *icon*を表示する。*icon*が文字列ならアイコンをロードするファイル名 (Windows のアイコンファイル`.ico`) を指定すること。*icon*が文字列以外、またはこのパラメーターが指定されなければ Emacs の標準アイコンが使用される。

`:tip` *tip*
: 通知のツールチップに *tip*を使用する。*tip*が文字列なら、通知により追加されたトレーアイコン上にマウスポインターを移動した際に表示されるツールチップのテキスト。*tip*が文字列以外またはこのパラメーターが指定されていなければ、ツールチップのデフォルトのテキストは‘`Emacs notification`’。ツールチップのテキストは 127 文字まで (Windows 2000 以前は 63 文字)。それより長い文字列は切り捨てられる。

`:level` *level*
: 通知の重大度レベルで `info`、`warning`、`error`のいずれか。値が与えられた場合には、`:title`パラメーターも指定されて、かつ文字列の場合のみ通知アイコンの左に表示されるアイコンを決定する。

`:title` *title*
: 通知のタイトル。*title*が文字列なら body テキストの直上に大きなフォントで表示される。タイトルのテキストは 63 文字まで。それより長い文字列は切り捨てられる。

`:body` *body*
: 通知の body(本文)。*body*が文字列なら通知メッセージのテキストを指定する。テキストを行に分割する方法を制御するには埋め込みの改行を使用する。body のテキストは 255 文字までで、それより長ければ切り捨てられる。D-Bus とは異なり body テキストはマークアップを含まないプレーンテキストであること。

Windows 2000 以前の Windows では`:icon`と`:tip`だけがサポートされることに注意。他のパラメーターを渡すことは可能だが、それらの古いシステムでは無視されるだろう。

一度にアクティブな通知は最大でも常に 1 つ。新たな通知を表示できるようにするには`w32-notification-close`を呼び出してアクティブな通知を削除しなければならない。

タスクバーから通知とアイコンを削除するには以下の関数を使用します:

`w32-notification-close` *id* [Function]
: この関数は与えられた一意な *id*でトレー通知を削除する。

40.20 ファイル変更による通知

いくつかのオペレーティングシステムはファイル変更にたいするファイルシステムの監視をサポートします。正しく設定されていれば、Emacs は`inotify`、`kqueue`、`gfilenotify`、`w32notify`のようなライブラリーを静的にリンクします。これらのライブラリーによりローカルマシン上でのファイルシステムの監視が有効になります。

リモートマシン上のファイルシステムの監視も可能です。Section “Remote Files” in *The GNU Emacs Manual* を参照してください。これは Emacs にリンク済みのライブラリーのいずれかに依存する訳ではありません。

通知されたファイル変更によりこれらすべてのライブラリーは異なるイベントを発行するので、Emacs は一意な参照を提供するライブラリー`filenotify`を提供しています。ファイル通知を受け取りたい Lisp プログラムは、ネイティブのライブラリーよりこのライブラリーを優先する必要があります:

`file-notify-add-watch` *file flags callback* [Function]
: *file*に関するファイルシステムイベントの監視を追加する。これは *file*に関するファイルシステムイベントが Emacs に報告されるように取り計らう。

リターン値は追加された監視ディスクリプター (descriptor)。タイプは背景にあるライブラリーに依存しており、以下の例に示すとおり整数とみなすことはできない。比較には `equal`を使用すること。

何らかの理由により *file*が監視不可能なら、この関数はエラー `file-notify-error`をシグナルする。

マウントされたファイルシステムでファイル変更を監視できないことがある。これはこの関数により検出されないので、非 `nil`のリターン値が *file*の変更の通知を保証するものではない。

*flags*は何を監視するかセットするためのコンディションのリスト。以下のシンボルを含めることができる:

`change` ファイル変更を監視。

`attribute-change`
: パーミッションや変更時刻のようなファイル属性の変更を監視。

*file*がディレクトリーならディレクトリー内のすべてのファイルの変更が通知される。これは再帰的に機能しない。

Emacs は何らかのイベント発生時には以下の形式の *event*を単一の引数として関数 *callback*を呼び出す:

```
(descriptor action file [file1])
```

*descriptor*はこの関数がリターンするオブジェクトと同じ。*action*はイベントを示し、以下のシンボルのいずれか:

`created` *file*が作成された。

`deleted` *file*が削除された。

`changed` *file*の内容が変更された。`w32notify`ライブラリーでは属性の変更でも同様に報告される。

`renamed` *file*が *file1*にリネームされた。

`attribute-changed`
: *file*の属性が変更された。

`stopped` 監視中の *file*が削除された。

`w32notify`ライブラリーは `attribute-changed`イベントを報告しないことに注意。このライブラリーはパーミッションや変更時刻のようなファイル属性が何か変更された際には `changed`イベントを報告する。同じように `kqueue`ライブラリーでは、ディレクトリー監視時にはファイル属性変更の報告に信頼性がない。

`stopped`イベントはファイル監視の停止を報告する。これは `file-notify-rm-watch`の呼び出された (以下参照)、監視中のファイルが削除された、または背後にあるライブラリーから別のエラーが報告された可能性がある。

*file*と *file1*はイベントが報告されたファイルの名前。たとえば:

```
(require 'filenotify)
     ⇒ filenotify

(defun my-notify-callback (event)
  (message "Event %S" event))
     ⇒ my-notify-callback
```

```
(file-notify-add-watch
  "/tmp" '(change attribute-change) 'my-notify-callback)
     ⇒ 35025468

(write-region "foo" nil "/tmp/foo")
     ⇒ Event (35025468 created "/tmp/.#foo")
        Event (35025468 created "/tmp/foo")
        Event (35025468 changed "/tmp/foo")
        Event (35025468 deleted "/tmp/.#foo")

(write-region "bla" nil "/tmp/foo")
     ⇒ Event (35025468 created "/tmp/.#foo")
        Event (35025468 changed "/tmp/foo")
        Event (35025468 deleted "/tmp/.#foo")

(set-file-modes "/tmp/foo" (default-file-modes))
     ⇒ Event (35025468 attribute-changed "/tmp/foo")
```

アクション `renamed`がリターンされるかどうかは使用する監視ライブラリーに依存する。それ以外では `deleted`と `created`のアクションがランダムな順にリターンされる。

```
(rename-file "/tmp/foo" "/tmp/bla")
     ⇒ Event (35025468 renamed "/tmp/foo" "/tmp/bla")

(delete-file "/tmp/bla")
     ⇒ Event (35025468 deleted "/tmp/bla")
```

`file-notify-rm-watch` *descriptor* [Function]

*descriptor*に指定された既存のファイル監視を削除する。*descriptor*は `file-notify-add-watch`がリターンしたオブジェクトであること。

`file-notify-valid-p` *descriptor* [Function]

*descriptor*で指定された監視の有効性をチェックする。*descriptor*は `file-notify-add-watch`がリターンしたオブジェクトであること。

監視するファイルやディレクトリーの削除や別の理由による監視スレッドの異常 exit により監視が無効になる可能性がある。`file-notify-rm-watch`の呼び出しで削除することにより監視も無効になる。

```
(make-directory "/tmp/foo")
     ⇒ Event (35025468 created "/tmp/foo")

(setq desc
      (file-notify-add-watch
        "/tmp/foo" '(change) 'my-notify-callback))
     ⇒ 11359632

(file-notify-valid-p desc)
     ⇒ t
```

```
(write-region "bla" nil "/tmp/foo/bla")
     ⇒ Event (11359632 created "/tmp/foo/.#bla")
        Event (11359632 created "/tmp/foo/bla")
        Event (11359632 changed "/tmp/foo/bla")
        Event (11359632 deleted "/tmp/foo/.#bla")

;; ディレクトリーのファイル削除では監視は無効にならない
(delete-file "/tmp/foo/bla")
     ⇒ Event (11359632 deleted "/tmp/foo/bla")

(write-region "bla" nil "/tmp/foo/bla")
     ⇒ Event (11359632 created "/tmp/foo/.#bla")
        Event (11359632 created "/tmp/foo/bla")
        Event (11359632 changed "/tmp/foo/bla")
        Event (11359632 deleted "/tmp/foo/.#bla")

;; ディレクトリー削除により監視は無効になる
;; 別の監視ディスクリプターからイベントが到着
(delete-directory "/tmp/foo" 'recursive)
     ⇒ Event (35025468 deleted "/tmp/foo")
        Event (11359632 deleted "/tmp/foo/bla")
        Event (11359632 deleted "/tmp/foo")
        Event (11359632 stopped "/tmp/foo")

(file-notify-valid-p desc)
     ⇒ nil
```

40.21 動的にロードされるライブラリー

ダイナミックにロードされるライブラリー *(dynamically loaded library)* とは機能が最初に必要になったときにオンデマンドでロードされるライブラリーです。Emacs は自身の機能をサポートするライブラリーのオンデマンドロードのように、それらをサポートします。

`dynamic-library-alist` [Variable]

ダイナミックライブラリーとそれらを実装する外部ライブラリーファイルの alist。

要素はそれぞれ (*library files...*) という形式のリスト。ここで `car`はサポートされた外部ライブラリーを表すシンボル、残りはそのライブラリーにたいして候補となるファイル名を与える文字列。

Emacs はリスト内のファイル出現順でライブラリーのロードを試みる。何も見つからなければ Emacs セッションはライブラリーにアクセスできず、それが提供する機能は利用できない。

いくつかのプラットフォーム上におけるイメージのサポートはこの機能を使用している。以下は、S-Windows 上でイメージをサポートするためにこの変数をセットする例:

```
(setq dynamic-library-alist
      '((xpm "libxpm.dll" "xpm4.dll" "libXpm-nox4.dll")
        (png "libpng12d.dll" "libpng12.dll" "libpng.dll"
             "libpng13d.dll" "libpng13.dll")
        (jpeg "jpeg62.dll" "libjpeg.dll" "jpeg-62.dll"
```

```
      "jpeg.dll")
(tiff "libtiff3.dll" "libtiff.dll")
(gif "giflib4.dll" "libungif4.dll" "libungif.dll")
(svg "librsvg-2-2.dll")
(gdk-pixbuf "libgdk_pixbuf-2.0-0.dll")
(glib "libglib-2.0-0.dll")
(gobject "libgobject-2.0-0.dll")))
```

イメージタイプ `pbm`と `xbm`は外部ライブラリーに依存せず Emacs で常に利用可能なので、この変数内にエントリーがないことに注意。

これは外部ライブラリーへのアクセスにたいする一般的な機能を意図したものではないことにも注意。Emacs にとって既知のライブラリーだけがこれを通じてロードできる。

与えられた *library*が Emacs に静的にリンクされていれば、この変数は無視される。

40.22 セキュリティへの配慮

すべてのアプリケーションと同じように、アクセス等のルールを励行するオペレーティングシステムでは、Emacs を安全な環境で実行できます。注意を払えば Emacs ベースのアプリケーションがそのようなルールをチェックするセキュリティ境界の一部になることもできます。Emacs のデフォルトのセッティングでは典型的なソフトウェア開発環境としても良好に機能しますが、アタッカーを含んだ信頼されないユーザーの存在する環境では調整を要します。以下はそのようなアプリケーションを開発する際に助けとなるセキュリティ問題の要覧です。これは完全なものではありません。これはセキュリティチェックリストではなく、セキュリティに関する問題にたいするアイデアを与えることを意図したものです。

ファイルローカル変数
: Emacs が visit するファイルには、そのファイルを visit するバッファーに効果を及ぼす変数のセッティングを含めることができる。Section 12.12 [File Local Variables], page 197 を参照のこと。同じようにディレクトリーはそのディレクトリー内のすべてのファイルに共通なローカル変数を指定できる。Section 12.13 [Directory Local Variables], page 200 を参照のこと。これらの変数の誤用にたいしてたとえ Emacs が幾らかの努力を行っているにしても、あるパッケージがあまりに楽観的に `safe-local-variable`をセットすることによってセキュリティーホールは簡単に作成されるし、これはあまりに一般的な問題である。ファイルとディレクトリーの両方にたいしてこの機能を無効にするには、`enable-local-variables`に `nil`をセットすればよい。

アクセスコントロール
: たとえ Emacs が通常は背後にあるオペレーティングシステムのアクセスパーミッションを尊重するとしても、あるケースにおいてはアクセスを特別に処理する。たとえばファイル名は独自のアクセスチェックによりファイルを特別に扱うハンドラーをもつことができる。Section 25.12 [Magic File Names], page 569 を参照のこと。さらにバッファーは対応するファイルが書き込み可でも読み取り専用にできるしその逆も可能であり、‘`File passwd is write-protected; try to save anyway? (yes or no)`’のようなメッセージを結果としてもたらすかもしれない。Section 27.7 [Read Only Buffers], page 599 を参照のこと。

認証
: Emacs には`read-passwd`のようなパスワードを扱う関数がいくつかある。Section 20.9 [Reading a Password], page 369 を参照のこと。これらの関数はパスワードを公に喧伝しないにしても、Emacs 内部にアクセスする猛者なアタッカーにたいする実装の証

左はない。たとえばパスワード使用後にメモリーをクリアーするために elisp コードが `clear-string`を使用しても、パスワードの残滓は依然としてガーベージコレクトされたフリーリスト内に存在する。Section 4.4 [Modifying Strings], page 55 を参照のこと。

コードインジェクション
: Emacs は他の多くのアプリケーションにコマンドを送信できる。アプリケーションはこれらのコマンドのオペランドとして送信された文字列はディレクティブとして誤解釈しないこと。たとえばファイル *a*を *b*にリネームするシェルコマンドを使用する際に、単に文字列 `mv a b`を使用しないこと。なぜならファイル名のいずれかが '`-`'で始まるかもしれず、'`;`'のようなシェルのメタ文字が含まれるかもしれないから。この種の問題の回避のために `shell-quote-argument`のような関数が助けになるとしても、POSIX プラットフォームの `shell-quote-argument`はシェルのメタ文字はクォートするが先頭の '`-`' のクォートはしない。MS-Windows での '`%`'にたいするクォートでは名前に '`^`'がある環境変数を想定していない。Section 38.2 [Shell Arguments], page 936 を参照のこと。通常はサブシェルより `call-process`を使用するほうが安全である。Section 38.3 [Synchronous Processes], page 937 を参照のこと。そして Emacs のビルトイン関数を使用するほうが安全である。たとえば `mv`を呼び出すかわりに `(rename-file "a" "b" t)`を使用する。Section 25.7 [Changing Files], page 550 を参照のこと。

コーディングシステム
: Emacs はアクセスするファイルとネットワークのコーディングシステムを推察する。Section 33.10 [Coding Systems], page 864 を参照のこと。Emacs の推察が誤っていたりネットワークの相手先が Emacs の推察に不同意なら、結果となるコーディングシステムは信頼できないかもしれない。更にその推察が正しいときでさえ、他のプログラムが使用できないバイトを Emacs が使用できる場合がよくある。たとえば EmacsEmacs にとっては null バイトは他と同じ単なる文字だとしても、他の多くのアプリケーションは null 文字を文字列終端として扱うので、null バイトを含む文字列やファイルを誤って処理する。

Environment and configuration variables
: POSIX は Emacs の挙動に影響し得る環境変数をいくつか指定する。ASCII 英大文字、数字、アンダースコアだけから構成される名前をもつ任意の環境変数が Emacs の内部の動作に影響を及ぼし得る。Emacs はその種の `EMACSLOADPATH`のような変数をいくつか使用する。See Section 16.3 [Library Search], page 272 を参照のこと。Emacs が呼び出すかもしれないユーティリティーすべてにたいして標準の挙動を得るためには、いくつかの環境変数 (`PATH`、`POSIXLY_CORRECT`、`SHELL`、`TMPDIR`) が正しく設定されていることを要するシステムがいくつかある。`TZ`のような一見は無害な変数でさえセキュリティに影響し得る。Section 40.3 [System Environment], page 1100 を参照のこと。

  Emacs にはカスタマイズと同義な変数が他にある。たとえば変数 `shell-file-name`に非標準的な動作を行うシェルを指定すれば、Emacs ベースのアプリケーションはご堂する可能性がある。

Installation
: Emacs のインストールの際にインストール先のディレクトリー階層が信頼できないユーザーに変更可能なら、そのアプリケーションは信頼できない。これは Emacs が使用するプログラムや読み書きするファイルのディレクトリー階層にも適用される。

Network access
: Emacs では多くの場合にネットワークにアクセスするので、通常行うようなネットワークアクセスを回避したいと思うかもしれない。たとえば `tramp-mode`を `nil`にセットしていなければ、特定の構文を使用するファイル名はネットワークファイルとして解釈されて、ネットワーク越しに取得される。*The Tramp Manual* を参照のこと。

Race conditions
: Emacs アプリケーションには、他のアプリケーションが行う競合状態に関するものと同種の問題がある。たとえば `(file-readable-p "foo.txt")`が `t`をリターンしたときでさえ、`file-readable-p`の呼び出しからその時点の間に別のアプリケーションがファイルの権限を変更したために読み取りできないかもしれない。Section 25.6.1 [Testing Accessibility], page 541 を参照のこと。

Resource limits
: Emacs がメモリーや他のオペレーティングシステムのリソースを使い切ったときには、通常は完了まで実行される計算が異常終了でトップレベルに戻るかもしれないので挙動の信頼性が減少し得る。これにより通常は完了する操作を Emacs が放棄するかもしれない。

41 配布用 Lisp コードの準備

Emacs Lisp コードをユーザーに配布するために、Emacs は標準的な方法を提供します。パッケージ *(package)* はユーザーが簡単にダウンロード、インストール、アンインストール、および更新できるような方法でフォーマットと同梱された 1 つ以上のファイルのコレクションです。

以降のセクションではパッケージを作成する方法、およびそれを他の人がダウンロードできるようにパッケージアーカイブ *(package archive)* に配置する方法を説明します。パッケージングシステムのユーザーレベル機能の説明は Section “Packages” in *The GNU Emacs Manual* を参照してください。

41.1 パッケージ化の基礎

パッケージはシンプルパケージ *(simple package)* か複数ファイルパッケージ *(multi-file package)* のいずれかです。シンプルパッケージは単一の Emacs Lisp ファイル内に格納される一方、複数ファイルパッケージは tar ファイル (複数の Lisp ファイルとマニュアルのような非 Lisp ファイルが含まれる可能性がある) に格納されます。

通常の使い方ではシンプルパッケージと複数ファイルパッケージとの違いは比較的重要ではありません。Package Menu インターフェースでは、それらの間に差異はありません。しかし以降のセクションで説明するように作成する手順は異なります。

パッケージ (シンプルか複数ファイル) はそれぞれ特定の属性 *(attributes)* をもっています:

Name
: 短い単語 (たとえば ‘`auctex`’)。これは通常はそのプログラム内でシンボルプレフィクスとしても使用される (Section D.1 [Coding Conventions], page 1162 を参照)。

Version
: 関数 `version-to-list`が理解できる形式のバージョン番号 (たとえば ‘`11.86`’)。パッケージの各リリースではユーザーがパッケージアーカイブの問い合わせでアップグレードとして認識できるようにバージョン番号のアップも行うこと。

Brief description
: そのパッケージが Package Menu にリストされる際にが表示される。理想的には 36 文字以内の単一行であること。

Long description
: これは *`C-h P`* (`describe-package`) により作成されたバッファーに表示されて、その後にそのパッケージの簡単な説明 (brief description) とインストール状態 (installation status) が続く。これには通常はパッケージの能力とインストール後に使用を開始する方法を複数行に渡って完全に記述すること。

Dependencies
: そのパッケージが依存する他のパッケージ (恐らく最低のバージョン番号を含む)。このリストは空でもよく、その場合にはパッケージに依存パッケージがないことを意味する。それ以外ならパッケージをインストールすることにより依存パッケージも自動的かつ再帰的にインストールされる。依存パッケージのいずれかが見つからなければパッケージをインストールすることはできない。

コマンド `package-install-file`、または Package Menu のいずれかを介したパッケージのインストールでは、`package-user-dir`に *`name-version`*という名前のサブディレクトリーが作成されます。ここで *name*はパッケージ名、*version*はバージョン番号です (たとえば `~/.emacs.d/elpa/auctex-11.86/`)。わたしたちはこれをパッケージのコンテンツディレクトリー *(content directory)* と呼んでいます。これは Emacs がパッケージのコンテンツ (シンプルパッ

ケージでは単一の Lisp ファイル、または複数ファイルパッケージから抽出されたファイル) を配置する場所です。

その後に Emacs は autoload マジックコメント (Section 16.5 [Autoload], page 274 を参照) にたいしてコンテンツディレクトリー内のすべての Lisp ファイルを検索します。これらの autoload 定義はコンテンツディレクトリーの *name*`-autoloads.el`という名前のファイルに保存されます。これらは通常はパッケージ内で定義された主要なユーザーコマンドの autoload に使用されますが、`auto-mode-alist`への要素の追加 (Section 23.2.2 [Auto Major Mode], page 468 を参照) 等の別のタスクを行うこともできます。パッケージは通常はその中で定義された関数と変数のすべてを autoload しないことに注意してください— 通常はそのパッケージの使用を開始するために呼び出される一握りのコマンドだけが autoload されます。それから Emacs はそのパッケージ内のすべての Lisp ファイルをバイトコンパイルします。

インストール後はインストールされたパッケージはロード済み *(loaded)* になります。Emacs は`load-path`にコンテンツディレクトリーを追加して *name*`-autoloads.el`内の autoload 定義を評価します。

Emacs のスタートアップ時には、カレントセッションでインストール済みパッケージを利用可能にするために、自動的に関数`package-activate-all`を呼び出します。これは早期 init ファイルロード後、かつ通常 init ファイルロード後に行われます (Section 40.1.1 [Startup Summary], page 1090 を参照)。早期 init ファイルでユーザーオプション `package-enable-at-startup`が `nil`にセットされている場合には、パッケージは自動的に利用可能にはなりません。

`package-activate-all` [Function]

この関数はカレントセッションでパッケージを利用可能にする。ユーザーオプション`package-load-list`は利用可能にするパッケージを指定する。デフォルトではインストール済みのパッージすべてが利用可能になる。Section “Package Installation” in *The GNU Emacs Manual* を参照のこと。

ほとんどの場合には、スタートアップの間に自動的に行われるので`package-activate-all`を呼び出す必要はないはずである。単に早期 init ファイル内に`package-activate-all`の前に実行される必要のあるコードを配置するとともに、`package-activate-all`の後に実行される必要のあるコードを主 linit ファイルに配置することを確実に行なえばよい (Section “Init File” in *The GNU Emacs Manual* を参照)。

`package-initialize` **&optional** *no-activate* [Command]

この関数は何のパッケージがインストール済みかに関する Emacs の内部レコードを初期化してから`package-activate-all`を呼び出す。

オプション引数 *no-activate*が非 `nil`なら、インストール済みパッケージを実際に利用可能にせずにこのレコードを更新する。これは内部でのみ使用される。

41.2 単純なパッケージ

シンプルパッケージは単一の Emacs Lisp ソースファイルで構成されます。このファイルは Emacs Lisp ライブラリーのヘッダー規約に準拠していなければなりません (Section D.8 [Library Headers], page 1172 を参照)。以下の例に示すようにパッケージの属性は種々のヘッダーから取得されます:

```
;;; superfrobnicator.el --- Frobnicate and bifurcate flanges

;; Copyright (C) 2011 Free Software Foundation, Inc.
```

```
;; Author: J. R. Hacker <jrh@example.com>
;; Version: 1.3
;; Package-Requires: ((flange "1.0"))
;; Keywords: multimedia, hypermedia
;; URL: https://example.com/jrhacker/superfrobnicate

...

;;; Commentary:

;; This package provides a minor mode to frobnicate and/or
;; bifurcate any flanges you desire.  To activate it, just type
...

;;;###autoload
(define-minor-mode superfrobnicator-mode
...
```

そのパッケージの名前は 1 行目のファイル名の拡張子を除いた部分と同じです。ここでは‘`superfrobnicator`’です。

brief description(簡単な説明) も 1 行目から取得されます。ここでは ‘`Frobnicate and bifurcate flanges`’です (訳注: ‘`flange` をフロブニケートして二股化する’のフロブニケートとはある技術にたいする無目的で非生産的な具体的行為を意味する)。

バージョン番号は、もしあれば‘`Package-Version`’ヘッダー、それ以外は‘`Version`’ヘッダーから取得されます。これらのヘッダーのいずれかが提供されていなければなりません。ここでのバージョン番号は 1.3 です。

そのファイルに‘`;;; Commentary:`’セクションがあれば、そのセクションは長い説明 (long description) として使用されます (その説明を表示する際には Emacs は‘`;;; Commentary:`’の行とコメント内のコメント文字列を省略する)。

そのファイルに‘`Package-Requires`’ヘッダーがあればパッケージの依存関係 (package dependencies) として使用されます。上の例ではパッケージはバージョン 1.0 以上の‘`flange`’パッケージに依存します。‘`Package-Requires`’ヘッダーの説明は Section D.8 [Library Headers], page 1172 を参照してください。このヘッダーが省略された場合にはパッケージに依存関係はありません。

ヘッダー‘`Keywords`’と‘`URL`’はオプションですが含めることを推奨します。コマンド `describe-package`は出力にリンクを追加するためにこれらを使用します。‘`Keywords`’ヘッダーには `finder-known-keywords`リストからの標準的キーワードを少なくとも 1 つ含めるべきです。

ファイルには Section 41.1 [Packaging Basics], page 1134 で説明したように 1 つ以上の autoload マジックコメントも含めるべきです。上の例ではマジックコメントにより `superfrobnicator-mode` が自動ロードされます。

パッケージアーカイブに単一ファイルのパッケージを追加する方法は Section 41.4 [Package Archives], page 1138 を参照してください。

41.3 複数ファイルのパッケージ

複数ファイルパッケージは単一ファイルパッケージより作成の手軽さが少し劣りますが、より多くの機能を提供します。複数ファイルパッケージには複数の Emacs Lisp ファイル、Info マニュアル、および (イメージのような) 他のファイルタイプを含めることができます。

インストールに先立ち複数パッケージはファイルとしてパッケージアーカイブに含まれます。この tar ファイルは *name*`-`*version*`.tar`という名前でなければなりません。ここで *name*はパッケージ名、*version*はバージョン番号です。tar のコンテンツは一度解凍されたなら、コンテンツディレクトリ *content directory)* である *name*`-`*version*という名前のディレクトリーにすべて解凍されなければなりません (Section 41.1 [Packaging Basics], page 1134 を参照)。このコンテンツディレクトリーのサブディレクトリーにもファイルが抽出されるかもしれません。

このコンテンツディレクトリー内のファイルのうち 1 つは *name*`-pkg.el`という名前のファイルでなければなりません。このファイルには、以下で説明する関数 `define-package`の呼び出しから構成される単一の Lisp フォームを含まなければなりません。これはパッケージの属性、簡単な説明 (brief description)、必要条件 (requirements) を定義します。

たとえば、複数ファイルパッケージとして superfrobnicator のバージョン 1.3 を配布する場合の tar ファイルは `superfrobnicator-1.3.tar`になります。これのコンテンツは `superfrobnicator-1.3`に解凍されて、そのうちの 1 つはファイル `superfrobnicator-pkg.el` になるでしょう。

`define-package` *name version* **&optional** *docstring requirements* [Function]

この関数はパッケージを定義する。*name*はパッケージの名前 (文字列)、*version*は関数 `version-to-list`が理解できる形式のバージョン (文字列)、*docstring*は簡単な説明 (brief description)。

*requirements*は必要となるパッケージとバージョン番号。このリスト内の各要素は `(`*dep-name dep-version*`)`という形式であること。ここで *dep-name*はその依存するパッケージ名が名前であるようなシンボル、*dep-version*は依存するパッケージのバージョン番号 (文字列)。

コンテンツディレクトリーに `README`という名前のファイルがあれば、(すべての ‘`;;; Commentary:`’セクションをオーバーライドして) 長い説明 (long description) として使用されます。

コンテンツディレクトリーに `dir`という名前のファイルがあれば、`install-info`で作成される Info ディレクトリーファイル名とみなされます。Section “Invoking install-info” in *Texinfo* を参照してください。関係のある Info ファイルもコンテンツディレクトリー内に解凍される必要があります。この場合には、パッケージがアクティブ化されたときに Emacs が自動的に `Info-directory-list` にコンテンツディレクトリーを追加します。

パッケージ内に`.elc`ファイルを含めないでください。これらはパッケージのインストール時に作成されます。ファイルがバイトコンパイルされる順序を制御する方法は存在しないことに注意してください。

name`-autoloads.el`という名前のファイルを含めてはなりません。このファイルはパッケージの autoload 定義のために予約済みです (Section 41.1 [Packaging Basics], page 1134 を参照)。これはパッケージのインストール時にパッケージ内のすべての Lisp ファイルから autoload マジックコメントを検索する際に自動的に作成されます。

複数パッケージファイルが、(イメージのような) 補助的なデータファイルを含む場合には、パッケージ内の Lisp ファイルは変数 `load-file-name`を通じてそれらのファイルを参照できます (Chapter 16 [Loading], page 269 を参照)。以下は例です:

```
(defconst superfrobnicator-base (file-name-directory load-file-name))

(defun superfrobnicator-fetch-image (file)
  (expand-file-name file superfrobnicator-base))
```

41.4 パッケージアーカイブの作成と保守

Package Menu を通じてパッケージアーカイブ *(package archives)* からユーザーはパッケージをダウンロードできます。このようなアーカイブは変数 `package-archives`で指定されます。この変数のデフォルト値のデフォルト値として `https://elpa.gnu.org`という GNU プロジェクトがホストするアーカイブが単一のエントリーとして含まれています。このセクションではパッケージアーカイブのセットアップと保守の方法について説明します。

`package-archives` [User Option]

この変数の値は Emacs パッケージマネージャーが認識するパッケージアーカイブのリスト。

この alist の要素はそれぞれが 1 つのアーカイブに対応する `(`*`id . location`*`)`という形式であること。ここで *id*はパッケージ名 (文字列)、*location*は文字列であるようなベースロケーション *(base location)*。

ベースロケーションが '`http:`'か '`https:`'で始まる場合には HTTP(S) の URL として扱われて、(デフォルトの GNU アーカイブのように)HTTP(S) を介してこのアーカイブからパッケージがダウンロードされる。

それ以外ならベースロケーションはディレクトリー名であること。この場合には Emacs は通常のファイルアクセスを通じて、そのアーカイブからパッケージを取得する。local のようなアーカイブは主としてテストに有用。

パッケージアーカイブはパッケージ、および関連するファイルが格納された単なるディレクトリーです。HTTP を介してそのアーカイブに到達できるようにしたければ、このディレクトリーがウェブサーバーにアクセスできなければなりません。Section 41.5 [Archive Web Server], page 1139 を参照してください。

手軽なのは `package-x`を通じてパッケージアーカイブのセットアップと更新を行う方法です。これは Emacs に含まれていますがデフォルトではロードされません。ロードするには *`M-x load-library RET package-x RET`*、または `(require 'package-x)` を init ファイルに追加します。Section "Lisp Libraries" in *The GNU Emacs Manual* を参照してください。

アーカイブ作成後に、それが `package-archives`内になければ Package Menu インターフェースからアクセスできないことを忘れないでください。

公的なパッケージアーカイブの保守には責任が併ないます。アーカイブから Emacs ユーザーがパッケージをインストールする際には、それらのパッケージはそのユーザーの権限において任意のコードを実行できるようになります (これはパッケージにたいしてだけでなく一般的な Emacs コードにたいしても真といえる)。そのためアーカイブの保守を保つとともにホスティングシステムが安全であるよう維持するべきです。

暗号化されたキーを使用してパッケージにサイン *(sign)* するのがパッケージのセキュリティーを向上する 1 つの方法です。gpg の private キーと public キーを生成してあれば以下のようにそのパッケージにサインするために gpg を使用できます:

```
gpg -ba -o file.sig file
```

単一ファイルパッケージにたいしては、*file*はそのパッケージの Lisp ファイルです。複数ファイルパッケージではそのパッケージの tar ファイルです。同じ方法によりアーカイブのコンテンツファイルにもサインできます。これを行うにはパッケージと同じディレクトリーで`.sig`ファイルを利用可能できるようにしてください。ダウンロードする人にたいしても、`https://pgp.mit.edu/`のようなキーサーバーにアップロードすることにより public キーを利用できるようにするべきです。その人がアーカイブからパッケージをインストールする際には署名の検証に public キーを使用できます。

これらの方法についての完全な説明はマニュアルの範囲を超えます。暗号化キーとサインに関する詳細は Section “GnuPG” in *The GNU Privacy Guard Manual*、Emacs に付属する GNU Privacy Guard へのインターフェースについては Section “EasyPG” in *Emacs EasyPG Assistant Manual* を参照してください。

41.5 アーカイブウェブサーバーとのインターフェイス

パッケージアーカイブへのアクセスを提供するウェブサーバーは、以下のクエリーをサポートしなければなりません:

archive-contents
: アーカイブ内容を記述する lisp フォーム。このフォームはリストの最初の要素がアーカイブバージョンであることを除けば’package-desc’ 構造 (`package.el`を参照) のリストである。

<package name>-readme.txt
: パッケージの長い説明 (long description) をリターンする。

<file name>.sig
: そのファイルの署名をリターンする。

<file name>
: そのファイルをリターンする。これは複数ファイルパッケージでは tarball、シンプルパッケージでは単一ファイルかもしれない。

Appendix A Emacs 26 のアンチニュース

時代に逆らって生きるユーザーのために、以下は Emacs バージョン 26.3 へのダウングレードに関する情報です。Emacs 27.1 機能の不在による結果としての偉大なる単純さをぜひ堪能してください。

- C レベルでの Lisp オヴジェクトは再びポインターから整数で実装されるようになりました。これは Emacs Lisp のユーザーにとっては小さなステップかもしれませんが、C レベルで作業する Emacs ユーザーにとっては大きな飛躍です。なぜならデバッガで Lisp オブジェクトを 10 進フォーマットで再び簡単にプリントできるようになったからです (デバッグ中ならとても簡単にできる)。これによりデバッガから Emacs 関数を呼び出すのが簡単になり、C コードで整数と Lisp オブジェクトを混ぜ合せるのが簡単になりました。
- 配布用 tarball からテストスーツを削除しました。わたしたちは今までに実行されたテストは、特にエンドユーザーにとって不必要と信じています。テストを削除することにより tarball のサイズは小さくなり、これは時代を遡るにつれてディスクスペースはより高価になっていくので重要なことなのです。
- ダイナミックモジュールのサポートはデフォルトで無効になりました。これにより Emacs が軽量化される (これ自体が価値あるゴール) とともに、モジュールサポートファイルのインストールに関する混乱と余計な複雑化、ランダム共有オブジェクトを Emacs に共有させて Emacs を混乱させる機会を削除しました。
- インストール済みパッケージは、init ファイルのロード後にのみアクティブ化を行わなければなりません。これには init ファイル内での明示的な `package-initialize`呼び出しが必要です。これはあなたのセッションのどこで、本当にパッケージを利用可能にしたいのかを、あなたに真剣に考えさせる点においてよい事なのです。利便性は素朴さによって踏破されるべきです!
- 重要でなはい機能に関連する Emacs のコード量を削減するために、イメージのネイティブなローテーションとリサイズを削除しました。Emacs 内部でイメージのリサイズやローテートを行いたければ、ImageMagick とともに Emacs をビルドする必要があります。これを失念する人がいるとは思えません。
- XFT フォントバックエンドによるカラーフォントの使用を再び有効化しました。わたしたちはそこかしこでランダムに発生するクラッシュよりも、これらのフォントの可用性のほうが重要だと考えます。特に Emoji を表示するためのこれらのフォントの使用は時代を遡るにつれて重要さを失い、Emacs のより過去のバージョンでは完全に消滅するはずだからです。
- 関数 `network-interface-list`は IPv4 アドレスだけをリターンできるようになりました。わたしたちは IPv6 がもたらす複雑性はその正当性を大きく凌駕すると考えており、その一方で IPv6 は時代を逆行するにつれて完全に消え去ることが予想されていることから、IPv6 の削除は正しい方向に向けたステップなのです。
- 正規表現における繰り返しリミットは $2^{15} - 1$ に削減されました。わたしたちの想定では遠い過去へと向うにつれて、正規表現はよりシンプルになっていくはずです。
- コード単純化と複雑性低減のために、`executable-find`におけるリモートホストでのプログラム検索能力を削除しました。本当にこの機能が必要 (何故に?) なら、いつでも独自のシェルスクリプトを記述してリモートホスト上で実行することができます。
- フェイス属性`:extend`は利用できなくなりました。すべてのフェイスのバックグラウンドカラーは、デフォルトでは行末を超えて拡張されます。これはフェイス管理を有意に簡略化と不必要なコードの増殖の削除、同様にフェイスの理解と使用を大幅に簡略化するはずです。
- `display-blink-cursor-p`および `display-symbol-keys-p`の述語を削除しました。これらが必要になるのは稀であり、`display-graphic-p`のような旧来の実証済みな適切な API 呼び

出しで簡単に置き換えることができます。追加ボーナスとしてこの機能に依存する Lisp プログラムを記述することにより、そのディスプレイ端末において真に必要な機能は何なのかを、プログラマーにより深く理解させることが保証されます。

- 環境変数 `HOME`の値内の相対ディレクトリーは、再びカレントバッファーの `default-directory`に相対的と解釈されるようになりました。こちらのほうがより単純であり、さらにバッファーが異なれば `HOME`が異なる場所に解決されるために、いくつかの興味深いアプリケーションが可能になります。

  同様の理由により、既知のユーザー '`foo`'が存在しなくても、`file-name-absolute-p`は`~foo`を再び絶対ファイル名とみなすようになりました。これはそのようなファイル名を使用する Lisp プログラムが、すべてのシステムにおいて、ユーザーが既知であるかどうかとは無関係に常に同じように機能することを意味します。

- `file-attributes`、`file-modes`、`file-newer-than-file-p`、およびその他のファイル関連のプリミティブは、基盤となる低レベル API の失敗時にはエラーをシグナルするかわりに再び `nil`をリターンするようになりました。エラーをシグナルする関数は、それを使用する Lisp プログラムにたいしてより複雑なコードを要求すると判断するとともに、`nil`をリターンするようにした際にはこの複雑性は不当であることを見出しました。

- 同様に旧スタイルのバッククォートはエラーをシグナルするのではなく、警告を生成するようになりました。これでバッククォートを使用するプログラムからエラー処理を削除することができます。

- 浮動小数点数のフォーマットは、数値が正確にフォーマットできないほど大きい場合にエラーをシグナルするかわりに、予測不可能な値を背景となる実装に生成させることにより高速化されました。わたしたちは Emacs Lisp プログラマーは浮動小数点値を扱う際には、自分が何を行っているかを常に承知していると信じています。

- 関数 `read-char-from-minibuffer`を削除しました。わたしたちは Emacs Lisp ハッカーにとって座して別種のインタラクティブな Q&A 関数を記述して、それを特定のケースごとに最適化するに優る喜びはないという認識の元に、ユーザーにたいして 1 文字の入力を尋ねる必要があるすべての Lisp プログラムにとっては `read-char`で十分なはずだと判断しました。その結果としてそのような応答のヒストリーは提供されません (いずれにせよ、誰が何故に単一キーストロークのヒストリーを望むのか?)。

- 関数 `ngettext`を削除しました。時代を遡るにつれて英語以外の言語は普及しなくなり、それが便利であることは言うまでもないことなので、わたしたちはこの方向に向けて小さなステップを踏にだすとともに、イカしたボーナスとして Emacs を簡略化しました。

- テキストモードフレームでのフォーカス変更通知はもはや認識されず、サポートされません。これで TTY フレーム上のそのような通知を受信する可能性を安全に無視できるようになりました。これは TTY フレーム上でのすべての非文字の入力イベントにたいする Emacs サポートの削除という長い道程に向けた小さな一歩です。

- `face-remapping-alist`内のフェイス仕様は同一バッファーを表示するウィンドウ間の差異とな無関係にバッファー固有であることが要求されます。これにより Emacs から不必要なコードの肥大化を防ぐとともに、フェイス処理がより単純化されます。

- フォーマット '`%o`'および '`%x`'は、あなたが期待通り常に符号なしの値を生成するようになりました。これによってあなたは背後にあるアーキテクチャーごとに異なるマシン表現 (わたしたちはこれを何か価値のある機能とみなしている) を暴くことが可能になりました。

- わたしたちはもはや U+2018 のような紛らわしい引用符を含むシンボルを `font-lock-warning-face`でハイライトしません。これらの検出には無視できない量のコードが必要であり、Lisp プ

ログラマーは自分が何をしているか常に承知していて、目障りで紛らわしい書体に煩わされる必要などないとわたしたちは固く信じています。

- 関数 `file-system-info`は `df`呼び出しと出力の解析が常に可能なことから、Posix プラットフォームで削除しました。
- ‘`base64url`’エンコーディングを実装する関数は、通常の base-64 エンコーディングを適切に調整することで常にエミュレートできるので削除しました。Emacs の肥大化、およびこれのために Lisp プログラマーにさらなるインターフェースの学習を強いる必要はありません。
- 簡略化にたいする継続要求の一環として、他の多くの関数と変数が排除されました。

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
`https://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix C GNU General Public License

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. https://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see https://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.

Appendix D ヒントと規約

このチャプターでではEmacs Lispの追加機能については説明しません。かわりに以前のチャプターで説明した機能を効果的に使う方法、およびEmacs Lispプログラマーがしたがうべき慣習を説明します。

以降で説明する慣習のいくつかはLispファイルのvisit時にコマンド*M-x checkdoc RET*を実行することにより自動的にチェックできます。これはすべての慣習はチェックできませんし、与えられた警告すべてが必ずしも問題に対応する訳ではありませんが、それらすべてを検証することには価値があります。カレントバッファーの慣習をチェックするにはコマンド*M-x checkdoc-current-buffer RET*、(たとえば*M-x compile RET*で実行するコマンドとともに)batchモードでファイルをチェックしたければ`checkdoc-file`をかわりに使用してください。

D.1 Emacs Lisp コーディング規約

以下は幅広いユーザーを意図したEmacs Lispコードを記述する際にしたがうべき慣習です:

- 単なるパッケージのロードがEmacsの編集の挙動を変更するべききではない。コマンド、その機能を有効や無効にするコマンド、その呼び出しが含まれる。

  この慣習はカスタム定義を含むすべてのファイルに必須である。そのようなファイルを慣習にしたがうために修正するのが非互換の変更を要するなら、構うことはないから非互換の修正を行うこと。先送りにしてはならない。

- 他のLispプログラムと区別するための短い単語を選択すること。あなたのプログラム内のグローバルなシンボルすべて、すなわち変数、定数、関数の名前はその選択したプレフィクスで始まること。そのプレフィクスと名前の残りの部分はハイフン'-'で区切る。Emacs Lisp内のすべてのグローバル変数は同じネームスペース、関数はすべて別のネームスペースを共有するので、これの実践は名前の競合を回避する[1]。他のパッケージから使用されることを意図しない場合にはプレフィクス名前を2つのハイフンで区切ること。

  ユーザーの使用を意図したコマンド名では、何らかの単語がそのパッケージ名のプレフィクスの前にあると便利なことがある。たとえばわたしちの慣習ではオブジェクトをリストするコマンドの名前なら'`list-`*`something`*'、すなわち'`frob`'と呼ばれるパッケージのグローバルシンボルが'`frob-`'で始まれば'`list-frobs`'というコマンドをもつかもしれない。さらに関数や変数等を定義する構文が'`defun`'や'`defvar`'で始まればより良く機能するので、名前内でそれらの後に名前プレフィクスを置くこと。

  この勧告は`copy-list`のようなEmacs Lisp内のプリミティブではなく、伝統的なLispプリミティブにさえ適用される。信じようと信じまいと`copy-list`を定義する尤もらしい方法は複数あるのだ。安全第一である。かわりに`foo-copy-list`や`mylib-copy-list`のような名前を生成するために、あなたの名前プレフィクスを追加しよう。

  `twiddle-files`のような特定の名前でEmacsに追加されるべきだと考えている関数を記述する場合には、プログラム内でそれを名前で呼び出さないこと。プログラム内ではそれを`mylib-twiddle-files`で呼び出して、わたしたちがそれをEmacsに追加するため提案メールを、'`bug-gnu-emacs@gnu.org`'に送信すること。もし追加することになったときに、わたしたちは十分容易にその名前を変更できるだろう。

  1つのプレフィクスで十分でなければ、それらに意味があるかぎり、あなたのパッケージは2つか3つの一般的なプレフィクス候補を使用できる。

[1] Common Lispスタイルのパッケージシステムの恩恵はコストを上回るとは考えられない。

- 個々の Lisp ファイルすべての終端に `provide`呼出を配置すること。Section 16.7 [Named Features], page 279 を参照のこと。
- 事前に他の特定の Lisp プログラムのロードを要するファイルはファイル先頭のコメントでそのように告げるべきである。また、それらが確実にロードされるように `require`を使用すること。Section 16.7 [Named Features], page 279 を参照のこと。
- ファイル *foo*が別のファイル *bar*内で定義されたマクロを使用するが、*bar*内の他の関数や変数を何も使用しない場合には *foo*に以下の式を含めること:

  ```
  (eval-when-compile (require 'bar))
  ```

  これは *foo*のバイトコンパイル直前に *bar*をロードするよう Emacs に告げるので、そのマクロはコンパイル中は利用可能になる。`eval-when-compile`の使用によりコンパイル済みバージョンの *foo*が中古なら *bar*のロードを避けられる。これはファイル内の最初のマクロ呼び出しの前に呼び出すこと。Section 14.3 [Compiling Macros], page 244 を参照のこと。
- 実行時に本当に必要でなければ、追加ライブラリーのロードを避けること。あなたのファイルが単に他のいくつかのライブラリーなしでは機能しないなら、トップレベルでそのライブラリーを単に `require`してこれを行うこと。しかしあなたのファイルがいくつかの独立した機能を含み、それらの 1 つか 2 つだけが余分なライブラリーを要するなら、トップレベルではなく関連する関数内部への `require`の配置を考慮すること。または必要時に余分のライブラリーをロードするために `autoload`ステートメントを使用すること。この方法ではあなたのファイルの該当部分を使用しない人は、余分なライブラリーをロードする必要がなくなる。
- Common Lisp 拡張が必要なら古い `cl`ライブラリーではなく、`cl-lib`ライブラリーを使うこと。`cl`ライブラリーはクリーンなネームスペースを使用しない (定義が‘`cl-`’で始まらない)。パッケージが実行時に `cl`をロードする場合には、そのパッケージを使用しないユーザーにたいして名前の衝突を起こすかもしれない。

  `(eval-when-compile (require 'cl))`でコンパイル時に `cl`を使用するのは問題ない。コンパイラーはバイトコードを生成する前にマクロを展開するので `cl`内のマクロを使用するには十分である。ただしこの場合においても現代的な `cl-lib`を使用するほうが良い。
- メジャーモードを定義する際にはメジャーモードの慣習にしたがってほしい。Section 23.2.1 [Major Mode Conventions], page 464 を参照のこと。
- マイナーモードを定義する際にはマイナーモードの慣習にしたがってほしい。Section 23.3.1 [Minor Mode Conventions], page 480 を参照のこと。
- ある関数の目的が特定の条件の真偽を告げることであるなら、(述語である “predicate” を意味する) ‘`p`’で終わる名前を与えること。その名前が 1 単語なら単に ‘`p`’、複数単語なら ‘`-p`’を追加する。例は `framep`や `frame-live-p`。変数が述語関数でないなら、この`-p`サフィックスの使用を避けるよう推奨する。かわりに`-flag`サフィックスや `is-foo`のような名前を使用すること。
- ある変数の目的が単一の関数の格納にあるなら、‘`-function`’で終わる名前を与えること。ある変数の目的が関数のリストの格納にあるなら (たとえばその変数がフックなら)、フックの命名規約にしたがってほしい。Section 23.1 [Hooks], page 461 を参照のこと。
- そのファイルをロードすることによりフックに関数が追加されるなら、*feature*`-unload-function`という関数を定義すること。ここで *feature*はパッケージが提供する機能の名前であり、そのような変更をアンドゥするためのフックにする。そのファイルのアンロードに `unload-feature`を使用することにより、この関数が実行されるようになる。Section 16.9 [Unloading], page 282 を参照のこと。
- Emacs のプリミティブにエイリアスを定義するのは悪いアイデアである。かわりに通常は標準の名前を使用すること。エイリアスが有用になるかもしれないケースは後方互換性や可搬性を向上させる場合である。

- パッケージで別のバージョンの Emacs にたいする互換性のためにエイリアスや新たな関数の定義が必要なら、別のバージョンにあるそのままの名前ではなくパッケージのプレフィクスを名前に付加すること。以下はそのような互換性問題を多く提供する Gnus での例。

```
(defalias 'gnus-point-at-bol
  (if (fboundp 'point-at-bol)
      'point-at-bol
    'line-beginning-position))
```

- Emacs のプリミティブの再定義や advise は悪いアイデアである。これは特定のプログラムには正しいことを行うが結果として他のプロラムが破壊されるかもしれない。
- 同様にある Lisp パッケージで別の Lisp パッケージ内の関数に advise するのも悪いアイデアである。
- ライブラリやパッケージでの `eval-after-load`と `with-eval-after-load`の使用を避けること (Section 16.10 [Hooks for Loading], page 283 を参照)。この機能は個人的なカスタマイズを意図している。Lisp プログラム内でこれを使用すると別の Lisp 内ではそれが見えず、その挙動を変更するために不明瞭になる。これは別のパッケージ内の関数への advise と同様にデバッグの障害になる。
- Emacs の標準的な関数やライブラリープログラムの何かをファイルが置換するなら、そのファイル冒頭の主要コメントでどの関数が置換されるか、置換によりオリジナルと挙動がどのように異なるかを告げること。
- 関数や変数を定義するコンストラクターは関数ではなくマクロにして名前は '`define-`'で始まること。そのマクロは定義される名前を 1 つ目の引数で受け取ること。これは自動的に定義を探す種々のツールの助けとなる。マクロ自身の中でその名前を構築するのは、それらのツールを混乱させるので避けること。
- 別のいくつかのシステムでは '`*`'が先頭や終端にある変数名を選択する慣習がある。Emacs Lisp ではその慣習を使用しないので、あなたのプログラム内でそれを使用しないでほしい (Emacs では特別な目的をもつバッファーだけにそのような名前を使用する)。すべてのライブラリーが同じ慣習を使用するなら人は Emacs がより整合性があることを見い出すだろう。
- Emacs Lisp ソースファイルのデフォルトのファイルコーディングシステムは UTF である (Section 33.1 [Text Representations], page 852 を参照)。あなたのプログラムが UTF-8 以外の文字を含むような稀なケースでは、ソースファイル内の '`-*-`'行かローカル変数リスト内で適切なコーディングシステムを指定すること。Section "Local Variables in Files" in *The GNU Emacs Manual* を参照のこと。
- デフォルトのインデントパラメーターでファイルをインデントすること。
- 自分で行に閉カッコを配置するのを習慣としてはならない。Lisp プログラマーはこれに当惑させられる。
- コピーを配布する場合は著作権表示と複製許可表示を配置してほしい。Section D.8 [Library Headers], page 1172 を参照のこと。

D.2 キーバインディング規約

- Dired、Info、Compilation、Occur などの多くのメジャーモードではハイパーリンクを含む読み取り専用テキストを処理するようデザインされている。そのようなメジャーモードはリンクをフォローするように *`mouse-2`*と `RET`を再定義すること。そのリンクが `mouse-1-click-follows-link`にしたがうように `follow-link`条件もセットアップすること。Section 32.19.8 [Clickable

Text], page 828 を参照のこと。そのようなクリック可能リンクを実装する簡便な手法については Section 39.19 [Buttons], page 1068 を参照のこと。

- Lisp プログラム内のキーとして *C-c letter*を定義してはならない。*C-c*とアルファベット (大文字小文字の両方) からなるシーケンスはユーザー用に予約済みである。これらはユーザー用として唯一予約されたシーケンスなので阻害してはならない。

  すべてのメジャーモードがこの慣習を尊重するよう変更するには多大な作業を要する。この慣習を捨て去ればそのような作業は不要になりユーザーは不便になるだろう。この慣習を遵守してほしい。
- 修飾キーなしの F5から F9までのファンクションキーもユーザー定義用に予約済み。
- 後にコントロールキーか数字が続く *C-c*シーケンスはメジャーモード用に予約済みである。
- 後に*{*、*}*、*<*、*>*、*:*、*;*が続く *C-c*シーケンスもメジャーモード用に予約済み。
- 後に他の ASCII区切り文字やシンボル文字が続く *C-c*シーケンスはマイナーモードに割り当てられている。メジャーモード内でのそれらの使用は絶対禁止ではないが、もしそれを行えばそのメジャーモードがマイナーモードにより時々シャドーされるかもしれない。
- 後にプレフィクス文字 (*C-c*を含む) が続く *C-h*をバインドしてはならない。*C-h*をバインドしなければ、そのプレフィクス文字をもつサブコマンドをリストするためのヘルプ文字として自動的に利用可能になる。
- 別の ESCが後に続く場合を除き ESCで終わるキーシーケンスをバインドしてはならない (つまり *ESC ESC*で終わるキーシーケンスのバインドは OK)。

  このルールの理由は任意のコンテキストにおける非プレフィクスであるような ESCのバインディングは、そのコンテキストにおいてファンクションキーとなるようなエスケープシーケンスの認識を阻害するからである。
- 同様に *C-g*は一般的にはキーシーケンスのキャンセルに使用されるので、*C-g*で終わるキーシーケンスをバインドしてはならない。
- 一時的なモードやユーザーが出入り可能な状態のような動作は、すべてエスケープ手段として *ESC ESC*か *ESC ESC ESC*を定義すること。

  通常の Emacs コマンドを受け入れる状態、より一般的には後にファンクションキーか矢印キーが続く ESC内のような状態は潜在的な意味をもつので *ESC ESC*を定義してはならない。なぜならそれは ESCの後のエスケープシーケンスの認識を阻害するからである。これらの状態においては、エスケープ手段として *ESC ESC ESC*を定義すること。それ以外ならかわりに *ESC ESC*を定義すること。

D.3 Emacs プログラミングのヒント

以下の慣習にしたがうことによりあなたのプログラムが実行時により Emacs に適合するようになります。

- プログラム内で `next-line`や `previous-line`を使用してはならない。ほとんど常に `forward-line`のほうがより簡便であり、より予測可能かつ堅牢である。Section 30.2.4 [Text Lines], page 758 を参照のこと。
- あなたのプログラム内でマークのセットが意図した機能でないなら、マークをセットする関数を呼び出してはならない。マークはユーザーレベルの機能なので、ユーザーの益となる値を提供する場合を除きマークの変更は間違いである。Section 31.7 [The Mark], page 772 を参照のこと。

  特に以下の関数は使用しないこと:

  - `beginning-of-buffer`、`end-of-buffer`

- `replace-string`、`replace-regexp`
- `insert-file`、`insert-buffer`

インタラクティブなユーザーを意図した別の機能がないのにポイントの移動、特定の文字列の置換、またはファイルやバッファーのコンテンツを挿入したいだけなら単純な 1、2 行の Lisp コードでそれらの関数を置き換えられる。

- ベクターを使用する特別な理由がある場合を除きベクターではなくリストを使用すること。Lisp ではベクターよりリストを操作する機能のほうが多く、リストを処理するほうが通常は簡便である。

  要素の挿入や削除がなく (これはリストだけで可能)、ある程度のサイズがあって、(先頭か末尾から検索しない) ランダムアクセスがあるテーブルではベクターが有利。

- エコーエリア内にメッセージを表示する推奨方法は`princ`ではなく`message`関数。Section 39.4 [The Echo Area], page 983 を参照のこと。
- エラーコンディションに遭遇したときは関数 `error` (または `signal`) を呼び出すこと。関数`error`はリターンしない。Section 11.7.3.1 [Signaling Errors], page 165 を参照のこと。

  エラーの報告に `message`、`throw`、`sleep-for`、`beep`を使用しないこと。

- エラーメッセージは大文字で始まり、ピリオドで終わらないこと。
- ミニバッファー内で `yes-or-no-p`か `y-or-n-p`で答えを求める質問を行う場合には大文字で始めて‘`? `’で終わること。
- ミニバッファーのプロンプトでデフォルト値を示すときは、カッコ内に単語‘`default`’を配置すること。これは以下のようになる:

```
Enter the answer (default 42):
```

- `interactive`で引数リストを生成する Lisp 式を使用する場合には、リージョンやポジションの引数にたいして正しいデフォルト値を生成しようと試みではならない。それらの引数が指定されていなければかわりに `nil`を提供して、引数が `nil`のときに関数の body でデフォルト値を計算すること。たとえば以下のように記述する:

```
(defun foo (pos)
  (interactive
   (list (if specified specified-pos)))
  (unless pos (setq pos default-pos))
  ...)
```

以下のようにはしない:

```
(defun foo (pos)
  (interactive
   (list (if specified specified-pos
             default-pos)))
  ...)
```

これはそのコマンドを繰り返す場合に、そのときの状況にもとづいてデフォルト値が再計算されるからである。

interactive の‘`d`’、‘`m`’、‘`r`’指定を使用する際にはコマンドを繰り返すときの引数値の再計算にたいして特別な段取りを行うので、このような注意事項を採用する必要はない。

- 実行に長時間を要する多くのコマンドは開始時に‘`Operating...`’、完了時に‘`Operating...done`’のような何らかのメッセージを表示すること。これらのメッセージのスタイルは‘`...`’の周囲にスペースを置かず、‘`done`’の後にピリオドを置かないよう一定に保って

ほしい。そのようなメッセージを生成する簡便な方法は Section 39.4.2 [Progress], page 985 を参照のこと。

- 再帰編集の使用を避けること。かわりに Rmail の `e`コマンドが行うように、元のローカルキーマップに戻るよう定義したコマンドを含んだ新たなローカルキーマップを使用するか、単に別のバッファーにスイッチしてユーザーが自身で戻れるようにすること。Section 21.13 [Recursive Editing], page 417 を参照のこと。

D.4 コンパイル済みコードを高速化ためのヒント

以下はバイトコンパイル済み Lisp プログラムの実行速度を改善する方法です。

- 時間がどこで消費されているか見つかるためにプログラムのプロファイルを行う。Section 18.5 [Profiling], page 329 を参照のこと。
- 可能なら常に再帰ではなく繰り返しを使用する。Emacs Lisp ではコンパイル済み関数が別のコンパイル済み関数を呼び出すときでさえ関数呼び出しは低速である。
- プリミティブのリスト検索関数 `memq`、`member`、`assq`、`assoc`は明示的な繰り返しより更に高速である。これらの検索プリミティブを使用できるようにデータ構造を再配置することにも価値が有り得る。
- 特定のビルトイン関数は通常の関数呼び出しの必要を回避するようにバイトコンパイル済みコードでは特別に扱われる。別の候補案のかわりにこれらの関数を使用するのは良いアイデアである。コンパイラーにより特別に扱われる関数かどうかを確認するには `byte-compile`プロパティを調べればよい。そのプロパティが非 `nil`ならその関数は特別に扱われる。

  たとえば以下を入力すると `aref`が特別にコンパイルされえることが示される (Section 6.3 [Array Functions], page 107 を参照):

  ```
  (get 'aref 'byte-compile)
       ⇒ byte-compile-two-args
  ```

  この場合 (および他の多くの場合) には、最初に `byte-compile`プロパティを定義する `bytecomp` ライブラリーをロードしなければならない。
- プログラム内で実行時間のある程度を占める小さい関数を呼び出すなら関数を inline にする。これにより関数呼び出しのオーバーヘッドがなくなる。関数の inline 化はプログラム変更の自由度を減少させるのでユーザーがスピードを気にするに足るほど低速であり、inline 化により顕著に速度が改善されるのでなければ行ってはならない。Section 13.13 [Inline Functions], page 235 を参照のこと。

D.5 コンパイラー警告を回避するためのヒント

- 以下のようにダミーの `defvar`定義を追加して未定義のフリー変数に関するコンパイラーの警告の回避を試みる:

  ```
  (defvar foo)
  ```

  このような定義はファイル内での変数 `foo`の使用にたいしてコンパイラーが警告しないようにする以外に影響はない。
- 同様に `declare-function`ステートメントを使用して、定義されるこが既知な未定義関数に関するコンパイラーの警告の回避を試みる (Section 13.15 [Declaring Functions], page 238 を参照)。
- 特定のファイルから多くの関数、マクロ、変数を使用する場合には、それらに関するコンパイラー警告を回避するために、以下のようにパッケージに `require`を追加できる (Section 16.7 [Named Features], page 279 を参照):

```
(require 'foo)
```

何らかののファイルのマクロだけが必要ならコンパイル時だけ require できる (Section 17.5 [Eval During Compile], page 289 を参照)。たとえば、

```
(eval-when-compile
  (require 'foo))
```

- ある関数内で変数をバインドして別の関数内で使用やセットする場合には、その変数が定義をもたなければ別関数に関してコンパイラーは警告を行う。しかしその変数が短い名前をもつ場合には、Lisp パッケージは短い変数名を定義するべきではないので定義の追加により不明瞭になるかもしれない。行うべき正しい方法はパッケージ内の他の関数や変数に使用されている名前プレフィクスで始まるように変数をリネームすることである。
- 警告を回避する最後の手段は通常なら間違いであるが、その使用法では間違いではないと解っている何かを行う際には `with-no-warnings`の内側に置くこと。Section 17.6 [Compiler Errors], page 290 を参照のこと。

D.6 ドキュメント文字列のヒント

以下はドキュメント文字列記述に関するいくつかのヒントと慣習です。コマンド *M-x checkdoc-minor-mode*を実行すれば慣習の多くをチェックできます。

- ユーザーが理解することを意図したすべての関数、コマン、変数はドキュント文字列をもつこと。
- Lisp プログラムの内部的な変数とサブルーチンは同様にドキュメント文字列をもつことができる。ドキュメント文字列は実行中の Emacs 内で非常に僅かなスペースしか占めない。
- 80 列スクリーンの Emacs ウィンドウに適合するようにドキュメント文字列をフォーマットすること。ほとんどの行を 60 文字以下に短くするのは良いアイデアである。最初の行は 67 文字以下にすること。さもないと `apropos`の出力で見栄えが悪くなる。

  見栄えがよくなるならそのテキストをフィルできる。Emacs Lisp モードは `emacs-lisp-docstring-fill-column`で指定された幅にドキュメント文字列をフィルする。しかしドキュメント文字列の行ブレークを注意深く調整すればドキュメント文字列の可読性をより向上できることがある。ドキュメント文字列が長い場合にはセクション間に空行を使用すること。
- ドキュンメント文字列の最初の行は、それ自身が要約となるような 1 つか 2 つの完全なセンテンスから成り立つこと。*M-x apropos*は最初の行だけを表示するので、その行のコンテンツが自身で完結していなければ結果の見栄えは悪くなる。特に最初の行は大文字で始めてピリオドで終わること。

  関数では最初の行は “その関数は何を行うのか?”、変数にたいしては最初の行は “その値は何を意味するのか?” という問いに簡略に答えること。

  ドキュメント文字列を 1 行に制限しないこと。その関数や変数の使用法の詳細を説明する必要に応じてその分の行数を使用すること。テキストの残りの部分にたいしても完全なセンテンスを使用してほしい。
- ユーザーが無効化されたコマンドの使用を試みる際には、Emacs はそれのドキュメント文字列の最初のパラグラフ (最初の空行までのすべて) だけを表示する。もし望むなら、その表示をより有用になるように最初の空行の前に何の情報を含めるか選択できる。
- 最初の行ではその関数のすべての重要な引数と、関数呼び出しで記述される順にそれらに言及すること。その関数が多くの引数をもつなら最初の行でそれらすべてに言及するのは不可能である。この場合にはもっとも重要な引数を含む最初の引数数個について最初の行で言及すること。

- ある関数のドキュメント文字列がその関数の引数の値に言及する際には、引数を大文字にした名前が引数の値であるかのように使用すること。つまり関数 `eval`のドキュメント文字列では最初の引数の名前が `form`なので ‘`FORM`’で参照する:

  ```
  Evaluate FORM and return its value.
  ```

  同様にリストやベクターのサブユニットへの分解で、それらのいくつかを異なるように示すような際にはメタ構文変数 (metasyntactic variables) を大文字で記述すること。以下の例の ‘`KEY`’ と ‘`VALUE`’はこれの実践例:

  ```
  The argument TABLE should be an alist whose elements
  have the form (KEY . VALUE).  Here, KEY is ...
  ```

- ドキュメント文字列内で Lisp シンボルに言及する際には case(大文字小文字) を絶対に変更しないこと。そのシンボルの名前が `foo`なら “Foo” ではなく “foo”(“Foo” は違うシンボル)。

  これは関数の引数の値の記述ポリシーと反するように見えるかもしれないが矛盾は実際には存在しない。引数の *value* はその関数が値の保持に使用する *symbol* と同じではない。

  これによりセンテンス先頭に小文字を置くことになり、それが煩しいならセンテンス開始がシンボルにならないようにセンテンスを書き換えること。

- ドキュメント文字列の開始と終了に空白文字を使用しないこと。
- ソースコード内の後続行のテキスト、最初の行と揃うようにドキュメント文字列の後続行をインデントしてはならない。これはソースコードでは見栄えがよいがユーザーがドキュメントを閲覧する際は奇妙な見栄えになる。開始のダブルクォーテーションの前のインデントは文字列の一部には含まれないことを忘れないこと!
- ドキュメント文字列が Lisp シンボルを参照する際には、たとえば (‘ . . ’) のように、それがプリントされるとき (通常は小文字を意味する) のように、前後を curved single quote で括って記述すること。例外が 2 つある。`t`と `nil`は前後の区切り記号を記述しない。たとえば:

  ```
  CODE can be ‘lambda’, nil, or t.
  ```

  curved single quote を入力する方法については Section “Quotation Marks” in *The GNU Emacs Manual* を参照のこと。

  ドキュメント文字列は‘`like-this`’ではなく‘`like-this`’のように、grave accent ‘ と apostrophe ’でシンボルをクォートする古いシングルクォートの慣習も使用できる。この古い慣習は grave accent と apostrophe が鏡像となる今では時代遅れのディスプレイ用にデザインされている。この慣習を使用するドキュメントは help バッファーへコピー時にはユーザーが望む形式に変換される。Section 24.3 [Keys in Documentation], page 522 を参照のこと。

  Help モードはシングルクォートされたシンボル名がドキュメント文字列で使用されている際には、それが関数と変数のいずれかの定義をもっていれば自動的にハイパーリンクを作成する。これらの機能を使用するために何か特別なことを行う必要はない。しかしあるシンボルが関数と変数の両方の定義をもち一方だけを参照したい場合には、そのシンボル名の直前に ‘`variable`’、‘`option`’、‘`function`’、‘`command`’の単語のいずれかを記述してそれを指定できる (これらの指示語の識別では大文字小文字に差はない)。たとえば以下を記述すると

  ```
  This function sets the variable `buffer-file-name'.
  ```

  これのハイパーリンクは `buffer-file-name`の変数のドキュメントだけを参照して関数のドキュメントは参照しない。

  あるシンボルが関数および/または変数の定義をもつがドキュメントしているシンボルの使用とそれらが無関係なら、すべてのハイパーリンク作成を防ぐためにシンボル名の前に単語 ‘`symbol`’ か ‘`program`’を記述できる。たとえば、

  ```
  If the argument KIND-OF-RESULT is the symbol `list',
  ```

```
this function returns a list of all the objects
that satisfy the criterion.
```

これは無関係な関数 `list`のドキュメントにハイパーリンクを作成しない。

変数ドキュメントがない変数には、通常はハイパーリンクは作成されない。そのような変数の前に単語 '`variable`'と '`option`'のいずれかを記述すればハイパーリンクの作成を強制できる。

フェイスにたいするハイパーリンクはフェイスの前か後に単語 '`face`'があれば作成される。この場合にはたとえそのシンボルが変数や関数として定義されていてもフェイスのドキュメントだけが表示される。

Info ドキュメントにハイパーリンクを作成するには、'`info node`'、'`Info node`'、'`info anchor`'、'`Info anchor`'のいずれかの後 Info のノード (かアンカー) をシングルクォートして記述する。Info ファイル名のデフォルトは '`emacs`'。たとえば、

```
See Info node `Font Lock' and Info node `(elisp)Font Lock Basics'.
```

最後に URL のハイパーリンクを作成するには '`URL`'の後に URL をシングルクォートして記述する。たとえば、

```
The home page for the GNU project has more information (see URL
`https://www.gnu.org/').
```

- ドキュメント文字列内に直接キーシーケンスを記述しないこと。かわりに、それらを表すために '`\\[...]`'構文を使用すること。たとえば '`C-f`'と記述するかわりに '`\\[forward-char]`'と記述する。Emacs がドキュメント文字列を表示する際には何であれカレントで `forward-char`にバインドされたキーに置き換える (これは通常は '`C-f`'だがユーザーがキーバインディングを変更していれば何か他の文字かもしれない)。Section 24.3 [Keys in Documentation], page 522 を参照のこと。
- メジャーモードのドキュメント文字列ではグローバルマップではなく、そのモードのローカルマップを参照したいだろう。したがってどのキーマップを使用するか指定するために、ドキュメント文字列内で一度 '`\\<...>`'構文を使用する。最初に '`\\[...]`'を使用する前にこれを行うこと。'`\\<...>`'の内部のテキストはメジャーモードにたいするローカルキーマップを含む変数名であること。

  ドキュメント文字列の表示が低速になるので非常に多数回の '`\\[...]`'の使用は実用的ではない。メジャーモードのもっとも重要なコマンドの記述にこれを使用して、そのモードの残りのキーマップの表示には '`\\{...}`'を使用する。
- 一貫性を保つために関数のドキュメント文字列の最初のセンテンス内の動詞は、命令形で表すこと。たとえば "Return the cons of A and B."、好みによっては "Returns the cons of A and B." を使用する。通常は最初のパラグラフの残りの部分にたいして同様に行っても見栄えがよい。各センテンスが叙実的で適切な主題をもつなら後続のパラグラフの見栄えはよくなる。
- yes-or-no 述語であるような関数のドキュメント文字列は、何が真を構成するか明示的に示すために、"Return t if" のような単語で始まること。単語 "return" は小文字の "t" で開始される幾分紛らわしい可能性のあるセンテンスを避ける。
- ドキュメント文字列は受動態ではなく能動態、未来形ではなく現在形で記述すること。たとえば "A list containing A and B will be returned." ではなく、"Return a list containing A and B." と記述すること。
- 不必要な "cause"(や同等の単語) の使用を避けること。"Cause Emacs to display text in boldface" ではなく、単に "Display text in boldface" と記述すること。
- 多くの人にとってなじみがなく typo と間違えるであろうから、"iff"("if and only if" を意味する数学用語) の使用を避けること。ほとんどの場合には、その意味は単なる "if" で明快である。それ以外ではその意味を伝える代替えフレーズを探すよう試みること。

- “for example(たとえば)” に “e.g.”、“that is(つまり)” に “i.e.”、“number(数値)” に “no.”、“in contrast to(対照的に)” に “c.f.”、“with respect to(に関しては)” にたいする “w.r.t.” のような略語の使用は可能なかぎり避けるよう努力すること。ほとんど常に展開されたバージョンのほうが読み易い [2]。
- 特定のモードや状況でのみコマンドに意味がある際にはドキュメント文字列内でそれに言及すること。たとえば `dired-find-file`のドキュメントは:

  ```
  In Dired, visit the file or directory named on this line.
  ```
- ユーザーがセットしたいと望むかもしれないオプションを表す変数を定義する際には `defcustom` を使用すること。Section 12.5 [Defining Variables], page 178 を参照のこと。
- yes-or-no フラグであるような変数のドキュメント文字列は、すべての非 `nil`値が等価であることを明確にして、`nil`と非 `nil`が何を意味するかを明示的に示すために “Non-nil means” のような単語で始めること。
- ドキュメント文字列内の開カッコで始まる行は、以下のように開カッコの前へのバックスラッシュの記述を考慮すること:

  ```
  The argument FOO can be either a number
  \(a buffer position) or a string (a file name).
  ```

  これは ‘(’を defun の開始として扱う 27.1 より古いバージョンの Emacs でのバグを回避する (Section “Defuns” in *The GNU Emacs Manual* を参照)。コードを古いバージョンの Emacs で編集する誰かをあなたが予期せぬのなら、この回避策は必要ない。

D.7 コメント記述のヒント

コメントにたいして以下の慣習を推奨します:

‘;’ 1 つのセミコロン ‘;’で始まるコメントはソースコードの右側の同じ列にすべて揃えられる。そのようなコメントは通常はその行のコードがどのように処理を行うかを説明する。たとえば:

```
(setq base-version-list                 ; There was a base
      (assoc (substring fn 0 start-vn)  ; version to which
             file-version-assoc-list))  ; this looks like
                                        ; a subversion.
```

‘;;’ 2 つのセミコロン ‘;;’で始まるコメントはコードと同じインデントレベルで揃えられる。そのようなコメントは通常はその後の行の目的や、その箇所でのプログラムの状態を説明する。たとえば:

```
(prog1 (setq auto-fill-function
             ...
             ...
  ;; Update mode line.
  (force-mode-line-update)))
```

わたしたちは通常は関数の外側のコメントにも 2 つのセミコロンを使用する。

```
;; This Lisp code is run in Emacs when it is to operate as
;; a server for other processes.
```

関数がドキュメント文字列をもたなければ、かわりにその関数の直前にその関数が何を行うかと、正しく呼び出す方法を説明する 2 つのセミコロンのコメントをもつこと。各引数の意味と引数で可能な値をその関数が解釈する方法を正確に説明すること。しかしそのようなコメントはドキュメント文字列に変換するほうがはるかに優れている。

[2] わたしたちは時折これを使用しますが、やりすぎないでください。

‘;;;’ 3つのセミコロン‘;;;’デ始まるコメントは左マージンから始まる。わたしたちはOutlineマイナーモードのheading(ヘッダー)とみなされるべきコメントにそれらを使用している。デフォルトでは少なくとも(後に1つの空白文字と非空白文字が続く)3つのセミコロンはheadingとみなして、2つ以下のセミコロンで始まるものはheadingとみなさない。歴史的に3つのセミコロンのコメントは関数内での行のコメントアウトに使用されてきたがこの使用は推奨しない。

関数全体をコメントアウトするときは2つのセミコロンを使用する。

‘;;;;’ 4つ(またはそれ以上)のセミコロン‘;;;;’で始まるコメントは左マージンに揃えられプログラムのメジャーセクションのheadingに使用される。たとえば:

```
;;;; The kill ring
```

これらのheadingの下にsub-headingsをもちたければ、sub-headingsをネストするためにより多くのセミコロンを使用すること。

一般的に言うとコマンド *M-;* (`comment-dwim`) は適切なタイプのコメントを自動的に開始するか、セミコロンの数に応じて既存のコメントを正しい位置にインデントします。Section “Manipulating Comments” in *The GNU Emacs Manual* を参照してください。

D.8 Emacs ライブラリーのヘッダーの慣習

Emacsにはセクションに分割してそれの記述者のような情報を与えるために、Lispライブラリー内で特別なコメントを使用する慣習があります。それらのアイテムにたいして標準的なフォーマットを使用すれば、ツール(や人)が関連する情報を抽出するのが簡単になります。このセクションでは以下の例を出発点にこれらの慣習を説明します。

```
;;; foo.el --- Support for the Foo programming language

;; Copyright (C) 2010-2020 Your Name

;; Author: Your Name <yourname@example.com>
;; Maintainer: Someone Else <someone@example.com>
;; Created: 14 Jul 2010
;; Keywords: languages
;; URL: https://example.com/foo

;; This file is not part of GNU Emacs.

;; This file is free software...
...
;; along with this file.  If not, see <https://www.gnu.org/licenses/>.
```

一番最初の行は以下のフォーマットをもつべきです:

```
;;; filename --- description
```

この説明は1行に収まる必要があります。そのファイルに‘-*-’指定が必要なら *description*の後に配置してください。これにより最初の行が長くなりすぎるようなら、そのファイル終端でLocal Variablesセクションを使用してください。

著作権表示には、(あなたがそのファイルを記述したなら)通常はあなたの名前をリストします。あなたの作業の著作権を主張する雇用者がいる場合には、かわりに彼らをリストする必要があるかもしれません。Emacsディストリビューションにあなたのファイルが受け入れられていなければ、著作権者をFree Software Foundation(またはそのファイルがGNU Emacsの一部)だと告知しないでください。著作権とライセンス通知の形式に関するより詳細な情報はthe guide on the GNU website (`https://www.gnu.org/licenses/gpl-howto.html`) を参照してください。

著作権表示の後は、それぞれが '`;; header-name:`'で始まる複数のヘッダーコメント *(header comment)* を記述します。以下は慣習的に利用できる *header-name*のテーブルです:

'`Author`' この行は少なくともそのライブラリーの主要な作者の名前と email アドレスを示す。複数の作者がいる場合には前に`;;`とタブか少なくとも 2 つのスペースがある継続行で彼らをリストする。わたしたちは '`<...>`'という形式で連絡用 email アドレスを含めることを推奨する。たとえば:

```
;; Author: Your Name <yourname@example.com>
;;       Someone Else <someone@example.com>
;;       Another Person <another@example.com>
```

'`Maintainer`'
このヘッダーは Author ヘッダーと同じフォーマット。これは現在そのファイルを保守 (バグレポートへの応答等) をする人 (か人々) をリストする。

Maintainer の行がなければ Author フィールドの人 (人々) が Maintainer とみなされる。Emacs 内のいくつかのファイルは、そのファイルのオリジナル作者がもはや責任をもっておらず Emacs の一部として保守されていることを意味するために、Maintainer に '`emacs-devel@gnu.org`'を使用している。

'`Created`' このオプションの行はファイルのオリジナルの作成日付を与えるもので歴史的な興味のためだけに存在する。

'`Version`' 個々の Lisp プログラムにたいしてバージョン番号を記録したいならこの行に配置する。Emacs とともに配布された Lisp ファイルは Emacs のバージョン番号自体が同じ役割を果たすので一般的には '`Version`'ヘッダーをもたない。複数ファイルのコレクションを配布する場合には、各ファイルではなく主となるファイルにバージョンを記述することを推奨する。

'`Keywords`'
この行はヘルプコマンド `finder-by-keyword`でリストするキーワード。意味のあるキーワードのリストの確認にこのコマンドを使用してほしい。コマンド *`M-x checkdoc-package-keywords RET`*は`finder-known-keywords`にないすべてのキーワードを探して表示する。変数 `checkdoc-package-keywords-flag`を非 `nil`にセットすると、checkdoc コマンドはチェックにキーワード検証を含める。

このフィールドはトピックでパッケージを探す人が、あなたのパッケージを見つける手段となる。キーワードを分割するにはスペースとカンマの両方を使用できる。

人はしばしばこのフィールドを単に Finder(訳注: `finder-by-keyword`がオープンするバッファー) に関連したキーワードではなくパッケージを説明する任意のキーワードを記述する箇所だとみなすのは不運なことだ。

'`Homepage`'
'`URL`' この行はライブラリーのホームページを示す。

'`Package-Version`'
'`Version`'がパッケージマネージャーによる使用に適切でなければ、パッケージは '`Package-Version`'を定義でき、かわりにこれが使用される。これは '`Version`' が RCS や `version-to-list`でパース不能な何かであるようなら手軽である。Section 41.1 [Packaging Basics], page 1134 を参照のこと。

'`Package-Requires`'
これが存在する場合にはカレントパッケージが正しく動作するために依存するパッケージを示す。Section 41.1 [Packaging Basics], page 1134 を参照のこと。これは (パッ

ケージの完全なセットがダウンロードされることを確実にするために) ダウンロード時と、(すべての依存パッケージがあるときだけパッケージがアクティブになることを確実にするために) アクティブ化の両方でパッケージマネージャーにより使用される。

このフォーマットは単一行からなるリストのリスト。サブリストの `car`はそれぞれパッケージの名前 (シンボル)、`cadr`は `version-to-list`でパース可能な許容し得る最小のバージョン番号 (文字列)。バージョンが欠落したエントリー (単なるシンボルやある要素のサブリストであるようなエントリー) はバージョンが`"0"`のエントリーと等価。たとえば:

```
;; Package-Requires: ((gnus "1.0") (bubbles "2.7.2") cl-lib (seq))
```

パッケージのコードは自動的に、実行中の Emacs のカレントのバージョン番号をもつ‘`emacs`’という名前のパッケージを定義する。これはパッケージが要求する Emacs の最小のバージョンに使用できる。

ほぼすべての Lisp ライブラリーは‘`Author`’と‘`Keywords`’のヘッダーコメント行をもつべきです。適切なら他のものを使用してください。ヘッダー行内で別のヘッダー行の名前も使用できます。これらは標準的な意味をもたないので害になることを行うことはできません。

わたしたちはライブラリーファイルのコンテンツを分割するために追加の提携コメントを使用します。これらは空行で他のものと分離されている必要があります。以下はそれらのテーブルです:

‘`;;; Commentary:`’
: これはライブラリーが機能する方法を説明する、概論コメントを開始する。これは複製許諾の直後にあり‘`Change Log`’、‘`History`’、‘`Code`’のコメント行で終端されていること。このテキストは Finder パッケージで使用されるのでそのコンテキスト内で有意であること。

‘`;;; Change Log:`’
: これは時間とともにそのファイルに加えられたオプションの変更ログを開始する。このセクションに過剰な情報を配置してはならない。(Emacs が行うように) バージョンコントロールシステムの詳細ログや個別の `ChangeLog`ファイルに留めるほうがよい。‘`History`’は‘`Change Log`’の代替え。

‘`;;; Code:`’
: これはプログラムの実際のコードを開始する。

‘`;;; `*`filename`* ends here`’
: これはフッター行 *(footer line)*。これはそのファイルの終端にある。これの目的はフッター行の欠落から、人がファイルの切り詰められたバージョンを検知することを可能にする。

Appendix E GNU Emacs の内部

このチャプターでは実行可能な Emacs 実行可能形式を事前ロードされた Lisp ライブラリーとともにダンプする方法、ストレージが割り当てられる方法、および C プログラマーが興味をもつかもしれない GNU Emacs の内部的な側面のいくつかを説明します。

E.1 Emacs のビルド

このセクションでは Emacs 実行可能形式のビルドに関するステップの説明をします。makefile がこれらすべてを自動的に行うので、Emacs をビルドやインストールをするためにこの題材を知る必要はありません。この情報は Emacs 開発者に適しています。

Emacs のビルドには GNU Make のバージョン 3.81 以降が必要です。

`src`ディレクトリー内の C ソースファイルをコンパイルすることにより、`temacs`と呼ばれる実行可能形式ファイルが生成されます。これは *bare impure Emacs(*裸で不純な *Emacs)* とも呼ばれます。これには Emacs Lisp インタープリターと I/O ルーチンが含まれますが編集コマンドは含まれません。

コマンド `temacs -l loadup` は `temacs`を実行して `loadup.el`をロードするように計らいます。`loadup`ライブラリーは通常の Emacs 編集環境をセットアップする追加の Lisp ライブラリーをロードします。このステップの後には Emacs 実行可能形式は *bare(*裸*)* ではなくなります。

標準的な Lisp ファイルのロードには若干の時間を要するので、ユーザーが直接 `temacs`実行可能形式を実行することは通常はありません。そのかわり、Emacs ビルドの最終ステップの 1 つとしてコマンド '`temacs -batch -l loadup --temacs=`*`dump-method`*' が実行されます。特別なオプション`--temacs`は `temacs`にたいして、この後に続けて Emacs を実行した際に開始がより高速になるように、標準のすべての事前ロードされる Lisp の関数と変数の記録方法を指示します。オプション`--temacs`には引数 *dump-method*が必要であり、以下のいずれかを指定できます:

'`pdump`' ダンプファイル *(dump file)* に事前ロード Lisp データを記録する。このメソッドは Emacs の開始時にロードされることになる追加データファイルを生成する。生成されたダンプファイルは通常は `emacs.pdmp`と呼ばれて、Emacs の `exec-directory`にインストールされる (Section 24.6 [Help Functions], page 526 を参照)。これはモダンなシステムにおいてセキュリティとプライバシーを強化するために使用されるさまざまなメモリーレイアウト技術を妨害する可能性のある、メモリー割り当ての特別なテクニックの使用を Emacs に要求しないので、もっとも好ましいメソッドの 1 つ。

'`pbootstrap`'
'`pdump`'と同様だが、以前の Emacs バイナリおよびバイトコンパイル済み Lisp ファイル `*.elc`が存在しないときに、Emacs のブートストラップ *(bootstrapping)* の間に使用される。この場合には、生成されるダンプファイルは通常は `bootstrap-emacs.pdmp`。

'`dump`' このメソッドはすべての標準 Lisp ファイルがすでに事前ロード済みの `emacs`という実行可能プログラムを `temacs`にダンプさせる ('`-batch`'引数は `temacs`にその端末上のすべてのデータ初期化の試みを抑制するので、ダンプされた Emacs の端末情報テーブルは空になる)。このメソッドは実行中プロセスからプログラムファイルを生成するため、プログラムを実行 (exec) してプロセスを開始することとある意味で反対なことから *unexec* として知られている。このメソッドは伝統的に Emacs の状態を保存する手段だったが、今や非推奨となった。

'`bootstrap`'
'`dump`'と同様だが、`unexec`メソッドで Emacs をブートストラップする際に使用する。

インストールされる Emacs は、ダンプされた `emacs`実行可能形式です (*pure* な Emacs とも呼ばれる)。Emacs のビルドにポータブルダンパーを使用した場合には、`emacs`実行可能形式は実際には `temacs`の正確なコピーであり、対応機種 `emacs.pdmp`ファイルも同様にインストールされます。変数 `preloaded-file-list`にはダンプファイルやダンプされた Emacs 実行可能形式に記録された事前ロード済み Lisp ファイルのリストが格納されます。新たなオペレーティングシステムに Emacs をポートする際に、その OS が何の種類のダンプも実装していなければ Emacs は起動時に毎回 `loadup.el` をロードしなければなりません。

ダンプされた `emacs`にはデフォルトではビルド時刻やホスト名のような詳細が記録されます。これらの詳細を抑制するために `configure`のオプションに`--disable-build-details`を使用すれば、同一のソースから Emacs を 2 回ビルドしてインストールする際に同一の Emacs のコピーが生成される可能性が高くなります。

`site-load.el`という名前のライブラリーを記述することにより、事前ロードするファイルを追加指定できます。追加するファイルを保持するために純粋 (pure) なスペース *n*バイトを追加するように、以下の定義

```
#define SITELOAD_PURESIZE_EXTRA n
```

で Emacs をリビルドする必要があるでしょう。`src/puresize.h`を参考にしてください (十分大きくなるまで 20000 ずつ増加させる)。しかし追加ファイルの事前ロードの優位はマシンの高速化により減少します。現代的なマシンでは通常はお勧めしません。

`loadup.el`が `site-load.el`を読み込んだ後に `Snarf-documentation`を呼び出すことにより、それらが格納された場所のファイル `etc/DOC`内にあるプリミティブと事前ロードされる関数 (と変数) のドキュメント文字列を探します ([Accessing Documentation], page 522 を参照)。

`site-init.el`という名前のライブラリー名に配置することにより、ダンプ直前に実行する他の Lisp 式を指定できます。このファイルはドキュメント文字列を見つけた後に実行されます。

関数や変数の定義を事前ロードしたい場合には、それを行うために 3 つの方法があります。それらにより定義ロードしてその後の Emacs 実行時にドキュメント文字列をアクセス可能にします:

- `etc/DOC`の生成時にそれらのファイルをスキャンするよう計らい `site-load.el`でロードする。
- ファイルを `site-init.el`でロードして Emacs インストール時に Lisp ファイルのインストール先ディレクトリーにファイルをコピーする。
- それらの各ファイルでローカル変数として `byte-compile-dynamic-docstrings`に `nil`値を指定して `site-load.el`か `site-init.el`でロードする (この手法には Emacs が毎回そのドキュメント文字列用のスペースを確保するという欠点がある)。

通常の未変更の Emacs でユーザーが期待する何らかの機能を変更するような何かを `site-load.el`や `site-init.el`内に配置することはお勧めしません。あなたのサイトで通常の機能をオーバーライドしなければならないと感じた場合には、`default.el`でそれを行えばユーザーが望む場合にあなたの変更をオーバーライドできます。Section 40.1.1 [Startup Summary], page 1090 を参照してください。`site-load.el`か `site-init.el`のいずれかが `load-path`を変更する場合には変更はダンプ後に失われます。Section 16.3 [Library Search], page 272 を参照してください。`load-path`を永続的に変更するには `configure`の`--enable-locallisppath`オプションを指定してください。

事前ロード可能なパッケージでは、その後の Emacs スタートアップまで特定の評価の遅延が必要 (または便利) なことがあります。そのようなケースの大半はカスタマイズ可能な変数の値に関するものです。たとえば `tutorial-directory`は事前ロードされる `startup.el`内で定義される変数です。これのデフォルト値は `data-directory`にもとづいてセットされます。この変数は Emacs ダンプ時ではなくスタート時に `data-directory`の値を必要とします。なぜなら Emacs 実行可能形式はダンプされたものなので、恐らく異なる場所にインストールされるからです。

`custom-initialize-delay` *symbol value* [Function]
この関数は次回の Emacs 開始まで *symbol*の初期化を遅延する。通常はカスタマイズ可能変数の`:initialize`プロパティとしてこの関数を指定することにより使用する (引数 *value*はフォーム Custom 由来の互換性のためだけに提供されており使用しない)。

`custom-initialize-delay`が提供するより一般的な機能を要する稀なケースでは`before-init-hook`を使用できます (Section 40.1.1 [Startup Summary], page 1090 を参照)。

`dump-emacs-portable` *to-file* **&optional** *track-referrers* [Function]
この関数は`pdump`メソッドを使用して、Emacs のカレント状態をダンプファイル *to-file*にダンプする。ダンプファイルは通常は *`emacs-name`*`.dmp`と呼ばれる。ここで *emacs-name*は Emacs の実行可能形式ファイル名。オプション引数 *track-referrers*が非`nil`なら、ポータブルダンパーは`pdump`メソッドでは未サポートのオブジェクトタイプの出所追跡の助けとなる追加情報を維持する。

多くのプラットフォームでポータブルダンパーのコードが実行可能だとしても、それが生成するダンプファイルに可搬性はない (ダンプした Emacs 実行可能形式だけがロードできる)。

すでにダンプ済みの Emacs 内でこの関数を使用する場合には '`-batch`'オプションで Emacs を実行しなければならない。

`dump-emacs` *to-file from-file* [Function]
この関数は`unexec`メソッドを使用して、Emacs のカレント状態を実行可能ファイル *to-file*にダンプする。これは *from-file* (通常はファイル`temacs`) からシンボルを取得する。

この関数をすでにダンプ済みの Emacs で使用することはできない。この関数は非推奨であり、Emacs はデフォルトでは`unexec`サポートなしでビルドされているので利用できない。

`pdumper-stats` [Function]
カレントの Emacs セッションの状態がダンプファイルからリストアされると、この関数はダンプファイルに関する情報と Emacs 状態のリストアに要した時間をリターンする。値は `((dumped-with-pdumper . t) (load-time . `*`time`*`) (dump-file-name . `*`file`*`))` のような alist。ここで *file*はダンプファイル名、*time*はダンプファイルから状態をリストアするのに要した秒数。カレントセッションがダンプファイルからリストアされたものでなければ値は nil。

E.2 純粋ストレージ

Emacs Lisp はユーザー作成 Lisp オブジェクトにたいして、通常ストレージ *(normal storage)* と純粋ストレージ *(pure storage)* という 2 種のストレージをもちます。通常ストレージは Emacs セッションが維持される間に新たにデータが作成される場所です。純粋ストレージは事前ロードされた標準 Lisp ファイル内の特定のデータのために使用されます。このデータは実際の Emacs 使用中に決して変更されるべきではないデータです。

純粋ストレージは`temacs`が標準的な事前ロー Lisp ライブラリーのロード中にのみ割り当てられます。ファイル`emacs`ではこのメモリースペースは読み取り専用とマークされるのでマシン上で実行中のすべての Emacs ジョブで共有できます。純粋ストレージは拡張できません。Emacs のコンパイル時に固定された量が割り当てられて、それが事前ロードされるライブラリーにたいして不足なら`temacs`はそれに収まらない部分を動的メモリーに割り当てます。Emacs を`pdump`メソッド (Section E.1 [Building Emacs], page 1175 を参照) を使用してダンプする場合には純粋ストレージのオーバーフローは特に重要ではありません (単に事前ロード済みのライブラリーのいくつかが別の Emacs ジョブで共有できないことを意味する)。しかし Emacs を時代遅れとなった`unexec`メソッドでダンプす

る場合には結果イメージは動作するでしょうが、この状況ではメモリーリークとなるのでガーベージコレクション (Section E.3 [Garbage Collection], page 1178 を参照) は無効です。そのような通常なら発生しないオーバーフローは、あなたが事前ロードライブラリの追加や標準的な事前ロードライブラリに追加を試みないかぎり発生しません。Emacs が `unexec`を使用してダンプされていたら、Emacs は開始時にオーバーフローに関する警告を表示するでしょう。これが発生したらファイル `src/puresize.h`内のコンパイルパラメーターを `SYSTEM_PURESIZE_EXTRA`を増やして Emacs をリビルドする必要があります。

`purecopy` *object* [Function]

この関数は純粋ストレージに *object*のコピーを作成してリターンする。これは同じ文字で新たに文字列を作成することにより文字列をコピーするが、純粋ストレージではテキストプロパティはない。これはベクターとコンスセルのコンテンツを再帰的にコピーする。シンボルのような他のオブジェクトのコピーは作成しないが未変更でリターンする。マーカーのコピーを試みるとエラーをシグナルする。

この関数は Emacs のビルド中とダンプ中を除き何もしない。通常は事前ロードされる Lisp ファイル内でのみ呼び出される。

`pure-bytes-used` [Variable]

この変数の値は、これまでに割り当てられた純粋ストレージのバイト数。ダンプされた Emacs では通常は利用可能な純粋ストレージの総量とほとんど同じであり、もしそうでないならわたしたちは事前割り当てをもっと少なくするだろう。

`purify-flag` [Variable]

この変数は `defun`が純粋ストレージにその関数定義のコピーを作成するべきか否かを判断する。これが非 `nil`ならその関数の定義は純粋ストレージにコピーされる。

このフラグは Emacs のビルド用の基本的な関数の初回ロード中は `t`となる。実行可能形式として Emacs をダンプすることにより、ダンプ前後の実際の値とは無関係に常にこの変数に `nil`が書き込まれる。

実行中の Emacs でこのフラグを変更しないこと。

E.3 ガーベージコレクション

プログラムがリストを作成するときや、(ライブライのロード等により) ユーザーが新しい関数を定義する際には、そのデータは通常ストレージに配置されます。通常ストレージが少なくなると Emacs はもっとメモリーを割り当てるようにオペレーティングシステムに要求します。シンボル、コンスセル、小さいベクター、マーカー等のような別のタイプの Lisp オブジェクトはメモリー内の個別のブロックに隔離されます (大きいベクター、長い文字列、バッファー、および他の特定の編集タイプは非常に巨大であり 1 つのオブジェクトにたいして個別のブロックが割り当てられて、小さな文字列は 8k バイトのブロック、小さいベクターは 4k バイトのブロックにパックされる)。

基本的なベクター以上のマーカー、オーバーレイ、バッファーのような多くのオブジェクトが、あたかもベクターであるかのように管理されています。対応する C データ構造体には `union vectorlike_header`フィールドが含まれ、そのメンバー `size`には `enum pvec_type`で列挙されたサブタイプ、その構造体が含む `Lisp_Object`フィールドの数に関する情報、および残りのデータのサイズが含まれます。この情報は、オブジェクトのメモリーフットプリントの計算に必要であり、ベクターブロックの繰り返し処理の際のベクター割り当てコードにより使用されます。

しばらくの間いくつかのストレージを使用して、(たとえば) バッファーの kill やあるオブジェクトを指す最後のポインターの削除によりそれを解放するのは非常に一般的です。この放棄されたスト

レージを再利用するために Emacs はガーベージコレクター *(garbage collector)* を提供します。ガーベージコレクターは、いまだ Lisp プログラムからアクセス可能なすべての Lisp オブジェクトを検索、マークすることにより動作します。これを開始するにはすべてのシンボル、それらの値と関連付けられている関数定義、現在スタック上にあるすべてのデータをアクセス可能であると仮定します。別のアクセス可能オブジェクトを介して間接的に到達できるすべてのオブジェクトもアクセス可能とみなされます。

マーキングが終了してもマークされないオブジェクトはすべてガーベージ (garbage: ごみ) です。Lisp プログラムかユーザーの行為かに関わらず、それらに到達する手段はもはや存在しないので参照することは不可能です。誰もそれを失うことはないので、それらのスペースは再利用されることになります。ガーベージコレクターの 2 つ目ののフェーズ (sweep: スイープ、一掃) ではそれらの再利用を計らいます。

スイープフェーズは将来の割り当て用に、シンボルやマーカーと同様に未使用のコンスセルをフリーリスト *(free list)* 上に配置します。これはアクセス可能な文字列は少数の 8k ブロックを占有するように圧縮して、その後に他の 8k ブロックを解放します。ベクターブロックから到達不可能はベクターは可能なかぎり最大のフリーエリアを作成するために統合して、フリーエリアが完全な 4k ブロックに跨がるようならブロックは解放されます。それ以外ならフリーエリアはフリーリスト配列に記録されます。これは各エントリーが同サイズのエリアのフリーリストに対応します。巨大なベクター、バッファー、その他の巨大なオブジェクトは個別に割り当てと解放が行われます。

> **Common Lisp** に関する注意**:** 他の Lisp と異なり GNU Emacs Lisp はフリーリストが空のときにガーベージコレクターを呼び出さない。かわりに単にオペレーティングシステムに更なるストレージの割り当てを要求して、`gc-cons-threshold`バイトを使い切るまで処理を継続する。
>
> これは特定の Lisp プログラムの範囲の実行直前に明示的にガーベージコレクターを呼び出せば、その範囲の実行中はガーベージコレクターが実行されないだろうと確信できることを意味する (そのプログラム範囲が 2 回目のガーベージコレクションを強制するほど多くのスペースを使用しないという前提)。

`garbage-collect` [Command]

このコマンドはガーベージコレクションを実行して使用中のスペース量の情報をリターンする (前回のガーベージコレクション以降に `gc-cons-threshold`バイトより多い Lisp データを使用した場合には自然にガーベージコレクションが発生することもあり得る)。

`garbage-collect`は使用中のスペース量の情報をリストでリターンする。これの各エントリーは‘`(`*`name size used`*`)`’という形式をもつ。このエントリーで *name*はそのエントリーが対応するオブジェクトの種類を記述するシンボル、*size*はそれが使用するバイト数、*used*はヒープ内で生きていることが解ったオブジェクトの数、オプションの *free*は生きていないが Emacs が将来の割り当て用に保持しているオブジェクトの数。全体的な結果は以下のようになる:

```
((conses cons-size used-conses free-conses)
 (symbols symbol-size used-symbols free-symbols)
 (strings string-size used-strings free-strings)
 (string-bytes byte-size used-bytes)
 (vectors vector-size used-vectors)
 (vector-slots slot-size used-slots free-slots)
 (floats float-size used-floats free-floats)
 (intervals interval-size used-intervals free-intervals)
 (buffers buffer-size used-buffers)
 (heap unit-size total-size free-size))
```

以下は例:

```
(garbage-collect)
      ⇒ ((conses 16 49126 8058) (symbols 48 14607 0)
                (strings 32 2942 2607)
                (string-bytes 1 78607) (vectors 16 7247)
                (vector-slots 8 341609 29474) (floats 8 71 102)
                (intervals 56 27 26) (buffers 944 8)
                (heap 1024 11715 2678))
```

以下は各要素を説明するためのテーブル。最後の `heap`エントリーはオプションであり、背景にある `malloc`実装が `mallinfo`関数を提供する場合のみ与えられることに注意。

cons-size コンスセルの内部的サイズ (`sizeof (struct Lisp_Cons)`)。

used-conses
使用中のコンスセルの数。

free-conses
オペレーティングシステムから取得したスペースにあるがカレントで未使用のコンスセルの数。

symbol-size
シンボルの内部的サイズ (`sizeof (struct Lisp_Symbol)`)。

used-symbols
使用中のシンボルの数。

free-symbols
オペレーティングシステムから取得したスペースにあるがカレントで未使用のシンボルの数。

string-size 文字列ヘッダーの内部的サイズ (`sizeof (struct Lisp_String)`)。

used-strings
使用中の文字列ヘッダーの数。

free-strings
オペレーティングシステムから取得したスペースにあるがカレントで未使用の文字列ヘッダーの数。

byte-size これは利便性のために使用されるもので `sizeof (char)`と同じ。

used-bytes すべての文字列データの総バイト数。

vector-size 長さ 1 のベクターのヘッダーを含めたバイトサイズ。

used-vectors
ベクターブロックから割り当てられたベクターブロック数。

slot-size ベクタースロットの内部的なサイズで常に `sizeof (Lisp_Object)`と等しい。

used-slots 全使用済みベクターのスロット数。スロット数にはプラットフォームに応じてベクターのヘッダーに由来する一部、またはすべてのオーバーヘッドが含まれるかもしれない。

free-slots すべてのベクターブロックのフリースロットの数。

float-size 浮動小数点数オブジェクトの内部的なサイズ (`sizeof (struct Lisp_Float)`)。(ネイティブプラットフォームの `float`や `double`と混同しないこと。)

used-floats
使用中の浮動小数点数の数。

free-floats オペレーティングシステムから取得したスペースにあるがカレントで未使用の浮動小数点数の数。

interval-size
インターバルオブジェクト (interval object) の内部的なサイズ (`sizeof (struct interval)`)。

used-intervals
使用中のインターバルの数。

free-intervals
オペレーティングシステムから取得したスペースにあるがカレントで未使用のインターバルの数。

buffer-size バッファーの内部的なサイズ (`sizeof (struct buffer)`)。(`buffer-size`関数がリターンする値と混同しないこと。)

used-buffers
使用中のバッファーオブジェクトの数。これにはユーザーからは不可視の kill されたバッファー、つまりリスト `all_buffers`内のバッファーすべてが含まれる。

unit-size ヒープスペースを計る単位であり常に 1024 バイトと等しい。

total-size *unit-size*単位での総ヒープサイズ。

free-size *unit-size*単位でのカレントで未使用のヒープスペース。

純粋スペース (Section E.2 [Pure Storage], page 1177 を参照) 内にオーバーフローがあり、かつ Emacs が (時代遅れとなった)`unexec`メソッド (Section E.1 [Building Emacs], page 1175 を参照) を使用してダンプされていたら、この場合は実際にガーベージコレクションを行うことは不可能なので `garbage-collect`は `nil`をリターンする。

`garbage-collection-messages` [User Option]
この変数が非 `nil`なら Emacs はガーベージコレクションの最初と最後にメッセージを表示する。デフォルト値は `nil`。

`post-gc-hook` [Variable]
これはガーベージコレクションの終わりに実行されるノーマルフック。ガーベージコレクションはこのフックの関数の実行中は抑制されるので慎重に記述すること。

`gc-cons-threshold` [User Option]
この変数の値は別のガーベージコレクションをトリガーするために、ガーベージコレクション後に Lisp オブジェクト用に割り当てなければならないストレージのバイト数。特定のオブジェクトタイプに関する情報を取得するために、`garbage-collect`がリターンした結果を使用できる。バッファーのコンテンツに割り当てられたスペースは勘定に入らない。

threshold の初期値は `GC_DEFAULT_THRESHOLD`であり、これは `alloc.c`内で定義されている。これは `word_size`単位で定義されているので、デフォルトの 32 ビット設定では 400,000、64 ビット設定では 800,000 になる。大きい値を指定するとガーベージコレクションの頻度が下

る。これはガーベージコレクションにより費やされる時間を減少させるがメモリーの総使用量は増大する。大量の Lisp データを作成するプログラムの実行時にはこれを行いたいと思うかもしれない。

`GC_DEFAULT_THRESHOLD`の 1/10 まで下げた小さな値を指定することにより、より頻繁にガーベージコレクションを発生させることができる。この最小値より小さい値は後続のガーベージコレクションで、`garbage-collect`が threshold を最小値に戻すときまでしか効果をもたないだろう。

`gc-cons-percentage` [User Option]

この変数の値はガーベージコレクション発生するまでのコンス (訳注: これは `gc-cons-threshold`や `gc-cons-percentage`の '`-cons-`'のことで、これらの変数が定義されている `alloc.c`内では Lisp 方言での '`cons`'をより一般化したメモリー割り当てプロセスのことを指す模様) の量をカレントヒープサイズにたいする割り合いで指定する。この条件と `gc-cons-threshold`を並行して適用して、条件が両方満足されたときだけガーベージコレクションが発生する。

ヒープサイズ増加にともないガーベージコレクションの処理時間は増大する。したがってガーベージコレクションの頻度割合を減らすのが望ましいことがある。

`gc-cons-threshold`および `gc-cons-percentage`を介した制御は単なる近似です。たとえ Emacs が定期的にしきい値 (threshold) の枯渇をチェックしていても、効率上の理由によりヒープ、または `gc-cons-threshold`や `gc-cons-percentage`の変更のそれぞれにたいして、その後即座にガーベージコレクターをトリガーする訳ではありません。更にしきい値計算の効率化のために、Emacs はヒープ内のカレントでアクセス可能なオブジェクトを計数してヒープサイズを近似します。

`garbage-collect`がリターンする値はデータ型に分類された Lisp データのメモリー使用量を記述します。それとは対照的に関数 `memory-limit`は Emacs がカレントで使用中の総メモリー量の情報を提供します。

`memory-limit` [Function]

この関数は Emacs がカレントで使用中の仮想メモリーの総バイト数を 1024 で除してリターンする。あるアクションがメモリー使用にどのよな効果を及ぼすかについて概観を得るためにこの関数を使用できる。

`memory-full` [Variable]

この変数は Lisp オブジェクト用のメモリーが不足に近い状態なら `t`、それ以外なら `nil`。

`memory-use-counts` [Function]

これはその Emacs セッションで作成されたオブジェクト数をカウントしたリスト。これらのカウンターはそれぞれ特定の種類のオブジェクトを数える。詳細はドキュメント文字列を参照のこと。

`memory-info` [Function]

この関数はシステムメモリーのトータル量とフリーな量をリターンする。サポートされないシステムでは値は `nil`かもしれない。

`gcs-done` [Variable]

この変数はその Emacs セッションでそれまでに行われたガーベージコレクションの合計回数。

`gc-elapsed` [Variable]

この変数はその Emacs セッションでガーベージコレクションの間に費やされた経過時間を浮動小数点数で表した総秒数。

E.4 スタックに割り当てられたオブジェクト

上述のガーベージコレクターは Lisp プログラムから可視なデータ、同様に Lisp インタープリターが内部的に使用するほとんどのデータの管理に使用されます。インタプリターの C スタックを使用して一時的に内部オブジェクトを割り当てることが有用なときがあります。割り当てとガーベージコレクターによる解放は、ヒープメモリーよりスタック割り当てを使用するほうが通常は高速なので、これはパフォーマンスの改善の助けになります。これには解放後にそのようなオブジェクトを使用することにより未定義の挙動を引き起こすという欠点があるので、使用においては熟考するとともに`GC_CHECK_MARKED_OBJECTS`機能 (`src/alloc.c`を参照) を使用して慎重にデバッグするべきです。特にスタックに割り当てられたオブジェクトはユーザーの Lisp コードからは決して可視にならないようにする必要があります。

現在のことろコンスセルと文字列をこの方法で割り当てできます。これは block 寿命をもつ名前つき`Lisp_Object`を定義する`AUTO_CONS`や`AUTO_STRING`のような C マクロで実装されています。これらのオブジェクトはガーベージコレクターでは解放されません。かわりにこれらは自動記憶期間 (automatic storage duration) をもちます。つまりそれらはすべてローカル変数のように割り当てられて、そのオブジェクトを定義する C ブロックの実行の最後に自動的に解放されます。

性能的な理由によりスタックに割り当てられる文字列は ASCII文字に限定されており、それらの多くが不変です。つまりそれらにたいして`ASET`を呼び出すと未定義の挙動を引き起こします。

E.5 メモリー使用量

以下の関数と変数は Emacs が行なったメモリー割り当ての総量に関する情報をデータ型ごとに分類して提供します。これらの関数や変数と`garbage-collect`がリターンする値との違いに注意してください。`garbage-collect`はカレントで存在するオブジェクトを計数しますが、以下の関数や変数はすでに解放されたオブジェクトを含めて割り当てのすべての数やサイズを計数します。

`cons-cells-consed` [Variable]
: その Emacs セッションでそれまでに割り当てられたコンスセルの総数。

`floats-consed` [Variable]
: その Emacs セッションでそれまでに割り当てられた浮動小数点数の総数。

`vector-cells-consed` [Variable]
: その Emacs セッションでそれまでに割り当てられたベクターセルの総数。これにはマーカー、オーバーレイのようなベクター様のオブジェクトに加えてユーザーには不可視な特定のオブジェクトが含まれる。

`symbols-consed` [Variable]
: その Emacs セッションでそれまでに割り当てられたシンボルの総数。

`string-chars-consed` [Variable]
: その Emacs セッションでそれまでに割り当てられた文字列の文字の総数。

`intervals-consed` [Variable]
: その Emacs セッションでそれまでに割り当てられたインターバルの総数。

`strings-consed` [Variable]
: その Emacs セッションでそれまでに割り当てられた文字列の総数。

E.6 C 方言

Emacs の C 部分は C99 にたいして可搬性があります。‘<stdalign.h>’や ‘_Noreturn’のような C11 固有の機能は通常は configure 時に行われるチェックなしでは使用しておらず、Emacs のビルド手順は必要なら代替えの実装を提供します。無名な構造体や共用体のようないくつかの C11 機能はエミュレートが非常に困難なので完全に無視しています。

そう遠くない将来のある時点で基本となる C 方言は間違いなく C11 に変更されるでしょう。

E.7 Emacs プリミティブの記述

Lisp プリミティブとは C で実装された Lisp 関数です。Lisp から呼び出せるように C 関数インターフェースの詳細は C のマクロで処理されます。新たな C コードの記述のしかたを真に理解するにはソースを読むのが唯一の方法ですが、ここではいくつかの事項について説明します。

スペシャルフォームの例として以下は eval.cの orです (通常の関数は同様の一般的な外観をもつ)。

```
DEFUN ("or", For, Sor, 0, UNEVALLED, 0,
       doc: /* Eval args until one of them yields non-nil,
then return that value.
The remaining args are not evalled at all.
If all args return nil, return nil.
usage: (or CONDITIONS...)  */)
  (Lisp_Object args)
{
  Lisp_Object val = Qnil;

  while (CONSP (args))
    {
      val = eval_sub (XCAR (args));
      if (!NILP (val))
        break;
      args = XCDR (args);
      maybe_quit ();
    }

  return val;
}
```

では DEFUNマクロの引数について詳細に説明しましょう。以下はそれらのテンプレートです:

```
DEFUN (lname, fname, sname, min, max, interactive, doc)
```

lname これは関数名として定義する Lisp シンボル名。上記例では or。

fname これは関数の C 関数名。これは C コードでその関数を呼び出すために使用される名前。名前は慣習として ‘F’の後に Lisp 名をつけて、Lisp 名のすべてのダッシュ(‘-’) をアンダースコアに変更する。つまり C コードから呼び出す場合には Forを呼び出す。

sname これは Lisp でその関数を表す subr オブジェクト用にデータ保持のための構造体に使用される C 変数名。この構造体はそのシンボルを作成してそれの定義に subr オブジェクトを格納する初期化ルーチンで Lisp シンボル名を伝達する。慣習により常に *fname*の ‘F’を ‘S’に置き換えた名前になる。

min これは関数が要求する引数の最小個数。関数 orは最小で 0 個の引数を受け入れる。

max これは関数が受け入れる引数の最大個数が定数なら引数の最大個数。あるいは UNEVALLED なら未評価の引数を受け取るスペシャルフォーム、MANYなら評価される引数の個数に制

限がないことを意味する (&rest と等価)。UNEVALLED と MANY はいずれもマクロ。*max* が数字なら *min* より大きく 8 より小さいこと。

interactive

これは Lisp 関数で interactive の引数として使用されるようなインタラクティブ仕様 (文字列) である (Section 21.2.1 [Using Interactive], page 376 を参照)。or の場合は 0 (null ポインター) であり or がインタラクティブに呼び出せないことを示す。値 "" はインタラクティブに呼び出し時に、関数が引数を受け取るべきではないことを示す。値が '"(' で始まる場合には、その文字列は Lisp フォームとして評価される。たとえば:

```
DEFUN ("foo", Ffoo, Sfoo, 0, 3,
       "(list (read-char-by-name \"Insert character: \")\
              (prefix-numeric-value current-prefix-arg)\
              t)",
       doc: /* ... */)
```

doc これはドキュメント文字列。複数行を含むために特別なことを要しないので、これには C の文字列構文ではなく C コメント構文を使用する。'doc:' の後のコメントはドキュメント文字列として認識される。コメントの開始と終了の区切り文字 '/*' と '*/' はドキュメント文字列の一部にはならない。

ドキュメント文字列の最後の行がキーワード 'usage:' で始まる場合には、その行の残りの部分は引数リストをドキュメント化するためのものとして扱われる。この方法により C コード内で使用される引数名とは異なる引数名をドキュメント文字列内で使用することができる。その関数の引数の個数に制限がなければ 'usage:' は必須。

プロットフォームにたいして 1 つといったように、複数の定義をもつプリミティブがいくつかある (たとえば x-create-frame)。このような場合には各定義に同じドキュメント文字列を記述するよりも、ただ 1 つの定義が実際のドキュメントをもつようにしたほうがよい。他の定義は DOC ファイルをパースする関数から無視される、'SKIP' で始まるプレースホルダーをもつ。

Lisp コードでのドキュメント文字列にたいするすべての通常ルール (Section D.6 [Documentation Tips], page 1168 を参照) は C コードのドキュメント文字列にも適用される。

以下のようにドキュメント文字列の後に、そのプリミティブを実装する C 関数にたいする C 関数属性のリストがあるかもしれない:

```
DEFUN ("bar", Fbar, Sbar, 0, UNEVALLED, 0
       doc: /* ... */
       attributes: attr1 attr2 ...)
```

後に続けることにより複数の属性を指定できる。現在のところ以下の属性が認識される:

`noreturn` 決してリターンしない C 関数を宣言する。これは C11 のキーワード _Noreturn、GCC の属性 `__attribute__ ((__noreturn__))` に対応している (Section "Function Attributes" in *Using the GNU Compiler Collection* を参照)。

`const` 引数以外の値を検査せず、リターン値以外に影響しない関数を宣言する。これは GCC の属性 `__attribute__ ((__const__))` に対応している。

`noinline` これは関数がインラインとみなされることを抑止する GCC の属性 `__attribute__ ((__noinline__))` に対応している。これはたとえば

スタックベースの変数にたいするリンク時の最適化の効果を取り消すために必要になるかもしれない。

DEFUNマクロ呼び出しの後には、その C 関数にたいする引数リストを引数のタイプを含めて記述しなければなりません。そのプリミティブが Lisp で固定された最大個数をもつ引数を受け入れるなら、Lisp 引数それぞれにたいして 1 つの C 引数をもち、各引数のタイプは Lisp_Objectでなければなりません (ファイル lisp.hではタイプ Lisp_Objectの値を作成する種々のマクロと関数が宣言されている)。プリミティブがスペシャルフォームなら、評価されないタイプ Lisp_Objectの単一の Lisp 引数を含む Lisp リストを受け取らなければなりません。プリミティブの Lisp の最大引数個数に上限がない場合には正確に 2 つの C 引数をもたなければなりません。1 つ目は Lisp 引数の個数、2 つ目はそれらの値を含むブロックのアドレスです。これらはそれぞれ ptrdiff_t、Lisp_Object *のタイプをもちます。Lisp_Objectは任意のデータ型と任意の Lisp オブジェクトを保持できるので、実行時のみ実際のデータ型を判断できます。特定のタイプの引数だけを受け入れるプリミティブを記述したい場合は、適切な述語を使用してタイプを明確にチェックしなければなりません (Section 2.7 [Type Predicates], page 29 を参照)。

関数 For自体ではローカル変数 argsは Emacs のスタックマーキングによるガーベージコレクターが制御するオブジェクトを参照します。ガーベージコレクターはたとえ C の Lisp_Objectスタック変数から到達可能なオブジェクトを回収しなくても、文字列コンテンツやバッファーのテキストのようなオブジェクトの何らかのコンポーネントを移動するかもしれません。したがってこれらのコンポーネントにアクセスする関数は Lisp の評価を行なった後に、それらのアドレスの再取得に留意しなければなりません。これはコードが文字列コンテンツやバッファーテキストにたいする C ポインターを維持するかわりに、バッファーや文字列の位置を維持して、Lisp の評価を行なった後にその位置から C ポインターを再計算する必要があることを意味しています。Lisp 評価は直接と間接を問わず、eval_sub や Fevalの呼び出しを通じて発生する可能性があります。

ループ内部の maybe_quit呼び出しに注意してください。この関数はユーザーが *C-g*を渡したかどうかをチェックして、もしそうなら処理を abort します。多数の繰り返しを要する可能性があるすべてのループ内でこれを行うべきです。この場合には引数のリストは非常に長くなるかもしれません。これは Emacs の応答性とユーザーエクスペリエンスを向上させます。

Emacs が一度ダンプされた後に変数に何か書き込まれているときには、その静的変数やグローバル変数に C の初期化を使用してはなりません。初期化されたこれらの変数は Emacs のダンプの結果として、(特定のオペレーティングシステムでは) 読み取り専用となるメモリーエリアに割り当てられます。Section E.2 [Pure Storage], page 1177 を参照してください。

C 関数の定義だけでは Lisp プリミティブを利用可能にするのに十分ではありません。そのプリミティブにたいして Lisp シンボルを作成して関数セルに適切な subr オブジェクトを格納しなければなりません。このコードは以下のようになるでしょう:

```
defsubr (&sname);
```

ここで *sname*は DEFUNの 3 つ目の引数として使用する名前です。

すでに Lisp プリミティブが定義されたファイルにプリミティブを追加する場合には、(そのファイル終端付近にある)syms_of_*something*という名前の関数を探して defsubrの呼び出しを追加してください。ファイルにこの関数がない、または新たなファイルを作成する場合には syms_of_*filename* (例: syms_of_myfile) を追加します。それから emacs.cでそれらの関数が呼び出されるすべての箇所を探して syms_of_*filename*の呼び出しを追加してください。

関数 syms_of_*filename*は Lisp 変数として可視となるすべての C 変数を定義する場所でもあります。DEFVAR_LISPはタイプ Lisp_Objectの C 変数を Lisp から可視にします。DEFVAR_INTはタイプ intの C 変数を常に整数となる値をもつようにして Lisp から可視にします。DEFVAR_BOOLはタイプ intの C 変数を常に tか nilのいずれかとなる値をもつようにして Lisp から可視にします。

DEFVAR_BOOLで定義された変数はバイトコンパイラーに使用されるリスト byte-boolean-varsに自動的に追加されることに注意してください。

これらのマクロはすべて 3 つの引数を期待します:

lname Lisp プログラムが使用する変数名。

vname C ソース内の変数名。

doc C コメントとしての変数用のドキュメント。詳細は Section 24.1 [Documentation Basics], page 519 を参照のこと。

慣例として “ネイティブ” なタイプ (intと bool) の変数の定義時には、C の変数名は Lisp 変数の -が_で置換されます。変数がタイプ Lisp_Objectをもつ際には、C の変数名に Vも前置します。たとえば

```
DEFVAR_INT ("my-int-variable", my_int_variable,
            doc: /* An integer variable.  */);

DEFVAR_LISP ("my-lisp-variable", Vmy_lisp_variable,
             doc: /* A Lisp variable.  */);
```

Lisp ではシンボルの値ではなくシンボル自身の参照を要する状況が存在します。1 つは変数の値の一時的なオーバーライドであり、これは Lisp では letで行われます。これは C ソースでは、specbind を使用して対応する定数シンボルを定義することにより行われます。慣例により Qmy_lisp_variable は Vmy_lisp_variableに対応します。これを定義するには DEFSYMマクロを使用します。たとえば

```
DEFSYM (Qmy_lisp_variable, "my-lisp-variable");
```

実際にバインディングを行うには:

```
specbind (Qmy_lisp_variable, Qt);
```

Lisp シンボルではクォートの使用が必要な場合がありますが、C で同様の効果を達成するためには対応する定数シンボル Qmy_lisp_variableを使用します。たとえば Lisp でバッファーローカル変数 (Section 12.11 [Buffer-Local Variables], page 190 を参照) を作成する際には以下のように記述します:

```
(make-variable-buffer-local 'my-lisp-variable)
```

C 側の対応するコードは、以下のように DEFSYMと組み合わせて Fmake_variable_buffer_localを使用します。

```
DEFSYM (Qmy_lisp_variable, "my-lisp-variable");
Fmake_variable_buffer_local (Qmy_lisp_variable);
```

C で定義された Lisp 変数を defcustomで宣言された変数のように振る舞わせたい場合には、cus-start.elに適切なエントリーを追加します。使用するフォーマットの説明は Section 15.3 [Variable Definitions], page 253 を参照してください。

タイプ Lisp_Objectのファイルをスコープとする C 変数を直接定義する場合には、以下のように syms_of_*filename*内で staticproを呼び出してガーベージコレクションから保護しなければなりません:

```
staticpro (&variable);
```

以下はより複雑な引数をもつ別の関数例です。これは window.cからのコードであり、Lisp オブジェクトを操作するためのマクロと関数の使用を示すものです。

```
DEFUN ("coordinates-in-window-p", Fcoordinates_in_window_p,
       Scoordinates_in_window_p, 2, 2, 0,
       doc: /* Return non-nil if COORDINATES are in WINDOW.
  ...
```

```
  or `right-margin' is returned.  */)
  (register Lisp_Object coordinates, Lisp_Object window)
{
  struct window *w;
  struct frame *f;
  int x, y;
  Lisp_Object lx, ly;

  w = decode_live_window (window);
  f = XFRAME (w->frame);
  CHECK_CONS (coordinates);
  lx = Fcar (coordinates);
  ly = Fcdr (coordinates);
  CHECK_NUMBER (lx);
  CHECK_NUMBER (ly);
  x = FRAME_PIXEL_X_FROM_CANON_X (f, lx) + FRAME_INTERNAL_BORDER_WIDTH (f);
  y = FRAME_PIXEL_Y_FROM_CANON_Y (f, ly) + FRAME_INTERNAL_BORDER_WIDTH (f);

  switch (coordinates_in_window (w, x, y))
    {
    case ON_NOTHING:            /* NOT in window at all.  */
      return Qnil;

    ...

    case ON_MODE_LINE:          /* In mode line of window.  */
      return Qmode_line;

    ...

    case ON_SCROLL_BAR:         /* On scroll-bar of window.  */
      /* Historically we are supposed to return nil in this case.  */
      return Qnil;

    default:
      emacs_abort ();
    }
}
```

C コードは C で記述されていなければ名前で呼び出すことはできないことに注意してください。Lisp で記述された関数を呼び出すには関数 `funcall` を C で具現化した `Ffuncall` を使用します。Lisp 関数 `funcall` は個数制限なしの引数を受け付けるので、C での引数は Lisp レベルでの引数個数とそれらの値を含む 1 次元配列という 2 個の引数になります。Lisp レベルでの 1 つ目の引数は呼び出す関数、残りはそれに渡す引数です。

C 関数 `call0`、`call1`、`call2`、... は個数が固定された引数で Lisp 関数を手軽に呼び出す便利な方法を提供します。これらは `Ffuncall` を呼び出すことにより機能します。

`eval.c` は例を探すのに適したファイルです。`lisp.h` には重要なマクロと関数の定義がいくつか含まれています。

副作用がない関数や純粋関数を定義したら、`side-effect-free` や `pure` のプロパティに非 `nil` を与えてください (Section 9.4.2 [Standard Properties], page 130 を参照)。

E.8 動的にロードされるモジュールの記述

このセクションでが Emacs のモジュール API、および Emacs 用の拡張モジュール記述の一部としてそれらを使用する方法について説明します。モジュール API は C プログラム言語で定義されているの

で、このセクション内の記述と例はモジュールが C で記述されていると仮定します。別のプログラム言語では C コード呼び出しのための適切なバインディングやインターフェースと機能の使用が必要になるでしょう。Emacs の C コードには C99 以降のコンパイラー (Section E.6 [C Dialect], page 1184 を参照) が必要であり、このセクションもこの標準にしたがいます。

モジュールの記述と Emacs への統合には以下のタスクが含まれます:

- モジュール用の初期化コードの記述。
- 1 つ以上のモジュール関数の記述。
- Emacs とモジュール間での値とオブジェクトのやり取り。
- エラーコンディションと非ローカル脱出のハンドリング。

以下のサクセクションこれらのタスクと API自体の詳細を説明します。

モジュールを一度記述したら共有ライブラリーを生成するために、背景のプラットドームの慣習に応じてモジュールをコンパイルします。その後で `load-path` (Section 16.3 [Library Search], page 272 を参照) に言及されたディレクトリー内 (Emacs が共有ライブラリーを探す場所) にそれを配置します。

Emacs ダイナミックモジュール APIにたいしてモジュールの適合性を検証したければ*`--module-assertions`*オプションで Emacs を呼び出します。Section “Initial Options” in *The GNU Emacs Manual* を参照してください。

E.8.1 モジュールの初期化コード

ヘッダーファイル `emacs-module.h`のインクルードと GPL 互換性シンボル (GPL compatibility symbol) によりモジュールの記述を始めましょう:

```
#include <emacs-module.h>

int plugin_is_GPL_compatible;
```

Emacs インストールの一部としてシステムのインクルードツリーに `emacs-module.h`ファイルがインストールされます。かわりに Emacs のソースツリー内で見つけることもできます。

次にモジュール用の初期化関数を記述します。

`int emacs_module_init (`*struct emacs_runtime *`runtime`*`)` [Function]

Emacs はモジュールをロードする際にこの関数を呼び出す。モジュールが `emacs_module_init` という名前の関数をエクスポートしていなければモジュールはエラーをシグナルする。この初期化関数は初期化成功時には 0、それ以外は非 0 をリターンすること。後者の場合には Emacs はエラーをシグナルして、モジュールのロードは失敗する。初期化中にユーザーが *`C-g`*を押下すると、Emacs は初期化関数のリターン値を無視して quit する (Section 21.11 [Quitting], page 413 を参照)(必要なら初期化関数内でユーザーの quit を catch できる。[should_quit], page 1200 を参照)。

引数 *runtime*は 2 つのパブリックなフィールドを含む C の `struct`へのポインター、`size`はその構造体のバイトサイズ、`get_environment`は Emacs の環境オブジェクトとそのインターフェースにモジュール初期化関数をアクセス可能にする関数へのポインターを提供する。

初期化関数はモジュールに必要な初期化は何であれすべて行うこと。さらに以下のタスクを行うことができる:

互換性の検証

モジュールにコンパイルされた *runtime*構造体の `size`メンバーの値を比較することにより、モジュールをロードする Emacs 実行可能形式がモジュールと互換性があるか検証できる。

```
int
emacs_module_init (struct emacs_runtime *ert)
{
  if (ert->size < sizeof (*ert))
    return 1;
}
```

モジュールに渡された runtime オブジェクトの size が期待する値より小さければ、ロードしようとしているモジュールが Emacs の新しい (最近の) バージョン向けにコンパイルされていることを意味する (そのモジュールは Emacs のバイナリーとは非互換かもしれない)。

さらにモジュールは期待しているモジュール APIの互換性も検証できる。以下の例では上述の `emacs_module_init`関数の一部であることを仮定している:

```
emacs_env *env = ert->get_environment (ert);
if (env->size < sizeof (*env))
  return 2;
```

これは APIの環境 *(environment)* へのポインターの取得用に構造体 `runtime` (`size`フィールドに構造体のバイトサイズも保有する C`struct`) で提供されるポインターを使用して `get_environment`関数を呼び出す。

最後に Emacs から渡された環境のサイズと既知のサイズを比較することによって、以下のように古いバージョンの Emacs で動作するモジュールを記述できる:

```
emacs_env *env = ert->get_environment (ert);
if (env->size >= sizeof (struct emacs_env_26))
  emacs_version = 26;  /* Emacs 26 or later.  */
else if (env->size >= sizeof (struct emacs_env_25))
  emacs_version = 25;
else
  return 2; /* Unknown or unsupported version.  */
```

新しいバージョンの Emacs ではメンバーを環境に常に追加して削除は決して行わないので、新たな Emacs のリリースではサイズは増加するだけであることにより機能する。与えられたバージョンの Emacs にたいして、そのバージョンに存在するモジュール APIは新しいバージョンのものと等しいので、その部分だけを使用できる。

`emacs-module.h`はプリプロセッサマクロ `EMACS_MAJOR_VERSION`を定義する。これはヘッダーがサポートする Emacs の最新バージョンとなる整数リテラルに展開される。Section 1.4 [Version Info], page 6 を参照のこと。`EMACS_MAJOR_VERSION`はコンパイル時の定数であり、モジュールをロードしたカレントで実行中の Emacs のバージョンを表さないことに注意。`emacs-module.h`や Emacs の種々なバージョンにたいして互換性をもたせたければ、`EMACS_MAJOR_VERSION`にもとづいた条件コンパイルを使用できる。

モジュールが処理全体を初期化関数内で行い、Lisp オブジェクトにアクセスしたり環境構造体を通じてアクセス可能な Emacs 関数を使用することがない場合を除いて、モジュールの互換性検証を常に行うことを推奨する。

Lisp シンボルへのモジュール関数のバインド
: これは Lisp コードが名前で呼び出せるようにモジュール関数に名前を与える。これを行う方法については以下の Section E.8.2 [Module Functions], page 1191 で説明する。

E.8.2 モジュール関数の記述

Emacs モジュールを記述する主な理由は、そのモジュールをロードした Lisp プログラムが追加の関数を利用できるようにするためです。このサブセクションでは、そのようなモジュール関数 *(module functions)* の記述方法を説明します。

モジュール関数は以下のような一般的なフォームとシグネチャをもっています:

`emacs_value module_func` (*emacs_env* ***env**, *ptrdiff_t* **nargs**, *emacs_value* ***args**, *void* ***data**) [Function]

引数 *env*は Emacs のオブジェクトや関数へのアクセスに必要となる API環境へのポインターを提供する。引数 *nargs*は要求される引数の個数であり 0 もあり得る (引数の個数にたいするより柔軟な仕様については以下の `make_function`を参照)。*args*は関数の引数へのポインター。引数 *data*は関数により要求される追加データへのポインターであり、`module_func`から Emacs 関数を作成するために `make_function` (以下参照) が呼び出される際にアレンジされる。

モジュール関数は Emacs とモジュール間で Lisp オブジェクトをやり取りするためにタイプ `emacs_value`を使用する (Section E.8.3 [Module Values], page 1193 を参照)。以下で説明する APIと以降のサブセクションでは C の基本データ型と、それらに対応する `emacs_value` オブジェクトの慣習にたいする機能を提供する。

モジュール関数は常に値をリターンする。関数が正常にリターンすると、それに呼び出された Lisp コードは関数がリターンした `emacs_value`値に対応する Lisp オブジェクトを目にすることになる。しかしユーザーが **C-g**をタイプしたり、モジュール関数やその呼び出し先がエラーをシグナルしたり非ローカルな exit (Section E.8.5 [Module Nonlocal], page 1201 を参照) を行なった場合には、Emacs はリターン値を無視して Lisp コードが同じ状況に遭遇した際のように quit や thrrow を行う。

モジュール関数用の C コード記述後には、そこから `make_function`を使用して Lisp 関数オブジェクトを作成する必要があります。`make_function`関数へのポインターは環境内で提供されます (環境へのポインターは `get_environment`がリターンすることを思い出してほしい)。これは通常は APIのモジュール初期化関数 ([module initialization function], page 1189 を参照) の内部で互換性検証後に行われます。

`emacs_value make_function` (*emacs_env* ***env**, *ptrdiff_t* **min_arity**, *ptrdiff_t* **max_arity**, *subr* **func**, *const char* ***docstring**, *void* ***data**) [Function]

これは上述の `module_func` (ここでは `subr`として `typedef`されたものとしている) で説明したようなシグネチャをもつ C 関数 *func*によって作成された Emacs 関数をリターンする。引数 *min_arity*と *max_arity*は、*func*が受け付ける引数の最大個数と最小個数を指定する。引数 *max_arity*は特別な値 `emacs_variadic_function`をもつことができる。これは Lisp の`&rest`キーワードのように、関数の受け付ける引数の個数を無制限にする (Section 13.2.3 [Argument List], page 212 を参照)。

引数 *data*は *func*の呼び出し時に任意の追加データを渡すための手段である。`make_function` に渡されたポインターが何であれ、それは変更されずに *func*に渡される。

引数 *docstring*はその関数用のドキュメント文字列を指定する。これは ASCII文字列か UTF-8 にエンコードされた非 ASCII文字列、または `NULL`ポインターのいずれかであること。後者の

場合には関数がドキュメントをもたないことを意味する。ドキュメント文字列は advertised-calling-convention を指定する行で終端できる。Section 13.2.4 [Function Documentation], page 213 を参照のこと。

すべてのモジュール関数は 1 つ目の引数として環境へのポインターを受け取らなければならないので、`make_function`は任意のモジュール関数から呼び出され得るが、モジュールが一度ロードされればすべてのモジュール関数が Emacs に既知となるように、通常はこれをモジュール初期化関数で行うことを望むだろう。

最後に Lisp コードが関数を名前で呼び出せるように Lisp 関数をシンボルにバインドする必要があります。これを行うにはモジュール API関数 `intern` ([intern], page 1200 を参照) を使用します。この関数のポインターもモジュール関数がアクセス可能な環境内で提供されています。

上述のステップを組み合わせて、モジュール初期化関数の中で以下のように C 用にアレンジしたコード `module_func`が、Lisp から `module-func`として呼び出し可能になります:

```
emacs_env *env = ert->get_environment (ert);
emacs_value func = env->make_function (env, min_arity, max_arity,
                                       module_func, docstring, data);
emacs_value symbol = env->intern (env, "module-func");
emacs_value args[] = {symbol, func};
env->funcall (env, env->intern (env, "defalias"), 2, args);
```

`env->intern`の呼び出しによりシンボル `module-func`は Emacs が知ることとなり、それから関数をこのシンボルにバインドするために Emacs から `defalias`を呼び出します。`defalias`のかわりに `fset`の使用も可能なことに注意してください。両者の違いは Section 13.4 [Defining Functions], page 215 に説明があります。

`emacs_module_init`関数 ([module initialization function], page 1189 を参照) を含むモジュール関数は、直接間接を問わず Emacs から呼び出されていれば、何らかの生きた `emacs_env`ポインターによる環境関数呼び出しにより、Emacs だけと対話する可能性があります。言い換えると、モジュール関数による Lisp 関数や Emacs プリミティブを呼び出し、`emacs_value`と C データタイプ (Section E.8.3 [Module Values], page 1193 を参照) の間の変換、あるいは別の方法による Emacs との対話を行いたい場合には、Emacs からの `emacs_module_init`やモジュール関数への呼び出しのいくつかがコールスタック上になければなりません。モジュール関数はガーベージコレクションの間は Emacs と対話しないかもしれません。Section E.3 [Garbage Collection], page 1178 を参照してください。モジュール関数は Emacs が作成した Lisp インタープリタースレッド (メインスレッドを含む) とだけ対話するかもしれません。Chapter 37 [Threads], page 929 を参照してください。コマンドラインオプション`--module-assertions`は、上記の要件にたいする違反のいくつかを検知できます。Section “Initial Options” in *The GNU Emacs Manual* を参照してください。

モジュール APIの使用により、より複雑な関数やデータ型 (インタラクティブ関数、インライン関数、マクロ等) の定義が可能になります。ただし C の結果コードは扱いにくく、可読性も低下します。したがって関数やデータ構造を作成するモジュールコードを可能なかぎり最小限に留めるとともに、残りの部分をモジュールに付属する Lisp パッケージに付託することを推奨します。なぜならこれらの追加タスクを Lisp で行うのはより用意であり、より可読性の高いコードが生成されるでしょう。たとえば上記のように定義されたモジュール関数 `module-func`がある場合には、以下のようなシンプルな Lisp ラッパーにもとづくインタラクティブなコマンド `module-cmd`を作成するのも 1 つの方法です:

```
(defun module-cmd (&rest args)
  "Documentation string for the command."
  (interactive spec)
  (apply 'module-func args))
```

モジュールに同梱される Lisp パッケージは、Emacs にパッケージがロードされる際に `load`プリミティブ (Section 16.11 [Dynamic Modules], page 284 を参照) を使用してモジュールをロードできます。

E.8.3 Lisp · モジュール間の値変換

非常に少数の例外を除くほとんどのモジュールでは、モジュールを呼び出す Lisp プログラムとの間でモジュール関数への引数やリターン値の受け渡しでデータのやり取りが必要になります。この目的にたいしてモジュール APIは `emacs_value`タイプを提供しています。これは APIを通じたやり取りにおいて Emacs の Lisp オブジェクトを表現するタイプであり、Emacs の C プリミティブ (Section E.7 [Writing Emacs Primitives], page 1184 を参照) で使用される `Lisp_Object`タイプと機能的には等価です。このセクションでは Lisp の基本データ型に対応する `emacs_value`オブジェクトの作成を可能とするモジュール APIの部分と、Lisp オブジェクトに対応する `emacs_value`オブジェクト内の C データへのアクセス方法について説明します。

以下で説明するすべての関数は、実際にはすべてのモジュール関数が受け取る環境へのポインターを介して提供される関数ポインター *(function pointers)* です。したがってモジュールのコードでは以下のように環境ポインターを通じてこれらの関数を呼び出す必要があります:

```
emacs_env *env;  /* the environment pointer */
env->some_function (arguments...);
```

`emacs_env`ポインターは通常はモジュール関数の 1 つ目の引数、モジュール初期化関数内で環境が必要な場合には `get_environment`の呼び出しから取得できます。

以下で説明するもののほとんどは Emacs 25 で利用可能になった関数であり、Emacs 25 はダイナミックモジュールを最初にサポートした最初の Emacs リリースです。それ以降のリリースで利用可能になったいくつかの関数につていは、それらをサポートする最初の Emacs バージョンを付記します。

以下の API関数は `emacs_value`オブジェクトから種々の C データ型を抽出します。これらすべては引数の `emacs_value`オブジェクトがその関数の期待するタイプでなければ、エラーコンディション `wrong-type-argument`を raise します (Section 2.7 [Type Predicates], page 29 を参照)。Emacs モジュール内でエラーをシグナルする方法、および Emacs にエラーが報告される前にモジュール内部でエラーコンディションを catch する方法の詳細は Section E.8.5 [Module Nonlocal], page 1201 を参照してください。`emacs_value`のタイプ取得には API関数 `type_of`を使用できます (Section E.8.4 [Module Misc], page 1199 を参照)。

`intmax_t extract_integer` (*emacs_env *env, emacs_value arg*) [Function]
: この関数は *arg*で指定された Lisp 整数の値をリターンする。リターン値の C データ型 `intmax_t`は C コンパイラーがサポートする最大の整数型であり、一般的には `long long`。*arg*の値が `intmax_t`に収まらなければ、関数はエラーシンボル `overflow-error`を使用してエラーをシグナルする。

`bool extract_big_integer` (*emacs_env *env, emacs_value arg, int *sign, ptrdiff_t *count, emacs_limb_t *magnitude*) [Function]
: この Emacs 27 から利用可能になった関数は、*arg*の整数値を抽出する。*arg*の値は整数 (fixnum か bignum) でなければならない。*sign*が `NULL`以外なら、*arg*の符号 (-1、0、+1) を`*sign`に格納する。マグニチュード (magnitude: 大きさ) は次のように *magnitude*に格納される。*count* と *magnitude*がいずれも非 `NULL`なら、*magnitude*は少なくとも`*count` `unsigned long`要素の配列を指さなければならない。*magnitude*が *arg*のマグニチュードを保持するのに十分大きければ、この関数は *magnitude*配列にリトルエンディアン形式でマグニチュードを書き込み、配列の要素数を`*count`に格納して `true`をリターンする。*magnitude*の大きさが十分でなけれ

ば、必要な配列サイズを`*count`に格納，エラーをシグナルして`false`をリターンする。*count*が非`NULL`かつ *magnitude*が`NULL`なら、必要となる配列サイズを`*count`に格納して`true`をリターンする。

Emacs は`*count`に要求される最大値が `min (PTRDIFF_MAX, SIZE_MAX) / sizeof (emacs_limb_t)`を超えないことを保証するので、`magnitude`配列の割り当てではサイズ計算で整数のオーバーフローを心配せずに `malloc (*count * sizeof *magnitude)`を使用できる。

`emacs_limb_t` [Type alias]

これは大きい整数向け変換関数のマグニチュード配列の要素タイプとして使用される符号なし整数タイプ。このタイプは一意なオブジェクト表現をもつ (パディングビットがない) ことが保証されている。

`EMACS_LIMB_MAX` [Macro]

これは`emacs_limb_t`にたいして可能な最大値を指定する定数式に展開されるマクロ。この式は`#if`内での利用に適する。

`double extract_float` (*emacs_env* ***env**, *emacs_value* ***arg***) [Function]

この関数は *arg*で指定された Lisp 浮動小数の値を C の `double`値としてリターンする。

`struct timespec` *extract_time* (*emacs_env* ***env**, *emacs_value* ***time***) [Function]

この Emacs 27 から利用可能になった関数は *time*を Emacs Lisp の time 値として解釈して、それに対応する`struct timespec`をリターンする。Section 40.5 [Time of Day], page 1104 を参照のこと。`struct timespec`はナノ秒の精度のタイムスタンプを表す。以下のメンバーをもつ:

`time_t tv_sec`
: 整数秒。

`long tv_nsec`
: ナノ秒としての少数秒数。`extract_time`がリターンするタイムスタンプでは常に非負かつ十億未満 (`tv_nsec`のタイプが`long`であることを POSIX がが要求しているとしても、非標準的なプラットフォームでは`long long`である)。

Section “Elapsed Time” in `libc`を参照のこと。

*time*がナノ秒より高い精度をもつ場合には、この関数はナノ秒の精度へ負の無限大方向に切り詰める。`struct timespec`が (ナノ秒に切り詰めた)*time*を表現できなければ、この関数はエラーをシグナルする。たとえば`time_t`が 32 ビット整数タイプなら 100 億秒という *time*値はエラーをシグナルするが、600 ピコ秒の *time*値は 0 に切り詰められるだろう。

`struct timespec`で表現できない time 値を処理する必要があったり、より高い精度が必要なら Lisp 関数`encode-time`を呼び出してリターン値を処理すればよい。Section 40.7 [Time Conversion], page 1106 を参照のこと。

`bool copy_string_contents` (*emacs_env* ***env**, *emacs_value* ***arg***, *char* ***buf**, *ptrdiff_t* ***len***) [Function]

これは *arg*で指定された Lisp 文字列を UTF-8 にエンコードしたテキストを *buf*が指す`char`配列に格納する。*buf*は少なくとも終端の null バイトを含む`*len`バイトを保持するために十分なスペースをもつこと。引数 *len*は`NULL`ポインターであってはならない。この関数の呼び出し時には *buf*のバイトサイズを指定する値を指していること。

*len*で指定されたバッファーサイズが文字列のテキストを保持するために十分大きければ、関数は終端の null バイト含む実際にコピーされる*len*バイトを *buf* にコピーして `true`をリターンする。バッファーが小さすぎる場合には、関数はエラーコンディション `args-out-of-range` を raise するとともに、必要なバイト数を*len*に格納して `false`をリターンする。保留中のエラーコンディションのハンドル方法は Section E.8.5 [Module Nonlocal], page 1201 を参照のこと。

引数 *buf* は `NULL`ポインターでもよく、この場合には関数は *arg*のコンテンツの格納に必要なバイト数を*len*に格納して `true`をリターンする。これは特定の文字列を格納するために必要な *buf*サイズを決定する手段となり得る。1 回目は *buf* を `NULL`で `copy_string_contents` を呼び出して、関数により*len*に格納されたバイト数の保持に十分なメモリーを割り当ててから、実際にテキストのコピーを行うために非 `NULL`の *buf* で関数を再び呼び出す。

`emacs_value vec_get` (*emacs_env *env, emacs_value vector, ptrdiff_t index*) [Function]

この関数は *vector*の *index*の要素をリターンする。ベクターの最初の要素の *index*は 0。*index* の値が無効ならこの関数はエラーコンディション `args-out-of-range`を raise する。関数のリターン値から C データを抽出するためには、ベクターの当該要素に格納された Lisp データタイプに応じて、ここで説明している他の抽出関数を使用すればよい。

`ptrdiff_t vec_size` (*emacs_env *env, emacs_value vector*) [Function]

この関数は *vector*内の要素数をリターンする。

`void vec_set` (*emacs_env *env, emacs_value vector, ptrdiff_t index, emacs_value value*) [Function]

この関数は *vector*のインデックス *index*の要素に *value*を格納する。*index*の値が無効ならこの関数はエラーコンディション `args-out-of-range`を raise する。

以下は C の基本データ型から `emacs_value`オブジェクトを作成する API関数です。これらはすべて作成した `emacs_value`オブジェクトをリターンします。

`emacs_value make_integer` (*emacs_env *env, intmax_t n*) [Function]

この関数は引数 *n* (整数) を受け取り対応する `emacs_value`オブジェクトをリターンする。*n* の値が `most-negative-fixnum`と `most-positive-fixnum`の内外いずれであるかに応じて fixnum か bignum のいずれかをリターンする (Section 3.1 [Integer Basics], page 37 を参照)。*n*の値を Emacs 整数で表現できない、すなわち `most-negative-fixnum`と `most-positive-fixnum`の範囲外 (Section 3.1 [Integer Basics], page 37 を参照) ならエラーコンディション `overflow-error`を raise する。

`emacs_value make_big_integer` (*emacs_env *env, int sign, ptrdiff_t count, const emacs_limb_t *magnitude*) [Function]

この Emacs 27 から利用可能になった関数は任意のサイズの整数の引数を受け取り、それに対応する `emacs_value`オブジェクトをリターンする。*sign*引数はリターン値の符号を与える。*sign*が 0 以外なら、*magnitude*はリターン値のマグニチュードをリトルエンディアンで指定する、少なくとも *count*個の要素数の配列を指さなければならない。

以下は与えられた整数の次の確率的素数を計算するために GNU Multiprecision Library (GMP) を使用する例です。GMP の概要は `gmp`、`magnitude`と GMP の `mpz_t`値との間の変換方法については Section “Integer Import and Export” in `gmp`を参照してください。

```
#include <emacs-module.h>
```

```
int plugin_is_GPL_compatible;

#include <assert.h>
#include <limits.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#include <gmp.h>

static void
memory_full (emacs_env *env)
{
  static const char message[] = "Memory exhausted";
  emacs_value data = env->make_string (env, message,
                                       strlen (message));
  env->non_local_exit_signal
    (env, env->intern (env, "error"),
     env->funcall (env, env->intern (env, "list"), 1, &data));
}

enum
{
  order = -1, endian = 0, nails = 0,
  limb_size = sizeof (emacs_limb_t),
  max_nlimbs = ((SIZE_MAX < PTRDIFF_MAX ? SIZE_MAX : PTRDIFF_MAX)
                / limb_size)
};

static bool
extract_big_integer (emacs_env *env, emacs_value arg, mpz_t result)
{
  ptrdiff_t nlimbs;
  bool ok = env->extract_big_integer (env, arg, NULL, &nlimbs, NULL);
  if (!ok)
    return false;
  assert (0 < nlimbs && nlimbs <= max_nlimbs);
  emacs_limb_t *magnitude = malloc (nlimbs * limb_size);
  if (magnitude == NULL)
    {
      memory_full (env);
      return false;
    }
  int sign;
  ok = env->extract_big_integer (env, arg, &sign, &nlimbs, magnitude);
  assert (ok);
  mpz_import (result, nlimbs, order, limb_size, endian, nails, magnitude);
```

```
  free (magnitude);
  if (sign < 0)
    mpz_neg (result, result);
  return true;
}

static emacs_value
make_big_integer (emacs_env *env, const mpz_t value)
{
  size_t nbits = mpz_sizeinbase (value, 2);
  int bitsperlimb = CHAR_BIT * limb_size - nails;
  size_t nlimbs = nbits / bitsperlimb + (nbits % bitsperlimb != 0);
  emacs_limb_t *magnitude
    = nlimbs <= max_nlimbs ? malloc (nlimbs * limb_size) : NULL;
  if (magnitude == NULL)
    {
      memory_full (env);
      return NULL;
    }
  size_t written;
  mpz_export (magnitude, &written, order, limb_size, endian, nails, value);
  assert (written == nlimbs);
  assert (nlimbs <= PTRDIFF_MAX);
  emacs_value result = env->make_big_integer (env, mpz_sgn (value),
                                              nlimbs, magnitude);
  free (magnitude);
  return result;
}

static emacs_value
next_prime (emacs_env *env, ptrdiff_t nargs, emacs_value *args,
            void *data)
{
  assert (nargs == 1);
  mpz_t p;
  mpz_init (p);
  extract_big_integer (env, args[0], p);
  mpz_nextprime (p, p);
  emacs_value result = make_big_integer (env, p);
  mpz_clear (p);
  return result;
}

int
emacs_module_init (struct emacs_runtime *ert)
{
  emacs_env *env = ert->get_environment (ert);
```

```
    emacs_value symbol = env->intern (env, "next-prime");
    emacs_value func
      = env->make_function (env, 1, 1, next_prime, NULL, NULL);
    emacs_value args[] = {symbol, func};
    env->funcall (env, env->intern (env, "defalias"), 2, args);
    return 0;
  }
```

`emacs_value make_float` (*emacs_env* ***env**, *double* ***d***) [Function]

この関数は `double`の引数 *d*を受け取り対応する Emacs 浮動小数点値をリターンする。

`emacs_value make_time` (*emacs_env* ***env**, *struct timespec* ***time***) [Function]

この Emacs 27 から利用可能になった関数は `struct timespec`の引数 *time*を受け取り、(***ticks . hz***)というペアーとしてそれに対応する Emacs タイムスタンプをリターンする。リターン値は正確に *time*と同一のタイムスタンプを表す。つまりすべての入力値は表現可能であり、精度を失うことは決してない。***time***`.tv_sec`および ***time***`.tv_nsec`は任意の値をとり得る。特に *time*が正規化されている必要はない。これは ***time***`.tv_nsec`が負、あるいは 999,999,999 より大きくなり得ることを意味する。

`emacs_value make_string` (*emacs_env* ***env**, *const char* ***str**, *ptrdiff_t* ***strlen***) [Function]

この関数は *str*が指す、終端の null バイトを含まないバイト長が *strlen*であるような C テキスト文字列から Emacs 文字列を作成する。*str*の元文字列は ASCII文字列か UTF-8 にエンコードされた非 ASCII文字列が可能であり、文字列には埋め込みの null バイトを含むことができ、***str***[***strlen***]にある null バイトで終端される必要はない。*strlen*が負、または Emacs 文字列の最大長を超過する場合には、この関数はエラーコンディション `overflow-error`を raise する。

この APIはたとえば `cons`と `list`によるリスト作成 (Section 5.4 [Building Lists], page 75 を参照)、`car`と `cdr`によるリストメンバーの抽出 (Section 5.3 [List Elements], page 73 を参照)、`vector`によるベクター作成 (Section 6.5 [Vector Functions], page 109 を参照) 等のような Lisp データ構造を操作する関数は提供しません。これらにたいしてはたいおうする Lisp 関数を呼び出すために、次のサブセクションで説明する `intern`と `funcall`を使用します。

`emacs_value`オブジェクトのライフタイムはかなり短いのが普通です。このライフタイムはオブジェクトの作成に使用された `emacs_env`ポインターがスコープ外になると終了します。`emacs_value`が望む間は行き続けるようなグローバル参照 *(global references)* を作成を要する場合もあるかもしれません。そのようなオブジェクトの管理には以下の 2 つの関数を使用します。

`emacs_value make_global_ref` (*emacs_env* ***env**, *emacs_value* ***value***) [Function]

この関数は *value*のグローバル参照をリターンする。

`void free_global_ref` (*emacs_env* ***env**, *emacs_value* ***global_value***) [Function]

この関数は以前に `make_global_ref`で作成した *global_value*を解放する。*global_value*はこの呼び出し後は無効になる。モジュールのコードでは `make_global_ref`と対応する `free_global_ref`の呼び出しそれぞれをペアーとすること。

後でモジュール関数に渡す必要がある C データ構造体を追跡するための代替え手段はユーザーポインター *(user pointer)* オブジェクトの作成です。ユーザーポインター (または `user-ptr`) は C ポインターをカプセル化した Lisp オブジェクトであり、関連付けられたファイナライザー (オブジェクトがガーベージコレクトされる際に呼び出される。Section E.3 [Garbage Collection], page 1178 を参照) をもつことができます。モジュール APIは `user-ptr`オブジェクトの作成やアクセスを行う関数を提供します。これらの関数は `user-ptr`オブジェクトを表現しない `emacs_value`で呼び出されるとエラーコンディション `wrong-type-argument`を raise します。

`emacs_value make_user_ptr` (*emacs_env *env*, *emacs_finalizer fin*, *void *ptr*) [Function]

この関数は C ポインター *ptr*をラップした `user-ptr`オブジェクトを作成してリターンする。ファイナライザー関数 *fin*は `NULL` (ファイナライザーなし)、または以下のシグネチャをもつ関数のいずれか:

```
typedef void (*emacs_finalizer) (void *ptr);
```

*fin*が `NULL`ポインターでなければ、`user-ptr`オブジェクトがガーベージコレクトされる際に *ptr*を引数として呼び出される。Emacs の応答性を維持するために GC は短時間で終了しなければならないので、ファイナライザーでは高価なコードの実行は行ってはならない。

`void *get_user_ptr` (*emacs_env *env*, *emacs_value val*) [Function]

この関数は *val*で表される Lisp オブジェクトから C ポインターを抽出する。

`void set_user_ptr` (*emacs_env *env*, *emacs_value value*, *void *ptr*) [Function]

この関数は *value*で表される `user-ptr`オブジェクトに埋め込まれた C ポインターに *ptr*をセットする。

`emacs_finalizer get_user_finalizer` (*emacs_env *env*, *emacs_value val*) [Function]

この関数は *val*で表される `user-ptr`オブジェクトのファイナライザー、ファイナライザーがなければ `NULL`をリターンする。

`void set_user_finalizer` (*emacs_env *env*, *emacs_value val*, *emacs_finalizer fin*) [Function]

この関数は *val*で表される `user-ptr`オブジェクトのファイナライザーを *fin*に変更する。*fin*が `NULL`なら `user-ptr`オブジェクトのファイナライザーはなくなる。

E.8.4 その他の便利なモジュール用関数

このサブセクションではモジュール APIが提供する便利な関数をいくつか説明します。前のサブセクションで説明した関数と同じようにこれらの関数は実際には関数ポインターであり、`emacs_env`ポインターを介して呼び出す必要があります。Emacs 25 以降に導入された関数の説明はそれらが利用可能になった最初のバージョンを付記します。

`bool eq` (*emacs_env *env*, *emacs_value val1*, *emacs_value val2*) [Function]

この関数は *val1*と *val2*が表す Lisp オブジェクトが等しければ `true`、それ以外なら `false`をリターンする。これは Lisp 関数 `eq` (Section 2.8 [Equality Predicates], page 33 を参照) と同じだが、引数が表すオブジェクトの intern の要否を無視する。

等価性に関する他の述語は API関数には存在しないので、より複雑な等価性のテストを行うためには、以下で説明する `intern`と `funcall`を使う必要がある。

`bool is_not_nil` (*emacs_env* ***env**, *emacs_value* `val`) [Function]
このえ関数は *val*で表される Lisp オブジェクトをテストして非 `nil`なら `true`、それ以外は `false`をリターンする。

等価性をテストするたに `intern`を使って `nil`を表す `emacs_value`を取得して、上述の `eq`を使用すれば自身で等価性テストを実装できることに注意。しかしこの関数を使用するほうが簡便だろう。

`emacs_value type_of` (*emacs_env* ***env**, *emacs_value* `object`) [Function]
この関数はシンボルを表す値 (文字列は `string`、整数は `integer`、プロセスなら `process`等) として *object*のタイプをリターンする。Section 2.7 [Type Predicates], page 29 を参照のこと。オブジェクトのタイプにコードが依存する必要があれば、既知のタイプシンボルと比較するために `intern`と `eq`を使用できる。

`emacs_value intern` (*emacs_env* ***env**, *const char* ***name**) [Function]
この関数は名前が *name* (null 終端された ASCII文字列であること) であるような、intern された Emacs シンボルをリターンする。すでに存在していなければ新たにシンボルを作成する。

この関数は以下で説明する `funcall`と共に用いることにより、Lisp で呼び出し可能な Emcas 関数 (名前が純粋な ASCII文字列である場合にかぎる) 純粋な ASCII文字列であるようなを呼び出す手段を提供する。たとえば以下はより協力な Emacs の `intern`関数 (Section 9.3 [Creating Symbols], page 126 を参照) を呼び出すことにより、名前 `name_str`が非 ASCIIであるようなシンボルを intern する方法:

```
emacs_value fintern = env->intern (env, "intern");
emacs_value sym_name =
  env->make_string (env, name_str, strlen (name_str));
emacs_value intern_args[] = { sym_name, env->intern (env, "nil") };
emacs_value symbol = env->funcall (env, fintern, 2, intern_args);
```

`emacs_value funcall` (*emacs_env* ***env**, *emacs_value* ***func**, *ptrdiff_t* ***nargs**, *emacs_value* ***args**) [Function]
この関数は *args*が指す配列の *nargs*個の引数を渡して *func*の指定先を呼び出す。引数 *func*は (上述の `intern`がリターンした) 関数シンボル、`make_function`がリターンしたモジュール関数 (Section E.8.2 [Module Functions], page 1191 を参照)、C で記述されたサブルーチン等。*nargs*が 0 なら *args*は `NULL`ポインターでもよい。

この関数は *func*がリターンした値をリターンする。

モジュールに長時間実行される可能性のあるコードが含まれる場合には、たとえば *C-g*をタイプする (Section 21.11 [Quitting], page 413 を参照) 等によりユーザーが quit を望むかどうかをコード内でときどきチェックするのはよいアイデアです。Emacs 26.1 から利用可能になった以下の関数は、この目的のために提供されました。

`bool should_quit` (*emacs_env* ***env**) [Function]
この関数はユーザーが quit を望むようなら `true`をリターンする。この場合にはモジュール関数は実行中の処理を abort して可能なかぎり速やかにリターンすることを推奨する。ほとんどの場合は `process_input`を使用すること。

ユーザーが quit を望むかどうかをチェックすることに加えて入力イベントを処理するには、Emacs 27.1 以降で利用可能になった以下の関数を使用してください。

`enum emacs_process_input_result` *process_input* (*emacs_env* ***env*) [Function]
この関数は保留中の入力イベントを処理する。ユーザーが quit を望んでいたり、シグナル処理中にエラーが発生したら `emacs_process_input_quit`をリターンする。この場合にはモジュール関数は行っているすべての処理を abort して可能なかぎり即座にリターンすることを推奨する。モジュールコードが実行を継続できるなら、`process_input`は `emacs_process_input_continue`をリターンする。`env`内に保留中の非ローカル exit が存在しない場合のみ、リターン値は `emacs_process_input_continue`。`process_input`呼び出し後にモジュールが継続する場合には、変数値やバッファーコンテンツのとうなグローバル状態は任意の手段で変更され得る。

E.8.5 モジュールでの非ローカル脱出

Emacs Lisp は非ローカル脱出 (nonlocal exits) をサポートしており、これによりプログラムの制御はプログラムのあるポイントから別の離れたポイントに転送されます。Section 11.7 [Nonlocal Exits], page 162 を参照してください。したがってモジュールから呼び出された Lisp 関数は `signal`や `throw`を呼び出して非ローカルに exit するかもしれず、そのような非ローカル脱出をモジュール関数は正しくハンドルしなければなりません。このようなハンドリングは C プログラムがリソースを自動的に解放せず、このような場合には別のクリーンアップを行うために必要になります。モジュールコードは自身でこれを行わなければなりません。そのための機能をモジュール APIは提供しており、このサブセクションではそれを説明します。これらは一般的には Emacs 25 以降で利用可能です。これ以降のリリースで利用可能になったものについては、APIに含まれるようになった最初の Emacs のバージョンを付記します。

モジュール関数から呼び出された Lisp コードがエラーをシグナルしたり throw を行う際には、非ローカル脱出は trap されて保留中の exit と関連するデータは環境内に格納されます。環境内で非ローカル脱出が保留中の際には、環境へのポインターで呼び出されたすべてのモジュール API関数は何も処理を行わずに即座にリターンします (関数 `non_local_exit_check`、`non_local_exit_get`、`non_local_exit_clear`はこのルールの例外)。モジュール関数が何も行わずに Emacs にリターンすれば、保留中の非ローカル脱出にたいして Emacs がエラーをシグナルしたり、対応する `catch`への throw という対処を行うでしょう。

したがって特別なことな何も行わずに、何事もなかったかのようにコードの残りを実行するのが、モジュール関数での非ローカル脱出にけるもっともシンプルな “ハンドリング” です。しかしこれは 2 つのクラスの問題を引き起こすかもしれません:

– 期待する値を生成することなく API関数は即座にリターンするので、初期化や定義が行われていない値をモジュール関数が使用するかもしれない。
– リソースを解放する機会がないかもしれないのでモジュールがリソースをリークするかもしれない。

したがってモジュール関数は以下に説明する関数を使用して、非ローカル脱出のコンディションのチェックとリカバリングを行うことを推奨します。

`enum emacs_funcall_exit` *non_local_exit_check* (*emacs_env* ***env*) [Function]
この関数は *env*に格納された非ローカル脱出のコンディションをリターンする。可能な値は:

`emacs_funcall_exit_return`
最後の API関数は正常に exit した。

`emacs_funcall_exit_signal`
最後の API関数はエラーをシグナルした。

`emacs_funcall_exit_throw`
最後の API関数は `throw`を通じて exit した。

`enum emacs_funcall_exit` *non_local_exit_get* (*emacs_env* ***env*, *emacs_value* ***symbol*, *emacs_value* ***data*) [Function]
この関数は `non_local_exit_check`が行うように *env*に格納された非ローカル脱出の種別をリターンするが、もしあれば非ローカル脱出に関する完全な情報もリターンする。リターン値が `emacs_funcall_exit_signal`なら関数は`*symbol`にエラーシンボル、`*data`にエラーデータを格納する (Section 11.7.3.1 [Signaling Errors], page 165 を参照)。リターン値が `emacs_funcall_exit_throw`なら関数は`*symbol`に `catch`されたたタグシンボル、`*data`に `throw`された値を格納する。リターン値が `emacs_funcall_exit_return`なら関数はこれらの引数が指すメンモリー内に何も格納しない。

何らかのリソースの割り当て前や解放を要するリソースの割り当て後、あるいは失敗がそれ以上の処理が不可能もしくは実行不能を意味するような場合のように、非ローカル脱出が問題になるようならチェックするべきです。

モジュール関数が保留中の非ローカル脱出を一度検知すれば、(必要なローカルクリーンアップの実施後に)Emacs にリターンしたり、非ローカル脱出からのリカバリーを試みることができます。以下の API関数はこれらのタスクの助けとなるでしょう。

`void non_local_exit_clear` (*emacs_env* ***env*) [Function]
この関数は保留中の非ローカル脱出のコンディションと *env*由来のデータをクリアーする。これの呼び出し後にはモジュール API関数は通常どおり機能するだろう。モジュール関数が呼び出した Lisp 関数の非ローカル脱出からリカバーして継続可能な場合、あるいは以下の 2 つの関数のいずれか (非ローカル脱出が保留中の際に他の API関数に意図した動作を行わせたい場合にはそれらの API関数も) を呼び出す前にもこの関数を使用すること

`void non_local_exit_throw` (*emacs_env* ***env*, *emacs_value* *tag*, *emacs_value* *value*) [Function]
この関数は *tag*で表される Lisp の `catch`シンボルにリターン値として *value*を渡して throw を行う。モジュール関数は一般的にはこの関数の呼び出し後は即座にリターンすること。この関数は呼び出された API関数や Lisp 関数のいずれかから非ローカル脱出を再 throw したい際の 1 つの手段である。

`void non_local_exit_signal` (*emacs_env* ***env*, *emacs_value* *error*, *emacs_value* *data*) [Function]
この関数は *error*で表されるエラーを、指定したエラーデータ *data*とともにシグナルする。モジュール関数はこの関数の呼び出し後は即座にリターンすること。この関数はたとえばモジュール関数から Emacs にエラーをシグナルする際に有用かもしれない。

E.9 オブジェクトの内部

Emacs Lisp は豊富なデータタイプのセットを提供します。コンスセル、整数、文字列のようにこれらのいくつかは、ほとんどすべての Lisp 方言で一般的です。マーカやバッファーのようなそれ以外のものは Lisp 内でエディターコマンドを記述するための基本的サポートを提供するために極めて特別かつ必要なものです。そのような種々のオブジェクトタイプを実装してインタープリターのサブシステムとの間でオブジェクトを渡す効果的な方法を提供するために、C データ構造体セットとそれらすべてにたいするポインターを表すタグ付きポインター *(tagged pointer)* と呼ばれる特別なタイプが存在します。

C ではタグ付きポインターはタイプ Lisp_Objectのオブジェクトです。そのようなタイプの初期化された変数は基本的なデータタイプである整数、シンボル、文字列、コンスセル、浮動小数点数、ベクター類似オブジェクトのいずれかを値として常に保持します。これらのデータタイプのそれぞれは対応するタグ値をもちます。すべてのタグは enum Lisp_Typeにより列挙されており、Lisp_Objectの 3 ビットのビットフィールドに配置されます。残りのビットはそれ自身の値です。整数は即値 (値ビットで直接表される)、他のすべてのオブジェクトはヒープに割り当てられた対応するオブジェクトへの C ポインターで表されます。Lisp_Objectのサイズはプラットフォームと設定に依存します。これは通常は背景プラットフォームのポインターと同一 (32 ビットマシンなら 32 ビット、64 ビットマシンなら 64 ビット) ですが Lisp_Objectが 64 ビットでも、すべてのポインターが 32 ビットのような特別な構成もあります。後者は Lisp_Objectにたいして 64 ビットの long longタイプを使用することにより、32 ビットシステム上の Lisp 整数にたいする値範囲の制限を乗り越えるためにデザインされたトリックです。

以下の C データ構造体は整数ではない基本的なデータタイプを表すために lisp.hで定義されています:

struct Lisp_Cons
: コンスセル。リストを構築するために使用されるオブジェクト。

struct Lisp_String
: 文字列。文字シーケンスを表す基本的オブジェクト。

struct Lisp_Vector
: 配列。インデックスによりアクセスできる固定サイズの Lisp オブジェクトのセット。

struct Lisp_Symbol
: シンボル。一般的に識別子として使用される一意な名前のエンティティ。

struct Lisp_Float
: Floating-point value.

これらのタイプは内部的タイプシステムのファーストクラスの市民です。タグスペースは限られているので他のすべてのタイプは Lisp_Vectorlikeのサブクラスです。サブタイプのベクターは enum pvec_typeにより列挙されておりウィンドウ、バッファー、フレーム、プロセスのようなほとんどすべての複雑なオブジェクトはこのカテゴリーに分類されます。

Lisp_Vectorlikeのいくつかのサブタイプを説明します。バッファーオブジェクトは表示や編集を行うテキストを表します。ウィンドウはバッファーを表示したり、同一フレーム上で再帰的に他のウィンドウを配置するためのコンテナーとして使用される表示構造の一部です (Emacs Lisp のウィンドウオブジェクトと、のようなユーザーインターフェースシステムに管理されるエンティティとしてのウィンドウを混同しないこと。Emacs の用語では後者はフレームと呼ばれる)。最後にプロセスオブジェクトはサブプロセスの管理に使用されます。

E.9.1 バッファーの内部

C でバッファーを表すために 2 つの構造体 (buffer.hを参照) が使用されます。buffer_text構造体にはバッファーのテキストを記述するフィールドが含まれます。buffer構造体は他のフィールドを保持します。インダイレクトバッファーの場合には、2 つ以上の buffer構造体が同じ buffer_text構造体を参照します。

以下に struct buffer_text内のフィールドをいくつか示します:

beg
: バッファーコンテンツのアドレス。バッファーコンテンツは途中にギャップをもつ char の線形 C 配列。

`gpt`
`gpt_byte` バッファーのギャップの文字位置とバイト位置。Section 27.13 [Buffer Gap], page 606 を参照のこと。

`z`
`z_byte` バッファーテキストの終端の文字位置とバイト位置。

`gap_size` バッファーのギャップのサイズ。Section 27.13 [Buffer Gap], page 606 を参照のこと。

`modiff`
`save_modiff`
`chars_modiff`
`overlay_modiff`
これらのフィールドは、そのバッファーで行われたバッファー変更イベントの数をカウントする。`modiff`はバッファー変更イベントのたびに増分されて、それ以外では決して変化しない。`save_modiff`にはバッファーが最後に visit や保存されたときの`modiff`の値が含まれる。`chars_modiff`はバッファー内の文字にたいする変更だけをカウントして、(テキストプロパティのように) その他すべての種類の変更を無視する。`overlay_modiff`はバッファーのオーバーレイにたいする変更だけをカウントする。

`beg_unchanged`
`end_unchanged`
最後の再表示完了以降に未変更だと解っているテキスト、開始と終了の箇所での文字数。

`unchanged_modified`
`overlay_unchanged_modified`
それぞれ最後に再表示が完了した後の `modiff`と `overlay_modiff`の値。これらのカレント値が `modiff`や `overlay_modiff`とマッチしたら、それは `beg_unchanged`と `end_unchanged`に有用な情報が含まれないことを意味する。

`markers` このバッファーを参照するマーカー。これは実際には単一のマーカーであり、自身のマーカーチェーン (リンクリスト) 内の一連の要素がバッファー内のテキストを参照する他のマーカーになる。

`intervals`
そのバッファーのテキストプロパティを記録するインターバルツリー。

`struct buffer`のいくつかのフィールドを以下に示します:

`header` タイプ`union vectorlike_header`のヘッダーは、すべてのベクター類似のオブジェクトに共通。

`own_text` 構造体`struct buffer_text`は通常はバッファーのコンテンツを保持する。このフィールドはインダイレクトバッファーでは使用されない。

`text` そのバッファーの`buffer_text`構造体へのポインター。通常のバッファーでは上述の`own_text`フィールド。インダイレクトバッファーではベースバッファーの `own_text` フィールド。

`next` kill されたバッファーを含むすべてのバッファーのチェーン内において次のバッファーへのポインター。このチェーンは kill されたバッファーを正しく回収するために割り当てとガーベージコレクションのためだけに使用される。

`pt`
`pt_byte` バッファー内のポイントの文字位置とバイト位置。

`begv`
`begv_byte`
: そのバッファー内のアクセス可能範囲の先頭位置の文字位置とバイト位置。

`zv`
`zv_byte` そのバッファー内のアクセス可能範囲の終端位置の文字位置とバイト位置。

`base_buffer`
: インダイレクトバッファーではベースバッファーのポイント。通常のバッファーでは null。

`local_flags`
: このフィールドはバッファー内でローカルな変数にたいしてそれを示すフラグを含む。そのような変数は C コードでは `DEFVAR_PER_BUFFER`を使用して宣言され、それらのバッファーローカルなバインディングはバッファー構造体自身内のフィールドに格納される (これらのフィールドのいくつかはこのテーブル内で説明している)。

`modtime` visit されているファイルの変更時刻。これはファイルの書き込みと読み込み時にセットされる。そのバッファーをファイルに書き込む前にファイルがディスク上で変更されていないことを確認するために、このフィールドとそのファイルの変更時刻を比較する。Section 27.5 [Buffer Modification], page 596 を参照のこと。

`auto_save_modified`
: そのバッファーが最後に自動保存されたときの時刻。

`last_window_start`
: そのバッファーが最後にウィンドウに表示されたときのバッファー内での `window-start` 位置。

`clip_changed`
: このフラグはバッファーでのナローイングが変更されているかを示す。Section 30.4 [Narrowing], page 766 を参照のこと。

`prevent_redisplay_optimizations_p`
: このフラグはバッファーの表示において再表示最適化が使用されるべきではないことを示す。

`overlay_center`
: このフィールドはカレントオーバーレイの中心位置を保持する。Section 39.9.1 [Managing Overlays], page 1000 を参照のこと。

`overlays_before`
`overlays_after`
: これらのフィールドはカレントオーバーレイ中心、またはその前で終わるオーバーレイのリスト、およびカレントオーバーレイの後で終わるオーバーレイのリスト。Section 39.9.1 [Managing Overlays], page 1000 を参照のこと。`overlays_before`は終端位置の記述順、`overlays_after`は先頭位置増加順で格納される。

`name` バッファーを命名する Lisp 文字列。一意であることが保証されている。Section 27.3 [Buffer Names], page 592 を参照のこと。このフィールドと以降のフィールドは以下のように `BVAR`を介するアクセス以外の方法で直接アクセスするべきではないことを示すために C 構造体定義内の名前の最後に`_`をもつ:

```
Lisp_Object buf_name = BVAR (buffer, name);
```

`save_length`
: そのバッファーが visit しているファイルを最後に読み込み、または保存したときの長さ。2 つの特別な値をもつことができる。−1 はそのバッファーで自動保存がオフであること、−2 はバッファーのテキストが大量に減少するようなら自動保存をオフに切り替えないことを意味する。インダイレクトバッファーは決して保存されることはないので、保存に関して、このフィールドとその他のフィールドは `buffer_text`構造体で維持されない

`directory`
: 相対ファイル名を展開するディレクトリー。これはバッファーローカル変数 `default-directory`の値 (Section 25.9.4 [File Name Expansion], page 559 を参照)。

`filename`
: そのバッファーが visit しているファイルの名前。これはバッファーローカル変数 `buffer-file-name`の値 (Section 27.4 [Buffer File Name], page 594 を参照)。

`undo_list`
`backed_up`
`auto_save_file_name`
`auto_save_file_format`
`read_only`
`file_format`
`file_truename`
`invisibility_spec`
`display_count`
`display_time`
: これらのフィールドは自動的にバッファーローカル (Section 12.11 [Buffer-Local Variables], page 190 を参照) になる Lisp 変数の値を格納する。これらに対応する変数は名前に追加のプレフィクス `buffer-`がつき、アンダースコアがダッシュで置換される。たとえば `undo_list`は `buffer-undo-list`の値を格納する。

`mark`
: そのバッファーにたいするマーク。マークはマーカーなのでリスト `markers`内にも含まれる。Section 31.7 [The Mark], page 772 を参照のこと。

`local_var_alist`
: この連想リストはバッファーのバッファーローカル変数のバインディングを記述する。これにはバッファーオブジェクト内に特別なスロットをもつ、ビルトインのバッファーローカルなバインディングは含まれない (このテーブルではそれらのスロットは省略している)。Section 12.11 [Buffer-Local Variables], page 190 を参照のこと。

`major_mode`
: そのバッファーのメジャーモードを命名するシンボル (例: `lisp-mode`)。

`mode_name`
: そのメジャーモードの愛称 (例: `"Lisp"`)。

`keymap`
`abbrev_table`
`syntax_table`
`category_table`
`display_table`
: これらのフィールドはバッファーのローカルキーマップ (Chapter 22 [Keymaps], page 422 を参照)、abbrev テーブル (Section 36.1 [Abbrev Tables], page 922 を参照)、構文テーブル (Chapter 35 [Syntax Tables], page 906 を参照)、カテゴリーテーブル (Section 35.8 [Categories], page 918 を参照)、ディスプレーテーブル (Section 39.22.2 [Display Tables], page 1079 を参照) を格納する。

`downcase_table`
`upcase_table`
`case_canon_table`
: これらのフィールドはテキストを小文字、大文字、および case-fold 検索でのテキストの正規化の変換テーブルを格納する。Section 4.10 [Case Tables], page 69 を参照のこと。

`minor_modes`
: そのバッファーのマイナーモードの alist。

`pt_marker`
`begv_marker`
`zv_marker`
: これらのフィールドはインダイレクトバッファー、またはインダイレクトバッファーのベースバッファーであるようなバッファーでのみ使用される。これらはそれぞれバッファーがカレントでないときにバッファーにたいする `pt`、`begv`、`zv`を記録するマーカーを保持する。

mode_line_format
header_line_format
case_fold_search
tab_width
fill_column
left_margin
auto_fill_function
truncate_lines
word_wrap
ctl_arrow
bidi_display_reordering
bidi_paragraph_direction
selective_display
selective_display_ellipses
overwrite_mode
abbrev_mode
mark_active
enable_multibyte_characters
buffer_file_coding_system
cache_long_line_scans
point_before_scroll
left_fringe_width
right_fringe_width
fringes_outside_margins
scroll_bar_width
indicate_empty_lines
indicate_buffer_boundaries
fringe_indicator_alist
fringe_cursor_alist
scroll_up_aggressively
scroll_down_aggressively
cursor_type
cursor_in_non_selected_windows
: これらのフィールドは自動的にバッファーローカル (Section 12.11 [Buffer-Local Variables], page 190 を参照) になる Lisp 変数の値を格納する。これらに対応する変数は名前のアンダースコアがダッシュで置換される。たとえば mode_line_formatは mode-line-formatの値を格納する。

last_selected_window
: これは最後に選択されていたときにそのバッファーを表示していたウィンドウ、またはそのウィンドウがすでにそのバッファーを表示していなければ nil。

E.9.2 ウィンドウの内部

ウィンドウのフィールドには以下が含まれます (完全なリストは window.hの struct windowを参照):

frame
: そのウィンドウがあるフレーム (Lisp オブジェクト)。

`mini` そのウィンドウがミニバッファーウィンドウ、ミニバッファーかエコーエリアを表示しているウィンドウなら非 0。

`pseudo_window_p`
そのウィンドウが疑似ウィンドウ *(pseudo window)* なら非 0。疑似ウィンドウとはメニューバーかツールバーの表示に使用されているウィンドウ (自身のメニューバーやツールバーを表示しないツールキットを Emacs が使用している場合)、タブバー、またはツールチップフレーム上でツールチップを表示しているウィンドウのいずれか。一般的には疑似ウィンドウは Lisp コードからアクセスできない。

`parent` Emacs は内部的にウィンドウをツリーにアレンジする。ウィンドウの兄弟グループは、そのエリアがすべての兄弟を含むような親ウィンドウをもつ。このフィールドはツリー内でのウィンドウの親を Lisp オブジェクトとして指す。これはツリーのルートウィンドウとミニバッファーウィンドウでは常に `nil`。

親ウィンドウはバッファーを表示せず、子ウィンドウ形成を除いて表示では少ししか役割を果たさない。Emacs Lisp プログラムは親ウィンドウを直接操作できない。Emacs Lisp プログラムでは実際にバッファーを表示するツリーの子ノードのウィンドウにたいして操作を行う。

`contents` リーフウィンドウ (leaf window) およびツールチップを表示中のウィンドウでは、そのウィンドウが表示している Lisp オブジェクトとしてのバッファー、内部ウィンドウ (“親” ウィンドウ) では最初の子ウィンドウ、メニューバーやツールバーを表示している疑似ウィンドウでは `nil`、削除されたウィンドウでも `nil`。

`next`
`prev` そのウィンドウの次の兄弟と前の兄弟 (Lisp オブジェクト)。自身のグループ内でそのウィンドウが右端か下端なら `next`は `nil`。自身のグループ内でそのウィンドウが左端か上端なら `prev`は `nil`。兄弟が左右あるいは上下であるかは兄弟の親の `horizontal` フィールドで判断される。これが非 0 なら兄弟は水平に配置されている。

特別なケースとしてミニバッファーのみのフレームやミニバッファーなしのフレームでなければ、フレームのルートウィンドウの `next`は、そのフレームのミニバッファーウィンドウを指す。そのようなフレームのミニバッファーウィンドウの `prev`は、そのフレームのルートウィンドウを指す。それ以外の場合にはルートウィンドウの `next`フィールド、および (もしあれば) ミニバッファーの `prev`フィールドは `nil`。

`left_col` そのウィンドウの左端をウィンドウのネイティブフレームの最左列 (列 0) から相対的に数えた列数。

`top_line` そのウィンドウの上端をウィンドウのネイティブフレームの最上行 (行 0) から相対的に数えた行数。

`pixel_left`
`pixel_top`
そのウィンドウの左側上端をウィンドウのネイティブフレームの左上隅 (0, 0) から相対的に計測したピクセル数。

`total_cols`
`total_lines`
列数または行数で数えた、そのウィンドウの幅または高さの合計。値にはスクロールバーとフリンジ、ディバイダー、および/または (もしあれば) ウィンドウ右側のセパレーターラインが含まれる。

`pixel_width;`
`pixel_height;`
: ピクセルで計測したウィンドウの幅または高さの合計。

`start`
: そのウィンドウ内に表示されるバッファーで、ウィンドウに最初 (ロジカル順。Section 39.26 [Bidirectional Display], page 1085 を参照) に表示される文字の位置を指すマーカー。

`pointm`
: これはウィンドウが選択されているときのカレントバッファーのポイント値。選択されていなければ前の値が保たれる。

`old_pointm`
: 最後の再表示時の `pointm`の値。

`force_start`
: このフラグが非 `nil`なら Lisp プログラムによりそのウィンドウが明示的にスクロールされたことを示し、再表示のためにウィンドウの `start`の値がセットされる。これはポイントがスクリーン外の場合の次回再表示に影響を与える。影響とはポイント周辺のテキストを表示するためにウィンドウをスクロールするかわりに、スクリーン上にある位置にポイントを移動するというものである。

`optional_new_start`
: これは `force_start`と同様だが、次回表示ではポイントが可視の場合のみしたがう。

`start_at_line_beg`
: 非 `nil`は `start`のカレント値がウィンドウ選択時に先頭行だったことを意味する。

`use_time`
: これはウィンドウが最後に選択された時刻。関数 `get-lru-window`はこの値を使用する。

`sequence_number`
: そのウィンドウ作成時に割り当てられた一意な番号。

`last_modified`
: 前回のそのウィンドウの再表示完了時のウィンドウのバッファーの `modiff`フィールド。

`last_overlay_modified`
: 前回のウィンドウの再表示完了時のウィンドウのバッファーの `overlay_modiff`フィールド。

`last_point`
: 前回のウィンドウの再表示完了時のウィンドウのバッファーのポイント値。

`last_had_star`
: 非 0 値はウィンドウが最後に更新されたとき、そのウィンドウのバッファーが変更されたことを意味する。

`vertical_scroll_bar_type`
`horizontal_scroll_bar_type`
: そのウィンドウの垂直スクロールバーおよび水平スクロールバーのタイプ。

`scroll_bar_width`
`scroll_bar_height`
: そのウィンドウの垂直スクロールバーの幅および水平スクロールバーの高さ (ピクセル単位)。

`left_margin_cols`
`right_margin_cols`
: そのウィンドウの左マージンと右マージンの幅。値 0 はマージンがないことを意味する。

`left_fringe_width`
`right_fringe_width`
: そのウィンドウの左フリンジと右フリンジのピクセル幅。値 −1 はフレームの値の使用を意味する。

`fringes_outside_margins`
: 非 0 値はディスプレーマージン外側のフリンジ、それ以外ならフリンジはマージンとテキストの間にあることを意味する。

`window_end_pos`
: これは `z`から、そのウィンドウのカレントマトリクス内の最後のグリフのバッファー位置を減じて算出される。この値は `window_end_valid`が非 0 のときだけ有効である。

`window_end_bytepos`
: `window_end_pos`に対応するバイト位置。

`window_end_vpos`
: `window_end_pos`を含む行のウィンドウに相対的な垂直位置。

`window_end_valid`
: このフィールドは `window_end_pos`および `window_end_vpos`が真に有効なら非 0 値にセットされる。これは重要な再表示が先に割り込んだ場合には、`window_end_pos`を算出した表示がスクリーン上に出現しなくなるので 0 になる。

`cursor` そのウィンドウ内でカーソルがどこにあるかを記述する構造体。

`last_cursor_vpos`
: 最後の再表示完了時にカーソルを表示していた行の、ウィンドウに相対的な垂直位置。

`phys_cursor`
: そのウィンドウのカーソルが物理的にどこにあるかを記述する構造体。

`phys_cursor_type`
`phys_cursor_height`
`phys_cursor_width`
: そのウィンドウの最後の表示でのカーソルのタイプ、高さ、幅。

`phys_cursor_on_p`
: このフィールドはカーソルが物理的にオンなら非 0。

`cursor_off_p`
: 非 0 はそのウィンドウのカーソルが論理的にオフであることを意味する。これはカーソルの点滅に使用される。

`last_cursor_off_p`
: このフィールドは最後の再表示時の `cursor_off_p`の値を含む。

`must_be_updated_p`
: これはウィンドウを更新しなければならないとき、再表示の間は 1 にセットされる。

`hscroll` これはウィンドウ内の表示が左へ水平スクロールされている列数。通常は 0。カレント行だけが水平スクロールされている際には、カレント行がどれだけ左へ水平スクロールされているかを示す。

`min_hscroll`
`set-window-hscroll`を通じてユーザーがセットする `hscroll`の最小値 (Section 28.23 [Horizontal Scrolling], page 680 を参照)。カレント行だけが水平スクロールされている際には、カレント行以外の行がどれだけ左へ水平スクロールされているかを示す。

`vscroll` ピクセル単位での垂直スクロール量。これは通常は 0。

`dedicated`
そのウィンドウがそれのバッファー専用 (dedicated) なら非 `nil`。

`combination_limit`
このウィンドウの組み合わせ限界は親ウィンドウにとってのみ意味がある。これが `t`ならそのウィンドウの削除は許されず、そのウィンドウの他の兄弟と子ウィンドウを再組み合わせする。

`window_parameters`
そのウィンドウのパラメーターの alist。

`display_table`
そのウィンドウのディスプレーテーブル、何も指定されていなければ `nil`。

`update_mode_line`
非 0 はウィンドウのモードラインの更新が必要なことを意味する。

`mode_line_height`
`header_line_height`
モードラインおよびヘッダーラインのピクセル高さ、不明なら −1。

`base_line_number`
そのバッファーの特定の位置の行番号か 0。これはモードラインでポイントの行番号を表示するために使用される。

`base_line_pos`
行番号が既知であるバッファー位置、不明なら 0。これが −1 なら、そのウィンドウがバッファーを表示するかぎり行番号は表示されない。

`column_number_displayed`
そのウィンドウのモードラインに表示されているカレント列番号、列番号が表示されていなければ −1。

`current_matrix`
`desired_matrix`
そのウィンドウのカレント、および望まれる表示を記述するグリフ。

E.9.3 プロセスの内部

プロセスのフィールドには以下が含まれます (完全なリストは `process.h`の `struct Lisp_Process` の定義を参照):

`name` プロセス名 (Lisp 文字列)。

`command` そのプロセスの開始に使用されたコマンド引数を含むリスト。ネットワークプロセスとシリアルプロセスではプロセスが実行中なら `nil`、停止していたら `t`。

`filter` そのプロセスから出力を受け取るために使用される Lisp 関数。

`sentinel` そのプロセスの状態が変化したら常に呼び出される Lisp 関数。

`buffer` そのプロセスに関連付けられたバッファー。

`pid` オペレーティングシステムのプロセス ID (整数)。ネットワークプロセスやシリアルプロセスのような疑似プロセスでは値 0 を使用する。

`childp` フラグ。実際に子プロセスなら `t`。ネットワークプロセスやシリアルプロセスでは `make-network-process`や `make-serial-process`にもとづく plist。

`mark` そのプロセスの出力からバッファーに挿入された終端位置を示すマーカー。常にではないがこれはバッファー終端であることが多い。

`kill_without_query`
これが非 0 ならプロセス実行中に Emacs を kill してもプロセスの kill にたいして確認を求めない。

`raw_status`
システムコール `wait`がリターンする raw プロセス状態。

`status` `process-status`がリターンするようなプロセス状態。Lisp シンボル、コンスセル、またはリストのいずれか。

`tick`
`update_tick`
これら 2 つのフィールドが等しくないなら、センチネル実行かプロセスバッファーへのメッセージ挿入によりプロセスの状態変更が報告される必要がある。

`pty_flag` そのサブプロセスが pty を使用して対話する場合は非 0。パイプを使用する場合には 0。

`infd` そのプロセスからの入力にたいするファイルディクリプター。

`outfd` そのプロセスへの出力にたいするファイルディクリプター。

`tty_name` そのサブプロセスが使用する端末の名前、パイプを使用する場合には `nil`。

`decode_coding_system`
そのプロセスからの入力のデコーディングにたいするコーディングシステム。

`decoding_buf`
デコーディング用の作業バッファー。

`decoding_carryover`
デコーディングでのキャリーオーバーのサイズ。

`encode_coding_system`
そのプロセスからの出力のエンコーディングにたいするコーディングシステム。

`encoding_buf`
エンコーディング用の作業バッファー。

`inherit_coding_system_flag`
プロセス出力のデコードに使用されるコーディングシステムからプロセスバッファーの `coding-system`をセットするフラグ。

`type` プロセスのタイプを示す `real`、`network`、`serial`のいずれかのシンボル。

E.10 C の整数型

以下は Emacs の C ソースコード内で整数タイプを使用する際のガイドラインです。これらのガイドラインはときに相反するアドバイスを与えることがありますが一般的な常識に沿ったものがアドバイスです。

- 任意の制限の使用を避ける。たとえば `s`の長さを `int`の範囲に収めることが要求されるのでなければ `int len = strlen (s);`を使用しないこと。
- 符号付き整数の算術演算のオーバーフローのラップアラウンドを前提としてはならない。Emacs のポート対象先によっては成立しない。実際には符号付き整数のオーバーフローは未定義であり、コアダンプや早晩に非論理的な振る舞いさえ起こし得る。符号なし整数のオーバーフローは 2 のべき乗の剰余に確実にラップアラウンドされることが保証されている。
- 符号なしタイプと符号付きタイプを組み合わせるとコードが混乱するので符号なしタイプより符号付きタイプを優先すること。他のガイドラインの多くはタイプが符号付きだとみなしている。符号なしタイプを要する稀なケースでは、符号付きの符号なし版 (`ptrdiff_t`のかわりに `size_t`、`intptr_t`のかわりに `uintptr_t`) にたいして同様のアドバイスを適用できる。
- 0 から 0x3FFFFF までの範囲では Emacs 文字コードには `int`を優先すること。より一般的には、たとえばスクリーン列数のように `int`範囲と既知である整数には `int`を優先すること。
- サイズ (たとえばすべての個別の C オブジェクトの最大サイズや、すべての C 配列の最大要素数にバインドされる整数) にたいしては `ptrdiff_t`を優先すること。これは符号付きタイプにたいする Emacs の一般的な優先事項である。`ptrdiff_t`の使用によりオブジェクトは `PTRDIFF_MAX`に制限されるが、より大きいオブジェクトはポインター減算を破壊するかもしれず結局のところ問題を起こす可能性があるので、これは一方的に制限を課すものではない。
- `ssize_t`関連の制限をもつ低レベル API ト対話する際を除いて `ssize_t`を避けること。これは典型的なプラトフォームでは `ptrdiff_t`と等価だとしても、`ssize_t`は範囲が狭いときがあり使用によりサイズ関連の計算がオーバーフローするかもしれない。同じく `ptrdiff_t`はより一般的で標準化されており、標準的な `printf`フォーマットをもち、Emacs の内部的なサイズオーバーフローのチェックの基礎である。`ssize_t`を使用する際には POSIX ガ – から `SSIZE_MAX`の範囲の値にたいするサポートだけを要求することに注意してほしい。
- 通常はポインターの内部表現や与えられた任意のタイミングで存在可能なオブジェクト数や割り当て可能な総バイト数にのみバインドされる整数には `intptr_t`を優先すること。しかしページ境界を横切る可能性のあるポインター演算を表す場合には `uintptr_t`を優先すること。たとえば 32 ビットのアドレス空間をもつマシンでの配列は 0x7fffffff/0x80000000 境界を横断する可能性があり、`(intptr_t) 0x7fffffff`に 1 を加算することによって整数のオーバーフローが発生し得る。
- Emacs Lisp の fixnum への変換や逆変換を表す値では fixnum 演算が `EMACS_INT`にもとづくので Emacs で定義されたタイプ `EMACS_INT`を優先すること。
- (ファイルサイズやエポック以降の経過秒数等の) システム値を表す際には、(`off_t`や `time_t`等の) システムタイプを優先すること。安全だと解っていなければシステムタイプが符号付きだと仮定してはならない。たとえば `off_t`は常に符号付きだが `time_t`は符号付きである必要はない。
- 符号付き整数かもしれない値を表す場合には `intmax_t`を優先すること。`printf`族の関数は `"%"PRIdMAX`のようなフォーマットを使用してこのような値をプリントできる。
- ブーリーンには `bool`、`false`、`true`を使用すること。`bool`の使用によりプログラムの可読性が増して、`int`を使用するより若干高速になる。`int`、`0`、`1`を使用しても大丈夫だが旧スタイルは段階的に廃止される。`bool`を使用する際にはソースファイル `lib/stdbool.in.h`に文書化されている `bool`の代替実装の制限を尊重すること。特にブーリーンのビットフィールドは `bool`では

なく `bool_bf` タイプであること。そうすれば標準の GCC で Objective C をコンパイルするときでさえ正しく機能する。

- ビットフィールドでは `int` は可搬性に劣るので、`int` より `unsigned int` か `signed int` を優先すること。単一ビットのビットフィールドの値は 0 か 1 なので `unsigned int` か `bool_bf` を使用すること。

Appendix F 標準的なエラー

以下は標準的な Emacs における、より重要なエラーシンボルを概念別にグループ分けしたリストです。このリストには各シンボルのメッセージ、およびエラーを発生し得る方法へのクロスリファレンスが含まれています。

これらのエラーシンボルはそれぞれ、親となるエラーコンディションのセットをシンボルのリストとして保持します。通常このリストにはエラーシンボル自身とシンボル `error`が含まれます。このリストは `error`より狭義ですが、単一のエラーシンボルより広義であるような中間的なクラス分けのための追加シンボルを含む場合があります。たとえばファイルアクセスでのすべてのエラーはコンディション `file-error`をもちます。ここでわたしたちが、特定のエラーシンボルにたいする追加エラーコンディションに言及していなければ、それがないことを意味しています。

特別な例外としてエラーシンボル `quit`は、quit はエラーとみなされないのでコンディション `error`をもっていません。

これらのエラーシンボルのほとんどは C(主に `data.c`) で定義されていますが、いくつかは Lisp で定義されています。たとえばファイル `userlock.el`では `file-locked`と `file-supersession`のエラーが定義されています。Emacs とともに配布される専門的な Lisp ライブラリーのいくつかは、それら自身のエラーシンボルを定義しています。それらのすべてをここではリストしません。

エラーの発生とそれを処理する方法については Section 11.7.3 [Errors], page 164 を参照してください。

`error` メッセージは '`error`'。Section 11.7.3 [Errors], page 164 を参照のこと。

`quit` メッセージは '`Quit`'。Section 21.11 [Quitting], page 413 を参照のこと。

`args-out-of-range`
: メッセージは '`Args out of range`'。これはシーケンス、バッファー、その他コンテナー類似オブジェクトにたいして範囲を超えた要素にアクセスを試みたときに発生する。Chapter 6 [Sequences Arrays Vectors], page 95 と Chapter 32 [Text], page 777 を参照のこと。

`arith-error`
: メッセージは '`Arithmetic error`'。これは 0 による整数除算を試みたときに発生する。Section 3.5 [Numeric Conversions], page 42 と Section 3.6 [Arithmetic Operations], page 43 を参照のこと。

`beginning-of-buffer`
: メッセージは '`Beginning of buffer`'。Section 30.2.1 [Character Motion], page 756 を参照のこと。

`buffer-read-only`
: メッセージは '`Buffer is read-only`'。Section 27.7 [Read Only Buffers], page 599 を参照のこと。

`circular-list`
: メッセージは '`List contains a loop`'。これは循環構造に遭遇時に発生する。Section 2.6 [Circular Objects], page 29 を参照のこと。

`cl-assertion-failed`
: メッセージは '`Assertion failed`'。これは `cl-assert`マクロのテスト失敗時に発生する。Section "Assertions" in *Common Lisp Extensions* を参照のこと。

`coding-system-error`
: メッセージは '`Invalid coding system`'。Section 33.10.3 [Lisp and Coding Systems], page 867 を参照のこと。

`cyclic-function-indirection`
: メッセージは '`Symbol's chain of function indirections contains a loop`'。See Section 10.1.4 [Function Indirection], page 135 を参照のこと。

`cyclic-variable-indirection`
: メッセージは '`Symbol's chain of variable indirections contains a loop`'。See Section 12.15 [Variable Aliases], page 204 を参照のこと。

`dbus-error`
: メッセージは '`D-Bus error`'。Section "Errors and Events" in *D-Bus integration in Emacs* を参照のこと。

`end-of-buffer`
: メッセージは '`End of buffer`'。Section 30.2.1 [Character Motion], page 756 を参照のこと。

`end-of-file`
: メッセージは '`End of file during parsing`'。これはファイル I/O ではなく Lisp リーダーに属するので `file-error`のサブカテゴリーではないことに注意のこと。Section 19.3 [Input Functions], page 334 を参照のこと。

`file-already-exists`
: これは `file-error`のサブカテゴリー。Section 25.4 [Writing to Files], page 538 を参照のこと。

`file-date-error`
: これは `file-error`のサブカテゴリー。これは `copy-file`を試行して出力ファイルの最終変更時刻のセットに失敗したときに発生する。Section 25.7 [Changing Files], page 550 を参照のこと。

`file-error`
: このエラーメッセージは、通常はエラーコンディション `file-error`が与えられたときはデータアイテムだけから構築されるので、エラー文字列とサブカテゴリーはここにリストしない。つまりエラー文字列は特に関連しない。しかしこれらのエラーシンボルは `error-message`プロパティをもち、何もデータが与えられなければ `error-message`が使用される。Chapter 25 [Files], page 530 を参照のこと。

`file-missing`
: これは `file-error`のサブカテゴリー。これは存在しないファイルの処理を試みた際に発生する。Section 25.7 [Changing Files], page 550 を参照のこと。

`compression-error`
: これは圧縮ファイルの処理の問題を起因とする `file-error`のサブカテゴリー。Section 16.1 [How Programs Do Loading], page 269 を参照のこと。

`file-locked`
: これは `file-error`のサブカテゴリー。Section 25.5 [File Locks], page 539 を参照のこと。

`file-supersession`
: これは`file-error`のサブカテゴリー。Section 27.6 [Modification Time], page 597 を参照のこと。

`file-notify-error`
: これは`file-error`のサブカテゴリー。Section 40.20 [File Notifications], page 1127 を参照のこと。

`ftp-error`
: これは ftp を使用したリモートファイルへのアクセスの問題を起因とする`file-error`のサブカテゴリー。Section “Remote Files” in *The GNU Emacs Manual* を参照のこと。

`invalid-function`
: メッセージは‘`Invalid function`’。Section 10.1.4 [Function Indirection], page 135 を参照のこと。

`invalid-read-syntax`
: メッセージは通常は‘`Invalid read syntax`’。Section 2.1 [Printed Representation], page 8 を参照のこと。このエラーは後の式が続くようなテキストがある際の、`eval-expression`のようなコマンドでも raise され得る。この場合にはメッセージは‘`Trailing garbage following expression`’。

`invalid-regexp`
: メッセージは‘`Invalid regexp`’。Section 34.3 [Regular Expressions], page 883 を参照のこと。

`mark-inactive`
: メッセージは‘`The mark is not active now`’。Section 31.7 [The Mark], page 772 を参照のこと。

`no-catch`
: メッセージは‘`No catch for tag`’。Section 11.7.1 [Catch and Throw], page 162 を参照のこと。

`range-error`
: メッセージは`Arithmetic range error`。

`overflow-error`
: メッセージは‘`Arithmetic overflow error`’。これは`range-error`のサブカテゴリー。これはリミット`integer-width`を超過する整数で発生し得る。Section 3.1 [Integer Basics], page 37 を参照のこと。

`scan-error`
: メッセージは‘`Scan error`’。これは特定の構文解析関数が無効な構文やマッチしないカッコを見つけたときに発生する。慣習的に人間が可読なエラーメッセージ、移動を妨害する開始位置、妨害の終了位置という 3 つの引数で raise される。Section 30.2.6 [List Motion], page 762 と Section 35.6 [Parsing Expressions], page 914 を参照のこと。

`search-failed`
: メッセージは‘`Search failed`’。Chapter 34 [Searching and Matching], page 881 を参照のこと。

`setting-constant`
: メッセージは‘`Attempt to set a constant symbol`’。これは`nil`、`t`、`most-positive-fixnum`、`most-negative-fixnum`およびキーワードシンボルへの

値の割り当て時に発生する。これは `enable-multibyte-characters`や何らかの理由により値の直接割り当てが許されていないインボルへの値の割り当て時にも発生する。Section 12.2 [Constant Variables], page 174 を参照のこと。

`text-read-only`
: メッセージは‘`Text is read-only`’。これは `buffer-read-only`のサブカテゴリー。Section 32.19.4 [Special Properties], page 820 を参照のこと。

`undefined-color`
: メッセージは‘`Undefined color`’。Section 29.22 [Color Names], page 748 を参照のこと。

`user-error`
: メッセージは空文字列。Section 11.7.3.1 [Signaling Errors], page 165 を参照のこと。

`user-search-failed`
: これは‘`search-failed`’と似ているが、‘`user-error`’のようにデバッガーをトリガーしない。Section 11.7.3.1 [Signaling Errors], page 165 と Chapter 34 [Searching and Matching], page 881 を参照のこと。これは Info ファイル内での検索に使用される。Section “Search Text” in *Info* を参照のこと。

`void-function`
: メッセージは‘`Symbol's function definition is void`’。Section 13.9 [Function Cells], page 225 を参照のこと。

`void-variable`
: メッセージは‘`Symbol's value as variable is void`’。Section 12.7 [Accessing Variables], page 182 を参照のこと。

`wrong-number-of-arguments`
: メッセージは‘`Wrong number of arguments`’。Section 10.1.3 [Classifying Lists], page 135 を参照のこと。

`wrong-type-argument`
: メッセージは‘`Wrong type argument`’。Section 2.7 [Type Predicates], page 29 を参照のこと。

`unknown-image-type`
: メッセージは‘`Cannot determine image type`’. See Section 39.17 [Images], page 1049。

Appendix G 標準的なキーマップ

このセクションでは、より一般的なキーマップをリストします。これらの多くは Emacs の初回起動時に存在しますが、それらのいくつかは各機能へのアクセス時にロードされます。

他にもより特化された多くのキーマップがあります。それらは特にメジャーモードやマイナーモードに関連付けられています。ミニバッファーはいくつかのキーマップを使用します (Section 20.6.3 [Completion Commands], page 355 を参照)。キーマップの詳細については Chapter 22 [Keymaps], page 422 を参照してください。

`2C-mode-map`
: プレフィクス *C-x 6*のサブコマンドにたいする sparse キーマップ。
Section “Two-Column Editing” in *The GNU Emacs Manual* を参照のこと。

`abbrev-map`
: プレフィクス *C-x a*のサブコマンドにたいする sparse キーマップ。
Section “Defining Abbrevs” in *The GNU Emacs Manual* を参照のこと。

`button-buffer-map`
: バッファーを含むバッファーに有用な sparse キーマップ。
これを親キーマップとして使用したいと思うかもしれない。Section 39.19 [Buttons], page 1068 を参照のこと。

`button-map`
: ボタンにより使用される sparse キーマップ。

`ctl-x-4-map`
: プレフィックス *C-x 4*のサブコマンドの sparse キーマップ。

`ctl-x-5-map`
: プレフィックス *C-x 5*のサブコマンドの sparse キーマップ。

`ctl-x-map`
: *C-x*コマンドにたいする完全なキーマップ。

`ctl-x-r-map`
: プレフィクス *C-x r*のサブコマンドにたいする sparse キーマップ。
Section “Registers” in *The GNU Emacs Manual* を参照のこと。

`esc-map` `ESC`(または `Meta`) コマンドにたいする完全なキーマップ。

`facemenu-keymap`
: プレフィクスキー *M-o*にたいして使用される sparse キーマップ。

`function-key-map`
: すべての `local-function-key-map`のインスタンスの親キーマップ (`local-function-key-map`を参照)。

`global-map`
: デフォルトのグローバルキーバインディングを含む完全なキーマップ。
モードでこのグローバルマップを変更しないこと。

`goto-map` プレフィクスキー *M-g*にたいして使用される sparse キーマップ。

`help-map` ヘルプ文字 *C-h*に後続するキーにたいする sparse キーマップ。
Section 24.6 [Help Functions], page 526 を参照のこと。

`Helper-help-map`
: ヘルプユーティリティパッケージにより使用される完全なキーマップ。
これは値セルと関数セルに同じキーマップをもつ。

`input-decode-map`
: キーパッドとファンクションキーの変換にたいするキーマップ。
存在しなければ空の sparse キーマップを含む。Section 22.14 [Translation Keymaps], page 442 を参照のこと。

`key-translation-map`
: キー変換にたいするキーマップ。`local-function-key-map`と異なり通常のキーバインディングをオーバーライドする。Section 22.14 [Translation Keymaps], page 442 を参照のこと。

`kmacro-keymap`
: プレフィクス検索 *C-x C-k*に後続するキーにたいする sparse キーマップ。
Section “Keyboard Macros” in *The GNU Emacs Manual* を参照のこと。

`local-function-key-map`
: キーシーケンスを優先する代替えに変換するキーマップ。
存在しなければ空の sparse キーマップが含まれる。Section 22.14 [Translation Keymaps], page 442 を参照のこと。

`menu-bar-file-menu`
`menu-bar-edit-menu`
`menu-bar-options-menu`
`global-buffers-menu-map`
`menu-bar-tools-menu`
`menu-bar-help-menu`
: これらのキーマップはメニューバー内のメインとなるトップレベルメニューを表示する。
これらのいくつかはサブメニューを含む。たとえばEdit メニューは`menu-bar-search-menu`を含む等。Section 22.17.5 [Menu Bar], page 453 を参照のこと。

`minibuffer-inactive-mode-map`
: ミニバッファーが非アクティブ時に使用される完全なキーマップ。
Section “Editing in the Minibuffer” in *The GNU Emacs Manual* を参照のこと。

`mode-line-coding-system-map`
`mode-line-input-method-map`
`mode-line-column-line-number-mode-map`
: これらのキーマップはモードライン内の種々のエリアを制御する。
Section 23.4 [Mode Line Format], page 485 を参照のこと。

`mode-specific-map`
: *C-c*に後続する文字にたいするキーマップ。これはグローバルキーマップ内にあることに注意。これは実際にはモード固有のものではない。プフィクスキー *C-c*の使用方法を主に記述する *C-h b* (`display-bindings`) 内で有益なのでこの名前が選ばれた。

`mouse-appearance-menu-map`
: *S-mouse-1*キーにたいして使用される sparse キーマップ。

`mule-keymap`
: プレフィクスキー *C-x RET*にたいして使用されるグローバルキーマップ。

`narrow-map`
: プレフィクス `C-x n`のサブコマンドにたいする sparse キーマップ。

`prog-mode-map`
: Prog モードにより使用されるキーマップ。
Section 23.2.5 [Basic Major Modes], page 472 を参照のこと。

`query-replace-map`
`multi-query-replace-map`
: `query-replace`での応答と関連するコマンド、`y-or-n-p`と `map-y-or-n-p`にたいしても使用される sparse キーマップ。このマップを使用する関数はプレフィクスキーを使用せず一度に 1 つのイベントを照会する。複数バッファーの置換では `multi-query-replace-map`が `query-replace-map`を拡張する。Section 34.7 [Search and Replace], page 902 を参照のこと。

`search-map`
: 検索関連コマンドにたいしてグローバルバインディングを提供する sparse キーマップ。

`special-mode-map`
: Special モードにより使用されるキーマップ。
Section 23.2.5 [Basic Major Modes], page 472 を参照のこと。

`tab-prefix-map`
: タブバー関連コマンド用のプレフィックスキー `C-x t`にたいして使用されるグローバルキーマップ。
Section “Tab Bars” in *The GNU Emacs Manual* を参照のこと。

`tab-bar-map`
: タブバーのコンテンツを定義するキーマップ。
Section “Tab Bars” in *The GNU Emacs Manual* を参照のこと。

`tool-bar-map`
: ツールバーのコンテンツを定義するキーマップ。
Section 22.17.6 [Tool Bar], page 455 を参照のこと。

`universal-argument-map`
: `C-u`処理中に使用される sparse キーマップ。
Section 21.12 [Prefix Command Arguments], page 415 を参照のこと。

`vc-prefix-map`
: プレフィクスキー `C-x v`にたいして使用されるグローバルキーマップ。

`x-alternatives-map`
: グラフィカルなフレームでの特定キーのマップに使用される sparse キーマップ。
関数 `x-setup-function-keys`はこれを使用する。

Appendix H 標準的なフック

以下は Emacs で適切なタイミングで呼び出す関数を提供するためのいくつかのフック関数のリストです。

これらの変数のほとんどは '`-hook`'で終わる名前をもちます。これらはノーマルフック *(normal hooks)* と呼ばれており `run-hooks`により実行されます。そのようなフックの値は関数のリストです。これらの関数は引数なしで呼び出されて値は完全に無視されます。そのようなフック上に新たに関数を配置するためには `add-hook`を呼び出す方法を推奨します。フック使用についての詳細は Section 23.1 [Hooks], page 461 を参照してください。

'`-functions`'で終わる名前の変数は通常はアブノーマルフック *(abnormal hooks)* です (古いコードの中には非推奨の '`-hooks`'サフィクスを使用するものもある)。これらの値は関数のリストですが関数は特殊な方法で呼び出されます (引数を渡されたりリターン値が使用される)。'`-function`'で終わる名前の変数は値として単一の関数をもちます。

以下のリストはすべてを網羅したリストではなく、より一般的なフックだけをカバーしています。たとえばメジャーモードはそれぞれ '*`modename`*`-mode-hook`'という名前のフックを定義します。メジャーモードは自身が行う最後のこととして `run-mode-hooks`でこのノーマルフックを実行します。Section 23.2.6 [Mode Hooks], page 473 を参照してください。ほとんどのマイナーモードにもフックがあります。

特別な機能によりファイルがロードされたときに評価する式を指定できます (Section 16.10 [Hooks for Loading], page 283 を参照)。この機能は正確にはフックではありませんが同様のことを行います。

`activate-mark-hook`
`deactivate-mark-hook`
: Section 31.7 [The Mark], page 772 を参照のこと。

`after-change-functions`
`before-change-functions`
`first-change-hook`
: Section 32.32 [Change Hooks], page 850 を参照のこと。

`after-change-major-mode-hook`
`change-major-mode-after-body-hook`
: Section 23.2.6 [Mode Hooks], page 473 を参照のこと。

`after-init-hook`
`before-init-hook`
`emacs-startup-hook`
`window-setup-hook`
: Section 40.1.2 [Init File], page 1093 を参照のこと。

`after-insert-file-functions`
`write-region-annotate-functions`
`write-region-post-annotation-function`
: Section 25.13 [Format Conversion], page 573 を参照のこと。

`after-make-frame-functions`
`before-make-frame-hook`
`server-after-make-frame-hook`
: Section 29.1 [Creating Frames], page 697 を参照のこと。

`after-save-hook`
`before-save-hook`
`write-contents-functions`
`write-file-functions`
: Section 25.2 [Saving Buffers], page 534 を参照のこと。

`after-setting-font-hook`
: フレームのフォント変更後に実行されるフック。

`auto-save-hook`
: See Section 26.2 [Auto-Saving], page 583 を参照のこと。

`before-hack-local-variables-hook`
`hack-local-variables-hook`
: Section 12.12 [File Local Variables], page 197 を参照のこと。

`buffer-access-fontify-functions`
: Section 32.19.7 [Lazy Properties], page 828 を参照のこと。

`buffer-list-update-hook`
: バッファーリスト変更時に実行されるフック (Section 27.8 [Buffer List], page 600 を参照)。

`buffer-quit-function`
: カレントバッファーを quit するために呼び出されるフック。

`change-major-mode-hook`
: Section 12.11.2 [Creating Buffer-Local], page 192 を参照のこと。

`comint-password-function`
: これは派生モードにたいして、ユーザーにプロンプト表示せずに、背後にあるコマンドインタープリター用にパスワードを供給することを許可するアブノーマルフック。

`command-line-functions`
: Section 40.1.4 [Command-Line Arguments], page 1096 を参照のこと。

`delayed-warnings-hook`
: コマンドループは `post-command-hook`(以下参照) の直後にこれを実行する。

`focus-in-hook`
`focus-out-hook`
: Section 29.10 [Input Focus], page 732 を参照のこと。

`delete-frame-functions`
`after-delete-frame-functions`
: Section 29.7 [Deleting Frames], page 730 を参照のこと。

`delete-terminal-functions`
: Section 29.2 [Multiple Terminals], page 698 を参照のこと。

`pop-up-frame-function`
`split-window-preferred-function`
: Section 28.13.4 [Choosing Window Options], page 649 を参照のこと。

`echo-area-clear-hook`
: Section 39.4.4 [Echo Area Customization], page 988 を参照のこと。

`find-file-hook`
`find-file-not-found-functions`
: Section 25.1.1 [Visiting Functions], page 530 を参照のこと。

`font-lock-extend-after-change-region-function`
: Section 23.6.9.2 [Region to Refontify], page 508 を参照のこと。

`font-lock-extend-region-functions`
: Section 23.6.9 [Multiline Font Lock], page 506 を参照のこと。

`font-lock-fontify-buffer-function`
`font-lock-fontify-region-function`
`font-lock-mark-block-function`
`font-lock-unfontify-buffer-function`
`font-lock-unfontify-region-function`
: Section 23.6.4 [Other Font Lock Variables], page 502 を参照のこと。

`fontification-functions`
: Section 39.12.7 [Automatic Face Assignment], page 1024 を参照のこと。

`frame-auto-hide-function`
: Section 28.16 [Quitting Windows], page 662 を参照のこと。

`quit-window-hook`
: Section 28.16 [Quitting Windows], page 662 を参照のこと。

`kill-buffer-hook`
`kill-buffer-query-functions`
: Section 27.10 [Killing Buffers], page 603 を参照のこと。

`kill-emacs-hook`
`kill-emacs-query-functions`
: Section 40.2.1 [Killing Emacs], page 1097 を参照のこと。

`menu-bar-update-hook`
: Section 22.17.5 [Menu Bar], page 453 を参照のこと。

`minibuffer-setup-hook`
`minibuffer-exit-hook`
: Section 20.14 [Minibuffer Misc], page 373 を参照のこと。

`mouse-leave-buffer-hook`
: マウスコマンドでのウィンドウ切り替え時に実行されるフック。

`mouse-position-function`
: Section 29.16 [Mouse Position], page 743 を参照のこと。

`prefix-command-echo-keystrokes-functions`
: (*C-u*のような) プレフィクスコマンドにより実行されるアブノーマルフックであり、カレントのプレフィクス状態を記述する文字列をリターンすること。たとえば *C-u*は 'C-u-' や 'C-u 1 2 3-'を生成する。フック関数はそれぞれ引数なしで呼び出されてカレントのプレフィクス状態を記述する文字列、プレフィクス状態がなければ `nil`をリターンすること。Section 21.12 [Prefix Command Arguments], page 415 を参照のこと。

`prefix-command-preserve-state-hook`
: プレフィクスコマンドが次のコマンドにカレントのプレフィクスコマンドを渡すことによりプレフィクスを確保する必要はある際にフックが実行される。たとえば `C-u`はユーザーが `C-u` -や `C-u`の後に数字をタイプした際には、その状態を次のコマンドに渡す必要がある。

`pre-redisplay-functions`
: フックはそれぞれのウィンドウで再表示の直前に実行される。Section 39.2 [Forcing Redisplay], page 981 を参照のこと。

`post-command-hook`
`pre-command-hook`
: Section 21.1 [Command Overview], page 375 を参照のこと。

`post-gc-hook`
: Section E.3 [Garbage Collection], page 1178 を参照のこと。

`post-self-insert-hook`
: Section 23.3.2 [Keymaps and Minor Modes], page 482 を参照のこと。

`suspend-hook`
`suspend-resume-hook`
`suspend-tty-functions`
`resume-tty-functions`
: Section 40.2.2 [Suspending Emacs], page 1098 を参照のこと。

`syntax-begin-function`
`syntax-propertize-extend-region-functions`
`syntax-propertize-function`
`font-lock-syntactic-face-function`
: Section 23.6.8 [Syntactic Font Lock], page 505 と Section 35.4 [Syntax Properties], page 912 を参照のこと。

`temp-buffer-setup-hook`
`temp-buffer-show-function`
`temp-buffer-show-hook`
: Section 39.8 [Temporary Displays], page 996 を参照のこと。

`tty-setup-hook`
: Section 40.1.3 [Terminal-Specific], page 1095 を参照のこと。

`window-configuration-change-hook`
`window-scroll-functions`
`window-size-change-functions`
: Section 28.28 [Window Hooks], page 691 を参照のこと。

Index

既

既存の関数の再定義。 216

"

‘"’ in printing 337
‘"’ in strings 19

#

‘##’ read syntax 14
‘#$’ 288
‘#'’ syntax 221
‘#(’ read syntax 21
‘#:’ read syntax 14
‘#^’ read syntax 21
‘#@*count*’ 288
‘#n#’ read syntax 29
‘#n=’ read syntax 29

$

‘$’ in display 982
‘$’ in regexp 886

%

% 45
‘%’ in format 62

&

‘&’ in replacement 898
&optional 212
&rest 212

'

‘'’ for quoting 139

(

‘(’ in regexp 889
‘(...)’ in lists 15
‘(?:’ in regexp 890

)

‘)’ in regexp 889

*

* 44
‘*’ in interactive 377
‘*’ in regexp 884
*get_user_ptr 1199
scratch 469

+

+ 44
‘+’ in regexp 885

,

, (with backquote) 139
,@ (with backquote) 140

—

- 44
--disable-build-details
  option to configure 1176
--enable-locallisppath option
  to configure 1176
--temacs option, and dumping method 1175

.

‘.’ in lists 17
‘.’ in regexp 884
.#, lock file names 539
.emacs 1093

/

/ 44
/= 41
/dev/tty 974

:

:deferred, JSONRPC keyword 848
:notification-dispatcher 846
:request-dispatcher 846

;

‘;’ for commenting 10

<

< 41
<= 41

=

= 41

>

> 41
>= 41

?

‘?’ in character constant 11
? in minibuffer 346
‘?’ in regexp 885

[

‘[’ in regexp 885
[...] (Edebug) 322

]

‘]’ in regexp 885

^

‘^’ in interactive 377
‘^’ in regexp 886

`

` 139
‘ (list substitution) 139

@

‘@’ in interactive 377

\

‘\’ in character constant 12
‘\’ in display 982
‘\’ in printing 337
‘\’ in regexp 887
‘\’ in replacement 898
‘\’ in strings 19
‘\’ in symbols 14
‘\'’ in regexp 891
\(in strings 763
‘\<’ in regexp 891
‘\=’ in regexp 891
‘\>’ in regexp 891
‘_<’ in regexp 891
‘_>’ in regexp 891
‘\`’ in regexp 891
‘\a’ 11
‘\b’ 11
‘\b’ in regexp 891
‘\B’ in regexp 891
‘\e’ 11
‘\f’ 11
‘\n’ 11
‘\n’ in print 339
‘\n’ in replacement 898
‘\r’ 11
‘\s’ 11
‘\s’ in regexp 891
‘\S’ in regexp 891
‘\t’ 11
‘\v’ 11
‘\w’ in regexp 890
‘\W’ in regexp 891

|

‘|’ in regexp 889

1

1+ 43
1- 43
1value 329

2

2C-mode-map 428
2D box 1013

3

3D box 1013

A

abbrev ... 922
abbrev properties ... 927
abbrev table properties ... 928
abbrev tables ... 922
abbrev tables in modes ... 466
`abbrev-all-caps` ... 926
`abbrev-expand-function` ... 926
`abbrev-expansion` ... 925
`abbrev-file-name` ... 924
`abbrev-get` ... 927
`abbrev-insert` ... 925
`abbrev-map` ... 1220
`abbrev-minor-mode-table-alist` ... 927
`abbrev-prefix-mark` ... 925
`abbrev-put` ... 927
`abbrev-start-location` ... 926
`abbrev-start-location-buffer` ... 926
`abbrev-symbol` ... 925
`abbrev-table-get` ... 928
`abbrev-table-name-list` ... 923
`abbrev-table-p` ... 922
`abbrev-table-put` ... 928
`abbreviate-file-name` ... 559
abbreviated file names ... 559
abbrevs, looking up and expanding ... 925
`abbrevs-changed` ... 924
abnormal hook ... 461
`abort-recursive-edit` ... 418
aborting ... 417
`abs` ... 42
absolute edges ... 705
absolute file name ... 557
absolute frame edges ... 705
absolute frame position ... 705
absolute position ... 705
accept input from processes ... 957
`accept-change-group` ... 849
`accept-process-output` ... 958
access control list ... 548
access minibuffer contents ... 372
`access-file` ... 542
accessibility of a file ... 541
accessible portion (of a buffer) ... 766
`accessible-keymaps` ... 445
accessing documentation strings ... 520
accessing hash tables ... 120
accessing plist properties ... 93
ACL entries ... 548
`acos` ... 49
`action` (button property) ... 1068
action alist for buffer display ... 645
action function, for buffer display ... 641
`action`, customization keyword ... 264
`activate-change-group` ... 849
`activate-mark-hook` ... 775
active display table ... 1080
active keymap ... 429
active keymap, controlling ... 432
`active-minibuffer-window` ... 371
`ad-activate` ... 233
`adaptive-fill-first-line-regexp` ... 802
`adaptive-fill-function` ... 802
`adaptive-fill-mode` ... 801
`adaptive-fill-regexp` ... 802
`add-face-text-property` ... 816
`add-function` ... 228
`add-hook` ... 462
`add-name-to-file` ... 550
`add-text-properties` ... 815
`add-to-history` ... 348
`add-to-invisibility-spec` ... 993
`add-to-list` ... 79
`add-to-ordered-list` ... 80
`add-variable-watcher` ... 184
address field of register ... 15
`adjust-window-trailing-edge` ... 618
adjusting point ... 387
advertised binding ... 523
advice, add and remove ... 228
`advice-add` ... 231
`advice-eval-interactive-spec` ... 229
`advice-function-mapc` ... 229
`advice-function-member-p` ... 229
`advice-mapc` ... 231
`advice-member-p` ... 231
`advice-remove` ... 231
advices, porting from `defadvice` ... 233
advising functions ... 227
advising named functions ... 230
AEAD cipher ... 839
`after-change-functions` ... 850
`after-change-major-mode-hook` ... 474
`after-delete-frame-functions` ... 730
`after-find-file` ... 533
`after-focus-change-function` ... 735
`after-init-hook` ... 1094
`after-init-time` ... 1091
`after-insert-file-functions` ... 577
`after-load-functions` ... 283
`after-make-frame-functions` ... 697
`after-revert-hook` ... 588
`after-save-hook` ... 536
`after-setting-font-hook` ... 1224
`after-string` (overlay property) ... 1005
alias, for coding systems ... 866
alias, for faces ... 1023
alias, for functions ... 216
alias, for variables ... 204
alist ... 88
alist vs. plist ... 93
`alist-get` ... 90
`all-completions` ... 352
`all-threads` ... 930
`allow-no-window`, a buffer display action alist entry ... 648

alnum character class, regexp ... 887
alpha character class, regexp ... 887
`alpha`, a frame parameter ... 727
alt characters ... 13
alternatives, defining ... 382
amalgamating commands, and undo ... 795
`amalgamating-undo-limit` ... 795
`and` ... 148
animation ... 1065
anonymous face ... 1011
anonymous function ... 220
apostrophe for quoting ... 139
`append` ... 76
`append-to-file` ... 538
`apply` ... 217
`apply`, and debugging ... 305
`apply-partially` ... 218
applying customizations ... 266
`apropos` ... 526
archive web server ... 1139
`aref` ... 108
`args`, customization keyword ... 263
argument ... 209
argument binding ... 212
argument lists, features ... 212
arguments for shell commands ... 936
arguments, interactive entry ... 376
arguments, reading ... 342
`argv` ... 1097
`arith-error` example ... 169
`arith-error` in division ... 45
arithmetic operations ... 43
arithmetic shift ... 46
array ... 107
array elements ... 108
`arrayp` ... 107
ascii character class, regexp ... 887
`ascii-case-table` ... 70
ASCII character codes ... 11
ASCII control characters ... 1078
`aset` ... 108
`ash` ... 46
`asin` ... 49
`ask-user-about-lock` ... 540
`ask-user-about-supersession-threat` ... 598
asking the user questions ... 366
`assoc` ... 89
`assoc-default` ... 90
`assoc-delete-all` ... 92
`assoc-string` ... 59
association list ... 88
`assq` ... 89
`assq-delete-all` ... 91
asynchronous subprocess ... 941
`atan` ... 49
`atom` ... 72
atomic changes ... 848
atomic windows ... 669
atoms ... 15
attributes of text ... 813
Auto Fill mode ... 802
`auto-coding-alist` ... 872
`auto-coding-functions` ... 873
`auto-coding-regexp-alist` ... 872
`auto-fill-chars` ... 802
`auto-fill-function` ... 802
`auto-hide-function`, a frame parameter ... 722
`auto-hscroll-mode` ... 680
`auto-lower`, a frame parameter ... 724
`auto-mode-alist` ... 469
`auto-raise`, a frame parameter ... 723
`auto-raise-tool-bar-buttons` ... 457
`auto-resize-tool-bars` ... 457
`auto-save-default` ... 585
`auto-save-file-name-p` ... 583
`auto-save-hook` ... 585
`auto-save-interval` ... 585
`auto-save-list-file-name` ... 586
`auto-save-list-file-prefix` ... 586
`auto-save-mode` ... 583
`auto-save-timeout` ... 585
`auto-save-visited-file-name` ... 584
auto-selection of window ... 685
`auto-window-vscroll` ... 679
autoload ... 274, 275
autoload by prefix ... 278
autoload cookie ... 276
autoload cookie, and safe values of variable ... 198
autoload errors ... 276
autoload object ... 210
autoload, when to use ... 278
`autoload-compute-prefixes` ... 278
`autoload-do-load` ... 277
`autoloadp` ... 276
automatic face assignment ... 1024
automatically buffer-local ... 191

B

`back-to-indentation` ... 811
`background-color`, a frame parameter ... 728
`background-mode`, a frame parameter ... 727
backing store ... 753
backquote (list substitution) ... 139
backquote-style patterns ... 157
backslash in character constants ... 12
backslash in regular expressions ... 889
backslash in strings ... 19
backslash in symbols ... 14
backspace ... 11
`backtrace` ... 304
backtrace buffer ... 301
backtrace from emacsclient's `--eval` ... 297
backtrace of thread ... 933
`backtrace-debug` ... 305
`backtrace-frame` ... 305

backtracking ... 323
backtracking and POSIX regular expressions ... 897
backtracking and regular expressions ... 884
backup file ... 578
backup files, rename or copy ... 580
backup-buffer ... 578
backup-by-copying ... 580
backup-by-copying-when-linked ... 580
backup-by-copying-when-mismatch ... 580
backup-by-copying-when-privileged-mismatch ... 581
backup-directory-alist ... 579
backup-enable-predicate ... 579
backup-file-name-p ... 582
backup-inhibited ... 579
backups and auto-saving ... 578
backward-button ... 1072
backward-char ... 756
backward-delete-char-untabify ... 785
backward-delete-char-untabify-method ... 785
backward-list ... 762
backward-prefix-chars ... 913
backward-sexp ... 763
backward-to-indentation ... 811
backward-up-list ... 763
backward-word ... 757
backward-word-strictly ... 758
balance-windows ... 621
balance-windows-area ... 621
balanced parenthesis motion ... 762
balancing parentheses ... 1077
balancing window sizes ... 621
barf-if-buffer-read-only ... 600
base 64 encoding ... 837
base buffer ... 605
base coding system ... 865
base direction of a paragraph ... 1086
base for reading an integer ... 37
base location, package archive ... 1138
base remapping, faces ... 1022
base64-decode-region ... 838
base64-decode-string ... 838
base64-encode-region ... 837
base64-encode-string ... 837
base64url-encode-region ... 837
base64url-encode-string ... 838
basic code (of input character) ... 388
basic faces ... 1024
batch mode ... 1121
batch-byte-compile ... 287
baud, in serial connections ... 976
baud-rate ... 1119
beep ... 1083
before point, insertion ... 781
before-change-functions ... 850
before-hack-local-variables-hook ... 198
before-init-hook ... 1094
before-init-time ... 1090
before-make-frame-hook ... 697
before-revert-hook ... 587
before-save-hook ... 536
before-string (overlay property) ... 1005
beginning of line ... 759
beginning of line in regexp ... 886
beginning-of-buffer ... 758
beginning-of-defun ... 763
beginning-of-defun-function ... 764
beginning-of-line ... 758
bell ... 1083
bell character ... 11
benchmark.el ... 330
benchmarking ... 330
best practices ... 1162
bidi-display-reordering ... 1085
bidi-find-overridden-directionality ... 1088
bidi-paragraph-direction ... 1087
bidi-paragraph-separate-re ... 1086
bidi-paragraph-start-re ... 1086
bidi-string-mark-left-to-right ... 1088
bidirectional class of characters ... 858
bidirectional display ... 1085
bidirectional reordering ... 1085
big endian ... 977
bignum range ... 38
bignump ... 40
binary coding system ... 865
binary I/O in batch mode ... 334
bindat-get-field ... 979
bindat-ip-to-string ... 980
bindat-length ... 980
bindat-pack ... 980
bindat-unpack ... 979
binding arguments ... 212
binding local variables ... 175
binding of a key ... 423
bitmap-spec-p ... 1015
bitmaps, fringe ... 1037
bitwise arithmetic ... 46
blink-cursor-alist ... 726
blink-matching-delay ... 1077
blink-matching-open ... 1077
blink-matching-paren ... 1077
blink-matching-paren-distance ... 1077
blink-paren-function ... 1077
blinking parentheses ... 1077
bobp ... 778
body height of a window ... 615
body of a window ... 613
body of function ... 212
body size of a window ... 616
body width of a window ... 615
bolp ... 778
bool-vector ... 113
bool-vector length ... 95
bool-vector-count-consecutive ... 113
bool-vector-count-population ... 114

`bool-vector-exclusive-or` ... 113
`bool-vector-intersection` ... 113
`bool-vector-not` ... 113
`bool-vector-p` ... 113
`bool-vector-set-difference` ... 113
`bool-vector-subsetp` ... 113
`bool-vector-union` ... 113
Bool-vectors ... 113
boolean ... 2
`booleanp` ... 3
bootstrapping Emacs ... 1175
`border-color`, a frame parameter ... 728
`border-width`, a frame parameter ... 719
bottom dividers ... 1042
`bottom-divider-width`, a frame parameter ... 720
`bottom-visible`, a frame parameter ... 723
`boundp` ... 178
box diagrams, for lists ... 16
break ... 296
breakpoints (Edebug) ... 312
bucket (in obarray) ... 126
buffer ... 590
buffer boundaries, indicating ... 1035
buffer contents ... 777
buffer display ... 639
buffer display action alist ... 645
buffer display action function ... 641
buffer display action functions, precedence ... 651
buffer display conventions ... 655
buffer display display action ... 640
buffer file name ... 594
buffer gap ... 606
buffer input stream ... 331
buffer internals ... 1203
buffer list ... 600
buffer modification ... 596
buffer names ... 592
buffer output stream ... 335
buffer portion as string ... 778
buffer position ... 755
buffer text notation ... 4
buffer, read-only ... 599
`buffer-access-fontified-property` ... 828
`buffer-access-fontify-functions` ... 828
`buffer-auto-save-file-format` ... 576
`buffer-auto-save-file-name` ... 583
`buffer-backed-up` ... 578
`buffer-base-buffer` ... 606
`buffer-chars-modified-tick` ... 597
`buffer-disable-undo` ... 796
`buffer-display-count` ... 636
`buffer-display-table` ... 1081
`buffer-display-time` ... 636
`buffer-enable-undo` ... 796
`buffer-end` ... 756
`buffer-file-coding-system` ... 866
`buffer-file-format` ... 575
`buffer-file-name` ... 594
`buffer-file-number` ... 595
`buffer-file-truename` ... 594
`buffer-hash` ... 839
`buffer-invisibility-spec` ... 993
`buffer-list` ... 600
`buffer-list`, a frame parameter ... 721
`buffer-list-update-hook` ... 602, 1224
`buffer-live-p` ... 605
buffer-local variables ... 190
buffer-local variables in modes ... 466
`buffer-local-value` ... 193
`buffer-local-variables` ... 193
`buffer-modified-p` ... 596
`buffer-modified-tick` ... 597
`buffer-name` ... 593
`buffer-name-history` ... 349
`buffer-narrowed-p` ... 766
`buffer-offer-save` ... 604
`buffer-predicate`, a frame parameter ... 721
`buffer-quit-function` ... 1224
`buffer-read-only` ... 599
`buffer-save-without-query` ... 605
`buffer-saved-size` ... 586
`buffer-size` ... 756
`buffer-stale-function` ... 588
`buffer-string` ... 779
`buffer-substring` ... 778
`buffer-substring-filters` ... 780
`buffer-substring-no-properties` ... 779
`buffer-substring-with-bidi-context` ... 1089
`buffer-swap-text` ... 606
`buffer-undo-list` ... 794
`bufferp` ... 590
`bufferpos-to-filepos` ... 853
buffers to display on frame ... 721
buffers without undo information ... 592
buffers, controlled in windows ... 635
buffers, creating ... 602
buffers, killing ... 603
bugs ... 1
bugs in this manual ... 1
build details ... 1176
building Emacs ... 1175
building lists ... 75
built-in function ... 209
`bury-buffer` ... 601
`butlast` ... 75
`button` (button property) ... 1069
button buffer commands ... 1071
button properties ... 1068
button types ... 1069
`button-activate` ... 1071
`button-at` ... 1071
button-down event ... 393
`button-end` ... 1070
`button-face`, customization keyword ... 264
`button-get` ... 1070
`button-has-type-p` ... 1071

`button-label` ... 1071
`button-prefix`, customization keyword ... 264
`button-put` ... 1070
`button-start` ... 1070
`button-suffix`, customization keyword ... 264
`button-type` ... 1071
`button-type-get` ... 1071
`button-type-put` ... 1071
`button-type-subtype-p` ... 1071
buttons in buffers ... 1068
byte compilation ... 285
byte compiler warnings, how to avoid ... 1167
byte packing and unpacking ... 977
byte to string ... 855
`byte-boolean-vars` ... 205, 1186
byte-code ... 285
byte-code function ... 291
byte-code object ... 291
`byte-code-function-p` ... 211
`byte-compile` ... 286
`byte-compile-debug` ... 285
`byte-compile-dynamic` ... 289
`byte-compile-dynamic-docstrings` ... 288
`byte-compile-error-on-warn` ... 291
`byte-compile-file` ... 287
byte-compiler errors ... 290
byte-compiler warnings ... 290
byte-compiling macros ... 244
byte-compiling `require` ... 280
`byte-recompile-directory` ... 287
`byte-to-position` ... 853
`byte-to-string` ... 855
bytes ... 51
bytesize, in serial connections ... 976

C

C programming language ... 1184
C-c ... 428
C-g ... 413
C-h ... 428
`C-M-x` ... 307
C-x ... 428
C-x 4 ... 428
C-x 5 ... 428
C-x 6 ... 428
`C-x C-a C-m` ... 309
C-x RET ... 428
C-x t ... 428
C-x v ... 428
`C-x X =` ... 317
`caaaar` ... 75
`caaadr` ... 75
`caaar` ... 75
`caadar` ... 75
`caaddr` ... 75
`caadr` ... 75
`caar` ... 75
`cadaar` ... 75
`cadadr` ... 75
`cadar` ... 75
`caddar` ... 75
`cadddr` ... 75
`caddr` ... 75
`cadr` ... 75
calendrical computations ... 1113
calendrical information ... 1106
call stack ... 304
`call-interactively` ... 382
`call-process` ... 937
`call-process`, command-line arguments from minibuffer ... 936
`call-process-region` ... 940
`call-process-shell-command` ... 940
`call-shell-region` ... 940
`called-interactively-p` ... 384
calling a function ... 217
`cancel-change-group` ... 849
`cancel-debug-on-entry` ... 299
`cancel-debug-on-variable-change` ... 299
`cancel-timer` ... 1116
canonical character height ... 707
canonical character width ... 707
capitalization ... 68
`capitalize` ... 68
`capitalize-region` ... 811
`capitalize-word` ... 812
caption bar ... 703
`car` ... 73
`car-safe` ... 73
case conversion in buffers ... 811
case conversion in Lisp ... 67
case in replacements ... 897
`case-fold-search` ... 883
`case-replace` ... 883
`case-table-p` ... 69
`catch` ... 163
categories of characters ... 918
`category` (overlay property) ... 1004
`category` (text property) ... 820
category set ... 918
category table ... 918
category, regexp search for ... 891
`category-docstring` ... 919
`category-set-mnemonics` ... 920
`category-table` ... 920
`category-table-p` ... 920
`cdaaar` ... 75
`cdaadr` ... 75
`cdaar` ... 75
`cdadar` ... 75
`cdaddr` ... 75
`cdadr` ... 75
`cdar` ... 75
`cddaar` ... 75
`cddadr` ... 75

`cddar` ... 75
`cdddar` ... 75
`cddddr` ... 75
`cdddr` ... 75
`cddr` ... 75
`cdr` ... 73
`cdr-safe` ... 73
`ceiling` ... 43
centering point ... 678
change hooks ... 850
change hooks for a character ... 824
change `load-path` at configure time ... 1176
`change-major-mode-after-body-hook` ... 474
`change-major-mode-hook` ... 195
changing key bindings ... 438
changing text properties ... 814
changing to another buffer ... 590
changing window size ... 617
`char-after` ... 777
`char-before` ... 777
`char-category-set` ... 920
`char-charset` ... 862
`char-code-property-description` ... 860
`char-displayable-p` ... 1029
`char-equal` ... 56
`char-from-name` ... 857
`char-or-string-p` ... 52
`char-property-alias-alist` ... 814
`char-script-table` ... 861
`char-syntax` ... 911
char-table length ... 95
`char-table-extra-slot` ... 111
`char-table-p` ... 111
`char-table-parent` ... 111
`char-table-range` ... 112
`char-table-subtype` ... 111
char-tables ... 110
`char-to-string` ... 61
`char-width` ... 1007
`char-width-table` ... 861
character alternative (in regexp) ... 885
character arrays ... 51
character case ... 67
character categories ... 918
character classes in regexp ... 887
character code conversion ... 865
character codepoint ... 852
character codes ... 856
character event ... 388
character insertion ... 783
character printing ... 525
character properties ... 857
character set, searching ... 863
character sets ... 861
character to string ... 61
character translation tables ... 863
character width on display ... 1007
`characterp` ... 856
characters ... 51
characters for interactive codes ... 378
characters, multi-byte ... 852
characters, representation in buffers and strings ... 852
charset ... 861
charset, coding systems to encode ... 869
`charset`, text property ... 877
`charset-after` ... 863
`charset-list` ... 861
`charset-plist` ... 862
`charset-priority-list` ... 861
`charsetp` ... 861
charsets supported by a coding system ... 870
`check-coding-system` ... 868
`check-coding-systems-region` ... 869
`checkdoc` ... 1162
`checkdoc-current-buffer` ... 1162
`checkdoc-file` ... 1162
`checkdoc-minor-mode` ... 1168
`checkdoc-package-keywords` ... 1173
`checkdoc-package-keywords-flag` ... 1173
child frames ... 739
child process ... 934
child window ... 610
`child-frame-parameters`, a buffer display action alist entry ... 648
choice, customization types ... 263
cipher, AEAD ... 839
cipher, symmetric ... 839
circular list ... 71
circular structure, read syntax ... 29
`cl` ... 2
`cl-call-next-method` ... 225
`cl-defgeneric` ... 222
`cl-defmethod` ... 223
`cl-next-method-p` ... 225
`cl-old-struct-compat-mode` ... 117
CL note—allocate more storage ... 1179
CL note—case of letters ... 14
CL note—default optional arg ... 213
CL note—interning existing symbol ... 127
CL note—lack `union`, `intersection` ... 85
CL note—no continuable errors ... 166
CL note—no `setf` functions ... 208
CL note—only `throw` in Emacs ... 163
CL note—`rplaca` vs `setcar` ... 81
CL note—special forms compared ... 138
CL note—symbol in obarrays ... 127
classification of file types ... 542
classifying events ... 398
cleanup forms ... 172
`clear-abbrev-table` ... 922
`clear-image-cache` ... 1066
`clear-message-function` ... 984
`clear-string` ... 56
`clear-this-command-keys` ... 386
`clear-visited-file-modtime` ... 598

click event ... 390
clickable buttons in buffers ... 1068
clickable text ... 828
clipboard ... 747
clipboard support (for MS-Windows) ... 747
clone-indirect-buffer ... 605
clone-of, a window parameter ... 690
closepath ... 1058
closure ... 227
closures, example of using ... 188
CLOS ... 222
clrhash ... 120
coded character set ... 861
codepoint, largest value ... 856
codes, interactive, description of ... 378
codespace ... 852
coding conventions in Emacs Lisp ... 1162
coding standards ... 1162
coding system ... 864
coding system for operation ... 874
coding system, automatically determined ... 871
coding system, validity check ... 868
coding systems for encoding a string ... 868
coding systems for encoding region ... 868
coding systems, priority ... 875
coding-system-aliases ... 866
coding-system-change-eol-conversion ... 868
coding-system-change-text-conversion ... 868
coding-system-charset-list ... 870
coding-system-eol-type ... 868
coding-system-for-read ... 874
coding-system-for-write ... 875
coding-system-get ... 866
coding-system-list ... 867
coding-system-p ... 868
coding-system-priority-list ... 875
coding-system-require-warning ... 875
collapse-delayed-warnings ... 992
color names ... 748
color-defined-p ... 748
color-gray-p ... 749
color-supported-p ... 749
color-values ... 749
colors on text terminals ... 749
column width ... 707
columns ... 806
COM1 ... 974
combine-after-change-calls ... 850
combine-and-quote-strings ... 937
combine-change-calls ... 851
combining conditions ... 148
command ... 210
command descriptions ... 4
command history ... 419
command in keymap ... 435
command loop ... 375
command loop variables ... 385
command loop, recursive ... 417
command-debug-status ... 305
command-error-function ... 167
command-execute ... 383
command-history ... 419
command-line ... 1096
command-line arguments ... 1096
command-line options ... 1096
command-line-args ... 1096
command-line-args-left ... 1097
command-line-functions ... 1097
command-line-processed ... 1096
command-remapping ... 441
command-switch-alist ... 1096
commandp ... 382
commandp example ... 358
commands, defining ... 376
comment style ... 910
comment syntax ... 909
comment-auto-fill-only-comments ... 803
comment-end-can-be-escaped ... 917
commentary, in a Lisp library ... 1174
comments ... 10
comments, Lisp convention for ... 1171
Common Lisp ... 2
compare-buffer-substrings ... 781
compare-strings ... 59
compare-window-configurations ... 687
comparing buffer text ... 781
comparing file modification time ... 597
comparing numbers ... 40
comparing time values ... 1113
compatibility, between modules and Emacs ... 1189
compilation (Emacs Lisp) ... 285
compilation functions ... 285
compile-defun ... 286
compile-time constant ... 290
compiled function ... 291
compiler errors ... 290
complete key ... 423
completing-read ... 353
completing-read-function ... 355
completion ... 350
completion styles ... 362
completion table ... 350
completion table, modifying ... 353
completion tables, combining ... 353
completion, file name ... 563
completion-at-point ... 365
completion-at-point-functions ... 365
completion-auto-help ... 356
completion-boundaries ... 352
completion-category-overrides ... 362
completion-extra-properties ... 363
completion-ignore-case ... 353
completion-ignored-extensions ... 564
completion-in-region ... 366
completion-regexp-list ... 353
completion-styles ... 362

`completion-styles-alist` ... 362
`completion-table-case-fold` ... 353
`completion-table-dynamic` ... 364
`completion-table-in-turn` ... 353
`completion-table-merge` ... 353
`completion-table-subvert` ... 353
`completion-table-with-cache` ... 365
`completion-table-with-predicate` ... 353
`completion-table-with-quoting` ... 353
`completion-table-with-terminator` ... 353
complex arguments ... 342
complex command ... 419
composite types (customization) ... 258
`composition` (text property) ... 825
`composition` property, and point display ... 387
`compute-motion` ... 761
`concat` ... 53
concatenating bidirectional strings ... 1087
concatenating lists ... 84
concatenating strings ... 53
concurrency ... 929
`cond` ... 147
condition name ... 171
`condition-case` ... 168
`condition-case-unless-debug` ... 168
`condition-mutex` ... 932
`condition-name` ... 932
`condition-notify` ... 932
`condition-variable-p` ... 932
`condition-wait` ... 932
conditional evaluation ... 146
conditional selection of windows ... 635
`confirm-kill-processes` ... 961
connection local variables ... 201
`connection-local-criteria-alist` ... 203
`connection-local-profile-alist` ... 202
`connection-local-set-profile-variables` ... 201
`connection-local-set-profiles` ... 202
`cons` ... 75
cons cells ... 75
`cons-cells-consed` ... 1183
consing ... 76
`consp` ... 72
constant variables ... 174, 180
`constrain-to-field` ... 832
content directory, package ... 1134
continuation lines ... 982
`continue-process` ... 952
control character key constants ... 438
control character printing ... 525
control characters ... 13
control characters in display ... 1079
control characters, reading ... 410
control structures ... 145
control-ja.texi.po ... 179
`Control-X-prefix` ... 428
controller part, model/view/controller ... 1076
controlling terminal ... 1098
`controlling-tty-p` ... 1099
conventions for writing major modes ... 464
conventions for writing minor modes ... 480
conversion of strings ... 60
convert buffer position to file byte ... 853
convert file byte to buffer position ... 853
convert sequence to another type ... 105
`convert-standard-filename` ... 565
converting file names from/to
MS-Windows syntax ... 555
converting numbers ... 42
coordinate, relative to frame ... 681
`coordinates-in-window-p` ... 682
`copy-abbrev-table` ... 922
`copy-alist` ... 91
`copy-category-table` ... 920
`copy-directory` ... 568
`copy-file` ... 551
`copy-hash-table` ... 122
`copy-keymap` ... 426
`copy-marker` ... 770
`copy-overlay` ... 1001
`copy-region-as-kill` ... 789
`copy-sequence` ... 96
`copy-syntax-table` ... 911
`copy-tree` ... 78
`copy_string_contents` ... 1194
copying alists ... 91
copying bidirectional text,
preserve visual order ... 1089
copying files ... 550
copying lists ... 76
copying sequences ... 96
copying strings ... 53
copying vectors ... 110
`copysign` ... 39
`cos` ... 49
`count-lines` ... 759
`count-loop` ... 5
`count-screen-lines` ... 761
`count-words` ... 759
counting columns ... 806
counting set bits ... 49
coverage testing ... 329
coverage testing (Edebug) ... 317
create subprocess ... 934
`create-file-buffer` ... 533
`create-fontset-from-fontset-spec` ... 1027
`create-image` ... 1061
`create-lockfiles` ... 540
creating buffers ... 602
creating hash tables ... 118
creating keymaps ... 425
creating markers ... 769
creating strings ... 52
creating, copying and deleting directories ... 568
cryptographic hash ... 838, 839
`ctl-arrow` ... 1079

ctl-x-4-map ... 428
ctl-x-5-map ... 428
ctl-x-map ... 428
ctl-x-r-map ... 1220
curly quotes ... 523, 524, 1169
curly quotes, in formatted messages ... 61
current binding ... 175
current buffer ... 590
current buffer mark ... 773
current buffer point and mark (Edebug) ... 319
current buffer position ... 755
current command ... 385
current stack frame ... 301
current-active-maps ... 430
current-bidi-paragraph-direction ... 1087
current-buffer ... 590
current-case-table ... 70
current-column ... 806
current-fill-column ... 800
current-frame-configuration ... 739
current-global-map ... 432
current-idle-time ... 1117
current-indentation ... 807
current-input-method ... 878
current-input-mode ... 1118
current-justification ... 799
current-kill ... 791
current-left-margin ... 800
current-local-map ... 432
current-message ... 985
current-minor-mode-maps ... 432
current-prefix-arg ... 416
current-thread ... 930
current-time ... 1105
current-time-string ... 1105
current-time-zone ... 1106
current-window-configuration ... 686
current-word ... 780
currying ... 218
cursor ... 671
cursor (text property) ... 823
cursor position for display properties and overlays ... 823
cursor, and frame parameters ... 725
cursor, fringe ... 1036
cursor-color, a frame parameter ... 728
cursor-in-echo-area ... 988
cursor-in-non-selected-windows ... 726
cursor-intangible (text property) ... 823
cursor-intangible-mode ... 823
cursor-sensor-functions (text property) ... 825
cursor-sensor-inhibit ... 823
cursor-sensor-mode ... 825
cursor-type ... 726
cursor-type, a frame parameter ... 725
curved quotes ... 523, 524, 1169
curved quotes, in formatted messages ... 61
curveto ... 1059
cust-print ... 316
custom '%'-sequence in format ... 65
custom format string ... 65
custom themes ... 267
custom-add-frequent-value ... 256
custom-group property ... 253
custom-initialize-delay ... 1177
custom-known-themes ... 268
custom-reevaluate-setting ... 256
custom-set-faces ... 266
custom-set-variables ... 266
custom-theme-p ... 268
custom-theme-set-faces ... 267
custom-theme-set-variables ... 267
custom-unlispify-remove-prefixes ... 253
custom-variable-history ... 349
custom-variable-p ... 257
customizable variables, how to define ... 253
customization groups, defining ... 252
customization item ... 250
customization keywords ... 250
customization types ... 257
customization types, define new ... 265
customize-package-emacs-version-alist ... 252
cyclic ordering of windows ... 633
cygwin-convert-file-name-from-windows ... 555
cygwin-convert-file-name-to-windows ... 555

D

data type ... 8
data-directory ... 528
datagrams ... 967
date-days-in-month ... 1114
date-leap-year-p ... 1114
date-ordinal-to-time ... 1114
date-to-time ... 1109
deactivate-mark ... 774, 775
deactivate-mark-hook ... 775
debug ... 303
debug-ignored-errors ... 297
debug-on-entry ... 298
debug-on-error ... 296
debug-on-error use ... 167
debug-on-event ... 298
debug-on-message ... 298
debug-on-next-call ... 305
debug-on-quit ... 298
debug-on-signal ... 297
debug-on-variable-change ... 299
debugger ... 304
debugger command list ... 301
debugger for Emacs Lisp ... 296
debugger, explicit entry ... 300
debugger-bury-or-kill ... 300
debugger-stack-frame-as-list ... 305
debugging changes to variables ... 299
debugging errors ... 296

debugging invalid Lisp syntax 328
debugging lisp programs 296
debugging specific functions 298
declare 237
declare-function 238, 239
declaring functions 238
decode process output 957
decode-char 862
decode-coding-inserted-region 877
decode-coding-region 877
decode-coding-string 877
decode-time 1107
decoded-time-add 1108
decoding file formats 573
decoding in coding systems 876
decrement field of register 15
dedicated window 661
dedicated, a buffer display action alist entry .. 648
def-edebug-spec 320
defadvice 233
defalias 216
defalias-fset-function property 216
default argument string 378
default character height 707
default character size 707
default character width 707
default coding system 871
default coding system,
  functions to determine 873
default filter function of a process 955
default font 707
default height of character 707
default init file 1093
default key binding 424
default sentinel function of a process 959
default value 195
default value of char-table 111
default width of character 707
default-boundp 195
default-directory 560
default-file-modes 553
default-font-height 1034
default-font-width 1033
default-frame-alist 714
default-input-method 878
default-justification 798
default-minibuffer-frame 732
default-process-coding-system 873
default-text-properties 814
default-toplevel-value 196
default-value 195
default.el 1091
defconst 180
defcustom 253
deferred evaluation 143
defface 1015
defgroup 253
defimage 1061
define customization group 252
define customization options 253
define hash comparisons 121
define image 1061
define new customization types 265
define-abbrev 923
define-abbrev-table 923
define-advice 231
define-alternatives 382
define-button-type 1069
define-category 919
define-derived-mode 470
define-error 171
define-fringe-bitmap 1038
define-generic-mode 477
define-globalized-minor-mode 484
define-hash-table-test 121
define-inline 236
define-key 438
define-key-after 458
define-minor-mode 482
define-obsolete-face-alias 1024
define-obsolete-function-alias 235
define-obsolete-variable-alias 204
define-package 1137
define-prefix-command 429
defined-colors 749
defining a function 215
defining abbrevs 923
defining commands 376
defining customization variables in C 1187
defining faces 1015
defining Lisp variables in C 1186
defining macros 244
defining menus 447
defining tokens, SMIE 512
defining-kbd-macro 420
definition-prefixes 278
definitions of symbols 125
defmacro 244
defsubr, Lisp symbol for a primitive 1186
defsubst 235
deftheme 267
defun 215
defun-prompt-regexp 763
DEFUN, C macro to define Lisp primitives 1184
defvar 179
defvar-local 193
DEFVAR_INT, DEFVAR_LISP,
  DEFVAR_BOOL, DEFSYM 1186
defvaralias 204
delay-mode-hooks 474
delay-warning 992
delayed warnings 991
delayed-warnings-hook 992, 1224
delayed-warnings-list 992
delete 87
delete-and-extract-region 785

`delete-auto-save-file-if-necessary` 585
`delete-auto-save-files` 585
`delete-backward-char` 785
`delete-before`, a frame parameter 722
`delete-blank-lines` 787
`delete-by-moving-to-trash` 552, 568
`delete-char` 785
`delete-directory` 568
`delete-dups` 88
`delete-exited-processes` 946
`delete-field` 832
`delete-file` 552
`delete-frame` 730
`delete-frame` event 395
`delete-frame-functions` 730
`delete-horizontal-space` 786
`delete-indentation` 786
`delete-minibuffer-contents` 373
`delete-old-versions` 581
`delete-other-frames` 731
`delete-other-windows` 625
`delete-other-windows`, a window parameter .. 689
`delete-overlay` 1000
`delete-process` 946
`delete-region` 784
`delete-selection`, symbol property 776
`delete-selection-helper` 776
`delete-selection-pre-hook` 776
`delete-terminal` 698
`delete-terminal-functions` 699
`delete-to-left-margin` 801
`delete-trailing-whitespace` 787
`delete-window` 625
`delete-window`, a window parameter 689
`delete-windows-on` 625
deleting files 550
deleting frames 730
deleting list elements 85
deleting previous char 785
deleting processes 946
deleting text vs killing 784
deleting whitespace 786
deleting windows 625
`delq` 85
dependencies 1134
derived mode 470
`derived-mode-p` 472
describe characters and events 525
`describe-bindings` 447
`describe-buffer-case-table` 70
`describe-categories` 921
`describe-current-display-table` 1080
`describe-display-table` 1080
`describe-mode` 470
`describe-prefix-bindings` 527
`describe-syntax` 912
description for interactive codes 378
description format 4
deserializing 977
desktop notifications 1123
desktop save mode 518
`desktop-buffer-mode-handlers` 518
`desktop-save-buffer` 518
`destroy-fringe-bitmap` 1038
destructive list operations 81
destructuring with pcase patterns 158
`detect-coding-region` 869
`detect-coding-string` 869
deterministic build 1176
diagrams, boxed, for lists 16
dialog boxes 745
`digit-argument` 416
`ding` 1083
`dir-locals-class-alist` 201
`dir-locals-directory-cache` 201
`dir-locals-file` 200
`dir-locals-set-class-variables` 201
`dir-locals-set-directory-class` 201
`direction`, a buffer display action alist entry .. 648
directional overrides 1088
directory file name 558
directory local variables 200
directory name 558
directory part (of file name) 555
`directory-abbrev-alist` 559
`directory-file-name` 558
`directory-files` 566
`directory-files-and-attributes` 567
`directory-files-recursively` 566
`directory-name-p` 558
directory-oriented functions 565
`dired-kept-versions` 582
`disable-command` 419
`disable-point-adjustment` 387
`disable-theme` 268
`disabled` 418
disabled command 418
`disabled-command-function` 419
disabling multibyte 854
disabling undo 796
`disassemble` 293
disassembled byte-code 292
`discard-input` 412
discarding input 412
dispatch of methods for generic function 225
`display` (overlay property) 1004
`display` (text property) 1043
display action 640
display area 705
display feature testing 751
display margins 1047
display message in echo area 983
display name on X 699
display origin 705
display properties, and bidi
  reordering of text 1086

`display` property, and point display 387
display property, unsafe evaluation 1043
display specification 1043
display table 1079
`display`, a frame parameter 715
display, abstract 1072
display, arbitrary objects 1072
`display-backing-store` 753
`display-buffer` 640
`display-buffer-alist` 641
`display-buffer-at-bottom` 644
`display-buffer-base-action` 641
`display-buffer-below-selected` 644
`display-buffer-fallback-action` 641
`display-buffer-in-atom-window` 670
`display-buffer-in-child-frame` 644
`display-buffer-in-direction` 643
`display-buffer-in-previous-window` 642
`display-buffer-in-side-window` 665
`display-buffer-no-window` 645
`display-buffer-overriding-action` 641
`display-buffer-pop-up-frame` 644
`display-buffer-pop-up-window` 642
`display-buffer-reuse-mode-window` 642
`display-buffer-reuse-window` 642
`display-buffer-same-window` 642
`display-buffer-use-some-frame` 645
`display-buffer-use-some-window` 643
`display-color-cells` 753
`display-color-p` 752
`display-completion-list` 356
`display-delayed-warnings` 992
`display-graphic-p` 751
`display-grayscale-p` 752
`display-images-p` 752
`display-message-or-buffer` 985
`display-mm-dimensions-alist` 753
`display-mm-height` 753
`display-mm-width` 753
`display-monitor-attributes-list` 700
`display-mouse-p` 752
`display-pixel-height` 752
`display-pixel-width` 752
`display-planes` 753
`display-popup-menus-p` 751
`display-save-under` 753
`display-screens` 752
`display-selections-p` 752
display-start position 671
`display-supports-face-attributes-p` 752
`display-table-slot` 1080
`display-type`, a frame parameter 715
`display-visual-class` 753
`display-warning` 990
displaying a buffer 639
displaying faces 1021
displays, multiple 698
distance between strings 59
distinguish interactive calls 384
`dnd-protocol-alist` 748
`do-auto-save` 585
`doc`, customization keyword 264
`doc-directory` 522
`DOC` (documentation) file 519
Document Object Model 842
`documentation` 520
documentation conventions 519
documentation for major mode 470
documentation notation 3
documentation string of function 213
documentation strings 519
documentation strings,
  conventions and tips 1168
documentation, keys in 522
`documentation-property` 520
`dolist` 160
`dolist-with-progress-reporter` 987
`dom-node` 843
DOM 842
`dotimes` 160
`dotimes-with-progress-reporter` 987
dotted list 71
dotted lists (Edebug) 323
dotted pair notation 17
double-click events 393
`double-click-fuzz` 394
`double-click-time` 394
double-quote in strings 19
`down-list` 763
`downcase` 67
`downcase-region` 812
`downcase-word` 812
downcasing in `lookup-key` 405
drag and drop 748
drag event 392
`drag-internal-border`, a frame parameter 723
`drag-n-drop` event 396
`drag-with-header-line`, a frame parameter 723
`drag-with-mode-line`, a frame parameter 723
dribble file 1119
dump file 1175
`dump-emacs` 1177
`dump-emacs-portable` 1177
dumping Emacs 1175
dynamic binding 185
dynamic extent 185
dynamic libraries 1130
dynamic loading of documentation 288
dynamic loading of functions 288
dynamic modules 284
dynamic modules, writing 1188
dynamic scope 185
`dynamic-library-alist` 1130

E

eager macro expansion 270
early init file 1093
early-init.el 1093
easy-menu-define 458
easy-mmode-define-minor-mode 484
echo area 983
echo area customization 988
echo-area-clear-hook 988
echo-keystrokes 989
edebug 313
Edebug debugging facility 306
Edebug execution modes 308
Edebug specification list 321
edebug, failure to instrument 308
edebug-after-instrumentation-function 327
edebug-all-defs 325
edebug-all-forms 325
edebug-backtrace-hide-instrumentation 311
edebug-backtrace-show-instrumentation 311
edebug-behavior-alist 327
edebug-continue-kbd-macro 326
edebug-defun 308
edebug-display-freq-count 317
edebug-eval-macro-args 321, 325
edebug-eval-top-level-form 308
edebug-global-break-condition 327
edebug-initial-mode 326
edebug-max-depth 318
edebug-new-definition-function 327
edebug-on-error 327
edebug-on-quit 327
edebug-print-circle 316, 326
edebug-print-length 316, 326
edebug-print-level 316, 326
edebug-print-trace-after 317
edebug-print-trace-before 317
edebug-remove-instrumentation 308
edebug-save-displayed-buffer-points 325
edebug-save-windows 325
edebug-set-global-break-condition 313
edebug-set-initial-mode 309
edebug-setup-hook 325
edebug-sit-for-seconds 310, 327
edebug-sit-on-break 327
edebug-temp-display-freq-count 317
edebug-test-coverage 326
edebug-trace 317, 326
edebug-tracing 317
edebug-unwrap-results 326
edge detection, images 1051
edit distance between strings 59
edit-and-eval-command 347
editing types 25
editor command loop 375
eight-bit, a charset 861
electric-future-map 6
element (of list) 71
elements of sequences 96
elliptical-arc 1060
elp.el 330
elt 96
Emacs event standard notation 525
Emacs process run time 1112
emacs, a charset 861
emacs-build-number 6
emacs-build-time 6
emacs-init-time 1113
emacs-internal coding system 865
emacs-lisp-docstring-fill-column 1168
emacs-major-version 6
emacs-minor-version 6
emacs-pid 1103
emacs-repository-branch 7
emacs-repository-version 7
emacs-save-session-functions 1122
emacs-session-restore 1122
emacs-startup-hook 1094
emacs-uptime 1113
emacs-version 6
emacs_funcall_exit 1201, 1202
emacs_funcall_exit, enumeration 1201
emacs_funcall_exit_return 1201
emacs_funcall_exit_signal 1201
emacs_funcall_exit_throw 1202
emacs_module_init 284, 1189
emacs_process_input_result 1201
emacs_value data type 1193
emacs_variadic_function 1191
EMACS_LIMB_MAX 1194
emacsclient, getting a backtrace 297
EMACSLOADPATH environment variable 273
embedded widgets 1066
empty lines, indicating 1035
empty list 16
empty overlay 1000
empty region 776
emulation-mode-map-alists 434
enable-command 419
enable-connection-local-variables 203
enable-dir-local-variables 201
enable-local-eval 199
enable-local-variables 197
enable-multibyte-characters 852, 854
enable-recursive-minibuffers 373
enable-theme 268
encapsulation, ewoc 1072
encode-char 862
encode-coding-region 876
encode-coding-string 876
encode-time 1109
encoding file formats 573
encoding in coding systems 876
encrypted network connections 965
end of line in regexp 886
end-of-buffer 758

`end-of-defun` ... 763
`end-of-defun-function` ... 764
`end-of-file` ... 334
`end-of-line` ... 759
end-of-line conversion ... 865
endianness ... 977
environment ... 133
environment variable access ... 1101
environment variables, subprocesses ... 935
`eobp` ... 778
eol conversion of coding system ... 868
eol type of coding system ... 868
EOL conversion ... 865
`eolp` ... 778
epoch ... 1104
`eq` ... 33, 1199
`eql` ... 41
`equal` ... 34
`equal-including-properties` ... 35
equality ... 33
`erase-buffer` ... 784
`error` ... 165
error cleanup ... 172
error debugging ... 296
error description ... 169
error display ... 983
error handler ... 167
`error` in debug ... 303
error message notation ... 3
error name ... 171
error symbol ... 171
`error-conditions` ... 171
`error-message-string` ... 169
errors ... 164
`esc-map` ... 428
`ESC-prefix` ... 428
escape (ASCII character) ... 11
escape characters ... 339
escape characters in printing ... 337
escape sequence ... 12
`ESC` ... 437
`eval` ... 141
eval during compilation ... 289
`eval`, and debugging ... 305
`eval-and-compile` ... 289
`eval-buffer` ... 141
`eval-buffer` (Edebug) ... 307
`eval-current-buffer` ... 142
`eval-current-buffer` (Edebug) ... 307
`eval-defun` (Edebug) ... 307
`eval-defun`, and `defcustom` forms ... 254
`eval-defun`, and `defface` forms ... 1016
`eval-defun`, and explicit entry to debugger ... 300
`eval-expression` (Edebug) ... 308
`eval-expression`, and `lexical-binding` ... 189
`eval-expression-debug-on-error` ... 297
`eval-expression-print-length` ... 341
`eval-expression-print-level` ... 341
`eval-minibuffer` ... 347
`eval-region` ... 141
`eval-region` (Edebug) ... 307
`eval-when-compile` ... 289
evaluated expression argument ... 381
evaluation ... 133
evaluation error ... 177
evaluation list group ... 315
evaluation notation ... 3
evaluation of buffer contents ... 141
evaluation of special forms ... 137
`evaporate` (overlay property) ... 1006
`even-window-sizes` ... 650
event printing ... 525
event translation ... 408
event type ... 398
event, reading only one ... 405
`event-basic-type` ... 399
`event-click-count` ... 394
`event-convert-list` ... 399
`event-end` ... 400
`event-modifiers` ... 398
`event-start` ... 400
`eventp` ... 388
events ... 387
ewoc ... 1072
`ewoc-buffer` ... 1073
`ewoc-collect` ... 1075
`ewoc-create` ... 1073
`ewoc-data` ... 1074
`ewoc-delete` ... 1074
`ewoc-enter-after` ... 1074
`ewoc-enter-before` ... 1074
`ewoc-enter-first` ... 1073
`ewoc-enter-last` ... 1073
`ewoc-filter` ... 1075
`ewoc-get-hf` ... 1073
`ewoc-goto-next` ... 1074
`ewoc-goto-node` ... 1074
`ewoc-goto-prev` ... 1074
`ewoc-invalidate` ... 1074
`ewoc-locate` ... 1074
`ewoc-location` ... 1074
`ewoc-map` ... 1075
`ewoc-next` ... 1074
`ewoc-nth` ... 1074
`ewoc-prev` ... 1074
`ewoc-refresh` ... 1074
`ewoc-set-data` ... 1074
`ewoc-set-hf` ... 1073
examining text properties ... 813
examining the `interactive` form ... 378
examining windows ... 635
examples of using `interactive` ... 381
excess close parentheses ... 328
excess open parentheses ... 328
excursion ... 765
`exec-directory` ... 935

`exec-path` ... 935, 936
`exec-suffixes` ... 935
`executable-find` ... 549
execute program ... 934
execute with prefix argument ... 383
`execute-extended-command` ... 383
`execute-kbd-macro` ... 420
`executing-kbd-macro` ... 420
execution order of buffer display
  action functions ... 651
execution speed ... 1167
`exit` ... 417
exit recursive editing ... 417
`exit-minibuffer` ... 370
`exit-recursive-edit` ... 418
exiting Emacs ... 1097
`exp` ... 49
`expand-abbrev` ... 925
`expand-file-name` ... 559
expanding abbrevs ... 925
expansion of file names ... 559
expansion of macros ... 242
explicit selective display ... 994
`explicit-name`, a frame parameter ... 715
expression ... 133
`expt` ... 50
extended file attributes ... 548
extended menu item ... 448
`extended-command-history` ... 349
extent ... 185
external border ... 703
external menu bar ... 703
external tool bar ... 704
`external-debugging-output` ... 337
extra slots of char-table ... 110
`extra-keyboard-modifiers` ... 408
`extract_big_integer` ... 1193
`extract_float` ... 1194
`extract_integer` ... 1193

F

`face` (button property) ... 1068
face (non-removability of) ... 1015
`face` (overlay property) ... 1004
`face` (text property) ... 820
face alias ... 1023
face attributes ... 1012
face attributes, access and modification ... 1018
face codes of text ... 820
face merging ... 1021
face name ... 1011
face number ... 1023
face property of face symbols ... 1023
face remapping ... 1021
face spec ... 1015
`face-all-attributes` ... 1018
`face-attribute` ... 1018
`face-attribute-relative-p` ... 1018
`face-background` ... 1020
`face-bold-p` ... 1020
`face-differs-from-default-p` ... 1023
`face-documentation` ... 520, 1023
`face-equal` ... 1023
`face-extend-p` ... 1020
`face-filters-always-match` ... 826
`face-font` ... 1020
`face-font-family-alternatives` ... 1025
`face-font-registry-alternatives` ... 1026
`face-font-rescale-alist` ... 1026
`face-font-selection-order` ... 1026
`face-foreground` ... 1020
`face-id` ... 1023
`face-inverse-video-p` ... 1020
`face-italic-p` ... 1020
`face-list` ... 1023
`face-name-history` ... 349
`face-remap-add-relative` ... 1022
`face-remap-remove-relative` ... 1023
`face-remap-reset-base` ... 1023
`face-remap-set-base` ... 1023
`face-remapping-alist` ... 1022
`face-spec-set` ... 1017
`face-stipple` ... 1020
`face-underline-p` ... 1020
`facemenu-keymap` ... 428
`facep` ... 1012
faces ... 1011
faces for font lock ... 504
faces, automatic choice ... 1024
false ... 2
`fboundp` ... 226
`fceiling` ... 46
`feature-unload-function` ... 282
`featurep` ... 281
features ... 279, 281
`fetch-bytecode` ... 289
`ffloor` ... 46
`field` (overlay property) ... 1004
`field` (text property) ... 823
field numbers in format spec ... 63
field width ... 64
`field-beginning` ... 831
`field-end` ... 831
`field-string` ... 832
`field-string-no-properties` ... 832
fields ... 831
fifo data structure ... 115
file accessibility ... 541
file age ... 545
file attributes ... 545
file classification ... 542
file contents, and default coding system ... 871
file format conversion ... 573
file hard link ... 550
file local variables ... 197, 1131

file locks ... 539
file mode specification error ... 468
file modes ... 542
file modes and MS-DOS ... 542
file modes, setting ... 552
file modification time ... 545
file name abbreviations ... 559
file name completion subroutines ... 563
file name handler ... 569
file name of buffer ... 594
file name of directory ... 558
file name, and default coding system ... 872
file names ... 555
file names in directory ... 565
file names, trailing whitespace ... 540
file notifications ... 1127
file open error ... 533
file permissions ... 542
file permissions, setting ... 552
file with multiple names ... 550
file, information about ... 540
`file-accessible-directory-p` ... 541
`file-acl` ... 548
`file-already-exists` ... 552
`file-attributes` ... 546
`file-chase-links` ... 544
`file-coding-system-alist` ... 872
`file-directory-p` ... 543
`file-equal-p` ... 545
`file-error` ... 271
`file-executable-p` ... 541
`file-exists-p` ... 541
`file-expand-wildcards` ... 567
`file-extended-attributes` ... 548
`file-in-directory-p` ... 545
`file-local-copy` ... 572
`file-local-name` ... 572
`file-local-variables-alist` ... 198
`file-locked` ... 540
`file-locked-p` ... 540
`file-modes` ... 542
`file-modes-symbolic-to-number` ... 554
file-name encoding, MS-Windows ... 867
`file-name-absolute-p` ... 557
`file-name-all-completions` ... 563
`file-name-as-directory` ... 558
`file-name-base` ... 557
`file-name-case-insensitive-p` ... 545
`file-name-coding-system` ... 867
`file-name-completion` ... 564
`file-name-directory` ... 555
`file-name-extension` ... 556
`file-name-handler-alist` ... 569
`file-name-history` ... 349
`file-name-nondirectory` ... 555
`file-name-quote` ... 561
`file-name-quoted-p` ... 561
`file-name-sans-extension` ... 556
`file-name-sans-versions` ... 556
`file-name-unquote` ... 561
`file-newer-than-file-p` ... 545
`file-newest-backup` ... 583
`file-nlinks` ... 548
`file-notify-add-watch` ... 1127
`file-notify-rm-watch` ... 1129
`file-notify-valid-p` ... 1129
`file-ownership-preserved-p` ... 542
`file-precious-flag` ... 536
`file-readable-p` ... 541
`file-regular-p` ... 544
`file-relative-name` ... 557
`file-remote-p` ... 572
`file-selinux-context` ... 548
`file-supersession` ... 598
`file-symlink-p` ... 543
`file-truename` ... 544
`file-writable-p` ... 541
`filepos-to-bufferpos` ... 853
`fill-column` ... 800
`fill-context-prefix` ... 801
`fill-forward-paragraph-function` ... 799
`fill-individual-paragraphs` ... 798
`fill-individual-varying-indent` ... 798
`fill-nobreak-predicate` ... 801
`fill-paragraph` ... 797
`fill-paragraph-function` ... 799
`fill-prefix` ... 800
`fill-region` ... 797
`fill-region-as-paragraph` ... 798
`fill-separate-heterogeneous-words-with-space` ... 799
`fillarray` ... 108
filling text ... 797
filling, automatic ... 802
filter function ... 955
filter multibyte flag, of process ... 957
`filter-buffer-substring` ... 779
`filter-buffer-substring-function` ... 779
`filter-buffer-substring-functions` ... 780
filtering killed text ... 788
filtering sequences ... 102
find file in path ... 549
find library ... 272
`find-auto-coding` ... 873
`find-backup-file-name` ... 583
`find-buffer-visiting` ... 595
`find-charset-region` ... 863
`find-charset-string` ... 863
`find-coding-systems-for-charsets` ... 869
`find-coding-systems-region` ... 868
`find-coding-systems-string` ... 868
`find-file` ... 531
`find-file-hook` ... 532
`find-file-literally` ... 531, 533
`find-file-name-handler` ... 571
`find-file-noselect` ... 531

find-file-not-found-functions ... 532
find-file-other-window ... 532
find-file-read-only ... 532
find-file-wildcards ... 532
find-font ... 1031
find-image ... 1062
find-operation-coding-system ... 873
find-word-boundary-function-table ... 757
finding files ... 530
finding windows ... 634
first-change-hook ... 851
fit-frame-to-buffer ... 620
fit-frame-to-buffer-margins ... 620
fit-frame-to-buffer-margins, a frame parameter ... 719
fit-frame-to-buffer-sizes ... 620
fit-frame-to-buffer-sizes, a frame parameter ... 719
fit-window-to-buffer ... 619
fit-window-to-buffer-horizontally ... 619
fixed-size window ... 616
fixnump ... 40
fixup-whitespace ... 787
flags in format specifications ... 63
flatten-tree ... 78
float ... 42
float-e ... 50
float-output-format ... 341
float-pi ... 50
float-time ... 1105
floating-point functions ... 49
floatp ... 40
floats-consed ... 1183
floor ... 42
flowcontrol, in serial connections ... 976
flushing input ... 412
fmakunbound ... 226
fn in function's documentation string ... 277
focus event ... 395
focus-follows-mouse ... 735
focus-in-hook ... 1224
focus-out-hook ... 1224
follow links ... 828
follow-link (button property) ... 1069
follow-link (text or overlay property) ... 830
following-char ... 777
font and color, frame parameters ... 726
font entity ... 1031
font information for layout ... 1033
font lock faces ... 504
Font Lock mode ... 496
font lookup ... 1027
font object ... 1030
font property ... 1029
font registry ... 1030
font selection ... 1025
font spec ... 1030
font, a frame parameter ... 728
font-at ... 1030
font-backend, a frame parameter ... 726
font-face-attributes ... 1031
font-family-list ... 1015
font-get ... 1031
font-info ... 1032
font-lock-add-keywords ... 501
font-lock-builtin-face ... 505
font-lock-comment-delimiter-face ... 505
font-lock-comment-face ... 505
font-lock-constant-face ... 505
font-lock-debug-fontify ... 497
font-lock-defaults ... 497
font-lock-doc-face ... 505
font-lock-ensure &optional beg end ... 497
font-lock-ensure-function ... 503
font-lock-extend-after-change-region-function ... 508
font-lock-extra-managed-props ... 502
font-lock-face (text property) ... 820
font-lock-flush &optional beg end ... 497
font-lock-flush-function ... 503
font-lock-fontify-buffer ... 496
font-lock-fontify-buffer-function ... 503
font-lock-fontify-region beg end &optional loudly ... 497
font-lock-fontify-region-function ... 503
font-lock-function-name-face ... 504
font-lock-keyword-face ... 505
font-lock-keywords ... 498
font-lock-keywords-case-fold-search ... 501
font-lock-keywords-only ... 505
font-lock-mark-block-function ... 502
font-lock-multiline ... 507
font-lock-negation-char-face ... 505
font-lock-preprocessor-face ... 505
font-lock-remove-keywords ... 502
font-lock-string-face ... 505
font-lock-syntactic-face-function ... 506
font-lock-syntax-table ... 506
font-lock-type-face ... 505
font-lock-unfontify-buffer ... 497
font-lock-unfontify-buffer-function ... 503
font-lock-unfontify-region beg end ... 497
font-lock-unfontify-region-function ... 503
font-lock-variable-name-face ... 505
font-lock-warning-face ... 504
font-put ... 1031
font-spec ... 1030
font-xlfd-name ... 1032
fontification-functions ... 1024
fontified (text property) ... 821
fontp ... 1029
fontset ... 1027
foo ... 4
for ... 245
force coding system for operation ... 874
force entry to debugger ... 300

`force-mode-line-update` ... 485
`force-window-update` ... 982
forcing redisplay ... 981
`foreground-color`, a frame parameter ... 728
form ... 133
form, self-evaluating ... 134
`format` ... 61
format definition ... 574
format of gnutls cryptography inputs ... 839
format of keymaps ... 423
format specification ... 62
`format`, customization keyword ... 264
`format-alist` ... 574
`format-find-file` ... 575
`format-insert-file` ... 576
`format-message` ... 61
`format-mode-line` ... 493
`format-network-address` ... 974
`format-seconds` ... 1112
`format-spec` ... 65
`format-time-string` ... 1110
`format-write-file` ... 575
formatting numbers for rereading later ... 64
formatting strings ... 61
formatting time values ... 1109
formfeed ... 11
forms for cleanup ... 172
forms for control structures ... 145
forms for handling errors ... 167
forms for nonlocal exits ... 162
forms for sequential execution ... 145
forms, backquote ... 139
forms, conditional ... 146
forms, function call ... 136
forms, iteration ... 159
forms, list ... 135
forms, macro call ... 137
forms, quote ... 139
forms, special ... 137
forms, symbol ... 134
`forward-button` ... 1072
`forward-char` ... 756
`forward-comment` ... 914
`forward-line` ... 759
`forward-list` ... 762
`forward-sexp` ... 763
`forward-to-indentation` ... 811
`forward-word` ... 757
`forward-word-strictly` ... 757
frame ... 696
frame configuration ... 739
frame creation ... 697
frame geometry ... 701
frame height ratio ... 717
frame interaction parameters ... 721
frame layout ... 702
frame layout parameters ... 719
frame parameters ... 712
frame parameters for windowed displays ... 714
frame position ... 701, 708, 715
frame position changes, a hook ... 709
frame size ... 701, 709
frame stacking order ... 737
frame title ... 730
frame visibility ... 736
frame width ratio ... 717
frame without a minibuffer ... 732
frame Z-order ... 737
frame, which buffers to display ... 721
`frame-alpha-lower-limit` ... 727
`frame-ancestor-p` ... 742
`frame-auto-hide-function` ... 664
`frame-char-height` ... 707
`frame-char-width` ... 707
`frame-current-scroll-bars` ... 1040
`frame-edges` ... 707
`frame-first-window` ... 612
`frame-focus-state` ... 735
`frame-geometry` ... 706
`frame-height` ... 709
`frame-inherited-parameters` ... 698
`frame-inhibit-implied-resize` ... 711
`frame-inner-height` ... 710
`frame-inner-width` ... 710
`frame-list` ... 731
`frame-list-z-order` ... 731
`frame-live-p` ... 731
`frame-monitor-attributes` ... 701
`frame-native-height` ... 709
`frame-native-width` ... 709
`frame-old-selected-window` ... 695
`frame-outer-height` ... 709
`frame-outer-width` ... 709
`frame-parameter` ... 713
`frame-parameters` ... 713
`frame-parent` ... 742
`frame-pointer-visible-p` ... 744
`frame-position` ... 708
`frame-predicate`, a buffer display action alist entry ... 646
frame-relative coordinate ... 681
`frame-resize-pixelwise` ... 710
`frame-restack` ... 738
`frame-root-window` ... 610
`frame-scroll-bar-height` ... 1040
`frame-scroll-bar-width` ... 1040
`frame-selected-window` ... 632
`frame-size-changed-p` ... 711
`frame-terminal` ... 696
`frame-text-height` ... 710
`frame-text-width` ... 710
`frame-title-format` ... 730
`frame-visible-p` ... 736
`frame-width` ... 709
`frame-window-state-change` ... 694
`framep` ... 696

frames, scanning all ... 731
free list ... 1179
free variable ... 190
free variable, byte-compiler warning ... 290
`free_global_ref` ... 1198
frequency counts ... 317
`frexp` ... 39
fringe bitmaps ... 1037
fringe bitmaps, customizing ... 1038
fringe cursors ... 1036
fringe indicators ... 1035
`fringe-bitmaps-at-pos` ... 1038
`fringe-cursor-alist` ... 1037
`fringe-indicator-alist` ... 1036
fringes ... 1034
fringes, and empty line indication ... 1035
`fringes-outside-margins` ... 1034
`fround` ... 46
`fset` ... 226
`ftp-login` ... 172
`ftruncate` ... 46
full keymap ... 423
full-height window ... 615
full-width window ... 615
fullboth frames ... 718
fullheight frames ... 718
`fullscreen`, a frame parameter ... 718
`fullscreen-restore`, a frame parameter ... 719
fullwidth frames ... 718
`func-arity` ... 210
`funcall` ... 217, 1200
`funcall`, and debugging ... 305
`funcall-interactively` ... 383
`function` ... 221
function aliases ... 216
function call ... 136
function call debugging ... 298
function cell ... 124
function cell in autoload ... 275
function declaration ... 238
function definition ... 214
function descriptions ... 4
function form evaluation ... 136
function input stream ... 332
function invocation ... 217
function keys ... 389
function name ... 214
function not known to be defined, compilation warning ... 290
function output stream ... 335
function quoting ... 221
function safety ... 239
function's documentation string ... 213
`function-documentation` property ... 519
`function-get` ... 130
`function-put` ... 130
functionals ... 218
`functionp` ... 210
functions in modes ... 465
functions, making them interactive ... 376
`fundamental-mode` ... 463
`fundamental-mode-abbrev-table` ... 927
future history in minibuffer input ... 343

G

gamma correction ... 727
`gap-position` ... 607
`gap-size` ... 607
garbage collection ... 1178
garbage collection protection ... 1186
`garbage-collect` ... 1179
`garbage-collection-messages` ... 1181
`gc-cons-percentage` ... 1182
`gc-cons-threshold` ... 1181
`gc-elapsed` ... 1182
`gcs-done` ... 1182
generalized variable ... 205
`generate-autoload-cookie` ... 277
`generate-new-buffer` ... 603
`generate-new-buffer-name` ... 593
`generated-autoload-file` ... 277
generators ... 160
generic commands ... 382
generic functions ... 222
generic mode ... 477
`gensym` ... 127
geometry specification ... 728
`get` ... 129
`get`, `defcustom` keyword ... 255
`get-buffer` ... 593
`get-buffer-create` ... 602
`get-buffer-process` ... 954
`get-buffer-window` ... 636
`get-buffer-window-list` ... 636
`get-buffer-xwidgets` ... 1067
`get-byte` ... 857
`get-char-code-property` ... 860
`get-char-property` ... 813
`get-char-property-and-overlay` ... 814
`get-charset-property` ... 862
`get-device-terminal` ... 698
`get-file-buffer` ... 595
`get-internal-run-time` ... 1113
`get-largest-window` ... 635
`get-load-suffixes` ... 272
`get-lru-window` ... 634
`get-mru-window` ... 635
`get-pos-property` ... 814
`get-process` ... 947
`get-register` ... 835
`get-text-property` ... 813
`get-unused-category` ... 919
`get-variable-watchers` ... 184
`get-window-with-predicate` ... 635
`get_user_finalizer` ... 1199

`getenv` ... 1101
`gethash` ... 120
GID ... 1104
global binding ... 175
global break condition ... 313
global keymap ... 430
global variable ... 174
`global-abbrev-table` ... 927
`global-buffers-menu-map` ... 1221
`global-disable-point-adjustment` ... 387
`global-key-binding` ... 437
`global-map` ... 432
`global-mode-string` ... 490
`global-set-key` ... 444
`global-unset-key` ... 444
glyph ... 1081
glyph code ... 1081
`glyph-char` ... 1081
`glyph-face` ... 1081
glyphless characters ... 1081
`glyphless-char-display` ... 1081
`glyphless-char-display-control` ... 1082
gnutls cryptographic functions ... 840
gnutls cryptography inputs format ... 839
`gnutls-ciphers` ... 841
`gnutls-digests` ... 840
`gnutls-hash-digest` ... 840
`gnutls-hash-mac` ... 840
`gnutls-macs` ... 840
`gnutls-symmetric-decrypt` ... 841
`gnutls-symmetric-encrypt` ... 841
`goto-char` ... 756
`goto-history-element` ... 370
`goto-map` ... 428
grammar, SMIE ... 511
graphical display ... 696
graphical terminal ... 696
`group`, customization keyword ... 250
`group-gid` ... 1104
`group-name` ... 1104
`group-real-gid` ... 1104
`gui-get-selection` ... 747
`gui-set-selection` ... 747
guidelines for buffer display ... 655
`gv-define-expander` ... 207
`gv-define-setter` ... 207
`gv-define-simple-setter` ... 207
`gv-letplace` ... 208

H

`hack-connection-local-variables` ... 203
`hack-connection-local-variables-apply` ... 203
`hack-dir-local-variables` ... 200
`hack-dir-local-variables-non-file-buffer` ... 200
`hack-local-variables` ... 197
`hack-local-variables-hook` ... 198
Hamming weight ... 49
handle Lisp errors ... 167
`handle-focus-in` ... 734
`handle-shift-selection` ... 775
`handle-switch-frame` ... 734
handling errors ... 167
hardening ... 1131
hash code ... 121
hash notation ... 8
hash table access ... 120
hash tables ... 118
hash, cryptographic ... 838, 839
`hash-table-count` ... 122
`hash-table-p` ... 122
`hash-table-rehash-size` ... 123
`hash-table-rehash-threshold` ... 123
`hash-table-size` ... 123
`hash-table-test` ... 122
`hash-table-weakness` ... 122
hashing ... 126
header comments ... 1172
header line (of a window) ... 493
`header-line` prefix key ... 405
`header-line-format` ... 493
`header-line-format`, a window parameter ... 690
height of a line ... 1010
height of a window ... 613
height spec ... 1010
`height`, a frame parameter ... 718
help for major mode ... 470
help functions ... 526
`help-buffer` ... 528
`help-char` ... 527
`help-command` ... 526
`help-echo` (button property) ... 1069
`help-echo` (overlay property) ... 1004
`help-echo` (text property) ... 821
`help-echo` event ... 396
help-echo text, avoid command-key substitution ... 821
`help-echo`, customization keyword ... 264
`help-echo-inhibit-substitution` (text property) ... 821
`help-event-list` ... 527
`help-form` ... 527
`help-map` ... 526
`help-setup-xref` ... 528
`help-window-select` ... 528
`Helper-describe-bindings` ... 527
`Helper-help` ... 527
`Helper-help-map` ... 528
hex numbers ... 37
hidden buffers ... 592
history list ... 348
history of commands ... 419
`history-add-new-input` ... 349
`history-delete-duplicates` ... 349
`history-length` ... 349

HOME environment variable 934
hook variables, list of 1223
hooks 461
hooks for changing a character 824
hooks for loading 283
hooks for motion of point 825
hooks for text changes 850
hooks for window operations 691
horizontal combination 610
horizontal position 806
horizontal scrolling 680
horizontal-lineto 1059
horizontal-scroll-bar 1042
horizontal-scroll-bar prefix key 405
horizontal-scroll-bar-mode 1042
horizontal-scroll-bars, a frame parameter 720
horizontal-scroll-bars-available-p 1040
how to visit files 530
HTML DOM 842
hyper characters 13
hyperlinks in documentation strings 1169

I

icon-left, a frame parameter 716
icon-name, a frame parameter 724
icon-title-format 730
icon-top, a frame parameter 717
icon-type, a frame parameter 724
iconified frame 736
iconify-child-frame 742
iconify-frame 737
iconify-frame event 396
identical-contents objects, and byte-compiler ... 34
identity 218
idle timers 1116
idleness 1116
IEEE floating point 38
if 146
ignore 219
ignore-error 170
ignore-errors 170
ignore-window-parameters 689
ignored-local-variables 199
image animation 1065
image cache 1065
image descriptor 1049
image formats 1049
image frames 1065
image maps 1052
image slice 1063
image types 1049
image-animate 1065
image-animate-timer 1065
image-cache-eviction-delay 1066
image-current-frame 1065
image-default-frame-delay 1065
image-flush 1066
image-format-suffixes 1054
image-load-path 1062
image-load-path-for-library 1062
image-mask-p 1053
image-minimum-frame-delay 1065
image-multi-frame-p 1065
image-property 1062
image-scaling-factor 1063
image-show-frame 1065
image-size 1064
image-transforms-p 1053
image-type-available-p 1049
image-types 1049
ImageMagick images 1054
imagemagick-enabled-types 1054
imagemagick-types 1054
imagemagick-types-inhibit 1054
images in buffers 1049
images, support for more formats 1054
Imenu 494
imenu-add-to-menubar 494
imenu-case-fold-search 495
imenu-create-index-function 495
imenu-extract-index-name-function 495
imenu-generic-expression 494
imenu-prev-index-position-function 495
imenu-syntax-alist 495
implicit progn 145
implied frame resizing 711
implied resizing of frame 711
inactive minibuffer 343
inc 242
indefinite extent 185
indent-according-to-mode 808
indent-code-rigidly 810
indent-for-tab-command 807
indent-line-function 808
indent-region 809
indent-region-function 809
indent-relative 810
indent-relative-first-indent-point 810
indent-rigidly 809
indent-tabs-mode 807
indent-to 807
indent-to-left-margin 801
indentation 807
indentation rules, SMIE 514
indicate-buffer-boundaries 1035
indicate-empty-lines 1035
indicators, fringe 1035
indirect buffers 605
indirect specifications 322
indirect-function 136
indirect-variable 204
indirection for functions 135
infinite loops 298
infinite recursion 177

infinity 39
inheritance, for faces 1014
inheritance, keymap 427
inheritance, syntax table 906
inheritance, text property 827
inhibit-default-init 1094
inhibit-double-buffering, a frame parameter 724
inhibit-eol-conversion 875
inhibit-field-text-motion 757
inhibit-file-name-handlers 571
inhibit-file-name-operation 571
inhibit-iso-escape-detection 869
inhibit-local-variables-regexps 197, 468
inhibit-message 984
inhibit-modification-hooks 851
inhibit-nul-byte-detection 869
inhibit-point-motion-hooks 826
inhibit-quit 414
inhibit-read-only 599
inhibit-read-only (text property) 822
inhibit-same-window, a buffer display action alist entry 646
inhibit-splash-screen 1092
inhibit-startup-echo-area-message 1092
inhibit-startup-message 1092
inhibit-startup-screen 1092
inhibit-switch-frame, a buffer display action alist entry 646
inhibit-x-resources 751
init file 1093
init-file-user 1103
init.el 1093
initial-buffer-choice 1092
initial-environment 1102
initial-frame-alist 714
initial-major-mode 469
initial-scratch-message 1093
initial-window-system 1084
initial-window-system, and startup 1090
initialization of Emacs 1090
initialize, defcustom keyword 255
inline completion 365
inline functions 235
inline-const-p 236
inline-const-val 236
inline-error 237
inline-letevals 236
inline-quote 236
inner edges 705
inner frame 705
inner height 705
inner size 705
inner width 705
innermost containing parentheses 916
input events 387
input events, prevent recording 411
input focus 732
input methods 878
input modes 1118
input stream 331
input-decode-map 442
input-method-alist 879
input-method-function 409
input-pending-p 411
insert 782
insert-abbrev-table-description 923
insert-and-inherit 828
insert-before-markers 782
insert-before-markers-and-inherit 828
insert-behind-hooks (overlay property) 1005
insert-behind-hooks (text property) 824
insert-buffer 783
insert-buffer-substring 782
insert-buffer-substring-as-yank 789
insert-buffer-substring-no-properties 783
insert-button 1070
insert-char 782
insert-default-directory 361
insert-directory 567
insert-directory-program 567
insert-file-contents 537
insert-file-contents-literally 537
insert-for-yank 789
insert-image 1063
insert-in-front-hooks (overlay property) 1005
insert-in-front-hooks (text property) 824
insert-register 835
insert-sliced-image 1063
insert-text-button 1070
inserting killed text 790
insertion before point 781
insertion of text 781
insertion type of a marker 771
inside comment 916
inside string 916
installation-directory 1102
instrumenting for Edebug 307
int-to-string 60
intangible (overlay property) 1005
intangible (text property) 822
integer range 38
integer to decimal 60
integer to hexadecimal 62
integer to octal 62
integer to string 60
integer types (C programming language) 1214
integer-or-marker-p 769
integer-width 38
integerp 40
integers 37
integers in specific radix 37
interaction parameters between frames 721
interactive 376
interactive call 382
interactive code description 378

interactive completion 378
interactive function 376
interactive spec, using 376
interactive specification in primitives 1185
`interactive`, examples of using 381
`interactive-form` 378
`interactive-form` property 376
`interactive-form`, symbol property 376
`interactive-only` property 376
`intern` 127, 1200
`intern-soft` 128
internal menu bar 703
internal representation of characters 852
internal tool bar 704
internal windows 608
`internal-border-width`, a frame parameter ... 719
internals, of buffer 1203
internals, of process 1212
internals, of window 1208
interning 126
interpreter 133
`interpreter-mode-alist` 469
`interprogram-cut-function` 792
`interprogram-paste-function` 792
interrupt Lisp functions 413
`interrupt-process` 952
`interrupt-process-functions` 952
intersection of sequences 105
intervals 832
`intervals-consed` 1183
invalid prefix key error 438
`invalid-function` 135
`invalid-read-syntax` 8
`invalid-regexp` 891
`invert-face` 1020
`invisible` (overlay property) 1005
`invisible` (text property) 822
invisible frame 736
invisible text 992
`invisible-p` 993
invisible/intangible text, and point 387
`invocation-directory` 1102
`invocation-name` 1102
invoking input method 409
invoking lisp debugger 303
is this call interactive 384
`is_not_nil` 1200
`isnan` 39
`iso8601-parse` 1109
`ISO 8601 date/time strings` 1109
italic text 1012
`iter-close` 161
`iter-defun` 160
`iter-do` 161
`iter-lambda` 161
`iter-next` 161
`iter-yield` 161
`iter-yield-from` 161
iteration 159
iteration over vector or string 106

J

JavaScript Object Notation 844
`jit-lock-register` 503
`jit-lock-unregister` 503
joining lists 84
`json-insert` 845
`json-parse-buffer` 846
`json-parse-string` 845
`json-serialize` 845
JSON remote procedure call protocol 846
`jsonrpc-async-request` 846
`jsonrpc-connection` 846
`jsonrpc-connection-ready-p` 848
`jsonrpc-connection-receive` 847
`jsonrpc-connection-send` 847
`jsonrpc-error` 846
`jsonrpc-lambda` 848
`jsonrpc-notify` 846
`jsonrpc-process-connection` 847
`jsonrpc-request` 846
`jsonrpc-running-p` 847
`jsonrpc-shutdown` 847
JSON 844
JSONRPC 846
JSONRPC application interfaces 846
JSONRPC connection initargs 847
JSONRPC deferred requests 848
JSONRPC object format 848
JSONRPC process-based connections 847
jumbled display of bidirectional text 1087
`just-one-space` 787
`justify-current-line` 798

K

`kbd` 422
`kbd-macro-termination-hook` 421
`keep-ratio`, a frame parameter 722
`kept-new-versions` 581
`kept-old-versions` 581
key 422
key binding 423
key binding, conventions for 1164
key lookup 434
key sequence 422
key sequence error 438
key sequence input 404
key substitution sequence 522
key translation function 443
`key-binding` 430
`key-description` 525
`key-translation-map` 442
keyboard events 388
keyboard events in strings 402

keyboard events, data in ... 400
keyboard input ... 403
keyboard input decoding on X ... 879
keyboard macro execution ... 383
keyboard macro termination ... 1083
keyboard macro, terminating ... 412
keyboard macros ... 420
keyboard macros (Edebug) ... 309
`keyboard-coding-system` ... 878
`keyboard-quit` ... 415
`keyboard-translate` ... 409
`keyboard-translate-table` ... 408
keymap ... 422
`keymap` (button property) ... 1068
`keymap` (overlay property) ... 1006
`keymap` (text property) ... 821
keymap entry ... 434
keymap format ... 423
keymap in keymap ... 435
keymap inheritance ... 427
keymap inheritance from multiple maps ... 427
keymap of character ... 821
keymap of character (and overlays) ... 1006
keymap prompt string ... 424
`keymap-parent` ... 427
`keymap-prompt` ... 448
`keymapp` ... 425
keymaps for translating events ... 442
keymaps in modes ... 465
keymaps, scanning ... 445
keymaps, standard ... 1220
keys in documentation strings ... 522
keys, reserved ... 1165
keystroke ... 422
keyword symbol ... 174
`keywordp` ... 175
kill command repetition ... 385
kill ring ... 788
`kill-all-local-variables` ... 194
`kill-append` ... 792
`kill-buffer` ... 604
`kill-buffer-hook` ... 604
`kill-buffer-query-functions` ... 604
`kill-emacs` ... 1097
`kill-emacs-hook` ... 1097
`kill-emacs-query-functions` ... 1098
`kill-local-variable` ... 194
`kill-new` ... 792
`kill-process` ... 952
`kill-read-only-ok` ... 789
`kill-region` ... 789
`kill-ring` ... 793
`kill-ring-max` ... 793
`kill-ring-yank-pointer` ... 793
killing buffers ... 603
killing Emacs ... 1097
`kmacro-keymap` ... 1221

L

`lambda` ... 221
lambda expression ... 211
`lambda` in debug ... 303
`lambda` in keymap ... 435
lambda list ... 211
lambda-list (Edebug) ... 323
`language-change` event ... 397
largest fixnum ... 38
largest window ... 635
`last` ... 75
last visible position in a window ... 672
`last-abbrev` ... 926
`last-abbrev-location` ... 926
`last-abbrev-text` ... 926
`last-buffer` ... 601
`last-coding-system-used` ... 867
`last-command` ... 385
`last-command-event` ... 387
`last-event-frame` ... 387
`last-input-event` ... 411
`last-kbd-macro` ... 420
`last-nonmenu-event` ... 387
`last-prefix-arg` ... 416
`last-repeatable-command` ... 385
`lax-plist-get` ... 94
`lax-plist-put` ... 94
layout of frame ... 702
layout on display, and bidirectional text ... 1087
layout parameters of frames ... 719
lazy evaluation ... 143
lazy loading ... 288
`lazy-completion-table` ... 353
`ldexp` ... 39
least recently used window ... 634
left position ratio ... 716
`left`, a frame parameter ... 715
`left-fringe`, a frame parameter ... 720
`left-fringe-width` ... 1034
`left-margin` ... 801
`left-margin-width` ... 1048
`length` ... 95
`let` ... 176
`let*` ... 176
`let-alist` ... 92
`letrec` ... 176
Levenshtein distance ... 59
lexical binding ... 185
lexical binding (Edebug) ... 314
lexical comparison of strings ... 57
lexical environment ... 187
lexical scope ... 185
`lexical-binding` ... 189
library ... 269
library compilation ... 287
library header comments ... 1172
library search ... 272
`libxml-available-p` ... 841

libxml-parse-html-region ... 842
libxml-parse-xml-region ... 842
line end conversion ... 865
line height ... 707, 1010
line number ... 760
line truncation ... 982
line wrapping ... 982
line-beginning-position ... 759
line-end-position ... 759
line-height (text property) ... 824, 1010
line-move-ignore-invisible ... 994
line-number-at-pos ... 760
line-number-display-width ... 1010
line-pixel-height ... 1009
line-prefix ... 983
line-spacing ... 1011
line-spacing (text property) ... 823, 1011
line-spacing, a frame parameter ... 721
lines ... 758
lines in region ... 759
lineto ... 1058
link, customization keyword ... 250
linked list ... 15
linking files ... 550
Lisp debugger ... 296
Lisp expression motion ... 762
Lisp history ... 1
Lisp library ... 269
Lisp nesting error ... 142
Lisp object ... 8
Lisp objects, stack-allocated ... 1183
Lisp package ... 1134
Lisp printer ... 338
Lisp reader ... 331
Lisp timestamp ... 1104
lisp variables defined in C, restrictions ... 205
lisp-indent-function property ... 249
lisp-mode-abbrev-table ... 927
lisp-mode.el ... 478
list ... 76
list all coding systems ... 867
list elements ... 73
list form evaluation ... 135
list in keymap ... 435
list length ... 95
list modification ... 79
list motion ... 762
list of threads ... 932
list predicates ... 72
list reverse ... 97
list structure ... 15, 71
list to vector ... 105
list, replace element ... 81
list-buffers-directory ... 596
list-charset-chars ... 862
list-fonts ... 1031
list-load-path-shadows ... 274
list-processes ... 947
list-system-processes ... 961
list-threads ... 932
list-timers ... 1116
listify-key-sequence ... 411
listing all buffers ... 600
listp ... 72
lists ... 71
lists and cons cells ... 71
lists as sets ... 85
literal evaluation ... 134
literate programming ... 808
little endian ... 977
live buffer ... 603
live windows ... 608
ln ... 552
load ... 269
load error with require ... 279
load errors ... 270
load, customization keyword ... 251
load-average ... 1102
load-file ... 271
load-file-name ... 271
load-file-rep-suffixes ... 272
load-history ... 281
load-in-progress ... 271
load-library ... 271
load-path ... 272
load-prefer-newer ... 272
load-read-function ... 271
load-suffixes ... 271
load-theme ... 268
loading ... 269
loading hooks ... 283
loading, and non-ASCII characters ... 274
loadup.el ... 1175
local binding ... 175
local keymap ... 430
local part of remote file name ... 572
local variables ... 175
local variables, killed by major mode ... 194
local, defcustom keyword ... 255
local-abbrev-table ... 927
local-function-key-map ... 442
local-key-binding ... 437
local-map (overlay property) ... 1006
local-map (text property) ... 822
local-set-key ... 445
local-unset-key ... 445
local-variable-if-set-p ... 193
local-variable-p ... 193
locale ... 879
locale-coding-system ... 879
locale-dependent string comparison ... 58
locale-dependent string equivalence ... 57
locale-info ... 880
locate file in path ... 549
locate-dominating-file ... 566
locate-file ... 549

`locate-library` ... 273
`locate-user-emacs-file` ... 565
lock file ... 539
`lock-buffer` ... 540
`log` ... 49
`logand` ... 47
`logb` ... 39
`logcount` ... 49
logging echo-area messages ... 988
logical arithmetic ... 46
logical order ... 1085
logical shift ... 47
`logior` ... 48
`lognot` ... 49
`logxor` ... 48
looking up abbrevs ... 925
looking up fonts ... 1027
`looking-at` ... 895
`looking-at-p` ... 896
`looking-back` ... 896
lookup tables ... 118
`lookup-key` ... 436
loops, infinite ... 298
lower case ... 67
`lower-frame` ... 738
lowering a frame ... 737
LRO ... 1088
`lsh` ... 47
`lwarn` ... 990

M

M-g ... 428
M-o ... 428
M-s ... 428
M-x ... 383
Maclisp ... 2
macro ... 209
macro argument evaluation ... 246
macro call ... 242
macro call evaluation ... 137
macro caveats ... 245
macro compilation ... 286
macro descriptions ... 4
macro expansion ... 243
macro, how to define ... 244
`macroexpand` ... 243
`macroexpand-1` ... 243
`macroexpand-all` ... 243
`macrop` ... 242
macros ... 242
macros, at compile time ... 290
magic autoload comment ... 276
magic file names ... 569
`magic-fallback-mode-alist` ... 469
`magic-mode-alist` ... 469
`mail-host-address` ... 1101
main window ... 664
main window of a frame ... 664
`main-thread` ... 930
major mode ... 463
major mode command ... 463
major mode conventions ... 464
major mode hook ... 467
major mode keymap ... 430
major mode, automatic selection ... 468
`major-mode` ... 464
`major-mode-restore` ... 464
`major-mode-suspend` ... 464
`make-abbrev-table` ... 922
`make-auto-save-file-name` ... 584
`make-backup-file-name` ... 582
`make-backup-file-name-function` ... 580
`make-backup-files` ... 579
`make-bool-vector` ... 113
`make-button` ... 1070
`make-byte-code` ... 292
`make-category-set` ... 920
`make-category-table` ... 920
`make-char-table` ... 111
`make-composed-keymap` ... 427
`make-condition-variable` ... 932
`make-decoded-time` ... 1108
`make-directory` ... 568
`make-display-table` ... 1079
`make-empty-file` ... 568
`make-finalizer` ... 24
`make-frame` ... 697
`make-frame-invisible` ... 737
`make-frame-on-display` ... 699
`make-frame-on-monitor` ... 701
`make-frame-visible` ... 737
`make-frame-visible` event ... 396
`make-glyph-code` ... 1081
`make-hash-table` ... 118
`make-help-screen` ... 528
`make-indirect-buffer` ... 605
`make-keymap` ... 426
`make-list` ... 76
`make-local-variable` ... 192
`make-marker` ... 769
`make-mutex` ... 931
`make-nearby-temp-file` ... 563
`make-network-process` ... 968
`make-obsolete` ... 234
`make-obsolete-variable` ... 204
`make-overlay` ... 1000
`make-pipe-process` ... 943
`make-process` ... 941
`make-progress-reporter` ... 986
`make-record` ... 116
`make-ring` ... 114
`make-serial-process` ... 975
`make-sparse-keymap` ... 426
`make-string` ... 52
`make-symbol` ... 127

make-symbolic-link ... 552
make-syntax-table ... 910
make-temp-file ... 562
make-temp-name ... 563
make-text-button ... 1070
make-thread ... 929
make-translation-table ... 863
make-translation-table-from-alist ... 864
make-translation-table-from-vector ... 864
make-variable-buffer-local ... 193
make-vector ... 110
make-xwidget ... 1067
make_big_integer ... 1195
make_float ... 1198
make_function ... 1191
make_global_ref ... 1198
make_integer ... 1195
make_string ... 1198
make_time ... 1198
make_user_ptr ... 1199
making backup files ... 578
making buttons ... 1069
makunbound ... 178
malicious use of directional overrides ... 1088
managing overlays ... 1000
manipulating buttons ... 1070
map-char-table ... 112
map-charset-chars ... 862
map-keymap ... 446
map-y-or-n-p ... 368
mapatoms ... 128
mapbacktrace ... 306
mapc ... 220
mapcan ... 220
mapcar ... 219
mapconcat ... 220
maphash ... 120
mapped frame ... 736
mapping functions ... 219
margins, display ... 1047
margins, filling ... 800
mark ... 773
mark excursion ... 766
mark ring ... 773
mark, the ... 772
mark-active ... 775
mark-even-if-inactive ... 774
mark-marker ... 773
mark-ring ... 775
mark-ring-max ... 775
marker argument ... 380
marker creation ... 769
marker garbage collection ... 768
marker information ... 771
marker input stream ... 332
marker output stream ... 335
marker relocation ... 768
marker, how to move position ... 772
marker-buffer ... 771
marker-insertion-type ... 771
marker-position ... 771
markerp ... 769
markers ... 768
markers as numbers ... 768
markers, predicates for ... 769
match data ... 897
match, customization keyword ... 265
match-alternatives, customization keyword ... 262
match-beginning ... 899
match-data ... 900
match-end ... 899
match-inline, customization keyword ... 265
match-string ... 899
match-string-no-properties ... 899
match-substitute-replacement ... 898
matching, structural ... 157
mathematical functions ... 49
max ... 41
max-char ... 856
max-image-size ... 1064
max-lisp-eval-depth ... 142
max-mini-window-height ... 372
max-specpdl-size ... 177
maximize-window ... 621
maximized frames ... 718
maximizing windows ... 621
maximum fixnum ... 38
maximum value of character codepoint ... 856
maximum value of sequence ... 106
maximum-scroll-margin ... 677
maybe_quit, use in Lisp primitives ... 1186
md5 ... 839
MD5 checksum ... 838, 839
measuring resource usage ... 329
member ... 87
member-ignore-case ... 88
membership in a list ... 85
memory allocation ... 1178
memory usage ... 329, 1183
memory-full ... 1182
memory-info ... 1182
memory-limit ... 1182
memory-use-counts ... 1182
memq ... 85
memql ... 86
menu bar ... 453
menu bar keymaps ... 1221
menu definition example ... 452
menu item ... 447
menu keymaps ... 447
menu modification ... 458
menu prompt string ... 447
menu separators ... 450
menu-bar prefix key ... 405
menu-bar-file-menu ... 1221

menu-bar-final-items ... 454
menu-bar-help-menu ... 1221
menu-bar-lines, a frame parameter ... 720
menu-bar-options-menu ... 1221
menu-bar-tools-menu ... 1221
menu-bar-update-hook ... 454
menu-item ... 448
menu-prompt-more-char ... 452
menus, popup ... 744
merge-face-attribute ... 1018
message ... 983
message digest ... 838
message, finding what causes a particular message ... 298
message-box ... 985
message-log-max ... 988
message-or-box ... 985
message-truncate-lines ... 989
messages-buffer ... 988
meta character key constants ... 438
meta character printing ... 525
meta characters ... 13
meta characters lookup ... 425
meta-prefix-char ... 437
min ... 42
min-height, a frame parameter ... 718
min-margins, a window parameter ... 691
min-width, a frame parameter ... 718
minibuffer ... 342
minibuffer completion ... 353
minibuffer contents, accessing ... 372
minibuffer history ... 348
minibuffer input ... 417
minibuffer input, and command-line arguments ... 936
minibuffer input, reading lisp objects ... 347
minibuffer input, reading text strings ... 343
minibuffer window, and next-window ... 633
minibuffer windows ... 371
minibuffer, a frame parameter ... 721
minibuffer-allow-text-properties ... 346
minibuffer-auto-raise ... 738
minibuffer-beginning-of-buffer-movement ... 357
minibuffer-complete ... 356
minibuffer-complete-and-exit ... 356
minibuffer-complete-word ... 355
minibuffer-completion-confirm ... 355
minibuffer-completion-help ... 356
minibuffer-completion-predicate ... 355
minibuffer-completion-table ... 355
minibuffer-confirm-exit-commands ... 355
minibuffer-contents ... 372
minibuffer-contents-no-properties ... 372
minibuffer-depth ... 373
minibuffer-exit, a frame parameter ... 722
minibuffer-exit-hook ... 373
minibuffer-frame-alist ... 714
minibuffer-help-form ... 374
minibuffer-history ... 349
minibuffer-inactive-mode ... 374
minibuffer-less frame ... 705
minibuffer-local-completion-map ... 356
minibuffer-local-filename-completion-map ... 357
minibuffer-local-map ... 346
minibuffer-local-must-match-map ... 356
minibuffer-local-ns-map ... 346
minibuffer-local-shell-command-map ... 362
minibuffer-message ... 374
minibuffer-message (text property) ... 826
minibuffer-message-timeout ... 374
minibuffer-only frame ... 705, 714
minibuffer-prompt ... 372
minibuffer-prompt-end ... 372
minibuffer-prompt-properties ... 344
minibuffer-prompt-width ... 372
minibuffer-scroll-window ... 374
minibuffer-selected-window ... 374
minibuffer-setup-hook ... 373
minibuffer-window ... 371
minibuffer-window-active-p ... 371
minibuffer-with-setup-hook ... 373
minibufferp ... 373
minimize-window ... 621
minimized frame ... 736
minimizing windows ... 621
minimum fixnum ... 38
minimum value of sequence ... 105
minor mode ... 479
minor mode conventions ... 480
minor-mode-alist ... 490
minor-mode-key-binding ... 437
minor-mode-list ... 480
minor-mode-map-alist ... 433
minor-mode-overriding-map-alist ... 433
mirroring of characters ... 858
mkdir ... 568
mod ... 45
mode ... 461
mode bits ... 542
mode help ... 470
mode hook ... 467
mode line ... 485
mode line construct ... 486
mode loading ... 468
mode variable ... 480
mode, a buffer display action alist entry ... 646
mode-class (property) ... 467
mode-line prefix key ... 405
mode-line-buffer-identification ... 489
mode-line-client ... 489
mode-line-coding-system-map ... 1221
mode-line-column-line-number-mode-map ... 1221
mode-line-end-spaces ... 490
mode-line-format ... 487

`mode-line-format`, a window parameter 690
`mode-line-frame-identification` 489
`mode-line-front-space` 490
`mode-line-input-method-map` 1221
`mode-line-misc-info` 490
`mode-line-modes` 489
`mode-line-modified` 488
`mode-line-mule-info` 488
`mode-line-percent-position` 489
`mode-line-position` 489
`mode-line-process` 489
`mode-line-remote` 489
`mode-name` 489
`mode-specific-map` 428
model/view/controller 1072
modification flag (of buffer) 596
modification of lists 84
modification time of buffer 597
modification time of file 546
`modification-hooks` (overlay property) 1004
`modification-hooks` (text property) 824
modifier bits (of input character) 388
modifiers of events 408
modify a list 79
`modify-all-frames-parameters` 713
`modify-category-entry` 920
`modify-frame-parameters` 713
`modify-syntax-entry` 911
modifying strings 55
module API 1188
module functions 1191
module initialization 1189
module runtime environment 1190
module values, conversion 1193
`module-file-suffix` 284
`module-load` 284
`module_func` 1191
modulus 45
`momentary-string-display` 999
most recently selected windows 632
most recently used window 635
`most-negative-fixnum` 38
`most-positive-fixnum` 38
motion based on parsing 914
motion by chars, words, lines, lists 756
motion event 395
mouse click event 390
mouse drag event 392
mouse dragging parameters 723
mouse events, data in 400
mouse events, in special parts of frame 405
mouse events, repeated 393
mouse motion events 395
mouse pointer shape 746
mouse position 743
mouse position list 391
mouse position list, accessing 400
mouse tracking 742
`mouse`, a face 728
mouse, availability 752
mouse-1 828
`mouse-1-click-follows-link` 829
mouse-2 1164
`mouse-absolute-pixel-position` 744
`mouse-action` (button property) 1068
`mouse-appearance-menu-map` 1221
`mouse-autoselect-window` 685
`mouse-color`, a frame parameter 728
`mouse-face` (button property) 1068
`mouse-face` (overlay property) 1004
`mouse-face` (text property) 820
`mouse-fine-grained-tracking` 395
`mouse-leave-buffer-hook` 1225
`mouse-movement-p` 399
`mouse-on-link-p` 831
`mouse-pixel-position` 743
`mouse-position` 743
`mouse-position-function` 743
`mouse-wheel-down-event` 396
`mouse-wheel-frame`, a frame parameter 722
`mouse-wheel-up-event` 396
move to beginning or end of buffer 758
`move-frame-functions` 709
`move-marker` 772
`move-overlay` 1001
`move-point-visually` 1087
`move-to-column` 806
`move-to-left-margin` 800
`move-to-window-group-line` 761
`move-to-window-group-line-function` 761
`move-to-window-line` 761
`movemail` 935
`moveto` 1058
moving across syntax classes 913
moving markers 772
MS-DOS and file modes 542
MS-Windows file-name syntax 555
`mule-keymap` 428
multi-file package 1136
multi-frame images 1065
multi-mode indentation 808
multi-monitor 700
`multi-query-replace-map` 904
multi-tty 698
multibyte characters 852
multibyte text 852
`multibyte-char-to-unibyte` 855
`multibyte-string-p` 853
`multibyte-syntax-as-symbol` 917
multiline font lock 506
multiple terminals 698
multiple windows 608
multiple X displays 698
multiple yes-or-no questions 368
multiple-dispatch methods 225
`multiple-frames` 730

mutable lists ... 81
mutable objects ... 35
mutex-lock ... 931
mutex-name ... 931
mutex-unlock ... 931
mutexp ... 931

N

name, a frame parameter ... 715
named function ... 214
naming backup files ... 582
NaN ... 39
narrow-map ... 1222
narrow-to-page ... 766
narrow-to-region ... 766
narrowing ... 766
native edges ... 704
native frame ... 704
native height ... 704
native position ... 704
native size ... 704
native width ... 704
natnump ... 40
natural numbers ... 40
nbutlast ... 75
nconc ... 84
negative infinity ... 39
negative-argument ... 417
nest frame ... 739
network byte ordering ... 977
network connection ... 964
network connection, encrypted ... 965
network servers ... 967
network service name, and default coding system ... 872
network-coding-system-alist ... 872
network-interface-info ... 973
network-interface-list ... 972
network-lookup-address-info ... 974
network-stream-use-client-certificates ... 966
new file message ... 533
newline ... 11, 784
newline and Auto Fill mode ... 784
newline in print ... 338
newline in strings ... 19
newline-and-indent ... 808
next input ... 410
next-button ... 1072
next-char-property-change ... 818
next-complete-history-element ... 370
next-frame ... 731
next-history-element ... 370
next-matching-history-element ... 370
next-overlay-change ... 1007
next-property-change ... 817
next-screen-context-lines ... 678
next-single-char-property-change ... 818
next-single-property-change ... 817
next-window ... 633
nil ... 2
nil as a list ... 16
nil in keymap ... 435
nil input stream ... 332
nil output stream ... 335
nlistp ... 72
no-accept-focus, a frame parameter ... 724
no-byte-compile ... 285
no-catch ... 163
no-conversion coding system ... 865
no-delete-other-windows, a window parameter ... 689
no-focus-on-map, a frame parameter ... 724
no-other-frame, a frame parameter ... 722
no-other-window, a window parameter ... 690
no-redraw-on-reenter ... 981
no-self-insert property ... 923
no-special-glyphs, a frame parameter ... 721
node, ewoc ... 1072
non-ASCII characters ... 852
non-ASCII characters in loaded files ... 274
non-ASCII text in keybindings ... 444
non-capturing group ... 890
non-greedy repetition characters in regexp ... 885
non_local_exit_clear ... 1202
non_local_exit_signal ... 1202
non_local_exit_throw ... 1202
nondirectory part (of file name) ... 555
noninteractive ... 1122
nonlocal exits ... 162
nonlocal exits, cleaning up ... 172
nonlocal exits, in modules ... 1201
nonprinting characters, reading ... 410
noreturn ... 329
normal hook ... 461
normal-auto-fill-function ... 802
normal-backup-enable-predicate ... 579
normal-mode ... 468
not ... 148
not recording input events ... 411
not-modified ... 597
notation ... 3
notifications, on desktop ... 1123
notifications-close-notification ... 1125
notifications-get-capabilities ... 1125
notifications-get-server-information ... 1126
notifications-notify ... 1123
nreverse ... 98
ns-appearance, a frame parameter ... 725
ns-transparent-titlebar, a frame parameter ... 725
nth ... 74
nthcdr ... 74
null ... 72
null bytes, and decoding text ... 869
num-input-keys ... 405

`num-nonmacro-input-events` ... 407
number comparison ... 40
number conversions ... 42
number of bignum bits, limit on ... 38
`number-or-marker-p` ... 769
`number-sequence` ... 78
`number-to-string` ... 60
numbered backups ... 581
`numberp` ... 40
numbers ... 37
numeric prefix argument ... 415
numeric prefix argument usage ... 380
numerical RGB color specification ... 748

O

obarray ... 126, 128
obarray in completion ... 351
object ... 8
object internals ... 1202
object to string ... 338
objects with identical contents, and byte-compiler ... 34
obsolete functions ... 234
octal character code ... 12
octal character input ... 410
octal escapes ... 1078
octal numbers ... 37
old advices, porting ... 233
`old-selected-frame` ... 695
`old-selected-window` ... 695
`one-window-p` ... 634
`only-global-abbrevs` ... 924
opacity, frame ... 727
`open-dribble-file` ... 1119
`open-network-stream` ... 965
`open-paren-in-column-0-is-defun-start` ... 763
`open-termscript` ... 1120
OpenType font ... 1030
operating system environment ... 1100
operating system signal ... 1097
`operations` (property) ... 571
optimize regexp ... 893
option descriptions ... 5
optional arguments ... 212
options on command line ... 1096
`options`, `defcustom` keyword ... 254
`or` ... 148
ordering of windows, cyclic ... 633
origin of display ... 705
`other-buffer` ... 601
`other-window` ... 634
`other-window`, a window parameter ... 689
`other-window-scroll-buffer` ... 676
outer border ... 703
outer edges ... 702
outer frame ... 702
outer height ... 702
outer position ... 702
outer size ... 702
outer width ... 702
`outer-window-id`, a frame parameter ... 724
output from processes ... 953
output stream ... 335
output-controlling variables ... 339
overall prompt string ... 424
`overflow-newline-into-fringe` ... 1036
overlay properties ... 1002
overlay, empty ... 1000
`overlay-arrow-position` ... 1039
`overlay-arrow-string` ... 1039
`overlay-arrow-variable-list` ... 1039
`overlay-buffer` ... 1000
`overlay-end` ... 1000
`overlay-get` ... 1003
`overlay-properties` ... 1003
`overlay-put` ... 1003
`overlay-recenter` ... 1002
`overlay-start` ... 1000
`overlayp` ... 1000
overlays ... 999
overlays, managing ... 1000
overlays, scalability ... 999
overlays, searching for ... 1006
`overlays-at` ... 1006
`overlays-in` ... 1006
overlined text ... 1013
override existing functions ... 216
override redirect frames ... 725
override spec (for a face) ... 1017
`override-redirect`, a frame parameter ... 725
overriding bidirectional properties ... 1088
`overriding-local-map` ... 433
`overriding-local-map-menu-flag` ... 433
`overriding-terminal-local-map` ... 433
`overwrite-mode` ... 784

P

package ... 1134
package archive ... 1138
package archive security ... 1138
package attributes ... 1134
package autoloads ... 1135
package dependencies ... 1134
package name ... 1134
package signing ... 1138
package version ... 1134
`package-activate-all` ... 1135
`package-archives` ... 1138
`package-initialize` ... 1135
`package-version`, customization keyword ... 252
packing ... 977
padding ... 64
`page-delimiter` ... 905
`paragraph-separate` ... 905

paragraph-separate, and
bidirectional display ... 1086
paragraph-start ... 905
paragraph-start, and bidirectional display ... 1086
parameters for moving frames
with the mouse ... 723
parameters for resizing frames
with the mouse ... 723
parameters of initial frame ... 713
parent frames ... 739
parent of char-table ... 111
parent process ... 934
parent window ... 610
parent-frame, a buffer display
action alist entry ... 648
parent-frame, a frame parameter ... 722
parenthesis ... 15
parenthesis depth ... 916
parenthesis matching ... 1077
parenthesis mismatch, debugging ... 328
parity, in serial connections ... 976
parse state for a position ... 915
parse-colon-path ... 1102
parse-partial-sexp ... 916
parse-sexp-ignore-comments ... 917
parse-sexp-lookup-properties ... 913
parse-time-string ... 1109
parser state ... 915
parsing buffer text ... 906
parsing expressions ... 914
parsing html ... 841
parsing xml ... 842
parsing, control parameters ... 917
partial application of functions ... 218
partial-width windows ... 982
passwords, reading ... 369
path-separator ... 1102
PATH environment variable ... 934
pattern matching, programming style ... 149
PBM ... 1061
pcase ... 149, 150
pcase pattern ... 150
pcase, defining new kinds of patterns ... 156
pcase-defmacro ... 156
pcase-dolist ... 159
pcase-let ... 159
pcase-let* ... 159
pdumper-stats ... 1177
peculiar error ... 171
peeking at input ... 410
percent symbol in mode line ... 486
perform-replace ... 902
performance analysis ... 329
performance analysis (Edebug) ... 317
permanent local variable ... 195
permissions, file ... 542, 552
persistent window parameters ... 688
phishing using directional overrides ... 1088
piece of advice ... 227
pipe, when to use for subprocess
communications ... 941
pixel height of a window ... 614
pixel width of a window ... 615
pixelwise, resizing windows ... 619
place form ... 205
play-sound ... 1120
play-sound-file ... 1121
play-sound-functions ... 1121
plist ... 92
plist access ... 93
plist vs. alist ... 93
plist-get ... 93
plist-member ... 94
plist-put ... 93
plugin_is_GPL_compatible ... 284
point ... 755
point excursion ... 765, 766
point in window ... 671
point with narrowing ... 755
point-entered (text property) ... 825
point-left (text property) ... 825
point-marker ... 770
point-max ... 755
point-max-marker ... 770
point-min ... 755
point-min-marker ... 770
pointer (text property) ... 823
pointer shape ... 746
pointers ... 15
polymorphism ... 222
pop ... 74
pop-mark ... 774
pop-to-buffer ... 639
pop-up-frame-alist ... 650
pop-up-frame-function ... 650
pop-up-frame-parameters, a buffer
display action alist entry ... 648
pop-up-frames ... 650
pop-up-frames, replacement for ... 651
pop-up-windows ... 649
pop-up-windows, replacement for ... 651
popcount ... 49
port number, and default coding system ... 872
pos-visible-in-window-group-p ... 674
pos-visible-in-window-group-p-function ... 674
pos-visible-in-window-p ... 674
position (in buffer) ... 755
position argument ... 379
position in window ... 671
position of frame ... 701, 708
position of mouse ... 743
position-bytes ... 853
positive infinity ... 39
posix-looking-at ... 897
posix-search-backward ... 897
posix-search-forward ... 897

posix-string-match ... 897
posn-actual-col-row ... 401
posn-area ... 400
posn-at-point ... 402
posn-at-x-y ... 402
posn-col-row ... 401
posn-image ... 401
posn-object ... 401
posn-object-width-height ... 401
posn-object-x-y ... 401
posn-point ... 400
posn-string ... 401
posn-timestamp ... 401
posn-window ... 400
posn-x-y ... 400
posnp ... 400
post-command-hook ... 375
post-gc-hook ... 1181
post-self-insert-hook ... 783
pp ... 339
pre-command-hook ... 375
pre-redisplay-function ... 982
pre-redisplay-functions ... 982
precedence of buffer display action functions ... 651
preceding-char ... 778
precision in format specifications ... 64
predicates for lists ... 72
predicates for markers ... 769
predicates for numbers ... 40
predicates for strings ... 52
prefer-utf-8 coding system ... 865
prefix argument ... 415
prefix argument unreading ... 410
prefix command ... 429
prefix key ... 428
prefix, defgroup keyword ... 253
prefix-arg ... 416
prefix-command-echo-keystrokes-functions ... 1225
prefix-command-preserve-state-hook ... 1226
prefix-help-command ... 527
prefix-numeric-value ... 416
preloaded Lisp files ... 1176
preloaded-file-list ... 1176
preloading additional functions and variables ... 1176
prepare-change-group ... 849
preserve-size, a buffer display action alist entry ... 648
preserving window sizes ... 621
pretty-printer ... 339
preventing backtracking ... 322
preventing prefix key ... 435
preventing quitting ... 414
previous complete subexpression ... 916
previous-button ... 1072
previous-char-property-change ... 818
previous-complete-history-element ... 370
previous-frame ... 732
previous-history-element ... 370
previous-matching-history-element ... 370
previous-overlay-change ... 1007
previous-property-change ... 817
previous-single-char-property-change ... 818
previous-single-property-change ... 818
previous-window ... 634
previous-window, a buffer display action alist entry ... 646
primary selection ... 747
primitive ... 209
primitive function ... 23
primitive function internals ... 1184
primitive type ... 8
primitive-undo ... 795
prin1 ... 338
prin1-to-string ... 338
princ ... 338
print ... 338
print example ... 335
print name cell ... 124
print-charset-text-property ... 340
print-circle ... 341
print-continuous-numbering ... 341
print-escape-control-characters ... 340
print-escape-multibyte ... 340
print-escape-newlines ... 339
print-escape-nonascii ... 340
print-gensym ... 341
print-length ... 340
print-level ... 341
print-number-table ... 341
print-quoted ... 339
printable ASCII characters ... 1078
printable-chars ... 861
printed representation ... 8
printed representation for characters ... 11
printing ... 331
printing (Edebug) ... 316
printing circular structures ... 316
printing limits ... 340
printing notation ... 3
priority (overlay property) ... 1003
priority order of coding systems ... 875
process ... 934
process creation ... 934
process filter ... 955
process filter multibyte flag ... 957
process information ... 947
process input ... 950
process internals ... 1212
process output ... 953
process sentinel ... 959
process signals ... 951
process-adaptive-read-buffering ... 953
process-attributes ... 961
process-buffer ... 954

process-coding-system ... 949
process-coding-system-alist ... 872
process-command ... 947
process-connection-type ... 946
process-contact ... 947
process-datagram-address ... 968
process-environment ... 1101
process-exit-status ... 949
process-file ... 939
process-file-shell-command ... 940
process-file-side-effects ... 939
process-filter ... 956
process-get ... 950
process-id ... 948
process-kill-buffer-query-function ... 953
process-lines ... 941
process-list ... 947
process-live-p ... 949
process-mark ... 954
process-name ... 948
process-plist ... 950
process-put ... 950
process-query-on-exit-flag ... 961
process-running-child-p ... 951
process-send-eof ... 951
process-send-region ... 950
process-send-string ... 950
process-sentinel ... 960
process-status ... 948
process-thread ... 959
process-tty-name ... 949
process-type ... 949
processes, threads ... 958
processing of errors ... 166
processor run time ... 1112
processp ... 934
profile ... 329
profiler-report ... 329
profiler-start ... 329
profiler-stop ... 329
profiling ... 329
prog-first-column ... 809
prog-indentation-context ... 809
prog-mode ... 472
prog-mode, and bidi-paragraph-direction ... 1087
prog-mode-hook ... 472
prog1 ... 146
prog2 ... 146
progn ... 145
program arguments ... 935
program directories ... 935
program name, and default coding system ... 872
programmed completion ... 363
programming conventions ... 1165
programming types ... 10
progress reporting ... 985
progress-reporter-done ... 987
progress-reporter-force-update ... 987
progress-reporter-update ... 986
prompt for file name ... 359
prompt string (of menu) ... 447
prompt string of keymap ... 424
proper list ... 71
proper-list-p ... 72
properties of text ... 813
propertize ... 816
property category of text character ... 820
property list ... 92
property list cell ... 124
property lists vs association lists ... 93
protect C variables from garbage collection ... 1186
protected forms ... 172
provide ... 280
provide-theme ... 267
providing features ... 279
pseudo window ... 1209
pty, when to use for subprocess communications ... 941
pure function ... 209
pure property ... 131
pure storage ... 1177
pure-bytes-used ... 1178
purecopy ... 1178
purify-flag ... 1178
push ... 79
push-button ... 1071
push-mark ... 774
put ... 129
put-char-code-property ... 861
put-charset-property ... 862
put-image ... 1064
put-text-property ... 815
puthash ... 120

Q

quadratic-bezier-curveto ... 1059
query-font ... 1033
query-replace-history ... 349
query-replace-map ... 903
querying the user ... 366
question mark in character constant ... 11
quietly-read-abbrev-file ... 924
quit-flag ... 414
quit-process ... 952
quit-restore, a window parameter ... 690
quit-restore-window ... 662
quit-window ... 662
quit-window-hook ... 662
quitting ... 413
quitting from infinite loop ... 298
quote ... 139
quote character ... 916
quote special characters in regexp ... 892
quoted character input ... 410

`quoted-insert` suppression 440
quoting and unquoting
command-line arguments 936
quoting characters in printing 337
quoting using apostrophe 139

R

radio, customization types 261
radix for reading an integer 37
`raise-frame` 738
raising a frame 737
`random` 50
random numbers 50
`rassoc` 89
`rassq` 90
`rassq-delete-all` 92
raw prefix argument 415
raw prefix argument usage 380
raw syntax descriptor 917
`raw-text` coding system 865
`re-builder` 883
`re-search-backward` 895
`re-search-forward` 894
`read` 334
read command name 383
read file names 359
read input 403
read syntax 8
read syntax for characters 11
`read-answer` 369
`read-answer-short` 369
`read-buffer` 357
`read-buffer-completion-ignore-case` 358
`read-buffer-function` 358
`read-char` 406
`read-char-choice` 408
`read-char-exclusive` 407
`read-char-from-minibuffer` 369
`read-circle` 334
`read-coding-system` 871
`read-color` 358
`read-command` 358
`read-directory-name` 361
`read-event` 406
`read-expression-history` 349
`read-file-modes` 553
`read-file-name` 359
`read-file-name-completion-ignore-case` 361
`read-file-name-function` 360
`read-from-minibuffer` 343
`read-from-string` 334
`read-hide-char` 369
`read-input-method-name` 879
`read-kbd-macro` 526
`read-key` 407
`read-key-sequence` 404
`read-key-sequence-vector` 405
`read-minibuffer` 347
`read-multiple-choice` 408
`read-no-blanks-input` 346
`read-non-nil-coding-system` 871
`read-only` (text property) 822
read-only buffer 599
read-only buffers in interactive 377
read-only character 822
read-only variables 175
`read-only-mode` 599
`read-passwd` 370
`read-quoted-char` 410
`read-quoted-char` quitting 414
`read-regexp` 345
`read-regexp-defaults-function` 345
`read-shell-command` 361
`read-string` 344
`read-variable` 358
`read-variable`, history list 349
reading 331
reading a single event 405
reading from files 537
reading from minibuffer with completion 353
reading interactive arguments 379
reading numbers in hex, octal, and binary 37
reading order 1085
reading symbols 126
`real-last-command` 385
rearrangement of lists 84
rebinding 438
`recent-auto-save-p` 585
`recent-keys` 1118
`recenter` 678
`recenter-positions` 679
`recenter-redisplay` 678
`recenter-top-bottom` 678
`recenter-window-group` 678
`recenter-window-group-function` 678
recombining windows 626
`record` 116
record command history 383
recording input 1118
`recordp` 116
records 116
rectangle, as contents of a register 834
recursion 159
`recursion-depth` 418
recursive command loop 417
recursive editing level 417
recursive evaluation 133
recursive minibuffers 373
recursive traverse of directory tree 566
`recursive-edit` 418
`redirect-frame-focus` 734
`redisplay` 981
redo 793
`redraw-display` 981
`redraw-frame` 981

reducing sequences ... 102
reference to free variable, compilation warning ... 290
references, following ... 1164
refresh the screen ... 981
regexp ... 883
regexp alternative ... 889
regexp grouping ... 889
regexp searching ... 894
regexp syntax ... 884
regexp, special characters in ... 884
`regexp-history` ... 349
`regexp-opt` ... 893
`regexp-opt-charset` ... 894
`regexp-opt-depth` ... 893
`regexp-quote` ... 892
`regexp-unmatchable` ... 894
regexps used standardly in editing ... 905
region ... 776
region argument ... 380
`region-beginning` ... 776
`region-end` ... 776
register preview ... 835
`register-alist` ... 834
`register-definition-prefixes` ... 278
`register-read-with-preview` ... 835
registers ... 834
regular expression ... 883
regular expression searching ... 894
regular expressions, developing ... 883
`reindent-then-newline-and-indent` ... 808
relative file name ... 557
relative remapping, faces ... 1022
remainder ... 45
remapping commands ... 441
`remhash` ... 120
`remote-file-name-inhibit-cache` ... 573
`remove` ... 87
`remove-from-invisibility-spec` ... 993
`remove-function` ... 229
`remove-hook` ... 463
`remove-images` ... 1064
`remove-list-of-text-properties` ... 815
`remove-overlays` ... 1001
`remove-text-properties` ... 815
`remove-variable-watcher` ... 184
removing from sequences ... 102
`remq` ... 86
`rename-auto-save-file` ... 586
`rename-buffer` ... 593
`rename-file` ... 551
rendering html ... 842
reordering, of bidirectional text ... 1085
reordering, of elements in lists ... 84
reparent frame ... 739
repeat events ... 393
repeated loading ... 278
replace bindings ... 440
replace characters ... 833
replace characters in region ... 833
replace list element ... 81
replace matched text ... 897
replace part of list ... 83
`replace-buffer-contents` ... 836
`replace-buffer-in-windows` ... 637
`replace-match` ... 898
`replace-re-search-function` ... 905
`replace-regexp-in-string` ... 902
`replace-region-contents` ... 836
`replace-search-function` ... 904
replacement after search ... 902
replacing display specs ... 1043
`require` ... 280
`require`, customization keyword ... 251
`require-final-newline` ... 536
requiring features ... 279
reserved keys ... 1165
resize window ... 617
`resize-mini-frames` ... 372
`resize-mini-windows` ... 371
rest arguments ... 212
restacking a frame ... 737
`restore-buffer-modified-p` ... 596
restricted-sexp, customization types ... 262
restriction (in a buffer) ... 766
resume (cf. `no-redraw-on-reenter`) ... 981
`resume-tty` ... 1099
`resume-tty-functions` ... 1099
rethrow a signal ... 169
return (ASCII character) ... 11
return value ... 209
`reusable-frames`, a buffer display action alist entry ... 646
`reverse` ... 97
reversing a list ... 98
reversing a string ... 98
reversing a vector ... 98
`revert-buffer` ... 586
`revert-buffer-function` ... 587
`revert-buffer-in-progress-p` ... 587
`revert-buffer-insert-file-contents-function` ... 587
`revert-without-query` ... 587
reverting buffers ... 586
rgb value ... 749
right dividers ... 1042
`right-divider-width`, a frame parameter ... 720
`right-fringe`, a frame parameter ... 720
`right-fringe-width` ... 1034
`right-margin-width` ... 1048
right-to-left text ... 1085
ring data structure ... 114
`ring-bell-function` ... 1083
`ring-copy` ... 115
`ring-elements` ... 114
`ring-empty-p` ... 115

ring-insert ... 115
ring-insert-at-beginning ... 115
ring-length ... 114
ring-p ... 114
ring-ref ... 115
ring-remove ... 115
ring-resize ... 115
ring-size ... 114
risky, defcustom keyword ... 256
risky-local-variable-p ... 199
RLO ... 1088
rm ... 552
root window ... 610
root window of atomic window ... 669
round ... 43
rounding in conversions ... 42
rounding without conversion ... 46
rplaca ... 81
rplacd ... 81
run time stack ... 304
run-at-time ... 1115
run-hook-with-args ... 462
run-hook-with-args-until-failure ... 462
run-hook-with-args-until-success ... 462
run-hooks ... 462
run-mode-hooks ... 473
run-with-idle-timer ... 1116
run-with-timer ... 1115
running a hook when a window gets selected ... 631

S

S-expression ... 133
safe local variable ... 198
safe, defcustom keyword ... 256
safe-length ... 75
safe-local-eval-forms ... 199
safe-local-variable-p ... 199
safe-local-variable-values ... 199
safe-magic (property) ... 571
safely encode a string ... 868
safely encode characters in a charset ... 869
safely encode region ... 868
safety of functions ... 239
same-window-buffer-names, replacement for ... 651
same-window-regexps, replacement for ... 651
save abbrevs in files ... 924
save-abbrevs ... 924
save-buffer ... 534
save-buffer-coding-system ... 866
save-current-buffer ... 592
save-excursion ... 765
save-mark-and-excursion ... 766
save-match-data ... 901
save-restriction ... 766
save-selected-window ... 632
save-some-buffers ... 534
save-some-buffers-default-predicate ... 534
save-window-excursion ... 686
SaveUnder feature ... 753
saving buffers ... 534
saving text properties ... 573
saving window information ... 686
scalability of overlays ... 999
scalable fonts ... 1026
scalable-fonts-allowed ... 1026
scan-lists ... 914
scan-sexps ... 914
scanning expressions ... 914
scanning for character sets ... 863
scanning keymaps ... 445
scope ... 185
scoping rule ... 185
screen layout ... 27
screen lines, moving by ... 760
screen of terminal ... 608
screen refresh ... 981
screen-gamma, a frame parameter ... 727
script symbols ... 861
scroll bar events, data in ... 402
scroll bars ... 1039
scroll-bar-background, a frame parameter ... 728
scroll-bar-event-ratio ... 402
scroll-bar-foreground, a frame parameter ... 728
scroll-bar-height ... 1042
scroll-bar-height, a frame parameter ... 720
scroll-bar-mode ... 1042
scroll-bar-scale ... 402
scroll-bar-width ... 1042
scroll-bar-width, a frame parameter ... 720
scroll-command property ... 677
scroll-conservatively ... 677
scroll-down ... 676
scroll-down-aggressively ... 677
scroll-down-command ... 676
scroll-error-top-bottom ... 678
scroll-left ... 680
scroll-margin ... 676
scroll-other-window ... 676
scroll-other-window-down ... 676
scroll-preserve-screen-position ... 677
scroll-right ... 681
scroll-step ... 677
scroll-up ... 675
scroll-up-aggressively ... 677
scroll-up-command ... 676
scrolling textually ... 675
search-backward ... 882
search-failed ... 881
search-forward ... 881
search-map ... 428
search-spaces-regexp ... 896
searching ... 881
searching active keymaps for keys ... 431
searching and case ... 883
searching and replacing ... 902

searching for overlays ... 1006
searching for regexp ... 894
searching text properties ... 817
secondary selection ... 747
secondary storage ... 554
`secure-hash` ... 838
`secure-hash-algorithms` ... 838
security ... 1131
security, and display specifications ... 1043
seed, for random number generation ... 50
select safe coding system ... 870
select window hooks ... 631
`select-frame` ... 734
`select-frame-set-input-focus` ... 733
`select-safe-coding-system` ... 870
`select-safe-coding-system-accept-default-p` ... 871
`select-safe-coding-system-function` ... 871
`select-window` ... 631
selected frame ... 732
selected window ... 609
`selected-frame` ... 733
`selected-window` ... 609
`selected-window-group` ... 609
`selected-window-group-function` ... 609
selecting a buffer ... 590
selecting a font ... 1025
selecting a window ... 631
selection (for window systems) ... 747
`selection-coding-system` ... 747
`selective-display` ... 995
`selective-display-ellipses` ... 996
self-evaluating form ... 134
`self-insert-and-exit` ... 370
`self-insert-command` ... 783
`self-insert-command` override ... 440
`self-insert-command`, minor modes ... 482
`self-insert-uses-region-functions` ... 783
self-insertion ... 783
SELinux context ... 548
`send-string-to-terminal` ... 1119
sending signals ... 951
`sentence-end` ... 905
`sentence-end-double-space` ... 799
`sentence-end-without-period` ... 799
`sentence-end-without-space` ... 799
sentinel (of process) ... 959
seq library ... 99
`seq-concatenate` ... 104
`seq-contains-p` ... 103
`seq-count` ... 103
`seq-difference` ... 105
`seq-do` ... 101
`seq-doseq` ... 106
`seq-drop` ... 100
`seq-drop-while` ... 101
`seq-elt` ... 100
`seq-empty-p` ... 103
`seq-every-p` ... 103
`seq-filter` ... 102
`seq-find` ... 102
`seq-group-by` ... 105
`seq-intersection` ... 105
`seq-into` ... 105
`seq-length` ... 100
`seq-let` ... 106
`seq-map` ... 101
`seq-map-indexed` ... 101
`seq-mapcat` ... 104
`seq-mapn` ... 101
`seq-max` ... 106
`seq-min` ... 105
`seq-partition` ... 105
`seq-position` ... 104
`seq-random-elt` ... 106
`seq-reduce` ... 102
`seq-remove` ... 102
`seq-set-equal-p` ... 103
`seq-some` ... 102
`seq-sort` ... 103
`seq-sort-by` ... 103
`seq-subseq` ... 104
`seq-take` ... 100
`seq-take-while` ... 101
`seq-uniq` ... 104
`seqp` ... 100
sequence ... 95
sequence destructuring ... 106
sequence functions in seq ... 99
sequence iteration ... 106
sequence length ... 95
sequence maximum ... 106
sequence minimum ... 105
sequence reverse ... 97
`sequencep` ... 95
sequences, generalized ... 99
sequences, intersection of ... 105
sequencing ... 145
sequencing pattern ... 151
sequential execution ... 145
serial connections ... 974
`serial-process-configure` ... 976
`serial-term` ... 975
serializing ... 977
`server-after-make-frame-hook` ... 698
session file ... 1122
session manager ... 1122
`set` ... 183
`set`, `defcustom` keyword ... 254
`set-advertised-calling-convention` ... 235
`set-after`, `defcustom` keyword ... 256
`set-auto-coding` ... 873
`set-auto-mode` ... 468
`set-binary-mode` ... 335
`set-buffer` ... 591
`set-buffer-auto-saved` ... 585

set-buffer-major-mode 469
set-buffer-modified-p 596
set-buffer-multibyte 855
set-case-syntax 70
set-case-syntax-delims 70
set-case-syntax-pair 70
set-case-table 70
set-category-table 920
set-char-table-extra-slot 111
set-char-table-parent 111
set-char-table-range 112
set-charset-priority 861
set-coding-system-priority 875
set-default 196
set-default-file-modes 553
set-default-toplevel-value 197
set-display-table-slot 1080
set-face-attribute 1019
set-face-background 1019
set-face-bold 1019
set-face-extend 1020
set-face-font 1019
set-face-foreground 1019
set-face-inverse-video 1019
set-face-italic 1019
set-face-stipple 1019
set-face-underline 1019
set-file-acl 554
set-file-extended-attributes 554
set-file-modes 552
set-file-selinux-context 554
set-file-times 554
set-fontset-font 1028
set-frame-configuration 739
set-frame-font 707
set-frame-height 710
set-frame-parameter 713
set-frame-position 708
set-frame-selected-window 632
set-frame-size 710
set-frame-width 711
set-frame-window-state-change 694
set-fringe-bitmap-face 1039
set-input-method 879
set-input-mode 1118
set-keyboard-coding-system 878
set-keymap-parent 427
set-left-margin 800
set-mark 773
set-marker 772
set-marker-insertion-type 771
set-match-data 901
set-message-function 984
set-minibuffer-window 371
set-mouse-absolute-pixel-position 744
set-mouse-pixel-position 743
set-mouse-position 743
set-network-process-option 972
set-process-buffer 954
set-process-coding-system 949
set-process-datagram-address 968
set-process-filter 956
set-process-plist 950
set-process-query-on-exit-flag 961
set-process-sentinel 960
set-process-thread 959
set-process-window-size 954
set-register 835
set-right-margin 800
set-standard-case-table 69
set-syntax-table 912
set-terminal-coding-system 878
set-terminal-parameter 729
set-text-properties 815
set-transient-map 434
set-visited-file-modtime 598
set-visited-file-name 595
set-window-buffer 635
set-window-combination-limit 629
set-window-configuration 686
set-window-dedicated-p 662
set-window-display-table 1081
set-window-fringes 1034
set-window-group-start 674
set-window-group-start-function 674
set-window-hscroll 681
set-window-margins 1048
set-window-next-buffers 659
set-window-parameter 688
set-window-point 671
set-window-prev-buffers 659
set-window-scroll-bars 1040
set-window-start 673
set-window-vscroll 679
set-xwidget-plist 1067
set-xwidget-query-on-exit-flag 1068
set_user_finalizer 1199
set_user_ptr 1199
setcar 81
setcdr 83
setenv 1101
setf 206
setplist 129
setq 183
setq-default 195
setq-local 192
sets 85
setting modes of files 550
setting-constant error 174
severity level 989
sexp 133
sexp motion 762
SHA hash 838, 839
shadowed Lisp files 274
shadowing of variables 175
shared structure, read syntax 29

shell command arguments ... 936
shell-command-history ... 349
shell-command-to-string ... 941
shell-quote-argument ... 936
shift-selection, and interactive spec ... 377
shift-translation ... 405
should_quit ... 1200
show image ... 1063
show-help-function ... 826
shr-insert-document ... 842
shrink-window-if-larger-than-buffer ... 620
shy groups ... 890
sibling window ... 610
side effect ... 133
side windows ... 664
side, a buffer display action alist entry ... 648
side-effect-free property ... 132
SIGHUP ... 1097
SIGINT ... 1097
signal ... 166
signal-process ... 952
signaling errors ... 165
signals ... 951
SIGTERM ... 1097
SIGTSTP ... 1098
sigusr1 event ... 396
sigusr2 event ... 396
simple package ... 1135
sin ... 49
single file package ... 1135
single-function hook ... 461
single-key-description ... 525
sit-for ... 412
site-init.el ... 1176
site-lisp directories ... 273
site-load.el ... 1176
site-run-file ... 1094
site-start.el ... 1091
size of frame ... 701
size of image ... 1064
size of text on display ... 1007
size of window ... 613
skip-chars-backward ... 765
skip-chars-forward ... 764
skip-syntax-backward ... 913
skip-syntax-forward ... 913
skip-taskbar, a frame parameter ... 724
skipping characters ... 764
skipping characters of certain syntax ... 913
skipping comments ... 917
sleep-for ... 413
slice, image ... 1063
slot, a buffer display action alist entry ... 648
small-temporary-file-directory ... 562
smallest fixnum ... 38
smie-bnf->prec2 ... 510
smie-close-block ... 510
smie-config ... 517
smie-config-guess ... 517
smie-config-local ... 517
smie-config-save ... 517
smie-config-set-indent ... 517
smie-config-show-indent ... 517
smie-down-list ... 510
smie-merge-prec2s ... 510
smie-prec2->grammar ... 510
smie-precs->prec2 ... 510
smie-rule-bolp ... 515
smie-rule-hanging-p ... 515
smie-rule-next-p ... 515
smie-rule-parent ... 515
smie-rule-parent-p ... 515
smie-rule-prev-p ... 515
smie-rule-separator ... 515
smie-rule-sibling-p ... 515
smie-setup ... 509
SMIE ... 509
SMIE grammar ... 511
SMIE lexer ... 512
smooth-curveto ... 1059
smooth-quadratic-bezier-curveto ... 1060
snap-width, a frame parameter ... 723
Snarf-documentation ... 522
sort ... 98
sort-columns ... 806
sort-fields ... 805
sort-fold-case ... 804
sort-lines ... 805
sort-numeric-base ... 806
sort-numeric-fields ... 806
sort-pages ... 805
sort-paragraphs ... 805
sort-regexp-fields ... 804
sort-subr ... 803
sorting lists ... 98
sorting sequences ... 103
sorting text ... 803
sorting vectors ... 98
sound ... 1120
source breakpoints ... 313
space (ASCII character) ... 11
space display spec, and bidirectional text ... 1088
spaces, pixel specification ... 1045
spaces, specified height or width ... 1044
sparse keymap ... 423
SPC in minibuffer ... 346
special events ... 412
special form descriptions ... 4
special forms ... 137
special forms for control structures ... 145
special modes ... 467
special read syntax ... 9
special variables ... 189
special-event-map ... 434
special-form-p ... 137
special-mode ... 473

`special-variable-p` ... 190
Specified time is not representable ... 1105
specify coding system ... 874
specify color ... 748
speedups ... 1167
splicing (with backquote) ... 140
`split-height-threshold` ... 649
`split-string` ... 54
`split-string-and-unquote` ... 936
`split-string-default-separators` ... 55
`split-width-threshold` ... 650
`split-window` ... 622
`split-window`, a window parameter ... 689
`split-window-below` ... 624
`split-window-keep-point` ... 624
`split-window-preferred-function` ... 649
`split-window-right` ... 624
`split-window-sensibly` ... 649
splitting windows ... 622
`sqrt` ... 50
stable sort ... 98
stack allocated Lisp objects ... 1183
stack frame ... 301
standard abbrev tables ... 927
standard colors for character terminals ... 727
standard error process ... 943
standard errors ... 1216
standard hooks ... 1223
standard regexps used in editing ... 905
standard syntax table ... 906
`standard-case-table` ... 70
`standard-category-table` ... 920
`standard-display-table` ... 1081
`standard-input` ... 334
`standard-output` ... 339
`standard-syntax-table` ... 906
`standard-translation-table-for-decode` ... 864
`standard-translation-table-for-encode` ... 864
standards of coding style ... 1162
`start-file-process` ... 945
`start-file-process-shell-command` ... 946
`start-process` ... 944
`start-process`, command-line arguments from minibuffer ... 936
`start-process-shell-command` ... 945
STARTTLS network connections ... 965
startup of Emacs ... 1090
startup screen ... 1092
`startup.el` ... 1090
`staticpro`, protection from GC ... 1187
`stderr` stream, use for debugging ... 337
sticky text properties ... 827
`sticky`, a frame parameter ... 724
stop points ... 307
`stop-process` ... 952
stopbits, in serial connections ... 976
stopping an infinite loop ... 298
stopping on events ... 313
storage of vector-like Lisp objects ... 1178
`store-match-data` ... 901
`store-substring` ... 55
stream (for printing) ... 335
stream (for reading) ... 331
strike-through text ... 1013
`string` ... 52
string creation ... 52
string equality ... 56
string in keymap ... 435
string input stream ... 332
string length ... 95
string modification ... 55
string predicates ... 52
string reverse ... 97
string search ... 881
string to number ... 60
string to object ... 334
string to vector ... 105
string, number of bytes ... 853
string, writing a doc string ... 519
`string-as-multibyte` ... 856
`string-as-unibyte` ... 856
`string-bytes` ... 853
`string-chars-consed` ... 1183
`string-collate-equalp` ... 57
`string-collate-lessp` ... 58
`string-distance` ... 59
`string-equal` ... 56
`string-greaterp` ... 58
`string-lessp` ... 58
`string-match` ... 895
`string-match-p` ... 895
`string-or-null-p` ... 52
`string-prefix-p` ... 59
`string-suffix-p` ... 59
`string-to-char` ... 61
`string-to-int` ... 60
`string-to-multibyte` ... 855
`string-to-number` ... 60
`string-to-syntax` ... 918
`string-to-unibyte` ... 855
`string-version-lessp` ... 58
`string-width` ... 1007
`string<` ... 57
`string=` ... 56
`stringp` ... 52
strings ... 51
strings with keyboard events ... 402
strings, formatting them ... 61
`strings-consed` ... 1183
structural matching ... 157
sub-sequence ... 104
submenu ... 452
subprocess ... 934
subr ... 209
`subr-arity` ... 211
`subrp` ... 210

`subst-char-in-region` ... 833
substitute characters ... 833
`substitute-command-keys` ... 523
`substitute-in-file-name` ... 560
`substitute-key-definition` ... 440
substituting keys in documentation ... 522
`substring` ... 52
`substring-no-properties` ... 53
subtype of char-table ... 110
suggestions ... 1
super characters ... 13
`suppress-keymap` ... 440
surrogate minibuffer frame ... 732
suspend (cf. `no-redraw-on-reenter`) ... 981
suspend evaluation ... 418
suspend major mode temporarily ... 463
`suspend-emacs` ... 1098
`suspend-frame` ... 1100
`suspend-hook` ... 1099
`suspend-resume-hook` ... 1099
`suspend-tty` ... 1099
`suspend-tty-functions` ... 1099
suspending Emacs ... 1098
svg path commands ... 1058
`svg-circle` ... 1056
`svg-clip-path` ... 1057
`svg-create` ... 1055
`svg-ellipse` ... 1056
`svg-embed` ... 1057
`svg-gradient` ... 1055
`svg-image` ... 1058
`svg-line` ... 1056
`svg-node` ... 1058
`svg-path` ... 1057
`svg-polygon` ... 1056
`svg-polyline` ... 1056
`svg-rectangle` ... 1056
`svg-remove` ... 1058
`svg-text` ... 1057
SVG images ... 1055
SVG object ... 1055
swap text between buffers ... 606
`switch-to-buffer` ... 637
`switch-to-buffer-in-dedicated-window` ... 638
`switch-to-buffer-obey-display-actions` ... 638
`switch-to-buffer-other-frame` ... 638
`switch-to-buffer-other-window` ... 638
`switch-to-buffer-preserve-window-point` ... 638
`switch-to-next-buffer` ... 660
`switch-to-prev-buffer` ... 660
`switch-to-prev-buffer-skip` ... 660
switches on command line ... 1096
switching to a buffer ... 637
`sxhash-eq` ... 121
`sxhash-eql` ... 122
`sxhash-equal` ... 121
symbol ... 124
symbol components ... 124
symbol equality ... 126
symbol evaluation ... 134
symbol forms ... 134
symbol function indirection ... 135
symbol in keymap ... 435
symbol name hashing ... 126
symbol property ... 129
symbol that evaluates to itself ... 174
symbol with constant value ... 174
symbol, where defined ... 281
`symbol-file` ... 281
`symbol-function` ... 225
`symbol-name` ... 127
`symbol-plist` ... 129
`symbol-value` ... 182
symbolic links ... 542, 543
`symbolp` ... 124
`symbols-consed` ... 1183
symmetric cipher ... 839
synchronous subprocess ... 937
syntactic font lock ... 505
syntax class ... 907
syntax class table ... 907
syntax code ... 917
syntax descriptor ... 907
syntax entry, setting ... 911
syntax error (Edebug) ... 323
syntax flags ... 909
syntax for characters ... 11
syntax of regular expressions ... 884
syntax table ... 906
syntax table example ... 478
syntax table internals ... 917
syntax tables (accessing elements of) ... 917
syntax tables in modes ... 466
`syntax-after` ... 918
`syntax-class` ... 918
`syntax-ppss` ... 915
`syntax-ppss-context` ... 916
`syntax-ppss-flush-cache` ... 915
`syntax-ppss-toplevel-pos` ... 916
`syntax-propertize-extend-region-functions` ... 913
`syntax-propertize-function` ... 913
`syntax-table` ... 912
`syntax-table` (text property) ... 912
`syntax-table-p` ... 906
system abbrev ... 922
system processes ... 961
system type and name ... 1100
`system-configuration` ... 1100
`system-groups` ... 1104
`system-key-alist` ... 1121
`system-messages-locale` ... 879
`system-name` ... 1101
`system-time-locale` ... 879
`system-type` ... 1100
`system-users` ... 1104

T

`t` ... 3
`t` input stream ... 332
`t` output stream ... 335
tab (ASCII character) ... 11
tab deletion ... 785
`tab-always-indent` ... 808
`tab-bar` prefix key ... 405
`tab-line` prefix key ... 405
`tab-line-format`, a window parameter ... 690
`tab-prefix-map` ... 428
`tab-stop-list` ... 811
`tab-to-tab-stop` ... 811
`tab-width` ... 1079
`TAB` in minibuffer ... 346
tabs stops for indentation ... 811
Tabulated List mode ... 474
`tabulated-list-clear-all-tags` ... 477
`tabulated-list-delete-entry` ... 476
`tabulated-list-entries` ... 475
`tabulated-list-format` ... 475
`tabulated-list-get-entry` ... 476
`tabulated-list-get-id` ... 476
`tabulated-list-gui-sort-indicator-asc` ... 474
`tabulated-list-gui-sort-indicator-desc` ... 475
`tabulated-list-header-overlay-p` ... 477
`tabulated-list-init-header` ... 476
`tabulated-list-mode` ... 474
`tabulated-list-padding` ... 477
`tabulated-list-print` ... 476
`tabulated-list-printer` ... 475
`tabulated-list-put-tag` ... 477
`tabulated-list-revert-hook` ... 475
`tabulated-list-set-col` ... 477
`tabulated-list-sort-key` ... 476
`tabulated-list-tty-sort-indicator-asc` ... 475
`tabulated-list-tty-sort-indicator-desc` ... 475
`tabulated-list-use-header-line` ... 477
tag on run time stack ... 163
`tag`, customization keyword ... 250
`tan` ... 49
TCP ... 964
`temacs` ... 1175
`temp-buffer-max-height` ... 998
`temp-buffer-max-width` ... 998
`temp-buffer-resize-mode` ... 998
`temp-buffer-setup-hook` ... 997
`temp-buffer-show-function` ... 997
`temp-buffer-show-hook` ... 997
`temp-buffer-window-setup-hook` ... 998
`temp-buffer-window-show-hook` ... 998
`TEMP` environment variable ... 562
temporary buffer display ... 996
temporary display ... 996
temporary file on a remote host ... 563
temporary files ... 561
`temporary-file-directory` ... 562, 563
`term-file-aliases` ... 1095
`term-file-prefix` ... 1095
`TERM` environment variable ... 1095
Termcap ... 1095
terminal ... 696
terminal input ... 1118
terminal input modes ... 1118
terminal output ... 1119
terminal parameters ... 729
terminal screen ... 608
terminal type ... 26
`terminal-coding-system` ... 878
`terminal-list` ... 698
`terminal-live-p` ... 696
terminal-local variables ... 699
`terminal-name` ... 698
`terminal-parameter` ... 729
`terminal-parameters` ... 729
terminal-specific initialization ... 1095
termscript file ... 1120
`terpri` ... 338
`test-completion` ... 352
`testcover-mark-all` ... 329
`testcover-next-mark` ... 329
`testcover-start` ... 329
testing types ... 30
text ... 777
text area ... 705
text area of a window ... 613
text comparison ... 56
text conversion of coding system ... 868
text deletion ... 784
text height of a frame ... 709
text insertion ... 781
text near point ... 777
text parsing ... 906
text properties ... 813
text properties in files ... 573
text properties in the mode line ... 492
text properties, changing ... 814
text properties, examining ... 813
text properties, read syntax ... 21
text properties, searching ... 817
text representation ... 852
text size of a frame ... 709
text terminal ... 696
text width of a frame ... 709
`text-char-description` ... 525
`text-mode` ... 472
`text-mode-abbrev-table` ... 927
`text-properties-at` ... 814
`text-property-any` ... 818
`text-property-default-nonsticky` ... 827
`text-property-not-all` ... 818
`text-property-search-backward` ... 819
`text-property-search-forward` ... 818
`text-quoting-style` ... 523, 524
text-terminal focus notification ... 734
textual order ... 145

textual scrolling 675
thing-at-point 780
this-command 385
this-command-keys 386
this-command-keys-shift-translated 405
this-command-keys-vector 386
this-original-command 386
thread backtrace 933
thread list 932
thread--blocker 930
thread-join 929
thread-last-error 930
thread-list-refresh-seconds 932
thread-live-p 930
thread-name 930
thread-signal 930
thread-yield 930
threadp 929
threads 929
three-step-help 529
throw 163
throw example 417
thunk 143
thunk-delay 143
thunk-force 143
thunk-let 143
thunk-let* 144
tiled windows 608
time calculations 1113
time conversion 1106
time formatting 1109
time of day 1104
time parsing 1109
time value 1105
time zone rule 1106
time zone rules 1105
time zone, current 1106
time-add 1113
time-convert 1106
time-equal-p 1113
time-less-p 1113
time-subtract 1113
time-to-day-in-year 1114
time-to-days 1114
timer-max-repeats 1115
timerp 1114
timers 1114
timespec 1194
timestamp of a mouse event 401
timestamp, Lisp 1104
timing programs 330
tips for writing Lisp 1162
title bar 703
title, a frame parameter 715
TLS network connections 965
TMP environment variable 562
TMPDIR environment variable 562
toggle-enable-multibyte-characters 854
tool bar 455
tool-bar-add-item 456
tool-bar-add-item-from-menu 456
tool-bar-border 457
tool-bar-button-margin 457
tool-bar-button-relief 457
tool-bar-lines, a frame parameter 720
tool-bar-local-item-from-menu 457
tool-bar-map 456
tool-bar-position, a frame parameter 720
tooltip face 1084
tooltip for help strings 821
tooltip frames 1084
tooltip-event-buffer 1085
tooltip-frame-parameters 1084
tooltip-functions 1085
tooltip-help-tips 1084
tooltip-mode 1084
tooltips 1084
top frame 738
top position ratio 716
top, a frame parameter 716
top-level 418
top-level default value 196
top-level form 269
top-level frame 697
top-visible, a frame parameter 723
total height of a window 613
total pixel height of a window 614
total pixel width of a window 615
total width of a window 614
tq-close 964
tq-create 963
tq-enqueue 964
trace buffer 317
tracing Lisp programs 296
track-mouse 742
tracking frame size changes 711
trailing blanks in file names 540
transaction queue 963
transcendental functions 49
transient keymap 434
transient-mark-mode 774
translate-region 834
translating input events 408
translation keymap 442
translation tables 863
translation-table-for-input 864
transparency, frame 727
transpose-regions 835
trash 552, 568
tray notifications, MS-Windows 1126
triple-click events 393
true 3
true list 71
truename (of file) 544
truncate 42
truncate-lines 982

truncate-partial-width-windows ... 982
truncate-string-ellipsis ... 1008
truncate-string-to-width ... 1007
truth value ... 2
try-completion ... 350
tty-color-alist ... 750
tty-color-approximate ... 750
tty-color-clear ... 750
tty-color-define ... 750
tty-color-mode, a frame parameter ... 727
tty-color-translate ... 750
tty-erase-char ... 1103
tty-setup-hook ... 1095
tty-top-frame ... 738
turn multibyte support on or off ... 854
two's complement ... 37
type ... 8
type (button property) ... 1069
type checking ... 29
type checking internals ... 1186
type predicates ... 30
type, defcustom keyword ... 257
type-of ... 32
type_of ... 1200
typographic conventions ... 2
TZ, environment variable ... 1105

U

UBA ... 1085
UDP ... 964
UID ... 1104
umask ... 553
unassigned character codepoints ... 857
unbalanced parentheses ... 328
unbinding keys ... 444
unbury-buffer ... 602
undecided coding system ... 865
undecided coding-system, when encoding ... 876
undecorated, a frame parameter ... 725
undefined ... 437
undefined in keymap ... 435
undefined key ... 423
underline-minimum-offset ... 1015
underlined text ... 1013
undo avoidance ... 833
undo-ask-before-discard ... 797
undo-auto-amalgamate ... 795
undo-auto-current-boundary-timer ... 795
undo-boundary ... 795
undo-in-progress ... 795
undo-limit ... 796
undo-outer-limit ... 796
undo-strong-limit ... 796
unexec ... 1175, 1177
unhandled-file-name-directory ... 572
unibyte buffers, and bidi reordering ... 1086
unibyte text ... 852
unibyte-char-to-multibyte ... 855
unibyte-string ... 853
Unicode ... 852
unicode bidirectional algorithm ... 1085
unicode character escape ... 12
unicode general category ... 857
unicode, a charset ... 861
unicode-category-table ... 861
unintern ... 128
uninterned symbol ... 126
unique file names ... 561
universal-argument ... 416
universal-argument-map ... 1222
unless ... 147
unload-feature ... 282
unload-feature-special-hooks ... 283
unloading packages ... 282
unloading packages, preparing for ... 1163
unlock-buffer ... 540
unmapped frame ... 736
unnumbered group ... 890
unpacking ... 977
unread-command-events ... 410
unsafep ... 240
unsplittable, a frame parameter ... 721
unused lexical variable ... 190
unwind-protect ... 172
unwinding ... 172
up-list ... 762
upcase ... 67
upcase-initials ... 68
upcase-region ... 812
upcase-word ... 812
update-directory-autoloads ... 276
update-file-autoloads ... 276
upper case ... 67
upper case key sequence ... 405
uptime of Emacs ... 1113
use time of window ... 633
use-empty-active-region ... 776
use-global-map ... 432
use-hard-newlines ... 799
use-local-map ... 432
use-region-p ... 776
user errors, signaling ... 166
user groups ... 1104
user identification ... 1103
user options, how to define ... 253
user pointer object ... 284
user pointer, using in module functions ... 1199
user signals ... 396
user-defined error ... 171
user-emacs-directory ... 1094
user-error ... 166
user-full-name ... 1103, 1104
user-init-file ... 1094
user-login-name ... 1103, 1104
user-mail-address ... 1103

`user-position`, a frame parameter 717
user-ptr object 284
`user-ptrp` 284
`user-real-login-name` 1103, 1104
`user-real-uid` 1104
`user-size`, a frame parameter 718
`user-uid` 1104
`utf-8-emacs` coding system 865

V

valid windows 608
validity of coding system 868
value cell 124
value of expression 133
value of function 209
`values` 142
variable 174
variable aliases 204
variable definition 178
variable descriptions 5
variable limit error 177
variable watchpoints 184
variable with constant value 174
variable write debugging 299
variable, buffer-local 190
`variable-documentation` property 519
variable-width spaces 1044
variant coding system 865
`vc-mode` 489
`vc-prefix-map` 428
`vc-responsible-backend` 545
`vconcat` 110
`vec_get` 1195
`vec_set` 1195
`vec_size` 1195
`vector` 109
vector (type) 109
vector evaluation 134
vector length 95
vector reverse 97
vector to list 105
`vector-cells-consed` 1183
vector-like objects, storage 1178
`vectorp` 109
`verify-visited-file-modtime` 597
version number (in file name) 555
`version`, customization keyword 252
`version-control` 581
vertical combination 610
vertical fractional scrolling 679
vertical scroll position 679
vertical tab 11
`vertical-line` prefix key 405
`vertical-lineto` 1059
`vertical-motion` 760
`vertical-scroll-bar` 1041
`vertical-scroll-bar` prefix key 405
`vertical-scroll-bars`, a frame parameter 720
view part, model/view/controller 1072
`view-register` 835
virtual buffers 606
`visibility`, a frame parameter 723
visible frame 736
`visible-bell` 1083
`visible-frame-list` 731
visited file 594
visited file mode 468
`visited-file-modtime` 598
visiting files 530
visiting files, functions for 530
visual order 1085
visual order, preserve when copying bidirectional text 1089
visual-order cursor motion 1087
void function 135
void function cell 226
void variable 177
`void-function` 225
`void-text-area-pointer` 746
`void-variable` error 177

W

`w32-collate-ignore-punctuation` 57
`w32-notification-close` 1127
`w32-notification-notify` 1126
`wait-for-wm`, a frame parameter 724
waiting 412
waiting for command key input 411
`waiting-for-user-input-p` 961
`walk-windows` 634
`warn` 990
warning options 991
warning type 989
warning variables 990
`warning-fill-column` 991
`warning-fill-prefix` 991
`warning-levels` 990
`warning-minimum-level` 991
`warning-minimum-log-level` 991
`warning-prefix-function` 990
`warning-series` 990
`warning-suppress-log-types` 991
`warning-suppress-types` 991
`warning-type-format` 991
warnings 989
watch, for filesystem events 1127
watchpoints for Lisp variables 184
webkit browser widget 1066
`wheel-down` event 396
`wheel-up` event 396
`when` 147
where was a symbol defined 281
`where-is-internal` 446
`while` 159

`while-no-input` ... 411
`while-no-input-ignore-events` ... 412
whitespace ... 11
`wholenump` ... 40
`widen` ... 766
widening ... 766
width of a window ... 614
`width`, a frame parameter ... 717
window ... 608
`window` (overlay property) ... 1003
window auto-selection ... 685
window body ... 613
window body height ... 615
window body size ... 616
window body width ... 615
window buffer change ... 691
window change functions ... 691
window combination ... 610
window combination limit ... 629
window configuration (Edebug) ... 319
window configuration change ... 693
window configurations ... 686
window dividers ... 1042
window end position ... 672
window excursions ... 765
window header line ... 493
window height ... 613
window history ... 659
window in direction ... 612
window internals ... 1208
window layout in a frame ... 27
window layout, all frames ... 27
window manager interaction, and frame parameters ... 723
window order by time of last use ... 633
window ordering, cyclic ... 633
window parameters ... 688
window pixel height ... 614
window pixel width ... 615
window point ... 671
window point internals ... 1210
window position ... 671, 681
window position on display ... 715
window positions and window managers ... 717
window resizing ... 617
window selected within a frame ... 609
window selection change ... 692
window size ... 613
window size change ... 692
window size on display ... 717
window size, changing ... 617
window splitting ... 622
window start position ... 671
window state ... 687
window state change ... 692
window state change flag ... 694
window that satisfies a predicate ... 635
window top line ... 672
window tree ... 610
window use time ... 633
window width ... 614
`window`, a buffer display action alist entry ... 648
`window-absolute-body-pixel-edges` ... 684
`window-absolute-pixel-edges` ... 684
`window-absolute-pixel-position` ... 684
`window-adjust-process-window-size-function` ... 955
`window-at` ... 682
`window-at-side-p` ... 612
`window-atom`, a window parameter ... 690
`window-atom-root` ... 669
`window-body-edges` ... 682
`window-body-height` ... 615
`window-body-pixel-edges` ... 683
`window-body-size` ... 616
`window-body-width` ... 615
`window-bottom-divider-width` ... 1043
`window-buffer` ... 635
`window-buffer-change-functions` ... 692
`window-child` ... 611
`window-combination-limit` ... 628, 629
`window-combination-resize` ... 630
`window-combined-p` ... 611
`window-configuration-change-hook` ... 693
`window-configuration-frame` ... 687
`window-configuration-p` ... 687
`window-current-scroll-bars` ... 1041
`window-dedicated-p` ... 661
`window-display-table` ... 1080
`window-edges` ... 682
`window-end` ... 672
`window-font-height` ... 1034
`window-font-width` ... 1034
`window-frame` ... 609
`window-fringes` ... 1035
`window-full-height-p` ... 615
`window-full-width-p` ... 615
`window-group-end` ... 672
`window-group-end-function` ... 672
`window-group-start` ... 672
`window-group-start-function` ... 672
`window-header-line-height` ... 493, 616
`window-height`, a buffer display action alist entry ... 647
`window-hscroll` ... 681
`window-id`, a frame parameter ... 724
`window-in-direction` ... 612
`window-largest-empty-rectangle` ... 684
`window-left-child` ... 611
`window-line-height` ... 675
`window-lines-pixel-dimensions` ... 1009
`window-list` ... 609
`window-live-p` ... 608
`window-main-window` ... 666
`window-make-atom` ... 670
`window-margins` ... 1048

window-max-chars-per-line ... 616
window-min-height ... 616, 617
window-min-height, a buffer display action alist entry ... 647
window-min-size ... 617
window-min-width ... 616, 617
window-minibuffer-p ... 371
window-mode-line-height ... 616
window-next-buffers ... 659
window-next-sibling ... 611
window-old-body-pixel-height ... 695
window-old-body-pixel-width ... 694
window-old-buffer ... 694
window-old-pixel-height ... 694
window-old-pixel-width ... 694
window-parameter ... 688
window-parameters ... 688
window-parameters, a buffer display action alist entry ... 646
window-parent ... 610
window-persistent-parameters ... 688
window-pixel-edges ... 683
window-pixel-height ... 614
window-pixel-width ... 615
window-point ... 671
window-point-insertion-type ... 671
window-preserve-size ... 622
window-preserved-size ... 622
window-preserved-size, a window parameter ... 690
window-prev-buffers ... 659
window-prev-sibling ... 611
window-resizable ... 618
window-resize ... 618
window-resize-pixelwise ... 619
window-right-divider-width ... 1043
window-scroll-bar-height ... 1041
window-scroll-bar-width ... 1041
window-scroll-bars ... 1041
window-scroll-functions ... 691
window-selection-change-functions ... 692
window-setup-hook ... 1094
window-side, a window parameter ... 690
window-sides-reversed ... 666
window-sides-slots ... 666
window-sides-vertical ... 665
window-size-change-functions ... 692
window-size-fixed ... 621
window-slot, a window parameter ... 690
window-start ... 672
window-state-change-functions ... 692
window-state-change-hook ... 693
window-state-get ... 687
window-state-put ... 687
window-swap-states ... 688
window-system ... 1083, 1084
window-system-initialization ... 1090
window-text-pixel-size ... 1008
window-toggle-side-windows ... 666
window-top-child ... 611
window-total-height ... 614
window-total-size ... 614
window-total-width ... 614
window-tree ... 613
window-use-time ... 633
window-valid-p ... 609
window-vscroll ... 679
window-width, a buffer display action alist entry ... 647
windowp ... 608
windows, controlling precisely ... 635
windows, recombining ... 626
with-case-table ... 70
with-coding-priority ... 875
with-connection-local-variables ... 203
with-current-buffer ... 592
with-current-buffer-window ... 998
with-demoted-errors ... 170
with-displayed-buffer-window ... 998
with-eval-after-load ... 283
with-file-modes ... 553
with-help-window ... 528
with-local-quit ... 414
with-mutex ... 931
with-no-warnings ... 291
with-output-to-string ... 339
with-output-to-temp-buffer ... 996
with-selected-window ... 632
with-silent-modifications ... 597, 816
with-suppressed-warnings ... 291
with-syntax-table ... 912
with-temp-buffer ... 592
with-temp-buffer-window ... 997
with-temp-file ... 539
with-temp-message ... 985
with-timeout ... 1116
word-search-backward ... 883
word-search-backward-lax ... 883
word-search-forward ... 882
word-search-forward-lax ... 882
word-search-regexp ... 882
words in region ... 759
words-include-escapes ... 757
wrap-prefix ... 983
write-abbrev-file ... 925
write-char ... 338
write-contents-functions ... 536
write-file ... 535
write-file-functions ... 535
write-region ... 538, 554
write-region-annotate-functions ... 577
write-region-inhibit-fsync ... 539
write-region-post-annotation-function ... 577
writing a documentation string ... 519
writing buffer display actions ... 655
writing emacs modules ... 1188

writing Emacs primitives 1184
writing module functions 1191
writing to files 538
wrong-number-of-arguments 212
wrong-type-argument 29

X

x-alt-keysym 1121
x-alternatives-map 1222
x-bitmap-file-path 1015
x-close-connection 700
x-color-defined-p 748
x-color-values 749
x-ctrl-keysym 1121
x-defined-colors 749
x-display-color-p 752
x-display-list 700
x-dnd-known-types 748
x-dnd-test-function 748
x-dnd-types-alist 748
x-double-buffered-p 737
x-family-fonts 1027
x-focus-frame 733
x-get-resource 750
x-gtk-use-system-tooltips 1084
x-hyper-keysym 1121
x-list-fonts 1027
x-meta-keysym 1121
x-open-connection 700
x-parse-geometry 728
x-pointer-shape 746
x-popup-dialog 745
x-popup-menu 744
x-resource-class 751
x-resource-name 751
x-sensitive-text-pointer-shape 746
x-server-vendor 754
x-server-version 754
x-setup-function-keys 1222
x-stretch-cursor 726
x-super-keysym 1121
X display names 699
X Window System 1083
X11 keysyms 1121
XBM 1053
xdigit character class, regexp 887
XML DOM 842
xor 149
XPM 1054
xwidget 1066
xwidget-buffer 1067
xwidget-info 1068
xwidget-plist 1067
xwidget-query-on-exit-flag 1068
xwidget-resize 1067
xwidget-size-request 1067
xwidget-webkit-execute-script 1067
xwidget-webkit-execute-script-rv 1067
xwidget-webkit-get-title 1067
xwidget-webkit-goto-uri 1067
xwidgetp 1067

Y

y-or-n-p 366
y-or-n-p-with-timeout 367
yank 790
yank suppression 440
yank-excluded-properties 790
yank-handled-properties 790
yank-pop 791
yank-undo-function 791
yanking and text properties 790
yes-or-no questions 366
yes-or-no-p 367

Z

z-group, a frame parameter 717
Z-order 737
zerop 40
zlib-available-p 837
zlib-decompress-region 837