PIMOS マニュアル
(第 3.0 版)

— 操作編 —

ICOT PIMOS 開発グループ
1991 年 10 月

(財) 新世代コンピュータ技術開発機構

はじめに

このマニュアルは並列推論マシンのためのオペレーティングシステム PIMOS 3.0 版の使い方を解説するものです. PIMOS の機能の充実にともなって説明すべきことがらも増えてきたことから, マニュアルは「操作編」と「プログラミング編」の二分冊となっています. 本編 (操作編) では, PIMOS を利用する際の操作法について解説します. プログラムインタフェースについてはプログラミング編を御覧ください. なお, ページ番号は両編を通して振られており, 目次, 索引も全体を通したものが両編共に付けられています.

PIMOS は並列論理型言語 KL1 で記述されており, PIMOS の下で動く利用者のプログラムも KL1 で記述することを前提としています. KL1 は Flat GHC を基礎にさまざまな拡張を施した言語です. プログラミング編には KL1 言語の Flat GHC に対する拡張部分の仕様の概略を収めてありますが, KL1 によるプログラミング技法については, 別途 KL1 プログラミングの教科書である『KL1 プログラミング 入門編 / 初級編 / 中級編』 (ICOT 第 4 研究室) などで勉強されるのが適当です.

目次

1 ログイン **1**

1.1 SIMPOS のユーザ環境の準備 1
1.2 ログイン手順 1
1.2.1 通常のログイン手順 1
1.2.1.1 ユーザ環境初期化ファイル 3
1.2.2 簡易ログイン機能利用時のログイン方法 3
1.2.3 環境変数の設定 3
1.2.4 パスワードの変更 4
1.3 ログインに関するメッセージ 4
1.3.1 CSP に表示されるエラー 4
1.3.2 ログイン要求ウインドウ / ログイン区分を選択するためのウインドウに表示されるエラー 5

2 シェル **7**

2.1 シェル 7
2.1.1 概要 7
2.1.2 シェルの起動方法 8
2.1.2.1 通常のシェルの起動方法 8
2.1.2.2 ログイン用シェルの起動方法 9
2.1.3 シェルの基本操作 9
2.1.3.1 コマンド一般規則 9
2.1.3.2 ユーザゴールの起動 10
2.1.4 シェルやジョブに対するコマンド 11
2.1.4.1 ジョブ管理用のコマンド 12
2.1.4.2 アテンションキー 12
2.1.4.3 シェルに対するコマンド 12
2.1.4.4 子シェルを起動するコマンド 13
2.1.4.5 コマンドヒストリを操作するコマンド 13
2.1.4.6 プロンプトを操作するコマンド 14
2.1.4.7 カレントディレクトリを操作するコマンド 14
2.1.4.8 印字パラメタを操作するコマンド 15
2.1.4.9 コマンド別名を操作するコマンド 16
2.1.4.10 コマンド行を逐次実行するためのコマンド 16
2.1.5 シェルが表示するメッセージ 17
2.1.5.1 情報メッセージ 17
2.1.5.2 エラーメッセージ 17
2.1.5.3 附帯メッセージ 20
2.1.6 エラー処理 21
2.1.7 資源不足時の処理 22

2.1.8 ジョブを実行するノードの指定 . . . 22
2.1.9 ジョブの実行優先度範囲の指定 . . . 23
2.1.10 環境変数の操作 . . . 23
2.1.11 シェルが使用する環境変数 . . . 23
2.1.12 コマンドファイル . . . 24
2.1.13 初期化ファイル . . . 25
2.1.14 ログインファイル . . . 25
2.1.15 ログインファイルと初期化ファイルの例 . . . 25
2.2 シェルユーティリティプログラム . . . 26
2.2.1 GC の起動 . . . 26
2.2.2 状態表示 . . . 27
2.2.2.1 資源木の表示 . . . 27
2.2.2.2 タスク木の表示 . . . 28
2.2.3 PIMOS の終了 . . . 28
2.2.4 ユーザ管理 . . . 28
2.2.4.1 ログインユーザの表示 . . . 28
2.2.4.2 特権ユーザへの移行 . . . 29
2.2.4.3 パスワードの変更 . . . 29
2.2.5 コピー . . . 29
2.2.6 モジュール使用方法の表示 . . . 29
2.2.7 文字列およびテキストファイルに対するコマンド . . . 30
2.2.7.1 エコー . . . 30
2.2.7.2 文字列の検索 . . . 30
2.2.7.3 行数のカウント . . . 31
2.2.8 ディレクトリの表示 . . . 31
2.2.9 変数チェッカ . . . 31

3 コンパイラ **33**
3.1 セルフコンパイラ . . . 33
3.1.1 基本的事項の説明 . . . 34
3.1.2 バッチコンパイル . . . 35
3.1.3 インタラクティブコンパイル . . . 35
3.1.3.1 コマンドシンタックス . . . 36
3.1.3.2 リンクの順序 . . . 36
3.1.3.3 インタラクティブコンパイルの終了 . . . 36
3.1.3.4 インタラクティブコンパイルのコマンド . . . 37
3.1.4 環境変数 . . . 46
3.1.5 PIMOS が用意するプリプロセッサ . . . 49
3.2 再リンカ . . . 50
3.2.1 再リンカの使用方法 . . . 51
3.3 アンローダ . . . 52
3.4 ローダ . . . 53
3.5 パッケージ . . . 53
3.6 モジュール . . . 54

4 デバッギングツール **56**
4.1 リスナ . . . 57
4.1.1 機能概要 . . . 57
4.1.2 操作方法 . . . 58

4.1.2.1 起動 . . . 58
4.1.2.2 環境変数 . . . 59
4.1.2.3 デフォルトパッケージの取り扱い . . . 60
4.1.2.4 ゴールのマクロ展開 . . . 60
4.1.2.5 ゴールの起動 . . . 61
4.1.2.6 ゴール列の再表示と再実行 . . . 64
4.1.2.7 実行の中断 . . . 65
4.1.2.8 コマンドファイル . . . 70
4.1.2.9 初期実行ファイル . . . 70
4.1.2.10 プログラムの実行に関する情報の獲得 . . . 70
4.1.2.11 終了 . . . 70
4.1.3 変数プール . . . 71
4.1.3.1 変数の表記法 . . . 71
4.1.3.2 変数の格納方法 . . . 71
4.1.3.3 格納されている変数の一覧 . . . 72
4.1.3.4 変数の解放 . . . 72
4.1.3.5 実行終了時の変数の値の表示 . . . 73
4.1.3.6 実行に使用した変数に対する問い合わせ . . . 73
4.1.4 標準入出力 . . . 73
4.1.5 デッドロック検出 . . . 74
4.1.5.1 概要 . . . 74
4.1.5.2 デッドロックゴールの検出機能 . . . 74
4.1.5.3 デッドロックゴールの種類とその出力例 . . . 75
4.1.6 コマンド . . . 80
4.1.6.1 基本コマンド . . . 81
4.1.6.2 リスナウインドウ操作コマンド . . . 88
4.1.6.3 環境コマンド . . . 88
4.1.6.4 環境変数の操作 . . . 93
4.1.6.5 変数プール操作コマンド . . . 94
4.1.6.6 ディレクトリコマンド . . . 98
4.1.6.7 ライブラリコマンド . . . 101
4.1.6.8 デバッグコマンド . . . 102
4.1.6.9 ParaGraph 関連コマンド . . . 108
4.1.6.10 ユーティリティコマンド . . . 111
4.1.6.11 シェルユーティリティプログラム . . . 112
4.1.7 トレーサ . . . 113
4.1.7.1 概要 . . . 113
4.1.7.2 トレースの開始と終了 . . . 115
4.1.7.3 トレース出力 . . . 115
4.1.7.4 デフォルトのトレースモード . . . 118
4.1.7.5 クローズ番号の表示 . . . 119
4.1.7.6 自動トレースと自動スパイ . . . 119
4.1.7.7 マルチウインドウを利用したトレース . . . 120
4.1.7.8 タスク (荘園) 内トレース . . . 120
4.1.7.9 トレースコマンド . . . 121
4.1.7.10 トレーサのコマンド一覧 . . . 138
4.1.8 エラー処理 . . . 138
4.1.8.1 エラー報告時に入力できるコマンド . . . 138
4.1.8.2 エラーの再表示 . . . 144

4.1.8.3 エラーメッセージ一覧 . . . 144
4.2 インスペクタ . . . 152
4.2.1 機能概要 . . . 152
4.2.2 操作方法 . . . 152
4.2.2.1 起動 . . . 152
4.2.2.2 コマンドの入力 . . . 153
4.2.2.3 変数プール . . . 153
4.2.2.4 ウインドウへの表示に関するパラメタ . . . 153
4.2.2.5 入出力先の切り換え . . . 154
4.2.2.6 出力の中断 . . . 155
4.2.2.7 終了 . . . 155
4.2.2.8 インスペクタの環境変数 . . . 155
4.2.3 データの表示形式と解析 / 表示 / 監視位置の指定方法 . . . 156
4.2.3.1 データの表示形式 . . . 156
4.2.3.2 解析 / 表示 / 監視位置の指定方法 . . . 159
4.2.4 コマンド . . . 161
4.2.4.1 基本コマンド . . . 161
4.2.4.2 調査コマンド . . . 163
4.2.4.3 監視コマンド . . . 167
4.2.4.4 環境コマンド . . . 170
4.2.4.5 変数プール操作コマンド . . . 172
4.2.5 データの監視と表示 . . . 173
4.2.5.1 概要 . . . 173
4.2.5.2 操作方法 . . . 174
4.2.5.3 コマンド . . . 177
4.2.6 インスペクタ, モニタコマンド一覧 . . . 180

5 チューニングツール 181
5.1 パフォーマンスメータ . . . 181
5.1.1 機能概要 . . . 181
5.1.2 操作方法 . . . 181
5.1.3 環境変数 . . . 183
5.2 ParaGraph . . . 185
5.2.1 機能概要 . . . 185
5.2.2 実行状況情報の収集 . . . 186
5.2.2.1 ノードの稼働状況の計測 . . . 186
5.2.2.2 ノード間通信の発生状況の計測 . . . 186
5.2.2.3 プログラムの実行状況の計測 . . . 186
5.2.2.4 収集時間間隔の変更 . . . 187
5.2.2.5 データ格納用プール . . . 187
5.2.3 操作方法 . . . 187
5.2.3.1 ParaGraph の起動 . . . 187
5.2.3.2 View selector の操作方法 . . . 187
5.2.3.3 What manipulator の操作方法 . . . 192
5.2.3.4 History manipulator の操作方法 . . . 193
5.2.3.5 ウインドウインタフェース . . . 194
5.2.4 コマンド . . . 195
5.2.4.1 View selector 操作コマンド . . . 195
5.2.4.2 What manipulator 操作コマンド . . . 199

5.2.4.3 History manipulator 操作コマンド 201

6 システムの管理 203
6.1 IPL の設定 203
6.1.1 概要 203
6.1.2 IPL でロードするファイルの設定 203
6.1.3 ユーザプログラムの IPL でのロード 204
6.1.4 IPL の異常 204
6.1.5 CSP による限定トレースのかけ方 204
6.2 起動とシャットダウン 205
6.2.1 起動方法 205
6.2.2 シャットダウン方法 206
6.3 タスク環境・ユーザ定義サーバの設定 206
6.3.1 タスク環境の設定方法 206
6.3.2 リリース時のタスク環境ファイルの内容 207
6.4 ログインハンドラ 207
6.4.1 ログインハンドラの起動 207
6.4.1.1 ログイン要求受付パラメータ 207
6.4.1.2 ログイン区分パラメータ 208
6.4.1.3 利用記録パラメータ 209
6.4.2 ログイン用初期化ファイル 209
6.4.2.1 ユーザ定義ファイル 209
6.4.2.2 グループ定義ファイル 210
6.4.2.3 リリース時のログイン用初期化ファイルの内容 210
6.5 システム定義サーバの登録方法 211
6.5.1 システム定義サーバの外部仕様 211
6.5.2 システム定義サーバの登録 211
6.5.3 PIMOS 起動時に関するエラーメッセージ 212
6.5.4 CSP に表示されるエラー 212

A ネットワーク環境の設定 215
A.1 TCP/IP 215
A.2 LIA-net 216

B PIMOS のバージョンを知るには 218
B.1 CSP に表示するバージョン 218
B.2 バージョンを獲得するためのユーティリティ 219
B.3 FEP プログラムのバージョン 219

7 KL1 の言語仕様 221
7.1 入出力モード表示の読み方 222
7.2 シンタクス 223
7.2.1 モジュール構造 223
7.2.1.1 モジュールの定義 223
7.2.1.2 モジュール間に渡る呼出し 223
7.2.2 非対称な節の扱い 224
7.2.2.1 節の優先関係指定 (alternatively) 224
7.2.2.2 否定の略記 (otherwise) 225
7.2.3 ガード内 OR 225
7.2.4 その他の指示語 225

7.2.4.1 オペレータ指定指示語 . . . 226
7.2.4.2 ファイル読み込み指示語 . . . 226
7.2.5 データの表記 . . . 226
7.2.6 実行優先度指定 . . . 228
7.2.6.1 所属荘園内割合指定 . . . 229
7.2.6.2 所属荘園内自己相対指定 . . . 229
7.2.7 ノード指定 . . . 229
7.3 マクロ記法 . . . 232
7.3.1 概要 . . . 232
7.3.2 マクロ展開の抑制 . . . 232
7.3.3 システム定義のマクロ . . . 233
7.3.3.1 定数記述 . . . 233
7.3.3.2 ユニフィケーション . . . 234
7.3.3.3 数式マクロ . . . 234
7.3.3.4 条件分岐 . . . 236
7.3.3.5 暗黙の引数 . . . 237
7.3.3.6 条件コンパイル機能 . . . 242
7.3.4 ユーザ定義のマクロ . . . 245
7.3.4.1 マクロ展開機能の利用手順 . . . 246
7.3.4.2 マクロ展開の順序 . . . 247
7.3.4.3 マクロ展開規則 . . . 247
7.3.4.4 マクロモジュールの構成 . . . 254
7.3.4.5 KL1 プログラムへのコンパイル . . . 256
7.3.4.6 旧仕様マクロの取り扱い . . . 258
7.3.4.7 マクロ関連ツール . . . 258
7.3.4.8 マクロ定義の例 . . . 259
7.3.5 旧仕様マクロ . . . 262
7.4 組込述語 . . . 264
7.4.1 タイプチェック . . . 264
7.4.2 diff . . . 266
7.4.3 整数の比較 . . . 266
7.4.4 整数の演算 . . . 267
7.4.5 浮動小数点数の比較 . . . 270
7.4.6 浮動小数点数の演算 . . . 271
7.4.7 整数と浮動小数点数の変換 . . . 272
7.4.8 整数の倍精度演算 . . . 273
7.4.9 多倍長整数の比較 . . . 273
7.4.10 多倍長整数演算 . . . 274
7.4.11 ベクタ関係 . . . 274
7.4.12 ストリングの比較 . . . 275
7.4.13 ストリング操作 . . . 276
7.4.14 ストリングの要素の検索 . . . 277
7.4.15 汎用ストリング演算 . . . 278
7.4.16 ストリングマクロユーティリティ . . . 279
7.4.17 アトム関係 . . . 283
7.4.18 コード / モジュール関係 . . . 283
7.4.19 ストリームサポート . . . 285
7.4.20 特殊入出力 . . . 285
7.4.21 その他 . . . 286

7.4.22 参照数管理について 288
7.5 荘園機能 290
7.5.1 概要 290
7.5.2 荘園の生成 290
7.5.3 制御ストリーム 291
7.5.4 報告ストリーム 292
7.5.5 例外の生起 300
7.5.6 例外タグ 301
7.5.7 例外処理用ユーティリティ 302
7.5.7.1 述語情報の取り出し 302
7.5.7.2 荘園プロファイル情報変換ユーティリティ 302

8 ユーティリティ 303
8.1 比較 303
8.1.1 標準の比較ルーチン 303
8.1.2 標準比較ルーチンでのデータの大小関係 304
8.1.3 比較ルーチンの指定方法 304
8.2 ハッシング 305
8.3 プール 305
8.3.1 Bag 306
8.3.2 Stack 307
8.3.3 Queue 307
8.3.4 Sorted bag 308
8.3.5 Keyed bag 309
8.3.6 Keyed set 311
8.3.7 Keyed sorted bag 312
8.3.8 Keyed sorted set 313
8.3.9 キーなし及びただひとつのキー付きプールが表示するエラーメッセージ 314
8.4 複数キー付きのプール (multiply keyed bag) 315
8.4.1 概要 315
8.4.2 使用方法 316
8.4.2.1 要素の検索 317
8.4.2.2 要素の部分取り出し / 部分参照の指定 318
8.4.2.3 属性の指定方法 318
8.4.2.4 メッセージプロトコル 318
8.4.3 例 322
8.4.4 Multiply keyed bag が表示するエラーメッセージ 323
8.5 プールへのメッセージ一覧 327
8.6 ソーティング用フィルタ 328
8.6.1 キーなしのソーティング用フィルタ 328
8.6.2 キー付きのソーティング用フィルタ 328
8.7 マージング用フィルタ 329
8.7.1 キーなしのマージング用フィルタ 329
8.7.2 キー付きのマージング用フィルタ 329
8.8 ソーティング / マージング用フィルタが表示するエラーメッセージ 330

9 入出力管理 332
9.1 入出力資源の獲得 . . . 333
9.1.1 まず, ジェネラルリクエストデバイスの獲得 . . . 333
9.1.2 次に, リクエストデバイスの獲得 . . . 334
9.1.3 そして, デバイスの獲得 . . . 335
9.1.4 入出力をする . . . 336
9.1.5 入出力ユーティリティのフィルタをつけて入出力をする . . . 336
9.2 入出力ユーティリティ . . . 338
9.2.1 バッファ . . . 338
9.2.1.1 共通プロトコル . . . 338
9.2.1.2 入力用フィルタ . . . 344
9.2.1.3 出力用フィルタ . . . 345
9.2.1.4 入出力両用フィルタ . . . 345
9.2.1.5 無限長バッファ . . . 346
9.2.1.6 オペレータプール . . . 346
9.2.1.7 変数プール . . . 348
9.2.2 通常の項と wrap された項 . . . 348
9.2.2.1 Wrap . . . 349
9.2.2.2 Unwrap . . . 350
9.2.3 タイマに関するユーティリティ . . . 350
9.2.3.1 日付に関するユーティリティ . . . 351
9.2.3.2 時刻に関するユーティリティ . . . 352
9.2.3.3 日付と時刻の両方を扱うユーティリティ . . . 353
9.2.3.4 時間単位変換ユーティリティ . . . 353
9.3 入出力制御 . . . 354
9.3.1 デバイスの階層 . . . 354
9.3.2 デバイスとの通信方式 . . . 355
9.3.3 デバイスのアボート / アテンション処理 . . . 356
9.3.3.1 ウインドウデバイスのアボート処理 . . . 357
9.3.4 入出力デバイスのデバイス名 . . . 358
9.3.5 ウインドウ . . . 359
9.3.5.1 ウインドウリクエストデバイス . . . 359
9.3.5.2 ウインドウ . . . 360
9.3.6 ファイルとディレクトリ . . . 363
9.3.6.1 ファイルリクエストデバイス . . . 363
9.3.6.2 パス名の解釈 . . . 365
9.3.6.3 ファイル . . . 365
9.3.6.4 ディレクトリ . . . 367
9.3.7 タイマ . . . 368
9.3.7.1 タイマリクエストデバイス . . . 368
9.3.7.2 タイマ . . . 369
9.3.8 ストリング入出力デバイス . . . 371
9.3.8.1 ストリング入出力リクエストデバイス . . . 371
9.3.8.2 ストリング入出力デバイス . . . 371
9.3.9 デバイスへのメッセージ一覧 . . . 373
9.4 標準入出力 . . . 375
9.4.1 標準入出力機能とは . . . 375
9.4.1.1 資源木との関係 . . . 375
9.4.1.2 入出力ストリームの共有と排他制御 . . . 376

9.4.1.3 入出力ストリームの割り当てと切り替え . . . 376
9.4.1.4 入出力ストリームを使った入出力動作 . . . 376
9.4.2 標準入出力を利用して入出力を行う方法 . . . 376
9.4.2.1 アテンションの制御を行わない場合 (一般のプログラム) . . . 377
9.4.2.2 アテンションの制御を行う場合 (特殊なプログラム) . . . 377
9.4.2.3 旧版との互換性 . . . 379
9.4.3 標準入出力をタスクに割り当てそれを管理するには . . . 379
9.4.3.1 タスクに標準入出力を割り当てる方法 . . . 379
9.4.3.2 タスクに割り当てた標準入出力を制御する方法 . . . 381
9.4.3.3 標準入出力の制御 . . . 381
9.4.4 標準メッセージ出力 . . . 383
9.4.5 標準入出力を使ったプログラムの作成例 . . . 383

10 プログラムの管理 385
10.1 アトムの管理 . . . 385
10.1.1 アトムキャッシュデータベース . . . 385
10.1.1.1 生成 . . . 385
10.1.1.2 述語によるアクセス . . . 385
10.1.2 アトムデバイス . . . 386
10.1.2.1 生成 . . . 386
10.1.2.2 プロトコル . . . 386
10.2 モジュールの管理 . . . 386
10.2.1 モジュールキャッシュデータベース . . . 386
10.2.1.1 生成 . . . 387
10.2.1.2 プロトコル . . . 387
10.2.1.3 ダイナミックリンク . . . 390
10.2.2 モジュールデバイス . . . 390
10.2.2.1 生成 . . . 391
10.2.2.2 プロトコル . . . 391
10.2.2.3 ユーティリティ . . . 393
10.2.3 パッケージ . . . 393
10.2.3.1 継承 . . . 393
10.2.3.2 組込パッケージ . . . 394
10.2.3.3 ユーティリティ . . . 394
10.3 ライブラリ . . . 394
10.3.1 コンパイラの呼び出し . . . 395
10.3.2 再リンカの呼び出し . . . 395
10.3.3 ローダの呼び出し . . . 395
10.3.4 アンローダの呼び出し . . . 396
10.4 プログラミング例 . . . 396

11 資源管理・実行管理 398
11.1 資源の獲得と管理 . . . 399
11.1.1 種類 . . . 399
11.1.2 資源サービス表 . . . 400
11.1.3 I/O ノード . . . 400
11.1.4 タスク . . . 400
11.1.5 標準入出力資源 . . . 401
11.1.6 資源木 . . . 401

11.1.7 環境変数 . . . 401
11.1.8 システム定義サーバが提供する資源の獲得階層 . . . 402
11.1.9 ジェネラルリクエストデバイス . . . 403
11.1.9.1 獲得法 . . . 403
11.1.9.2 プロトコル . . . 403
11.1.10 標準入出力資源の獲得 . . . 409
11.1.10.1 獲得方法 . . . 409
11.1.11 資源の保護モード . . . 410
11.1.11.1 資源の持ち主 . . . 410
11.1.11.2 保護モード . . . 410
11.2 タスク . . . 410
11.2.1 タスク制御ストリームへのメッセージ . . . 410
11.2.2 タスク報告ストリームからのメッセージ . . . 412
11.3 サーバの実現方法 . . . 412
11.3.1 クライアントサーバ機能の概要 . . . 412
11.3.1.1 クラスインスタンス . . . 413
11.3.1.2 保護の機能 . . . 413
11.3.2 利用方法 . . . 414
11.3.2.1 サービスの登録 (サーバ側) . . . 414
11.3.2.2 資源の獲得 (クライアント側) . . . 415
11.3.2.3 資源要求の受付開始 (サーバ側) . . . 416
11.3.2.4 子資源の生成 . . . 418
11.3.3 クライアント・サーバ機能による資源の提供と獲得 . . . 418
11.3.3.1 サービスの登録方法 . . . 418
11.3.3.2 サービス要求の受付 . . . 420
11.3.3.3 ジェネラルリクエストデバイスからの要求 . . . 421
11.3.3.4 標準入出力資源の生成要求 . . . 422
11.3.3.5 サーバが獲得するユーザ情報 . . . 423
11.3.3.6 クライアントタスクへの資源の利用登録 . . . 423
11.3.3.7 資源の獲得 . . . 424
11.3.4 一般のサーバに標準入出力機能を持たせる方法 . . . 424
11.4 ユーザ情報の管理 . . . 424
11.4.1 ユーザとグループの識別子 . . . 424
11.4.2 ユーザ情報のデータベース . . . 425
11.4.2.1 ユーザリクエストデバイス . . . 425
11.4.2.2 ユーザデバイス . . . 426
11.4.2.3 グループデバイス . . . 427
11.4.2.4 ログインデバイス . . . 428
11.5 標準に提供するエラー報告方法 . . . 428
11.5.1 エラー報告の形式 . . . 428

C FEP 上のストリング入出力デバイスの作成方法 430

D プリプロセッサの作成方法 435
D.1 概要 . . . 435
D.2 プリプロセッサの機構 . . . 435
D.3 パーサ, プリプロセッサ, アンパーサ, ベリファイアに共通な述語 . . . 438
D.4 パーサモジュールの作成方法 . . . 439
D.5 プリプロセッサモジュールの作成方法 . . . 439

D.6 アンパーサモジュールの作成方法 440
D.7 ベリファイアモジュールの作成方法 440
D.8 プリプロセッサ作成ツール 441
D.8.1 KL1 標準プリプロセッサ 441
D.8.2 コンパイラオプション格納プール 441
D.8.3 ライブラリメッセージ出力 443
D.9 プログラム例 444

E プロトコル定義 445
E.1 概要 445
E.2 プロトコル定義言語 445
E.2.1 文法 445
E.2.2 システム定義プロトコル 445
E.2.3 文法の説明 446
E.2.3.1 プロトコル定義 446
E.2.3.2 継承 446
E.2.3.3 符号 446
E.2.3.4 パラメータ付きプロトコル定義 446
E.2.3.5 エラー情報指定 446
E.3 プロトコル定義の読み方 446
E.4 プロトコル定義 449

F サーバプログラムの記述例 468
F.1 概要 468
F.2 サーバプログラム例 469

図目次

5.1 rmonitor の表示例 182
5.2 amonitor の出力画面例 183
5.3 View selector の画面例 187
5.4 What × When の表示例 188
5.5 Overall What × When の表示例 189
5.6 What × Where の表示例 190
5.7 Overall What × Where の表示例 190
5.8 When × Where の表示例 191
5.9 Result の表示例 191

A.1 ネットワークの設定に関連するファイル 217

7.1 実行優先度の算出方式 230
7.2 例外タグのビット位置割り当て 301

11.1 システム定義サーバが提供する資源の階層 402
11.2 サービスの登録 414
11.3 資源の獲得 415
11.4 資源要求の受付開始 417
11.5 子資源の生成 419

表目次

3.1 デフォルトの別名 45
3.2 !keep コマンドで保存されるコンパイルオプション 46
3.3 コマンドと環境変数の関係およびデフォルトの設定値 47
3.4 環境変数と参照しているプログラム 50

4.1 トレーサのコマンド一覧 139
4.2 リスナのエラーメッセージ一覧 145
4.3 インスペクタ, モニタコマンド一覧 180

7.1 KL1 のデータ表記例 228
7.2 算術比較マクロ 235
7.3 算術演算マクロ 236
7.4 汎用ストリング演算表 280
7.5 例外タグのビット位置 301

8.1 プールへのメッセージ一覧 327

9.1 標準の演算子 347
9.2 PIMOS のデバイスへのメッセージ一覧 373

E.1 メッセージプロトコルと送り先デバイスの対応 449

第 1 章

ログイン

この章ではユーザが SIMPOS から PIMOS を利用するためのログインの手順について述べる. なお PIMOS 自身の立ち上げ方法については第 6 章 (208 頁) を参照されたい.

1.1 SIMPOS のユーザ環境の準備

SIMPOS のログインファイル me:login.com のシステムメニュー項目に以下を追加する.

```
{"Login PIMOS", pimfep##pimos_remote_login_requester}
```

1.2 ログイン手順

PIMOS では PIMOS のタスク環境ファイル (6.3.1 節タスク環境の設定方法 (211 頁) を参照のこと) の設定により 以下の 2 つのログイン手順が存在する.

- 通常のログイン
- 簡易ログイン機能を用いたログイン

それぞれ説明する.

1.2.1 通常のログイン手順

通常のログイン機能が起動された場合のログイン手順である. ログインするログイン区分等, ユーザがログイン時に入力するパラメータがいくつかある.

1. CSP 上に, 既にログイン可能状態であることを表すメッセージ

   ```
   DC> **** xxx LOGIN SERVICE STARTS
   DC> ( INTo) 00 0000000X
   ```

   が表示されていることを確認する.

   【注意】xxx はログインに利用できる LAN の種類, X はログインサービス受付可能な FEP 番号である.

2. SIMPOS の システムメニュー の "Login PIMOS" を選択する.

3. ログイン要求ウインドウが表示されるので, 次の入力を行う.

(a) ログインサービスを受け付ける適当な FEP の I/O ノード名 (プログラミング編 11.1節 資源の獲得 (399 頁) を参照) を入力する. もし FEP 上の SIMPOS からログインするならば local と入力する.

(b) PIMOS のユーザ名を入力する.

(c) 対応するパスワードを入力する.

4. ユーザ名およびパスワードを PIMOS が確認する. 確認できなかった場合にはその旨が表示されログインに失敗する.

5. PIMOS が各ログイン区分におけるログイン状況を例えば

```
Part:     Mode      :Node No.:  User name (LID)
================================================
0   :  available    :0-31    :
1   : not available :32-47   : daigo (100)
2   : (multi user)  :48-63   : masahiro (102)
```

と表示する. 各項目の意味は以下の通り.

Part: ログイン区分番号

Mode: ログイン可能性

available ログイン可能

not available ログイン不可

(multi user) マルチユーザ利用. 絶えずログイン可能

Node No: 利用ノード範囲

User name: もし利用しているユーザがいればそのユーザ名

LID: User Name に対応するログイン ID

【注意】ログイン開始時に指定した FEP からログインできない区分であった場合には, ユーザの有無に関わらずログイン不可となる.

6. (a) ログインできる区分が存在する場合

例えば前述の例の場合には,

```
select your partition (0 2 exit kill) ?
```

と表示されるのでユーザはログインするためにはログイン区分番号 ('0' '2'), PIMOS 利用を終了するためには 'e' , ログイン中のプログラムを中止させるために 'k' をそれぞれ入力する. 'k' を選択した場合には具体的にプログラムを選択するために例えば

```
enter your partition ?1↩
enter your LID ?2↩
```

と区分番号とログイン ID を入力する.

(b) 利用できる区分がない場合

次のような表示がある.

```
Confirm(<return> or kill) :
```

そこでもし PIMOS の利用を中止するのならばそのまま <return> キーを入力し, さもなければ 'k' を入力して前述の処理を行う.

7. 区分選択処理終了後, 通常はシェルを起動してシェルウインドウを表示する. 起動する初期プログラムの設定法については第 6.4.2 節 (214 頁) を参照されたい.

1.2.1.1 ユーザ環境初期化ファイル

ユーザは, 上述のような利用区分の選択を対話的に行うために PIMOS が最初に表示するウインドウを利用する. ユーザはこのウインドウをカストマイズする情報をユーザ環境初期化ファイルに記述することが可能である. ファイル名はログイン要求をした I/O ノード上のホームディレクトリ下の `pimos.init` である.

カストマイズ情報は, 現在のところ次のパラメータのみである.

```
console_window(Pos, Size, Font).
```

Pos, Size ウインドウデバイスのプロトコル reshape/3 (プログラミング編 9.3 入出力制御 (354 頁) を参照) の 1 番目と 2 番目の引数と同様である.

Font フォントファイルを示すストリングである.

1.2.2 簡易ログイン機能利用時のログイン方法

簡易ログイン機能利用時には, 複数人数で全ノードを利用可能である.

1. `CSP` 上に

   ```
   DC> **** xxx LOGIN STARTS
   DC> ( INTo) 00 0000000X
   ```

   と表示されていることを確認する.

   【注意】xxx はログインに利用できる LAN の種類, X はログインサービス受付可能な FEP 番号である.

2. 次に SIMPOS システムメニューの `"Login PIMOS"` を選択する.

3. ログイン要求ウインドウが表示されるので, 次の手順でログイン手続きを行う.

   (a) ログインサービスを受け付ける適当な FEP の I/O ノード名 (プログラミング編 11.1節 資源の獲得 (399 頁) を参照) を入力する. もし FEP 上の SIMPOS からログインするならば local と入力する.

   (b) PIMOS のユーザ名を入力する.

   (c) 対応するパスワードを入力する.

4. ログインの成功もしくは失敗の旨が表示されログイン処理を終了する.

5. ログインに成功した場合, 通常はシェルを起動してシェルウインドウを表示する. 起動するプログラムの設定法については第 6 章 (208 頁) を参照されたい.

1.2.3 環境変数の設定

ログイン時に次の環境変数を設定する.

task:package

パッケージ名アトム.

task:directory

初期ディレクトリ名ストリング.

task:home

ホームディレクトリ名ストリング.

task:io_node

ログイン要求を行なった I/O ノード名.

task:fep

ログイン要求を受け付けた FEP 番号.

task:user

ログインユーザ名ストリング

【注意】ログイン直後のパッケージ名はログインしたユーザ名である.

1.2.4 パスワードの変更

パスワードを変更手順を説明する.

【注意】初めて PIMOS にログインした時には必ずパスワードの変更を行なうこと.

1. シェルからコマンド change_password を実行する.
2. パスワード変更用ウインドウが表示されるので, その指示に従って新しいパスワードを設定する.
3. ログインシェルを終了させる. するとファイルが書換えらる.

【注意】ユーザ定義ファイル (pimos:pimos.logins) を書き換える権利をもつ SIMPOS のユーザ (通常, システム管理者) で FEP にログインすること.

1.3 ログインに関するメッセージ

この章ではログイン時に出力される可能性のあるメッセージについて説明する. なお, ネットワーク設定に関しては, 付録 A をそれぞれ参照されたい.

1.3.1 CSP に表示されるエラー

1. someone has erased a login information

   原因 特権ユーザがログイン情報を消してしまった.

   対処 特になし.

2. CANNOT Make a remote device

   原因 ログインしたユーザの利用している I/O ノード上にデバイスが作成できない.

   - ユーザの利用している I/O ノード上のネットワークが FEP に対して不正状況にある.
   - SIMPOS のパッケージ mpsicsp がリリース時と異なる状況にある.

   対処 ネットワークもしくは SIMPOS のパッケージ mpsicsp を正常状態にする.

1.3.2 ログイン要求ウインドウ / ログイン区分を選択するためのウインドウに表示されるエラー

1. Unknown FEP node name

   原因 指定された FEP のノード名が不正または不明である.

   対処
   - 正しい FEP のノード名を入力する.
   - ネットワーク情報に修正する (付録 A 参照).

2. Network Error

   原因 ネットワークを通じてログインサーバ (ログイン処理用内部デバイス) へアクセスする際に, エラーが発生した.
   - ログインサーバのアドレスが,SIMPOS のネットワークシステムに登録されていない.
   - ログインサーバが, 指定した FEP 上に起動されていない.
   - FEP が立ち上がっていない.
   - ネットワークの設定が正しく行われていないため, 指定した FEP 上のログインサーバにアクセスできない.
   - ネットワークの障害のため, 指定した FEP 上のログインサーバにアクセスできない.

   対処
   - ネットワークの状態, 及び, 設定をチェックする.
   - タスク環境ファイル `pimos:pimos.conf` で, リモートからのログイン要求を受け付けるように設定する.

3. Receive timeout

   原因 PIMOS のログインサーバからのメッセージの受信待ちでタイムアウトとなった.

   対処 マルチ PSI または FEP の負荷が大きく, PIMOS がログインの要求に対応できない可能性がある. したがって, マルチ PSI, FEP の負荷を調べる.

4. Login incorrect

   原因 ユーザ名もしくはパスワードに誤りがある.

   対処 正しいユーザ名およびパスワードを入力する.

5. Input Error!!

   原因 ログイン区分選択時等で発生する.
   - 入力プロンプトで表示されている括弧内にあるログイン区分番号もしくは exit/kill 以外を入力した.

6. Your ID has been erased!!

   原因 特権ユーザがログイン処理中にユーザを登録から抹消した.

   対処 特権ユーザに再登録を依頼する.

7. Someone has logged on PIMOS

原因 シングルユーザモードのログイン区分にログインする場合，ログイン処理中に別のユーザがログインをしてしまった．

対処 ログインしてしまったユーザにログアウトすることを依頼する．

8. Your initial program code doesn't exist

原因 ユーザのログイン初期プログラムのプログラムコードが存在しない．

対処 特権ユーザにそのプログラムモジュールをモジュールデータベースに登録してもらう．

第 2 章

シェル

2.1 シェル

2.1.1 概要

シェルは, ユーザが指定したゴールを起動し, またその実行を管理するためのユーザインタフェースの一つである. 以下に, その機能を概説する.

タスクの生成, ゴールの実行

シェルは, ユーザがシェルウインドウを介してインタラクティブに指定したゴールを起動する. その際, 新たにタスクを生成し, そのタスク内にゴールを挿入して実行する.

標準入出力ストリーム

シェルがユーザタスク (ユーザゴール) を起動する際には, タスクに対し, **標準入力**, **標準出力**, **標準メッセージ出力**の三つのデバイス (ストリーム) を割り当てる. これらのデバイスは, ユーザがゴールと共に指定することが可能であるが, 特に指定がなければ, シェル自身に与えられている各ストリーム (通常は, シェルウインドウへのストリーム) を与える.

また, それ以外に**標準インタラクション**と呼ばれるストリームも与える. このストリームは, 「シェルウインドウを持ち, かつユーザタスクの祖先タスク中でユーザタスクの最も近くにあるシェル (通常は, ユーザタスクを起動したシェル) 」のシェルウインドウにつながれている. すなわち標準インタラクションストリームは, いずれかのシェルのシェルウインドウにつながれることが保証されている.

標準入力, **標準出力**, **標準インタラクション**は, 標準入出力機能 (プログラミング編 9.4 章, 標準入出力 (375) 参照) を使って実現している. 標準メッセージ出力は, シェルが独自に提供している機能である.

タスク間通信機能 (パイプ)

あるタスクの出力を他のタスクの入力として, タスク同士を連結したい場合がある. そのような場合には, シェルが提供するタスク間通信機能 (パイプ) を利用してタスクの標準入出力間を連結し, タスク間で通信を行うことができる.

パイプにはバッファリング機能がある. パイプ内のバッファが満杯になると, パイプ前段のタスク (生産者) は, そのバッファが空くまでの間, ある適当な期間だけ実行を停止させられる.

また, バッファが満杯であるにも関わらず, 消費される見込みのないデータを生産者が送出した場合には, パイプで連結されたすべてのタスクを強制的に終了する. これをパイプブロークン状態と呼ぶ.

ジョブ

パイプを用いて連結された一連のタスク群 (ユーザゴール群) をジョブと呼ぶ. パイプを使わない場合には, 一つのユーザゴールが一つのジョブとなる. シェルは, 自身が起動したユーザタスクを個々のタスクごとに管理するのではなく, このジョブを単位として管理する.

なお, ここで言う管理とは, ジョブの実行状況の監視あるいはジョブの起動, 停止, 再開, 終了などのジョブの状態変更, さらにはジョブの状態問い合わせなどをいう.

シェルウインドウの共有

シェルは通常, ユーザとのインタラクションを行うためにウインドウ (シェルウインドウ) を持っている. しかし, このウインドウは, 標準入出力にファイルやシェルウインドウ以外のウインドウを指定しないで起動したユーザゴールからも標準入出力として利用されるため, 複数のタスクが競合して使用する可能性がある. そのためシェルは, シェル自身と起動したジョブとの間で, シェルウインドウの排他制御を行う.

フォアグラウンドとバックグラウンド

シェルは, ジョブの実行環境として, フォアグラウンドとバックグラウンドと呼ぶ二種類の環境を提供し, シェルウインドウの排他制御を行っている.

バックグラウンドには, ジョブを複数個投入できるが, フォアグラウンドには, 一つのジョブだけが存在し得る.

シェルとフォアグラウンドに投入されたジョブとは, 排他的にシェルウインドウを利用する. バックグラウンドのジョブは, シェルウインドウに対するアクセスを常に抑制されている. 仮にバックグラウンドジョブが入出力メッセージをシェルウインドウに向けて送出したとしても, そのメッセージが実際にシェルウインドウに届くのは, そのジョブがフォアグラウンドになったときである.

ただし, 標準メッセージ出力のみは例外で, たとえバックグラウンドジョブであってもシェルウインドウにメッセージを表示することができる.

シェルコマンド

バックグラウンドのジョブをフォアグラウンドに移行させたり, ジョブの停止や再開を行うため, ジョブ管理用のコマンドをいくつか用意している.

また, フォアグラウンドのジョブをバックグラウンドに移行させるための機構も必要である. これは, ウインドウから送られてくるアテンションを検出することで実現している.

2.1.2 シェルの起動方法

2.1.2.1 通常のシェルの起動方法

シェルの起動方法には, 以下の三通りの方法がある. これらの内のいずれかのゴールを実行すると, 自動的にシェルウインドウが作成され, プロンプトが表示される. プロンプトが表示されれば, シェルは各種のコマンドを受け付けられる状態になる.

通常の場合 shell:go

ウインドウの位置およびサイズは, マウスで指定する. フォントは, SIMPOS のデフォルトのフォントファイルを使用する.

ウインドウ位置とサイズを指定 shell:go(at(X, Y), char(W, H))

ウインドウの位置およびサイズは, 指定したものを用いる. フォントは, デフォルトのフォントファイルを使用する.

ウインドウ位置, サイズ, フォントを指定 shell:go(at(X, Y), char(W, H), Font)
ウインドウの位置, サイズおよびフォントをすべて指定する.

ゴール中の各引数の意味は, 以下の通りである.

X ウインドウの左上端の位置 (X 座標) をドット単位で表わしたもの.

Y ウインドウの左上端の位置 (Y 座標) をドット単位で表わしたもの.
X, Y を個別に指定するかわりに, at(mouse) とすれば, 位置をマウスで指定することができる.

W ウインドウの一行あたりの文字数.

H ウインドウの行数.
W, H を個別に指定するかわりに, char(mouse) とすれば, サイズをマウスで指定することができる.

Font 表示に使用するフォントファイルのパス名 (文字列).

【注意 1】位置の指定に誤りが含まれていた場合 (負数, 型の不一致) には, 位置はマウスによる指定となる. サイズの指定についても同様である. また, 指定した位置およびサイズでは, ウインドウがディスプレイをはみだしてしまう場合には, デフォルトの位置およびサイズのウインドウが作られる. フォントファイルの指定が文字列でなかったり, 指定したフォントファイルが存在しない場合には, デフォルトのフォントファイルを使用する.

【注意 2】ウインドウは環境変数 `task:io_node` で指定したリモート上に生成される. 環境変数 `task:io_node` については, プログラミング編第 11.1.3 節 (400 頁) I/O ノードを参照されたい.

2.1.2.2 ログイン用シェルの起動方法

ログイン用シェルの起動方法にも前項 2.1.2.1 で説明した通常のシェルと同様に, 以下の三通りの方法がある. これらの内のいずれかのゴールを実行すると, 自動的にシェルウインドウが作成され, プロンプトが表示される. プロンプトが表示されれば, シェルは各種のコマンドを受け付けられる状態になる.

通常の場合 shell:login
ウインドウの位置およびサイズは, マウスで指定する. フォントは, デフォルトのフォントファイルを使用する.

ウインドウ位置とサイズを指定 shell:login(at(X, Y), char(W, H))
ウインドウの位置およびサイズは, 指定したものを用いる. フォントは, デフォルトのフォントファイルを使用する.

ウインドウ位置, サイズ, フォントを指定 shell:login(at(X, Y), char(W, H), Font)
ウインドウの位置, サイズおよびフォントをすべて指定する.

【注意】ログイン用シェルは, ユーザが PIMOS にログインした時に最初に起動するゴールとして利用するのが通常の使用方法である.

2.1.3 シェルの基本操作

2.1.3.1 コマンド一般規則

シェルのコマンドには, ユーザゴール起動のためのコマンドとシェルやジョブに対するコマンド (シェルコマンド) の二種類がある. ユーザゴール起動のためのコマンドは, 次項 2.1.3.2 (15頁) で述べる文法に従って, 一行中に複数個記述することができる. 一方, シェルやジョブに対するコマンドを一行中に複数個記述することはできない. また, コマンド行中に変数を記述することはできない.

2.1.3.2 ユーザゴールの起動

コマンド行では, 以下の文法に従って, 複数のユーザゴールを指定することができる. 指定された各ゴールは, 各々独立のタスク中で実行される.

```
<command line>   ::=  <job> [ "&" [ <command line> ] ]
<job>            ::=  <task group> [ "=>" <std_out> ]
<task group>     ::=  <task> [ "<=" <std_in> ] { "|" <task> }
<task>           ::=  <command> [ "->" <std_mes> ]
```

- <std_in>, <std_out>, <std_mes> の各デバイスは, 以下のように指定する.

  ファイル: `file("ファイルパス名")`

  ファイル: `file(^"ファイルパス名")`

  ウインドウ: `window("ウインドウ名")`

  ウインドウ: `window("ウインドウ名", at(X, Y), char(W, H))`

  ウインドウ: `window("ウインドウ名", at(X, Y), char(W, H), Font)`

  【注意 1】ファイルパス名に ^ を付けずに指定すると, そのファイルは出力 (write) モードで, ^ を付けて指定すると追加 (append) モードでオープンされる. モードについては, プログラミング編 ファイルリクエストデバイス (第 9.3.6.1 節 (363 頁)) を参照されたい. ウインドウ名は, 作成するウインドウに付けるタイトルであって, ウインドウに対応する PMACS のテキストバッファ名ではない. それ以外の各引数の意味は, シェルの起動方法の説明 (第 2.1.2.1項 (13頁)) を参照されたい.

- <command> は, プログラム中でのモジュール呼び出しと同様に package::module:predicate(Arg, ...) の形で指定する.

  パッケージ名と '::' は省略することができる. 省略した場合, 環境変数 `task:package` の値を使用する. `task:package` が指定されていないか, あるいはアトム以外のデータが格納されている場合には, パッケージ名は `'work'` となる.

  【例 1】引数がない場合.

  ```
  > test:start
  ```

  【例 2】引数がある場合.

  ```
  > test:start(arg1, arg2)
  ```

  ':' と述語名を省略することもできる. 省略した場合, 述語名として go が指定されたものとみなす.

- <task> には, 標準メッセージ出力デバイスを一つだけ指定することができる. 指定がない場合はシェルの標準メッセージ出力を共用する.

  【例 1】シェルの標準メッセージ出力を使用する.

  ```
  > test:start
  ```

  【例 2】標準メッセージ出力としてファイルを指定する.

```
> test:start -> file("mes.doc")
```

- <task group> が複数の <task> で構成される場合, 各 <task> 間の標準入出力は, パイプでつながれる.

【例 1】二つのタスクをパイプでつなぐ.

```
> test1:start | test2:start
```

【例 2】二つのタスクをパイプでつなぎ, 各々に標準メッセージ出力を指定する.

```
> test1:start -> file("mes1.doc") | test2:start -> file("mes2.doc")
```

- <job> には, 標準入力デバイスと標準出力デバイスをそれぞれ一つずつ指定することができる. 指定がない場合には, シェルウインドウがつながれる.

【例 1】標準入力, 標準出力, 標準メッセージ出力すべてを指定する.

```
> test:start -> file("mes.doc") <= window("in") => window("out")
```

【例 2】標準入力, 標準出力の指定は, 各々ジョブに対して一つずつ.

```
> test1:start <= window("in") | test2:start => file("out.doc")
```

- <command line> が単一の <job> である場合には, その <job> は & の有無に従って, フォアあるいはバックグラウンドで起動される.

【例 1】フォアグラウンドに投入される.

```
> test:start
```

【例 2】バックグラウンドに投入される.

```
> test:start &
```

- <command line> が複数の <job> で構成される場合, 各 <job> は並列実行される. その場合, 各 <job> の内で最後に記述されたものが, 指定にしたがってフォアまたはバックで起動され, それ以外は, すべてバックに投入される.

【例 1】test3:start のみがフォアで, それ以外はバックとなる.

```
> test1:start & test2:start & test3:start
```

【例 2】すべてのジョブがバックとなる.

```
> test1:start & test2:start & test3:start &
```

2.1.4 シェルやジョブに対するコマンド

シェルは, 起動したジョブやシェルに対する指示を与えるため, 次のようなコマンドを用意している.

2.1.4.1 ジョブ管理用のコマンド

stop JobNo

JobNo で指定したジョブを停止する.

fore JobNo

JobNo で指定したジョブの実行をフォアグラウンドで再開する. ただしそのジョブが資源不足で停止していた場合には, あらかじめ定められた資源量を追加した後に, 実行を再開する. 詳しくは, 資源不足時の処理 (第 2.1.7 項 (27頁)) を参照されたい. また資源 (計算機資源) については, プログラミング編荘園機能 (第 7.5 節 (290頁)) を参照されたい.

back JobNo

JobNo で指定したジョブの実行をバックグラウンドで再開する. ただしそのジョブが資源不足で停止していた場合には, あらかじめ定められた資源量を追加した後に, 実行を再開する. 詳しくは, 資源不足時の処理 (第 2.1.7 項 (27頁)) を参照されたい. また資源 (計算機資源) については, プログラミング編荘園機能 (第 7.5 節 (290頁)) を参照されたい.

kill JobNo

JobNo で指定したジョブを強制終了する. JobNo として all を指定すると, 現在実行中のすべてのジョブを強制終了する.

status

そのシェルが起動した全ジョブの情報を表示する. 表示される情報の内容は, JobNo (整数) , ジョブの実行状態, ジョブの名称 (ユーザゴールの名称) の三つである. ジョブの実行状態には次の種類がある.

running 実行中である.

stopped 停止中である. 実行を再開するためには, 本項で述べたコマンド fore または back を実行する必要がある.

rsc_low 資源不足のため停止中である. 実行を再開するためには, 本項で述べたコマンド fore または back を実行する必要がある.

terminate 終了処理を開始した. まもなく終了する.

zombi 終了処理中である. まもなく終了する.

abnormal ジョブが異常事態に陥った. この状態は通常は起こり得ない.

上記のコマンドのうちで JobNo を指定するものに関して, JobNo を省略した場合には, その時点において実行中のジョブの中で, 最後に起動したジョブの JobNo を指定したものとみなす.

2.1.4.2 アテンションキー

フォアグラウンドで実行中のジョブをバックグラウンドに移行させたいときは, アテンションキー (通常の設定では Control-C) を押下する. この場合ジョブは, バックグラウンドに移行すると共に停止する.

2.1.4.3 シェルに対するコマンド

exit

シェルを終了する. その際, 起動したジョブは, すべて強制的に終了させる.

suspend

シェル (以下の説明では親シェルと呼ぶ) の中から子シェル (次項 2.1.4.4 (18 頁)) を呼びだした場合に, 子シェルを一時的に停止させ, 親シェルに制御を戻したければ, このコマンドを実行する.

get

シェルが動作するにあたって必要なパラメタを保存するため, シェル変数と呼ぶ特殊な変数をシェルは用意している. これらすべてのシェル変数について, 現在の設定値を表示する.

help

コマンド一覧を表示する.

2.1.4.4 子シェルを起動するコマンド

shell

shell(at(X, Y), char(W, H))

shell(at(X, Y), char(W, H), Font)

新たなシェルウインドウを作り, 別のシェルを起動する. 引数の意味は, シェルの起動方法 (第 2.1.2.1 項 (13頁)) を参照されたい.

sh

シェルの中からシェル (子シェル) を再帰的に呼び出す. この場合には新たなシェルウインドウは生成しない.

2.1.4.5 コマンドヒストリを操作するコマンド

history

history N

現在ヒストリスタックに格納されているコマンド行を一覧表示する. 数値 N を指定した場合には, その行数だけ表示する. 格納している行数より指定した行数の方が多い場合には, すべての行を表示する. また負数や整数以外のデータを指定した場合にもすべてを表示する.

イベント番号

ヒストリスタックに格納したコマンドには, 必ず一連番号 (イベント番号) が付与される. そのイベント番号を指定することにより, 対応するコマンド行を取り出して実行することができる. ただし, 0 の場合には直前のコマンドを実行する.

get hl

ヒストリスタックのスタック長を表示する.

set hl = Length

ヒストリスタックの長さを設定する. Length は, 非負整数あるいは infinite. Length が 0 の場合には, ヒストリスタックへのコマンド格納を行わない. シェル起動時のデフォルト値は, 0 である.

!!

!! 文字列

ヒストリスタックに格納されているコマンド行の中で, 最も新しいものを取り出し, 既入力文字列としてコマンド行に表示する. ただし, ヒストリスタックの長さが 0 の場合には, エラーとなる. また !! に続けて文字列を記述することもできる. その場合には, !! を展開した後に文字列を結合し, 既入力文字列としてコマンド行に表示する.

【注意】ここでは文字列は, ダブルクォートで囲まない.

! 文字列 1

! 文字列 1 文字列 2

ヒストリスタックに格納されているコマンド行の中で, コマンドの先頭部分が文字列 1 と一致する行を探しだし, それを既入力文字列としてコマンド行に表示する. 該当する行が複数存在する場合には, 最も新しいものが優先的に選択される. 該当するコマンド行が存在しない場合には, エラーとなる. また文字列 1 に続けて空白および文字列 2 を記述することもできる. その場合には, 文字列 1 を展開した後に文字列 2 を結合し, 既入力文字列としてコマンド行に表示する.

【注意】ここでは文字列 1, 文字列 2 は, ダブルクォートで囲まない.

! イベント番号

! イベント番号 文字列

ヒストリスタックに格納されているコマンド行の中から, イベント番号で指定した行を探しだし, それを既入力文字列としてコマンド行に表示する. 該当するコマンド行が存在しない場合には, 空白行となる. またイベント番号に続けて空白および文字列を記述することもできる. その場合には, イベント番号による展開を行った後に文字列を結合し, 既入力文字列としてコマンド行に表示する. イベント番号が 0 の場合には直前のコマンドを指定したものとみなす.

【注意】ここでは文字列は, ダブルクォートで囲まない.

2.1.4.6 プロンプトを操作するコマンド

set prompt = Prompt

コマンド入力行のプロンプトを変更する. Prompt は, 文字列あるいはアトムで指定する. 文字列中に \! を記述しておくと, その位置にイベント番号を表示する.

get prompt

現在設定されているプロンプトを表示する.

2.1.4.7 カレントディレクトリを操作するコマンド

cd Directory

環境変数 task:directory の値を設定する. 変更する際には, 指定したディレクトリの存在確認を行う. 指定したディレクトリが存在しなければ, 環境変数の値は変更しない.
ディレクトリの指定は, > をデリミタとし, ~, .. およびサブディレクトリ名アトムを組み合わせて行う. ~ は, その時点における環境変数 task:home の値を意味する. task:home に値が設定されていな

い場合に ~ を使用しても, 指定したディレクトリが存在しないものとして扱われる. また .. は, 相対的な親ディレクトリを意味する.
先頭が > の場合には, 絶対指定であるとみなし, それ以外の場合には task:directory で指定しているディレクトリからの相対指定とみなす.
ディレクトリは, 文字列で指定することもできる. ただし, その場合には ~ の展開は行われない. ディレクトリが文字列でかつその先頭が > であった場合には, 絶対指定であるとみなす. ディレクトリが文字列でかつその先頭が > でない場合には, task:directory の現在値からの相対指定であるとみなす.

【注意 1】先頭文字が大文字で始まるディレクトリ名アトムやピリオドを含むディレクトリ名アトムを指定する場合には, そのアトムをクオート (') で囲む必要がある.

【注意 2】相対指定を行った時点で task:directory の値が不正な値であった場合には, 当然その相対指定は無効となる.

cd

cd ~ を実行した場合と同じである.

pushd Directory

pushd N

pushd

環境変数 task:directory の値を設定する. 変更する際には, 指定したディレクトリの存在確認を行う. 指定したディレクトリが存在しなければ, 環境変数の値は変更しない.
ディレクトリは, cd コマンドの場合と同様に指定できる.
環境変数の値を変更した場合には, ディレクトリ名をディレクトリスタックの先頭に格納する.
ディレクトリ名の代わりに数値 (正整数) を指定した場合には, スタックの N 番目の要素が先頭になるようにスタックをローテーションし, 先頭になったディレクトリ名を task:directory に設定する. ただしスタックの先頭は, 0 番目から始まる.
また引数を省略した場合には, スタックの先頭要素と 2 番目の要素を入れ替える.

popd

ディレクトリスタックから最後に格納したディレクトリ名を取り出し, その値を環境変数 task:directory に格納する. ただし, 変更の際にはディレクトリの存在確認を行い, ディレクトリが存在しなければ, 環境変数の値は変更しない.
またディレクトリスタックが空の場合には, エラーとなる.

dirs

ディレクトリスタックの内容を一覧表示する.

pwd

環境変数 task:directory の値をシェルの標準出力先 (通常はウインドウ) に出力する.

2.1.4.8 印字パラメタを操作するコマンド

set 印字パラメタ名 = 値

シェルが出力するメッセージを表示する際に用いる印字パラメタを変更する. 指定できる印字パラメタ名には, pd, pl, pal, psl の 4 種類がある. それぞれ入出力バッファの印字パラメタ名 print_depth,

print_length, print_atom_length, print_string_length に対応している．値は，非負整数でなければならない．シェル起動時のデフォルト値はそれぞれ，20, 20, 80, 80 である．

【注意】デフォルト値は，今後の調整で変わる予定である．

get 印字パラメタ名

シェルが出力するメッセージを表示するために用いる印字パラメタを問い合わせ，値をシェルの標準出力先 (通常はウインドウ) に出力する．指定できる印字パラメタ名には，pd, pl, pal, psl の 4 種類がある．それぞれ入出力バッファの印字パラメタ名 print_depth, print_length, print_atom_length, print_string_length に対応している．

2.1.4.9 コマンド別名を操作するコマンド

alias アトム = 文字列

以降入力するコマンドについて，コマンドの先頭部分にあるトークンが上記のアトムと一致する場合に，そのトークンを上記の文字列に置換してコマンドを実行する．1 コマンド行中に複数のコマンドが含まれていた場合には，各コマンドに対して置換が行われる．ただし，多段階にわたる置換は行わない．

【注意 1】アトムには英数字以外の文字を含むアトムを指定してはならない．もしも指定した場合には，置換が正しく行われる保証はない．

【注意 2】文字列には，コマンド行に記述できるすべての文字を含んで良い．ただし，文字列の先頭が ! であったとしても ! の処理は行われないため，文法エラーとなる．

【注意 3】シェルコマンド名をアトムとして指定する場合には，そのアトムを括弧でくくる必要がある．ただし alias と unalias は指定できない．

unalias アトム

指定したアトムに対応するコマンド別名を削除する．

alias アトム

指定したアトムに対応するコマンド別名を表示する．

alias

現在定義されているコマンド別名を一覧表示する．

2.1.4.10 コマンド行を逐次実行するためのコマンド

do [コマンド行, コマンド行, ...]

コマンド行のリスト中に記述した一連のコマンド行を逐次に実行する．各コマンド行は，文字列でなければならない．文字列以外のデータを記述した場合には，そのデータは無視する．

【注意 1】アテンションは，コマンド行リストで指定した個々のコマンド行に対しては有効であるが，do コマンド自体をキャンセルすることは現在のところできない．したがってアテンションを起こした時点で，コマンド行が残っていたら，その残りのコマンド行が解釈・実行される．

【注意 2】このコマンドは，将来改訂される可能性が高い．

2.1.5 シェルが表示するメッセージ

シェルが表示するメッセージには, 各種の情報をユーザに伝えるために出力される情報メッセージと, 何らかのエラーが発生した時にそれをユーザに伝えるために出力されるエラーメッセージとがある. またエラーメッセージには, エラーの詳細情報を知らせるための附帯メッセージが続けて表示される場合もある. 各メッセージは, メッセージ表示を引き起こすような事象が発生した時点で直ちに出力される. 以下ではこれら 3 種類のメッセージについて個別に記述する.

2.1.5.1 情報メッセージ

`Resource-increment in job` *JobNo* `is` *Inc* `.`
JobNo のジョブの資源不足時の資源量の増分を表示.

`The value of` *Key* `is` *Value* `.`
環境変数またはシェル変数の値を表示.

`Non existent value` *Key* `.`
変数値を表示しようとしたが, 指定した名前を持つ環境変数またはシェル変数は存在しない.

`No environment variable .`
すべての環境変数を表示しようとしたが, 環境変数は何も存在しない.

`Resource-increment` *Inc* `added into job` *JobNo* `.`
資源不足のためあらかじめ設定した量の資源を追加した.

`Alias :` *Atom* `=>` *String*
コマンド別名の表示. コマンド *Atom* の別名は *String* である.

`Un-alias :` *Atom*
コマンド *Atom* のコマンド別名を削除した.

`Environment variable removed` *Name*
変数 *Name* を削除した.

LineNo `:` *CommandLine*
ヒストリスタックの内容表示. 行番号が *LineNo* のコマンド行は *CommandLine* である.

JobNo `-->` *Status* *JobName*
ジョブの状態表示. *JobNo* のジョブの名前は *JobName* であり, 現在の状態は *Status* である.

Status: ジョブの実行状態.

running 実行中である.

stopped 停止中である.

rsc_low 資源不足のため停止中である.

terminate 終了処理を開始した.

zombi 終了処理中である.

abnormal ジョブが異常事態に陥った.

2.1.5.2 エラーメッセージ

`Syntax error !!`
入力コマンド行中に文法の誤りがある.

【注意】シェルのコマンド読み込みは, 入出力バッファのプロトコル `getwt/1` を利用している. もしもこのコマンド読み込み時に文法エラーが発見された場合には, 同じく入出力バッファのプロトコル `print_error/1` を利用してエラーメッセージを表示する. したがって, この場合のエラーメッセージは, 入出力バッファが規定するフォーマットによって表示される (プログラミング編 第 9.2 章 (338頁) 入出力ユーティリティ参照).

`Syntax error !!`

`( Variable(s) found in Command-line . )`
入力コマンド行中に文法の誤りがある. (変数名が含まれている)

`Job` *JobNo* `was aborted .`
JobNo のジョブが異常終了した.

`Unable to open take-file` *FileName*
FileName で指定したコマンドファイルが存在しない.

`Can't create the file-device for` *FileName* `.`
FileName で指定したファイルをオープンするために必要なファイルリクエストデバイスが作成できない.

`Illegal Alias-data,` *Atom* `or` *String* `!!`
コマンド別名の指定に誤りがある.

`No alias existed for` *Atom* `!!`
指定したコマンドのコマンド別名は存在しない.

String `can't found in history .`
文字列 *String* で始まるコマンド行は, ヒストリスタックの中にはない.

`Unable to change directory` *Directory*
カレントディレクトリを *Directory* に移すことに失敗した.

`Empty directory-stack .`
ディレクトリスタックが空である.

`Unable to exchange directory .`
ディレクトリスタックを操作することに失敗した.

`Non existent job-number` *JobNo*
ジョブ番号が *JobNo* のジョブは存在しない.

`Illegal variable name` *Name*
シェル変数の変数名が誤っている.

`Illegal value` *Value* `of` *Name*
シェル変数 *Name* に格納しようとした値が誤っている.

`Can't remove the environment variable` *Name*
変数 *Name* は削除できない.

`Can't connect to the node` *IoNode*
リモートノード *IoNode* に接続できなかった.

`Resource-low occured in job` *JobNo* `.`
資源不足が発生した.

Deadlock detected !!
デッドロックが発生した.

Merger-deadlock detected !!
マージャがデッドロックした.

Reduction failure occured !!
ユーザ定義述語がリダクションに失敗した.

Unification failure occured !!
ユニフィケーションに失敗した.

Illegal input occured !!
組込述語の入力引き数に誤りがある.

Range overflow occured !!
組込述語の入力引き数が大き過ぎる.

Out of bounds occured !!
組込述語の入力引き数中に範囲外のものがある.

Integer overflow occured !!
整数演算がオーバフローした.

Integer zero division occured !!
整数演算でゼロ除算が起こった.

Invalid floating-point operation occured !!
浮動小数点演算で誤りが発生した.

Arity mismatch occured !!
ユーザ定義述語の呼出し時に, 引数の数が誤っていた.

Firmware error occured (reserved_7) !!
ファームウエアのエラーが発生した.

Undefined Module Detected !!
起動したコードの中から未定義のモジュールを呼び出そうとした.

Undefined Predicate Detected !!
起動したコードの中から未定義の述語を呼び出そうとした.

Illegal merger input occured !!
マージャの入力引数が誤っている.

Trace exception detected !!
シェルがトレース例外事象を検出した.

Spy exception detected !!
シェルがスパイ例外事象を検出した.

Firmware error occured (reserved_14) !!
ファームウエアのエラーが発生した.

`Special exception detected !!`
特殊な例外事象が発生した.

【注意】このメッセージは, 現在のところファームウエアがサポートしていないため, 決して表示されることはない.

`Illegal situation occured !!`
ジョブの実行になんらかの異常が発生した.

`Unknown exception detected !!`
シェルの知らない例外事象が発生した.

`Error reported !!`
起動したジョブからエラーが報告された.

2.1.5.3 附帯メッセージ

`Job-number` : *JobNo*
ジョブ番号を表示する.

`NODE-number` : *Node*
ノード番号を表示する.

`PE-number` : *PE*
クラスタ内のプロセッサ番号を表示する.

`Predicate` : *Pred*
例外事象が発生した述語を表示する.

`Undefined` : *UndefPred*
未定義であった述語を表示する.

`Builtin` : *Bltn*
例外事象が発生した組込述語を表示する.

`Position` : *Pos*
例外事象の要因となった引数の位置を表示する.

`Called by` : *Caller*
例外事象を発生した組込述語を呼び出していた述語を表示する.

`Collected by` : *Collector*
デッドロックゴールを検出した場合に, それを回収したゴールを表示する.

`Type` : *Type*
デッドロックなどの異常の種類を表示する. なおデッドロックの種類については, 第 4.1.5.3 章を参照されたい.

- [0]Copying GC Copying GC を実行中にデッドロックを検出した.
- [1]Copying GC (in Tracing) Copying GC を実行中にトレースゴールのデッドロックを検出した.
- [2]Copying GC (in Spying) Copying GC を実行中にスパイゴールのデッドロックを検出した.
- [10]MRB-GC (Collect Value) Collect Value を行った時にデッドロックを検出した.

- [11]MRB-GC (Suspend) Suspend を行った時にデッドロックを検出した.
- [12]MRB-GC (Active Unify) Active Unify を行った時にデッドロックを検出した.
- [13]MRB-GC (Suspend) Suspend を行った時にデッドロックを検出した.
- [14]MRB-GC (Active Unify) Active Unify を行った時にデッドロックを検出した.
- [15]MRB-GC (Release) Release を行った時にデッドロックを検出した.
- raised(Node, PE, Type) シェルの知らない例外事象が発生した.
- illegal_filename(Direction, FileName) ファイルの指定方法に誤りがある.
- illegal_window_parameter(Direction, WindowParameter) ウインドウの指定方法に誤りがある.
- illegal_device_name(Direction, DeviceName) 存在しないデバイスを指定しようとした.
- cannot_create_file_request(Type, File) ファイルリクエストデバイスを生成できなかった.
- cannot_create_file(Type, File) ファイルをオープンできなかった.
- cannot_create_window_request(Window) ウインドウリクエストデバイスを生成できなかった.
- cannot_create_window(Window) ウインドウを生成できなかった.
- illegal_goal_name 起動しようとしたゴール名に誤りがある.
- undefined_code 起動しようとしたコードは存在しない.
- pipeBroken パイプブロークンを検出した.

【注意】 `Type` の種類は, 将来変更される可能性が高い.

`term1` : *Term1*

`term2` : *Term2*
ユニフィケーションに失敗したタームを表示する.

`Sub Code` : *SubCode*
例外事象の補助コードを表示する.

【注意】詳細な仕様は決定していない.

`Error ID` : *ErrID*
エラーの識別子を表示する. 例えば PIMOS に対して不正なメッセージを送った場合には, illegal_message_to_PIMOS と表示される. また標準入出力ストリームに対して不正なメッセージを送った場合には, illegal_message_to_Std_IO と表示される.

`Error Info` : *ErrInfo*
エラー情報を表示する.

`Error Data` : *ErrData*
エラーの詳細情報を表示する.

2.1.6 エラー処理

ユーザゴールが fail したり, exception を発生したりするなど, なんらかの異常を引き起こすと, そのゴールを実行しているタスクだけでなく, そのタスクが属するジョブ全体をシェルが強制終了させる.
またその際には, なんらかのエラーメッセージを表示する. そのエラーメッセージは, 以下のコマンドを用いて詳細に調査することも可能である.

last_error

最後に表示したエラーメッセージを再表示する. 印字パラメタの設定を変更してから, 直前のエラーメッセージを再確認するために用いる.

inspect

最後に表示したエラーメッセージの内容をデータとして, インスペクタを起動する. エラーメッセージの内容をより詳細に調査するために用いる (第 4.2 章 (157 頁) インスペクタ参照).

2.1.7 資源不足時の処理

ジョブの実行中に資源 (計算機資源) が不足した (resource_low が発生した) 場合, シェルは当該ジョブの実行を停止させ, 継続すべきかどうかを問い合わせる.

資源 (計算機資源) については, プログラミング編荘園機能 (第7.5節 (290頁)) を参照されたい.

資源を追加し, 実行を継続したい場合には, ジョブ管理用のコマンド (fore または back) を用いて, ジョブを再開させるだけで良い. フォア, バックにかかわらず, 決められた量の資源を追加した上で, 実行を再開する.

再開時に追加する資源量は, 以下のコマンドによって変更, 問合わせができる.

set rscinc = Inc

これ以降起動するジョブについて, 資源不足時に追加する資源量を Inc に設定する. Inc は非負で 10^9 以下の整数値, あるいはベクタ表記 { High, Low } , あるいはアトム infinite. High, Low の値域は, Inc と同じ.

Inc に infinite を指定したときは, 資源は一定量 (10^9) ずつ繰り返し自動的に追加され, 見かけ上, 無限大の資源を与えたことと同じになる. デフォルトは infinite.

【注意】2.5 版とはデフォルト値が変更されている.

get rscinc

資源追加単位量の既定値の問合せ. シェルの標準出力先 (通常はウインドウ) に出力する.

set rscinc(JobNo) = Inc

指定したジョブについて, 資源不足時に追加する資源量の既定値を Inc に変更する.

Inc は非負で 10^9 以下の整数値, あるいはベクタ表記 { High, Low } , あるいはアトム infinite. High, Low の値域は, Inc と同じ. infinite を指定したときは, 資源は一定量 (10^9) ずつ繰り返し自動的に追加され, 見かけ上, 無限大の資源を与えたことと同じになる.

get rscinc(JobNo)

ジョブに設定した資源追加単位量の問合せ. シェルの標準出力先 (通常はウインドウ) に出力する.

2.1.8 ジョブを実行するノードの指定

set node = NODE

これ以降起動するジョブを実行するノードを指定する. ジョブの下の各タスクのトップレベルゴールは, すべて非負整数 NODE で指定される番号のノードで実行を開始する. デフォルトはシェルが起動されたノードである.

get node

現在ジョブを起動したら, そのジョブの下の各タスクのトップレベルゴールを実行することになるノードの番号の問合せ. 番号をシェルの標準出力先 (通常はウインドウ) に出力する.

【注意】ノード指定機構は将来変更される可能性が高い.

2.1.9 ジョブの実行優先度範囲の指定

set minprio = Min

set maxprio = Max

それぞれ, これ以降起動するジョブの下の各タスクの優先度範囲の下限, 上限を指定する. Min, Max はシェル自身に与えられた優先度範囲との相対指定で, $0 \leq \mathrm{Min} \leq \mathrm{Max} \leq 4000$ であること. この範囲を逸脱した値を設定しようとしても受け付けられず, コマンドのエラーとなる. なおデフォルト値はそれぞれ 0, 4000 である.

get minprio

get maxprio

それぞれ, これ以降起動するジョブの下の各タスクの優先度範囲の下限, 上限の問合せ. 値をシェルの標準出力先 (通常はウインドウ) に出力する.

【注意】優先度指定機構は将来変更される可能性が高い.

2.1.10 環境変数の操作

setenv Key = Value

シェル自身のタスクが持つ環境変数プールに, Key をキーとして値 (Value) を設定する.

unsetenv Key

シェル自身のタスクが持つ環境変数プールから, Key をキーとする環境変数を削除する.

getenv Key

シェル自身のタスクが持つ環境変数プールから, Key をキーとする値を取り出し, シェルの標準出力に表示する.

getenv

シェル自身のタスクが持つ環境変数プールから, すべての値を取り出し, シェルの標準出力に表示する.

2.1.11 シェルが使用する環境変数

シェルで使用している環境変数には以下のものがある. (第 11.1.7 章 (401 頁) 参照).

task:directory

デフォルトディレクトリを指定する. ディレクトリ名 (ストリング) を設定する.

task:home

ホームディレクトリを指定する. ディレクトリ名 (ストリング) を設定する.

task:package

デフォルトパッケージを指定する. 値はパッケージ名 (アトム) である.

task:io_node

入出力を行う I/O ノードの名称を指定する.

環境変数 task:io_node については, プログラミング編第 11.1.3 節 (400 頁) I/O ノードを参照されたい.

2.1.12 コマンドファイル

シェルのコマンド入力は, 通常シェルウインドウから行うが, これを一時的にファイルに切り換え, ファイル中に書かれたコマンド列を実行することができる.
take コマンドを実行すると, まず始めにファイルからすべてのコマンド行を読み込み, シェルのコマンドバッファに格納する. その後, コマンドバッファから 1 行ずつコマンド行を取り出し, 解釈実行を行う.

take コマンドファイル名

シェルのコマンド入力をコマンドファイル名で指定したファイルから行い, コマンドを実行する. コマンドファイル名は, cd コマンド (第 2.1.4.7項 (19 頁) 参照) のディレクトリ指定と同様の記述方法が可能である.
もしもファイルが存在しなければ, ファイル名拡張子 .com を付加して再度検索を行う.
コマンドファイル中に take コマンドを記述することもできる.
またコマンドファイル内の各コマンド行は, 逐次に解釈され, 実行される.

【注意 1】コマンドファイルの読み込み中は, アテンションキーを受け付けない.

【注意 2】コマンドバッファからコマンド行を取り出して解釈実行している期間, アテンションは, 個々のコマンド行に対しては有効である. しかし take コマンド自体をキャンセルすることは現在のところできない. したがってアテンションを起こした時点で, コマンド行が残っていたら, その残りのコマンド行が解釈実行される.

set echo = MODE

コマンドファイル実行中に, コマンド行のエコーバックを行うか否かを指定する. MODE は, アトムでなければならない. MODE が off の場合には, エコーバックを行わない. それ以外の場合には, エコーバックを行う. シェル起動時のデフォルト値は off である.

get echo

コマンドファイル実行中に, コマンド行のエコーバックを行うか否かをシェルの標準出力先 (通常はウインドウ) に出力する.

2.1.13 初期化ファイル

シェルの起動時に自動的に実行するコマンドファイルである．ファイル名は, shell.com に固定である．シェル起動時に環境変数 task:io_node, task:directory の値にしたがってファイルをサーチする．ファイルが存在しなければ, 環境変数 task:io_node, task:home の値にしたがって再度ファイルをサーチする．それでも存在しなければ, なにも行わない．ファイルの記述方法については, コマンドファイル (第 2.1.12項 (29頁)) を参照されたい．

2.1.14 ログインファイル

ログイン用のシェル (pimos::shell:login) を起動した時に最初に自動的に実行するコマンドファイルである． ファイル名は, pimos.com に固定である． シェル起動時に環境変数 task:io_node, task:directory の値にしたがってファイルをサーチする．ファイルが存在しなければ, 環境変数 task:io_node, task:home の値にしたがって再度ファイルをサーチする．それでも存在しなければ, なにも行わない．ファイルの記述方法については, コマンドファイル (第 2.1.12項 (29頁)) を参照されたい．

ログイン用のシェルは, 前述の初期化ファイル読み込みも行う．実行の順序は, 先に shell.com を読み込み, pimos.com を後から読み込む．また pimos.com の読み込みは, shell.com の存在いかんに関わらず行われる．

2.1.15 ログインファイルと初期化ファイルの例

以降に述べる環境のもとで, 例に示したログインファイルと初期化ファイルを用意し, ログインシェルを起動すると, ログインシェル自身の環境設定はもちろん,

1. ログインシェルの環境設定を行う．
2. 子シェルをログインシェルから起動する．
3. その子シェルの環境設定を行う．
4. その子シェルから, コンパイラを起動する．
5. ログインシェルから, リスナを起動する．

といった, 一連の初期処理を自動的に行うことができる．

ログイン時 (ログインシェル起動時) の環境

- 環境変数

  task:directory = ">sys>user> ユーザ名 "

  task:home = ">sys>user> ユーザ名 "

- ディレクトリ構成

  >sys>user> **ユーザ名** ホームディレクトリ

  >sys>user> **ユーザ名**>src ソースファイルを格納したディレクトリ

- ホームディレクトリ下の pimos.com の内容 (初期起動タスクを指定する)

```
cd ~>src
shell(at(700, 3), char(50, 10), ">sys>font>test_11") &
cd ~
listener( at(300, 610), char(65, 12), ">sys>font>test_11") &
```

- ホームディレクトリ下の shell.com の内容 (全シェルに共通な環境設定を記述する)

```
set prompt = "Shell<<\!>> "
set minprio = 0
set maxprio = 4000
set rscinc = infinite
set node = 0
set hl = 50
set pd = 50
set pl = 50
set pal = 100
set psl = 100
alias h = "history"
alias printenv = "getenv"
alias fg = "fore"
alias bg = "back"
alias jobs = "status"
```

- ディレクトリ src の下の shell.com の内容 (個々のシェルの環境設定を記述する)

```
take ~>shell
compile
```

2.2 シェルユーティリティプログラム

シェルユーティリティプログラムは, シェルからの起動を前提として設計されたユーティリティプログラムである.

各コマンド名は, モジュール名に対応している. 述語名は, すべて go であるため省略してある. 当然, コマンド入力時にも特に記述する必要はない.

これらのモジュールは, シェル本体のプログラムとは全く係わりがなく, 通常のユーザ定義のモジュールとなんら変わりがない. このため, これらのモジュールを削除あるいは上書きしてもシェル本体の動作に差し障りはない.

2.2.1 GC の起動

ガーベジコレクション (GC) は, ゴールのリダクションにともなってメモリの空き領域が不足した際に, すでに使用していないメモリ領域 (garbage) を調べ, 空き領域として回収 (collection) するための機能である. 現在 Multi-PSI システムでは, MRB-GC と Copying-GC と呼ばれる 2 種類の GC 方式を提供している. MRB-GC は, すべて自動的に行われるため, ユーザが意識する必要はない. 一方 Copying-GC もメモリの空き領域が不足すると自動的に開始されるが, それとは別に任意の時点で強制的に行うこともできる. ここに述べるコマンドはこの Copying-GC を空き領域量の如何に関わらず強制的に起動するためのコマンドである.

【参考】デッドロックには, Copying-GC が行われるまで発見されないタイプのものがある. ユーザが起動したゴールがデッドロックした場合に, この検出されないデッドロックが発生すると, あたかもシェルがハングアップしたかのように見えることがある. そのような場合には, アテンションキーを押下してデッドロックしたゴールを含むジョブをバックグラウンドに移行させた上で, GC を起動すると検出可能なすべてのデッドロックゴールを検出することができる.

【注意】現在 Multi-PSI システムが提供している GC 方式は, ノードごとにメモリ回収を行う局所的な GC である. Multi-PSI システム全体に対する大域的な GC を行う手段は, 今のところ提供されていない.

gc

全ノードに対して一斉に GC 開始を指示する. ただし, このプログラム自身が動作しているノードに対しては, 最後に指示する.

gc(NODE)

引数 NODE で指定したノード番号のノードに対して, GC 開始を指示する. ただし, NODE にアトム self が指定された場合には, このプログラムが起動されたノードを調べ, そのノードに GC 開始を指示する. 指定したノードが存在しないなど, ノード指定に誤りがあった場合には, GC は行われずエラーとなる.

seqgc

全ノードに対してノード番号の若い順に GC 開始を指示する. ただし, このプログラム自身が動作しているノードに対しては, 最後に指示する.

seqgc(NODE)

引数 NODE で指定したノード番号のノードに対して, GC 開始を指示する. ただし, NODE にアトム self が指定された場合には, このプログラムが起動されたノードを調べ, そのノードに GC 開始を指示する. 機能的には, 前述の gc/1 となんら変わりはない. 指定したノードが存在しないなど, ノード指定に誤りがあった場合には, GC は行われずエラーとなる.

2.2.2 状態表示

2.2.2.1 資源木の表示

PIMOS の提供するタスクや入出力機器などの資源は, 資源木と呼ばれるひとつの木構造の下に管理されている. ここに述べるコマンドは, その資源木の状況を表示するためのものである.

1 つの資源は, 必ず 1 行で表す. 各行は, すべての PIMOS 資源を一意に識別するためにつけられている資源識別子, 資源の種類を表す資源タイプ, 及び資源の名称の 3 つの項目からなる. また行のインデンテーションは, 資源木の階層関係を表す.

指定されたすべての資源木を表示し終わると OK ? と尋ねる. ここで改行キーを押下すると, プログラムが終了する. 改行キーの前にアテンション以外のいかなるキーを入力しても無視する.

rs

このプログラムを起動したシェルの子資源について, 資源木を調べ, その内容を表示する.

rs(Id)

引数 Id で指定した資源の子資源について, 資源木を調べ, その内容を表示する. 引数 Id の記述方法は, general_request_stream のプロトコルを参照されたい.

rsall

PIMOS 管理下の全資源について, 資源木を調べ, その内容を表示する.

2.2.2.2 タスク木の表示

資源木のうち, タスクに関する情報のみを表示するためのコマンド群である.

1 つのタスクは, 必ず 1 行で表す. 各行は, すべてのタスクを一意に識別するためにつけられているタスク識別子, タスク名, タスクの実行状態, 資源消費量上限値, 資源消費量の 5 つの項目からなる. また行のインデンテーションは, タスクの階層関係を表す.

タスク名は, タスク起動時に設定したタスクの名称である. タスクの実行状態は, 次の 5 つの状態のうちのいずれかである.

running 実行中

resource_low 資源不足のため停止中

stop(Reason) 制御ストリームあるいは親タスクからの停止要求により停止中. 制御ストリームからの停止要求による場合には, Reason は, 報告ストリームからのメッセージ stopped/2 に指定した停止理由である. 一方, 尊属タスクからの停止要求の場合には parent が停止理由となる. なお停止理由は 1 つとは限らない.

terminating タスクの終了処理中

shut_terminating 親タスクの強制終了による終了処理状態

消費資源量上限値及び資源消費量は, 64 bit の数値であるが, これを 2 要素のベクタで表す. ベクタの第 1 要素が上位 32 bit に, 第 2 要素が下位 32 bit に対応する. ただし上位が 0 の場合には, ベクタとはせず単なる数値で表す. また値が不明の場合には, unknown と表示する.

指定されたすべての資源木を表示し終わると OK ? と尋ねる. ここで改行キーを押下すると, プログラムが終了する. 改行キーの前にアテンション以外のいかなるキーを入力しても無視する.

ts

このプログラムを起動したシェルの子タスクについて, 資源木を調べ, その内容を表示する.

ts(Id)

引数 Id で指定したタスクの子タスクについて, 資源木を調べ, その内容を表示する. 引数 Id の記述方法は, general_request_stream のプロトコルを参照されたい.

tsall

PIMOS 管理下の全タスクについて, 資源木を調べ, その内容を表示する.

2.2.3 PIMOS の終了

PIMOS 全体の実行を終了させるためのコマンド.

shutdown

PIMOS を終了させる.

2.2.4 ユーザ管理

2.2.4.1 ログインユーザの表示

現在ログインしているユーザを表示する.

who

現在ログインしているユーザを表示する. ユーザ名およびユーザの使用しているログイン区分, ログインした日付時間, FEP または I/O ノードのノード名, I/O ノード上のユーザ名およびローカルエリアネットワークの種類が表示される.

なおログイン区分については, 操作編 第 1 章 (6頁) ログインを参照されたい.

2.2.4.2 特権ユーザへの移行

特権ユーザになる. UNIX の su と同様のコマンドである.

特権ユーザについては, 操作編 第 11.4.1 節 (425頁) を参照されたい.

su

特権ユーザになる. このシェルコマンドを使用すると, パスワードを入力するためのウインドウが表示されるので, スーパユーザ (UID: 0) のパスワードを入力する. もし, パスワードが間違っていたら, シェルのウインドウに “Sorry” と出力される.

2.2.4.3 パスワードの変更

パスワードを変更する. なお, このパスワードの変更は, このログイン時に最初に起動したプログラム (通常はログイン用シェル) を終了させた時に, ファイル (pimos.logins) に反映される (第 6 章参照).

change_password

ユーザ名のパスワードを変更する. このシェルコマンドを使用すると, パスワードを入力するためのウインドウが表示されるので, ユーザ名とパスワードを入力する.

なお変更できるのは, ログインしているユーザ自身のパスワードだけであり, 他ユーザのパスワードは変更できない.

2.2.5 コピー

標準入出力間でコピーを行うためのコマンド群.

cat

標準入力から一行ずつ読み込み, それを標準出力に出力する. 入力した行が空行であった場合には, 読み込みの終了であるとみなしてプログラムを終了する.

tee

標準入力から一行ずつ読み込み, それを標準出力および標準メッセージ出力に出力する. 入力した行が空行であった場合には, 読み込みの終了であるとみなしてプログラムを終了する.

2.2.6 モジュール使用方法の表示

各種モジュールの使用方法を表示するためのコマンド群.

usage

usage コマンド自身の使用方法を標準出力に表示する.

usage(ModuleName)

usage(PackageName::ModuleName)

usage(PackageName, ModuleName)

引数 PackageName (アトム) および ModuleName (アトム) で指定したモジュールの使用方法を表示する. PackageName が省略された時には, task:package に格納されているパッケージ名をもつパッケージを対象として探索する.

実際には, 引数で指定したモジュール名を持つモジュールの述語 usage/1 を実行し, その引数に返される書式文字列と引数リストからなる 2 要素ベクタのリストを入出力バッファ (プログラミング編第 9.2.1 節 (338 頁)) のプロトコル putf/2 を用いて表示している. したがって, 指定したモジュールに述語 usage/1 がない場合や引数として返される値がおかしい場合には, usage が登録されていない旨のメッセージを表示する.

【例】シェルの場合

```
usage(Usage) :- true |
    Usage = [{string#"~s~n~s~n~s~n~s~n~s~n", [S1, S2, S3, S4, S5]}],
    S1 = string#"Usage : shell",
    S2 = string#"Usage : shell(at(X, Y), char(W, H))",
    S3 = string#"Usage : shell(X, Y, W, H)",
    S4 = string#"Usage : shell(at(X, Y), char(W, H), Font)",
    S5 = string#"Usage : shell(X, Y, W, H, Font)".
```

【注意】現在 PIMOS の提供するモジュールでも述語 usage を定義しているものはほとんどない.

2.2.7 文字列およびテキストファイルに対するコマンド

2.2.7.1 エコー

引数に与えた値をエコーバックするコマンド.

echo(X)

引数 X に与えられた内容を標準出力に出力する.

2.2.7.2 文字列の検索

文字列を検索するコマンド.

grep(Pattern)

標準入力から一行ずつ読み込み, その行に引数 Pattern に与えられた文字列が含まれていれば, その行を標準出力に出力する. 入力の終了判定は, 入力メッセージ getl が end_of_file をステータスに返すことで行っている. それゆえ, 標準入力にウインドウを指定した場合には, 決して end_of_file 状態にはならないため, プログラムを終了させるためには, アテンションキーによってプログラムを一時中断した後, シェルコマンド kill を用いて強制終了させねばならない.

grep(PathnameWithWildcard, Pattern)

引数 PathnameWithWildcard (文字列) に与えられたワイルドカードを含むファイルパス名にマッチしたファイルから一行ずつ読み込み, その行に引数 Pattern に与えられた文字列が含まれていれば, その行を標準出力に出力する. なお, ワイルドカードはディレクトリに含まれていても良い.

【注意】コマンド名とは裏腹に, 現在は regular expression の検索処理は行っていない. これは将来改訂される可能性がある.

2.2.7.3 行数のカウント

行数をカウントするコマンド.

lc

標準入力から一行ずつ読み込み, 行数を標準出力に出力する. 入力した行が空行であった場合には, 読み込みの終了であるとみなしてプログラムを終了する.

2.2.8 ディレクトリの表示

ディレクトリの内容を表示する.

ls

カレントディレクトリの下にあるファイルを一覧表示する.

ls(PathnameWithWildcard)

引数 PathnameWithWildcard(文字列) に与えられたワイルドカードを含むファイルパス名にマッチしたファイルを一覧表示する. なお, ワイルドカードはディレクトリに含まれていても良い.

2.2.9 変数チェッカ

モジュール varchk は指定された KL1 ソースプログラムを調べ, バグの兆し, 効率上望ましくないようなプログラムを指摘するシェルプログラムである.

具体的な機能としては現在以下の 2 つを備えている.

- クローズ中に一つしか出てこない変数を表示する
- MRB が黒くなる変数を表示する

これらの機能を選択している状態を「モード」と呼ぶ。現在、モードには以下のようなものがある。

one : クローズ中に一つしか出てこない変数を表示する

mrb : MRB が黒くなる変数を表示する

all : one と mrb の両モードの変数を表示する

また, プログラムの出力形式として、以下のような選択が可能である.

short : 該当するクローズを一行に出力する

long : 該当するクローズを改行 / インデントして出力する

varchk

インタラクテイブ変数チェッカを起動する. 起動後に対象となるソースプログラム、オプションを指定することによりプログラムの検査を行う.

【例】起動例

```
*** KL1 Variable Checker ***
VARCHK[o,l] me:demo1, me:demo2
      :
VARCHK[o,l] !exit
```

ここで、

- `VARCHK[o,l]`
  プロンプト。`[o,l]` 部分は現在のモード、出力形式 (後述) を表示する.
- `me:demo1, me:demo2`
  ユーザ入力 (ファイル名). ここでは、以下のような入力が許されている.

  ファイル名： 検査を行うファイル名を入力する。複数ファイルの指定が許されており、カンマ, 若しくは空白で区切る. 例で示すように, ファイル名は quotation(‘"’,‘'’) で囲ってはならない. 拡張子 .kl1 は省略可能. また, ファイル名は *.kl1 の様にワイルドカードでも指定できる. この場合, 拡張子 (通常 ‘.kl1’) は省略できない. (‘*’ のみであると, 拡張子のないファイル名とマッチしてしまう)

  !exit： 終了

  .： one モードに切り替える

  ..： mrb モードに切り替える

  ...： all モードに切り替える

  /： short 出力形式に切り替える

  //： long 出力形式に切り替える

  !!： 最後のコマンドでチェックを行ったファイル群を再度チェックする

  ?： ヘルプ

  !help： ヘルプ

  !cd： ワーキングディレクトリを変更する. ここで指定されるディレクトリ名は quotation (‘'’, ‘"’) で囲ってはならない.

  !pwd： ワーキングディレクトリを表示する

varchk(FileNameList, Mode, Form)

バッチ版変数チェッカを起動する.

FileNameList: ファイル名ストリングまたはそのリスト. 拡張子 .kl1 は省略可能.

Mode: モード

one, o： クローズ中に一つしか出てこない変数を表示する

mrb, m： MRB が黒くなる変数を表示する

all, a： one と mrb の両モードの変数を表示する

Form: 出力形式

short, s： 該当するクローズを一行に出力する

long, l： 該当するクローズを改行 / インデントして出力する

第 3 章

コンパイラ

本章では KL1 コンパイラとそれに関連する機能の操作法について解説する.

【参考】本章の説明中に FileNameList, ModuleNameList などと記載されている引数は, 一要素だけを指定する場合はリストにしなくとも良い. たとえば, FileNameList として単一のファイル名だけを指定する場合, “[FileName]” の形式で指定しても, 単に “FileName” としても同じ意味である.

3.1 セルフコンパイラ

主として KL1 プログラムをコンパイル, アセンブル, リンクするためのシェルプログラムである. 標準入出力を使用するため, シェルやリスナから起動される必要がある. ソースプログラムはファイルから読み込まれ, コンパイルが成功すればモジュールを生成し, モジュールデータベースに登録する.

シェル等からの起動の仕方には 2 通りある. ひとつはバッチコンパイル, もう一つはインタラクティブコンパイルである.

compile

インタラクティブコンパイルを起動する.

compile(FileNameList)

バッチコンパイルを起動する. 引数の FileNameList はファイル名 (ストリング) を要素とするリストで, コンパイル対象ファイルを指定する.

【例 1】バッチコンパイルの起動例 (その 1)

```
SHELL> compile(["me:testprog1.kl1", "me:testprog2.kl1"])
```

【例 2】バッチコンパイルの起動例 (その 2)
シェルの標準出力をファイルにすることで, コンパイルのログを取ることができる.

```
SHELL> compile(["me:test1", "me:test2"]) => file("me:comp.log")
```

【例 3】インタラクティブコンパイルの起動例

```
SHELL> compile
```

【例 4】

```
SHELL> compile(["me:demo1", "me:demo2"])
```

【参考】上の例は, 次に示すインタラクティブコンパイルの実行と等価である.

```
SHELL> compile
COMPILE> me:demo1,me:demo2
              :
COMPILE> !exit
SHELL>
```

3.1.1 基本的事項の説明

- コンパイラの初期化

  コンパイラには種々のパラメタがある. これらのパラメタの初期化方法には, 環境変数による方法と, コンパイラのパラメタファイルによる方法の二通りがある.

  コンパイラは起動時に, "compile.param" というファイルをカレントディレクトリ, ホームディレクトリの順に探し, 見つかればその内容を読み込んでパラメタを設定する. もし, 該当するファイルが見つからない場合は, 環境変数に設定されている値に従って各種パラメタを設定する.

  このパラメタ設定の規則は, バッチコンパイル, インタラクティブコンパイルの区別はない.

  【注意】パラメタファイルの内容は PIMOS 2.5 版と PIMOS 3.0 版で一部異なっているため, 旧版のパラメタファイルを読み込んだ時は, 警告メッセージが出力されるようになっている. 2.5 版から 3.0 版への変換は自動的にはおこなわれないので注意のこと.

  【参考】パラメタファイルは, インタラクティブコンパイルにおける `!keep` コマンド (50頁) で生成することができる.

  【参考】パラメタファイルが読み込まれると, パラメタファイルのフルパス名が表示される.

- ファイル名拡張子とファイルの因果関係の推定

  コンパイル対象と指定したファイルの名前には, 自動的にファイル名拡張子を補って解釈する. デフォルトの拡張子には, `mac` (マクロ定義ファイル), `kl1` (普通の kl1 ソースファイル), `pkl1` (マクロなし kl1 ソースファイル), `sav` (アンロードされたファイル) がこの順序で設定されており, これは `!default_extension` コマンド (43 頁) で変更できる. ファイル名拡張子の順序は, 同名ファイルの因果関係とみなされる. 後のものは前のものから生成されるものと見なし, ファイルの最終更新時間順序が逆転している場合は, その逆転しているファイルをコンパイルする.

- 負荷分散

  `!distribute` コマンド (46 頁) によりファイル単位でコンパイラの負荷分散を指定することができる. コンパイラは, `!parallel` コマンド (48 頁) で指定された数を, 最大とするファイルを指定されたノード (プロセッサ) に順次割り当てる. もし, コンパイルすべきファイルが〔ノード数×ファイルの最大数〕を上回る場合は, ファイルのコンパイル終了を待って, 終わったノードに残りのファイルを割り当てる.

  したがって, 一般にユーザの指定したファイルの順序にコンパイルが終了する保証はない. コンパイルにおける各種表示メッセージはコンパイルの終了した順とほぼ同じである.

  【参考】擬似 (シュード) システムにおいては, 一般に負荷分散をおこなった方がプロセスチェンジが頻発するため遅くなる.

- 表示メッセージ

  PIMOS 3.0 版からコンパイラが出力する表示メッセージの形態が大きく変わっている. 各表示メッセージの先頭には, メッセージを発する要因となったソースファイル名とその位置が示される.

【例】シンタックスエラーの表示例

```
!ERROR![test.kl1@1:pimos.h@3] *** Syntax Error ***
          :
```

このメッセージの意味は, test.kl1 というファイルの 2 番目の項 (ゼロオリジン) にある include 文で読み込まれた pimos.h というファイルの 4 番目の項でシンタックスエラーが検出された, ということになる.

ここで, 位置の単位はソースファイルを読み込んだパーサに依存する. 例えば, KL1 の標準パーサである kl1_standard_preprocessor では項の数が単位となっている.

【参考】このような機構を組み込んだ目的としては, プリプロセッサの用いた場合でも, メッセージの原因となった場所がソースファイルと容易に関連付けられるようにすることが挙げられる.

- プリプロセッサ

  セルフコンパイラでは, KL1 をベースとした上位言語のコンパイラの開発を容易にするための機能として, プリプロセッサ機能がある. これは, KL1 の上位言語をいくつかの言語変換フィルタを通すことで最終的に KL1 に変換し, それを KL1 コンパイラでコンパイルするものである.

  上位言語の開発者は, 言語独自のパーサや言語を変換するためのプリプロセッサ, 言語を清書するためのアンパーサ, そしてプログラムを静的にチェックするベリファイアなどの部品を適宜開発すれば良い. (プリプロセッサの定義方法については, プログラミング編 D 章 プリプロセッサの作成方法 435 頁を参照のこと)

  また, コンパイラのファイルの指定におけるパスやワイルドカードや負荷分散などのセルフコンパイラ機能がそのまま利用でき, 操作方法も統一されるという利点がある. さらに, 変換結果をファイルに保存する必要がないため, トータルでのコンパイル時間の短縮も期待できる.

  【注意】ベリファイア機能は, 3.0 版から加わった新機能である.

3.1.2 バッチコンパイル

バッチコンパイルは, 引数に指定されたファイルのみをコンパイル, アセンブル, リンクし, 生成されたモジュールを登録する.

リンク時に未定義外部参照が検出されると, エラーとなりそのモジュールは登録されずにコンパイラは終了する. このため, デバッグ中などでコンパイルに失敗する可能性が大きい場合は, 後述するインタラクティブコンパイルを用いた方が良い. バッチコンパイルは, シェルのコマンドファイルに用いると便利である.

3.1.3 インタラクティブコンパイル

インタラクティブコンパイルでは, シェルの標準入力からコンパイルすべきファイル名や各種コマンドを入力し, コンパイルエラーなどのメッセージを標準出力あるいは標準メッセージ出力に出力する.

インタラクティブコンパイルの入力コマンドを一括してファイルに保存しておき, そのファイルをコンパイラの起動時に標準入力として指定することも可能である. なお, 同様のことは `!take` コマンド (50頁参照) でも可能である.

【例】標準入力にファイルを指定する場合

```
SHELL> compile <= file("compile.com")
```

3.1.3.1 コマンドシンタックス

行の先頭文字が ”！” で始まる場合は, コンパイルコマンドとみなされる. コマンド名は, “！” に続く区切り文字 (空白, カンマ (,) または改行) までの文字列であり, 大文字, 小文字の区別がある. コマンド名以降のトークンはコマンドの引数となるが, 引数の型は, コマンドと引数の位置により適宜, 文字列, アトム, 複合項に解釈・変換される. 引数に任意の項が期待されているような場合は, 文字列であれば明示的に二重引用符で括る必要が有る.

引数に区切り文字が含まれる場合は, “！” で括ることができる. “！” で括られた内部に”！” を入れたければ, “！” を 2 回続けて書かなくてはならない.

その他のシンタックス規則を以下に列挙する.

- 継続行

  “！” で括られた内部の改行, あるいは行末がカンマ (,) の場合は継続行とみなされる.

- ファイル名の指定

  ファイル名の指定においては, `"~>pimos"`, `"..>.."` などといった指定も可能である.

- モジュールの指定

  モジュール名を指定する場合では, デフォルトパッケージのモジュールを指定する時はモジュール名のみ, デフォルトパッケージと異なる時は `Package::Module` というようにパッケージを陽に指定する.

【注意】コマンド名は (二重) 引用符で括ることはできない.

【例】以下のコマンド入力はいずれも等価である.

```
COMPILE> !preprocess abc def package::module

COMPILE> !preprocess "abc", def !package :: module!

COMPILE> !preprocess abc,
         def  !package ::
         module!
```

3.1.3.2 リンクの順序

外部参照の探索は, それまでにコンパイルされたモジュールが先に探索され, 次にモジュールデータベースが探索される. もし, 未定義モジュールが検出された場合は, 未解決モジュールとしてコンパイラ内部に保持される. 以後のコンパイルにより未定義モジュールがコンパイルされた時点でその未解決モジュールの外部参照は自動的に解決され, モジュールとして登録される.

未解決モジュールが不要になったならば, ピリオド (.) を入力することにより, 未解決モジュールを棄てることができる.

3.1.3.3 インタラクティブコンパイルの終了

“!exit” と入力することで, プログラムは終了する. 未解決モジュールがある場合には, 単に破棄される.

【例 1】インタラクティブコンパイルの実行例

```
SHELL> compile
** KL1 Compiler **
COMPILE> me:demo1.kl1
```

```
                :        % モジュールや述語のコンパイル情報が出力される
                :
COMPILE> me:demo2.kl1
COMPILE> !exit
SHELL>
```

【例 2】インタラクティブコンパイルの実行例

```
SHELL> compile
** KL1 Compiler **
COMPILE> me:demo1,me:demo2
                :
COMPILE> !exit
SHELL>
```

【注意】上の 2 例では, ファイル demo1 にあるモジュールとファイル demo2 のモジュール間に参照関係がないか, demo2 のモジュールが demo1 のモジュールを参照している場合には同じ結果になる. そうでない場合に例 1 のような指定を行なうと, コンパイル結果は保証しない.

3.1.3.4 インタラクティブコンパイルのコマンド

インタラクティブコンパイルでは, 以下に示す“!”で始まるコマンドがある.

- コンパイルに関するコマンド

  !compile *FileName*, ..

  FileName で指定されたファイルをコンパイルする.

  【注意】3.0 版で追加された新コマンドである.

  !klb *FileName*, ..

  FileName で指定されたファイルをコンパイルし, その結果としてのアセンブル命令列を同名のファイルで拡張子が ”klb” というファイルに出力する.

  【注意 1】このコマンドは高速化のためのチューニングやコンパイラのデバッグのために用意されたもので, 一般ユーザが用いるものではない.

  【注意 2】このコマンドの結果としてモジュールは生成されない.

  【注意 3】3.0 版で追加された新コマンドである.

  !make

  top_modules (コマンドまたは環境変数) で指定されたモジュールを頂点とするモジュール参照関係木において, **working_packages** (コマンドまたは環境変数) で指定されたパッケージ内にあるモジュールのソースファイルのコンパイル時点でのファイル変更日時より後に変更されたファイルを捜し出し, 再コンパイルする機能である. この機能において, **auto_link** (コマンドまたは環境変数) の指定は有効であり, 再リンクは自動的に行なわれる.

【注意 1】このコマンドはタイムスタンプ情報を持つモジュールにしか有効でない．また, IPL でロードされたモジュールは **make** コマンドの対象外となる．

【注意 2】モジュールのタイムスタンプは, PIMOS 3.0 版から導入された機能であるため, PIMOS 2.5 版以前にコンパイルされたモジュールに対してはこの make 機能は有効ではない．

【注意 3】対応するファイルの探索において path の指定は無効である．コンパイルされたソースファイルがコンパイル後にリネームされたり，ディレクトリを移動したような場合は，そのファイルについて陽に指定してコンパイルする必要がある．

【注意 4】3.0 版で追加された新コマンドである．

- ファイルの指定に関するコマンド

!!

直前に指定したファイルをもう一度コンパイルする．

!cd [*Directory*]

Directory で指定されたディレクトリをカレントディレクトリにする．“..>..” や “~>pimos” なども指定可能である．引数を省略した場合は，ホームディレクトリをカレントディレクトリとする．

!default_extension [*Extension, ..*]

コンパイルの対象となるファイル名において, デフォルトとなるファイル名拡張子を Extension で指定する．引数を省略した場合は, 設定されているファイル拡張子を表示する．ここで指定されたファイル名拡張子の順序は, 同名ファイルの時間的因果関係があるものとみなされる．後のものほど時間的に新しいものとし, この時間順序が逆転している場合は, その逆転しているファイルをコンパイル (あるいは, **!translate, !verify** コマンドの対象と) する．
デフォルトでは, `mac`(マクロ定義ファイル), `kl1`(普通の kl1 ソースファイル), `pkl1`(マクロなし kl1 ソースファイル), `sav` (アンロードされたファイル) がこの順序で設定されている．

【注意】ファイル拡張子の指定において, ファイル名と拡張子を分離するピリオド (.) は不要である．

【参考】同名の mac, kl1, pkl1, sav ファイルが存在する場合において, mac ファイルが kl1 よりも古く, kl1 ファイルが pkl1 ファイルよりも新しい場合は, kl1 ファイルをコンパイルの対象とする．

!ls [*Pattern, ..*]

path で指定されたディレクトリそれぞれのリスティングを出力する．ここで, Pattern はワイルドカードの指定を示す. 引数を省略した場合は “*.*” が指定されたものとみなされる．

!path [*PathName, ..*]

ソースファイルの探索パスを PathName で指定する．コンパイルで指定されたファイルがカレントディレクトリにない場合あるいはワイルドカードが指定された場合は, このパスで指定されたディレクトリを探索する．

【参考】これによりバージョンが異なるファイルの管理あるいは大きなプログラムでソースファイルを複数のディレクトリで管理している場合にはそれらのディレクトリをこの変数に指定するというような使い方が考えられる.

!pwd

カレントディレクトリを表示する.

- リンクに関するコマンド

!auto_link (*on* | *off*)

on で自動リンク, off で自動リンクをおこなわない. デフォルトは on である (従来と同じ). off の場合は明示的に `!link` コマンドを入力しなければならない.

【参考】モジュールを生成するコマンド (コンパイル, relink, load) では, コマンドの終了時に必ずリンクをおこなっていた. この方式では, 場合により無駄なリンク及び再リンクをおこなう可能性があり, メモリを無駄に消費する可能性が生じる.

【注意】再リンク (relink) はリンクが完了し完全な状態でロードされたモジュールに対してのみ有効である. 自動リンクを off にした場合, コンパイラ内部に未解決モジュール (unresolved module) として保持されている不完全なモジュールに対しては再リンクは無効であるので, 注意を要する. 自動リンク off の場合の再リンクは, 対象となるモジュールの外部モジュール参照を未定義にしたモジュールを新たに生成する, という意味になる.

!link

(**auto_link** コマンドまたは環境変数により) 自動リンクが off の場合に, 明示的なリンクを指定するためのコマンドである. (**top_modules** コマンドまたは環境変数により) 自動再リンクが指定されていた場合には, リンクが完了したモジュールが再リンクの対象となる. on の場合は無意味なコマンドである.

!link_error (*warning* | *error*)

リンクエラー (未定義モジュールあるいは未定義述語) の取り扱いを規定する. warning では, ワーニングメッセージを出力して無理矢理リンク (したフリを) してしまう. error では, 従来通りエラーとなってリンクはおこなわれない.

【注意】未定義外部モジュールあるいは未定義外部述語を含んだモジュールを実行させると, その部分の実行時に例外が発生する.

【参考】未定義参照を含むモジュールは, 未定義だったモジュールが生成されていれば再リンクやロードにより未定義参照を解消することができる.

!original_package (*on* | *off*)

再リンカまたはローダで生成されたモジュールをどこのパッケージに入れるかを指定するスイッチである. on で元のモジュールと同じパッケージに, off でデフォルトパッケージに生成される. デフォルトは off である.

!package *Package*

Package で指定されたパッケージをデフォルトパッケージとする.

【注意】`working_packages`を指定している場合には, このコマンドによりデフォルトパッケージを変更する場合には, 再リンクなどで探索するパッケージが指定したモジュールのパッケージと異なるため注意しなければならない.

!top_modules [*ModuleName, ..*]

コンパイルの正常終了後あるいはロードの正常終了後に, ModuleName で指定されたモジュールを最上位モジュールとみなして再リンクを自動的に行う. この設定がされていないと再リンクは行われない. 引数を省略するとこの変数の値を空にする.

- プリプロセッサに関するコマンド

!parser [*InLang* [*ParserModuleName*]]

InLang で指定された入力言語 (ファイルの拡張子) に対応した構文解析用モジュール (ParserModuleName) を指定する. モジュール名を省略した場合は, 設定されているモジュール名を表示する. また, 引数を全て省略した場合は, 設定されている全ての構文解析モジュールを表示する.

!preprocess [*InLang OutLang* [*PreprocModuleName, ..*]]

InLang で指定された入力言語 (ファイルの拡張子) を出力言語 (OutLang) に変換するプリプロセッサモジュール (PreprocModuleName) を指定する. この指定により, ファイルの拡張子から自動的に対応するプリプロセッサがコンパイル時に割り当てられる. ひとつの入力言語に対し複数のプリプロセッサが指定された場合には, 左から順にプリプロセッサがパイプライン的に割り当てられる.

最終的に KL1 言語としてコンパイルされるためには, マクロを含まない `pkl1` が出力言語となっている必要がある. もし, ひとつのプリプロセッサの指定で出力言語が `pkl1` にならない場合は, 他のプリプロセッサの指定を連鎖的に用いて, 出力言語が `pkl1` になるまで試される. ただし, このプリプロセッサの連鎖は一巡のみであり無限に連鎖することはない.

入力言語名のみ指定された場合には, その言語に対するプリプロセッサの設定を解除する. また, 全ての引数を省略した場合は, 設定されているプリプロセッサを全て表示する.

【例】プリプロセッサの連鎖

```
!preprocess abc def pp1 pp2
!preprocess ghi jkl pp3 pp4
!preprocess jkl pkl1 pp5
```

というような指定がされていたとすると, 言語 abc をコンパイルすると結果的にプリプロセッサは, pp1 → pp2 → pp3 → pp4 → pp5 というように自動的に連鎖する.

デフォルトでは, “mac” に対しマクロコンパイラ, “kl1” に対しマクロ展開, また特別に”sav” に対してはローダが割り当てられている. “sav” に対するプリプロセッサの指定, 変更はできない. 入力言語は小文字の名前に正規化されるため, 大文字, 小文字の違いによる区別はできない.

【注意 1】PIMOS 2.5 版には !attach というコマンドが存在したが, 3.0 版からはこの !preprocess コマンドに変更された. 従来の !attach コマンドは, !preprocess コマンドにおいて出力言語が pkl1 を指定したものとみなされる.

【注意 2】3.0 版で追加された新コマンドであるが, !attach コマンドは暫定的に利用可能になっている.

!translate *FileName*

FileName で指定されたファイルのプリプロセッサ出力をファイルに出力する. 生成されたファイルは, 同じディレクトリの同じ名前で, 拡張子が最後のプリプロセッサの出力言語名となる.

【注意】このコマンドの結果としてモジュールは生成されない.

!unparser [*OutLang* [*UnparserModuleName*]]

OutLang で指定された出力言語に対応した清書用モジュール (UnparserModuleName) を指定する. 値を省略した場合は, 設定されているモジュール名を表示する.

- ベリファイアに関するコマンド

!verifier [*InLang* [*VerifierModuleName*]]

VerifierModuleName で指定されたベリファイアモジュールを InLang で指定された言語に割り当てる. モジュールの指定を省略すると, 対応刷する言語に対するベリファイアを取り除き, 引数全てを省略すると設定されているベリファイアを表示する.

【注意】3.0 版で追加された新コマンドである.

!verify *FileName*, ..

FileName で指定されたファイルに対しベリファイをかける. ベリファイはファイルの拡張子に対応したベリファイア (**!verifier** コマンド参照) か, プリプロセッサの連鎖における下流の言語のベリファイアが用いられる.
プリプロセッサの連鎖においてベリファイアが指定されていなければエラーになる.

【注意 1】このコマンドの結果としてモジュールは生成されない.

【注意 2】3.0 版で追加された新コマンドである.

- コンパイルオプション

!distribute ([(*PE#* | *PE#* : *N* | *PE#* − *PE#*), ..] | ∗)

コンパイラ及びローダにおける負荷分散のためのノード (プロセッサあるいはクラスタ) を指定する. 基本的な指定方法は「負荷分散ユーティリティ」と同じである. ∗ を指定すると全てのノードが対象となりる. ここで言う「負荷分散」では, セルフコンパイラの本体 (トップレベル) の負荷分散は行われない.
以下に, ノード 1 〜 7 を割り当てる指定方法の例を示す.

1. ノードの明示的な指定

```
COMPILE> !distribute 1, 2, 3, 4, 5, 6, 7
```

2. 分散開始ノードとノード数による指定

```
COMPILE> !distribute 1:7
```

3. 分散開始ノードと最終ノードの指定

```
COMPILE> !distribute 1-7
```

4. 上記の混在

```
COMPILE> !distribute 1, 2, 3:3, 6-7
```

これらの負荷分散は, ファイル単位に行われる. また指定されたノードは巡回的に使用される. なお, 実際のプロセッサ構成が, 指定のプロセッサ番号あるいはノード数に足りない場合は, あふれたノードの指定は無視される. また, 無意味な負荷分散あるいは特に負荷分散の指定がない場合は, カレントノードだけが対象となる (負荷分散は行われない).

!define [*Key Value*]

条件コンパイルのためのパラメタを設定する. ここで, Key はパラメタ名, Value はパラメタの値 (任意の基底項) である. 引数を省略した場合は, 設定されている全てのパラメタを表示する.

【注意】3.0 版で追加された新コマンドである.

!undefine *Key*

Key で指定された条件コンパイルパラメタの設定を解除する.

【注意】3.0 版で追加された新コマンドである.

!describe [*PreprocModuleName*]

PreprocModuleName で指定されたプリプロセッサに設定されているコンパイル変数を表示する. プリプロセッサ名を省略した場合は, KL1 コンパイラに設定されている変数を表示する. この変数の設定には, **!set** コマンド あるいは環境変数を用いること.

【注意 1】このコマンドによりプリプロセッサが変数をどのように表示するか, あるいは表示する／しないは全てプリプロセッサの責任である.

【注意 2】3.0 版で追加された新コマンドである.

!get [*ParameterName*]

ParameterName で指定されたコンパイル変数の値を表示する. 変数名が省略された場合は, 設定されている全ての変数が表示される.

【注意】一般に, このコマンドにより表示される変数が, プリプロセッサまたは KL1 コンパイラで有効かどうかは分からない. 有効かどうかを調べるには, **!describe** コマンドを用いること.

!message (*silent* | *verbose*)

エラーメッセージ出力の際に表示されるソースファイルにおけるエラーの発生場所の表示レベルを設定する．この設定が有効かどうかは，コンパイルする言語のパーサに依存する．

【注意】3.0 版で追加された新コマンドである．

!parallel [*N*]

ノード当たりにいくつのモジュールのコンパイルを割り当てるかという最大数を設定する．デフォルトは 2 である．

【注意】3.0 版で追加された新コマンドである．

!set [*ParemeterName* [=] [*Value*, ..]]

ParameterName で指定されたプリプロセッサのための変数に Value で指定された値を設定する．変数名はプリプロセッサに依存した任意の名前であり，その値についてもプリプロセッサに依存する．値を省略した場合は，設定されている値を解除する．なお，引数における “=” に意味はない．

【注意】このコマンドにより表示される変数が，プリプロセッサまたは KL1 コンパイラで有効かどうかは分からない．

以下に，予約されているコンパイル変数を列挙する．なお，これらの変数は，KL1 のコンパイルに関するものであり，プリプロセッサによる言語に依存したものではない．言語に依存した変数については，対応するプリプロセッサを参照のこと．

- **compile:indexing** インデキシングによる最適化をおこなうかどうかを指定する．値が on (アトム) でインデキシングをおこない，その他の値ではインデキシングコードは生成されない．デフォルトは on である．
- **compile:profile** コンパイラが生成するコード中に，クローズに関する情報を埋め込むかどうかを指定する．値が on (アトム) でクローズ情報を埋め込んだコードを生成し，それ以外ではクローズ情報が埋め込まれていないコード (従来はこのモードである) が生成される．デフォルトは off である．(このモードを on にしてコンパイルしたコードをトレースすると，クローズ番号が表示される．詳細は操作編リスナの章のトレーサ (124頁) 参照のこと)

- 再リンク，ロード／アンロード

!load *FileName*, ..

FileName で指定された，アンロードされたファイルをロードする．シェルから `load([FileName, ...] )` と入力したのと同じ．

!Relink *ModuleName*, ..

ModuleName で指定されたモジュールを再リンクする．シェルから `relink( [ModuleName, .. ] )` と入力したのと同じ．

!relink *TopModuleName ModuleName, ..*

TopModuleName で指定されたモジュールをモジュールから ModuleName で指定されたモジュールに至る参照関係にあるモジュールを全て探索して再リンクする. シェルから `relink( TopModule, [ModuleName, .. ] )` と入力したのと同じ.

【注意】コマンドまたは環境変数で指定された **working_packages** にデフォルトのパッケージが含まれていない場合は, モジュールの指定において明にパッケージを指定しないと再リンクの対象となるモジュールが存在しないことになるので注意を要する.

!Unload *FileName*

直前に (コンパイルなどで) ロードされたモジュール (複数) を FileName で指定されたファイルにアンロードする.

【注意】3.0 版で追加された新コマンドである.

!unload *ModuleName, .. FileName*

ModuleName で指定されたモジュールを FileName で指定されたファイルにアンロードする. シェルから `unload( [Mod1, Mod2, .. ], Filename )` と入力したのと同じ.

!unload *FileName*

コマンドあるいは環境変数で指定された **top_modules** をトップモジュールとし, **working_packages** で設定されたパッケージに属するモジュールを FileName で指定されたファイルにアンロードする.

【注意】コマンドまたは環境変数で指定された **working_packages** にデフォルトのパッケージが含まれていない場合は, モジュールの指定において明にパッケージを指定しないとアンロードの対象となるモジュールが存在しないことになるので注意を要する.

!working_packages ([*Package, ..*] | *)

ユーザが作業するパッケージを Package で指定する. この変数が指定されていると再リンクでは, 指定されているパッケージ内のモジュールのみを探索する. * を指定すると全てのパッケージが探索の対象となる. 特に指定のない場合は, デフォルトパッケージのみが探索範囲として設定される.

- その他

!alias [*Alias* [*Command*]]

Command で指定されたコマンドの別名を Alias で指定する. 引数を指定しない場合は別名の設定を全て表示する. 引数がひとつの場合は別名の設定の解除とみなされる. 別名の指定は, **keep** コマンドにより保存される. ここで, コマンド名, コマンド別名とも先頭の "!" は不要である.

【注意】別名の別名は付けることができない.
デフォルトでは, 下表のような別名が設定されている.

表 3.1: デフォルトの別名

別名	正式なコマンド
dstr	distribute
orgp	original_package
pkg	package
tm	top_modules
wp	working_packages

!exit

コンパイルを終了する.

!help

コマンド一覧を表示する.

!keep

設定されたコンパイルオプションを, カレントディレクトリの compile.param というファイルに保存する. ただし, カレントディレクトリの設定は保存されない (このファイルを他のディレクトリにコピーしても有効なものとするため).

【注意 1】このファイルに保存される内容やデフォルトの値が 3.0 版で変更されていることから, 2.5 版以前からのユーザは古いバージョンのものを消去し, 新たに生成することが望まれる. 3.0 版では, 念のため, 古いバージョンのパラメタファイルを読み込んだ場合には警告メッセージを出す.

【注意 2】このファイルは基本的にセルフコンパイラのコマンドにより作成・修正されるものとする. ユーザがテキストエディタ等により修正することも可能ではあるが, 誤った指定などによりコンパイラが起動できなくなることもあり得るので注意を要する. もしこのような場合に陥った時は, ファイル compile.param を消去すれば回避できる.

!revert

そのディレクトリに keep された状態, あるいはコンパイラを起動した状態に, コンパイラのオプション設定を戻す.
コンパイラ起動時にカレントディレクトリにあるファイル compile.param の内容が読み込まれる. カレントディレクトリに存在しない場合は, ホームディレクトリの同名ファイルが読み込まれる. カレントあるいはホームのいずれにも存在しない場合は, 環境変数の値が用いられる.

!show[(*alias* | *preproc* | *param*)]

主要なコンパイラの設定を表示する. 引数に alias[es] が指定された場合は別名の設定を全て表示し, preproc[ess[or[s]]] が指定された場合は設定されているプリプロセッサ, ベリファイア, パーサ, アンパーサを全て表示し, param[eter[s]] が指定された場合はプリプロセッサオプション (パラメタ) と条件コンパイルパラメタを全て表示する. 引数を省略した場合は, その他の主要な設定状況を表示する.

表 3.2: !keep コマンドで保存されるコンパイルオプション

コマンド	環境変数
—	compile:prompt
—	compile:more_prompt
alias	
auto_link	compile:auto_link
default_extension	compile:default_file_extension
distribute	compile:distribution
link_error	compile:link_error
message	compile:message
original_package	compile:original_package
package	task:package
parallel	compile:parallel
parser	compile:parsers
path	compile:path
preprocess	compile:preprocessors
set	compile:language_options
top_modules	compile:top_modules
unparser	compile:unparsers
verifier	compile:verifiers
working_packages	compile:working_packages

!take *FileName*

コマンド列を FileName で指定されたファイルから読み込む.

【参考】プリプロセッサのインストール方法を記述すると便利である.

【例】プリプロセッサのインストールの記述例

```
!compile my_preproc                % プリプロセッサをコンパイル
!Unload mypp                       % コンパイルしたモジュールをアンロード
!preprocess mylang kl1 my_preproc  % プリプロセッサを指定
!keep                              % プリプロセッサの指定を保存
```

【注意】3.0 版で追加された新コマンドである.

!unalias *Alias*

Alias で指定されたコマンドの別名の設定を解除する. ここで, コマンド別名における先頭の "!" は不要である.

3.1.4 環境変数

セルフコンパイラで使用している環境変数には以下のものがある.

compile:auto_link

コンパイル終了毎にリンクをおこなうかどうかを指定する. アトム on または off を設定する.

表 3.3: コマンドと環境変数の関係およびデフォルトの設定値

環境変数	コマンド	デフォルトの値
compile:auto_link	auto_link	on
compile:default_file_extension	default_extension	[“mac”,”kl1”,”pkl1”,”sav”]
compile:distribution	distribute	[ ]
compile:language_options	set	[ ]
compile:link_error	link_error	error
compile:ls_column	—	4
compile:message	message	verbose
compile:more_prompt	—	プロンプトと同じ長さの空白文字列
compile:original_package	original_package	on
compile:parallel	parallel	2
compile:parsers	parser	[{“kl1”,kl1_standard_preprocessor}, {“mac”,kl1_macro_preprocessor}]
compile:path	path	[ ]
compile:preprocessors	preprocessor	[{“kl1”,kl1_standard_preprocessor}, {“mac”,kl1_macro_preprocessor}, {“pkl1”,kl1_standard_preprocessor}]
compile:prompt	—	“COMPILE> “
compile:top_modules	top_modules	[ ]
compile:unparsers	unparser	[{“kl1”,kl1_standard_preprocessor}]
compile:verifiers	verifier	[{“pkl1”,kl1_standard_preprocessor}]
compile:working_packages	working_packages	[ ]
task:directory	cd	task:directory の値
task:package	package	task:package の値

compile:default_file_extension

コンパイルするファイル名の指定においてデフォルトのファイル名拡張子を指定する．拡張子のストリングを要素とするリストを設定する．

compile:distribution

コンパイルの負荷分散プロセッサを指定する． プロセッサの指定を (プロセッサ番号または, PEi-PEj, PEi:n, * といった項を要素とする) リストで設定する (詳細については, **!distribute**コマンド (46頁) を参照のこと)．

compile:language_options

プリプロセッサに対するコンパイル変数を設定する．変数名 (基底項) と任意の値 (基底項) を対とするベクタが要素であるようなリストを設定する．

compile:link_error

未定義モジュールあるいは未定義外部述語によるエラーを無視するかどうかを設定する．アトム error または warning を設定する．

compile:ls_coulmn

ディレクトリのリスティングをおこなう際のカラム数を設定する.

compile:message

コンパイラのメッセージ出力のレベルを設定する. アトム silent または verbose を設定する.

compile:more_prompt

継続行における入力プロンプト文字列を指定する.

compile:original_package

再リンクあるいはロードにおけるパッケージを指定する (詳細については, **!original_package** コマンド (44頁) を参照のこと). アトム on または off を指定する.

compile:parallel

コンパイルの負荷分散において, ノード当たりにフォークする最大のモジュール数を設定する. 正の整数を設定する.

compile:parsers

ユーザ指定言語の構文解析モジュールを設定する. 入力言語名 (ストリング) とモジュール名 (アトム) あるいは {**(::), Package, ModuleName**} という形式の項が対となったベクタを要素とするリストを設定する.

compile:path

ファイルをオープンする際の探索パスを指定する. パス名を要素とするリストを設定する.

compile:preprocessors

ユーザ定義プリプロセッサを指定する. 入力言語名 (ストリング) とプリプロセッサモジュールのリストを対としたベクタを要素とするリストを設定する.

compile:prompt

インタラクティブコンパイル時の入力プロンプト (ストリング) を指定する.

compile:top_modules

コンパイルにおける自動再リンクのためのトップモジュールを指定する. トップモジュール名あるいは {**(::), Package, ModuleName**} という形式の項を要素とするリストを設定する.

compile:unparsers

ユーザ指定言語の清書モジュールを設定する. 出力言語名 (ストリング) とモジュール名 (アトム) あるいは {**(::), Package, ModuleName**} という形式の項が対となったベクタを要素とするリストを設定する.

compile:verifiers

ユーザ指定言語の静的解析モジュールを設定する. 出力言語名 (ストリング) とモジュール名 (アトム) あるいは {**(::), Package, ModuleName**} という形式の項が対となったベクタを要素とするリストを設定する.

compile:working_packages

再リンクあるいはプログラム単位でのアンロードをおこなう際の探索するパッケージ範囲を指定する. パッケージ (アトム) を要素とするリストを設定する.

task:directory

デフォルトディレクトリを指定する. ディレクトリ名 (ストリング) を設定する.

task:package

デフォルトパッケージを指定する. 値はパッケージ名 (アトム) である. この指定によりコンパイルの結果として生成されたモジュールは, このパッケージに登録され, パッケージ指定のない外部参照はデフォルトパッケージを探索起点とする.

3.1.5 PIMOS が用意するプリプロセッサ

- **kl1_macro_expander**

  KL1 のマクロ展開用プリプロセッサである. 入力言語は `kl1` であり, 出力言語は, `pkl1`(plain KL1) である.

- **kl1_macro_preprocessor**

  ユーザ定義マクロのコンパイル用プリプロセッサである. 入力言語は `mac` であり, 出力言語は `pkl1` である. マクロ定義言語用の構文解析機能を併せ持つ.

- **kl1_standard_preprocessor**

  デフォルトの構文解析および清書機能を持つプリプロセッサである. 言語の変換という意味からは何も処理をおこなわない. 入力言語, 出力言語とも **kl1** である. 構文解析機能としては, オペレータの定義 (:- op ..), include の処理とシンタックスエラー検出をおこなう. 清書機能としては, 各項を KL1 の節とみなして適当な形式で出力する. また, 検証 (ベリファイ) 機能としては, 節単位で出現する変数をチェック (従来の varchk と同じ) する.

  以下に示すオプション (`!set, !describe` コマンド (47 頁) 参照のこと) が設定可能である.

  - **kl1:include_max**

    パーサにおけるインクルードファイルのネスティング最大数 (正整数) を設定する.

  - **kl1:print_mode**

    表示メッセージの詳細出力モード (**!message** コマンド (47 頁参照) で verbose を設定) において, ガードのみを表示するかどうかを設定する. アトム guard でガード部のみを, それ以外で節を全て表示する.

  - **kl1:pretty_printer**

    表示メッセージの詳細出力モード (**!message** コマンド (47 頁参照) で verbose を設定) において, 節の表示出力をおこなうコードを設定する.

表 3.4: 環境変数と参照しているプログラム

環境変数	compile	relink	unload	load
compile:auto_link	○			
compile:default_file_extension	○			
compile:distribution	○			○
compile:language_options	○			
compile:link_error	○		○	○
compile:ls_column	○			
compile:message	○			
compile:more_prompt	○			
compile:original_package		○		○
compile:parallel	○			
compile:parsers	○			
compile:path	○			○
compile:preprocessors	○			
compile:prompt	○			
compile:top_modules	○			
compile:unparsers	○			
compile:verifiers	○			
compile:working_packages	○	○	○	
task:directory	○		○	○
task:package	○	○	○	○

3.2 再リンカ

モジュール間のリンクを取り直すシェルプログラムである.

コンパイラは与えられたモジュールのみをソースコードから生成し, そのモジュールを参照しているモジュールには影響をおよぼさない. あるモジュールを再コンパイルした場合, そのモジュールを参照しているモジュールは, 古いモジュール (コンパイル以前のモジュール) を参照したままである. 正しくコンパイル結果を反映させるためには, そのモジュールを参照しているモジュール全てを再コンパイルする必要がある. この時, ただ単にリンクを張り直すためにいちいちコンパイルし直すのでは効率が悪い. そこで, モジュールのリンクだけを取り直すプログラムがこの再リンカーである.

一方, コンパイル時において, リンクエラーを無視し, 未定義モジュール参照が含まれていた時は, 再リンクにより未定義参照を解消することができる, 当然のことであるが, この時には, 参照されるモジュールが予め登録されていなければならない.

また, トップモジュールからのモジュール探索において, 環境変数 **compile:working_packages** で指定されたパッケージのみをモジュール探索の範囲とする.

【注意】トップモジュールからの探索においてリンクを遡る際, 一度でも環境変数 **compile:working_packages** で指定されたパッケージの外に出てしまったら, それ以降の探索はおこなわれないので注意を要する.

relink(ModuleNameList)

再リンカーを起動する. 引数の ModuleNameList はモジュール名 (アトム) を要素とするリストである. ModuleNameList に記述されたモジュールから他のモジュールへの参照を更新する.

【参考】親パッケージのモジュールを更新したいがバグを入れないように元のモジュールを保存しておきたいような場合には, この再リンクを用いることでモジュールのコピーを子パッケージに作ることができる.

relink(TopModuleName, ModuleNameList)

再リンカーを起動する. 引数の TopModuleName で指定されたトップレベルのモジュールから, 引数 ModuleNameList で指定されたモジュールに関係するモジュール全てを探しだし, 再リンクする.

それぞれの引数におけるモジュールの指定は, **Package::ModuleName** (ベクタ表記では, {**(::), Package, ModuleName**}) というような 3 要素ベクタ (複合項), あるいは単に ModuleName (アトム) である. ここで, Package はパッケージ名 (アトム), ModuleName はモジュール名 (アトム) である.

relink:package

デフォルトパッケージ内にある全てのモジュールを再リンクする.

relink:package(Package)

指定されたパッケージ (Package) 内にある全てのモジュールを再リンクする.

3.2.1 再リンカの使用方法

再リンカーは, シェルから以下のように起動する.

【例 1】再リンカの使用例 (その 1)

```
SHELL> relink([module_1, module_2, module_3])
```

モジュール module_1, module_2, module_3 を再リンクする.

【例 2】再リンカの使用例 (その 2)

```
SHELL> relink(top_module, [util_module_1, util_module_2])
```

トップレベルモジュール top_module から, 下位モジュール util_module_1, util_module_2 に至るまで関連するモジュールを全て再リンクする. この場合には, util_module_1 および util_module_2 は, 事前にセルフコンパイラでコンパイルされているものとする.

【参考】再リンカーを利用して, 異なるパッケージにモジュールをコピーすることができる. このためには, まずデフォルトパッケージをターゲットとするパッケージに設定し, 次に `relink:package( SourcePackage )` としてソースパッケージを指定すれば, ソースパッケージに含まれる全てのモジュールがコピーされる. モジュール単位でコピーする場合は, relink:go/1 を用いる.

【例 3】パッケージ単位のモジュールコピー
パッケージ src_package にある全てのモジュールをパッケージ dst_package にコピーする.

```
SHELL> setenv task:package = dst_package
SHELL> relink:package(src_package)
```

3.3 アンローダ

コンパイラで生成されたモジュールを, ファイルに保存するシェルプログラムである. ファイルに保存されたモジュールは, 後述するローダで再びモジュールとしてメモリ上にロード可能である.

アンロードされたファイルの拡張子は “sav” をデフォルトとし, 指定時に拡張子を省略すると自動的に”sav” という拡張子が付加される.

【注意】アンロードされたファイルは, バイナリ形式であるため SIMPOS のエディタ PMACS などで編集することはできない.

unload(ModuleNameList, FileName)

引数の ModuleNameList は対象となるモジュールを指定するためのモジュール名を要素とするリストであり, ここで指定されたモジュールを, 引数 FileName (ストリング) で指定されたファイルにアンロードする.

ModuleNameList の各要素におけるモジュール名の指定は, **Package::ModuleName** (ベクタ表記では, {**(::),Package,ModuleName**}) というような 3 要素ベクタ (複合項), あるいは単に ModuleName (アトム) である. ここで, Package はパッケージ名 (アトム), ModuleName はモジュール名 (アトム) である.

`unload(Package::*, FileName)` と指定することで Package (アトム) で指定されたパッケージ内にある全てのモジュールをアンロードすることができる.

unload:package(Package, FileName)

引数の Package (アトム) に指定されたパッケージに含まれる全てのモジュールを指定されたファイル FileName に保存する. この方式で保存されたモジュールのロード方法はモジュール名を指定することで生成された場合と同じである.

【注意】この方式で保存されたファイルをロードする場合には, モジュールは全て保存されたパッケージと同じパッケージに生成される. したがって, ロード時には, 同名パッケージが存在している必要がある.

unload:program(TopModuleNameList, FileName)

引数 TopModuleNameList で指定されたトップモジュールから呼び出され, かつ環境変数 **compile:working_packages** で指定されたパッケージに存在するモジュールを, 全て指定されたファイルにアンロードする.

TopModuleNameList の各要素におけるモジュール名の指定は, **Package::ModuleName** (ベクタ表記では, {**(::),Package,ModuleName**}) というような 3 要素ベクタ (複合項), あるいは単に ModuleName (アトム) である. ここで, Package はパッケージ名 (アトム), ModuleName はモジュール名 (アトム) である.

【例】アンローダの使用例

```
SHELL> unload([module_1, module_2, module_3], "module123")
```

モジュール module_1, module_2, module_3 をファイル “module123.sav” にアンロードする.

【参考】通常, モジュールをアンロードしてロードする時間は, セルフコンパイラで再コンパイルしたり, CAL を用いて IPL し直す時間よりも短くなる.

3.4 ローダ

アンローダで保存されたファイルを読み込んで, 元のモジュールを生成するシェルプログラムである. ロードされたモジュールは, デフォルトパッケージに生成される.

ローダでロードされたモジュールのリンク (ロードされたモジュールからの参照) は, ロード時に新しく張り直される. ただし, セルフコンパイラの場合と同じように, ロードされたモジュールを参照するリンクはロードの影響を受けないため, 再リンクが必要である.

また, ロードするモジュールに未定義モジュール参照が含まれていた場合は, 予め参照されているモジュールが登録されているならば, ロードにより未定義が解消される.

アンロードされたファイルの拡張子が"sav" であれば, ファイル名の指定において拡張子を省略することができる. また, セルフコンパイラ同様, 環境変数 **compile:path** の値に従って, アンロードされたファイルの探索をおこなう.

環境変数 **compile:distribution** の設定により, ファイル単位での負荷分散を指定できる.

【注意 1】 PIMOS 1.5 版でアンロードしたファイルは, PIMOS 2.0 版以降のローダではロードできない.

【注意 2】 ネットワークを経由したファイル転送などで, ファイルの一部が破壊されているようなファイルをロードした場合には, ローダがフェイルや例外あるいは無限ループに陥る可能性がある. もし, このような状態になったら, ファイルのサイズをチェックして, アンロードあるいはファイルのコピーを再度やりなおしてみたほうが良い.

load(FileNameList)

ローダを起動する. 引数 FileNameList はアンローダにより生成されたファイル名 (またはファイル名のリスト) を指定する.

【例】 load:go/1 の使用例

```
SHELL> load("module123")
```

ファイル名 "module123.sav" でアンロードされているモジュールをロードする.

3.5 パッケージ

package:define(Package,SuperPackage)

SuperPackage (アトム) を親パッケージとし, Package (アトム) で指定された名前のパッケージを生成する.

既に, 同じ名前のパッケージが存在したり, 親パッケージが存在しない場合には, パッケージは生成されない.

package:define(Package,SuperPackage,Protect)

SuperPackage (アトム) を親パッケージとすし, Package (アトム) で指定された名前のパッケージを生成する. ここで, 引数 Protect はそのパッケージに対する保護モード (整数) である. (プログラミング編 11.1 章資源の獲得 410頁を参照のこと)

既に, 同じ名前のパッケージが存在したり, 親パッケージが存在しない場合には, パッケージは生成されない.

package:protect(Package,Protect)

Package (アトム) で指定されたパッケージの保護モードを変数 Protect (整数) の値にする. (プログラミング編 11.1 章 資源の獲得 410頁を参照のこと)

package:listing

その時点で, 登録されているパッケージに関する情報をシェルの標準出力に出力する. 出力は左から, パッケージ名, 親パッケージ名, オーナ名 (またはユーザ ID), 保護モードである. (プログラミング編 11.1 章 資源の獲得 410頁を参照のこと) パッケージの情報が保護されていて取れない場合は, **** や ????????? として表示される.

【例】パッケージ情報の一覧表示

```
SHELL> package:listing
  root       ----   0     r--r--r--
  user       pimos  0     rwxrwxr--
  privilege  root   ****  ?????????
  pimos      root   0     rwxrwxr-x
  work       pimos  0     rwxrwxrwx
SHELL>
```

package:listing(Package)

その時点で, Package (アトム) で指定されたパッケージに登録されている全てのモジュールに関する情報を表示する.

package:listing(Package,Sort)

その時点で, Package (アトム) で指定されたパッケージに登録されている全てのモジュールに関する情報を表示する. Sort (アトム) は, モジュール一覧におけるソートのキーであり, name の場合は名前, file の場合はファイル名, time の場合はモジュールの生成日時により昇順にソートされる.

3.6 モジュール

モジュールの情報を表示したり, モジュールを削除するシェルユーティリティである. 以下の説明において, 引数の ModuleName とは, ベクタ表記では, {**(::),Package,ModuleName**}) というような 3 要素ベクタ (複合項), あるいは単に ModuleName (アトム) のことである. ここで, Package はパッケージ名 (アトム), ModuleName はモジュール名 (アトム) である.

module(ModuleName)

ModuleName (**Package::ModuleName** で指定されたモジュールで定義されている述語の一覧を述語名でソートして表示する.

【注意】現在は非公開述語も表示しているが, PIM/X においても非公開述語が表示されるかどうかは現時点では不明である.

module:info(ModuleName)

ModuleName で指定されたモジュールのタイムスタンプ情報を表示する.

module:where(ModuleNameAtom)

ModuleNameAtom (アトム) で指定されたモジュールがデフォルトパッケージからルートパッケージに遡るまで, どのパッケージで定義されているかを表示する.

module:remove(ModuleName)

ModuleName で指定されたモジュールの登録を削除する.

モジュール名 (アトム) の代わりに * を指定することで指定されたパッケージあるいはデフォルトパッケージに含まれるモジュール全てを削除することができる.

第 4 章

デバッギングツール

PIMOS が提供する KL1 プログラムのデバッギングツールにはリスナ (デバッガ) とインスペクタ, モニタがある. これらのツールを利用して, ユーザは対話的に且つシンボリックに, 作成した KL1 プログラムのデバッグを行なうことができる.

リスナはユーザの要求によりプログラム情報を取り出し実行するユーザインタフェースシステムである. KL1 プログラムの実行トレースやスパイ, デッドロック検出などの基本的なデバッグ機能を備えている. また, KL1 プログラムの性能評価ツールである ParaGraph(第 5.2 節 190 頁) に渡すための実行情報を計測し, ParaGraph を起動するためのユーザインタフェースも提供している.

インスペクタは任意の KL1 データの内容を調べるための調査システムであり, リスナのデバッグ機能を支援する. インスペクタはリスナやシェル (第 2.1 節 12 頁) から呼び出すことができ, プログラムの実行結果やトレース中に現れる KL1 データを直ちに調査することができる.

モニタはユーザが指定した KL1 データを監視しその具体化を待って報告する機能を有する. インスペクタとよく似たユーザインタフェースを提供している. モニタはリスナのトレーサから起動され, 主に, プログラムの実行に伴う未定義変数の具体化の状況を知りたい場合に利用する.

4.1 リスナ

リスナは対話的なプログラムの実行を行うためのユーザインタフェースシステムである．同様の機能を持つシェルとの違いは，シェルが完成したプログラムを実行する際のタスク管理に主眼を置いているのに対して，リスナはプログラム開発途上のデバッグのための機能に重点を置いている点にある．

4.1.1 機能概要

リスナは概略以下のような機能を持つ．

◎ **ゴール列の実行**

リスナのトップレベルからユーザ定義述語や組込述語からなるゴール列を実行することができる．リスナは，この実行をタスクとして管理する．また，割り込み (アテンションキー = Control-C) を入力することでタスクの実行を中断して，タスクの資源消費量や資源状態の問い合わせなどを行うことができる．

◎ **シェルユーティリティプログラムの実行**

リスナ上でもシェルと同様に，コマンドとしてシェルユーティリティプログラムを実行することができる．従って，プログラミング環境として不可欠であるコンパイラやリンカ，ローダ，アンローダ，KL1 プログラムの静的解析ツールである変数チェッカなどをコマンドとして起動することができる．

◎ **プログラムの実行に関する情報の提供**

リスナはタスクの実行が終了すると，その実行に対する資源消費量と実行時間の報告を行う．入力されたゴール列中に変数が含まれている場合にはその値も報告する．また，プログラムの実行により呼び出される述語毎のリダクション数とサスペンション数を計測したり，ノード間のメッセージ通信の頻度を計測することもできる．計測されたデータは ParaGraph(第 5.2 節 p.190 参照) を利用してグラフ出力を行うことが可能であり，ユーザは視覚的にプログラムの性能解析を行うことができる．

◎ **変数プール**

ユーザはゴールの実行に用いた変数や，トレース中に出力されている変数をリスナが持つ変数プールに格納することができる．この機能を利用して格納されている変数を別のゴールの引数に用いたり，インスペクタでその値を詳しく調べたりすることができる．

◎ **標準入出力**

リスナはタスクを起動する際には，タスクに対して標準入出力デバイス (ストリーム) を割当てる．これらのデバイスはユーザがゴール列と共に指定することが可能であるが，特に指定がなければ，リスナウインドウへのストリームを与える．この機能により，シェルから起動するようなプログラムのデバッグが容易にできる．

◎ **トレース機能**

リスナの提供するトレース機能により，指定したゴールのリダクションが進んでいく過程を見ることができる．通常のステップトレース機能の他に，指定した述語の呼び出しのみをトレースするスパイ機能や，ゴールの実行を一時的に保留しておく機能も用意している．また，トレース情報を任意のプロセス (ゴール) 毎に別々のウインドウに出力するための，マルチウインドウトレース機能も提供している．

◎ **インスペクタ，モニタの起動**

複雑な構造を持つデータの中身を調べるためのインスペクタを呼び出すことができる．これを用いることにより，リスナのトップレベルから変数の値を調べたり，トレース実行中にゴールの引数の状態を調べたりすることができる．また，モニタ機能を利用することで，リアルタイムに変数が具体化していく様子を眺めながらデバッグを行うことができる (第 4.2 節 p.157 参照)．

◎ **エラー処理**

ユーザの指定したゴールが失敗したり, 例外に対する処理を誤っていたりするなど, なんらかの異常を引き起こすと, リスナはエラーの報告を行う. この時ユーザは, インスペクタを用いてエラーの内容を詳しく解析したり, エラーを起こしたゴールの代わりに新しいゴールを指定して実行を継続することができる.

4.1.2 操作方法

4.1.2.1 起動

リスナはシェルより起動する. "[1]" のプロンプトでゴール列の入力待ちとなる. プロンプト中の番号は **イベント番号** であり, ユーザがゴール列を実行する度にカウントアップされる. イベント番号は既に入力済みのゴール列を再度表示したり, 実行する場合に利用する (第 4.1.2.6 節 p.69 を参照のこと). この状態を **リスナトップレベル** と呼ぶ. ユーザはリスナトップレベルより任意のゴール列を実行することができる.

リスナの起動の仕方には大別して二通りある. 一つはリスナを起動するシェルのウインドウとは独立したリスナ専用のウインドウを生成する方法である. このウインドウを **リスナウインドウ** と呼ぶ. もう一つはリスナを起動するシェルのウインドウ (つまり, シェルの標準入出力) を利用する方法である.

◎ **リスナウインドウを生成して起動**

listener

listener(Position, Size)

listener(Position, Size, Font)

listener(Position, Size, Font, File) [listen]

リスナを起動する. 指定された Position, Size, Font でリスナウインドウを作成し, コマンドファイル File に記述されているリスナコマンド群を, リスナ起動時のコマンドとして実行する.

各引数には次の値を指定できる.

Position: `mouse` または `at(X,Y)` (`X, Y` はドット数).
指定が不正な場合や引数が省略された場合には `mouse` が設定される.

Size: `mouse` または `char(Characters, Lines)` (`Characters`, `Lines` はそれぞれ文字数, 行数).
指定が不正な場合や引数が省略された場合には `mouse` が設定される.

Font: フォントファイルのパス名 (文字列).
指定が不正な場合や引数が省略された場合にはデフォルトのフォントが設定される.

File: コマンドファイルのパス名 (文字列).
コマンドファイルの指定がない場合には初期実行ファイルを検索して実行する. 指定されたコマンドファイルが存在しない場合や初期実行ファイルが存在しない場合には何も行わない. 以下に参照を記す.

コマンドファイル → 第 4.1.2.8 節 p.75
初期実行ファイル → 第 4.1.2.9 節 p.75

【例】シェルより 位置, サイズ, フォント, コマンドファイルを指定してリスナを起動する

```
Shell>> listener(at(0,0),char(50,10),">sys>font>test_11.font","lis.com") &
```

【注意】リスナウインドウは環境変数 **task:io_node** で指定した FEP 上に生成される.

◎ シェルウインドウを利用して起動

listener:listener

listener:listener(File)

リスナをシェルウインドウ上に起動する. File が指定された場合はコマンドファイルとして実行する. コマンドファイルの指定がない場合には初期実行ファイルを検索して実行する. リスナが起動されると, PIMOS の版数, マクロモジュール名, デフォルトパッケージ, カレントディレクトリが順に表示される. 以下に参照を記す.

コマンドファイル → 第 4.1.2.8 節 p.75
初期実行ファイル → 第 4.1.2.9 節 p.75
マクロモジュール → 第 4.1.2.4 節 p.65
デフォルトパッケージ → 第 4.1.2.3 節 p.65

【例】シェルウインドウ上にリスナを起動する.

```
Shell>> listener:listener
*** KL1 LISTENER (V 3.0 17-Oct-91) ***
MACRO MODULES     : [pimos]
DEFAULT PACKAGE   : pimos
CURRENT DIRECTORY : >sys>user>icot

[1]
```

【参考】リスナトップレベルからシェルのトップレベルを呼び出す時, つまり, リスナをシェルのバックグラウンドへ移行させるには **suspend** コマンドを実行すればよい. **suspend** コマンドについては 第 4.1.6.1節 p.92 を参照のこと.

4.1.2.2 環境変数

リスナが参照する環境変数には以下のものがある. なお, 環境変数については プログラミング編 第 11.1.7 節 p.401 を参照のこと.

task:directory

カレントディレクトリを指定する. ディレクトリ名はストリングである.

task:home

ホームディレクトリを指定する. ディレクトリ名はストリングである.

task:package

デフォルトパッケージを指定する. パッケージ名はアトムである.

task:io_node

入出力を行う I/O ノードの名称を指定する.

4.1.2.3 デフォルトパッケージの取り扱い

リスナは起動されると環境変数 **task:package** の値を参照する．この時の値がリスナ起動時のデフォルトパッケージとなる．デフォルトパッケージは **default_package** コマンドにより変更することが可能である．変更方法は 第 4.1.6.3 節 p.95 を参照のこと．リスナのデフォルトパッケージに関する取り扱いには次のような規則がある．

1. ゴールの入力やリスナコマンドの引数においてパッケージ名の指定を省略できる．
2. トレースや，エラー報告時にパッケージ名の出力が省略される．
3. デフォルトパッケージ内の述語はトレース実行時に **デフォルトのトレースモード** が ON になる．デフォルトのトレースモードについては第 4.1.7.4 節 p.123 を参照のこと．

4.1.2.4 ゴールのマクロ展開

リスナトップレベルよりゴールが入力されると，リスナは自身に指定されているマクロモジュールの展開規則に従ってゴールの展開を行う (マクロ機能については プログラミング編 第 7.3.4 節 p.245 を参照のこと)．リスナ起動時のマクロモジュールは **pimos** である．リスナのマクロモジュールは **change_macro** コマンドにより変更することが可能である．複数のマクロモジュールを指定することもできる．変更方法は 第 4.1.6.3 節 p.94 を参照のこと．リスナのマクロ展開に関する注意事項として，

1. リスナに指定しているマクロモジュールをユーザが更新した場合，更新された展開規則が自動的に適用される．言い換えれば，マクロモジュールを更新する度に **change_macro** コマンドを実行する必要はない．

   【参考】マクロモジュールは，リスナトップレベルから入力がある度に，毎回動的に参照される．

2. リスナに指定しているマクロモジュールをユーザが不正なマクロモジュールに更新したり (例えば，マクロモジュールを通常のモジュールに更新した時)，マクロモジュールを削除した場合には，以降，リスナトップレベルにおけるゴールの**マクロ展開は抑制される** (**無変換モードに移行する**)．このような場合にはリスナから警告が発せられるので，ユーザは正しいマクロモジュールをロードし **change_macro** コマンドによりマクロモジュールの設定をやり直すべきである．

   【注意】警告が発せられた場合，入力したゴールの実行は放棄される．たとえ，マクロ展開を必要としないようなゴールを入力した場合でも警告は発せられる．

   【例 1】マクロモジュールが不正なモジュールに更新された．

```
[3] layered#queen(8,Ans).
!WARNING! illegal macro module: my_macro
!CAUTION! switched to non-expansible mode.
```

   【例 2】マクロモジュールが削除された．

```
[2] module_table:cache_table([remove_module(my_macro,S)]).
 S = normal

[3] layered#queen(8,Ans).
!WARNING! non-existent macro module: my_macro
!CAUTION! switched to non-expansible mode.
```

4.1.2.5 ゴールの起動

ユーザはリスナトップレベルから, ユーザ定義述語や組込述語, コマンドなど一度に複数のゴールを起動できる.

◎ 引数の表記

ゴールの引数のデータ表記はプログラム中の KL1 データ表記法と同様である. KL1 のシンタクスについては プログラミング編 第 7.2 節 p.223 を参照のこと.

◎ 入力の終了

入力の終わりには “.” (ピリオド) とリターンを入力する.

◎ 公開述語の呼び出し

プログラム中で公開述語宣言 (パブリック宣言, 即ち, `:- public`) をしているユーザ定義述語を呼び出すゴールは, プログラム中でのモジュール間呼び出しと同様に次の形式で指定する.

1. **パッケージを指定しないとき (デフォルトパッケージ)**

```
ModuleName : PublicPredicateName(Arg, ...)
```

【例】デフォルトパッケージ内の公開述語の呼び出し

```
[1] layered:n_queens(8,Ans).
```

2. **パッケージを指定するとき**

```
PackageName :: ModuleName : PublicPredicateName(Arg, ...)
```

【例】パッケージ指定つきの公開述語の呼び出し

```
[1] sample::layered:n_queens(8,Ans).
```

3. **デフォルトモジュールを宣言しているとき**

```
PublicPredicateName(Arg, ...)
```

【例】デフォルトモジュール内の公開述語の呼び出し

```
[3] n_queens(8,Ans).
```

デフォルトモジュールについては 第 4.1.6.3 節 p.95を参照のこと.

◎ 局所述語の呼び出し

プログラム中で公開述語宣言をしていないユーザ定義述語を呼び出すゴールは次の形式で指定する.

1. **パッケージを指定しないとき (デフォルトパッケージ)**

```
ModuleName ! LocalPredicateName(Arg, ...)
```

【例】デフォルトパッケージ内の局所述語の呼び出し

```
[1] layered!queens(1,8,begin,Ans).
```

2. **パッケージを指定するとき**

```
PackageName :: ModuleName ! LocalPredicateName(Arg, ...)
```

【例】パッケージ指定つきの局所述語の呼び出し

```
[1] sample::layered!queens(1,8,begin,Ans).
```

3. **デフォルトモジュールを宣言しているとき**

```
! LocalPredicateName(Arg, ...)
```

【例】デフォルトモジュール内の局所述語の呼び出し

```
[3] !queens(1,8,begin,Ans).
```

デフォルトモジュールについては 第 4.1.6.3 節 p.95を参照のこと.

【注意 1】次版では, 局所述語を呼び出すための演算子 ! は変更される可能性がある.

【注意 2】将来, 局所述語はリスナトップレベルから呼び出すことができなくなる可能性がある.

◎ 組込述語の呼び出し

組込述語はプログラムの節のボディ中と同様に記述する.

【例 1】組込述語の呼び出し

```
[1] predicate_to_code(module#layered,n_queens,2,Code).
```

【例 2】整数の演算

```
[1] Priority := (Pri-Min)/(Max-Min)*4096.
```

【例 3】浮動小数点数の演算

```
[1] Priority $:= (float(Pri)-float(Min))/(float(Max)-float(Min))*4096.0.
```

◎ 複数ゴールの実行

1. **並列実行**
   複数のゴールを並列に実行する場合はゴールとゴールの間を “,” (カンマ) で区切る.

```
Goal1, Goal2
```

【例 1】複数のゴールを並列に実行する

```
[1] test!jump(Jump,Step), sample::test:hop(Hop), test:step(Hop,Step).
```

【例 2】組込述語の混在

```
[1] current_node(Node,Nodes), X := Nodes-Node, layered:n_queens(X,Ans).
```

2. **逐次実行**
   複数のゴールを逐次に実行する場合はゴールとゴールの間を “;” (セミコロン) で区切る.

```
Goal1; Goal2
```

Goal1 と Goal2 を逐次に実行する. Goal1 の実行が終了しない限り Goal2 の実行は開始されない.

【例】複数のゴールを逐次に実行する

```
[1] test!jump(Jump,Step); sample::test:hop(Hop); test:step(Hop,Step).
```

◎ 実行ノードの指定

リスナトップレベルから入力されるゴールは, 通常はリスナが起動されたノードで起動されるが, ゴール列の入力時にゴール毎にこれを任意に変更することができる. ゴールを実行するノードの指定は, プログラム中での実行ノードの指定と同様に次のように入力する.

```
Goal @ node( NodeNumber )
```

【例】実行ノードの指定

```
[1] sample:generator(S)@node(7), sample:consumer(S,0)@node(3),
    utility:display(0).
```

実行ノード指定の考え方については, プログラミング編 第7.2.7 節 p.229 を参照のこと.

◎ 実行優先度の指定

リスナトップレベルから入力されるゴールは, 通常はそのゴールが所属するタスク内の最高の優先度で実行されるが, ゴール列の入力時にゴール毎にこれを任意に変更することができる. ゴールの実行優先度の指定は, プログラム中での実行優先度の指定と同様に次のように入力する.

```
Goal @ priority( Rate )              %所属荘園内割合指定

Goal @ relative_priority( Rate )     %所属荘園内自己相対指定
```

【例】実行優先度の指定

```
[1] sample:generator(S), sample:consumer(S,0)@priority(4000),
    utility:display(0)@relative_priority(-2000).
```

実行優先度指定の考え方については, プログラミング編 第7.2.6 節 p.228 を参照のこと.

◎ 標準入出力

標準入出力は通常はリスナウインドウ (またはシェルウインドウ) に設定されているが, ゴール列の入力時にこれを任意に変更できる. 詳しくは 第 4.1.4 節 p.78 に述べる.

4.1.2.6 ゴール列の再表示と再実行

リスナは既に入力 (実行) 済みのゴール列の再表示や再実行を実現するために, 入力されたゴール列を管理するためのヒストリキューを持っている. ユーザは入力の履歴を参照することは勿論のこと, リスナのプロンプトとして表示されるイベント番号を利用して, 既に入力済みのゴール列を再度表示したり実行したりすることができる.

◎ ゴール列の再表示

既に入力済みのゴール列を再度表示させるには次の二つの方法がある.

! イベント番号
: イベント番号で指定されたゴール列を表示する. 詳しくは 第 4.1.6.1 節 p.86 を参照のこと.

!!
: 直前に入力されたゴール列を表示する. 詳しくは 第 4.1.6.1 節 p.87 を参照のこと.

【注意】現在 !"文字列 " はサポートしていない.

◎ ゴール列の再実行

既に入力済みのゴール列を再度実行させるには次の三つの方法がある.

イベント番号
: イベント番号で指定されたゴール列を再び実行する. 詳しくは 第 4.1.6.1 節 p.87 を参照のこと.

!
: 直前に入力されたゴール列を再び実行する. 詳しくは 第 4.1.6.1 節 p.87 を参照のこと.

イベント番号リスト
: イベント番号のリストで指定されたゴール列群を, 指定された順に逐次に実行する. 詳しくは 第 4.1.6.1 節 p.88 を参照のこと.

◎ 入力の履歴の参照

ユーザはリスナのヒストリキューの内容を参照したり, キューの長さを変更することができる.

history
: ヒストリキューに格納されているゴール列の入力の履歴を一覧表示する. 詳しくは 第 4.1.6.1 節 p.88 を参照のこと.

history_length
: リスナのヒストリキューの長さを変更する. 詳しくは 第 4.1.6.1 節 p.88 を参照のこと.

4.1.2.7 実行の中断

ユーザはアテンションキー (“**Control-C**”) を押下する[1]ことによって, タスクの実行を中断することができる. リスナはアテンションを検出すると, タスクの実行を中断し, ユーザからの指示を待つ. この際にはタスクが実行を開始してから停止するまでの時間 (即ち, 実行時間) も報告される. なお, 報告される実行時間の単位はミリ秒である.

【例】実行の中断

```
[37] bestpath(a,z,Path).
 ?? Interruption (1167807 msec): ?
```

中断されたタスクの実行に対して以下のキー操作が行える.

< *cr* >: continue
- `g`: gc & continue
- `r`: consumed resource
- `s`: status
- `@`: status all
- `l`: list variables
- `v`: display variables
- `t`: list traced goals
- `k`: list kept goals
- `m`: list monitored terms
- `a`: abort
- `b`: break
- `&`: background
- `h`: help

以下に各操作の意味を記す.

Continue (リターン)
実行を継続 (再開) する.

GC & Continue (g)
全ノードに対してガーベジコレクタを起動し実行を継続 (再開) する.

【参考】ユーザプログラムの実行がデッドロックに陥っているような場合に, この操作を行うと便利である. デッドロックについては 第 4.1.5 節 p.79 を参照のこと.

【例】デッドロックの検出

```
[1] add(_,_,_).
 ?? Interruption (1326 msec): g
 *** Deadlock detected during GC.
 Builtin> add(A,B,C)
  Caller> pimos::listener_cmd_builtin: do_command/ 1
    Mode> 0
 !! Exception (7254 msec):
```

Consumed resource (r)
タスクの実行が中断されるまでに消費した資源量 (リダクション数) を問い合わせる.
unknown のときは不明を表す.

【参考】資源消費量の問い合わせは生成したタスクのコントロールストリームに

[1] コントロールキーを押した侭で C を押下する

```
status([], task, detail, ^Result)
```

メッセージを流すことで問い合わせを行っている. PIMOS の資源管理部では status/4 メッセージを受け取ると荘園のレポートストリームに statistics/2 メッセージを流して資源消費量を問い合わせるが, 問い合わせ先の荘園の実行が終了しているような場合 (例えば親荘園の実行により子荘園がアボートされた瞬間など) に備えてタイムアウト管理を行っている[2]. unknown は荘園に対して資源消費量を問い合わせて, ある一定時間内にその返事が返ってこない場合に表示される.

Status (s)

タスクの状態を問い合わせる. 例えば以下のような情報が表示される.

```
%%% STATUS %%%
id      : 203
name    : "Task_35"
status  : normal([stop(none)])
limit   : 75000000 reductions
consume : 10129 reductions
[children] (id name status limit consume)
33 "Task_35_child" normal([]) 75000000 unknown
```

各フィールドの意味は以下の通り.

id: ゴールを実行しているタスクの相対資源識別子. 相対資源識別子については 第 11.1.6 節 p.401 を参照のこと.

name: タスク名. 通常はイベント番号を “Task_” にアペンドしたもの.

status: タスクの状態. {Cond,Stops} という形式のベクタが表示される. ただし, 状態不明 (例えば, 親荘園が停止中である場合) であるときには **unknown** が表示される. Cond はタスクの実行状態を表し, 以下のもののいずれかである.

normal: 正常に実行中 (資源不足の状態ではないこと)

resource_low: 資源不足の状態にある

terminating: 終了処理中

shut_terminating: 親タスクの強制終了による終了処理中状態である

Stops はタスクの停止理由のリストで, 各要素は stop(Reason) の形式をしている. Reason は停止理由を表すアトムであり, 通常はリスナが決定する.

limit: 資源消費量の上限値.

consume: タスクの実行が中断されるまでに消費した資源量 (リダクション数).
unknown のときは不明を表す.

children: 子タスクの情報も獲得した場合, 全ての対象子タスクに対する同様の情報.

Status all (@)

タスクの資源状態を問い合わせる. 例えば以下のような情報が表示される.

[2] さもないと PIMOS がデッドロックに陥ることになる

```
%%% STATUS ALL %%%
id   : 203
type : task
name : "Task_35"
[children] (id type name)
5 atom "atom"
8 atom "atom"
12 module "module"
15 file_request "file request"
16 window "window"
18 window "window"
19 file ">sys>csp>PIMOS.DIR>pimos.mac"
20 fep_timer fep
23 atom "atom"
```

各フィールドの意味は以下の通り.

id: 資源の相対資源識別子. 相対資源識別子については 第 11.1.6 節 p.401 を参照のこと.

type: OS 定義資源のタイプ

name: 資源名 (資源に与えられているコメント)

children: 子タスクの情報も獲得した場合, 全ての対象子タスクに対する同様の情報.

List variables (l)

リスナの変数プールに格納されている変数の一覧を表示する. 詳しくは 第 4.1.6.5 節 **list** コマンド p.103 を参照のこと.

【参考】リスナトップレベルにおいてゴールの引数として入力された変数 (その名前が下線で始まるものは除く) は, プログラムの実行が終了するまでその値が束縛されているので, 変数の値を記憶するモードには関係なく一覧を見ることができる.

Display variables (v)

リスナの変数プールに格納されている変数の一覧をその値と共に表示する.

【参考】リスナトップレベルにおいてゴールの引数として入力された変数 (その名前が下線で始まるものは除く) は, プログラムの実行が終了するまでその値が束縛されているので, 変数の値を記憶するモードには関係なく一覧を見ることができる.

List traced goals (t)

トレースを指定して実行したゴールのうち, このコマンドの実行時点ではまだリダクションされていないものを一覧表示する. 詳しくは 第 4.1.7.9 節 p.137 を参照のこと.

List kept goals (k)

トレース実行中に実行を保留したゴールの一覧を表示する. 詳しくは 第 4.1.7.9 節 p.135 を参照のこと.

List monitored terms (m)

モニタリングを指定した項のうち, このコマンドの実行時点ではまだ具体化の報告が行われていないものを一覧表示する. 詳しくは 第 4.2.5.2 節 p.181 を参照のこと.

Abort (a)

実行を放棄する.

Break (b)

実行を中断したままリスナをトップレベルの状態にする. この時, トップレベルには中断のレベルが表示される. 中断のレベルを下がるには **exit** を入力する.

【例】実行中断時のトップレベルの呼び出しと復帰

```
 ?? Interruption (112794 msec): b

{1}
[7] inspect(X).
{0, code$sample::layered:queens/4$662, {1,f,begin,K}}> me
   ...

{1}
[8] exit.
 ?? ------------ (112794 msec):
```

Background (&)

実行を継続したままリスナをトップレベルの状態にする．この時，トップレベルにはレベルが表示される．レベルを下がるには **exit** を入力する．

【参考 1】現在の仕様は暫定的なものである．将来はシェルのような本格的なマルチタスク管理機能を導入する予定である．

【参考 2】例えばユーザが PIMOS の提供するデバイス資源を利用する場合を考える．ジェネラルリクエストを獲得するためにリスナのトップレベルから次のような入力を行うと ...

```
[1] shoen:raise(pimos_tag#task, general_request, GR).
```

ゴールの実行は中断してしまう．既にこの現象に遭遇したことのあるユーザは多いであろう．これは PIMOS が，ユーザからの資源を生成，操作するためのメッセージが送られてくるのを待ち続けているためである[3]．ユーザがメッセージを流さない限り実行は再開しないし終了もしない．そこで，バックグラウンド機能を利用すると ...

【例】階層的に資源を獲得してウインドウを作成する．

[3]厳密に言うと，PIMOS ではユーザが誤って PIMOS (この場合はジェネラルリクエスト) に対して不正なメッセージを流したり，ストリームを捨ててしまったりした時に，ユーザの誤りが PIMOS にまで波及しないようにプロテクションのためのフィルタを，ユーザタスクの中にユーザゴールと共に挿入する．この場合はフィルタがユーザからのメッセージプロトコルが流れてくる (ストリームが閉じられる) のを待ち続けているため，ゴールの実行が中断している．

```
% 変数の値を覚えるモードを ON にし，値の報告モードを all にする．
[1] remember, display_variables(all).

% ウインドウリクエストデバイスを獲得して，
% バックグラウンドへ移行する．
[2] shoen:raise(pimos_tag#task, general_request, GR),
    GR=[window(normal(WR,_,_))].
 ?? interruption (1007 msec): &

% ウインドウデバイスを獲得する．
{1}
[3] WR=[create(normal(W,_,_))].
 GR = [window(normal([ & ],D,E))]
 WR = [create(normal(W,F,G))]
 W = W

% ウインドウを作成する．
{1}
[4] W=[reshape(mouse,mouse,_),activate(_)|W1].
 GR = [window(normal([ & ],E,[]))]
 W = [reshape(mouse,mouse,F),activate(G)|W1]
 WR = [create(normal([ & , & |W1],H,I))]
 W1 = W1

% バッファを挿入して，バックグラウンドへ移行する．
{1}
[5] buffer:interaction_filter(BUF,W1).
 ?? interruption (1102 msec): &

% ウインドウへの出力．
{2}
[7] BUF=[putb("ABCDEFG"),nl,flush(_)|B1].
 BUF = [putb("ABCDEFG"),nl,flush(G)|H]
 GR = [window(normal([ & ],I,[]))]
 W = [reshape(mouse,mouse,normal),activate(normal)|W1]
 W1 = W1
 WR = [create(normal([ & , & |W1],J,K))]
 B1 = H

% ストリームを閉じてタスクを終了する．
{2}
[8] B1=[putt(end),nl,getl(L)].
 B1 = [putt(end),nl,getl(L)]
 BUF = [putb("ABCDEFG"),nl,flush(normal),putt(end),nl,getl(L)]
 GR = [window(normal([ & ],H,[]))]
 W = [reshape(mouse,mouse,normal),activate(normal),putb("ABCDEFG
",normal)|I]
 W1 = [putb("ABCDEFG
",normal)|I]
 WR = [create(normal([ & , & , & |I],J,K))]
 L = L
```

Help (h)

ヘルプメニューを表示する．

4.1.2.8 コマンドファイル

リスナのコマンド入力は, 通常はリスナトップレベルから行うが, これを一時的にファイルに切り替え, ファイル中に書かれたコマンド列を実行することができる. コマンドファイルの実行は **take** コマンドにより行う. 詳しくは 第 4.1.6.1 節 p.89 を参照のこと.

4.1.2.9 初期実行ファイル

リスナの起動時に自動的に実行するコマンドファイルである. ファイル名は "**listener.com**" に固定である. リスナ起動時にカレントディレクトリを検索する. ファイルが存在しない場合には何もしない.

4.1.2.10 プログラムの実行に関する情報の獲得

ユーザはリスナトップレベルから実行したプログラムについて, 幾つかの実行に関する情報を獲得することができる. リスナが提供する情報は次の通り.

1. **資源消費量 (総リダクション数) と実行時間 (ミリ秒)**
   プログラムの実行に関する情報を報告するモードを ON にする. モードの切り替えは **statistics** コマンドと **nostatistics** コマンドにより行う. 詳しくは 第 4.1.6.3 節 p.93 を参照のこと.

2. **実行状況情報**
   実行状況情報は ParaGraph 関連コマンドで計測する. 計測された情報は **paragraph** コマンドで ParaGraph を起動してグラフ出力を行うことができる. 現在, 実行状況情報として以下のものが計測可能である.

   (a) **述語毎のリダクション数とサスペンション数**
   profile コマンドで計測する.

   (b) **ノード間のメッセージの送受信情報と各ノードの稼働状況**
   profile_msg コマンドで計測する.

   (c) **各ノードの計算時間, アイドル時間, メッセージの送受信時間**, GC **実行時間**
   profile_node コマンドで計測する.

   なお, コマンドについては 第 4.1.6.9 節 p.113 を, ParaGraph については 第 5.2節 p.190 を参照のこと.

3. **実行途中のプログラムの状態**
   Control-C の入力による実行中断時のコマンドにより, プログラムの実行状態や資源消費量, OS 定義資源の使用状況などを知ることができる. 詳しくは 第 4.1.2.7 節 p.70 を参照のこと.

4. **ゴールの引数に用いた変数の値**
   ゴールの引数に用いた変数は, その名前が大文字で始まるものに限り, プログラムの実行が正常に終了するとその値が報告される. なお, 変数の値の報告モードは **display_variables** コマンドにより切り替えることができる. 詳しくは 第 4.1.6.5 節 p.102 を参照のこと.

4.1.2.11 終了

リスナトップレベルから **exit** を入力する. リスナと共にインスペクタやモニタ, ParaGraph が起動している場合は, リスナが終了するとそれらも終了する.

4.1.3 変数プール

リスナは変数を格納しておくための環境として **変数プール** と呼ばれるプールを持っている．ユーザはゴールの引数として用いた変数やトレース出力中に現れた変数を，リスナの変数プールに格納することができる．「変数を変数プールに格納する」とは「変数への値の束縛を保持する」ということである．変数プールに格納されている変数はユーザがそれを開放しない限り値が束縛されており，リスナトップレベルにおいてゴールの引数として何度でも利用できる．

4.1.3.1 変数の表記法

ゴールの引数として用いた変数を変数プールに格納したり，ゴールの実行終了時にその値を報告させるにはそれぞれモードがあるが，変数の名前の付け方により，これらのモードを全く関知しないような変数も利用することができる．

1. **名前が大文字で始まる変数**

   ```
   X, Priority, CODE, X1, Ans_1, ...
   ```

   変数の値を記憶するモード，実行終了時に値を報告するモード，問い合わせを行うモードに従う．

2. **名前が下線で始まる変数**

   ```
   _, _X, _priority, _x1, _Ans_1, __XX, ...
   ```

   モードに関係なく変数プールには格納されず，ゴールの実行が終了してもその値は報告されない．

   【例】名前が下線で始まる変数

   ```
   [1] _X=2, XX := _X*_X.
    XX = 4
   ```

   変数 `XX` は記憶モード，報告モード，問い合わせモードに従うが，変数`_X` はそれぞれのモードとは全く無関係であり，変数プールに格納されることはなく，その値も報告されない．

4.1.3.2 変数の格納方法

1. **変数の値を記憶するモードを** ON **にする**
   ゴールの引数として用いた変数を変数プールに格納するにはモードがある．このモードを ON にすると，ゴールの引数として用いた変数は，そのゴールの実行が成功すると自動的に変数プールに格納される．リスナ起動時の既定値は OFF である．モードの切り替えは **remember** コマンドと **forget** コマンドにより行う．詳しくは 第 4.1.6.5 節 p.99 を参照のこと．

2. **変数名の直前に**“^” (hat) **を付ける**
   リスナトップレベルにおいて，入力する変数名の直前に“^”を付けると，変数プールに格納するモードが OFF の時でも，その変数を変数プールに格納することができる．

   ^変数名

   【例】“^”を付けて変数を格納する (現在 forget モードとする)．

```
[1] samp:length([a],M), samp:length([a,b,c],^Len).  % Len だけを格納する.
 M = 1
^Len = 3

[2] value.  % 格納されている変数を一覧表示する (後述).
^Len = 3

[3] .       % 格納されている変数の値の束縛を解放する (後述).

[4] ^X=1, Y=2, ^Z := X+Y.
^X = 1
 Y = 2
^Z = 3

[5] value.  % 変数 Y は格納されていない.
^X = 1
^Z = 3

[6] X=Y.
 Y = 1
```

【注意 1】名前が下線で始まる変数に“^”を付けても, その変数は変数プールに格納されない (“^”は無視される).

```
[1] ^X=1, ^_Y=2.
^X = 1

[2] value.
^X = 1
```

【注意 2】変数の値を記憶するモードが ON の時には“^”の有無は意味を持たない.

3. **トレース中のコマンドを用いる**
   トレーサの **rem** コマンドにより, トレース出力中に現れた変数を格納することができる. 詳しくは 第 4.1.7.9 節 p.141 を参照のこと.

4. **インスペクタのコマンドを用いる**
   インスペクタの **name** コマンドにより, 調査中のデータを任意の変数名で格納することができる. 詳しくは 第 4.2.4.5 節 p.177 を参照のこと.

4.1.3.3 格納されている変数の一覧

変数プールに格納されている変数の名前や値を表示するには **value** コマンドや **list** コマンドを用いる. 詳しくは 第 4.1.6.5 節 p.103 を参照のこと.

4.1.3.4 変数の解放

変数プールに格納された変数を解放するには二つの方法がある.

1. **全ての変数を解放する**
   “.”, “cr” (ピリオドリターン) を入力すると. 変数プールに格納されている全ての変数が解放される. 詳しくは, 第 4.1.6.5 節 p.101 を参照のこと.

2. **個別に変数を解放する**
   forget(変数名) コマンドを利用すると, 個別に変数を解放することができる. 詳しくは 第 4.1.6.5 節 p.101 を参照のこと.

4.1.3.5 実行終了時の変数の値の表示

タスクの実行終了時に, ゴールの引数として用いた変数の値を表示させるにはモードがある. モードの切り替えは **display_variables** コマンドにより行う. 詳しくは 第 4.1.6.5 節 p.102 を参照のこと.

4.1.3.6 実行に使用した変数に対する問い合わせ

タスクの実行終了時に, ゴールの引数として用いた変数に対して幾つかの問い合わせを行うことができる. 変数値の参照やインスペクト, 変数プールへの格納などである. 問い合わせモードの設定は **query** コマンドにより行う. 詳しくは 第 4.1.6.5 節 p.102 を参照のこと.

4.1.4 標準入出力

リスナはタスクに対して 標準入力, 標準出力, 標準メッセージ出力, 標準インタラクション, 標準入出力の 5 つのデバイス (ストリーム) を用意している. 標準入力, 標準出力, 標準メッセージ出力, 標準入出力に関しては, ユーザは任意のデバイスを指定することが可能である.

【参考】リスナが提供する 標準出力, 標準メッセージ出力, 標準インタラクション, 標準入出力ストリームのホロフラスティングの出力制限値 (構造体の深さ, 長さ, ストリングの長さ, アトム名の長さ) の初期値は, 各々, ゴール実行時のリスナに設定されている値である. また, パーズ / アンパーズに用いる演算子は入出力用フィルタの標準演算子が定義してある. 標準演算子の一覧は 表 9.1 p.347 を参照のこと.

標準入出力は, 通常はリスナウインドウに設定されているが, これをユーザが指定する場合にはリスナトップレベルにおいて下記の演算子を用いてゴール列の後に記述して指定する.

標準入力: <= (デバイス)

標準出力: => (デバイス)

標準メッセージ出力: -> (デバイス)

またデバイスは以下のように指定する.

ファイル: `file( FilePathnameString )`
ファイルを **read** 若しくは **write** モードでオープンする. FilePathnameString は文字列.

ファイル: `file( ^ FilePathnameString )`
ファイルを **append** モードでオープンする.

【注意】PIMOS 3.0 版より, ファイルの write オープンと append オープンの指定方法 (“^” の有無) が入れ替わった.

ウインドウ: `window( WindowNameString )`

ウインドウ: `window( WindowNameString, Position, Size )`

ウインドウ: `window( WindowNameString, Position, Size, Font )`

ウインドウ: `window( WindowNameString, X, Y, W, H )`

ウインドウ: `window( WindowNameString, X, Y, W, H, Font )`
WindowNameString はウインドウのタイトル (文字列). X, Y はウインドウの位置 (ドット), W, H はウインドウのサイズ (文字数と行数), Font は表示に使用するフォントファイルのパス名 (文字列) である. また, Position には `mouse` または `at(X,Y)` を, Size には `mouse` または `char(W,H)` を指定する.

標準入力, 標準出力, 標準メッセージ出力は, それぞれ一度に入力するゴール列 (タスク) に対して高々一つずつ指定することができる.

【例 1】リスナの標準入出力を利用する.

```
[1] test1:start, test2!start.
```

【例 2】標準メッセージ出力としてファイルを指定する.

```
[1] test1:start, test2!start -> file(^ "mes.doc").
```

【例 3】標準入力, 標準出力, 標準メッセージ出力すべてを指定する.

```
[1] test:start -> file("mes.doc") <= window("in") => window("out").
```

タスク中からの標準入出力ストリームの獲得方法, メッセージプロトコルについては プログラミング編 第 9.4 節 p.375 を参照のこと.

4.1.5 デッドロック検出

4.1.5.1 概要

デッドロックとは, あるゴールの実行が中断し KL1 における通常の処理の範囲では決して再開されることのない行きづまりの状態をいう. 具体的には, どこからも参照されていない (参照パスがない) 変数の具体化を待つゴールの実行は永久に中断する.

一つのゴールがデッドロックに陥ると, そのゴールが生成すべきデータに依存関係があるゴール群も全てデッドロックに陥る可能性が高い. このため, 一般にデッドロックゴールの数は多くなり易い. デッドロックに陥っているプログラムをデバッグする時に実際に必要となるのは, デッドロックの因果関係の中で “因” となっているゴール群を検出することである. このようなデッドロックの原因となっているゴールをここでは “**極大ゴール**” と呼ぶ. PIMOS において報告されるデッドロックゴールは全て極大ゴールである. また, デッドロックの因果関係においてゴール群が循環構造をなしている場合には, どのゴールも依存関係において同値であるため, そのうちの一つのゴールが代表として報告される.

プログラムの実行の振る舞いを見て, ソースプログラムよりデッドロックを検出することは至難の技であるが, この機能によりプログラムのデバッグ効率は飛躍的に向上する.

【参考】デッドロックの予防策としてコンパイラが提供する標準の **ベリファイア** または **変数チェック** ユーティリティの利用を勧める. これらはプログラム節中に一度しか出てこない変数 (所謂ボイド変数) を静的に解析して検出するツールである. このチェックを行うだけでもかなりの効果がある. ベリファイアについては 第 3 章 p.38 を, 変数チェックユーティリティについては 第 2.2 節 p.31 を参照のこと.

4.1.5.2 デッドロックゴールの検出機能

PIMOS が提供するデッドロックゴール検出機能には二つのものがある. 一つは一括 GC (ガーベジコレクタ) の際に発見されるデッドロックであり, もう一つは, インクリメンタル GC により実行時に発見さ

れるデッドロック[4]である. 一括 GC の場合には, プログラムの実行がデッドロックに陥っていればひとつのノード内についてはほとんどの場合それが発見されるが, インクリメンタル GC 実行時の検出機能では, ある条件のものしか発見されない.

◎ 一括 GC によるデッドロックゴールの検出

一括 GC を利用したデッドロックゴールの検出を行うためには, ユーザプログラムがデッドロックに陥っている時に一括 GC が起動されなければならない. 一括 GC はメモリの空き領域が不足すると自動的に開始されるものであるが, 永久中断ゴールの検出を積極的に行うためには一括 GC も積極的に起動する必要がある. 一括 GC を積極的に起動するには次の方法がある.

1. ユーザプログラムの実行を中断して (Control-C によるアテンションの入力) 実行中断時の g コマンドにより一括 GC を起動する. 詳しくは第 4.1.2.7 節 p.70 を参照のこと.
2. シェルより一括 GC (**gc** コマンド) を起動する.

【参考】プログラムの実行が止まっていることを確認するには次の方法がある.

1. Pseudo Muilti-PSI では, CSP のサイレントモードが OFF の時 「マルチ PSI-CSP 状態監視パネル」 が黒くなっている.
2. Pseudo PIM/m では, CSP のサイレントモードが OFF の時 「マルチ PSI-CSP 状態監視パネル」 に **"idle"** の文字が表示されている.
3. ユーザプログラムの実行を何度か中断して (Control-C によるアテンションの入力) 実行中断時の **r** コマンドにより実行の資源消費量が増えていないことを確認する. 詳しくは 第 4.1.2.7 節 p.70 を参照のこと.

【注意】次のような場合にはデッドロックは検出されない

【例】変数プールの影響

```
[1] add(X,1,_).
```

変数 X が未定義の時, 組込述語 add/3 の実行は中断するが, 変数 X はリスナの変数プールに格納されているために (つまり, 変数プールから変数 X への参照パスが存在するため) デッドロックとしては取り扱われない. ユーザがデッドロックのデバッグを行う場合には下線で始まる名前の変数をうまく利用することが大切である.

4.1.5.3 デッドロックゴールの種類とその出力例

◎ 一括 GC により検出されるデッドロックゴール

- **一括 GC でユーザ定義述語のデッドロックが発見された**

```
[1] shoen:raise(pimos_tag#task, general_request, _GR).
*** Deadlock detected during GC.
Goal> pimos:: genreq_filter:genreq_filter(A,B,C,[])
Node> 0
```

[4] MRB (Multiple Reference Bit) の情報を利用したインクリメンタル GC により発見される. インクリメンタル GC では, データオブジェクトに対する参照数を実行時に計数することによって, そのデータオブジェクトに対する参照ポインタがなくなったことを検出し, データオブジェクトが使用していたメモリ領域を回収する. MRB は 1 ビットのカウンタであり, データオブジェクトに対する参照が "1" であるか "多数" であるかを判定する. 例えば, あるデータオブジェクトが参照数 "1" で生成され, その後参照数に変化がなければ, そのデータを消費したプロセスがデータオブジェクトを回収できる (所謂 MRB 白). 一方, 参照数が一旦 "多数" になると, その後は参照するものがなくなっても検出できないため, 不要データの回収は不完全になる

- **一括 GC で組込述語のデッドロックが発見された**

```
[1] add(X,_,Y), divide(Y,_,Z), modulo(Z,_,_).
 *** Deadlock detected during GC.
 Builtin> add(D,E,F)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
```

この例では組込述語 add/3 が極大ゴールとして検出されている.

- **一括 GC でマージャデッドロックが発見された**

```
[1] merge({_,[koichi],_,[nakao],_},OUT).
 *** Merger deadlock detected during GC.
 *** 3 merger inputs were abandoned.
 Abandoned> {B,C,D}
    Caller> pimos:: listener_cmd_builtin: do_command/ 1
      Node> 0
 !! Exception (6307 msec): e              % 代替ゴールを実行する（後述）.
 >Please execute substitutive goals.
 |?- B=[27], C=[10], D=[1961].            % マージャの入力引数を閉じる.
 OUT = [koichi,nakao,27,10,1961]          % マージャの実行は正常に終了.
```

マージャのデッドロックの検出では, 具体化されないまま捨てられたマージャの入力引数と, その組込述語 merge/2 を呼び出しているユーザ定義述語が報告される. 報告中の “Abandoned>” が示す変数 “B”, “C”, “D” が具体化されないまま捨てられたマージャの入力引数である. なお, 例中の「代替ゴールの実行」については 第 4.1.8.1 節 p.147 を参照のこと.

【参考】具体化されないまま捨てられたマージャの入力引数の利用法
この例からも分かるように, マージャのデッドロック発生時に報告される具体化されないまま捨てられたマージャの入力引数は, 実は例外を起こしたマージャに正しいデータを送るための入力変数となっている. 従って, ユーザはこれらの入力変数を任意のデータに具体化すればマージャの実行を再開することができる.

◎ インクリメンタル GC により検出されるデッドロックゴール

- **変数の具体化を待っているゴールがあるのに, その変数を具体化するゴールがなくなった**[5] **(collect_value)**

```
% collect_hook :- true | p(X), q(X).
% p(a) :- true | true.
% q(_) :- true | true.
%
% p/1 が q/1 よりも先に実行された場合.

[1] test:collect_hook.
 *** Deadlock occurred. [collect_value(HOOKo)]
 Goal> test: p(A)
 With> test: q/ 1
 Node> 0
```

- **マージャの入力変数を具体化するゴールがなくなった**[6] **(collect_value)**

[5] HOOKo を指す白パスを collect した.
[6] MGHOK を指す白パスを collect した

```
% collect_mghook :- true | merge({X},_), q(X).
% q(_) :- true | true.
%
% merge/2 が q/1 よりも先に実行された場合.

[1] test:collect_mghook.
 *** Merger deadlock occurred. [collect_value(MGHOKo)]
 Abandoned> {A}
    Caller> test: collect_mghook/ 0
      With> test: q/1
      Node> 0
```

```
% collect_mghook(Node) :- true | merge({X},_), q(X)@node(Node).
% q(_) :- true | true.
%
% merge/2 が q/1 よりも先に実行され, かつ,
% q/1 は merge/2 とは別のプロセッサで実行された場合.

[1] current_node(Node,_), test:collect_mghook(~(Node+1)).
 *** Merger deadlock occurred. [collect_value(MGHOKo)]
 Abandoned> {B}
    Caller> test: collect_mghook/ 1
      Node> 0
```

【注意】処理系の実装上の都合により, collect_value 命令によりマージャの永久中断が検出された場合で, 上記の例のようにマージャの入力変数への参照がマージャを実行しているノードとは別のノードで消滅した場合, その入力変数を捨てたゴールを検出することができない.

- **他のゴールによって具体化されることのない変数の具体化を待とうとした[7](suspend)**

```
% hook_hook_1 :- true | p(X), q(X).
% p(0) :- true | true.
% q(1) :- true | true.

[1] test:hook_hook_1.
 *** Deadlock occurred. [suspend(HOOKo)]
 Goal> test: p(A)
 With> test: q/ 1
 Node> 0
 !! Exception (115 msec):
 *** Deadlock occurred. [suspend(HOOKo)]
 Goal> test: q(A)
 With> test: p/ 1
 Node> 0
```

[7] HOOKo を指す白パスにゴールが hook した

```
% hook_hook_2 :- true | add(X,1,_), q(X).
% q(1) :- true | true.

[1] test:hook_hook_2.
 *** Deadlock occurred. [suspend(HOOKo)]
 Builtin> add(A,1,B)
  Caller> test: hook_hook_2/ 0
    With> test: q/ 1
    Node> 0
 !! Exception (116 msec):
 *** Deadlock occurred. [suspend(HOOKo)]
 Goal> test: q(A)
 With> add called from test: hook_hook_2/ 0
 Node> 0
```

```
% hook_hook_3 :- true | add(X,1,_), multiply(X,2,_).

[1] test:hook_hook_3.
 *** Deadlock occurred. [suspend(HOOKo)]
 Builtin> add(A,1,B)
  Caller> test: hook_hook_3/ 0
    With> multiply called from test: hook_hook_3/ 0
    Node> 0
 !! Exception (117 msec):
 *** Deadlock occurred. [suspend(HOOKo)]
 Builtin> multiply(A,2,C)
  Caller> test: hook_hook_3/ 0
    With> add called from test: hook_hook_3/ 0
    Node> 0
```

【注意】suspend 命令によりデッドロックゴールが検出された場合は, デッドロックの原因となったゴールが所属する各々の荘園に, 別々に報告される (必ず, 2 件のデッドロックが報告される).

- **自分だけしか参照していないマージャの入力変数の具体化を待とうとした[8](suspend)**

```
% mghook_hook_1 :- true | merge({X},_), q(X).
% q(1) :- true | true.

[1] test:mghook_hook_1.
 *** Merger Deadlock occurred. [suspend(HOOKo)]
 Abandoned> {}
    Caller> test: mghook_hook_1/ 0
      With> test: q/ 1
      Node> 0
 !! Exception (115 msec):
 *** Deadlock occurred. [suspend(MGHOKo)]
 Goal> test: q(A)
 With> merge called from test: mghook_hook_1/ 0
 Node> 0
```

[8] MGHOKo を指す白パスにゴールが hook した

```
% mghook_hook_2 :- true | merge({X},_), add(X,1,_).

[1] test:mghook_hook_2.
 *** Merger Deadlock occurred. [suspend(HOOKo)]
 Abandoned> {}
    Caller> test: mghook_hook_2/ 0
      With> add called from test: mghook_hook_2/ 0
      Node> 0
 !! Exception (116 msec):
 *** Deadlock occurred. [suspend(MGHOKo)]
 Builtin> add(A,1,B)
  Caller> test: mghook_hook_2/ 0
    With> merge called from test: mghook_hook_2/ 0
    Node> 0
```

- **具体化が待たれている変数同士をユニファイした[9](unify)**

```
% hook_hook_unify :- true | p(X), q(Y), r(X,Y).
% p(0) :- true | true.
% q(1) :- true | true.
% r(X,Y) :- true | X=Y.
%
% p/1 と q/1 が r/2 よりも先に実行された場合.

[1] test:hook_hook_unify.
 *** Deadlock occurred. [unify(HOOKo,HOOKo)]
 Goal> test: p(A)
 With> test: q/ 1
 Node> 0
 !! Exception (117 msec):
 *** Deadlock occurred. [unify(HOOKo,HOOKo)]
 Goal> test: q(A)
 With> test: p/ 1
 Node> 0
```

【注意】unify 命令によりデッドロックゴールが検出された場合は, デッドロックの原因となったゴールが所属する各々の荘園に, 別々に報告される (必ず, 2 件のデッドロックが報告される).

- **マージャの入力変数と具体化が待たれている変数をユニファイした[10](unify)**

[9] HOOKo を指す白パス同士を active-unify した

[10] MGHOKo を指す白パスに HOOKo を指す白パスを active-unify した

```
% mghook_hook_unify :- true | merge({X},_), q(Y), r(X,Y).
% q(1) :- true | true.
% r(X,Y) :- true | X=Y.
%
% merge/2 と q/1 が r/2 よりも先に実行された場合.

[1] test:mghook_hook_unify.
 *** Merger Deadlock occurred. [unify(MGHOKo,HOOKo)]
 Abandoned> {}
    Caller> test: mghook_hook_unify/ 0
      With> test: q/ 1
      Node> 0
 !! Exception (118 msec):
 *** Deadlock occurred. [unify(HOOKo,MGHOKo)]
 Goal> test: q(A)
 With> merge called from test: mghook_hook_unify/ 0
 Node> 0
```

- **マージャの入力変数同士をユニファイした[11](unify)**

```
% mghook_mghook_unify :- true | merge({X},_), merge({Y},_), r(X,Y).
% r(X,Y) :- true | X=Y.
%
% 二つの merge/2 が r/2 よりも先に実行された場合.

[1] test:mghook_mghook_unify.
 *** Merger Deadlock occurred. [unify(MGHOKo,MGHOKo)]
 Abandoned> {A}
    Caller> test: mghook_mghook_unify/ 0
      With> merge called from test: mghook_mghook_unify/ 0
      Node> 0
 !! Exception (119 msec):
 *** Merger Deadlock occurred. [unify(MGHOKo,MGHOKo)]
 Abandoned> {B}
    Caller> test: mghook_mghook_unify/ 0
      With> merge called from test: mghook_mghook_unify/ 0
      Node> 0
```

4.1.6 コマンド

リスナはプログラムのデバッグや対話的なプログラムの実行を行うために, いくつかのコマンドを用意している. 記述されている各コマンドの引数は, 特に説明がない場合には以下の値を与える. また, 引数に付加されている “^” や “?” については プログラミング編 第 7.1 節「モード表示の読み方」 p.222 を参照のこと.

[11] MGHOKo を指す白パス同士を active-unify した

Package	: パッケージ名アトム
ModuleName	: モジュール名アトム
ModuleSpec	: Package::ModuleName または ModuleName
	: Package を省略した場合はデフォルトパッケージを指定したものと見なす
Predicate	: 述語名アトム
Arity	: 述語の引数個数 (Arity>= 0)
Operator	: オペレータ名アトム
Precedence	: オペレータの優先順位 (0< Precedence =<1200)
OpType	: オペレータのタイプ. fx, fy, xfx, yfx, xfy, xf, yf のいずれか
Term	: 任意の KL1 データ
ONorOFF	: on (アトム) または off (アトム)
Depth	: 構造体の深さの出力制限値 (Depth ≥ 0)
Length	: 構造体などの長さの出力制限値 (Length ≥ 0)

4.1.6.1 基本コマンド

ModuleSpec : Public

ModuleSpec で示されるモジュールの公開述語である Public を実行する. 詳しくは 第 4.1.2.5 節「公開述語の呼び出し」 p.66 を参照のこと.

ModuleSpec ! Local

ModuleSpec で示されるモジュールの局所述語である Local を実行する. 詳しくは 第 4.1.2.5 節「局所述語の呼び出し」 p.67 を参照のこと.

Goal @ node(NodeNumber)

ノード番号 NodeNumber のノードの元で Goal を実行する. 詳しくは 第 4.1.2.5 節「実行ノードの指定」 p.68 を参照のこと.

Goal @ priority(Rate)

Goal @ relative_priority(Rate)

実行優先度 Rate で Goal を実行する. “priority” を指定した場合は所属荘園内割合指定, “relative_priority” を指定した場合は所属荘園内自己相対指定となる. 詳しくは 第 4.1.2.5 節「実行優先度の指定」 p.68 を参照のこと. また, 実行優先度指定の考え方については, プログラミング編 第 7.2.6 節 p.228 を参照のこと.

Goal1, Goal2

Goal1 と Goal2 を並列に実行する. 詳しくは 第 4.1.2.5 節「並列実行」 p.68 を参照のこと.

Goal1; Goal2

Goal1 と Goal2 を逐次に実行する. Goal1 の実行が終了しない限り Goal2 の実行は開始されない. 詳しくは 第 4.1.2.5 節「逐次実行」 p.68 を参照のこと.

!イベント番号

イベント番号で指定されたゴール列を表示する. 表示されたゴール列はユーザが任意に修正を施した後, リスナへの新たなゴール列の入力とすることができる. 指定されたイベントがヒストリキューに存在しない場合はエラーとなる. なお, 本コマンドはヒストリキューには格納されない.

【例】ゴール列を再表示する.

```
[10] !7.
[10] inter([prompt(" TERM>> "),getwt(^ WT)]) .
```

【参考】イベント番号に負の数を指定した場合は, 現在のイベント (番号) からの相対指定となる.

【注意】現在 !"文字列 " はサポートしていない.

!!

直前に入力されたゴール列を表示する. 表示されたゴール列はユーザが任意に修正を施した後, リスナへの新たなゴール列の入力とすることができる. ヒストリキューの長さが 0 の時はエラーとなる. なお, 本コマンドはヒストリキューには格納されない.

【例】直前に入力されたゴール列を再表示する.

```
[6] inter([prompt(" TERM>> "), getwt(WT)]).
 TERM>> abc(X).
 WT = normal(vector({atom(abc),variable("X")}))

[7] !!.
[7] inter([prompt(" TERM>> "),getwt(^WT)]) .     % 修正を施す.
```

【注意】現在 !"文字列 " はサポートしていない.

イベント番号

イベント番号で指定されたゴール列を再び実行する. 指定されたイベントがヒストリキューに存在しない場合はエラーとなる. なお, 本コマンドはヒストリキューには格納されない.

【例】ゴール列を再実行する.

```
[10] 7.
[10] inter([prompt(" TERM>> "),getwt(^ WT)]) .
 TERM>>
```

【参考】イベント番号に負の数を指定した場合は, 現在のイベント (番号) からの相対指定となる.

!

直前に入力されたゴール列を再び実行する. ヒストリキューの長さが 0 の時はエラーとなる. なお, 本コマンドはヒストリキューには格納されない.

【例】直前に入力されたゴール列を再実行する.

```
[6] inter([prompt(" TERM>> "), getwt(WT)]).
 TERM>> abc(X).
 WT = normal(vector({atom(abc),variable("X")}))

[7] !.
[7] inter([prompt(" TERM>> "),getwt(WT)]) .
 TERM>>
```

イベント番号リスト

イベント番号のリストで指定されたゴール列群を, 指定された順に逐次に実行する. 指定されたイベントがヒストリキューに存在しない場合はエラーとなる. イベント番号で指定されたゴール列の実行が失敗すると, 残っているイベントの実行は放棄される. なお, 本コマンドはヒストリキューには格納されない.

【例】既入力ゴール列群を再実行する.

```
[10] [3,5,6,7].

[10] setenv compile: link_error= warning .

[11] cd"~>TMP".

[12] compile"*.kl1".
```

【参考】イベント番号に負の数を指定した場合は, 現在のイベント (番号) からの相対指定となる.

history

history(N)　[his]

ヒストリキューに格納されているゴール入力の履歴を一覧表示する. 正整数 N が指定された場合は, 本コマンドの入力も含めて N 回分遡って入力の履歴を一覧表示する.

【例】最新の 5 回分の入力を表示する.

```
[37] history(5).
 33  test: check( [ 100 ], start , A , B , C < D )
 34  test: check([100],start,A,B,C,D)
 35  test: check([],start,A,B,C,D)
 36  .
 37  history(5)

[38] !34.
[38] test: check([100],start,A,B,C,D) .
```

【注意】空の入力 (変数プールをクリアするためのピリオドのみの入力) はピリオドが表示される.

history_length

history_length(Length)

history_length(^Old, Length)　[hl]

リスナのヒストリキューの長さを Length で示される値に設定する. Length に現在の長さよりも短い長さが指定された場合は, 指定された長さの数だけ最新の入力の履歴が保持される. 初期値は 50 である. 引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる. Old が与えられた場合には, 現在の値がユニファイされる.

prompt(FormatString)

prompt(^Old, FormatString)

コマンド入力行のプロンプトを書式 FormatString に従って変更する. 書式 FormatString はストリングで与え, \ で始まる書式項目以外の文字はそのまま出力される. 書式項目の一般形は \C で C には書式のタイプを指定する. 初期値は "[\!] " である. Old が与えられた場合には, 現在のプロンプトの書式ストリングがユニファイされる. 書式のタイプには次のものを指定できる.

\! : イベント番号を出力する
\d : カレントディレクトリを出力する
\p : デフォルトパッケージを出力する
\m : デフォルトモジュールを出力する
\M : マクロモジュールを出力する
\n : 改行を出力する
**** : 文字 "\" 自身を出力する

【例】プロンプトの変更例

```
[5] prompt(^Old,"?- ").
 Old = "[\!] "

?- prompt(Old).

[7] sample:test(abc,Ans).
 Ans = [a,b,c]
```

take(FilePathname)

FilePathname で指定されたコマンドファイルを実行する. FilePathname の先頭要素が “>” の場合には絶対指定であると見なす. 先頭要素が “~” の場合にはホームディレクトリからの相対指定 (“~” はホームディレクトリを意味する), それ以外の場合はカレントディレクトリからの相対指定であると見なす. また親ディレクトリの指定には “..” (ディレクトリを一段遡る) を用いることができる. なお, ファイル名拡張子が省略された場合は, まず拡張子無しでそのファイルを検索し, ファイルが存在しない場合には “.com” を自動的に付加して, ファイルを検索する.

- コマンドの入力形式はリスナトップレベルと同じである. コマンドファイル中にはリスナが実行できるコマンドは全て記述できる. 勿論, コマンドファイル中に **take** コマンドを記述することも可能である.
- コメントには % と /* ... */ を使うことができる.

【注意】コマンドファイルに指定してあるゴールの実行が失敗すると, その時点でコマンドファイルの実行は放棄される.

【参考】コマンドファイル中のコマンドの実行は逐次に行われるため, 次のようなことが実行できる.

【例】コマンドファイルの記述例

```
pushd(">sys>user>me>SAMPLE").
load("test.exe").
cd("..>PATCH").
compile( ["test1", "test2", "test3"] ).
relink( test, [ test1, test2, test3 ] ).
popd.
profile(test:go(1,Ans1)).
profile(test:go(2,Ans2)).
```

alias(Alias, Goals)

Goals の別名として Alias を登録する. 以降リスナトップレベルから入力されるゴールが Alias と一致する場合は, Goals に置き換えられて実行される. Alias として登録するゴールの引数には変数が含まれていても構わない. また, Alias の展開結果 (ゴール) が既に別名で登録されている場合は, 再帰的に展開される. 展開の適用回数はデフォルトで 5 回までである. なお, 適用回数の変更は **alias_expansion** コマンド (p.91) で行う.

【例 1】別名の登録

```
[1] alias( q(N,Ans), layered:n_queens(N,Ans) ).
 alias: q(N,Ans) => layered: n_queens(N,Ans)

[2] q(4,X).
 X = [1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]
```

【例 2】複数ゴールに展開する

```
[1] alias( q(N), (q(N,Ans); display(Ans)) ).
 alias: q(N) => q(N,Ans); display(Ans)

[2] q(4)
 [1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]
```

【注意 1】既に登録されている別名を再度登録した場合, 古い別名は抹消される.

【注意 2】入力ゴールが別名と一致するかどうかは, ファンクタ (述語名) とアリティ (引数個数) で判断している. 将来, 改良する予定である.

【参考】リスナコマンドの省略形はこの alias 機能を利用して実現されている.

unalias(Alias)

別名として登録されている Alias を抹消する.

【例】別名の抹消

```
[1] unalias(q(_)).
 unalias: q(N) => q(N,Ans); display(Ans)
```

alias(Alias)

別名である Alias の展開形式を表示する.

【例】別名の展開形式の表示

```
[1] alias(q(_,_)).
 alias: q(N,Ans) => layered: n_queens(N,Ans)
```

alias

リスナに登録されている別名の一覧を表示する.

alias_expansion

alias_expansion(N)

別名を再帰的に展開する場合の展開回数の上限値を N に設定する. N を 0 に設定すると展開は全く行われない. 初期値は 5 回である. 引数が省略された場合は現在の値を表示した後にユーザからの入力待ちとなる.

errors

リスナが持っているエラーキューに格納されているエラーを一覧表示する. エラーキューの長さの初期値は 5 である. 次のような内容の一覧が表示される.

"エラー番号" "[イベント番号]" "例外の種類" "例外情報"

【例】発生したエラーの一覧表示

```
[17] errors.
  1  [4]   unification_failure {{0,0},b, ...}
  2  [7]   illegal_input {{0,0},add( & ), ...}
  3  [10]  reduction_failure {{0,0}, code$nakao::...}
  4  [14]  merger_deadlock {{0,0},merge( & ), ...} ? 3  % エラー番号を入力
 Failure>> layered: queens/4
    Goal > queens(1,f,begin,C)
    Node > 0
 !! Exception[3.10]: ins
```

"?" のプロンプトの後に "エラー番号" を入力すると, そのエラーの詳しい内容が表示される. リターンのみの入力であれば最新のエラーの内容が表示される.

【参考】エラーが発生した時に, エラーの内容を良く見ていないのに (ホロフラステイングのレベルが小さいときなど) 誤ってエラー表示ルーチンから抜けてしまった場合などに利用すると便利である.

last_error

発生したエラーのうち最新のものを再度表示する.

error_queue_length

error_queue_length(Length)

発生したエラーを格納するためのエラーキューの長さを変更する. 初期値は 5 である. 引数が省略された場合は現在の値を表示した後にユーザからの入力待ちとなる.

display(?Term)

display(?Term, Depth, Length) [disp]

Term をリスナウインドウに表示する. Depth と Length が与えられた場合は構造体の深さの出力制限値 Depth, 長さの出力制限値 Length で表示する.

date

現在の日付と時刻を表示する.

suspend

シェルウインドウ上で動作しているリスナを, シェルのバックグラウンドに移行させる. コマンドが実行されるとシェルのトップレベル (プロンプト) が現れる. なお, リスナがシェルとは独立のウインドウで動作している場合は何も行わない.

【例】シェルウインドウ上のリスナをバックグラウンドに移行する

```
MyShell>> listener:listener
*** KL1 LISTENER (V 3.0 17-Oct-91) ***
MACRO MODULES     : [pimos]
DEFAULT PACKAGE   : pimos
CURRENT DIRECTORY : >sys>user>icot

[1] suspend.
MyShell>>                              % シェルのプロンプトが現れる
MyShell>> status
  2 --> stopped listener: listener
MyShell>> fore                         % リスナをフォアグラウンドに戻す

[2]                                    % リスナトップレベルが現れる
```

help(Type)　[he]

Type (アトム) で指定されたタイプのコマンドの一覧を表示する. Type には次のものを指定できる.

builtin	:	組込述語
basic	:	基本コマンド
window	:	リスナウインドウ操作コマンド
varpool	:	変数プール操作コマンド
env	:	環境コマンド
dir	:	ディレクトリコマンド
library	:	ライブラリコマンド
debug	:	デバッグコマンド
pg	:	ParaGraph 関連コマンド
util	:	ユーティリティコマンド
all	:	全コマンド

help　[he]

help/1 で指定できるコマンドタイプの一覧を表示する.

commands(Type)　[com]

Type で指定されたタイプのコマンド名の一覧を表示する. Type は **help/1** で与えるものと同じである.

commands　[com]

commands/1 で指定できるコマンドタイプの一覧を表示する.

exit [ex]

ネストした実行レベルを一段外に出る．レベルがゼロのとき (即ち，レベル表示が行われていないとき) はリスナは終了する．

4.1.6.2 リスナウインドウ操作コマンド

【注意】2.5 版にあった以下のコマンドは 3.0 版では廃止された．

title(FormatString), title(^Old, FormatString)
position, position(Position)
size, size(Size)
font, font(Pathname)
hide, close, open

タイトル，位置，サイズ，フォントの項目の設定は，リスナの立ち上げ時には可能である (第 4.1.2.1 節, p. 63 参照)．

operator [op]

リスナの全オペレータの定義を表示する．

【注意】ここでいうオペレータは，リスナがユーザプログラムに提供する標準入出力に付いているバッファのオペレータのことではない．リスナトップレベルから入力された項 (ゴール) をパーズ / アンパーズするためのリスナが参照するオペレータである．

operator(Operator) [op]

リスナのオペレータ Operator の定義を表示する．

operator(Operator, ^Definition) [op]

リスナのオペレータ Operator の定義を Definition にユニファイする．形式は { Precedence, OpType } のリストである．

add_op(Precedence, OpType, Operator)

リスナのオペレータを追加する．

remove_op(Precedence, OpType, Operator)

リスナのオペレータを削除する．

4.1.6.3 環境コマンド

statistics [st]

タスクの実行に関する情報を報告するモードを ON にする．報告される情報は，タスクの資源消費量 (リダクション数) と実行に要した時間 (ミリ秒単位) である．

【例】statistics ON のとき

```
[7] layered:n_queens(4,Ans).
 Ans = [1*[3*[],4*[ & ]],2*[4*[ & ]],3*[1*[ & ]], ... ]
 152 reductions
 432 msec
```

【参考 1】計測される時間はタスクの実行が開始されてから終了する (terminated が報告される) までである.

【参考 2】時間の計測にはガードの組込述語 system_timer/2 を利用している.

【注意】表示されるリダクション数にはリスナのオーバーヘッドが若干含まれる.

nostatistics [nst]

タスクの実行に関する情報を報告するモードを OFF にする.

【例】statistics OFF のとき

```
[1] layered:n_queens(4,Ans).
 Ans = [1*[3*[],4*[ & ]],2*[4*[ & ]],3*[1*[ & ]], ... ]
```

statistics(ONorOFF)

statistics(^Old, ONorOFF) [st]

タスクの実行に関する情報を報告するモードを ONorOFF に変更する. Old が与えられた場合には, 現在のモードがユニファイされる.

change_macro

change_macro(MacroModuleName)

リスナのマクロモジュールを MacroModuleName に変更する. 複数のマクロモジュールを設定する場合はマクロモジュール名 (アトム) のリストを指定する. 例えば, 複数のマクロモジュール `[a,b,c]` は, `:- with_macro a,b,c.` に相当する. 複数マクロの適用規則については プログラミング編 第 7.3.4.4 節 p.255 を参照のこと. リスナ起動時のマクロモジュールは **pimos** である. 引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる. 指定されたマクロモジュールが存在しない場合や, マクロモジュールでないモジュールが指定された場合はエラーとなる. また, マクロの展開を一切行いたくない場合には MacroModuleName に整数 0 を指定する. リスナのマクロ展開についての注意事項は 第 4.1.2.4 節 p.65 を参照のこと.

【例 1】リスナのマクロモジュールを変更する.

```
[1] change_macro.
 Current: pimos
 New    : my_macro
 pimos => my_macro
```

【例 2】リスナのマクロ展開を抑制する.

```
[2] change_macro
 Current: my_macro
 New    : 0
 my_macro => Non-expansible mode
```

【例 3】指定したマクロモジュールが存在しない.

```
[3] change_macro
 Current: pimos
 New    : another_macro
 Nonexistent module>> another_macro
```

【例 4】不正なモジュールが指定された.

```
[4] change_macro
 Current: pimos
 New    : my_module
 Illegal macro module>> my_module
```

change_nodes

リスナから起動されるタスク (即ち, ユーザプログラム) が利用できるノードを変更する. 利用するノードの指定は「Start 番のノードから Nodes 台」という形式で行う. コマンドを実行すると次のようなメッセージが出力される.

【例】利用ノードとして 8 番のノードから 8 台使用する

```
[1] change_nodes.
 Start node     : 0
 Number of nodes: 16
 >Input format is as follows.
   StartNode,NumberOfNodes
  ( 0 =< StartNode =< 15,
    1 =< NumberOfNodes,
    StartNode+NumberOfNodes =< 16 )
 => 8,8                              % 8番のノードから8台使用する.
```

default_package

default_package(Package)

default_package(^Old, Package)　[dfp]

デフォルトパッケージを Package に変更する. リスナ起動時のデフォルトパッケージは **task: package** の値となる. 引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる. Old が与えられた場合には, 現在の値がユニファイされる. リスナのデフォルトパッケージの取り扱いについては 第 4.1.2.3 節 p.65 を参照のこと.

default_module

default_module(ModuleSpec)

default_module(^Old, ModuleSpec)　[dfm]

デフォルトモジュールを ModuleSpec で指定されたモジュールに設定する. デフォルトモジュールが宣言された述語はリスナトップレベルにおいて, モジュール名の入力を省略して実行できる. 初期値は設定されていない (デフォルトモジュールなし). 引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる. Old が与えられた場合には, 現在の値がユニファイされる. なお, モジュール名を省略した場合の述語の呼び出し形式については, 第 4.1.2.5 節「公開述語の呼び出し」 p.66 および「局所述語の呼び出し」 p.67 を参照のこと.

print_depth

print_depth(Depth)

print_depth(^Old, Depth)　[pd]

構造体の深さの出力制限値を Depth で示される値に設定する．初期値は 4 である．引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる．Old が与えられた場合には, 現在の値がユニファイされる．なお, 出力する構造体が深さの出力制限値を越えた場合は“&”で表示される．

print_length

print_length(Length)

print_length(^Old, Length)　[pl]

構造体の長さ (要素の数) の出力制限値を Length で示される値に設定する．初期値は 10 である．引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる．Old が与えられた場合には, 現在の値がユニファイされる．なお, 出力する構造体が長さの出力制限値を越えた場合は“...”で表示される．

print_string_length

print_string_length(Length)

print_string_length(^Old, Length)　[psl]

ストリングの要素数の出力制限値を Length で示される値に設定する．初期値は 70 である．引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる．Old が与えられた場合には, 現在の値がユニファイされる．なお, 出力するストリングの要素数が出力制限値を越えた場合は“??”で表示される．

print_atom_length

print_atom_length(Length)

print_atom_length(^Old, Length)　[pal]

アトム印字名の文字数の出力制限値を Length で示される値に設定する．初期値は 40 である．引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる．Old が与えられた場合には, 現在の値がユニファイされる．なお, 出力するアトム印字名の文字数が出力制限値を越えた場合は“??”で表示される．

print_variable_mode

print_variable_mode(Mode)

print_variable_mode(^Old, Mode)　[pv]

変数を表す項の出力形式を Mode で示される形式に変更する．Mode にはアトムで name または address のいずれかを与える．初期値は **name** である．引数が省略された場合は, トグルで (現在が name であれば address に, address であれば name に) 変更される．Old が与えられた場合には, 現在の値がユニファイされる．

name	:	名前モード	::	A,B,C, ...
address	:	アドレスモード	::	Address1$Address2

【参考】マルチ PSI や PIM/m では Address1 は, その変数が存在しているプロセサの番号, Address2 はそのプロセサ内でのアドレスを表す.

rewrite_depth

rewrite_depth(Depth)

rewrite_depth(^Old, Depth)　[rwd]

トレース時やエラー処理時に, 一時的に出力制限値を変えて再表示を行う際の構造体の深さの出力制限値を設定する. 初期値は 30 である. 引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる. Old が与えられた場合には, 現在の値がユニファイされる.

rewrite_length

rewrite_length(Length)

rewrite_length(^Old, Length)　[rwl]

トレース時やエラー処理時に, 一時的に出力制限値を変えて再表示を行う際の構造体の長さの出力制限値を設定する. 初期値は 30 である. 引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる. Old が与えられた場合には, 現在の値がユニファイされる.

environment　[env]

リスナの実行環境パラメタの値を問い合わせる. 各パラメタの初期値は次の通り.

【例】リスナ起動時の実行環境パラメタの表示.

```
[1] environment.
 Current directory   : "icpsi255::>sys>user>me"
 Default package     : me
 Default module      : 0
 Macro module        : [pimos]
 Alias expansion     : 5
 Statistics          : off
 Remembering mode    : off
 Display variables   : used
 Query mode          : no
 Display ancestor    : abbr
 Scope of variables  : tracer
 Print depth         : 4
 Print length        : 10
 Print string length : 70
 Print atom   length : 40
 Print variable mode : name
 Re-write depth      : 30
 Re-write length     : 30
 Error queue length  : 5
 History length      : 50
 Event number        : 1
```

各フィールドの意味は次の通りである.

Current directory : カレントディレクトリ.

Default package : デフォルトパッケージ.

Default module : デフォルトモジュール.
デフォルトモジュールが 0 と表示されているのは, 初期値が設定されていないことを表す.

Macro module : マクロ展開モジュール.

Alias expansion : 別名の展開回数.

Statistics : 実行情報 (リダクション数と実行時間) を報告するかどうかのモード.

Remembering mode : 実行時に使用した変数の値を記憶するかどうかのモード.

Display variables : 実行終了時に変数の値を出力するモード

Query mode : 実行終了時に変数の問い合わせをするモード

Display ancestor : スパイ実行時の祖先ゴールの出力形式

Scope of variables : トレース時の変数の同一性の範囲

Print depth : 構造体の深さの出力制限値.

Print length : 構造体の長さの出力制限値.

Print string length : ストリングの要素数の出力制限値.

Print atom length : アトム印字名の文字数の出力制限値.

Print variable mode : 変数を表す項の出力形式.

Rewrite depth : 一時的に出力制限値を変えて再表示する時の構造体の深さの出力制限値.

Rewrite length : 一時的に出力制限値を変えて再表示する時の構造体の長さの出力制限値.

Error queue length : エラーキューの長さ.

History length : ヒストリキューの長さ.

Event number : リスナが起動されてから実行したコマンド数. 本コマンドの実行も含まれる.

4.1.6.4 環境変数の操作

environment(Env) [env]

リスナ自身のタスクが持つ環境変数プールからその名前が“Env:×××”であるものを取り出し表示する.

```
[7] env(task).
 task: io_node = 0
 task: directory = "icpsi255::>sys>user>nakao>LISTENER"
 task: user = nakao
 task: home = "icpsi255::>sys>user>nakao"
 task: package = nakao
```

printenv

リスナ自身のタスクが持つ環境変数プールから全ての値を取り出し表示する.

printenv(Name)

リスナ自身のタスクが持つ環境変数プールから Name をキーとする値を取り出し表示する.

getenv(^Status)

リスナ自身のタスクが持つ環境変数プールから全ての値を取り出し, 処理結果を Status にユニファイする. Status は以下のいずれかである.

normal(NameValueList) : 正常終了. NormalValueList には, もし環境変数がない場合には空リスト, あれば各々に対応して {**Name, Value**} の形式のベクタを要素として持つリストを返す. ここで Name は環境変数名, Value はその値である.

abnormal : 異常終了.

getenv(Name, ^Status)

リスナ自身のタスクが持つ環境変数プールから Name をキーとする値を取り出し, 処理結果を Status にユニファイする. Name は構造体なら要素に至るまで完全に具体化された値なければならない. Status は以下のいずれかである.

normal(Value) : 正常終了. 引数に値を返す.

abnormal : 指定された環境変数がない.

setenv Name = Value

setenv(Name, Value, ^Status)

リスナ自身のタスクが持つ環境変数プールに Name をキーとして Value で指定する値を設定する. Status には処理結果がユニファイされる. Name, Value は構造体なら要素に至るまで完全に具体化された値でなければならない. Status は以下のいずれかである.

normal : 正常終了.

abnormal : 異常終了.

【注意】現在, 名前が listener: ××× の環境変数は値を設定することができない.

unsetenv Name

unsetenv(Name, ^Status)

リスナ自身のタスクが持つ環境変数プールから Name をキーとする環境変数を削除する. Status には処理結果がユニファイされる. Name は構造体なら要素に至るまで完全に具体化された値でなければならない. Status は以下のいずれかである.

normal : 正常終了. 環境変数がない場合もこの結果が返る.

abnormal : 異常終了.

【注意】現在, 名前が listener: ××× の環境変数は削除することができない.

4.1.6.5 変数プール操作コマンド

remember [rem]

リスナトップレベルに新規に出現した変数を変数プールに格納するモードを ON にする. 変数プールに格納されている変数はユーザがそれを解放しない限り値が束縛されており, リスナトップレベルにおいてゴールの引数として何度でも利用できる. なお, このモードが ON であっても, 変数名が下線で始まる変数は格納されない. リスナ起動時の既定値は OFF である.

【例】変数を変数プールに格納するモード

```
[1] remember.                          % 変数を格納するモードを ON にする.

[2] layered:n_queens(4,Ans).           % 変数 Ans は格納される.
^Ans = [1*'[' & ']',2*'[' & ']',3*'[' & ']',4*'[' & ']']

[3] inspect Ans.                       % 変数 Ans は値が束縛されたままである.
[1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]> me
```

【参考 1】変数プールに格納されている変数は, その名前の前に“^”を付けて表示される.

【参考 2】変数名の直前に“^”を付けることでも, 変数プールに格納させることができる. 詳しくは第 4.1.3.2 節 p.76 を参照のこと.

【参考 3】変数プールに格納されている変数を解放するには, 後述の **ピリオドリターン** を入力するか, **forget** コマンドを利用する.

forget [fgt]

リスナトップレベルに新規に出現した変数を変数プールに格納するモードを OFF にする. このコマンド以降, リスナトップレベルに新規に出現する変数はタスクが終了すると値が忘れられる. なお, モードを OFF にしても既に変数プールに格納されているものは格納されたままである.

【例】変数を変数プールに格納しないモード

```
[1] remember.                  % 変数を格納するモードを ON にする.

[2] X=1.                       % 変数 X は格納される.
^X = 1

[3] forget.                    % 変数を格納するモードを OFF にする.

[4] Y=2.
 Y = 2

[5] value.                     % 格納されている変数の一覧を表示する.
^X = 1                         % 変数 X は値が束縛されたままである.
                               % 変数 Y は格納されていない
[6] .                          % ピリオドリターンで解放

[7] value.                     % 格納されている変数の一覧を表示する.
                               % 格納されている変数はない
[8]
```

remember(ONorOFF) [rem]

リスナトップレベルに新規に出現した変数を変数プールに格納するモードを ONorOFF に変更する.

remember(VariableName) [rem]

名前が VariableName である変数をリスナの変数プールに格納する. なお, 名前が下線で始まる変数を指定した場合, 指定は無効となる.

【例】特定の変数だけ変数プールに格納する

```
[1] layered:n_queens(4,Ans), remember(Ans).  % 変数 Ans は格納される.
^Ans = [1*[' & ']',2*[' & ']',3*[' & ']',4*[' & ']']

[2] inspect Ans.              % 変数 Ans は値が束縛されたままである.
[1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]> me
```

【参考 1】変数プールに格納されている変数は, その名前の前に“^”を付けて表示される.

【参考 2】変数名の直前に“^”を付けることでも, 変数プールに格納させることができる. 詳しくは第 4.1.3.2 節 p.76 を参照のこと.

forget(VariableName)　[fgt]

名前が VariableName である変数をリスナの変数プールから削除する.

【例】変数プールに格納されている変数を個別に解放する.

```
[11] value.          % 変数 X と Y が格納されている.
^X = 1
^Y = 2

[12] forget(X).      % 変数 X を変数プールから削除する.

[13] value.
^Y = 2               % 変数 X は削除された.

[14] X=Y.
 X = 2
^Y = 2
```

【参考】**forget** コマンドは **list** コマンドや **query** コマンドの中からも実行できる.

ピリオドリターン

変数プールに格納されている全ての変数を解放する. ピリオドリターンとは“.”, “cr”のみからなる空のゴールを入力することである.

【例】変数プールに格納されている変数を全て解放する.

```
[11] value.          % 変数 X と Y が格納されている.
^X = 1
^Y = 2

[12] .               % 格納されている変数を解放する.

[13] value.          % 格納されている変数はない.

[14] X=a, Y=b.
 X = a
 Y = b
```

display_variables

display_variables(Mode)

display_variables(^Old, Mode)　[dv]

タスクの実行終了時に, ゴールの引数に使用した変数の値を表示するモードを設定する. Mode にはアトムで all, used, new, no のいずれかを与える. 初期値は **used** である. 引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる. Old が与えられた場合には, 現在の値がユニファイされる. なお, 各モードの意味は次の通り.

all	:	変数プールに格納されている全変数 及び そのゴールで使った変数
used	:	そのゴールで使った変数
new	:	そのゴールで新たに使った変数
no	:	変数の値を表示しない

【注意】名前が "_" (下線) で始まる変数はいかなるモードでも表示されない.

query

query(Mode)

query(^Old, Mode)

タスクの実行終了時に, ゴールの引数に使用した変数に対する問い合わせのモードを設定する. Mode にはアトムで all, used, new, no のいずれかを与える. 初期値は **no** である. 引数が省略された場合は, 現在の値を表示した後にユーザからの入力待ちとなる. Old が与えられた場合には, 現在の値がユニファイされる. なお, 各モードの意味は次の通り.

all	:	変数プールに格納されている全変数 及び そのゴールで使った変数
used	:	そのゴールで使った変数
new	:	そのゴールで新たに使った変数
no	:	問い合わせをしない

【注意】名前が "_" (下線) で始まる変数は以下なるモードでも問い合わせの対象とならない.

【例】実行に与えた変数に対する問い合わせ

```
[1] display_variables(no), query(used).

[2] current_node(_,Total), layered:n_queens(Total,Ans).
( Ans, Total )? rem Ans                 % Ans を変数プールに格納する
(^Ans, Total )? ins Ans                 % Ans をインスペクトする
[1*[ & , & ],2*[ & ],3*[ & ],4*[ & , & ]]> me
```

問い合わせとして以下の操作が行える. <, > で囲まれた引数は省略可能である. なお, 変数名の指定が可能な操作において変数名の指定を省略した場合は, 全ての変数が操作の対象となる.

v		:	全ての変数の値を表示する
	< 変数名,...>	:	指定された変数の値を表示する
w	**< 変数名,...>**	:	出力制限値を一時的に大きくして変数の値を表示する
ins	**< 変数名,...>**	:	指定された変数をインスペクトする
rem	**< 変数名,...>**	:	指定された変数を変数プールに格納する
fgt	**< 変数名,...>**	:	指定された変数を変数プールから削除する
pd	**< 深さ >**	:	構造体の深さに関する出力制限値を変更する
pl	**< 長さ >**	:	構造体の長さに関する出力制限値を変更する
pal	**< 長さ >**	:	アトム印字名の長さに関する出力制限値を変更する
psl	**< 長さ >**	:	ストリングの長さに関する出力制限値を変更する
pv		:	変数の出力モード (名前 / アドレス) を変更する
q,	**リターン**	:	終了

value [v]

変数プールに格納されている変数をその値と共に一覧表示する.

【例】格納されている変数を一覧表示する (現在 forget モードとする).

```
[1] ^X=1, Y=2, ^Z := X+Y.        % ^ の付いている変数のみ格納される.
^X = 1
 Y = 2
^Z = 3

[2] value.
^X = 1
^Z = 3
```

list [l]

変数プールに格納されている変数の一覧を表示する.

【例】格納されている変数を一覧表示する (現在 forget モードとする).

```
[1] ^X=1, Y=2, ^Z := X+Y.        % ^ の付いている変数のみ格納される.
^X = 1
 Y = 2
^Z = 3

[2] list.
(^X,^Z )? fgt Z
(^X, Z )?
```

"?" のプロンプトの後に入力できるコマンドは **query** コマンド (p.102) と同様である.

4.1.6.6 ディレクトリコマンド

cd

ホームディレクトリに戻る.

cd(Directory)

cd(^Old, Directory)

カレントディレクトリを Directory に変更する. ディレクトリの指定において, 先頭が“>”の場合には絶対指定とみなす. 先頭が“〜”の場合にはホームディレクトリからの相対指定とみなす (“〜”はホームディレクトリを意味する). それ以外の場合にはカレントディレクトリからの相対指定であるとみなす. 親ディレクトリの指定には“..”(ディレクトリを一段遡る) を用いることができる. なお Old が与えられた場合には, 現在の値がユニファイされる.

【例 1】先頭が大文字で始まるディレクトリ名はクオートで囲む (但し, ディレクトリ名を文字列で指定した場合にはクオートは不要である)

```
[1] cd  >sys>user>me>'LISTENER'>'NEW'.

[1] cd ">sys>user>me>LISTENER>NEW".
```

【例 2】特殊アトム (“〜”, “..”) を使用する時はスペースを入れる (但し, ディレクトリ名を文字列で指定した場合にはスペースは不要である).

```
[1] cd  〜 > .. >me>'LISTENER'>'NEW'.    % スペースを入れる

[1] cd  "〜>..>me>LISTENER>NEW".         % スペースは不要

[1] cd  〜 .                             % ピリオドの前にスペースを入れる

[1] cd  .. > .. > me.                    % スペースを入れる

[1] cd  "..>..>me".                      % スペースは不要
```

pushd(Directory)

pushd(N)

pushd

カレントディレクトリを変更し, カレントディレクトリ名をディレクトリスタックの先頭に格納する. Directory の指定は cd コマンドの場合と同様である. ディレクトリ名の代わりに数値 (正整数) を指定した場合には, スタックの N 番目の要素が先頭になるようにローテーションを行う. 但し, スタックの先頭は 0 番目から始まる. また, 引数を省略した場合には, スタックの 0 番目の要素と 1 番目の要素を入れ替える.

popd

ディレクトリスタックの 0 番目の要素を取りだし, カレントディレクトリを 1 番目の要素に指定されたディレクトリに変更する. ディレクトリスタックが空の場合はエラーとなる.

dirs

ディレクトリスタックの内容を一覧表示する. 最初 (第 0 要素) に表示されるディレクトリがカレントディレクトリである.

pwd

カレントディレクトリを表示する.

pwd(^Directory)

カレントディレクトリを Directory にユニファイする.

def(Atom, PathnameString)

コマンド **con/1** で使用する “論理ディレクトリ名” Atom と “ディレクトリ名” PathnameString との対応を定義する. 既に定義済みの論理ディレクトリ名を再度定義した場合は, ディレクトリ名が置き換わる.

```
[1] def(samp,">sys>user>me>SAMPLE").
 Define: samp => ">sys>user>me>SAMPLE"
```

【参考】リスナを起動したシェル上で定義された論理ディレクトリ名はリスナ起動時に継承される.

【注意】PIMOS 3.0 版では SIMPOS の論理ディレクトリ名をサポートしていない. def, con コマンドは, カレントディレクトリを変更するときに論理ディレクトリ名を使用できないという不便さを多少なりとも補うために提供されている. 将来, ファイルシステムやワールドの管理機構が導入された場合, これらのコマンドは廃止する予定である.

def

現在設定されている論理ディレクトリ名の一覧を表示する.

【注意】このコマンドは, 将来, 廃止する予定である.

con(Atom)

def/1 で定義された “論理ディレクトリ名” Atom に対応する “ディレクトリ名” を取り出し, そのディレクトリにカレントディレクトリを変更する.

```
[1] con(samp).
>sys>user>me>SAMPLE
```

【注意】このコマンドは, 将来, 廃止する予定である.

listing(Wildcard, ^Files)　[ls]

ワイルドカード Wildcard (文字列) に適合する名前のファイルを Files にユニファイする.

delete(Wildcard)

カレントディレクトリにおいて, ワイルドカード Wildcard (文字列) に適合する名前のファイルを削除する.

deleted

カレントディレクトリにおいて, 削除されたファイルのファイル名を表示する.

deleted(Wildcard)

カレントディレクトリにおいて, 削除されたファイルで, ワイルドカード Wildcard (文字列) に適合するもののファイル名を表示する.

undelete(Wildcard)

カレントディレクトリにおいて, 削除されているファイルで, ワイルドカード Wildcard (文字列) に適合するものを回復する.

purge

カレントディレクトリにおいて, 最大バージョン番号を持つファイルだけを残し, それ以外のものを一括抹消する.

expunge

カレントディレクトリにおいて, 削除状態のファイルを一括抹消する.

attributes(Pathname)

Pathname で指定されたファイルの属性を表示する. Pathname の指定は cd コマンドの場合と同様である. 表示される属性は所有者 (Owner), 更新者 (Update user), 生成日時 (Created), 更新日時 (Modified), リンク数 (Rink counter), ファイルサイズ (File size), レコードサイズ (Record size) である.

【例】ファイルの属性を表示する.

```
[1] attributes(">sys>csp>PIMOS.DIR>pimos.conf").
 icpsi255::>sys>csp>PIMOS.DIR>pimos.conf.7
 Owner        : icpsi255!s-nakao
 Update user  : icpsi255!s-nakao
 Created      : 27-Oct-90 10:38:42
 Modified     : 27-Oct-90 10:38:42
 Link counter : 1
 File size    : 438 byte(s)
 Record size  : 0 byte(s)
```

4.1.6.7 ライブラリコマンド

predicates(ModuleSpec, Predicate, Arity)

ModuleSpec で指定されたモジュール中の述語で, 述語名が Predicate, 引数個数が Arity であるものを一覧表示する. パッケージ名, モジュール名, 述語名, 引数個数の指定にはワイルドカード ("*") を使用できる. なお, パッケージの継承は行わない.

info(ModuleSpec)

ModuleSpec で示されたモジュールのモジュール情報を表示する. 表示される内容は, モジュールの生成日時, リンク日時, ソースファイル名とその生成日時, コンパイルするに当たって依存したファイル名とその生成日時 (include したファイルなど) である.

【例】モジュール情報の表示

```
[3] info(listener_cmd_builtin).
 %%% nakao:: listener_cmd_builtin %%%
 Created: 13-Oct-1991 16:37:22
 Linked : 13-Oct-1991 16:37:22
 Source : icpsi255::>sys>user>nakao>LISTENER>NEW>com_built.kl1.1
        : 03-Oct-1991 17:08:49
 [Dependent Files]
 icpsi255::>sys>user>nakao>LISTENER>NEW>pimos.h.1 24-Sep-1991 17:28:05
```

【注意】IPL でロードされたモジュールはモジュール情報を見ることができない.

xref(ModuleSpecs)

xref(ModuleSpecs, Packages)

ModuleSpecs で示されたモジュールを (外部) 参照している Packages 内のモジュールの一覧を表示する. ModuleSpecs にはモジュール名もしくはモジュール名のリスト, Packages にはパッケージ名もしくはパッケージ名のリストを与えることができる. モジュール名にパッケージ名の指定がない場合にはデフォルトパッケージが指定されたものとみなす. また, Packages が省略された場合は, デフォルトパッケージ内のモジュールが対象となる.

【例】モジュールの参照関係の表示

```
[3] xref(listener).
[nakao:: listener]
  nakao::
    listen
    listener
    listener_trace_command
    listener_exception_handler
    listener_interrupter
```

デフォルトパッケージ (この例では nakao) 内のモジュール listener を参照しているデフォルトパッケージ内のモジュール名の一覧が表示された.

4.1.6.8 デバッグコマンド

以下でトレース方式に関する諸概念については 第 4.1.7 節 p.118 を参照のこと.

trace [tr]

トレースの実行モードにはいる. トレースモードでは次のプロンプトが出力される.

【例】トレースモードのプロンプト

```
{Trace}
[3]
```

trace(Goals) [tr]

Goals をトレースモードで実行する.

【例】トレースの実行

```
[1] trace( layered:n_queens(4,Ans) ).
@0@4096   1#1   layered: n_queens(4,Ans)
   2     * (1)  queens(1,4,begin,Ans)?
```

spy_reduction [sr]

リダクションスパイの実行モードにはいる．リダクションスパイモードでは次のプロンプトが出力される．

【例】リダクションスパイモードのプロンプト

```
{SpyReduction}
[3]
```

spy_reduction(Goals) [sr]

Goals をリダクションスパイモードで実行する．

【例】リダクションスパイの実行

```
{Trace}
[3] sr( layered:n_queens(4,Ans) ).
<1> layered: n_queens/2 -->
@0@4096    #1 + filter(begin,1,1,D)
                 D= begin?
```

spy_fork [sf]

フォークスパイの実行モードにはいる．フォークスパイモードでは次のプロンプトが出力される．

【例】フォークスパイモードのプロンプト

```
{SpyFork}
[3]
```

spy_fork(Goals) [sf]

Goals をフォークスパイモードで実行する．

【例】フォークスパイの実行

```
[3] sf( layered:n_queens(4,Ans) ).
<1> layered: n_queens/2 -->
@0@4096    #1 queen/4
   2   + * (1)  filter(begin,1,1,D)?
```

notrace [ntr]

トレース 及び スパイの実行モードを抜けて通常の実行モードに戻る．

notrace(Goals) [ntr]

Goals をトレース及びスパイモードを抑制して通常のモードで実行する．

【例】トレースモードの抑制

```
{Trace}
[3] notrace( layered:n_queens(4,Ans) ).
 Ans = [1*'[' & ']',2*'[' & ']',3*'[' & ']',4*'[' & ']']
```

【参考】このコマンドを alias に利用すると, ユーザ定義述語をリスナのコマンドとして登録することができる (トレースモードでもトレースが掛からないようにできる).

```
[1] alias( q4, (notrace(layered:n_queens(4,Ans)); display(Ans)) ).
 alias: q4 => notrace(layered: n_queens(4,Ans)); display(Ans)

[2] trace.

{Trace}
[3] q4.
[1*'[' & ']',2*'[' & ']',3*'[' & ']',4*'[' & ']']
```

spy(ModuleSpec, Predicate, Arity)

spy(Package, ModuleName, Predicate, Arity)

Package, ModuleName (ModuleSpec) で指定されたモジュール中の Predicate/Arity なる述語にスパイを設定する. 公開述語は勿論のこと, 局所述語にも設定は可能である. なお Package, ModuleName, Predicate, Arity にはワイルドカード “*” を使用することができる.

【例 1】ワイルドカードを利用したスパイの設定

```
[1] spy(listener, *, *).
 Spypoint set on listener:*/*

モジュール名 listener の全ての述語に対してスパイを設定する.
```

【注意】現在 ModuleName:Predicate/Arity 記法によるスパイの設定においてワイルドカードを使用する場合にはワイルドカードを “(”, “)” で囲まなければならない. なお “ /* ... */ ” はコメントとなっているため, 誤って “/*” を入力した場合には “*/” でコメントを閉じれば良い.

【例 2】ワイルドカードを利用したスパイの設定

```
[1] spy listener:(*)/(*).
```

ワイルドカードを利用してスパイの設定を行った場合, 既に設定されているスパイ対象述語との包含関係の解析が行われる.

【例 3】ワイルドカードとの包含関係の解析

```
[1] spy(listener, *, *).
 Spypoint set on listener:*/*

[2] spy(listener, listener, *).
 There already is a spypoint on listener:listener/*
```

spy(Predicate, Arity)

デフォルトモジュールを省略した場合のスパイの設定. Predicate, Arity にはワイルドカードを使用することができる.

nospy(ModuleSpec, Predicate, Arity)

nospy(Package, ModuleName, Predicate, Arity)

Package, ModuleName (ModuleSpec) で指定されたモジュール中の Predicate/Arity なる述語のスパイを解除する. Package, ModuleName, Predicate, Arity にはワイルドカード “*” を使用することができる.

【注意】ワイルドカードによりスパイが設定されている述語を個別に解除することはできない. 解除にはワイルドカードを用いて, 設定した時と同じ形式, 若しくは, それを包含する形式で行わなければならない.

【例】スパイの解除ができない場合

```
[1] spy(listener, listener, *).
 Spypoint set on listener:listener/*.

[2] nospy(listener, listener, 10).
 Spypoint can't reset on listener:listener/10

[3] nospy(listener, *, *).
 %%% Reset spied pattern (listener:*/*) %%%
listener:listener/*
```

nospy(Predicate, Arity)

デフォルトモジュールを省略した場合のスパイの解除. Predicate, Arity にはワイルドカードを使用することができる.

nospy

スパイの全解除. `nospy(*,*,*,*)` を実行した場合と同じである.

spying [si]

スパイの設定形式を解析して, スパイが設定されている述語を一覧表示する.

【例 1】スパイ対象述語の一覧表示

```
[1] spy(listener,*,*).
 Spypoint set on listener: */ *

[2] spying.
 %%% SPYING PATTERNS (1 pattern) %%%
nakao:: listener: */ *
 %%% SPYING PREDICATES (29 predicates) %%%
nakao::
  listener:
    go/ 0 (public) - nakao:: listener: */ *
    go/ 2 (public) - nakao:: listener: */ *
    go/ 3 (public) - nakao:: listener: */ *
    go/ 4 (public) - nakao:: listener: */ *
    terminate/ 8 - nakao:: listener: */ *
      ...
      ...
    prompt/ 4 - nakao:: listener: */ *
 ... (10/29) ...?
```

スパイ対象述語の一覧はデフォルトで 10 述語ごとに表示され, 公開述語には (public) の印が付けられる. また, 対応するスパイの設定形式も表示される. 一覧表示中に入力できるコマンドは次の通り.

$< cr >$	:	list 10 predicates
N(>0)	:	list N predicates
q	:	quit
h	:	help

以下に各操作の意味を記す.

List 10 predicates :: リターン
一覧表示の継続

List predicates :: N (N>0)
N で指定された数だけの述語を表示する. 以降, スパイ対象述語は N 個ずつ表示される.

Quit :: q
一覧表示の終了

Help :: h
ヘルプ

spying(ModuleSpec)

spying(Package, ModuleName) [si]

Package, ModuleName (ModuleSpec) で指定されたモジュール中のスパイの対象となっている述語を一覧表示する. Package, ModuleName にはワイルドカード "*" が使用できる. なお, 一覧表示中に入力できるコマンドは spying/0 と同じである.

【例】モジュールを指定したスパイ対象述語の一覧表示

```
[1] spy(*, go, *), spy(tee, *, *).
 Spypoint set on *:go/*
 Spypoint set on tee:*/*

[2] spying(tee).
 %%% SPIED PATTERNS (2 patterns) %%%
nakao:: *: go/ *
nakao:: tee: */ *
 %%% SPYING PREDICATES (2 predicates) %%%
nakao::
  tee:
    go/ 0 (public) - nakao:: tee: */ *, nakao:: *: go/ *
    loop/ 3 - nakao:: tee: */ *
```

trace(ModuleSpec, Predicate, Arity)

trace(Package, ModuleName, Predicate, Arity) [tr]

Package, ModuleName, Predicate, Arity で指定された述語にデフォルトのトレースモード (特に指定がなくてもリターンの入力のみでトレースを行うかどうかのモード) を設定する. Package, ModuleName, Predicate, Arity にはワイルドカード ("*") を使用できる. デフォルトのトレースモードが設定されている述語は, トレース実行中 "*" の印を付けてゴールの出力が行われる. なお, リスナ起動時には, デフォルトパッケージに属するモジュール内の述語には全てデフォルトのトレースモードが設定される. また, デフォルトパッケージを変更した場合も同様である. 詳しくは 第 4.1.7.4 節 p.123 を参照のこと.

【注意】デフォルトのトレースモードはスパイの実行に何ら影響を与えない.

notrace(ModuleSpec, Predicate, Arity)

notrace(Package, ModuleName, Predicate, Arity)　[ntr]

Package, ModuleName, Predicate, Arity なる述語のデフォルトのトレースモードを解除する. Package, ModuleName, Predicate, Arity にはワイルドカード ("*") を使用できる.

【注意】スパイ解除の場合と同様に, ワイルドカードによりデフォルトのトレースモードが設定されている場合は, 述語毎に個別に解除することができない.

notrace_all　[ntr_all]

デフォルトのトレースモードの全解除. `notrace(*,*,*,*)` を実行した場合と同じである.

tracing　[tri]

デフォルトのトレースモードの設定形式を一覧表示する.

display_ancestor

display_ancestor(Mode)　[da]

スパイ実行時に出力される祖先ゴールの表示形式を Mode に変更する. 祖先ゴールとはトレース中にリダクション (フォーク) スパイモードでの実行を指定したゴールのことである. Mode には full, abbr, no のいずれかを指定できる. 初期値は **abbr** である. なお, 引数が省略された場合は, 現在の値を表示してユーザからの入力待ちとなる.

full: 祖先ゴールの引数まで表示する.

```
{SpyReduction}
[3] layered:n_queens(4,Ans).
<1> layered: n_queens(4,[1*C, ...]) -->     % 祖先ゴール
@0@4096    #1 + filter(begin,1,1,D)         % スパイ対象ゴール
                 D= begin?
```

abbr: 祖先ゴールを ModuleSpec:Predicate/Arity の形式で表示する.

```
{SpyReduction}
[3] layered:n_queens(4,Ans).
<1> layered: n_queens/2  -->                % 祖先ゴール
@0@4096    #1 + filter(begin,1,1,D)         % スパイ対象ゴール
                 D= begin?
```

no: 祖先ゴールを表示しない (ゴール番号だけが表示される).

```
[3] layered:n_queens(4,Ans).
<1> -->                                     % 祖先ゴールのゴール番号
@0@4096    #1 + filter(begin,1,1,D)         % スパイ対象ゴール
                 D= begin?
```

inspect(?Term)　[ins]

リスナトップレベルからインスペクタを呼び出す. Term で指定される任意の KL1 データをインスペクトする. この機能を用いることで複雑な構造を持つデータの中身を調べることができる. インスペクタについては 第 4.2 節 p.157 を参照のこと.

monitor(?Term) [m]

リスナトップレベルからモニタを呼び出す. Term で指定される任意の KL1 データの具体化の状況を監視表示する. Term に変数が指定された場合は, その変数印字名が Term の名前となる. モニタについては 第 4.2.5節 p.178 を参照のこと.

monitor(?Term, Name) [m]

リスナトップレベルからモニタを呼び出す. Term で指定される任意の KL1 データの具体化の状況を名前 Name で監視表示する. Name には任意のアトムかストリング, 若しくは変数を指定できる. モニタについては 第 4.2.5 節 p.178 を参照のこと.

【注意】 Name に変数が指定された場合は, その変数の印字名 (ストリング) が指定されたものとみなされる. 従って, この変数はリスナの変数プールを参照することはなく, また, 登録されることもない.

monitor_and_fork(?Term) [mf]

リスナトップレベルからモニタを呼び出して, そのプロセスをフォークする. 起動されたモニタの入出力は新たに生成されるウインドウに切り替えられる. Term に変数が指定された場合は, その変数印字名が Term の名前となる. モニタについては 第 4.2.5 節 p.178 を参照のこと.

monitor_and_fork(?Term, Name) [mf]

リスナトップレベルからモニタを呼び出して, そのプロセスをフォークする. 起動されたモニタの入出力は新たに生成されるウインドウに切り替えられる. Name には任意のアトムかストリング, 若しくは変数を指定できる. モニタについては 第 4.2.5 節 p.178 を参照のこと.

【注意】 Name に変数が指定された場合は, その変数の印字名 (ストリング) が指定されたものとみなされる. 従って, この変数はリスナの変数プールを参照することはなく, また, 登録されることもない.

4.1.6.9 ParaGraph 関連コマンド

ParaGraph は並列プログラムの実行状況をグラフィック出力する性能評価支援ツールである. リスナの **profile, profile_msg, profile_node** コマンドで収集した実行状況情報をもとにプログラムの振る舞いを様々なグラフを用いて表示する. ParaGraph はリスナトップレベルから **paragraph** コマンドを用いて起動する. ParaGraph については 第 5.2 節 p.190 を参照のこと. なお, 現在, 実行状況情報として以下のものが計測可能である.

1. 各ノードで実行された述語毎のリダクション数とサスペンション数
2. ノード間のメッセージの送受信情報と各ノードの稼働状況
3. 各ノードの計算時間, アイドル時間, メッセージの送受信時間, GC 実行時間

profile(Goals)

Goals で指定されるプログラムの実行状況情報を計測する. 実行状況情報は一定の時間間隔毎に収集され, プログラムの実行が終了するとリスナが持っている **データ格納用プール** に保存される. この時リスナウインドウにはデータ識別番号と入力ゴール列, 実行に要した総リダクション数と実行時間が表示される. 計測される情報は **各ノードで実行された述語毎のリダクション数とサスペンション数** である. 保存されているデータは **paragraph** コマンドでグラフ出力を行うことができる. また,

実行状況情報を収集する時間間隔 (サイクル時間) は **interval** コマンドで変更することができる. 既定値は 2000 ミリ秒である.

ゴールの指定方法

ゴールの指定方法は通常のリスナトップレベルにおける入力方法と同じである. マクロ表記やゴール中の変数の取り扱い, 実行ノード指定, デフォルトモジュール, 局所述語やシェルユーティリティプログラムの実行など全て同様に記述可能である. 複数のゴールを指定しても良い. **但し, 組込述語やリスナのコマンドを指定することはできない**.

【例 1】実行状況情報の計測

```
[7] current_node(_,N), profile( layered:n_queens(N,Ans) ).
 1: layered: n_queens(N,Ans)              % "1" はデータ識別番号
  19421 reductions
  989 msec
 Ans = [1*'[' & ']',2*'[' & ']', ... ]
```

【例 2】複数ゴールの計測 (一緒に計測する)

```
[8] profile( (test:generator(100,Data), test:consumer(Data,Ans)) ).
 2: test: generator(100,Data),test: consumer(Data,Ans)
  204 reductions
  401 msec
 Ans = [0,2,4,6, ... ]
 Data = [0,1,2,3,4, ... ]
```

【例 3】複数ゴールの計測 (別々に計測する)

```
[9] profile(test:generator(100,Data)), profile(test:consumer(Data,Ans)).
 3: test: generator(100,Data)
  103 reductions
  231 msec
 4: test: consumer(Data,Ans)
  103 reductions
  331 msec
 Data = [0,1,2,3,4, ... ]
 Ans = [0,2,4,6, ... ]
```

【例 4】実行ノード指定, デフォルトモジュールの省略と局所述語の実行

```
[10] profile((generator(_D),consumer(_D,_)@node(3))),
     profile(layered!queens(1,8,begin,_)).
 5: genarator(_D),consumer(_D,_)@ node(3)
  204 reductions
  398 msec
 6: layered! queens(1,8,begin,_)
  19421 reductions
  1024 msec
```

データ識別番号

ゴールの実行が終了するとリスナウインドウにデータを識別するための番号と入力ゴール列, リダクション数と実行時間が出力される. データ識別番号は一つのリスナ内でユニークであり, 一度に複数の **profile** コマンドを実行した場合は先に入力されたコマンドから順に割り付けられる. また, ParaGraph の **History manipulator** から実行状況情報をロードした場合にも割り付けられる.

エラーが発生した時

実行状況情報を計測中のゴールが何らかのエラーを発生した場合はエラーが報告される． エラー発生時のコマンドで実行を放棄すると，それまでの実行状況情報がデータ格納用プールに保存される．

【例 5】エラーが発生した時

```
[11] profile( layered:n_queens(f,Ans) ).
 Failure>> layered: queens/ 4
    Goal > queens(1,f,begin,Ans)
    Node > 0
!! Exception (13452 msec): a
7: layered: n_queens(f,Ans)
 44 reductions
 1351 msec
```

実行の中断と放棄

実行状況情報を計測中に Control-C の入力により実行を中断して **abort** コマンドを用いて実行を放棄すると，途中までの実行状況情報を獲得することができる．

【例 6】計測中に実行を中断した時

```
[11] profile( test:loop ).
?? Interruption (37564 msec): a
8: test: loop
 848 reductions
 1228 msec
Execution aborted !!
```

【参考】ユーザプログラムの実行が無限ループを引き起こしているような場合には，profile コマンドの実行を途中で放棄して，実行結果を出力すると，ループしている述語を発見することができる．

【参考】リスナ内に保存されている実行状況情報は ParaGraph でサポートされているコマンドにより，削除したり，ファイルにセーブしたりすることができる．また，ファイルにセーブされているデータをロードすることも可能である．

【注意 1】**profile** コマンドを実行すると必ず実行状況情報がデータ格納用プールに保存される．従って，不要なデータは適宜削除することが望ましい．

【注意 2】リスナ内に保存されている実行状況情報はリスナが終了すると消滅する．大切なデータはリスナを終了する前にファイルにセーブした方がよいだろう．

profile_msg(Goals)

Goals で指定されるプログラムの実行における，**ノード間のメッセージの送受信情報 および 各ノードの稼働状況** を計測する．ゴールの指定方法や，実行の中断 / 放棄などの操作方法は **profile** コマンドと同様である．

profile_node(Goals)

Goals で指定されるプログラムの実行における各ノードの **計算時間，アイドル時間，メッセージの送受信時間，GC 実行時間** を計測する． ゴールの指定方法や，実行の中断 / 放棄などの操作方法は **profile** コマンドと同様である．

paragraph [pg]

ParaGraph を起動する. **profile, profile_msg, profile_node** コマンドにより計測された実行状況情報をグラフ出力する. ParaGraph の操作方法については 第 5.2 節 p.190 を参照のこと.

【参考】ParaGraph をバックグラウンドで実行しておけば, ParaGraph 起動後に計測したデータでも **History manipulator** の **renew** コマンドを用いて随時取り込むことができる. 計測コマンドを実行する度に毎回 ParaGraph を起動する必要がなく便利である.

interval

interval(Interval)

実行状況情報を収集する時間間隔 (サイクル時間) を Interval で指定される値に変更する. 収集時間間隔はミリ秒単位で指定する. 実行が終了するまで収集を行わない時はアトム infinite を指定する. リスナ起動時の既定値は 2000 ミリ秒である.

4.1.6.10 ユーティリティコマンド

number(Integer)

Integer で指定された整数を 2, 8, 10, 16 進数で表示する.

【例】

```
[1] number(~(16'FFFFFFFF xor exception_tag#trace)).
 Binary  : 11011111111111111111111111111111
 Octa    : 33777777777
 Decimal : 3758096383 (-536870913)
 Hexa    : DFFFFFFF
```

calculator [calc]

電卓を起動する. 電卓が起動されると “!” のプロンプトが出力される. 任意の計算式 (または整数) を入力できる. リターンのみの入力で電卓は終了する.

【例】

```
[1] calc.
>Hexa Calculator(? for help)
! (10+2'100)+16'FF/3>>8'7
 Binary  : 1110
 Octa    : 16
 Decimal : 14
 Hexa    : E
! 16'FFFFFFFF xor exception_tag#trace
 Binary  : 11011111111111111111111111111111
 Octa    : 33777777777
 Decimal : 3758096383 (-536870913)
 Hexa    : DFFFFFFF
!
```

in(Stream)

リスナの標準入力ストリームを獲得する.

【参考】入力用フィルタ (buffer:input_filter) のプロトコルを試行する時に便利である.

out(Stream)

リスナの標準出力ストリームを獲得する.

【参考】出力用フィルタ (buffer:output_filter) のプロトコルを試行する時に便利である.

inter(Stream)

リスナの標準インタラクションストリームを獲得する.

【参考】入出力両用フィルタ (buffer:interaction_filter) のプロトコルを試行する時に便利である.

4.1.6.11 シェルユーティリティプログラム

リスナトップレベルからもシェルと同様に, コマンドとしてシェルユーティリティプログラムを実行することができる. したがって, プログラミング環境として不可欠であるコンパイラやリンカ, ローダ, アンローダ, KL1 プログラムの静的解析ツールである変数チェッカなどをコマンドとして起動することができる.

- 述語名が go である述語は, 述語名を省略してモジュール名とその引数のみを指定して実行できる.

  `ModuleName( Arg, ... )` または `Package::ModuleName( Arg, ... )`

  【例】シェルユーティリティプログラム の実行.

```
[1] compile("*.kl1").
```

- **述語名 go を省略して実行されたゴールはトレース, 及び, スパイの対象とならない**. トレースやスパイを行う場合には述語名 go まで指定しなければならない.

  【例 1】述語名 go を省略するとトレースが掛からない.

```
{Trace}
[3] varchk.
*** KL1 Variable Checker ***
VARCHK[o,l,v]
```

  【例 2】述語名 go を入力するとトレースできる

```
{Trace}
[3] varchk:go.
@0@4096   1   varchk: go
   2   + * (1)  pimos:: shoen: raise(256,get_std(in), ... )
   3   + * (2)  pimos:: shoen: raise(256,get_std(out), ... )
   4     * (3)  variable_check(ready,B, ... )?
```

- 述語名 go を省略して指定されたゴールが, リスナのコマンド, 及び, デフォルトモジュールが省略されたゴールの指定と衝突するような場合には, 次のような順番で適用される.

  1. リスナのコマンド 若しくは 組込述語.
  2. シェルユーティリティプログラム
  3. デフォルトモジュールによりモジュール名を省略して入力されたゴール

4.1.7 トレーサ

4.1.7.1 概要

プログラムのデバッグのために実行をトレースする機能で重要なことは, 必要な部分以外のトレースを抑制し, 本来のデバッグ対象に注意を集中できるようにすることである.

通常の逐次型言語では, プログラム中の一箇所から起動されたプログラム部分が実行を終えるまでは, 他の部分の実行を開始しない. このため, 実行中のある適当な時間間隔についてその中の実行を全てトレースすれば, ある箇所から起動されたプログラムのみに限定したトレースを行うことができる.

並列言語である KL1 のプログラムでは, 概念上ゴールは全て並列に実行される. 実際にも, 実行は必要なデータがそろったゴールから行われる. このため, 時間間隔を限ったトレース方式では, 望むようなトレースの抑制を実現することはできない.

そこで, 本トレーサではゴールの親子関係だけに注目して実行を追っていくトレース方式を実現している. この方式では, 呼び出し関係の木構造中でトレースが不要なゴールの枝に対してはトレースを抑制して実行を進めていく.

本トレーサではゴールの“**リダクション**”(ゴールをガード部の実行が成功するプログラム節の適用により, ボディに指定された子ゴールに置き換えること) をトレースの対象としている. したがって, トレースの実行は 1 つのリダクションの終了ごとに, その結果生じた子ゴール (サブゴール) を報告する形で進められていく. 報告される子ゴールには, ユーザ定義述語の他に, ボディ組込述語やボディユニフィケーションも含まれる.

いま,

```
a :- b, c.
b :- d, e.
c :- f, g, h.
d.
e :- g, h.
f :- g, h.
g :- h.
h.
```

というプログラムを考えると, このプログラムの実行過程は

$$a \to \begin{cases} b \to \begin{cases} d \\ e \to \begin{cases} g \to h \\ h \end{cases} \end{cases} \\ c \to \begin{cases} f \to \begin{cases} g \to h \\ h \end{cases} \\ g \to h \\ h \end{cases} \end{cases}$$

といった木構造をなす. ここで, トレースの抑制は, この木構造のうち任意の部分木についてトレースを行わないことを指定することにあたる.

トレース

ゴールがリダクションされた時に, その内容 (子ゴール) を報告する.

【例】トレースの実行例

```
{Trace}
[7] sample:qsort([5,1,4,2,3],List).
@0@4096   1#1   sample: qsort([5,1,4,2,3],List)
   2     * (1)  append(B,[5|C],List)
   3     * (2)  qsort(D,B)
   4     * (3)  qsort(E,C)
   5     * (4)  part([1,4,2,3],5,D,E)?
```

トレースモードで実行した network:go/1 のリダクションが出力される.

スパイ

本トレーサにおいてスパイとは, 特定の述語のリダクションのみを報告することである. 本トレーサでは 2 つのタイプのスパイを実現している.

リダクションスパイ

スパイ対象のゴールがリダクションされた時に, その内容の報告を行う. 従って, このスパイではスパイ対象のゴールがサスペンドしている場合には, それが解消されない限りそのリダクションの報告は行われない.

【例】リダクションスパイの実行

```
[11] spy(layered,colors,*).

[12] sr.

{SpyReduction}
[13] layered:map(Ans).
<1> layered: map/1 -->
@0@4096    #1 + colors(B)
                 [red,yellow,blue,white]=B ?
```

スパイが設定されている layered:colors/1 がリダクションされた時に, その内容が報告される.

フォークスパイ

スパイ対象のゴールをボディ部で呼び出しているゴールがリダクションされた時に, その内容の報告を行う. 即ち, スパイ対象ゴールの親にあたるゴールを報告する.

【例】フォークスパイの実行

```
[11] spy(layered,colors,*).

[2] sf.

{SpyFork}
[3] layered:map(Ans).
<1> layered:map(A) -->
@0@4096       area/3
   2   + * (1)  colors(B)?
```

スパイが設定されている layered:colors/1 をボディ部で呼び出している layered:area/3 がリダクションされた時に, その内容が報告される.

【注意】フォークスパイを実行した場合, 実装上の都合により検出されたスパイ対象ゴールを呼び出しているゴール (親ゴール) の引数の内容や, 兄弟ゴールを見ることはできない.

4.1.7.2 トレースの開始と終了

トレーサを起動するには次の二つの方法がある.

1. リスナトップレベルの実行モードを変更する. 実行モードの変更は **trace** (トレースモード), **spy_reduction** (スパイリダクションモード), **spy_fork** (スパイフォーク モード) のいずれかのコマンドで行う. また, トレース (スパイ) モードの終了は **notrace** コマンドにより行う.
2. ゴールを **trace(Goals), spy_reduction(Goals), spy_fork(Goals)** のいずれかのコマンドで実行する. リスナトップレベルのゴールの実行モードとは関係なくトレースを開始することができる.

これらのコマンドについては 第 4.1.6.8 節 p.107 を参照のこと. また, トレースを終了するにはトレースコマンド **a** (実行放棄) または **ntr** (トレース終了) を実行する.

4.1.7.3 トレース出力

トレース出力に関する注意事項

トレーサの出力は次の点においてユーザが記述したものと必ずしも一致しない.

1. 組込述語およびユニファイの実行順序は コンパイラによって変えられる場合がある.
2. ユーザの記述した組込述語は, コンパイラによってさらに複数の組込述語, ユニファイの組み合わせに展開される場合がある.

   【例】組込述語の展開

   ```
   X := Y+Z*R
   --> A := Z*R, B := Y+A, X=B.
   ```

3. 算術演算はマクロ表記で出力される.

   (a) 整数の演算

   ```
   add(X,Y,Z)
   --> Z := X*Y
   ```

   (b) 浮動小数点数の演算

   ```
   floating_point_add(X,Y,Z)
   --> Z $:= X+Y
   ```

4. 変数名はトレーサ内で生成する. 変数は先頭を大文字で表記する. 同じ名前の変数はデフォルトではひとつのトレーサ内で同一性が保証されるが, トレース中にガーベジコレクタが起動された場合には, 同じ変数であっても違う名前で表示される.

   【参考】トレーサの **scope** コマンドにより変数の同一性チェックの範囲を, 一回のトレース出力単位 (1 クローズ単位) に変更することも可能である. 変数番号 (例えば “A49” の 49 などのように) が大きくならずに見易いこと, トレースの出力が速くなるなどの利点がある.

トレース出力

トレーサは次のような出力を行う.

```
@0@4096  20#2   layered:queen(2,4,[1*L,2*T,3*E1|F1],X)            (例 1)
                  X=[2*H1|I1]
                  J1:=2+1                                        (例 2)
   30  + * (1)  filter([1*L,2*T,3*E1|F1],2,1,H1)
1  31    * (2)  queen(J1,4,[1*L,2*T,3*E1|F1],I1)?                (例 3)

<2> layered: queen/4 -->                                         (例 4)
@0@4096    #1 + filter(begin,1,1,D)                              (例 5)
                  D=begin?

<3> layered: queens/4 -->
@0@4096       queen/4                                            (例 6)
   4   +   (1)  filter(C,1,1,D)?

@0@4096   6#2   job:create(createJob( & ,Q, ... ))
   11    * (1)  @node(0) <= create1(createJob( ... ),Q, ... ) ?  (例 7)

@0@4096   1#3   queenw:queen(4,[1,2,3,4],A,4,0,1)
                  calculate_priority_from_current(-3000,B)
   1       (1)  @pri(B) => queen_n([1,2,3,4],[],[],A,4,0,0,4,1)?  (例 8)

@0@4096   1     listener:go(mouse,mouse, ... )
                  calculate_priority_from_minimum(4092,A)
                ....
   10    * (9)  @pri(A) => get_command(prompt,K, ... )?           (例 9)

[37-62]@0@4096  79#3   inspector_io_manager:manager(M6,M4, ... )  (例 10)
   87    * (1)  manager(M6,M4, ... )?
```

【例 1】トレースゴール

```
@0@4096  20#2   layered:queen(2,4,[1*L,2*T,3*E1|F1],X)
(a) (b) (c)(d)           (e)
```

【例 2】ボディ組込述語

```
       J1:=2+1
         (f)
```

【例 3】子ゴール

```
1  31    * (2)  queen(J1,4,[1*L,2*T,3*E1|F1],I1)?
(g)(c)  (h)(i)             (j)               (k)
```

【例 4】祖先ゴール

```
<2> layered: queen/4 -->
(l)           (m)
```

【例 5】リダクションスパイにより検出されたゴール

```
@0@4096    #1 + filter(begin,1,1,D)
             (n)       (o)
```

【例 6】フォークスパイにより検出された親ゴール

```
@0@4096       queen/4
                (p)
```

【例 7】ノード指定付き子ゴール

```
11    * (1)  @node(0) <= create1(createJob( ... ),Q, ... ) ?
             (q)
```

【例 8】実行優先度指定付き子ゴール (所属荘園内自己相対指定)

```
                calculate_priority_from_current(-3000,B)
1       (1)  @pri(B) => queen_n([1,2,3,4],[],[],A,4,0,0,4,1)?
             (r)
```

【例 9】実行優先度指定付き子ゴール (所属荘園内割合指定)

```
                calculate_priority_from_minimum(4092,A)
              ....
10    * (9)  @pri(A) => get_command(prompt,K, ... )?
             (r)
```

【例 10】ユーザ定義タスク内のゴール

```
  [37-62]@0@4096  79#3   inspector_io_manager:manager(M6,M4, ... )
   (s)
```

上記の例の中で, トレース表示中の各フィールドの意味は以下の通りである.

(a) ノード番号: ゴールをリダクションしたノードの番号.

(b) 優先度: ゴールが所属する荘園 (タスク) 内での優先度 (所属荘園内割合).

(c) ゴール番号: ゴールの識別番号. ゴールをトレースモードで実行する時にトレーサがゴールに対して一意に与える番号. この番号を管理しておくことでどのゴールがリダクションされたのかを知ることができる.

(d) クローズ番号: ゴールがコミットした節の番号 (1 オリジン). コンパイラのプロファイルモードを ON にしてコンパイルするとこの情報が付加される. 詳しくは, 第 4.1.7.5 節 p.124 を参照のこと.

(e) トレースゴール: リダクションされたゴール.

(f) ボディ組込述語

(g) トレーサ番号: トレース出力を行うトレーサの識別番号. マルチウインドウを利用したトレースを行う場合に出力される. 詳しくは, 第 4.1.7.7 節 p.125 を参照のこと.

(h) トレースマーク: 子ゴールにトレースが指定されていることを示すマーク. このマークが付いているゴールはリターンの入力のみでトレースが掛かる.

(i) 子ゴール通し番号: トレースゴールの子ゴールに付けられる通し番号. トレースコマンドで子ゴールを指定する際に使用する.

(j) 子ゴール: 子ゴールはトレースゴールとモジュール名が等しい場合には, モジュール名を省略して表示される.

(k) プロンプト: トレースコマンドの入力待ち状態である.

(l) 祖先ゴール番号: 祖先ゴールのゴール番号. (c) と同じ.

(m) 祖先ゴール: スパイによって検出されたゴールの祖先にあたるゴール (親の場合もある). 即ち, ユーザはこのゴールに対してスパイによる実行を指定したことになる.

(n) スパイマーク: ゴールがスパイの対象となっている述語のときに付けられるマーク.

(o) スパイゴール: リダクションスパイの実行により検出されたスパイ対象ゴール.

(p) 親ゴール: フォークスパイによって検出されたゴールの親 (スパイ対象ゴールをボディ部で呼び出しているゴール) にあたるゴール. 表示は述語名と引数個数を表す. 祖先ゴールとモジュール名が等しい場合には, モジュール名を省略して表示される.

(q) 転送先ノード: 子ゴールがノード指定付きゴールの場合に表示される.

(r) 実行優先度指定: 子ゴールが実行優先度指定付きゴールの場合に表示される. トレーサは実行優先度の指定方法に応じて, 実行優先度指定と共に次に示す述語を表示する.

- 所属荘園内自己相対指定の場合

```
calculate_priority_from_current(Rate,X)
...
@pri(X) => Goal
```

- 所属荘園内割合指定の場合

```
calculate_priority_from_minimum(Rate,X)
...
@pri(X) => Goal
```

それぞれ, Rate はユーザが指定した優先度の割合である.

【注意】現在 フォークスパイにより実行優先度指定付きのゴールが検出された場合, 優先度の指定方法を表す述語の表示は行われない. 今後改修する予定である.

(s) タスク (荘園) 識別子: トレース (スパイ) で報告されたゴールが所属しているタスクを区別するための識別子. 識別子中の番号はタスクを生成したゴール (通常は shoen:raise/3) のゴール番号である. また, ゴール番号の並びはタスクのネストの深さを表す. 詳しくは 第 4.1.7.8 節 p.125 を参照のこと.

4.1.7.4 デフォルトのトレースモード

トレースやスパイで報告されたゴールの子ゴールの実行を, 再びトレースするかどうかについては, トレース実行時に個別に指定することができるが, デフォルトをどちらにするかについては **trace** コマンドを利用することで, パッケージ毎, モジュール毎, 述語毎にモードを設定することができる.

デフォルトのトレースモードが設定されている子ゴールは, 特に指定をしなくてもリターンの入力のみでトレースを行うことが可能である. そのような子ゴールは, トレース出力においてトレースマーク ("*") を付加して表示される.

リスナ起動時には, デフォルトパッケージに属するモジュール内の述語は全てデフォルトでトレースができるように設定される. また, デフォルトパッケージを変更した場合も, 変更されたパッケージに属するモジュール内の述語には自動的にデフォルトのトレースモードが設定される.

【例】モジュール tee にデフォルトのトレースモードが設定されている時

```
[7] tee:go.
@0@4096   1   tee: go
   2       (1)  pimos:: shoen: raise(256,get_std(in), ... )
   3       (2)  pimos:: shoen: raise(256,get_std(out), ... )
   4       (3)  pimos:: shoen: raise(256,get_std(mes), ... )
   5     * (4)  loop(A,B, ... )?       % リターンの入力
@0@4096   5   tee: loop("abcdefg",C, ... )
                C=[prompt("TEE>"),getl(D)|E]
                  ...
```

モジュール tee 内の述語 loop(A,B,...) にはデフォルトのトレースモードが設定されているので, トレースマーク ("*") をつけて表示される. ここで, リターンの入力を行うと (4) 番のゴール loop だけがトレースされ, パッケージ pimos の述語はトレースされない.

【注意】デフォルトのトレースモードはスパイの実行に何ら影響を与えない.

4.1.7.5 クローズ番号の表示

コンパイラのプロファイルモードを ON にしてコンパイルを行うと, 生成されたモジュール内の述語をトレースする時に, ゴールがコミット (選択) したクローズ (節) の番号が表示される. クローズ番号は 1 オリジンであり, プログラム中の記述で上のクローズから下のクローズへと順番に割り付けられる. クローズ番号はトレース出力中 "ゴール番号" の後に "**# クローズ番号**" の形式で表示される.

```
@0@4096  2O#2   layered:queen(2,4,[1*L,2*T,3*E1|F1],X)
                  X=[2*H1|I1]
                  J1:=2+1
   30  + * (1)  filter([1*L,2*T,3*E1|F1],2,1,H1)
```

一行目の **#2** がコミットしたクローズの番号である. プロファイルモードを ON にするには, インタラクティブコンパイラ上で次のように **!set** コマンドを実行する.

```
[7] compile.
** KL1 Compiler **
COMPILE> !set compile:profile on
compile: profile := on
COMPILE>
```

同様に, プロファイルモードを解除するには **compile:profile := off**にすれば良い. コンパイラのプロファイルモードについては 第 3.1.3.4 節 p. 48 を参照のこと.

【参考】インタラクティブコンパイラで設定した環境はコンパイラの **!keep** コマンドでセーブすることができる. ファイル名は "**compile.param**" である. 以降, コンパイラ (インタラクティブもバッチも両方とも) が起動される際にはこのファイルの内容でコンパイラの初期設定が行われる.

【注意 1】実装上の都合により, フォークスパイで報告される親ゴールにはクローズ番号は表示されない.

【注意 2】プロファイルモードでコンパイルしても, プログラムの実行速度にはほとんど影響しない.

4.1.7.6 自動トレースと自動スパイ

トレーサを利用したデバッグでは, 何回も続けてリターンを入力したり, 特定の述語の実行状況を見るために繰り返しスパイモードで実行することが結構多い. このような場合には, 自動トレース, 自動スパイ機能を利用する. これらの機能を利用するとトレース実行時のコマンド入力の手間を省くことができる. また, 実行ログを採ることも簡単である.

自動トレース

トレーサの **リターン または c** コマンド (トレースモードに従った実行の継続) を指定された回数だけ繰り返し実行する. 従って, デフォルトのトレースモード ("*") が付いているゴールだけがトレースされる. 実行例は 第 4.1.7.9 節「**at** (Auto Trace) コマンド」 p.127 を参照のこと.

自動スパイ

トレーサの **s または sr** コマンド (全ての子ゴールをリダクションスパイにより実行する) を指定された回数だけ繰り返し実行する. 特定のゴールのリダクションのみをトレースしたい時に非常に便利である. 実行例は第 4.1.7.9 節「**as** (Auto reduction-Spy) コマンド」 p.131 を参照のこと.

4.1.7.7 マルチウインドウを利用したトレース

概要でも述べたように, KL1 プログラムの実行は全て並列に行われる. 従って, このような実行をそのままトレースしたのでは複数のプロセスの実行情報が無秩序に混じり合って報告されるため, デバッグは困難を極める. これを解決する一つの方法としては, 適当な単位のプロセス毎にトレース情報を分別, 整理して出力することである. マルチウインドウを利用したトレース方式はこの機能を提供する. ユーザは任意のプロセス (ゴール) 毎に専用のトレーサを起動することができる. 個々のプロセスが並列に実行されると, 個々のトレーサへのトレース出力も並列に行われる. 個々のトレーサへのコマンドの入力順序を変えるだけで, 簡単に実行を制御 (スケジューリング) することもできる.

トレーサの指定

子ゴールのトレース出力を行なうトレーサの設定は **f** (Fork) コマンドでトレーサ番号を指定して行う. トレーサ番号は個々のトレーサを識別するための番号であり, 任意 の 0 以上の整数である. リスナ本体で起動されているトレーサのトレーサ番号は 0 である. 既に存在しているトレーサを指定することも可能であるが, その場合はトレース出力はマージされる. なお, コマンドの実行例は 第 4.1.7.9 節 p.132 を参照のこと.

トレーサウインドウ

リス名本体とは別のトレーサを指定されている子ゴールがリダクションされると, そのトレーサがまだ生成されていない場合, トレーサウインドウが生成される. トレーサウインドウのサイズは予め適当な大きさに設定されているので, ユーザは任意の位置をマウスで指定する. 生成されたトレーサウインドウのラベルには次のようにトレーサ番号が表示される.

```
1: CHILD LISTENER
```

“1:” の “1” がトレーサ番号である. “CHILD LISTENER” はウインドウのタイトル.

トレーサの終了

タスクの実行が終了するとトレーサウインドウに

```
!!! Return to exit. !!!
```

の文字が出力される. ここでリターンを入力するとトレーサは消滅する.

注意事項

1. デフォルトのトレースモード 及び スパイ設定述語はリスナ本体の環境が継承される. 以後, これらの環境はトレーサ毎に個別に変更できる.
2. 環境変数 (カレントディレクトリやデフォルトパッケージも含む) も全てリスナ本体の環境が継承される. 以後, これらの環境はトレーサ毎に個別に変更できる.

4.1.7.8 タスク (荘園) 内トレース

ユーザがプログラムの中でタスク (荘園) を生成した場合, そのタスクの中の実行に対してもトレースやスパイを行うことができる. タスク内トレース (スパイ) の指定は, 以下に示したタスク (荘園) を生成する述語に対して行う.

pimos::shoen:raise(pimos_tag#task,general_request,GR)

ジェネラルリクエストデバイスへのストリーム GR に **execute** メッセージが流れて来た時にトレースが適用される.

pimos::shoen:execute/7

pimos::shoen:profile/7

【例】タスク内トレースの実行

```
@0@4096   3   test:create_task(C,E, ... )
                GR=[execute(C,Arg, ...)],
   7       (1)  pimos::shoen:raise(pimos_tag#task,general_request,GR)
   8     * (2)  monitor(Rpt,Ctr, ... )? 1          % (1),(2) をトレースする
    ...

    % ジェネラルリクエストを獲得するためのゴールに対してトレースを
    % 指定することでタスク内トレースを行える.

[7]@0@4096   7   test:create_child_task(A1,B1, ... )
                GR1=[execute(C1,D1, ...)],
   23      (1)  pimos::shoen:raise(pimos_tag#task,general_request,GR1)
   24    * (2)  new_monitor(J1,K1, ... )? sr 1     % (1) をスパイで実行する
    ...                                           % (2) をトレースする

    % ジェネラルリクエストを獲得するためのゴールに対してスパイを
    % 指定することでタスク内スパイを行える.

<23> pimos::shoen:raise/3 -->
[7-23]@0@4096    + layered:filter([],X1, ... )
   35    * (1)  filter([],Y1, ... )
    ...
```

タスク内で実行されているゴールのトレース出力には, そのゴールが所属しているタスクを表す識別子が表示される. この例では “[7]” および “[7-23]” がタスクの識別子となる. 識別子中の番号はそのタスクを生成したゴールのゴール番号である. また, ゴール番号の並びはタスクのネストの深さを表す.

“[7-23]” は, ゴール番号 7 番 (shoen:raise/3) のゴールの実行により生成されたタスクの中で, 更にゴール番号 23 番 (shoen:raise/3) のゴールの実行により子タスクが生成されたことを表している. リダクションスパイによりトレースの報告が行われたゴール (layered:filter) は 23 番のタスクの中で実行されている.

【注意】トレース中は, ユーザが生成するタスクの例外処理用のマスクパターンのトレースビット (ビット位置 29) は, **強制的に OFF にされる**.

4.1.7.9 トレースコマンド

トレース中に入力できるコマンドについて述べる. 各コマンドに対する表題は以下のような意味である.

コマンド名 :: 入力形式

なお, “<”, “>” で囲まれた引数は省略可能であることを表す.

Continue :: リターン または **c**

子ゴールに設定されているトレースモードに従って実行を継続する. トレースマーク (“*”) が付いているゴールだけがトレースされる.

【例 1】全ての子ゴールをトレースする

```
@0@4096   4   layered:queens(2,4,C,A)
                G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? <cr>
```

(1), (2) の子ゴールはトレースされる.

【例 2】全ての子ゴールをトレースしない

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8       (2)  queens(G,4,H,A)? <cr>
```

(1), (2) の子ゴールはトレースされない.

【例 3】トレースマーク ("*") が付いているゴールだけがトレースされる

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? <cr>
```

(1) の子ゴールはトレースされないが (2) の子ゴールはトレースされる.

Auto trace :: at COUNT

COUNT (正整数) で指定された回数だけ実行を継続する. 即ち, **リターン** を COUNT 回数 入力することと同じである. 従って, デフォルトのトレースモード ("*") が設定されているゴールだけがトレースされる.

【例】自動トレース

```
@0@4096   1   layered: n_queens(4,Ans)
   2     * (1)  queens(1,4,begin,Ans)? at 3        % 自動トレースを３回行う
@0@4096   2   layered: queens(1,4,begin,Ans)
                  B:= 1+ 1
   3     * (1)  queen(1,4,begin,C)
   4     * (2)  queens(B,4,C,Ans)? c % 2           % 残り２回
@0@4096   3   layered: queen(1,4,begin,C)
                  C=[1* D |E]
                  F:= 1+ 1
   5     * (1)  filter(begin,1,1,D)
   6     * (2)  queen(F,4,begin,E)? c % 1          % 残り１回
@0@4096   5   layered: filter(begin,1,1,D)
                  D= begin?                        % コマンド入力待ち
```

No trace all and Continue :: n

子ゴールに設定されているトレースモード ("*") に関係なく, 全ての子ゴールをトレースを抑制して実行する.

【例】全ての子ゴールをトレースを抑制して実行する.

```
@0@4096  25   test:go(IN,Status)
   26      (1)  pimos::buffer:output_filter(IN,C)
   27    * (2)  display(C,D)
   28    * (3)  check_status(D,Status)? n
```

設定されているトレースモードとは無関係に (1), (2), (3) の子ゴールはトレースされない.

【参考】後述の x コマンド (Toggle trace mode all and Continue) は, 表示されている全ての子ゴールのトレースモードを反転して実行する (トレースモードが ON なら OFF に, OFF なら ON

に). この例のようにボディ部で PIMOS が提供している述語と, ユーザ定義述語の両方を呼び出している様な場合に x コマンドを用いると PIMOS が提供している述語にはトレースが掛かってしまう. 表示されている全ての子ゴールの実行を抑制したい場合には **n** コマンドを用いると便利である.

Toggle trace mode all and Continue :: x

子ゴールに設定されているトレースモードを反転して実行を継続する (トレースモードが ON なら OFF に, OFF なら ON に). トレースマーク ("*") が付いていないゴールだけがトレースされる.

【例 1】全ての子ゴールをトレースしない

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? x
```

(1), (2) の子ゴールはトレースされない.

【例 2】全ての子ゴールをトレースする

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8       (2)  queens(G,4,H,A)? x
```

(1), (2) の子ゴールはトレースされる.

【例 3】トレースマーク ("*") が付いていないゴールだけがトレースされる

```
@0@4096   4   layered:queens(2,4,C,A)
                  G:=2+1
   7       (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? x
```

(1) の子ゴールはトレースされるが (2) の子ゴールはトレースされない.

Toggle trace mode selectively and Continue :: N1,N2,...

子ゴール通し番号 N1,N2,... で指定した子ゴールのトレースモードを反転して実行を継続する. 指定されなかった子ゴールは個別に設定されているトレースモードに従って実行される.

【例 1】トレースを個別に解除して実行を継続する

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? 1,2,3
```

(1),(2),(3) の子ゴールはトレースされないが, (4),(5),(6) の子ゴールはトレースされる.

【例 2】トレースを個別に指定して実行を継続する

```
@0@4096   1   layered:map(A)
   2       (1)  area(ok,begin,B)
   3       (2)  area(on(ok),B,C)
   4       (3)  area(on(on(ok)),C,D)
   5       (4)  area(on(off(on(ok))),D,E)
   6       (5)  area(off(on(on(ok))),E,F)
   7       (6)  area(on(on(on(ok))),F,A)? 1,2,3
```

(1),(2),(3) の子ゴールはトレースされるが, (4),(5),(6) の子ゴールはトレースされない.

Toggle trace mode all :: t

全ての子ゴールのトレースモードを反転する. トレースモードを反転するだけで, ゴールの実行は行わない.

【例】トレースモードを反転する

```
@0@4096   4   layered:queens(2,4,C,A)
                 G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? t
@0@4096   4   layered:queens(2,4,C,A)
                 G:=2+1
   7       (1)  queen(1,4,C,H)
   8       (2)  queens(G,4,H,A)?
```

全ての子ゴールのトレースモードが反転された.

Toggle trace mode selectively :: t N1,N2,...

子ゴール通し番号 N1,N2,... で指定した子ゴールのトレースモードを反転する. トレースモードを反転するだけで, ゴールの実行はまだ行わない.

【例】トレースモードを個別に反転する

```
@0@4096   4   layered:queens(2,4,C,A)
                 G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? t 1
@0@4096   4   layered:queens(2,4,C,A)
                 G:=2+1
   7       (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)?
```

(1) の子ゴールのトレースの指定が解除された.

Set reduction-spy mode all and Continue :: s または sr

全ての子ゴールをリダクションスパイにより実行する. スパイの対象となる子孫ゴールがリデュースされるまでトレースの報告は行われない.

【例】全ての子ゴールをリダクションスパイで実行する.

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? s
```

全ての子ゴールがトレースモードに関係なく, リダクションスパイにより実行される.

【注意】スパイ対象となっている子ゴール (スパイマーク “+” が付いているゴール) をリダクションスパイで実行すると, そのゴール自身のリダクションが報告される. つまり, トレースモードで実行することと同じになる.

```
@0@4096  37   listener_utility: integer_to_string(123,X)
   38   + * (1)  i_to_s(123,"",X)? s              % i_to_s/3 にはスパイが設定されている
@0@4096  38 + listener_utility: i_to_s(123,"",X)  % 38 番のゴールがそのままトレースされる
                 B:= 123 mod 10
                 C:= 123 / 10
   39    * (1)  integer_to_string(B,D)
   40      (2)  pimos:: builtin_utility: append_string(D,"",E)
   41  + * (3)  i_to_s(C,E,X)?
```

38 番のゴール listener_utility:i_to_s/3 にはスパイが設定されているため, このゴールをスパイリダクションで実行すると, このゴール自身のリダクションが報告される.

Set reduction-spy mode selectively and Continue :: s N1,N2,... または sr N1,N2,...

子ゴール通し番号 N1,N2,... で指定した子ゴールをリダクションスパイにより実行する. 指定されなかった子ゴールは個別に設定されているトレースモードに従って実行される.

【例 1】リダクションスパイを個別に指定して実行する

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? s 1,2,3
```

(1), (2), (3) の子ゴールはリダクションスパイにより実行され, (4), (5), (6) の子ゴールはトレースされる.

【例 2】リダクションスパイを個別に指定して実行する

```
@0@4096   1   layered:map(A)
   2       (1)  area(ok,begin,B)
   3       (2)  area(on(ok),B,C)
   4       (3)  area(on(on(ok)),C,D)
   5       (4)  area(on(off(on(ok))),D,E)
   6       (5)  area(off(on(on(ok))),E,F)
   7       (6)  area(on(on(on(ok))),F,A)? s 1,2,3
```

(1), (2), (3) の子ゴールはリダクションスパイにより実行され, (4), (5), (6) の子ゴールはトレースされない.

Set fork-spy mode all and Continue :: sf

全ての子ゴールをフォークスパイにより実行する. スパイの対象となる子孫ゴールが生成されるとトレースの報告が行われる.

【例】全ての子ゴールをフォークスパイにより実行する

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? sf
```

全ての子ゴールがフォークスパイにより実行される

Set fork-spy mode selectively and Continue :: sf N1,N2,...

子ゴール通し番号 N1,N2,... で指定した子ゴールをフォークスパイにより実行する. 指定されなかった子ゴールは個別に設定されているトレースモードに従って実行される.

【例 1】フォークスパイを個別に指定して実行する

```
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? sf 1,2,3
```

(1), (2), (3) の子ゴールはフォークスパイにより実行され, (4), (5), (6) の子ゴールはトレースされる.

【例 2】フォークスパイを個別に指定して実行する

```
@0@4096   1   layered:map(A)
   2      (1)  area(ok,begin,B)
   3      (2)  area(on(ok),B,C)
   4      (3)  area(on(on(ok)),C,D)
   5      (4)  area(on(off(on(ok))),D,E)
   6      (5)  area(off(on(on(ok))),E,F)
   7      (6)  area(on(on(on(ok))),F,A)? sf 1,2,3
```

(1), (2), (3) の子ゴールはフォークスパイにより実行され, (4), (5), (6) の子ゴールはトレースされない.

Auto reduction-spy :: as COUNT

COUNT (正整数) で指定された回数だけリダクションスパイを実行する. 即ち, **s** または **sr** コマンドを COUNT 回数 続けて入力することと同じである.

【例】自動スパイ

```
[7] spy(layered,queen,*).
 Spypoint set on layered:queen/ * .

[8] trace( layered:n_queens(4,Ans) ).
@0@4096   1   layered: n_queens(4,Ans)
   2     * (1)  queens(1,4,begin,Ans)? as 3        % 自動スパイを 3回行う
<2> layered: queens/4 -->
@0@4096     + queen(1,4,begin,B)
                 B=[1* C |D]
                 E:= 1+ 1
   3     * (1)  filter(begin,1,1,C)
   4   + * (2)  queen(E,4,begin,D)? s % 2          % 残り 2回
@0@4096   4 + layered: queen(2,4,begin,D)
                 D=[2* F |G]
                 H:= 2+ 1
   5     * (1)  filter(begin,2,1,F)
   6   + * (2)  queen(H,4,begin,G)? s % 1          % 残り 1回
@0@4096   6 + layered: queen(3,4,begin,G)
                 G=[3* I |J]
                 K:= 3+ 1
   7     * (1)  filter(begin,3,1,I)
   8   + * (2)  queen(K,4,begin,J)?               % コマンド入力待ち
```

No trace anymore :: ntr

以降トレースしない.

Abort :: a

実行を放棄する.

Break :: b

トレースを中断する. 実行を中断したままリスナを一段深くネストしたトップレベルの状態にする. このときトップレベルには中断のレベルが表示される.

Fork all (set the tracer to trace) :: f

子ゴールのトレース出力を行うトレーサを指定する (マルチウインドウを利用したトレース). 以降, 全ての子ゴールのトレース出力は, ユーザが指定したトレーサのウインドウに対して行われる. このコマンドを入力すると

```
Tracer number:
```

のプロンプトが出力されるので, トレーサ番号 (任意の 0 以上の整数) を入力する. リスナ本体のトレーサ番号は 0 番である. 既に存在しているトレーサを指定した場合はトレース出力はマージされる. 子ゴールに出力先トレーサが設定されると, 行の先頭にトレーサ番号が表示される. なお, マルチウインドウを利用したトレース方式については 第 4.1.7.7 節 p.125 を参照のこと.

【例】マルチウインドウを利用したトレースの指定

```
@0@4096   4   layered:queens(2,4,C,A)
                 G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? f
 Tracer number: 1                          % トレーサ番号の入力
@0@4096   4   layered:queens(2,4,C,A)
                 G:=2+1
1  7     * (1)  queen(1,4,C,H)              % 先頭の "1" はトレーサ番号
1  8     * (2)  queens(G,4,H,A)? s 2        % (1) はトレース
                                           % (2) はリダクションスパイ
```

以降, (1),(2) の子ゴールのトレース出力はトレーサ番号 1 番のトレーサに対して行われる.

Fork selectively (set the tracer to trace) :: f N1,N2,...

子ゴールのトレース出力を行うトレーサを指定する (マルチウインドウを利用したトレース). 以降, 子ゴール通し番号 N1,N2,... で指定した子ゴールのトレース出力は, ユーザが指定したトレーサのウインドウに対して行われる. このコマンドを入力すると,

```
Tracer number:
```

のプロンプトが出力されるので, トレーサ番号 (任意の 0 以上の整数) を入力する. リスナ本体のトレーサ番号は 0 番である. 既に存在しているトレーサを指定した場合はトレース出力はマージされる. 子ゴールに出力先トレーサが設定されると, 行の先頭にトレーサ番号が表示される. なお, マルチウインドウを利用したトレース方式については 第 4.1.7.7 節 p.125 を参照のこと.

【例 1】マルチウインドウを利用したトレースの指定

```
@0@4096   4   layered: queens(2,4,C,A)
                G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? f 1       % 個別に出力先トレーサを指定
 Tracer number: 1                          % トレーサ番号の入力
@0@4096   4   layered: queens(2,4,C,A)
                G:=2+1
1  7     * (1)  queen(1,4,C,H)             % 先頭の "1" はトレーサ番号
   8     * (2)  queens(G,4,H,A)?           % (1) はトレーサ 1 番に
                                           % (2) の出力先は変わらない
```

以降, (1) の子ゴールのトレース出力はトレーサ番号 1 番のトレーサに対して行われる. (2) の子ゴールのトレース出力は現状のまま変わらない.

【例 2】マルチウインドウを利用したトレースの指定

```
@0@4096   1   layered: map(Ans)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off & )),D,E)
   6     * (5)  area(off(on & )),E,F)
   7     * (6)  area(on(on & )),F,Ans)? f 3,4      % 個別に出力先トレーサを指定
 Tracer number: 1                                  % トレーサ 1 番の指定
@0@4096   1   layered: map(Ans)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
1  4     * (3)  area(on(on(ok)),C,D)               % 先頭の "1" はトレーサ番号
1  5     * (4)  area(on(off & )),D,E)              % 先頭の "1" はトレーサ番号
   6     * (5)  area(off(on & )),E,F)
   7     * (6)  area(on(on & )),F,Ans)? f 5,6      % 個別に出力先トレーサを指定
 Tracer number: 2                                  % トレーサ 2 番の指定
@0@4096   1   layered: map(Ans)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
1  4     * (3)  area(on(on(ok)),C,D)
1  5     * (4)  area(on(off & )),D,E)
2  6     * (5)  area(off(on & )),E,F)              % 先頭の "2" はトレーサ番号
2  7     * (6)  area(on(on & )),F,Ans)?            % 先頭の "2" はトレーサ番号
```

以降, (3),(4) の子ゴールのトレース出力は 1 番のトレーサに, (5),(6) の子ゴールのトレース出力は 2 番のトレーサに対して行われる. (1),(2) の子ゴールのトレース出力は現状のまま変わらない.

Toggle keep mode all :: k

全ての子ゴールの実行保留モードを反転する. 子ゴールに実行の保留が指定されていなければ実行保留が設定される. 既に実行の保留が指定してある場合には指定が解除される. この機能を用いることで動的にゴールの実行のスケジューリングを変えることができる.

【例】実行保留モードを反転する

```
@0@4096   4   layered:queens(2,4,C,A)
                G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)? k
@0@4096   4   layered:queens(2,4,C,A)
                G:=2+1
   7     k (1)  queen(1,4,C,H)
   8     k (2)  queens(G,4,H,A)? k          % 保留指定
@0@4096   4   layered:queens(2,4,C,A)
                G:=2+1
   7     * (1)  queen(1,4,C,H)
   8     * (2)  queens(G,4,H,A)?            % 指定解除
```

サブゴール通し番号の前についている “k” の印は, ゴールの実行の保留が指定されていることを表す.

【参考 1】トレース中のゴールの状態には次の 3 つが存在することになる. なお, 一つのゴールが複数の状態を持つことはない.

1. 荘園内で実行中.
2. トレーサ内でトレース情報の報告待ち / 報告中.
3. 実行が保留されている状態.

Toggle keep mode selectively :: k N1,N2,...

子ゴール通し番号 N1,N2,... で指定した子ゴールの実行保留モードを反転する. 子ゴールに実行の保留が指定されていなければ実行保留が設定される. 既に実行の保留が指定してある場合には指定が解除される.

【例】実行保留モードを個別に反転する

```
{Trace}
[7] layered:map(Ans).
@0@4096   1   layered:map(A)
   2     * (1)  area(ok,begin,B)
   3     * (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? k 1,2,3   % 実行保留の指定
@0@4096   1   layered:map(A)
   2     k (1)  area(ok,begin,B)
   3     k (2)  area(on(ok),B,C)
   4     k (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? k 3       % 保留指定の解除
@0@4096   1   layered:map(A)
   2     k (1)  area(ok,begin,B)
   3     k (2)  area(on(ok),B,C)
   4     * (3)  area(on(on(ok)),C,D)
   5     * (4)  area(on(off(on(ok))),D,E)
   6     * (5)  area(off(on(on(ok))),E,F)
   7     * (6)  area(on(on(on(ok))),F,A)? 3,4       % トレースの抑制

(1),(2) - 実行が保留される
(3),(4) - トレースされない
(5),(6) - トレースされる
```

List kept goals :: kg <TRACERS>

実行が保留されているゴールの一覧を表示する. TRACERS にはトレーサ番号又は アトム all を指定する. TRACERS を省略した場合は, このコマンドを実行したトレーサが保留しているゴールの一覧を表示する. トレーサ番号を指定した場合は, 該当するトレーサが保留しているゴールの一覧を表示する. また, トレーサ番号に all を指定すると, 全トレーサに渡り保留されているゴールを全て表示する.

【例 1】保留ゴールの一覧表示

```
@0@4096   5   layered:area(on(off( & )),D,E)
   8     * (1)  colors(G)
   9     * (2)  area1(on(off(on(ok))),D,G,E)? kg
 %%% KEPT GOALS (3/5 goals) %%%
k:2 layered:area(ok,begin,B)
k:3 layered:area(on(ok),B,C)
k:4 layered:area(on(on(ok)),C,D)? tr 1          % トレースモードでの実行
```

タイトル “%%% KEPT GOALS (3/5 goals) %%%” 中の “3/5” はこのコマンドを実行したトレーサが保留しているゴールの数 (3 ゴール) と全トレーサで保留しているゴールの (5 ゴール) 数の割合を表す. また, k:“番号” の “番号” は **ゴール番号** である.

【例 2】全トレーサの保留ゴールの一覧を表示する

```
@0@4096   7   layered:area(on(on( & )),F,O)
   18    * (1)  colors(L)
   19    * (2)  area1(on(on( & )),F,L,O)? kg all
 %%% KEPT GOALS (2/6 goals) %%%
1:8  layered:colors(H)
1:9  layered:area1(ok,begin,H,B)
2:10 layered:colors(I)
2:11 layered:area1(on(on(ok)),C,I,D)
k:12 layered:colors(G)
k:13 layered:area(off(on( & )),E,G,F)? tr 12   % トレースモードでの実行
```

なお, 各行先頭の “1:8” や “2:10” などの “1” や “2” はトレーサ番号である. また, このコマンドを実行したトレーサが保留しているゴールには “k” のマークが付けられる.

一覧表示中に入力できるコマンドは次の通り.

ゴール番号群	:	保留ゴールをトレースを抑制して実行
tr ゴール番号群	:	保留ゴールをトレースする
s ゴール番号群	:	保留ゴールをリダクションスパイで実行
sr ゴール番号群	:	保留ゴールをリダクションスパイで実行
sf ゴール番号群	:	保留ゴールをフォークスパイで実行
spy モジュール名, 述語名, 引数個数	:	スパイの設定
nospy モジュール名, 述語名, 引数個数	:	スパイの解除
si < モジュール名 >	:	スパイの一覧表示
f < ゴール番号群 >	:	トレース出力を行うトレーサの指定
tg < トレーサ番号 >	:	リダクションされていないゴールの一覧
kg < トレーサ番号 >	:	保留ゴールの一覧
ins ゴール番号	:	保留ゴールのインスペクト
m 変数名 <, 名前 >	:	モニタの起動
mf 変数名 <, 名前 >	:	モニタを別ウインドウで起動
mm	:	モニタのストリーム自動監視モードの変更
mt	:	起動されているモニタの一覧
rem 変数名 <, 名前 >	:	変数の変数プールへの格納
l	:	変数プールに格納されている変数の一覧
v	:	変数プールに格納されている変数の値の表示
w < 深さ, 長さ >	:	一時的に出力制限値を変更して再表示
pd < 深さ >	:	構造体の深さの出力制限値の変更
pl < 長さ >	:	構造体の長さの出力制限値の変更
psl < 長さ >	:	ストリング要素数の出力制限値の変更
pal < 長さ >	:	アトム印字名文字数の出力制限値の変更
pv	:	変数の出力形式の変更
リターン または q	:	一覧から抜ける

<, > で囲まれた引数は省略できることを表す. また, **ゴール番号群** とはゴール番号をカンマ (“,”) で区切ったものである. 各用語, 概念については以下の節を参照のこと.

スパイの設定 → 第 4.1.6.8 節 p.109
スパイの解除 → 第 4.1.6.8 節 p.110
スパイの一覧 → 第 4.1.6.8 節 p.110
f コマンド → 第 4.1.7.9 節 p.132
tg コマンド → 第 4.1.7.9 節 p.137
インスペクタ → 第 4.2 節 p.157
モニタ → 第 4.2.5 節 p.178
ストリーム自動監視モード → 第 4.2.5.2 節 p.180
モニタの一覧 → 第 4.2.5.2 節 p.181
l コマンド → 第 4.1.6.5 節 p.103

【参考】保留ゴールの一覧はプログラムの実行を中断した時 (Control-C) にも見ることができる.

List traced goals :: tg <TRACERS>

トレースを指定して実行したゴールのうち, このコマンドの実行時点ではまだリダクションされていないものを一覧表示する. TRACERS にはトレーサ番号又は アトム all を指定する. TRACERS を省略した場合は, このコマンドを実行したトレーサに報告されるべきゴールのうち, まだリダクションされていないものを一覧表示する. トレーサ番号を指定した場合は, 該当するトレーサにおけるゴールの一覧を表示する. また, トレーサ番号に all を指定すると, 全トレーサに渡ってリダクションされていないゴールを全て表示する.

【参考】デッドロックするようなゴールはトレースを指定して実行していれば, 必ずこのゴール群の中に含まれている.

【例 1】リダクションされていないゴールの一覧

```
@0@4096   7   layered: colors(G)
                  [red,yellow,blue,white]= G ? tg
 %%% TRACED GOALS (5/7 goals) %%%
t:8  layered:area1(ok,begin,G,B)
t:10 layered:area1(on(ok),B,H,C)
t:12 layered:area1(on(on(ok)),C,I,D)
t:14 layered:area1(on(off( & )),D,J,E)
t:16 layered:area1(off(on( & )),E,K,F)?
```

タイトル “%%% TRACED GOALS (5/7 goals) %%%” 中の “5/7” はこのコマンドを実行したトレーサに報告されるべきゴールのうち, まだリダクションされていないゴールの数 (5 ゴール) と, 全トレーサでリダクションされていないゴールの数 (7 ゴール) の割合を表す. また, t:“番号” の “番号” は ゴール番号である.

【例 2】全トレーサに渡りリダクションされていないゴールを一覧表示する

```
@0@4096   7   layered: colors(G)
                  [red,yellow,blue,white]= G ? tg all
 %%% TRACED GOALS (2/5 goals) %%%
1:8  layered:area1(ok,begin,G,B)
2:10 layered:area1(on(ok),B,H,C)
2:12 layered:area1(on(on(ok)),C,I,D)
t:14 layered:area1(on(off( & )),D,J,E)
t:16 layered:area1(off(on( & )),E,K,F)?
```

なお, 各行先頭の “1:8” や “2:10” などの “1” や “2” はトレーサ番号である. また, このコマンドを実行したトレーサのゴールには “t” のマークが付けられる.

一覧表示中に入力できるコマンドは次の通り.

tg < トレーサ番号 >	:	リダクションされていないゴールの一覧
kg < トレーサ番号 >	:	保留ゴールの一覧
ins ゴール番号	:	ゴールのインスペクト
m 変数名 <, 名前 >	:	モニタの起動
mf 変数名 <, 名前 >	:	モニタを別ウインドウで起動
mm	:	モニタのストリーム自動監視モードの変更
mt	:	起動されているモニタの一覧
rem 変数名 <, 名前 >	:	変数の変数プールへの格納
l	:	変数プールに格納されている変数の一覧
v	:	変数プールに格納されている変数の値の表示
w < 深さ, 長さ >	:	一時的に出力制限値を変更して再表示
pd < 深さ >	:	構造体の深さの出力制限値の変更
pl < 長さ >	:	構造体の長さの出力制限値の変更
psl < 長さ >	:	ストリング要素数の出力制限値の変更
pal < 長さ >	:	アトム印字名文字数の出力制限値の変更
pv	:	変数の出力形式の変更
リターン または q	:	一覧から抜ける

<, > で囲まれた引数は省略できることを表す. 各用語, 概念については以下の節を参照のこと.

kg コマンド	→	第 4.1.7.9 節 p.135
インスペクタ	→	第 4.2 節 p.157
モニタ	→	第 4.2.5 節 p.178
ストリーム自動監視モード	→	第 4.2.5.2 節 p.180
モニタの一覧	→	第 4.2.5.2 節 p.181
l コマンド	→	第 4.1.6.5 節 p.103

【参考】リダクションされていないゴールの一覧はプログラムの実行を中断した時 (Control-C) にも見ることができる.

Set predicate spy on parent(trace) goal :: spy

親ゴール (トレースゴール) の述語にスパイが設定されていない時, その述語にスパイを設定する. スパイが設定されているゴールにはスパイマーク ("+") が付けられる.

【例】親ゴールの述語にスパイを設定する

```
@0@4096   3   layered:queens(2,4,C,A)
                  G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)? spy
@0@4096   3 + layered:queens(2,4,C,A)   % スパイマーク (+) が付く
                  G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)?
```

layered:queens/4 にスパイが設定された.

Reset predicate spy on parent(trace) goal :: spy

親ゴール (トレースゴール) の述語にスパイが設定されている時, その述語のスパイを解除する.

【例】親ゴールの述語のスパイを解除する

```
@0@4096   3 + layered:queens(2,4,C,A)
                G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)? spy
@0@4096   3   layered:queens(2,4,C,A)   % スパイマーク (+) が消える
                G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)?
```

layered:queens/4 のスパイが解除された.

【注意】ワイルドカードによりスパイ対象述語として登録されている述語は登録を抹消することができない.

Set predicate spy on subgoal :: spy N

子ゴール通し番号 N で指定されたゴールの述語にスパイを設定する.

【例】子ゴールの述語にスパイを設定する

```
@0@4096   3   layered:queens(2,4,C,A)
                G:=2+1
   6     * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)? spy 1
@0@4096   3   layered:queens(2,4,C,A)
                G:=2+1
   6   + * (1)  queen(1,4,C,H)          % スパイマーク (+) が付く
   7     * (2)  queens(G,4,H,A)?
```

layered:queen/4 にスパイが設定された.

Reset predicate spy on subgoal :: spy N

子ゴール通し番号 N で指定されたゴールの述語のスパイを解除する.

【例】子ゴールの述語のスパイを解除する

```
@0@4096   3   layered:queens(2,4,C,A)
                G:=2+1
   6   + * (1)  queen(1,4,C,H)
   7     * (2)  queens(G,4,H,A)? spy 1
@0@4096   3   layered:queens(2,4,C,A)
                G:=2+1
   6     * (1)  queen(1,4,C,H)          % スパイマーク (+) が消える
   7     * (2)  queens(G,4,H,A)?
```

layered:queen/4 のスパイが解除された.

Set predicate spy :: spy ModuleSpec, Predicate, Arity

ModuleSpec:Predicate/Arity なる述語にスパイを設定する. ModuleSpec, Predicate, Arity にはワイルドカード (“*”) を使用できる.

Reset predicate spy :: nospy ModuleSpec, Predicate, Arity

ModuleSpec:Predicate/Arity なる述語のスパイを解除する. ModuleSpec, Predicate, Arity にはワイルドカード (“*”) を使用できる.

List spying predicates :: si

スパイが設定されている述語を一覧表示する. 一覧表示中に入力できるコマンドは 4.1.6.8 を参照のこと.

List spying predicates in the specified module :: si ModuleSpec

ModuleSpec で指定されたモジュール内でスパイが設定されている述語を一覧表示する. 一覧表示中に入力できるコマンドは 4.1.6.8 を参照のこと.

Set default trace mode :: tr ModuleSpec, Predicate, Arity

ModuleSpec:Predicate/Arity なる述語のデフォルトのトレースモードを ON にする. ModuleSpec, Predicate, Arity の指定にはワイルドカード ("*") を使用できる. トレースモードが ON になったゴールにはトレースマーク ("*") が付けられる.

Reset default trace mode :: ntr ModuleSpec, Predicate, Arity

デフォルトのトレースモードが ON になっている ModuleSpec:Predicate/Arity なる述語を, トレースモード OFF に解除する. ModuleSpec, Predicate, Arity の指定にはワイルドカード ("*") を使用できる.

Reset default trace mode all :: ntr_all

デフォルトのトレースモードが ON になっている述語を全てトレースモード OFF に解除する.

List tracing predicates :: tri

デフォルトのトレースモードが ON になっている述語を一覧表示する.

Inspect parent(trace) goal :: ins

親ゴール (トレースゴール) をインスペクトする. インスペクタについては第 4.2 節 p.157 を参照のこと.

Inspect subgoal :: ins N

子ゴール通し番号 (N) で指定されたゴールをインスペクトする. インスペクタについては 第 4.2 節 p.157 を参照のこと.

Monitor variable :: m Variable <,?NewName>

Variable で指定された変数が具体化された時にその値をモニタする. NewName を指定すると, モニタする変数に適当な名前をつけることができ, 値の表示はその名前で表示される. NewName はアトム, 若しくは任意の変数である. NewName の指定がないときは Variable の変数名でモニタする. モニタについては 第 4.2.5 節 p.178 を参照のこと.

【例 1】変数の具体化をモニタする

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? m G     % 変数 G のモニタ
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? 1       % トレースを抑制する
aq:1 G == [red,yellow,blue,white] ?          % 具体化状況の報告
```

【例 2】名前を付けて変数の具体化をモニタする

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? m G,Colors
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? 1
aq:1 Colors == [red,yellow,blue,white] ?
```

Monitor variable and Fork :: mf Variable <,?NewName>

Variable で指定された変数が具体化された時にその値をモニタする. モニタの入出力は新たに生成されるウインドウに対して行われる. NewName を指定すると, モニタする変数に適当な名前をつけることができ, 値の表示はその名前で表示される. NewName はアトム, 若しくは任意の変数である. NewName の指定がないときは Variable の変数名でモニタする. モニタについては 第 4.2.5 節 p.178 を参照のこと.

Change monitor mode :: mm

モニタのストリーム自動監視モードを変更する. 以降, 起動されるモニタは変更されたモードで実行される. 詳しくは 第 4.2.5.2 節 p.180を参照のこと.

List monitored terms :: mt

モニタリングを指定した項のうち, このコマンドの実行時点ではまだ具体化の報告が行われていないものを一覧表示する. 詳しくは 第 4.2.5.2 節 p.181 を参照のこと.

Remember the binding :: rem Variable <,?NewVariable>

Variable で指定された変数をリスナの変数プールに登録する. NewVariable を指定すると, 新しい変数として登録される. NewVariable には下線で始まる名前や, リスナの変数プールに既に格納されている変数の名前を使うことができない.

【例 1】トレース出力中の変数を登録する

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? rem G     % 変数 G の格納
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(H)
   8     * (2)  area1(ok,begin,H,B)? 1         % トレースの抑制
@0@4096   2   layered: area(on(ok),B,C)
   9     * (1)  colors(I)
   10    * (2)  area1(on(ok),B,I,C)? v         % 変数の一覧
 Ans = Ans
 G = [red,yellow,blue,white]
```

【例 2】任意の名前でトレース出力中の変数を登録する

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? rem G,Colors
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(H)
   8     * (2)  area1(ok,begin,H,B)? 1
@0@4096   2   layered: area(on(ok),B,C)
   9     * (1)  colors(I)
   10    * (2)  area1(on(ok),B,I,C)? v
 Ans = Ans
 Colors = [red,yellow,blue,white]
```

List variables :: l

リスナの変数プールに格納されている変数の一覧を表示する. 詳しくは 第 4.1.6.5 節 **list** コマンド p.103を参照のこと.

【参考】リスナトップレベルにおいてゴールの引数として入力された変数 (その名前が下線で始まるものは除く) は, プログラムの実行が終了するまでその値が束縛されているので, 変数の値を記憶するモードには関係なく一覧を見ることができる.

Display variables :: v

リスナの変数プールに格納されている変数の一覧をその値と共に表示する.

【参考】リスナトップレベルにおいてゴールの引数として入力された変数 (その名前が下線で始まるものは除く) は, プログラムの実行が終了するまでその値が束縛されているので, 変数の値を記憶するモードには関係なく一覧を見ることができる.

Change holophrasting parameters :: ho

ホロフラスティングに関する環境変数値を変更する. 構造体の深さ, 長さ, ストリングの要素数, アトムの文字数の出力制限値の変更を行う. 現在の値を表示してユーザの値の入力待ちとなる. リターンを入力した場合は値は変更されない.

Change print depth :: pd Depth

構造体の深さの出力制限値を変更する. Depth は省略可能.

Change print length :: pl Length

構造体の長さ (要素の数) の出力制限値を変更する. Length は省略可能.

Change print string length :: psl Length

ストリングの要素の数の出力制限値を変更する. Length は省略可能.

Change print atom length :: pal Length

アトムの文字数の出力制限値を変更する. Length は省略可能.

Toggle print variable mode :: pv

変数の表示モード (名前 / アドレス) を切り替える.

Print environment variables :: env

トレーサに関する環境変数の値を表示する.

Rewrite reduction :: w

Print-Depth, Print-Length の値を一時的に変えてリダクションを再表示する. このコマンドを実行する時の Print-Depth, Print-Length の値は rewrite_depth, rewrite_length コマンドにより設定できる. なお, リスナ起動時の初期値は 30, 30 である. 詳しくは 第 4.1.6.3 節 p.97 を参照のこと.

【例】出力制限値を一時的に変えてリダクションを再表示する

```
@0@4096   7   layered: queens(3, ... )
                  0:= 3+ 1
   12    * (1)  queen(1, ... )
   13    * (2)  queens(0, ... )? w
@0@4096   7   layered: queens(3,4,[1* L |M],Ans)
                  0:= 3+ 1
   12    * (1)  queen(1,4,[1* L |M],P)
   13    * (2)  queens(0,4,P,Ans)? 1
@0@4096   8   layered: filter(begin, ... )
                  H= begin?
```

Rewrite reduction :: w Depth,Length

Print-Depth, Print-Length の値を一時的に Depth, Length に変えてリダクションを再表示する.

Change scope of variables :: scope <MODE>

トレース出力中の変数の同一性チェックの範囲を変更する. MODE には **tracer** または **clause**

を指定できる. MODE が tracer のときは, トレース出力中の同じ名前の変数はひとつのトレーサ内で (即ち, タスクの実行が終了するまで) 同一性が保証される. MODE が clause の時は, 一回のトレース出力単位 (節単位) で保証する.

【参考】clause モードでは, 変数番号 (例えば “A49” の 49 などのように) が大きくならずに見易いこと, トレースの出力が速くなるなどのメリットがあるが, ゴール間に渡る変数の同一性はチェックできない.

Change output format of ancestor goal :: da <MODE>
スパイ実行時に出力される祖先ゴールの表示形式を MODE に変更する. MODE には full, abbr, no のいずれかを指定できる. 詳しくは 第 4.1.6.8 節 p.112 を参照のこと.

Help :: h
ヘルプメニューを表示する.

4.1.7.10 トレーサのコマンド一覧

表 4.1 p.144 にトレーサのコマンドの一覧を示す.

4.1.8 エラー処理

ユーザの指定したゴールが失敗したり, 例外に対する処理を誤っていたりするなど, なんらかの異常を引き起こすと, リスナはエラーの報告を行う. この時ユーザは, インスペクタを用いてエラーの内容を詳しく解析したり, エラーを起こしたゴールの代わりに代替ゴールを指定して実行を継続することができる.

4.1.8.1 エラー報告時に入力できるコマンド

リスナはユーザプログラムの実行でエラーを検出すると, エラーの発生時刻 (プログラムの実行を開始してから, エラーが検出されるまでの実行時間) とその内容を報告する.

```
[7] layered:n_queens(f,Ans).
 Failure>> layered: queens/ 4
    Goal > queens(1,f,begin,Ans)
    Node > 0
 !! Exception (115 msec):
```

エラー発生時に入力できるコマンドは次の通り.

コマンド名	入力形式	意　　味	参照頁
Continue	**<cr>, c**	実行の継続	126
Auto trace	**at COUNT**	自動トレース	127
No trace & Continue	**n**	トレースの抑制	127
Toggle trace mode all & Continue	**x**	全ゴールのトレースモードの反転実行	128
Toggle trace mode & Continue	**N1,N2, ...**	トレースモードの反転実行	128
Toggle trace mode	**t <N1,N2, ...>**	トレースモードの反転	129
Apply reduction-spy	**s <N1,N2, ...>**	リダクションスパイによる実行	129
Apply reduction-spy	**sr <N1,N2, ...>**	リダクションスパイによる実行	129
Apply fork-spy	**sf <N1,N2, ...>**	フォークスパイによる実行	130
Auto reduction-spy	**as COUNT**	自動リダクションスパイ	131
No trace anymore	**ntr**	以降トレースしない	132
Abort	**a**	実行の放棄	132
Break	**b**	トレースの中断	132
Fork (set the tracer)	**f <N1,N2, ...>**	出力先トレーサの指定	132
Toggle keep mode	**k <N1,N2, ...>**	実行保留の指定	134
List kept goals	**kg <Tracers>**	実行保留ゴールの一覧	135
List traced goals	**tg <Tracers>**	未実行トレースゴールの一覧	137
Set(reset) spy on parent goal	**spy**	親ゴールのスパイの設定と解除	138
Set(reset) spy on subgoal	**spy Number**	サブゴールのスパイの設定と解除	139
Set spy	**spy M,P,A**	述語 M:P/A にスパイを設定する	139
Reset spy	**nospy M,P,A**	述語 M:P/A のスパイを解除する	139
List spying predicates	**si <ModSpec>**	スパイが設定述語の一覧	139
Set default trace mode	**tr M,P,A**	デフォルトトレースモードの設定	140
Reset default trace mode	**ntr M,P,A**	デフォルトトレースモードの解除	140
Reset default trace mode all	**ntr_all**	デフォルトトレースモードの全解除	140
List tracing predicates	**tri**	デフォルトトレースモード設定述語の一覧	140
Inspect parent goal	**ins**	親ゴールのインスペクト	140
Inspect subgoal	**ins Number**	サブゴールのインスペクト	140
Monitor variable	**m Var <,Name>**	モニタの起動	140
Monitor variable & Fork	**mf Var <,Name>**	モニタを別ウインドウで起動	141
Change monitor mode	**mm**	モニタのストーリーム自動監視モードの変更	141
List monitored terms	**mt**	モニタの一覧	141
Remember the binding	**rem Var <,Name>**	変数の変数プールへの格納	141
List variables	**l**	変数プールに格納されている変数の一覧	141
Display variables	**v**	変数プールに格納されている変数の値の表示	142
Change holophrasting parameters	**ho**	ホロフラスティングの変更	142
Change print depth	**pd <Dep>**	構造体の深さの出力制限値の変更	142
Change print length	**pl <Len>**	構造体の長さの出力制限値の変更	142
Change print string length	**psl <Len>**	ストリングの要素数の出力制限値の変更	142
Change print atom length	**pal <Len>**	アトム印字名の文字数の出力制限値の変更	142
Toggle print variable mode	**pv**	変数の出力モードの変更	142
Print environment variables	**env**	トレーサに関する環境変数値の表示	142
Rewrite reduction	**w <Dep,Len>**	一時的に出力制限値を変更して再表示	142
Change scope of variables	**scope <Mode>**	トレース出力中の変数のスコープの変更	142
Change format of ancestor	**da <Mode>**	祖先ゴールの出力形式の変更	143

表 4.1: トレーサのコマンド一覧

$< cr >$	:	abort & look into other exceptions
q	:	abort & look over other exceptions
a	:	abort & ignore other exceptions
b	:	break
c	:	continue
e	:	execute substitutive goals
ins	:	inspect exception
info	:	print exception informations
l	:	list variables
v	:	display variables
w <D,L>	:	rewrite exception
pd <D>	:	change print depth
pl <L>	:	change print length
psl <L>	:	change print string length
pal <L>	:	change print atom length
pv	:	toggle print variable mode
h, ?	:	help

Abort & Look into other exceptions :: リターン

実行を放棄する. 他にもエラーが検出されている場合には再びエラーの報告が行われ, ユーザはエラー発生時のコマンドを発行することができる (但し, 以降, 実行を継続したり, 代替ゴールを実行することはできない).

【例】エラー発生時にリターンを入力した時

```
[1] add(_,_,_), subtract(_,_,_).
 ?? Interruption (1017 msec): g          % 一括 GC の起動
 *** Deadlock detected during GC.
 Builtin> add(A,B,C)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
 !! Exception (5023 msec):               % リターンの入力
 *** Deadlock detected during GC.
 Builtin> subtract(D,E,F)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
 !! Exception (5123 msec): h             % コマンドの入力が可能
```

Abort & Look over other exceptions :: q

実行を放棄する. 他にもエラーが検出されている場合にはエラーの出力のみが行われる. ユーザはエラー発生時のコマンドを発行することができない.

【例】エラー発生時に q コマンドを入力した時

```
[1] add(_,_,_), subtract(_,_,_).
 ?? Interruption (1017 msec): g          % 一括 GC の起動
 *** Deadlock detected during GC.
 Builtin> add(A,B,C)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
 !! Exception (5023 msec): q             % q コマンドの入力
 *** Deadlock detected during GC.        % エラーの報告のみが行われる
 Builtin> subtract(D,E,F)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0

[2]                              % エラー発生時のコマンドは受け付けない
```

Abort & Ignore other exceptions :: a

実行を放棄する．他にエラーが検出されていても，エラーの報告は行われず，即座にリスナトップレベルに戻る．

【例】エラー発生時に **a** コマンドを入力した時

```
[1] add(_,_,_), subtract(_,_,_).
 ?? Interruption (1017 msec): g          % 一括 GC の起動
 *** Deadlock detected during GC.
 Builtin> add(A,B,C)
  Caller> pimos:: listener_cmd_builtin: do_command/ 1
    Node> 0
 !! Exception (5023 msec): a             % a コマンドの入力

[2]              % 他にエラーが検出されていても報告されない
```

【参考】一つのタスクの実行により，複数のエラーが報告される場合がある．例えば，一括 GC により一度に複数のデッドロックが検出されるような場合である．一般にエラーが発生する数は分からないものであるが，プログラムの性質によりたくさんのエラーが検出されることが予想できる場合には，このコマンドを利用することで不必要なエラーの報告を避けることができる．

Break :: b

エラー処理を中断したままリスナをトップレベルの状態にする．この時，トップレベルには中断のレベルが表示される．中断のレベルを下がるには **exit** を入力する．

Continue :: c

エラーを引き起こしたゴールの代わりに，代替ゴールとして true ゴール (何もしないゴール) を実行して実行を継続する．

【注意】発生したエラーがマージャデッドロック 若しくは マージャの不正入力である場合は，(捨てられた) マージャ入力変数に [ ] がユニファイされる．

【参考】このコマンドは未定義モジュール例外や未定義述語例外が発生した時に利用すると便利である．

【例】実行を継続する．

```
[1] test:go(A)
 Undefined module>> rsc_manager: create/1
             Goal > create(B)
           Caller > test: go/ 0
             Node > 0
 !! Exception (32452 msec): c              % 実行を継続する.
 A = A

[2]
```

Execute substitutive goals :: e

エラーを引き起こしたゴールの代わりに, 代替ゴールを指定して実行を継続する. 代替ゴールの指定方法はリスナトップレベルと同様である. 複数のゴールを指定しても良い. 但し, 組込述語は指定可能であるが, その他のリスナコマンドは指定できない.

【参考】このコマンドは未定義モジュール例外や未定義述語例外が発生した時に利用すると便利である.

【例】代替ゴールを実行する

```
[1] sieve:go(10,Ans).
 Undefined predicate>> generator: go/ 2
                Goal > go(10,B)
              Caller > sieve: go/ 2
                Node > 0
 !! Exception (43242 msec): e
 >Please execute substitutive goals.
 |?- B=[0,1,2,3,4,5,6,7,8,9].
 Ans = [0,2,4,6,8]
```

Inspect exception :: ins

例外情報をインスペクトする. インスペクタに渡される情報は, 例外事象

```
exception( Kind, Info, NewCode, NewArgv )
```

の Info である. Info については 第 7.5.4 節 p.292 を, インスペクタについては 第 4.2 節 p.157 を参照のこと.

Print exception informations :: info

例外を発生したゴールのモジュール情報を表示する. 表示される内容は, モジュールの生成日時, リンク日時, ソースファイル名とその生成日時, コンパイルするに当たって依存したファイル名とその生成日時 (include したファイルなど) である.

【例】モジュール情報の表示

```
[7] layered:n_queens(f,Ans).
v Failure>> layered: queens/4
    Goal > queens(1,f,begin,K)
    Node > 0
 !! Exception (135 msec): info
 %%% PREDICATE INFORMATIONS %%%
 Module  : nakao:: layered
 Created : 10-Aug-1991 20:03:14
 Linked  : 10-Aug-1991 20:03:14
 Source  : icpsi255:>sys>user>s-nakao>SAMPLE>...
         : 12-Jul-1990 19:55:53
```

【注意】IPL でロードされたモジュールはモジュール情報を見ることができない.

List variables :: l

リスナの変数プールに格納されている変数の一覧を表示する. 詳しくは 第 4.1.6.5 節 **list** コマンド p.103 を参照のこと.

【参考】リスナトップレベルにおいてゴールの引数として入力された変数 (その名前が下線で始まるものは除く) は, プログラムの実行が終了するまでその値が束縛されているので, 変数の値を記憶するモードには関係なく一覧を見ることができる.

Display variables :: v

リスナの変数プールに格納されている変数の一覧をその値と共に表示する.

【参考】リスナトップレベルにおいてゴールの引数として入力された変数 (その名前が下線で始まるものは除く) は, プログラムの実行が終了するまでその値が束縛されているので, 変数の値を記憶するモードには関係なく一覧を見ることができる.

Rewrite exception :: w

構造体の深さと長さの出力制限値を一時的に変えて例外情報を再表示する. このコマンドを実行する時の Print-Depth, Print-Length の値は rewrite_depth, rewrite_length コマンドにより設定できる. 詳しくは 第 4.1.6.3 節 p.97 を参照のこと.

【例】例外情報を再表示する.

```
[7] {0,1,2,3,4,5,6,7,8,9} = {0,1,2,3,4,5,6,7,8}.
 Unification failure>> {0,1,2, ... }
                With > {0,1,2, ... }
                Mode > 0
!! Exception (114 msec): w
Unification failure>> {0,1,2,3,4,5,6,7,8,9}
                With > {0,1,2,3,4,5,6,7,8}
                Mode > 0
!! --------- (114 msec):
```

Rewrite exception :: w Depth,Length

構造体の深さと長さの出力制限値を一時的に Depth, Length に変えて例外情報を再表示する.

Change print depth :: pd Depth

構造体の深さの出力制限値を変更する. Depth は省略可能.

Change print length :: pl Length

構造体の長さの出力制限値を変更する. Length は省略可能.

Change print string length :: psl Length

ストリングの要素数の出力制限値を変更する. Length は省略可能.

Change print atom length :: pal Length

アトムの文字数の出力制限値を変更する. Length は省略可能.

Toggle print variable mode :: pv

変数の表示モード (名前 / アドレス) を切替える.

Help :: h

ヘルプメニューを表示する.

4.1.8.2 エラーの再表示

エラーが発生した時に, エラーの内容をよく見ていないのに (例えばホロフラスティングのレベルが小さい時など), 誤ってエラー表示ルーチンから抜けてしまうことがある. また, 過去に発生したエラーの内容を, 再度, 確認したいこともよくある.

リスナが報告したエラーは, 通常はリスナが持っているエラーキューに格納される. エラーキューに保持されているエラー情報は **errors** や **last_error** コマンドにより, いつでも参照することができる. リスナ起動時のキューの長さは 5 である. エラーキューの長さは **error_queue_length** コマンドで変更することが可能である. エラーの発生件数がエラーキューの長さよりも多くなると, 古いエラー情報から抹消される. これらのコマンドについては 第 4.1.6.1 節 p.91 を参照のこと.

4.1.8.3 エラーメッセージ一覧

表 4.2 にリスナが表示するエラーメッセージの一覧を示す. 以下, 各エラーの発生例を紹介する. なお, デッドロック例外の発生例については 第 4.1.5.3 節 p.80 を参照のこと.

【例 1】イベントが存在しない

```
[77] !3.
 3: Event not found.
```

3 番のイベントは存在しない.

【例 2】リスナのマクロモジュールが不正なモジュールに更新された.

```
[7] layered#queen(8,Ans).
!WARNING! illegal macro module: my_macro
!CAUTION! switched to non-expansible mode
```

リスナのマクロモジュールが, マクロモジュールとしては不正なモジュール (例えば通常のモジュール) に更新されたため, 入力ゴールのマクロを展開できなくなった. このため, リスナは無展開モードに移行してしまったので注意を要する.

【例 3】リスナのマクロモジュールが存在しない (削除された)

```
[6] module_table:cache_table([remove_module(my_macro,S)]).
 S = normal

[7] layered#queen(8,Ans).
!WARNING! non-existent macro module: my_macro
!CAUTION! switched to non-expansible mode
```

リスナのマクロモジュールが削除されたため, 入力ゴールのマクロを展開できなくなった. このため, リスナは無展開モードに移行してしまったので注意を要する.

【例 4】パッケージが存在しない

```
[1] default_package(abc).
 Nonexistent package>> abc
```

パッケージ abc は存在しない.

【例 5】モジュールが存在しない

```
[1] hash:hash({1,2,3},Hash,Data).
 Nonexistent module>> hash
```

モジュール hash は存在しない.

【例 6】プロテクションのかかっているパッケージにアクセスした

`Event not found`	: イベントが存在しない	(例 1)
`!WARNING! illegal macro module`	: マクロモジュールが不正モジュールに更新された	(例 2)
`!WARNING! non-existent macro module`	: マクロモジュールが存在しない (削除された)	(例 3)
`Nonexistent package`	: パッケージが存在しない	(例 4)
`Nonexistent module`	: モジュールが存在しない	(例 5)
`Protection violation`	: 保護パッケージにアクセスした	(例 6)
`Nonexistent public predicate`	: 公開述語が存在しない	(例 7)
`Nonexistent local predicate`	: 局所述語が存在しない	(例 8)
`Local predicate`	: 局所述語を公開述語として実行した	(例 9)
`Public predicate`	: 公開述語を局所述語として実行した	(例 10)
`Default moduel is not set`	: デフォルトモジュールが未設定	(例 11)
`Nonexistent command`	: コマンド (組込述語) が存在しない	(例 12)
`Illegal argument`	: コマンドの引数の不正入力	(例 13)
`Illegal input`	: ゴールの入力が不正	(例 14)
`Illegal goal`	: ゴールの入力が不正	(例 15)
`Syntax error`	: 入力のシンタクスの誤り	(例 16)
`Illegal specification`	: 標準入出力デバイスの指定の誤り	(例 17)
`Cannot open file`	: 標準入出力デバイスが生成不能	(例 18)
`Failure`	: 候補節のガードが全て失敗した	(例 19)
`Unification failure`	: ボディユニフィケーションに失敗した	(例 20 - 21)
`Illegal merger input`	: マージャの不正入力	(例 22)
`Undefined module`	: 未定義モジュール内の述語を呼び出そうとした	(例 23)
`Undefined predicate`	: 未定義述語を呼び出そうとした	(例 24)
`Raised`	: リスナの知らない例外が発生した	(例 25)
`Trace exception occurred`	: リスナの知らないトレース例外が発生した	(例 26)
`Spy exception occurred`	: リスナの知らないスパイ例外が発生した	(例 27)
`Priority rate X is illegal input`	: ゴールの実行優先度指定が不正	(例 28)
`Priority rate X is range overflow`	: ゴールの実行優先度指定が正しい範囲にない	(例 29)
`Node number X is illegal input`	: ゴールの実行ノード指定が不正	(例 30)
`Cannot throw the goal at node X`	: ゴールの実行ノード指定が正しい範囲にない	(例 31)
`Exception`	: 組込述語の実行に例外が発生した	(例 32 - 38)
`Error`	: PIMOS に不正なメッセージが送られた	(例 39)
`Deadlock detected in during GC`	: 一括 GC 中にデッドロックを検出した	
`Merger deadlock detected in during GC`	: 一括 GC 中にマージャのデッドロックを検出した	
`Deadlock occurred. [×××]`	: MRB-GC によりデッドロックを検出した	
`Merger deadlock occurred. [×××]`	: MRB-GC によりマージャのデッドロックを検出した	
`Firmware error`	: 発生したら処理系のバグ	

表 4.2: リスナのエラーメッセージ一覧

```
[1] hanako::sample:n_queens(4,Ans).
 Protection violation>> hanako:: sample
```

パッケージ hanako は保護されているためアクセスできない.

【例 7】公開述語が存在しない

```
[1] pool:keyed_sort_bag([put(2,b), put(1,a), get_all(Content)]).
 Nonexistent public predicate>> pool: keyed_sort_bag/ 1
```

モジュール pool に keyed_sort_bag/1 は存在しない.

【例 8】局所述語が存在しない

```
[1] layered!area(Ans).
 Nonexistent local predicate>> layered: area/ 1
```

モジュール layered に area/1 は存在しない.

【例 9】局所述語を公開述語として実行した

```
[1] layered:queens(1,4,begin,List).
 Local predicate>> layered: queens/ 4
```

モジュール layered の queens/4 は局所述語である.

【例 10】公開述語を局所述語として実行した

```
[1] layered!n_queens(4,Ans).
 Public predicate>> layered: n_queens/ 2
```

モジュール layered の n_queens/2 は公開述語である.

【例 11】デフォルトモジュールが未設定

```
[1] !queens(1,4,begin,List).
 >> Default module is not set.
```

デフォルトのモジュールが設定されていないのに, モジュール名を省略したシンタクスで局所述語を実行しようとした.

【例 12】コマンドが存在しない

```
[1] string("listener",Size,Elm).
 Nonexistent command>> string("listener",Size,Elm)
```

コマンド string("listener",Size,Elm) は存在しない (ガード専用の組込述語だから).

【例 13】コマンドの引数の不正入力

```
[1] print_length(infinite).
 Illegal argument>> print_length infinite
```

コマンド print_length の引数 infinite は不正入力である.

【例 14】ゴールの入力が不正

```
[1] X:test(100).
 Illegal input>> X
```

モジュールの指定が変数である.

【例 15】ゴールの入力が不正

```
[1] X.
 Illegal goal>> X
```

ゴールとしての変数の入力は不正である.

【例 16】入力のシンタクスの誤り

```
[1] comparator:sort("applied",technology<Small,Large,Swapped).
*** Syntax Error ***
comparator:sort("applied",technology
*** Here ***
 <Small,Large,Swapped)
```

*** Here *** で示される位置にシンタクスの誤りがある.

【例 17】標準入出力デバイスの指定の誤り

```
[1] tee:go -> file("mes.doc") => windoe("OUT").
 Illegal specification>>  =>(windoe("OUT"))
```

標準出力のデバイスに windoe("OUT") の指定はできない.

【例 18】標準入出力デバイスが生成不能

```
[1] tee:go <= file("in.doc") => windoe("OUT").
 Cannot open file :  in.doc
```

標準入力のデバイスであるファイル “in.doc” は存在しないため生成不能.

【例 19】候補節のガードが全て失敗した

```
[1] layered:n_queens(*,Ans).
 Failure>> layered: queens/ 4
    Goal > queens(1,*,begin,Ans)
    Node > 0
```

ノード 0 番においてモジュール layered の述語 queens/4 の候補節が全て失敗した.

【例 20】ボディユニフィケーションに失敗した

```
[1] {1,2,3}={1,2,3,4}.
 Unification failure>> {1,2,3}
                With > {1,2,3,4}
              Caller > pimos:: listener_cmd_builtin: do_command/1
                Node > 0
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行された {1, 2, 3} と {1, 2, 3, 4} のユニフィケーションに失敗した.

【例 21】ボディユニフィケーションに失敗した

```
[1] {a,b,c}={a,d,c}.
  Unification failure>> d
                 With > b
                 Node > 0
```

ノード 0 番において d と b のユニフィケーションに失敗した.

【注意】構造体同士のユニフィケーションが要素のレベルで失敗した場合は, その要素が表示される.

【例 22】マージャの不正入力

```
[1] merge(abc,OUT).
 Illegal merger input>> abc
     New merger input > B
               Caller > pimos:: listener_cmd_builtin: do_command/1
                 Node > 0
 !! Execption (116 msec): e
 >Please execute substitutive goals.
 |?- B=[a,b,c].
 OUT = [a,b,c]
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行されたマージャに不正入力が発生した.

【参考】この例からも分かるように, マージャの不正入力が発生した場合は, 不正のあったマージャ入力 (この例ではアトム abc) と例外から復帰する時に使用するためのマージャ入力 (例外を起こしたマージャに正しいデータを送るための入力変数, この例では変数 B) が報告される. この入力変数を任意のストリーム (又はアトム []) に具体化することでマージャの実行を再開 (終了) することができる.

【例 23】未定義モジュール内の述語を呼び出そうとした.

```
[1] test:go.
 Undefined module>> abc: create/2
             Goal > create(100,A)
           Caller > test: go/ 0
             Node > 0
```

ノード 0 番において実行された test:go/0 の中で呼び出されている述語 abc:create/2 はモジュール abc が存在しないため実行不能である.

【例 24】未定義述語を呼び出そうとした.

```
[1] test:go.
 Undefined predicate>> test: create/2
                Goal > create(100,A)
              Caller > test: go/ 0
                Node > 0
```

ノード 0 番において実行された test:go/0 の中で呼び出されている述語 test:create/2 はモジュール test に定義されていないため実行不能である.

【例 25】リスナの知らない例外が発生した

```
[1] shoen:raise(user_tag#test,get_stream,Stream), Stream=[put(abc)].
 Raised>> get_stream
   Data > [put(abc)]
   Node > 0
```

ノード 0 番において shoen:raise(Tag,Info,Data) の Info=get_stream, Data=[put(abc)] の例外が発生した.

【例 26】リスナの知らないトレース例外が発生した

```
[1] module:get_code(layered,n_queens,2,Cde), apply_tracing(Cde,{4,Ans},1).
 >> Trace exception occurred.
 Trace ID> 1
 SubGoals>[internal( code$nakao::layered:queens/4$668 ,{1,4,begin,Ans})]
     Node> 0
```

ノード 0 番において, トレース識別子 1 のリスナの知らないトレース例外が発生した.

【例 27】リスナの知らないスパイ例外が発生した

```
[1] apply_spying(Cde,{4,Ans},{layered,filter,4},1).
 >> Spy exception occurred.
 Spy ID> 1
 Parent> layered: queen/ 4
   Goal> internal( code$nakao::layered:filter/4$668 ,begin(1,1,C))
   Node> 0
```

ノード 0 番において, スパイ識別子 1 のリスナの知らないスパイ例外が発生した.

【例 28】ゴールの実行優先度指定が不正

```
[1] pragma_test:go(1,rate).
 >> Priority rate "rate" is illegal input.
 Pragma> @ priority(*,rate)
 Caller> pragma_test: go/ 2
   Node> 0
```

ノード 0 番において pragma_test:go/2 のボディ部で呼ばれているゴールに指定してある実行優先度 @ priority(*,rate) は不正入力である.

【例 29】ゴールの実行優先度指定が正しい範囲にない

```
[1] pragma_test:go(1,-100).
 >> Priority rate "-100" is range overflow.
 Pragma> @ priority(*,-100)
 Caller> pragma_test: go/ 2
   Node> 0
```

ノード 0 番において pragma_test:go/2 のボディ部で呼ばれているゴールに指定してある実行優先度 @ priority(*,-100) は正しい範囲にない (所属荘園内割合指定では負の数による指定ができない).

【例 30】ゴールの実行ノード指定が不正

```
[1] pragma_test:go(3,node7).
 >> Node number "node7" is illegal input.
 Pragma> a/ 0 @ node(node7)
 Caller> pragma_test: go/ 2
   Node> 0
```

ノード 0 番において pragma_test:go/2 のボディ部で呼ばれているゴール a/0 に指定した実行ノード番号 node7 は不正入力である.

【例 31】ゴールの実行ノード指定が正しい範囲にない

```
[1] pragma_test:go(3,8).
 >> Cannot throw the goal at node "8".
 Pragma> a/ 0 @ node(8)
 Caller> pragma_test: go/ 2
   Node> 0
```

ノード 0 番において pragma_test:go/2 のボディ部で呼ばれているゴール a/0 に指定してある実行ノード 8 番は存在しないため実行不能 (例えば 8 台構成の場合).

【例 32】組込述語の引数が不正

```
[1] predicate_to_code(layered,n_queens,2,Code).
 Exception>> illegal_input_at_arg1
   Builtin > predicate_to_code(layered,n_queens,2,B)
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行された組込述語 predicate_to_code/4 の第 1 引数が不正である.

【例 33】組込述語の引数が正しい範囲にない

```
[1] new_string(String,10,64).
 Exception>> range_overflow
   Builtin > new_string(B,10,64)
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行された組込述語 new_string/3 の引数が正しい範囲にない (第 3 引数の要素サイズには 32 ビットよりも大きいサイズを指定できない).

【例 34】構造体のインデックス範囲逸脱

```
[1] new_vector(Vector,3),
   set_vector_element(Vector,3,Old,new,NewVector).
 Exception>> out_of_bounds
   Builtin > set_vector_element({0,0,0},3,D,new,E)
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行された組込述語 set_vector_element/5 の要素位置の指定が, 構造体のサイズを逸脱している.

【例 35】整数演算のオーバフロー

```
[1] current_priority(Pri,Min,Max), Rate := ((Pri-Min)*4096)/(Max-Min).
 Exception>> integer_overflow
   Builtin > multiply(-289851296,4096,E)
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行された組込述語 multiply/3 の演算でオーバフローが発生した.

【例 36】ゼロによる除算

```
[1] X := 512/0.
 Exception>> integer_zero_division
   Builtin > divide(512,0,B)
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行された組込述語 divide/3 の演算でゼロによる除算が行われた.

【例 37】浮動小数点数の無効演算

```
[1] X $:= 0.0/0.0.
 Exception>> invalid_floating_point_operation
   Builtin > floating_point_divide( +0.0 , +0.0 ,B)
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行された組込述語 floating_point_divide/3 は無効演算である.

【例 38】高階述語の引数個数不整合

```
[1] predicate_to_code(module#comparator,sort,5,Code),
   apply(Code,{"abc",abc, Small, Large}).
 Exception>> arity_mismatch
   Builtin > apply( code$comparator:sort/5$435 ,{"abc",abc,D,E})
    Caller > pimos:: listener_cmd_builtin: do_command/ 1
      Node > 0
```

ノード 0 番において pimos::listener_cmd_builtin:do_command/1 で実行された組込述語 apply/2 は comparator:sort/5 に与えている引数の個数が不整合である.

【例 39】エラーメッセージを検出した

```
[1] module_table:cache_table([get_atom_name(abc,Status)]).
 Error>> module_table_error
  Info > Illegal Message
  Data > get_atom_name(abc,Status)
  Node > 0
```

ノード 0 番において PIMOS に対する不正なメッセージを検出した.

【参考】ユーザが, PIMOS が管理しているデバイスやプールなどのユーティリティに対して, 不正なメッセージ (デバイスが知らないメッセージや, 引数のタイプが不正であるようなメッセージ) を流した場合, PIMOS はユーザに対してエラーの報告を行う. 詳しくは, プログラミング編 第 11.5 節 p.428 を参照のこと.

4.2 インスペクタ

インスペクタは, KL1 プログラムをデバッグするためのツールのひとつである. 任意の KL1 データの内容を, ユーザの指示により調査して表示する機能を持つ. リスナのトップレベルまたはトレース中に, 調査対象データを指定して呼び出すことができる. さらにシェルからも起動することが可能である. この場合, シェルが指定するデータが調査対象データとなる (操作編 2.1.6節 シェルの「エラー処理」(27頁) 参照).

4.2.1 機能概要

インスペクタの主な機能は以下の通りである.

データの調査 / 表示: ユーザの指示に従って調査対象データを解析し, 表示する. 任意の KL1 データを解析調査することができる. データの解析 / 表示位置の指定は, それぞれのデータタイプに従い, その要素番号や car, cdr, tail, '..' といったパラメタを用いる. また, 複数のパラメタを '>' で区切って指定することもできる.

データの監視 / 表示: ユーザの支持に従ってデータを監視し, 具体化を待って表示する. 主に, 実行に伴う未定義変数の具体化の状況を知りたい場合に用いる. 監視位置の指定は, それぞれのデータタイプに従い, その要素番号や car, cdr, tail, '..' といったパラメタを用いる. また, 複数のパラメタを '>' で区切って指定することもできる.

変数名と変数プール: 調査途中に出現したデータには変数名を付けることができる. それらの変数名は変数プールに登録され, 以降, その変数名でデータを参照することができる.

インスペクタは, 自身が独自に持つローカル変数プールと, その起動元のリスナの変数プールの 2 つに対して参照 / 登録を行うことができる.

出力の中断: ユーザは, インスペクタの出力を途中で中断することができる. その後ユーザの指示により, 出力を再開または中止する.

インスペクタのネストした呼び出し: インスペクタはネストして呼び出すことができる. 起動 / 終了の際にネストのレベルの深さが表示される.

入出力先の切り換え: インスペクタは起動されると, その起動元プロセスの持っていた入出力権を与えられて入出力を行う. 通常, 入出力権はインスペクタの終了時に起動元プロセスに返されるが, 新しいウインドウを生成してそのウインドウにインスペクタの入出力先を切り換えることにより, 入出力権を起動元プロセスに返すこともできる. 以降, 切り換えられたインスペクタと起動元プロセスは独立となり, 並列に動作する.

4.2.2 操作方法

4.2.2.1 起動

リスナのトップレベルからコマンドによって起動するか, またはトレース中にトレースコマンドによって起動する. 起動時に渡される調査対象データは, それがトレース中に起動されたものであれば, ノード番号とパッケージ名及びモジュール名を削ったそのゴール全体である. リスナのトップレベルから起動された場合は, ユーザの指定した任意の KL1 データである.

シェルからシェルのコマンドによって起動することも可能である. この場合, 調査対象データはシェルが指定する. 例えばシェル上でユーザゴール実行中に何らかのエラーが生じた場合, シェルからインスペクタを起動して, シェルが出力するエラーメッセージの内容を調査することができる (操作編 2.1.6節 シェルの「エラー処理」(27頁) 参照).

起動時に渡された調査対象データは, 探索の根として, ROOT という変数名でインスペクタのローカル変数プールに登録される.

【例 1】リスナのトップレベルからのインスペクタの起動

```
[1] inspect a.
a>
```

【例 2】トレース中のインスペクタの起動

```
{Trace}
[1] layered:n_queens(8,Out).
@0@4096 0     layered: n_queens(8,Out)
1     * (1)     queens(1,8,begin,Out)? ins
n_queens(8,Out)>
```

4.2.2.2 コマンドの入力

インスペクタが起動されるとプロンプトが表示され, ユーザのコマンド入力待ちとなる. プロンプトには, その時点で保持しているデータの内容が簡易表示される. プロンプトの後にコマンドを入力する.

【例】保持しているデータが {0,1,2,3,4,5,6,7,8,9} の時のプロンプトの表示

```
{0,1,2,3, ... }>
```

4.2.2.3 変数プール

調査途中に出現したデータには変数名を付けることができる. それらの変数名は変数プールに登録され, 以降, その変数名でデータを参照することができる. インスペクタは, 自身が独自に持つローカル変数プールと, その起動元のリスナの変数プールの 2 つに対して参照 / 登録を行うことができる. 通常は, インスペクタのローカル変数プールを用いる. ユーザが, 特にリスナの変数プールに対して変数を参照 / 登録したい場合は, その変数名の先頭に “_” をつけて入力する. それによって, リスナ変数プールには “_” を取り除いた変数名で参照 / 登録される.

ただし, データに変数名を付ける際, すでに登録済みの変数名を用いることはできない. この意味でユニフィケーションとは異なる. また, その時点で, 起動元のリスナで実行中のゴールがある場合は, そのゴールの呼び出し時に指定された変数名も用いることはできない.

インスペクタのローカル変数プールに登録された変数は, ROOT を除き, “.”, “cr” (ピリオドリターン) を入力することにより, 解放することができる.

4.2.2.4 ウインドウへの表示に関するパラメタ

インスペクタは, 以下の 6 つのパラメタの値にしたがってデータを表示する:

- 構造体の深さに関する出力制限値
- 構造体の長さに関する出力制限値
- 文字列の長さに関する出力制限値
- アトム印字名の長さに関する出力制限値
- 文字列の出力形態に関するパラメタ
- データ一般の出力形態に関するパラメタ

このうち，最初の 4 つは構造体のホロフラスティング出力に関するパラメタで，リスナと共通である． それぞれのコマンド **print_depth**(175頁参照)， **print_length**(176頁参照)， **print_string_length**(176頁参照), **print_atom_length**(176頁参照) を用いて参照したり設定したりすることができる．インスペクタを起動したリスナウインドウの値が，インスペクタ起動時の初期値となる．

文字列の出力形態に関するパラメタは，文字列の出力形態を文字印字形式またはコード形式のいずれにするかを指定するもので，インスペクタ内のみで有効である． **print_string_mode** コマンド (176頁参照) により参照 / 変更する．インスペクタ起動時の初期値は，文字印字形式である．

データ一般の出力形態に関するパラメタは，KL1 のデータ型にしたがって表示する一般形式か，或いはデータのタグ部と値部を表示する形式のいずれにするかを指定するもので，インスペクタ内のみで有効である． **print_mode** コマンド (177頁参照) により参照 / 変更する．インスペクタ起動時の初期値は，一般形式である．

【注意 1】なお，これらのパラメタの値をネストして呼び出されたプロセス内で変更した場合，そのプロセスの終了時にパラメタが元の値に戻ることはない．

【注意 2】本来ユーザが記述したプログラム上では MRB が白であっても，リスナからのゴールの実行中にトレーサ，インスペクタを用いることによって，データの MRB が黒となってしまう場合がある．従って，データ一般の出力形態に関するパラメタをタグ部と値部を表示する形式に設定してゴール中のデータを表示しても，そのデータの MRB の白黒を正確に判定することはできない．

4.2.2.5 入出力先の切り換え

fork コマンド (167頁参照) を実行することにより，新しいウインドウを生成してそのウインドウにインスペクタの入出力先を切り換えることができる．ウインドウへの表示に関するパラメタの初期値は，元のウインドウと同じである．入出力先の切り替えにより，元のウインドウは再び起動元プロセスのコマンド入力待ちとなる．以降，切り換えられたインスペクタと起動元プロセスは独立となり，並列に動作する．

インスペクタウインドウ上で動作しているインスペクタが終了すると，その後は，生き残っている監視プロセスの入出力のみが適宜行われる．入出力が行われていない状態の時には，`^C` を入力することにより，このウインドウ上で起動される監視プロセスに対するストリーム自動監視モード及び質問モードのデフォルト値の変更，このウインドウ上で起動されている監視プロセスの一覧表示，ヘルプメッセージの出力，インスペクタウインドウの終了を行うことができる．

【参考 1】監視プロセスについて，詳しくは 4.2.5.1節「データの監視と表示」(178頁) 参照．

【参考 2】将来，インスペクタもネストして新たに呼び出すことができるようになる予定である．

【例 1】インスペクタウインドウ上でインスペクタを終了させた後，インスペクタウインドウを閉じる

インスペクタウインドウ:

```
a> exit

 ?? Interruption (h for help): ex
```

【例 2】入出力が行われていないインスペクタウインドウに対する操作例 (インスペクタから監視プロセスを起動した後，インスペクタを終了させる．その後，インスペクタウインドウに対する操作のヘルプメッセージを出力し，存在する監視プロセスの一覧を表示)

```
get_device(A)> m 1 result
>monitor:1 result started
get_device(A)> ex

 ?? Interruption (h for help): h
     <cr> : continue
      sam : change stream auto mode
       qm : change query mode
       mt : show monitored terms list
       ex : kill all monitoring processes and exit
        h : help
(h for help) ? mt

 %%% MONITORED TERMS %%%
  aq: 1 result B == A ?
```

4.2.2.6 出力の中断

ユーザは, キーボードから ^C を入力することにより, インスペクタの出力を途中で中断することができる. その後, ユーザの指示により, 出力を再開または中止する. 出力を再開する場合はキーボードから改行コードまたは c を入力する. また t を入力することにより, 出力を中止し, インスペクタのコマンド入力レベルに復帰することができる.

【例】 1000 要素から成るベクタの 20 要素目から 100 個を出力しようとしたが, 途中で出力を中止した場合の実行例

```
{0,1,2,3, ... }> me 20,100
20 : 20
21 : 21
22 : 22
23 : 23
24 : 24
25 : 25
26 : 26

 ?? Interruption (h for help): h

   <cr>: continue
      t: return to inspector top level
      c: continue
      h: help

 ?? --------- (h for help): t
{0,1,2,3, ... }>
```

4.2.2.7 終了

exit コマンド (167頁参照) を入力する. ただし, 起動元のリスナが終了した場合は, インスペクタも同時に終了する.

4.2.2.8 インスペクタの環境変数

インスペクタの呼び出し時に以下の環境変数が設定されている場合, その値が採用される.

inspector:print_string_mode

文字列の出力形態を指定する. 値が code(アトム) の場合, 文字列の出力形態をコード形式にする. それ以外の場合は文字印字形式となる.

【例】環境変数による文字列の出力形態の指定

```
[1] env(inspector).                          % リスナの環境変数プールからインスペクタに
                                             % 関するものだけを取り出して表示する
[2] inspect(String).
"^@^@^@^@^@?? "> me
string : "^@^@^@^@^@^@^@^@^@^@"              % 環境変数が設定されていない場合は
"^@^@^@^@^@?? "> exit                        % 文字印字形式で表示される

                                             % 環境変数を設定すると ...
[3] setenv(inspector:print_string_mode,code,S).
 S = normal

[4] env(inspector).
 inspector: print_string_mode = code

[72] inspect(String).
"^@^@^@^@^@?? "> me
string : 16#{0,0,0,0,0,0,0,0,0,0}            % コード形式で表示される
"^@^@^@^@^@?? ">
```

inspector:initial_window_size

インスペクタウインドウの生成時の大きさを指定する. 値が char(Characters, Lines) で Characters, Lines が共に正整数の場合, ウインドウの大きさを Characters 文字, Lines 行で生成する. 値が mouse の場合, ウインドウの大きさ及び位置をマウスを用いて設定する. それ以外の場合は char(40,10) に設定される.

【注意】指定するサイズは, PIMOS が生成可能なウインドウの最小サイズより大きくなくてはならない.

inspector:window_size_increment

bigger コマンド実行時のインスペクタウインドウの大きさの変更の度合いを指定する. 値が char(Characters, Lines) で Characters, Lines が共に正整数の場合, ウインドウの大きさを Character 文字, Lines 行大きくする. 値が mouse の場合, ウインドウの大きさ及び位置をマウスを用いて設定する. それ以外の場合は char(10,5) に設定される.

inspector:window_size_increment_l

a_little_bigger コマンド実行時のインスペクタウインドウの大きさの変更の度合いを指定する. 値が char(Characters, Lines) で Characters, Lines が共に正整数の場合, ウインドウの大きさを Character 文字, Lines 行大きくする. 値が mouse の場合, ウインドウの大きさ及び位置をマウスを用いて設定する. それ以外の場合は char(5,2) に設定される.

4.2.3 データの表示形式と解析 / 表示 / 監視位置の指定方法

4.2.3.1 データの表示形式

以下にインスペクタが表示するデータの形式を示す.

整数:

integer：整数値

【例】整数の表示

```
1> me                                          % データを表示するコマンド（後述）
integer : 1
1>
```

浮動小数点数:

floating_point：浮動小数点数値

【例】浮動小数点数の表示

```
1.000000E+0> me                                % データを表示するコマンド（後述）
floating_point : 1.000000E+0
1.000000E+0>
```

アトム:

atom：アトム印字名

【例】アトムの表示

```
atom> me                                       % データを表示するコマンド（後述）
atom : atom
atom>
```

文字列:

string："文字の列 ", または,
string：N#{コード, ... } (ただし N は要素サイズを表す)

【例】文字列の表示

```
"abc"> me                                      % データを表示するコマンド（後述）
string : "abc"
"abc"> print_string_mode                       % 文字列の表示形式を変えるコマンド（後述）
 Print mode for string: character => code
"abc"> me
string : 16#{9057,9058,9059}
"abc">
```

リスト:

要素番号：要素の値, または,
car：リストの car の値
cdr：リストの cdr の値
tail：リストの tail の値

【注意】tail はリスト構造をたどって行った場合の最後を示す．すなわち，リスト [a,b,c] の tail は [] であり，[a|X] の tail はこの時点では X である．

【例】リストの表示

```
[a,b]> me                                   % データを表示するコマンド（後述）
0 : a
1 : b
tail : []
[a,b]> me car                               % リストの car を表示するコマンド（後述）
car : a
[a,b]> me cdr                               % リストの cdr を表示するコマンド（後述）
cdr : [b]
[a,b]>
```

ベクタ:

要素番号：要素の値

【例】ベクタの表示

```
getb(256,L)> me                             % データを表示するコマンド（後述）
0 : getb
1 : 256
2 : L
getb(256,L)>
```

モジュール:

module$ パッケージ名:: モジュール名 $ モジュールサイズ, または,
要素番号：要素の値

【例】モジュールの表示

```
 module$pimos::pool$231 > me ,4  % 構造体の 0 番目から 4 個分の要素を表示するコマンド（後述）
0 : 49
1 : pool
2 : pimos
3 : []
 module$pimos::pool$231 >
```

コード:

code：code$ パッケージ名:: モジュール名：コード名 / 引数個数 $ モジュールサイズ

【例】コードの表示

```
 code$pimos::pool:stack/1$231 > me          % データを表示するコマンド（後述）
code :  code$pimos::pool:stack/1$231
 code$pimos::pool:stack/1$231 >
```

未定義変数:

>Unbound

【例】未定義変数の表示

```
X> me                                       % データを表示するコマンド（後述）
>Unbound
X>
```

その他:

$Unknown/ データのタグ番号 / 値, または
(データのタグ名) 値

【例】インスペクタの知らないデータ型のデータの表示

```
 module$pimos::pool$231 > me 209          % 構造体の 209 要素目を表示するコマンド (後述)
209 :  $Unknown/36/16777426
 module$pimos::pool$231 > down 209        % 構造体の 209 要素目に潜るコマンド (後述)
unknown_data_type : (PREDH) 16777426
 $Unknown/36/16777426 >
```

なお, データ一般の出力形態パラメタがタグ部と値部の表示形式になっている場合は, すべてのデータが

$Unknown/ データのタグ番号 / 値, または
(データのタグ名) 値

の形式で表示される.

【例】タグ部と値部の表示形式によるデータの表示

```
 module$pimos::pool$231 > print_mode                    % データの表示形式を変えるコマンド (後述)
 Print mode: general => tag_and_value
 module$pimos::pool$231 > me ,8           % 構造体の 0 番目から 8 個分の要素を表示するコマンド (後述)
0 : (INTo) 49
1 : (ATOMo) 8388749
2 : (ATOMo) 8388608
3 : (ATOMo) 0
4 : (VECTx) -1073735427
5 : (INTo) 0
6 : (MOD) -1073739797
7 : (ATOMo) 25165966
 module$pimos::pool$231 >
```

4.2.3.2 解析 / 表示 / 監視位置の指定方法

複雑な構造体データに対して, ユーザが必要とする部分のみを解析 / 表示するための, 解析 / 表示位置の指定方法を示す. データの具体化を監視する場合も同様の方法で監視位置を指定する.

データの一段深い部分を指定するには, それぞれのデータ型にしたがって, 以下のようにパラメタを指定する.

文字列: 文字列の要素番号を指定する.

【例】文字列の 5 要素目を表示して潜る

```
[1] inspect "abcdefghijklmnopqrstuvwxyz".
"abcde?? "> me 5
string : " ... f ... "
"abcde?? "> down 5
integer : 9062
9062>
```

リスト:

リストの先頭からの要素番号を指定するか, または,
パラメタ **car**, **cdr**, **tail** を指定する.

【注意】リストは, 構造上は car と cdr から成る構造をしているが, インスペクタでは通常, リスト内の各要素に注目して, その先頭からの要素番号で指定する. ただし, 特に car, cdr の内容を解析 / 表示したい場合には, パラメタとしてアトムの **car**, **cdr** を指定すればよい. また, **tail** というパラメタを指定すれば, リストの最後を解析したり表示したりすることができる.

【例】リストの 5 要素目, car, cdr, tail を表示してから 5 要素目に潜る

```
[1] inspect [a,b,c,d,e,f,g].
[a,b,c,d, ... ]> me 5                    % リストの 5 要素目を表示する
5 : f
[a,b,c,d, ... ]> me car                  % リストの car を表示する
car : a
[a,b,c,d, ... ]> me cdr                  % リストの cdr を表示する
cdr : [b,c,d,e,f,g]
[a,b,c,d, ... ]> me tail                 % リストの tail を表示する
tail : []
[a,b,c,d, ... ]> down 5
atom : f
f>
```

ベクタ: ベクタの要素番号を指定する.

【例】ベクタの 5 要素目を表示して潜る

```
[1] inspect {a,b,c,d,e,f,g}.
a(b,c,d,e, ... )> me 5
5 : f
a(b,c,d,e, ... )> down 5
atom : f
f>
```

モジュール: モジュールの要素番号を指定する.

【例】モジュールの 209 要素目を表示して潜る

```
[1] inspect module#pool.
 module$pimos::pool$231 > me 209
209 :  $Unknown/36/16777426
 module$pimos::pool$231 > down 209
unknown_data_type : (PREDH) 16777426
 $Unknown/36/16777426 >
```

未定義変数: その時点では, それ以上深く解析することはできない.

【例】

```
X> me
>Unbound
X> down 0
>Unbound
X>
```

整数, 浮動小数点数, アトム, コード, その他: それ以上深く解析することはできない.

【例】

```
atom> me 1
>Illegal parameter: 1
atom> down 1
>Cannot go down: atom
atom>
```

データの一段浅い部分を指定するには, パラメタ '..' を指定する. ただし現在の調査対象データが探索の根である場合, それ以上浅い部分を指定することはできない.

データの N 段深い (浅い) 部分を指定するには, 以上のパラメタの列を '>' で区切って指定する.

4.2.4 コマンド

ユーザの入力するコマンドは, 第 1 引数にコマンド名, 第 2 引数以降に必要なパラメタをカンマまたは空白で区切って設定する形をしている. 各コマンド名には省略型があり, それを入力することによっても同様の操作を行うことができる.

第 2 引数以降のパラメタを適宜省略した形式のコマンドもある. その場合は, それぞれのコマンドによって適当なデフォルト値が用いられる.

なお, いくつかのコマンドの説明の後には例題を掲載した. 特に指定のない場合, それらは構造体の深さに関する出力制限値を 5, 構造体の長さに関する出力制限値を 4, 文字列の出力形態を文字印字形式として実行したものである.

4.2.4.1 基本コマンド

help Type [h, he]

指定されたタイプ Type (アトム) のコマンド群の一覧を表示する. タイプには次のものがある.

basic: 基本コマンドの一覧を表示する
insp: 調査コマンドの一覧を表示する
monitor: 監視コマンドの一覧を表示する
env: 環境コマンドの一覧を表示する
varpool: 変数プール操作コマンドの一覧を表示する
all: 全コマンドの一覧を表示する

【参考】これらのタイプはパラメタを伴わない help コマンド (help/0) で表示することができる.

help [h, he]

タイプ指定をする help コマンド (help/1) で指定できるコマンド群のタイプの一覧を表示する.

commands Type [com]

指定されたタイプ Type (アトム) のコマンド群のコマンド名一覧を表示する. タイプには次のものがある.

basic: 基本コマンドのコマンド名一覧を表示する
insp: 調査コマンドのコマンド名一覧を表示する
monitor: 監視コマンドのコマンド名一覧を表示する
env: 環境コマンドのコマンド名一覧を表示する
varpool: 変数プール操作コマンド名一覧を表示する
all: 全コマンドのコマンド名一覧を表示する

【参考】これらのタイプはパラメタを伴わない commands コマンド (commands/0) で表示することができる.

【参考】各コマンドの簡易説明は help コマンドを用いて表示することができる.

commands [com]

タイプ指定をする commands コマンド (commands/1) で指定できるコマンド群のタイプの一覧を表示する.

exit [ex]

インスペクタを終了し, ウインドウの入出力権を起動元プロセスに返す.

fork [f]

新しいウインドウを生成し, インスペクタの入出力先をそのウインドウに切り換える. 元のウインドウへの入出力権は起動元プロセスに返す.

inspect ?Term [i]

現在起動されているインスペクタの実行を中断し, 新しいインスペクタを起動する. Term は新しいインスペクタの調査対象データである.

【例】インスペクタのネストした呼び出し

```
[put(a,A),put(b,B),put(a,C),put(c,D), ... ]> inspect _Table
>Inspector break level: 1
{O,c(A,b(B,O)),a(C,a(D, & )),O}>
                ......
{O,c(A,b(B,O)),a(C,a(D, & )),O}> exit
>Inspector break level: 0
[put(a,A),put(b,B),put(a,C),put(c,D), ... ]>
```

set_title String

インスペクタウインドウのタイトルを, 文字列 `INSPECTOR:` の後に String (文字列) をアペンドしたものとする.

【注意】インスペクタウインドウ以外のウインドウ上で動作するインスペクタに対して実行した場合, 何もしない.

【例】インスペクタウインドウで以下の操作を実行した場合, ウインドウのタイトルは, `INSPECTOR: generator process` となる.

```
process(A,B)> set_title "generator process"
process(A,B)>
```

bigger [big]

インスペクタウインドウの大きさを 10 文字, 5 行大きくする. 大きさの度合いは環境変数 **inspector:window_size_increment** で設定することも可能である (4.2.2.8節「インスペクタの環境変数」 161頁参照). ただし, ウインドウの位置制限により指定分だけ大きくすることが不可能な場合は, 可能な範囲で大きさを変更する.

【注意】インスペクタウインドウ以外のウインドウ上で動作するインスペクタに対して実行した場合, 何もしない.

a_little_bigger [lbig]

インスペクタウインドウの大きさを 5 文字, 2 行大きくする. 大きさの度合いは環境変数 **inspector:window_size_increment_l** で設定することも可能である (4.2.2.8節「インスペクタの環境変数」 161頁参照). ただし, ウインドウの位置制限により指定分だけ大きくすることが不可能な場合は, 可能な範囲で大きさを変更する.

【注意】インスペクタウインドウ以外のウインドウ上で動作するインスペクタに対して実行した場合, 何もしない.

4.2.4.2 調査コマンド

me M, N

現在保持しているデータの M 要素目から N 個を表示する.

表示位置 M の指定が不正の場合はエラー, 表示可能な要素数が N に満たない場合は可能な分のみを表示する. 表示位置 M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」 (164頁) 参照.

N には負の整数も指定することができる. その場合, M が指す要素 E を含む一段上の構造体内における E の具体的位置 M0 に対して, M0+N+1 (ただし M0+N+1 < 0 の場合は 0) 要素目から –N 個表示することを意味する.

パラメタ M, N は省略することができる. 省略時には既定値としてそれぞれ 0, 構造体の長さに関する出力制限値が用いられる.

【参考】**me ,** はパラメタを伴わない **me** (後述) と同様.

me M

現在保持しているデータの M 要素目を表示する. 表示位置 M の指定が不正の場合はエラーとなる. 表示位置 M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」 (164頁) 参照.

me

現在保持しているデータの 0 番目の要素から, 構造体の長さに関する出力制限値の個数分を表示する.

表示可能な要素数が指定された個数に満たない場合は可能な分のみを表示する.

【例 1】 100 要素のベクタ `{0,1,2,3,4,...,99}` の内容の表示

```
{0,1,2,3, ... }> me
0 : 0
1 : 1
2 : 2
3 : 3
4 : 4
{0,1,2,3, ... }> me 50
50 : 50
{0,1,2,3, ... }> me 50,2
50 : 50
51 : 51
{0,1,2,3, ... }> me 9,-3
7 : 7
8 : 8
9 : 9
{0,1,2,3, ... }> me 80,
80 : 80
81 : 81
82 : 82
83 : 83
84 : 84
{0,1,2,3, ... }> me ,2
0 : 0
1 : 1
{0,1,2,3, ... }>
```

【例 2】リスト [0,1,2,3,4,5,6,7,8,9|A] の内容の表示 (A は変数とする)

```
[0,1,2,3, ... ]> me
0 : 0
1 : 1
2 : 2
3 : 3
4 : 4
[0,1,2,3, ... ]> me 5
5 : 5
[0,1,2,3, ... ]> me 8,10
8 : 8
9 : 9
tail : A
[0,1,2,3, ... ]> me 9,-3
7 : 7
8 : 8
9 : 9
[0,1,2,3, ... ]> me 10,-3
8 : 8
9 : 9
tail : A
[0,1,2,3, ... ]> me car
car : 0
[0,1,2,3, ... ]> me cdr
cdr : [1,2,3,4,5, ... ]
[0,1,2,3, ... ]> me tail
tail : A
[0,1,2,3, ... ]>
```

【例 3】ストリング “0123456789” の内容の表示

```
"01234?? "> me
string : "01234 ... "
"01234?? "> me 5
string : " ... 5 ... "
"01234?? "> me 8,10
string : " ... 89"
"01234?? "> me 9,-3
string : " ... 789"
"01234?? ">
```

【例 4】ベクタ {x(y(z(1,2,3))), s(t(u(4,5,6)))} の内容の表示

```
{x(y( & )),s(t( & ))}> me 0>1>1
1 : z(1,2,3)
{x(y( & )),s(t( & ))}> me 0>1>1>0,10
0 : z
1 : 1
2 : 2
3 : 3
{x(y( & )),s(t( & ))}>
```

down M　[空]

指定された位置 M までデータを深く潜る. 位置 M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】コマンドを省略し, パラメタ M のみを入力することができる.

【例】データ {{{{{{0},1},2},3},4},{5,{6,{7,{8,{9}}}}}} に対する down 操作

```
{{{ & ,3},4},{5,{6, &}}}> me
0 : {{{{ & ,1},2},3},4}
1 : {5,{6,{7,{8, & }}}}
{{{ & ,3},4},{5,{6, &}}}> 0
0 : {{{ & ,1},2},3}
1 : 4
{{{ & ,2},3},4}> 0>0>0>0>0
integer : 0
0> ..>..>..>..>..>..>..>1>0
integer : 5
5>
```

up N　[u]

データを N 段上がる.

up　[u]

データを 1 段上がる.

【参考】down .. を実行するのと同じである.

【例】データ {{{{{{0},1},2},3},4},{5,{6,{7,{8,{9}}}}}} に対する down と up 操作

```
{{{ & ,3},4},{5,{6, &}}}> me
0 : {{{{ & ,1},2},3},4}
1 : {5,{6,{7,{8, & }}}}
{{{ & ,3},4},{5,{6, &}}}> 0
0 : {{{ & ,1},2},3}
1 : 4
{{{ & ,2},3},4}> 0>0>0>0>0
integer : 0
0> up 99
>Root
{{{ & ,3},4},{5,{6, & }}}> 1>0
integer : 5
5>
```

length [len]

データの要素数を表示する.

データが文字列の場合は, その長さと要素サイズを表示する. リストで cdr 部がまだ具体化されていない場合は, わかっている分について表示する (【例 3】参照).

【例 1】ベクタの要素数を調べる

```
{0,1,2,3, ... }> length
 100
{0,1,2,3, ... }>
```

【例 2】文字列のサイズを調べる

```
"01234?? "> length
 Length of the string is 82,
 element size of the string is 16
"01234?? ">
```

【例 3】リスト `[0,1,2,3,4|A]` の要素数を調べる (`A` は変数)

```
[0,1,2,3, ... ]> length
>The tail of the list is variable: A,
 length of the list is greater or equal to 5
[0,1,2,3, ... ]>
```

show ?Term [sh]

指定されたデータ Term の値を簡易表示する.

history N

データを潜った経歴を過去 N 回に渡って表示する. down コマンドを用いて潜った位置と各時点でのデータの一覧を表示する.

history

データを潜った経歴を過去数回に渡って表示する. down コマンドを用いて潜った位置と各時点でのデータの一覧を表示する. 回数は, 構造体の長さに関する出力制限値に従う.

【例】データ `{0,{1,{2}}}` を数回潜った場合の経歴の表示

```
2> history
 0 root  :  {0,{1,{2}}}
 1 down 1 :  {1,{2}}
 2 down 1 :  {2}
 3 down 0 :  2
2>
```

push_root ?Term　[pu]

指定されたデータ Term をこれ以降の調査対象とする. インスペクタのローカル変数プールに登録されていた変数 ROOT の内容は, 新しい調査対象データに置き換えられる.

【例】

```
[0,1,2,3, ... ]> pu a
atom : a
a>
```

4.2.4.3　監視コマンド

monitor M Name　[m]

現在保持しているデータの M 要素目に対して監視プロセスを起動する. 監視データに対する覚え書きとして Name に空白を含まない任意の文字の並びを指定することができる. M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

【例】インスペクタからの監視プロセスの起動

```
process(A,B)> m 1 input_stream
>monitor:1 input_stream started
process(A,B)>
```

monitor M　[m]

現在保持しているデータの M 要素目に対して監視プロセスを起動する. M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

monitor_and_fork M Name　[mf]

現在保持しているデータの M 要素目に対して監視プロセスを起動する. 同時に新しいインスペクタウインドウを生成してその監視プロセスの入出力先を新しいウインドウに切り換える. 監視データに対する覚え書きとして Name に空白を含まない任意の文字の並びを指定することができる. 新しいウインドウのタイトルに, この監視プロセスの識別番号と Name が表示される. M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

【例】インスペクタからの起動例

```
process(A,B)> mf 1 input_stream
process(A,B)>
```

新しいウインドウに監視プロセスのメッセージが表示される:

```
>monitor:1 input_stream started
```

monitor_and_fork M [mf]

現在保持しているデータの M 要素目に対して監視プロセスを起動する．同時に新しいインスペクタウインドウを生成してその監視プロセスの入出力先を新しいウインドウに切り換える．新しいウインドウのタイトルに，この監視プロセスの識別番号が表示される．M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

conditional_monitor M Term Name [cm]

現在保持しているデータの M 要素目に対して監視プロセスを起動する．ただし監視プロセスは，監視対象となるデータが Term に指定された項と照合する値に具体化した場合のみ報告する．項 Term に変数が含まれる場合, 変数部分についての照合は一切行わない．監視データに対する覚え書きとして Name に空白を含まない任意の文字の並びを指定することができる．M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

【例】インスペクタからの条件付きの監視プロセスの起動

```
process(A,B)> cm 1 [get(_)|_] input_stream
>monitor:1 input_stream started
process(A,B)>
```

conditional_monitor M Term [cm]

現在保持しているデータの M 要素目に対して監視プロセスを起動する．監視プロセスは，監視対象となるデータが Term に指定された項と照合する値に具体化した場合のみ報告する．項 Term に変数が含まれる場合, 変数部分についての照合は一切行わない．M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

conditional_monitor_and_fork M Term Name [cmf]

現在保持しているデータの M 要素目に対して監視プロセスを起動する．同時に新しいインスペクタウインドウを生成してその監視プロセスの入出力先を新しいウインドウに切り換える．ただし監視プロセスは，監視対象となるデータが Term に指定された項と照合する値に具体化した場合のみ報告する．項 Term に変数が含まれる場合, 変数部分についての照合は一切行わない．監視データに対する覚え書きとして Name に空白を含まない任意の文字の並びを指定することができる．新しいウインドウのタイトルに，この監視プロセスの識別番号と Name 及び Term が表示される．M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

【例】インスペクタからの起動例

```
process(A,B)> cmf 1 [get(_)|_] input_stream
process(A,B)>
```

新しいウインドウに監視プロセスのメッセージが表示される:

```
>monitor:1 input_stream started
```

conditional_monitor_and_fork M Term [cmf]

現在保持しているデータの M 要素目に対して監視プロセスを起動する. 同時に新しいインスペクタウインドウを生成してその監視プロセスの入出力先を新しいウインドウに切り換える. ただし監視プロセスは, 監視対象となるデータが Term に指定された項と照合する値に具体化した場合のみ報告する. 項 Term に変数が含まれる場合, 変数部分についての照合は一切行わない. 新しいウインドウのタイトルに, この監視プロセスの識別番号と Name 及び Term が表示される. M の指定方法は 4.2.3.2節「解析 / 表示 / 監視位置の指定方法」(164頁) 参照.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

stream_auto_mode [sam]

監視プロセスを起動する際にデフォルトで設定されるストリーム自動監視モードの値をトグルで変更する.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁), ストリーム自動監視モードについて詳しくは, 180頁参照.

【例】インスペクタからの操作例

```
process(A,B)> sam
 Monitor: stream_auto_mode = off
process(A,B)>
```

query_mode [qm]

監視プロセスを起動する際にデフォルトで設定される質問モードの値をトグルで変更する.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁), 質問モードについて詳しくは, 180頁参照.

【例】インスペクタからの操作例

```
process(A,B)> qm
 Monitor: query_mode = off
process(A,B)>
```

silent_mode Param [sm]

監視プロセスの表示時の beep 音を鳴らすモードを変更する. Param として以下の値を指定することができる.

all_on: すべての監視プロセスの beep 音を鳴らさない

all_off: すべての監視プロセスの beep 音を鳴らす

window_on: 実行されたウインドウ上で入出力を行うすべての監視プロセスの beep 音を鳴らさない

window_off: 実行されたウインドウ上で入出力を行うすべての監視プロセスの beep 音を鳴らす

【注意 1】すべての監視プロセスとは, リスナから実行中の 1 ゴールに対して起動されている監視プロセス全体を指す. 既に実行が終了したゴールに対して生き残っているものは管理下に含まれない.

【注意 2】将来, 監視プロセス毎にモードを変更することが可能となる予定である. それに伴い, 現在の Param の指定方法は変更される可能性がある.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁) 参照.

monitored_terms [mt]

現在起動されている監視プロセスの一覧を表示する.

【参考】データの監視について詳しくは, 4.2.5.1節「データの監視と表示」(178頁), 監視プロセスの集合に対する操作について詳しくは, 「生成された監視プロセス集合に対する操作」(181頁) 参照.

【例】インスペクタからの操作例

```
process(A,B)> m 1 input_stream
>monitor:1 input_stream monitored
process(A,B)> cm 1 [get(_)|_] input_stream2
>monitor:2 input_stream2 monitored
process(A,B)> mt

 %%% MONITORED TERMS %%%
  aq:1 input_stream C == A
  aq:2 input_stream2 [get(D)|E] == A ?
```

4.2.4.4 環境コマンド

print_depth D [pd]

構造体の深さに関する出力制限値を D に設定する.

【例】

```
p({{ & ,3},4})> me
0 : p
1 : {{ & ,3},4}
p({{ & ,3},4})> pd 10
p({{ & ,3},4})> me
0 : p
1 : {{{{{0},1},2},3},4}
p({{ & ,3},4})>
```

print_depth [pd]

構造体の長さに関する出力制限値を参照 / 設定する.

現在の値を表示した後, ユーザの入力した新しい値に変更する. 改行コードのみが入力された場合, 値の変更は行わない.

【例】

```
p({{ & ,3},4})> me
0 : p
1 : {{ & ,3},4}
p({{ & ,3},4})> pd
 Print depth: 2 => 10
p({{ & ,3},4})> me
0 : p
1 : {{{{0},1},2},3},4}
p({{ & ,3},4})>
```

print_length L　[pl]

構造体の長さに関する出力制限値を L に設定する.

print_length　[pl]

構造体の長さに関する出力制限値を参照 / 設定する.

現在の値を表示した後, ユーザの入力した新しい値に変更する. 改行コードのみが入力された場合, 値の変更は行わない.

print_string_length L　[psl]

文字列の長さに関する出力制限値を L に設定する.

print_string_length　[psl]

文字列の長さに関する出力制限値を参照 / 設定する.

現在の値を表示した後, ユーザの入力した新しい値に変更する. 改行コードのみが入力された場合, 値の変更は行わない.

print_atom_length L　[pal]

アトム印字名の長さに関する出力制限値を L に設定する.

print_atom_length　[pal]

アトム印字名の長さに関する出力制限値を参照 / 設定する.

現在の値を表示した後, ユーザの入力した新しい値に変更する. 改行コードのみが入力された場合, 値の変更は行わない.

print_string_mode　[psm]

文字列の出力形態を変更する. 出力形態には文字印字形式 (character) とコード形式 (code) の 2 種類があり, トグルで変更される. インスペクタ起動時の初期値は文字印字形式である.

【例】文字列の出力形態を変更した場合の出力例

```
"ab"> me
string : "ab"
"ab"> psm
 Print mode for string: character => code
"ab"> me
string : 16#{9057,9058}
"ab"> psm
 Print mode for string: code => character
"ab">
```

print_mode [pm]

データの出力形態を変更する． 出力形態には一般形式 (general) とタグ部と値部を表示する形式 (tag_and_value) の 2 種類があり，トグルで変更される．インスペクタ起動時の初期値は一般形式である．

【注意】本来ユーザが記述したプログラム上では MRB が白であっても，リスナからのゴールの実行中にトレーサ，インスペクタを用いることによって，データの MRB が黒となってしまう場合がある．従って，このモードを用いてゴール中のデータの MRB の白黒を正確に判定することはできない．

environment [env]

インスペクタの表示用パラメタの値を表示する．

【例】出力例

```
a> environment
 Print depth           : 4
 Print length          : 10
 Print string length   : 20
 Print atom length     : 30
 Print mode for string : character
 Print mode            : general
a>
```

4.2.4.5 変数プール操作コマンド

name ^VarName [n]

現在保持しているデータを指定された変数に代入する．変数名として最初に“_”をつけたものを指定すると，リスナ変数プールに登録される．それ以外の名前の場合は，インスペクタのローカル変数プールに登録される．

【注意】すでに登録されている変数名を用いることはできない．この意味で，ユニフィケーションとは異なる．

list Type [ls]

インスペクタのローカル変数プール及びリスナ変数プールに登録されている変数の変数名とデータの一覧を表示する．リスナ変数プールに登録されている変数の変数名は，最初に“_”を加えたもので表示される．Type として以下のものを指定することができる．

local: インスペクタのローカル変数プール内のみ表示

listener: リスナ変数プール内のみ表示

all: インスペクタのローカル変数プール内とリスナ変数プール内の両方を表示

list [ls]

インスペクタのローカル変数プール及びリスナ変数プールに登録されている変数の変数名とデータの一覧を表示する.

リスナ変数プールに登録されている変数の変数名は, 最初に "_" を加えたもので表示される.

ピリオドリターン

".", "cr" を入力することにより, インスペクタのローカル変数プールに登録されている ROOT を除く全ての変数を解放する.

【注意】リスナの変数プール内の変数を解放するには, リスナのトップレベルで "ピリオドリターン" を実行しなければならない.

【例】変数プール操作コマンドの実行例

1. **name** コマンドにより現在のデータ a を変数 A に代入する.
2. **list** コマンドにより変数の一覧を表示する. 変数 A, T はユーザが定義したインスペクタローカル変数, ROOT はインスペクタ起動時に渡されたデータを表す変数, _L, _Table はリスナの変数プールに登録されている変数である.
3. '.' コマンドの実行によって, ユーザが定義したインスペクタローカル変数はすべて解放される.

```
a> name A                                                    % (1)
a> list                                                      % (2)
 A = a
 ROOT = a(b,c,A)
 T = {{"b",A,0},{"a",B,{"e",C,0}},{"d",D,0},{"c",E,0}}
 _L = [put(a,A),put(b,B),put(a,C),put(c,D),put(a,E), ... ]
 _Table = {0,c(A,b(B,0)),a(C,a(D,a(E, & ))),0}
a> .                                                         % (3)
 ROOT = a(b,c,A)
 _L = [put(a,A),put(b,B),put(a,C),put(c,D),put(a,E), ... ]
 _Table = {0,c(A,b(B,0)),a(C,a(D,a(E, & ))),0}
a>
```

4.2.5 データの監視と表示

4.2.5.1 概要

データを調査したものの, その時点ではまだ未定義の場合もある. このような未定義変数に対して, 具体化を監視して表示するのがデータの監視 (モニタ)/ 表示機能である. ユーザは, データのある部分を調査する場合と同様にして, 監視する位置を指定する.

データの監視を指示すると, そのデータの具体化を監視するプロセスが生成される. 以降, このプロセスを**監視プロセス**と呼ぶ. 監視プロセスは, 監視しているデータが深さ 1 のレベルで具体化すると, 報告の条件として指定された項と照合するかどうか検査し, 照合する場合は監視プロセスが起動されたウインドウにデータの内容を報告する. 報告と共にプロンプトが表示され, 一部を除くインスペクタのコマンドを実行することができる. 照合しない場合及び, 照合するかどうか決定できない場合は報告しない.

基本的には, 監視プロセスは深さ 1 のレベルでのデータの具体化のみを監視するが, データがリストに具体化した場合, その cdr 部に対してデフォルトで引き続き監視を継続するモードも用意されている. このモードは**ストリーム自動監視モード**と呼ばれる.

またデフォルトでは, 監視プロセスは, 具体化の表示の後ユーザのコマンドを受け付ける態勢となるが, 表示のみを行うことも可能である. そのためには**質問モード**と呼ばれるモードを用いる.

生成された監視プロセスに対して, 一覧の表示, 強制終了, ストリーム自動監視モード及び質問モードの変更を行うことができる.

また, 監視プロセスは, 監視の開始時と終了時にメッセージを出力する.

4.2.5.2 操作方法

起動

インスペクタまたはリスナからコマンドによって起動する. インスペクタからの起動コマンドには, **monitor**(172頁参照), 別ウインドウに報告する **monitor_and_fork**(172頁参照), 指定した条件を満たす値に具体化した時のみ報告する **conditional_monitor**(173頁参照), 同様に別ウインドウに報告する **conditional_monitor_and_fork** (173頁参照) がある. リスナからの監視プロセスの起動方法については, 操作編 4.1.6.8節 リスナの「デバッグコマンド」(107頁), または, トレース中に起動する方法については, 4.1.7.9節リスナの「トレースコマンド」(126頁) 参照.

監視プロセスが起動されると指定された出力先ウインドウに以下のようなメッセージを出力する.

```
>monitor:1 stream started
```

1, stream はそれぞれ, 起動された監視プロセスの識別番号及び覚え書き用文字列を表す.

【例 1】インスペクタからの監視プロセスの起動例

```
get_device(A)> m 1 result
>monitor:1 result started
get_device(A)> ex
aq:1 result E == normal(B,C,D) ?
```

【例 2】リスナからの監視プロセスの起動例

```
{Trace}
?- del:go.
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? m A, result
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? 1
aq:1 result  E == normal(B,C,D) ?
```

コマンドの入力

監視プロセスは, 監視しているデータが深さ 1 のレベルで具体化すると, 報告の条件として指定された項と照合するかどうか検査し, 照合する場合は監視プロセスが起動されたウインドウにデータの内容を表示する.

【例】具体化の表示例

```
aq:1 stream [N|O] == [putl(L)|M] ?
```

上記の例の a はストリーム自動監視モードが on であることを, q は質問モードが on であることを示す. モードが off の場合にはそれぞれの文字 a, q は表示されない. 1 はこの監視プロセスに与えられた識別番号を表す. 識別番号は, リスナからの 1 回のゴールの実行毎に一意に与えられる. また, stream は監視プロセス生成時にユーザが指定した覚え書き (文字列) である. 覚え書きが省略された場合には何も表示されない. [N|O] は報告の条件である. すなわちここでは被監視データがリストになった場合のみ報告することを意味する. [putl(L)|M] は具体化の内容である. 表示と共にプロンプト “？” が表示され, ユーザのコマンド入力待ちとなる. プロンプトの後にコマンドを入力する. コマンドは一部を除いてインスペクタのコマンドと同様のものが有効である (4.2.5.3節モニタの「コマンド」(182頁) 参照).

ストリーム自動監視モード

基本的にはデータは深さ 1 のレベルでのみ監視され, 具体化に伴う表示と上記のコマンド実行の終了後, 監視プロセスは終了する. しかし, プロセス間に張られたストリームを監視する場合等には, 引き続きリストの cdr 部に対して監視が継続されるのが望ましい.

そこで, インスペクタにはストリーム自動監視モードと呼ばれるモードが用意されている. ストリーム自動監視モードは on/off のいずれかの値をとる. on の時, 監視を指示されたデータがリストに具体化した場合, その cdr 部に対してデフォルトで引き続き監視が継続される. リスト以外のデータ型に具体化した場合, 或いはストリーム自動監視モードが off の場合には引き続く監視は行わない. ユーザはコマンドを用いて新たに監視プロセスを生成する必要がある. ストリーム自動監視モードの参照及び設定には **stream_auto_mode** コマンド (174, 183頁参照) を用いる.

質問モード

デフォルトでは, 監視プロセスは, データの具体化を表示した後, 会話形式でデータの調査及び新たな監視プロセスの生成等を行う. しかし, 会話形式のコマンド実行があらかじめ必要ないとわかっている場合には, 質問モードと呼ばれるモードを用いて具体化の表示のみを行うことができる.

質問モードは on/off のいずれかの値をとる. 質問モードが off でさらにストリーム自動監視モードが on の場合に, 監視プロセスは具体化の表示のみを行う. それ以外の場合にはユーザのコマンド入力待ちとなる. 質問モードの参照及び設定には **query_mode** コマンド (174, 183頁参照) を用いる.

監視プロセスの有効期限

リスナからのあるゴールの実行に際して (実行開始時あるいはトレース中) 生成された監視プロセスは, そのゴールの実行が終了するまでの間データの具体化を待って生き続け, ゴールの終了と共に強制終了する. ゴールの実行終了後も監視を続けたい場合には, 次に述べる方法により, 入出力先を変更する必要がある. このようにして切り換えられた監視プロセスは, ゴールの実行が終了しても強制終了されることはなく, 以降リスナが終了するまで独立して動作する.

監視プロセスの入出力先の変更

以下のいずれかの方法により, 監視プロセスの入出力先を変更することができる.

- **monitor_and_fork** あるいは **conditional_monitor_and_fork** コマンドを用いて, リスナ, インスペクタからの起動時に別ウインドウを指定する
- **fork** コマンドを用いて, 監視プロセスの入出力先を別ウインドウに切り換える

モードの初期値と環境変数

各モードの初期値は, リスナからの 1 回のゴールの実行開始毎に設定される. リスナからのゴールの実行開始時に以下の環境変数が設定されていた場合, 各モードはその値に従う.

monitor:stream_auto_mode

インスペクタの監視プロセス生成時のストリーム自動監視モードのデフォルト値を設定する. 値が off の場合, ストリーム自動監視モードの値は off に設定される. それ以外の場合は on に設定される.

monitor:query_mode

インスペクタの監視プロセス生成時の質問モードのデフォルト値を設定する. 値が off の場合, 質問モードの値は off に設定される. それ以外の場合は on に設定される.

monitor:silent_mode

監視プロセスの表示時の beep 音を鳴らすモードのデフォルト値を指定する．値が on の場合, 表示時に beep 音を鳴らさないモードに設定される．それ以外の場合は表示時に beep 音を鳴らすモードに設定される．

終了

コマンド入力待ちの監視プロセスを終了するには, **exit** コマンドを実行する．プロンプトに対して改行コードのみを入力した場合, 監視プロセスの動作は以下のように異なる．

- 監視プロセスが自動的に監視を継続しようとしている場合, 監視を継続する．
- それ以外の場合, 監視プロセスは終了する．

監視プロセスが終了すると, 指定された出力先ウインドウに以下のようなメッセージを出力する．

```
>monitor:1 stream terminated
```

1, stream はそれぞれ, 起動された監視プロセスの識別番号及び覚え書き用文字列を表す．

生成された監視プロセス集合に対する操作

ユーザはゴールの開始から終了までの間, ゴールの実行中生成された監視プロセスに対して, 一覧の表示, 強制終了, 報告条件の変更, ストリーム自動監視モード及び質問モードの変更を行うことができる．それには **monitored_terms** コマンド (175頁参照) を用いる．

monitored_terms コマンドを実行すると, 現在起動されている監視プロセスのうち, 入出力先がコマンドの実行されたウインドウに指定されているものの一覧を表示する．表示後, 会話形式でコマンドを入力することにより以下の操作を行うことができる．操作対象とする監視プロセスの指定は一覧に表示された識別番号によって行う．

kill N, ... :	識別番号 N, ... を持つ監視プロセスを強制終了させる．
sam N, ... :	識別番号 N, ... を持つ監視プロセスのストリーム自動監視モードを変更する．
qm N, ... :	識別番号 N, ... を持つ監視プロセスの質問モードを変更する．
cc N Term:	識別番号 N を持つ監視プロセスの報告条件を Term に変更する．
リターンまたは q:	一覧表示を終了する．
h:	ヘルプメッセージを出力する．

【例 1】監視プロセスの一覧表示例 (トレース中に監視プロセスを起動し, 一覧を表示)

```
{Trace}
?- del:go.
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? m A, result
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? mt

 %%% MONITORED TERMS %%%
aq:1 result B == A ? h

     <cr> : quit
  k N,... : kill monitor
sam N,... : change stream_auto_mode
 qm N,... : change query_mode
cc N Term : change condition
        q : quit
        h : help

(h for help) ?
```

【例 2】監視プロセスの一覧表示, 及びその操作例 (一覧を表示した後, 識別番号 1 の監視プロセスのストリーム自動監視モードを変更する)

```
 %%% MONITORED TERMS %%%
aq:1 file G == F
aq:2 pathname I == H
aq:4 message K == J ? sam 1

 %%% MONITORED TERMS %%%
 q:1 file G == F
aq:2 pathname I == H
aq:4 message K == J ?
```

4.2.5.3 コマンド

監視プロセスに対して実行可能なインスペクタのコマンドは以下の通りである. インスペクタからの実行と動作が異なる場合は動作内容も示してある.

- **inspect** を除くすべての基本コマンド (4.2.4.1節「基本コマンド」 166頁参照)
- **push_root** を除くすべての調査コマンド (4.2.4.2節「調査コマンド」 168頁参照)
- すべての監視コマンド (4.2.4.3節「監視コマンド」 172頁参照)
- すべての環境コマンド (4.2.4.4節「環境コマンド」 175頁参照)
- すべての変数プール操作コマンド (4.2.4.5節「変数プール操作コマンド」 177頁参照)

【参考】将来は **inspect** も実行可能となる予定である.

change_condition Term [cc]

その監視プロセスの報告条件を Term に変更する.

stream_auto_mode [sam]

監視プロセスに対して発行された場合は, その監視プロセスのストリーム自動監視モードの値を変更する.

query_mode [qm]

監視プロセスに対して発行された場合は, その監視プロセスの質問モードの値を変更する.

monitored_terms [mt]

監視プロセスに対して発行された場合は, 自身を除くその他の監視プロセスの一覧を表示する.

【例】バッファを挿入したウインドウに対して入出力を行った場合の, バッファ及びウインドウへの入力ストリームを監視した例

リスナウインドウ: トレース中, ウインドウとバッファへの入力ストリームを監視する. 監視状況は別のインスペクタウインドウに表示

```
{Trace}
[1] del:go.
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? mf A, device
@0@4096 0     del: go
1     * (1)     get_device(A)
2     * (2)     wait_result(A)? 1
@0@4096 2     del: wait_result(normal(B,C,D))
3       (1)     pimos:: buffer: interaction_filter(E,B)
4     * (2)     p(E)? mf E, buffer
@0@4096 2     del: wait_result(normal(B,C,D))
3       (1)     pimos:: buffer: interaction_filter(E,B)
4     * (2)     p(E)? 2
```

インスペクタウインドウ 1: ウインドウの入力ストリームを監視する. getb メッセージが来たら報告し, さらに入力内容を監視

```
>monitor:1 device started
aq:1 device D == normal(A,B,C) ? cm 1 [getb(_,_)|_] wait_getb_msg
>monitor:2 wait_getb_msg started
aq:1 device E == normal(A,B,C) ? ex
>monitor:1 device terminated

 ?? Interruption (h for help): mt

 %%% MONITORED TERMS %%%
  aq:2 wait_getb_msg [getb(F,G)|H] == A ?
aq:2 wait_getb_msg [getb(K,L)|M] == [getb(490,I)|J] ? m car>2 getb_result
>monitor:4 getb_result started
aq:2 wait_getb_msg [getb(N,O)|P] == [getb(490,I)|J] ?

 ?? Interruption (h for help): mt

 %%% MONITORED TERMS %%%
  aq:2 wait_getb_msg [getb(Q,R)|S] == J
  aq:4 getb_result T == I ?
aq:4 getb_result V == normal(U) ? m 1 input_line
>monitor:5 input_line started
aq:4 getb_result W == normal("pd.
") ?
>monitor:4 getb_result terminated
aq:5 input_line X == "pd.
" ?
>monitor:5 input_line terminated
```

インスペクタウインドウ 2: バッファの入力ストリームの監視

```
>monitor:3 buffer started

 ?? Interruption (h for help): mt

 %%% MONITORED TERMS %%%
  aq:3 buffer B == A ? qm 3

 %%% MONITORED TERMS %%%
  a :3 buffer C == A ?
a :3 buffer H == [set_size(char(20,8),D),activate(E),prompt("input>"),gett(F)|G]
a :3 buffer J == [activate(E),prompt("input>"),gett(F)|I]
a :3 buffer K == [prompt("input>"),gett(F)|I]
a :3 buffer L == [gett(F)|I]
a :3 buffer O == [print_depth(M,M),putb("pd= "),putt(M),nl|N]
a :3 buffer P == [putb("pd= "),putt(4),nl|N]
a :3 buffer Q == [putt(4),nl|N]
a :3 buffer R == [nl|N]
```

4.2.6 インスペクタ, モニタコマンド一覧

表 4.3: インスペクタ, モニタコマンド一覧

コマンド	省略名	意味	参照頁
help Type	he,h	Type のコマンド群一覧表示	166
help	he,h	help/1 で指定可能なタイプの一覧表示	166
commands Type	com	Type のコマンド名一覧表示	166
commands	com	commands/1 で指定可能なタイプの一覧表示	167
exit	ex	インスペクタの終了	167
fork	f	入出力先ウインドウ切り換え	167
inspect ?Term	i	子インスペクタの起動	167
set_title String		ウインドウのタイトル設定	167
bigger	big	ウインドウの大きさ変更	167
a_little_bigger	lbig	ウインドウの大きさ変更	168
me M,N		データの M 要素めから N 個表示	168
me M		データの M 要素を表示	168
me		データの 0 要素めから既定個数表示	168
down M	[空]	データを潜る	170
up N	u	データを N 段上がる	170
up	u	データを 1 段上がる	170
length	len	データの要素数表示	171
show ?Term	sh	データを簡易表示	171
history N		潜った経歴を過去 N 回分表示	171
history		潜った経歴を過去既定回数分表示	171
push_root ?Term	pu	調査対象データを設定	172
monitor M Name	m	データの M 要素めをモニタ	172
monitor M	m	データの M 要素めをモニタ	172
monitor_and_fork M Name	mf	別ウインドウでモニタ	172
monitor_and_fork M	mf	別ウインドウでモニタ	173
conditional_monitor M Term Name	cm	データの M 要素めを条件付きモニタ	173
conditional_monitor M Term	cm	データの M 要素めを条件付きモニタ	173
conditional_monitor_and_fork M Term Name	cmf	別ウインドウで条件付きモニタ	173
conditional_monitor_and_fork M Term	cmf	別ウインドウで条件付きモニタ	174
change_condition Term	cc	モニタの条件設定	182
stream_auto_mode	sam	ストリーム自動監視モードの設定	174,183
query_mode	qm	質問モードの設定	174,183
silent_mode Param	sm	beep 音の有無設定	174
monitored_terms	mt	モニター覧表示	175,183
print_depth <D>	pd	構造体深さ出力制限値設定	175
print_length <L>	pl	構造体長さ出力制限値設定	176
print_string_length <L>	psl	文字列長さ出力制限値設定	176
print_atom_length <L>	pal	アトム印字名長さ出力制限値設定	176
print_string_mode	psm	文字列出力形態設定	176
print_mode	pm	データ出力形態設定	177
environment	env	環境変数の値を表示	177
name VarName	n	データを変数に代入	177
list Type	ls	変数一覧表示	177
list	ls	変数一覧表示	178
ビリオドリターン		変数プールの初期化	178

第 5 章

チューニングツール

プログラムのチューニングツールとして, プログラム実行時に各ノードの稼働状態を表示するパフォーマンスメータ (pmeter, rmonitor, amonitor) とリスナが提供する profile 用コマンドでプログラムの実行状況を計測した後で各ノードの稼働状況, ノード間通信の発生状況及びプログラムの実行状況を各種のグラフで表示する ParaGraph がある.

5.1 パフォーマンスメータ

パフォーマンスメータは, 各ノードの稼働状態を観察するためのツールである. 各ノードの稼働状態をノード構成図上で表示する **pmeter**, 各ノードの稼働状態を濃淡グラフで表示する **rmonitor**[1]及び全ノードの平均稼働率を棒グラフで表示する **amonitor**[2]がある.

5.1.1 機能概要

パフォーマンスメータの主な機能の概要を以下に示す.

稼働状態の表示

パフォーマンスメータでは, 一定時間間隔ごとに各ノードの稼働状態を計測し, 表示する. 稼働状態として, 各時間間隔における計算時間 [3], 通信用メッセージの受信時間, 送信時間, GC 時間及びアイドル時間を計測している. **pmeter** 及び **rmonitor** では, 各時間間隔に対する非アイドル時間 (計算時間, 受信時間, 送信時間の和) の割合を各ノードの稼働率として表示する. 環境変数を設定することによって稼働状態別に表示することができる. **amonitor** では, 各時間間隔に対する非アイドル時間の割合の平均値を全ノードの平均稼働率として表示する.

【注意 1】マルチ PSI または PIM 上で, 同一ノードを複数のパフォーマンスメータで計測・表示すると正しく動作しない. また, パフォーマンスメータを起動しながらリスナの profile 用コマンドを実行しても同様に正しく動作しない.

【注意 2】Pseudo マルチ PSI, Pseudo PIM/m 上では正しく動作しない.

5.1.2 操作方法

シェルからコマンドとして呼び出す.

[1] rmonitor は runtime monitor の略称である.

[2] amonitor は runtime average monitor の略称である.

[3] 非アイドル時間から受信時間, 送信時間, GC 時間を引いたものを計算時間としている.

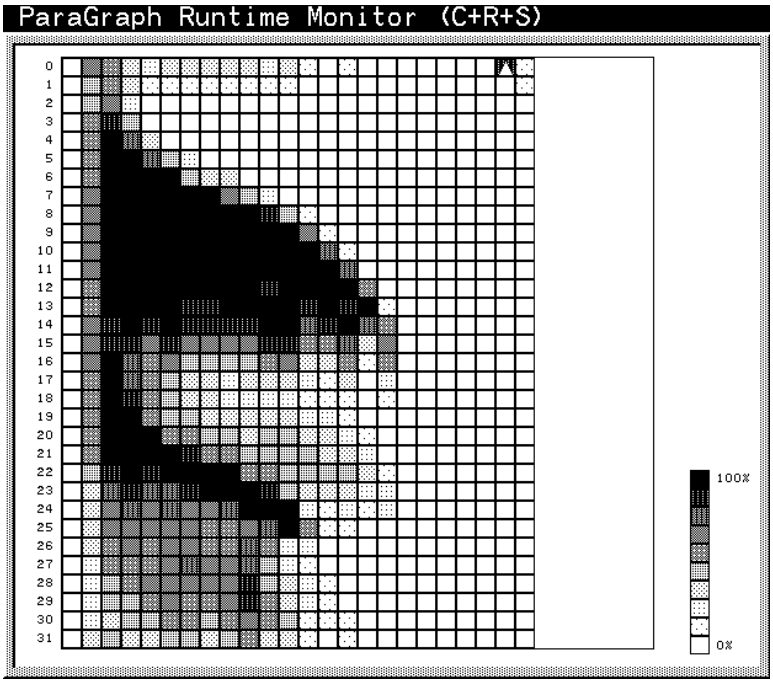

図 5.1: rmonitor の表示例

pmeter

pmeter を起動する. マトリクス上に各ノードを配置し, 色の濃淡で各ノードの稼働率を表示する.

【参考 1】環境変数 **pmeter:item** を設定することによって稼働状態 (計算, 送信, 受信) 別に表示することが可能である.

rmonitor

rmonitor を起動する. 各ノードの稼働状態を濃淡グラフで表示する. 横軸はサイクル番号 [4], 縦軸はノード番号である. 色の濃淡で各ノードの稼働率を表す. また, GC が発生すると, GC が発生したノードに対応するグラフ上に三角形 (△) を表示する. 図 5.1に **rmonitor** の出力画面例を示す.

【参考 1】環境変数 **pmeter:item** を設定することによって稼働状態 (計算, 送信, 受信) 別に表示することが可能である.

【参考 2】濃淡グラフでは, 左端から順に各ノードの稼働率を表示する. 画面上に表示できなくなると画面が自動的にスクロールする.

amonitor

amonitor を起動する. 全ノードの平均稼働率を稼働状態別に色分けした棒グラフで表示する. 横軸はサイクル番号, 縦軸は平均稼働率である. 図 5.2に **amonitor** の出力画面例を示す.

【注意 1】環境変数 **pmeter:item** の設定値とは関係なく表示される.

[4] 各時間間隔に 1 から順に番号付けしたものをサイクル番号と呼ぶ.

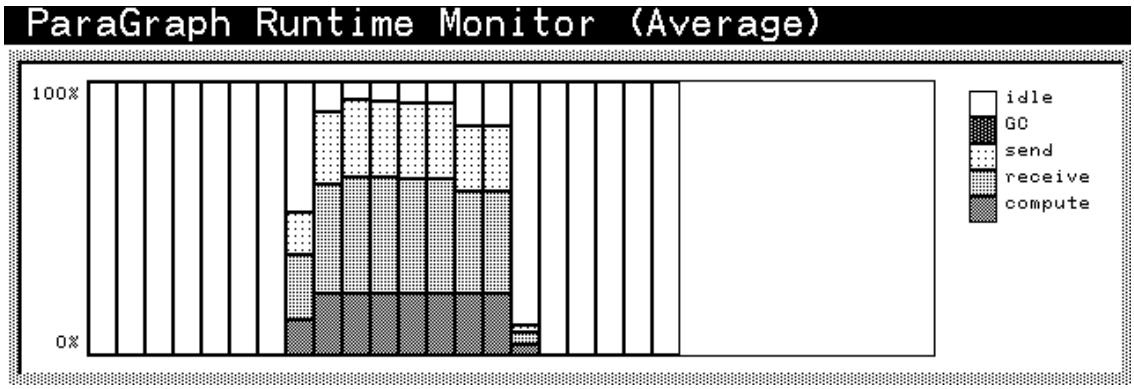

図 5.2: amonitor の出力画面例

5.1.3 環境変数

パフォーマンスメータの属性は以下に示す環境変数で設定できる.

pmeter:interval

計測する時間間隔をミリ秒単位の整数で設定する. 環境変数が設定されていない場合はインターバルを 2000 ミリ秒で計測する.

pmeter:display

カラーディスプレイ上でカラーで表示する場合はアトム color を設定し, モノクロで表示する場合はアトム mono を設定する. カラーディスプレイ使用時に環境変数が設定されていない場合にはカラーで表示する. また, モノクロディスプレイ上では環境変数の値に関係なくモノクロで表示する.

pmeter:timer_node

環境変数に内蔵のシステムタイマまたは FEP 上の計測用タイマを指定する. 内蔵のシステムタイマを指定する場合はアトム system_timer を設定し, FEP 上の計測用タイマを指定する場合は FEP 番号を設定する. FEP が 2 台以上あるシステムでは, 測定用タイマを設定する FEP は表示用ウインドウを設定する FEP とは別の FEP を割り当てるのが適当である (計測誤差を小さくできる). マスタ FEP のみの場合の FEP 番号は 0 である. FEP 番号は 0 〜 3 の整数で指定する. パフォーマンスメータを起動した FEP 番号または I/O ノード名以外の値を指定する場合は環境変数 **pmeter:timer_username** と **pmeter:timer_password** を必ず指定する必要がある. 環境変数が設定されていない場合は内蔵のシステムタイマを用いる.

【注意】環境変数未定義時には計測用タイマとして PIM またはマルチ PSI で内蔵しているシステムタイマが設定される. システムタイマを内蔵していない PIM を除けば, 通常は本環境変数を定義する必要はない.

pmeter:timer_username

pmeter:timer_password

環境変数 **pmeter:timer_node** でパフォーマンスメータを起動した FEP 番号または I/O ノード名以外の値を指定した場合は, 該当する FEP あるいは I/O ノードに登録されている SIMPOS のユーザ名とパスワードをアトムで設定する必要がある.

pmeter:display_node

表示用ウインドウを設定する FEP 番号あるいは I/O ノード名を指定する．スレーブ FEP を持つシステムでは 0 〜 3 の整数で指定する． マスタ FEP のみの場合は FEP 番号は 0 である． パフォーマンスメータを起動した FEP 番号または I/O ノード名以外の値を指定する場合は環境変数 **pmeter:display_username** と **pmeter:display_password** を必ず指定する必要がある．環境変数が設定されていない場合には, パフォーマンスメータを起動した FEP 番号または I/O ノード名となる．

pmeter:display_username

pmeter:display_password

環境変数 **pmeter:display_node** でパフォーマンスメータを起動した FEP 番号または I/O ノード名以外の値を指定した場合は, 該当する FEP あるいは I/O ノードに登録されている SIMPOS のユーザ名とパスワードをアトムで設定する必要がある．

pmeter:gateway_node

リモートで表示する場合にゲートウェイ用の FEP 番号を指定する． FEP が 2 台以上あるシステムでは, タイマ用, ゲートウェイ用の FEP を別々に割り当てるのが適当である (計測誤差を小さくできる)．マスタ FEP のみの場合の FEP 番号は 0 である．FEP 番号は 0 〜 3 の整数で指定する．環境変数が設定されていない場合は, 環境変数 task:fep に指定されている FEP 番号となる．

pmeter:item

稼働状態別に表示する内容を指定する．稼働状態は 3 ビットのビット列 (001 〜 111 の 2 進数表現の整数列) で指定する．ビット列は, 計算, 送信, 受信の順に対応し, 表示する場合は 1, 表示しない場合は 0 を指定する．例えば, 計算の割合にのみ着目して表示する場合は 100, 通信 (送信と受信) の割合にのみ着目して表示する場合は 011 と指定する．環境変数が設定されていない場合には, 全ての稼働状態 (111) を計測し, 表示する．

【例 1】環境変数の設定例

```
SHELL> setenv pmeter:display = mono        % モノクロで表示
SHELL> setenv pmeter:interval = 3000
SHELL> setenv pmeter:display_node = icpsi523
SHELL> setenv pmeter:display_username = akana
SHELL> setenv pmeter:display_password = rika
SHELL> setenv pmeter:timer_node = 0
SHELL> setenv pmeter:gateway_node = 0
SHELL> setenv pmeter:item = 100             % 計算のみを表示
SHELL> rmonitor &
```

5.2 ParaGraph

ParaGraph[5]は, 並列プログラムの実行過程をグラフィック表示する性能評価支援ツールである. ParaGraph はリスナの profile_node, profile_msg, profile コマンドで収集した実行状況情報に基づいて, 各ノードの稼働状況, ノード間通信の発生状況, プログラムの実行状況を各種のグラフで表示する. 本ツールはリスナのトップレベルから起動する.

5.2.1 機能概要

ParaGraph の主な機能の概要を以下に示す.

(1) 実行状況情報のグラフィック表示

ParaGraph では, 何が (What), いつ (When), どこで (Where), どれくらい処理されたかを基本的な表示単位と考え, これらを組み合わせた形式で各種の実行状況情報を表示する. 各表示単位は計測した実行状況情報の種類に応じて異なる. 実行状況情報がノードの稼働状況の場合は, 何が (What) は各ノードの稼働状態 (計算, 送信, 受信, GC)[6]に対応し, 各稼働状態名で表す. ノード間通信の発生状況の場合は通信用メッセージに対応し, 通信用メッセージ名で表す. プログラムの実行状況の場合は, ゴールに対応し, 述語名で表す. いつ (When) は実行状況情報を収集したサイクル[7]に対応し, サイクル番号[8]で表す. どこで (Where) はゴールを実行したノードに対応し, ノード番号で表す. どれくらい処理されたかは, ノードの稼働状況の場合はノードの稼働時間に対する各稼働状態の割合 (%) で表す. ノード間通信の発生状況の場合は送信回数または受信回数で表す. プログラムの実行状況の場合はゴール[9]のリダクション数またはサスペンション数で表す. 以下に表示形式の概略を示す (表示例は節 5.2.3.2を参照).

- **What × When の表示形式**
  何 (What) が, 各サイクル (When) で, どれくらい処理されたか (例えば何回リダクションしたか) を個別に折れ線グラフで表示する.
- **Overall What × When の表示形式**
  何 (What) が, 各サイクル (When) で, どれくらい処理されたかを同一折れ線グラフ中に積み上げた形式で表示する.
- **What × Where の表示形式**
  何 (What) が, 各ノード (Where) で, どれくらい処理されたかを個別に棒グラフで表示する.
- **Overall What × Where の表示形式**
  何 (What) が, 各ノード (Where) で, どれくらい処理されたかを同一棒グラフ中に積み上げた形式で表示する.
- **When × Where の表示形式**
  各サイクル (When) において, 各ノード (Where) で, どれくらい処理されたかを濃淡グラフで表示する. 色の濃さで処理量 (例えば, リダクション数) を表す.
- **Result の表示形式**
  計測結果の一覧を表形式で表示する. 計測結果は項目 (What) 別に集計し, 処理量 (総受信回数や総リダクション数) の多い順にソートして表示する. ただし, ノードの稼働状況では各稼働状態の平均値を表示し, 表示順序は固定である.

---

[5] ParaGraph ≒ Parallel+Graphic

[6] ノードの稼働時間＝計算時間＋送信時間＋受信時間＋ GC 時間＋アイドル時間　と考えて計測している.

[7] リスナの profile コマンドでは, 一定時間間隔ごとに実行状況情報を収集する. この各時間間隔をサイクルと呼ぶ.

[8] 各サイクルに 1 から順に番号付けしたものである.

[9] 各ゴールのリダクションとサスペンションの回数を述語名別に集計している.

(2) 表示項目 (What) の個別指定

What manipulator を用いて収集した実行状況情報中の全計測項目 (What) に対して個別に表示／非表示を設定することができる．これにより, 着目したい項目のみを各種の表示形式で表示することができる．

(3) 実行状況情報の操作

History manipulator を用いて収集した実行状況情報のセーブ, ロード, 表示する実行状況情報の切替え等を行なうことができる．

5.2.2 実行状況情報の収集

5.2.2.1 ノードの稼働状況の計測

リスナのトップレベルから **profile_node** コマンドを実行すると, 実行状況情報として, 一定時間間隔ごとに各ノードの計算時間, 受信時間, 送信時間, アイドル時間, GC 時間が計測される．ParaGraph では, 一定時間間隔における各稼働状態の割合 (%) を表示する．計測した実行状況情報はリスナのデータ格納用プールに保持される (詳細は節 4.1.6.9を参照).

【例】 profile_node コマンドの使用例

```
[7] profile_node(test:go(100,Ans)).
 3: test:go(100,Ans)
  3848reductions
  4228msec
 Ans=[1,2,3,4,...]
```

5.2.2.2 ノード間通信の発生状況の計測

リスナのトップレベルから **profile_msg** コマンドを実行すると, 実行状況情報として, 一定時間間隔ごとに各ノード間で発生した通信用メッセージの受信回数と送信回数を計測する．収集した実行状況情報はリスナのデータ格納用プールに保持される (詳細は節 4.1.6.9を参照).

【例】 profile_msg コマンドの使用例

```
[7] profile_msg(test:go(100,Ans)).
 3: test:go(100,Ans)
  3848reductions
  4228msec
 Ans=[1,2,3,4,...]
```

5.2.2.3 プログラムの実行状況の計測

リスナのトップレベルから **profile** コマンドを実行すると, 実行状況情報として, 一定時間間隔ごとに各ノードで実行したゴールのリダクション数とサスペンション数を計測する．収集した実行状況情報はリスナのデータ格納用プールに保持される (詳細は節 4.1.6.9を参照).

【例】 profile コマンドの使用例

```
[7] profile(test:go(100,Ans)).
 3: test:go(100,Ans)
  3848reductions
```

図 5.3: View selector の画面例

```
 4228msec
Ans=[1,2,3,4,...]
```

5.2.2.4 収集時間間隔の変更

実行状況情報を収集する時間間隔を **interval** コマンドで変更することができる．interval コマンドの引数にはミリ秒単位で時間間隔を指定する． デフォルトの時間間隔は 2000 ミリ秒である (詳細は節 4.1.6.9を参照).

【例】時間間隔の変更例

```
[6] interval(3000).     % 時間間隔を 3000 ミリ秒に変更
```

5.2.2.5 データ格納用プール

収集された実行状況情報はリスナのデータ格納用プールに保持される．データ格納用プールには収集されたすべての実行状況情報が保持されるため, 予めリスナで種々の実行状況情報を収集した後で, History manipulator で各実行状況情報を切替えながら表示することができる．また, データ格納用プールに登録された実行状況情報のセーブ, ロード, 削除等も History manipulator を通じて行なう.

5.2.3 操作方法

5.2.3.1 ParaGraph の起動

リスナのトップレベルから **paragraph** (または **pg**) コマンドを入力すると ParaGraph が起動される. ParaGraph の起動が完了すると, 表示形式の選択用パネルである View selector がスクリーン上に表示される.

【例】ParaGraph の起動

```
[7] profile(test:go(100,Ans)).
 3: test:go(100,Ans)
  3848reductions
  4228msec
 Ans=[1,2,3,4,...]
[8] paragraph.          % View selector がスクリーン右下に表示される.
```

5.2.3.2 View selector の操作方法

表示形式の選択

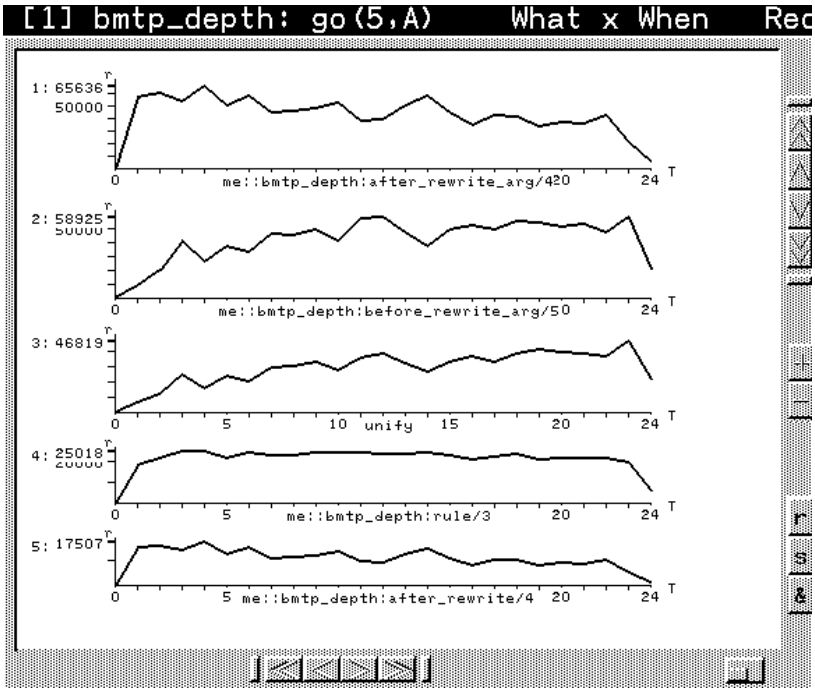

図 5.4: What × When の表示例

View selector の画面例を図 5.3に示す. View selector の項目 What × When, Overall What × When, What × Where, Overall What × Where, When × Where, Result のいづれか一つをマウスクリックすると, 項目が反転表示される. ここで, 項目 do_it をマウスクリックすると選択された項目名に応じた表示形式で実行状況情報が表示される. 各表示形式では, View selector 上のタイプ r, s, & と属性値 Num, To, Window に基づいて表示を行なう (詳細は節 5.2.4.1参照). 以下に各表示形式の選択方法および表示例について述べる.

What × When

項目 What × When を選択した状態で, 項目 do_it をマウスクリックすると What × When の表示形式で実行状況情報が表示される. 図 5.4にプログラムの実行状況の What × When の表示例を示す. 図では, 総リダクション数の多い順に 5 個のゴールが個別に折れ線グラフで表示されている (横軸はサイクル番号, 縦軸はリダクション数).

Overall What × When

項目 Overall What × When を選択した状態で, 項目 do_it をマウスクリックすると, Overall What × When の表示形式で実行状況情報が表示される. プログラムの実行状況の Overall What × When の表示例を図 5.5に示す. 図では, 総リダクション数の多い順に 5 個のゴールが個別の折れ線で表示され, その他のゴールは others として最下位の折れ線で表示されている (横軸はサイクル番号, 縦軸はリダクション数).

What × Where

項目 What × Where を選択した状態で, 項目 do_it をマウスクリックすると What × Where の表示形式で実行状況情報が表示される. プログラムの実行状況の What × Where の表示例を図 5.6に示す. 図では, 総リダクション数の多い順に 5 個のゴールが個別に棒グラフで表示されている. リダクション数は黒いパターン (カラーでは青色) で, サスペンション数はドットパターン (カラーでは緑色) で, リダクション数とサスペンション数が重なっている部分はハッチパターン (カラーでは水色) で表示されている (横軸はノード番号, 縦軸はリダクション数またはサスペンショ

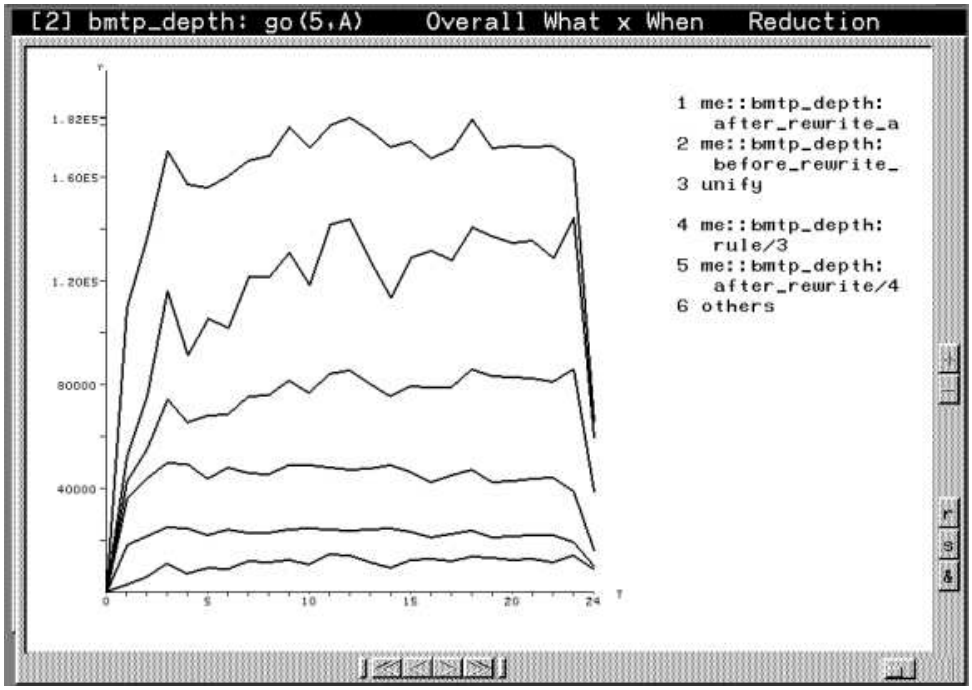

図 5.5: Overall What × When の表示例

ン数).

Overall What × Where

項目 Overall What × Where を選択した状態で, 項目 do_it をマウスクリックすると Overall What × Where の表示形式で実行状況情報が表示される. 図 5.7 にプログラムの実行状況の Overall What × Where の表示例を示す. 図では, 総リダクション数の多い順に上位 5 個のゴールが個別のパターンで上から順に表示されている. それ以外のゴールは others としてグラフの最下位に一つのパターンでまとめて表示されている (横軸はノード番号, 縦軸はリダクション数).

When × Where

項目 When × Where を選択した状態で, 項目 do_it をマウスクリックすると When × Where の表示形式で実行状況情報が表示される. 図 5.8にプログラムの実行状況の When × Where の表示例を示す. 図では, 色の濃さ (パターン) で各ノードで実行したリダクション数が表示されている (横軸はサイクル番号, 縦軸はノード番号).

Result

項目 Result を選択した状態で, 項目 do_it をマウスクリックすると Result の表示形式で実行状況情報が表示される. 図 5.9にプログラムの実行状況の Result の表示例を示す. 図では, 各ゴールの総リダクション数と総サスペンション数の一覧を左側に表示し, 右側に各ゴールの総リダクション数と総サスペンション数の相関関係を棒グラフで表示している (横軸はリダクション数またはサスペンション数, 縦軸は総リダクション数の多い順にゴールを並べた際の順位).

What manipulator の起動

View selector の項目 What をマウスクリックすると What manipulator がスクリーン上に表示される.

【注意】実行状況情報が存在しない (空の) 場合は, What manipulator はオープンできない.

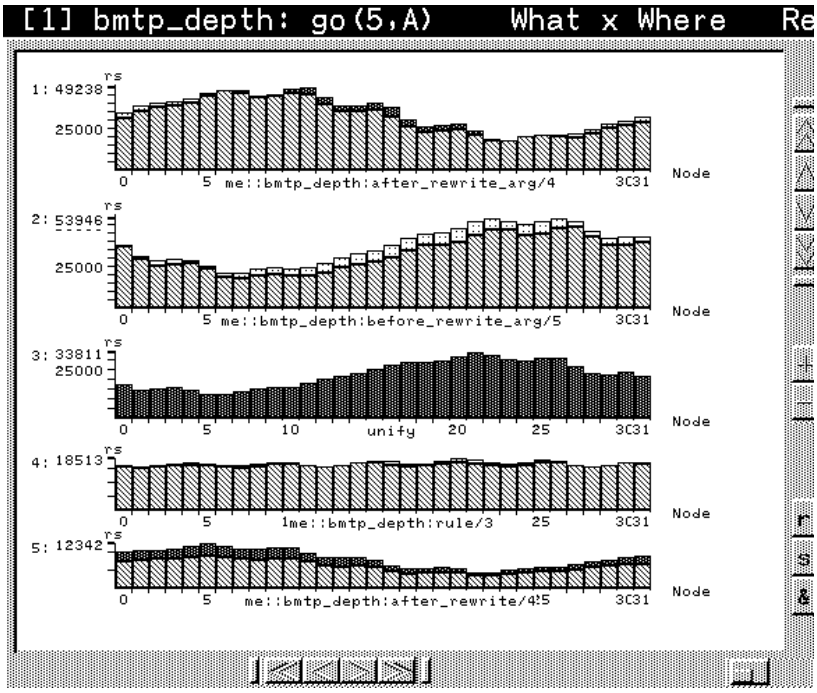

図 5.6: What × Where の表示例

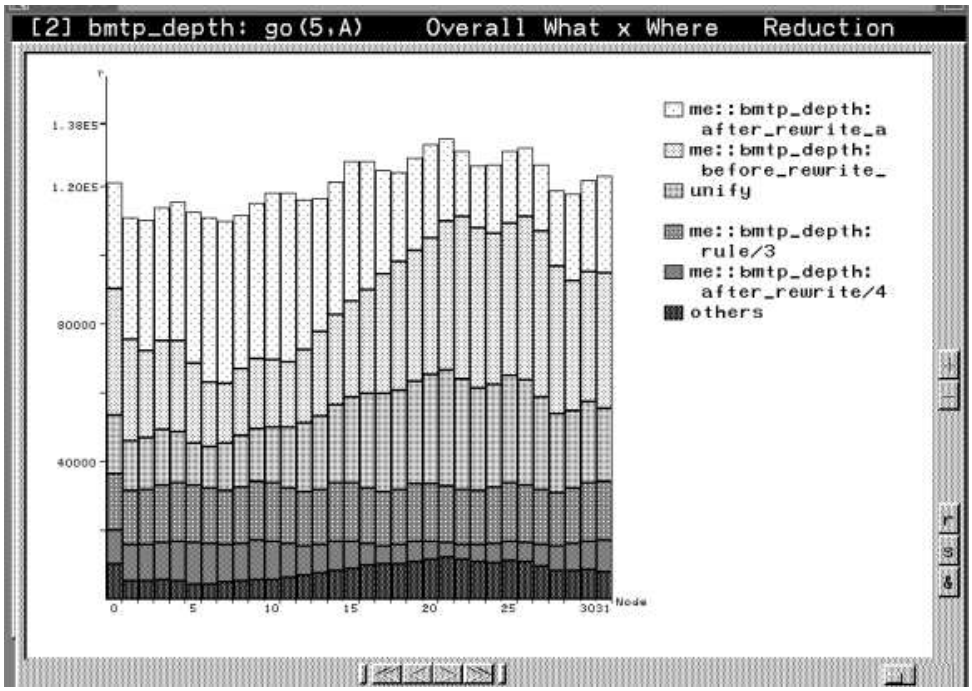

図 5.7: Overall What × Where の表示例

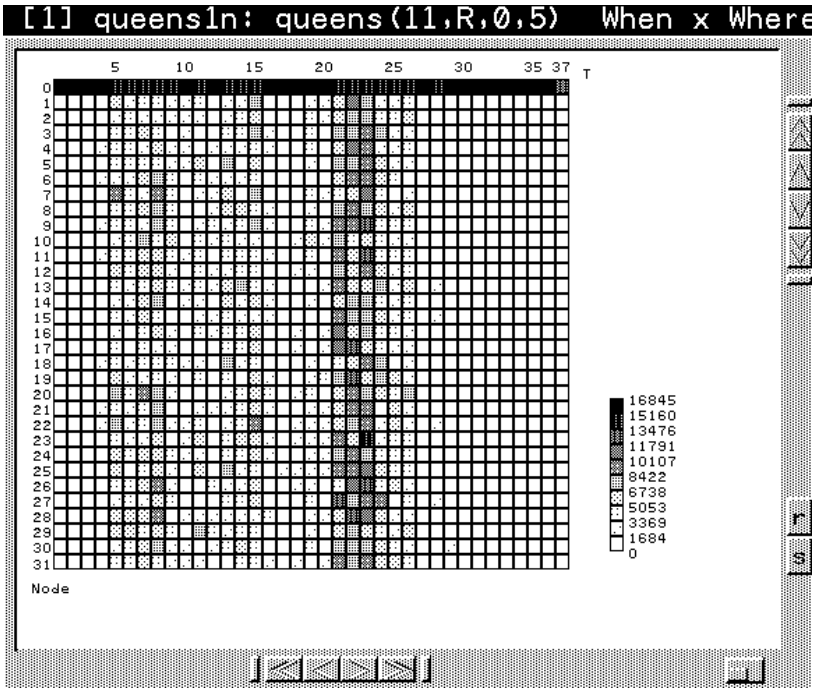

図 5.8: When × Where の表示例

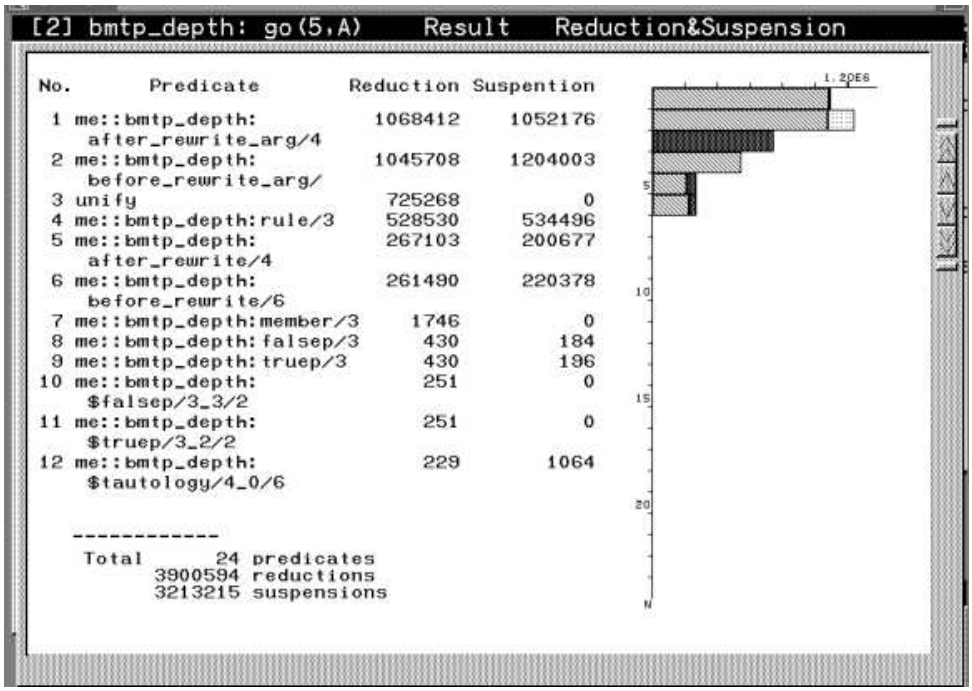

図 5.9: Result の表示例

History manipulator の起動

View selector の項目 History をマウスクリックすると History manipulator がスクリーン上に表示される.

【参考】実行状況情報が存在しない (空の) 場合でも History manipulator はオープンできる.

ParaGraph の終了

View selector の画面右隅のボタン (矩形) をマウスクリックすると現在表示されている全ウインドウ及び View selector がスクリーン上から消去され, ParaGraph が終了する. 実行状況情報はリスナのデータ格納用プールで保持されているため ParaGraph を一旦終了した場合でも再度 ParaGraph を起動すると再び実行状況情報を参照できる.

5.2.3.3 What manipulator の操作方法

ParaGraph で表示する項目 (What) を個別に指定するためのユーザインタフェースが What manipulator である. ParaGraph では, 表示状態が設定されている項目のみを対象として統計処理, グラフィック表示を行なう. What manipulator で着目したい項目に個別に表示／非表示を設定することによって各種の表示形式で設定した項目のみを表示することができる.

項目 (What) の選択

What manipulator のメニューには, 処理量の多い順に項目名の一覧が表示される. 各項目の選択はマウスクリックで行ない, 選択された項目は反転表示される. 一度に複数項目を選択することができる. ParaGraph 起動時および実行状況情報の切替え時の初期設定では, すべての項目が表示状態に設定される. 項目名は種類に応じて以下の形式で表示される.

稼働状態 :	compute(計算) | receive(受信) | send(送信) | GC(ガーベジコレクション)
通信用メッセージ:	通信用メッセージ名
ユーザ定義述語:	パッケージ名:: モジュール名: 述語名 / 引数個数
組込み述語:	組込み述語名

【例】項目の表示例

```
* me::test:go/2
* unify_vector
  pimos::variable:wrap/4
* unify
  unify_list
    .. .. ..
```

【注意】実行状況情報の種類がノードの稼働状況の場合は, 処理量に関係なく表示する順序は固定されている.

表示項目の設定

項目が表示対象として設定されている場合は項目名の先頭に * が表示されている. 各項目を表示対象として設定するには, まず What manipulator 中の項目名をマウスクリックで選択する. 選択された項目名は反転表示される. ここで, 項目 set をマウスクリックすると現在選択されている項目が表示対象として設定される. この他の表示項目の設定方法として, すべての項目を表示対象として設定する set*, キーボード入力で設定する What manipulator 用シェルがある (節 5.2.4.2参照).

表示項目の設定解除

項目の表示状態を解除するには, What manipulator 中の項目名をマウスクリックで選択する. 選択された項目名は反転表示される. ここで, 項目 reset をマウスクリックすると現在選択されている項目の表示状態が解除される. 表示状態が解除されると項目名の先頭の * は消去される. この他の表示状態の解除方法として, すべての項目の表示状態を解除する reset*, キーボード入力で設定解除する What manipulator 用シェルがある (節 5.2.4.2参照).

What manipulator のクローズ exit

項目 exit をマウスクリックすると What manipulator はクローズされる. この時点で設定されていた項目を表示対象とする. ただし, どの項目も設定されていない場合は, 項目が設定されるまで What manipulator はクローズできない.

5.2.3.4 History manipulator の操作方法

ParaGraph で表示する実行状況情報を選択するためのユーザインタフェースが History manipulator である. リスナのデータ格納用プールには複数の実行状況情報が保持されため, ParaGraph では, History manipulator で選択された実行状況情報を各種の表示形式で表示する.

実行状況情報の選択

History manipulator のメニューには, データ識別番号, 実行状況情報の種類及びリスナの profile 用コマンドの引数で指定した Goals の組が項目として表示される. 実行状況情報の種類がノードの稼働状況の場合は {n}, プロセッサ間通信の発生状況の場合は {m}, プログラムの実行状況の場合は {p} で表す. 項目の選択はマウスクリックで行ない, 選択された項目は反転表示される. 項目は一度に一つしか選択できない. ParaGraph 起動時の初期設定では, それまでにリスナで収集された最新の実行状況情報が選択される.

【例】項目の表示例

```
5:{n} test:go([5,4,3],Ans)     % ノードの稼働状況の場合
4:{m} bmtp:go(5,Ans)           % ノード間通信の場合
3:{p} bmtp:go(5,Ans)           % プログラムの実行状況の場合
   .. .. ..
```

実行状況情報のロード, セーブ

項目 load をマウスクリックするとファイル名入力用ウインドウが表示される. ここで, ファイル名入力後, リターンキーを押下するとロードが開始される. ロードが終了すると, メニューが最新の状態に更新されると共にリスナのデータ格納用プールに実行状況情報が登録される. 同様に, 項目 save をマウスクリックし, ファイル名を入力すると指定したファイルに実行状況情報がセーブされる. ファイル名は絶対パス名指定と相対パス名指定が可能である (節 5.2.4.3参照).

【例】絶対パス名指定

```
FileName: >sys>user>me>bmtp32.log
```

【例】相対パス名指定

```
FileName: bmtp32.log
```

History manipulator のクローズ exit

項目 exit をマウスクリックすると History manipulator はクローズされる．この時点で選択されている実行状況情報が表示対象となる．実行状況情報が選択されていない場合は実行状況情報が選択されるまで, History manipulator はクローズされない．ただし, 実行状況情報が無い (空の) 場合はそのままクローズされる．

5.2.3.5 ウインドウインタフェース

ウインドウのラベル

ウインドウのラベルには実行状況情報の種類と表示形式に応じて, ウインドウ番号, 実行状況情報の入力ゴール名, 表示形式名, 計算の割合, 通信の割合, GC の割合等が表示される．

グラフのスクロール

グラフを個別に表示する表示形式において, ウインドウ内に表示されていないグラフを簡単に参照できるようにスクロール用ボタンがウインドウ枠上の右中央または下中央に表示される．ボタンをマウスクリックするとグラフ 1 個分のスクロール, 1 ページ分のスクロール及び先頭または末尾へのスクロールを行なう．

【注意】現在は, スクロールを行なうとウインドウ内の最上位に位置するグラフの最大処理量に合わせてスケールが再調整される．従って, スクロールするたびにスケールが変更されるため絶えず目盛の単位に注意を払う必要がある．

グラフのスケール

グラフが一画面内に表示できない場合 (例えば, ノード数が多い場合や計測した時間が長い場合) は, 全体の様子を一つの画面に収めるために全体表示を行なう．全体表示時には, サイクル番号は複数サイクルをまとめて一目盛として表示し, ノード番号は 8 プロセッサ単位を一目盛として表示する．画面右下のスケール用ボタンをマウスクリックすると全体表示から標準の表示モードへの切り替わる．標準表示時に画面内に収まらない場合は各軸の名称を表す文字列のところに”...” が表示される．

【注意】スケール用ボタンをマウスクリックするたびに全体表示と標準表示が切り替わる．

グラフ数の調整

グラフを個別に表示する表示形式において, ウインドウ内に表示するグラフの個数を調整できるようにグラフ数調整用ボタンがウインドウ右中央に表示される．ボタン＋をマウスクリックするとウインドウ内に表示するゴールの個数が 1 個増加し, ボタン－をマウスクリックすると 1 個減少する．表示形式 What × When, What × Where では, マウスクリックを行なうたびにウインドウ内に表示するグラフの個数が増減し, Overall What × When, Overall What × Where では同一グラフ内に表示するゴールの個数が増減する．

表示タイプの切替え

実行状況情報の種類に応じて表示を切替えるためのボタンがウインドウの右下に表示される．実行状況情報の種類と各ボタンとの対応を表に示す.

実行状況情報の種類	r	s	&
ノードの稼働状況	各稼働状態の割合 (%)	なし	なし
ノード間通信の発生状況	受信回数	送信回数	相関表示
プログラムの実行状況	リダクション数	サスペンション数	相関表示

ウインドウのリシェイプ

SIMPOS のウインドウマニピュレータが提供する機能を利用してウインドウをリシェイプすることができる. リシェイプを行なうとウインドウ内の図形はクリッピングされる. ウインドウのサイズに合わせて表示するためには再度 View selector から指示する必要がある.

【注意】高さ 350 ドット, 幅 350 ドットより小さくすることはできない.

【注意】ウインドウのサイズが実行状況情報を表示するために必要なサイズより小さい場合は何も表示されない. この場合は, 再度ウインドウをリシェイプし, サイズを大きくする必要がある.

5.2.4 コマンド

ParaGraph は, View selector, What manipulator, History manipulator 上の項目で設定された内容にしたがって実行状況情報を表示する. 以下において, 各項目について記述する.

5.2.4.1 View selector 操作コマンド

What × When

What × When の表示形式では, 何 (What) が, 各サイクル (When) で, どれくらい処理されかを折れ線グラフで表示する. 横軸はサイクル番号, 縦軸は実行状況情報及びタイプに応じて異なる.

- **r :** 実行状況情報の種類がノードの稼働状況の場合は各稼働状態の割合 (%) を, ノード間通信の発生状況の場合は各通信用メッセージの受信回数を, プログラムの実行状況の場合は各ゴールのリダクション数を表示する. 表示する順番は処理量の多い順である. ただし, ノードの稼働状況の表示では表示順序は固定している.
- **s :** 実行状況情報がノード間通信の発生状況の場合は各通信用メッセージの送信回数を, プログラムの実行状況の場合は各ゴールのサスペンション数を表示する. 表示する順番は処理量の多い順である.
- **& :** 実行状況情報の種類がノード間通信の発生状況の場合は各通信用メッセージの受信回数と送信回数の相関関係を, プログラムの実行状況の場合は各ゴールのリダクション数とサスペンション数の相関関係を表示する. 同一グラフ中に受信回数またはリダクション数を実線 (カラーでは青色) で, 送信回数またはサスペンション数を点線 (カラーでは緑色) で表示する. 表示する順番は処理量の多い順である.

表示するグラフの個数は属性値 Num で設定された値である. 出力先のウインドウは属性値 Window に表示されている番号に対応するウインドウである.

【注意】実行状況情報がノードの稼働状況の場合は, s, & を選択することはできない.

Overall What × When

Overall What × When の表示形式では, 何 (What) が, 各サイクル (When) で, どれくらい処理されたかを同一折れ線グラフ中に折れ線を積み上げた形式で表示する. 横軸はサイクル番号, 縦軸は実行状況情報及びタイプに応じて異なる.

- **r :** 実行状況情報の種類がノードの稼働状況の場合は各稼働状態の割合 (%) を, ノード間通信の発生状況の場合は各通信用メッセージの受信回数を, プログラムの実行状況の場合は各ゴールのリダクション数を個別の折れ線で表示する. ただし, ノードの稼働状況の表示では表示する順序を固定している.
- **s :** 実行状況情報がノード間通信の発生状況の場合は各通信用メッセージの送信回数を, プログラムの実行状況の場合は各ゴールのサスペンション数を個別の折れ線で表示する.
- **& :** 実行状況情報の種類がノード間通信の発生状況の場合は各通信用メッセージの受信回数と送信回数の相関関係を, プログラムの実行状況の場合はゴールのリダクション数とサスペンション数の相関関係を表示する. 同一グラフ中に各サイクルにおける受信回数またはリダクション数を実線 (カラーでは青色) で, 送信回数またはサスペンション数を点線 (カラーでは緑色) で表示する.

表示する項目の個数は属性値 To で設定された値である. 総項目数が To で設定された値より大きい場合は, それ以外の項目すべてを others として最下位に位置する一本の折れ線で表示する. 出力先のウインドウは属性値 Window に表示されている番号に対応するウインドウである.

【注意】実行状況情報がノードの稼働状況の場合は, s, & を選択することはできない.

What × Where

What × Where の表示形式では, 何 (What) が, 各ノード (Where) で, どれくらい処理されたかを棒グラフで表示する. 横軸はノード番号, 縦軸は実行状況情報及びタイプに応じて異なる.

- **r :** 実行状況情報の種類がノードの稼働状況の場合は各稼働状態の割合 (%) を, ノード間通信の発生状況の場合は各通信用メッセージの受信回数を, プログラムの実行状況の場合は各ゴールのリダクション数を表示する. 表示する順番は処理量の多い順である. ただし, ノードの稼働状態の表示では表示する順序を固定している.
- **s :** 実行状況情報の種類がノード間通信の発生状況の場合は各通信用メッセージの送信回数を, プログラムの実行状況の場合は各ゴールのサスペンション数を表示する. 表示する順番は処理量の多い順である.
- **& :** 実行状況情報がノード間通信の発生状況の場合は各通信用メッセージの受信回数と送信回数の相関関係を, プログラムの実行状況の場合は各ゴールのリダクション数とサスペンション数の相関関係を表示する. 同一グラフ中に受信回数またはリダクション数を黒いパターン light_paint (カラーでは青色) で, 送信回数またはサスペンション数をドットパターン dotted (カラーでは緑色) で表示する. 双方が重なっている部分はハッチパターン right_hatch (カラーでは水色) で表示する. 表示する順番は処理量の多い順である.

表示する項目の個数は属性値 Num で設定された値である. 出力先のウインドウは属性値 Window に表示されている番号に対応するウインドウである.

【注意】実行状況情報がノードの稼働状況の場合は, s, & を選択することはできない.

Overall What × Where

Overall What × Where の表示形式では, 何 (What) が, 各ノード (Where) で, どれくらい処理されたかを同一棒グラフ中に棒を積み上げた形で表示する. 横軸はノード番号, 縦軸は実行状況情報及びタイプに応じて異なる.

- **r :** 実行状況情報の種類がノードの稼働状況の場合は各稼働状態の割合 (%) を, ノード間通信の発生状況の場合は各通信用メッセージの受信回数を, プログラムの実行状況の場合は各ゴールのリダクション数を個別のパターンで表示する. 表示する順番は処理量の多い順である. ただし, ノードの稼働状態の表示では表示する順序を固定している.

- **s :** 実行状況情報の種類がノード間通信の発生状況の場合は各通信用メッセージの送信回数を, プログラムの実行状況の場合は各ゴールのサスペンション数を個別のパターンで表示する. 表示する順番は処理量の多い順である.
- **& :** 実行状況情報の種類がノード間通信の発生状況の場合は各通信用メッセージの送信回数と受信回数の相関関係を, プログラムの実行状況の場合は各ゴールのリダクション数とサスペンション数の相関関係を表示する. 同一グラフ中に各ノードで実行した受信回数またはリダクション数を黒いパターン light_paint (カラーでは青色) で, 送信回数またはサスペンション数をドットパターン dotted (カラーでは緑色) で表示する. 双方が重なっている部分はハッチパターン right_hatch (カラーでは水色) で表示する.

表示する項目 (グラフ) の個数は属性値 To で設定された値である. 総項目数が To で設定された値より大きい場合は, それ以外の項目すべてを others として最下位に位置する一つのパターンでまとめて表示する. 出力先のウインドウは属性値 Window に表示されている番号に対応するウインドウである.

【注意】実行状況情報がノードの稼働状況の場合は, s, & を選択することはできない.

【注意】 ParaGraph で使用可能なパターンの処理量は最大 8 個までのため, 設定された項目の個数が 8 個以上の場合は同じパターンを繰り返し使用することになる.

When × Where

When × Where の表示形式では, 各サイクル (When) において, 各ノード (Where) で, どれくらい処理されたかを濃淡グラフで表示する. 横軸はサイクル番号, 縦軸はノード番号である. 濃淡グラフでは, 色の濃さ (パターン) で各ノードの処理量を表す. 各パターンと処理量との対応はウインドウの右下に表示される. 処理量はタイプに応じて全サイクル中の処理量の最大値に応じて, 10 段階に分割して表示される. ただし, ノードの稼働状況の場合は最大値を 100% に固定している.

- **r :** 実行状況情報の種類がノードの稼働状況の場合は各ノードの稼働状態の割合 (%) を, ノード間通信の発生状況の場合は各通信用メッセージの受信回数, プログラムの実行状況の場合は各ゴールのリダクション数を表示する.
- **s :** 実行状況情報の種類がノード間通信の発生状況の場合は通信用メッセージの送信回数, プログラムの実行状況の場合はゴールのサスペンション数を表示する.

出力先のウインドウは属性値 Window に表示されている番号に対応するウインドウである.

【注意】 When × Where の表示形式を選択している時は, タイプ & と属性値 Num, To は設定できない.

【注意】実行状況情報がノードの稼働状況の場合は, s, & を選択することはできない.

Result

Result の表示形式では, 実行状況情報を一覧表と棒グラフで表示する. 実行状況情報の種類がノードの稼働状況の場合は各ノードの各稼働状態の平均値を表示し, ノード間通信の発生状況の場合は通信用メッセージの総送信回数と総受信回数を, プログラムの実行状況の場合はゴールの総リダクション数と総サスペンション数を表示する. 画面左側に一覧表形式で表示し, 右側には全体に対する各項目の割合を棒グラフで表示する. 縦軸は各項目を処理量の多い順に並べた際の順位を表す. 同一グラフ中に各稼働状態の平均値 (%), 受信回数またはリダクション数を黒いパターン light_paint (カラーでは青色) で, 送信回数またはサスペンション数をドットパターン dotted (カラーでは緑色) で表示す

る. 双方が重なる部分はハッチパターン right_hatch (カラーでは水色) で表示する. 出力先のウインドウは属性値 Window に表示されている番号に対応するウインドウである.

【注意】 Result の表示形式では, スクロール用ボタンをマウスクリックしてスクロールするのは左側の一覧表のみである. 右側の棒グラフはスクロールしない.

【注意】 Result の表示形式を選択している時は, タイプ r, s, & と属性値 Num, To は設定できない.

do_it

View selector 上で指定された表示形式で実行状況情報をカレントウインドウに表示する. 但し, ウインドウ未作成時には新たにウインドウを作成してから表示する.

【注意】実行状況情報が存在しない (空の) 場合は, 何も表示しない.

new_win

新しいウインドウを作成し, View selector 上で指定された表示形式で実行状況情報を表示する.

【注意】実行状況情報が存在しない (空の) 場合は, 何も表示しない.

What

What manipulator をスクリーン上に表示する.

【注意】実行状況情報が存在しない (空の) 場合は, What manipulator はオープンできない.

History

History manipulator をスクリーン上に表示する.

【参考】実行状況情報ファイルをロードできるように, 実行状況情報が存在しない (空の) 場合でも History manipulator はオープンできる.

exit 用 (矩形) ボタン

現在表示されている全ウインドウ及び View selector をスクリーン上から消去し, ParaGraph を終了する. 実行状況情報はリスナのデータ格納用プールで保持されているため ParaGraph を一旦終了した場合でも再度 ParaGraph を起動すると再び実行状況情報を参照できる.

r

実行状況情報の種類に応じて, ノードの稼働状況の場合は各稼働状態の割合 (%) を表示し, ノード間通信の発生状況の場合は通信用メッセージの受信回数を, プログラムの実行状況の場合はゴールのリダクション数を表示する.

【注意】 Result の表示形式が選択されている場合は項目 r は選択できない.

s

実行状況情報の種類に応じて, ノード間通信の発生状況の場合は通信用メッセージの送信回数を表示し, プログラムの実行状況の場合はゴールのサスペンション数を表示する.

【注意】 Result の表示形式が選択されている場合は項目 s は選択できない.

【注意】実行状況情報がノードの稼働状況の場合は項目 s は選択できない.

&

実行状況情報の種類に応じて, ノード間通信の発生状況の場合は通信用メッセージの受信回数と送信回数の相関関係を表示し, プログラムの実行状況の場合はゴールのリダクション数とサスペンション数の相関関係を表示する.

【注意】 When × Where または Result の表示形式が選択されている場合は項目 & は選択できない.

【注意】実行状況情報がノードの稼働状況の場合は項目 & は選択できない.

Num

What × When または What × Where の表示形式において, ウインドウに表示する項目 (What) の個数を指定する. 項目数の初期値は 5 である.

【注意】他の表示形式が選択されている場合はマウスクリックは無視される.

【参考】最大 15 個のグラフをウインドウに表示することができる.

To

Overall What × When または Overall What × Where の表示形式においてウインドウに個別に表示する項目の個数を指定する. 項目数の初期値は 5 である.

【注意】他の表示形式が選択されている場合はマウスクリックは無視される.

【参考】最大 15 個の項目を個別に表示することができる.

Window

出力先のウインドウの番号 (ウインドウ番号) を表示する. 出力先のウインドウはカレントウインドウである. 他のウインドウが選択されると番号も更新される. ウインドウ番号は各ウインドウのタイトルの先頭に表示される.

5.2.4.2 What manipulator 操作コマンド

set

What manipulator 上で選択されている項目を表示対象として設定し, 項目名の先頭に * を表示する.

set*

すべての項目を表示対象として設定し, 項目名の先頭に * を表示する.

shell

What manipulator 用シェルを表示する. キーボードからのコマンド入力によって項目 (What) の表示／非表示の設定ができる. シェルを開くと実行状況情報に応じてメッセージが表示される. 以下に使用可能なコマンドを示す.

- **set**
  What manipulator に登録されている全ての項目に表示設定を行なう.
- **set** 項目名
  What manipulator に登録されている項目の中で指定された項目に表示設定を行なう. 項目名としてワイルドカード (*) が使用できる (例 set pimos*).
- **reset**
  What manipulator に登録されている全ての項目の表示設定を解除する.
- **reset** 項目名
  What manipulator に登録されている項目の中で指定された項目の表示設定を解除する. 項目名としてワイルドカード (*) が使用できる (例 reset pimos*).
- **show**
  What manipulator に登録されている全ての項目を表示する.
- **list**
  表示設定されている全ての項目の一覧を表示する.
- **head**
  環境変数に設定された値に基づいて項目を表示する. 先頭は環境変数 $from で指定し, 表示する項目の個数は環境変数 $num で指定する.
- **setenv** 環境変数 値
  環境変数 ($from, $num) に値 (整数 >0) を設定する (例 setenv $num 10). 環境変数のデフォルト値は $from=1, $num=30 である.
- **getenv**
  全環境変数に設定されている値を表示する.
- **help (?)**
  使用できるコマンドの一覧を表示する.
- **exit**
  シェルをクローズする. 本コマンドにより What manipulator のトップレベルに戻る.

reset

What manipalator 上で選択されている項目の設定状態を解除し, 項目名の先頭に表示されていた * を消去する.

reset*

全項目の設定状態を解除し, 項目名の先頭に表示されていた * をすべて消去する.

exit

What manipulator をクローズする. この時点で設定されていた項目を表示対象とする. ただし, 一つも項目が設定されていない場合は, 項目が設定されるまで What manipulator はクローズできない.

【注意】 What manipulator オープン時の View selector およびウインドウ上のボタンに対するマウスクリックは What manipulator がクローズされるまで保留される.

5.2.4.3 History manipulator 操作コマンド

renew

History manipulator の項目を最新の内容に更新する．主として, paragraph コマンドをバックグラウンドで実行する場合に使用する．これにより, バックグラウンドで ParaGraph を実行していてもリスナから最新の実行状況情報を入手できる．

remove

項目 remove が選択されると現在選択されている実行状況情報をメニューから削除すると共にリスナが保持するデータ格納用プールからも削除する．一旦削除すると, 復元することはできない．

load

実行状況情報ファイルをロードする．項目 load がマウスクリックされるとファイル名入力用ウインドウを表示する．ここで, ファイル名入力後, リターンキーが押下されるとロードを開始する．ロードが終了すると, 内部で renew を行ないメニューを最新の状態にすると共にリスナが保持するデータ格納用プールに実行状況情報を登録する．何も入力せずに, リターンキーが押下された場合は abort とみなし, ファイル名入力ウインドウをクローズする．ファイル名は絶対パス名指定と相対パス名指定が可能である．

【注意】存在しないファイル名あるいは実行状況情報ファイルでないファイル名を入力した場合は abort と同様の処理を行なう．

【例】絶対パス名指定

```
FileName: >sys>user>me>nqueen.log
```

【例】相対パス名指定

```
FileName: nqueen.log
```

save

選択した実行状況情報をファイルにセーブする．項目 save がマウスクリックされるとファイル名入力用ウインドウを表示する．ここで, ファイル名入力後リターンキーが押下されると指定したファイルに実行状況情報をセーブする．また, 何も入力せずに, リターンキーが押下された場合は abort とみなし, ファイル名入力ウインドウをクローズする．ファイル名は絶対パス名指定と相対パス名指定が可能である．

【例】絶対パス名指定

```
FileName: >sys>user>me>bmtp32.log
```

【例】相対パス名指定

```
FileName: bmtp32.log
```

【注意】存在しないディレクトリ名を入力した場合は abort と同様の処理を行なう．

【参考】実機 (マルチ PSI) 上でセーブした実行状況情報ファイルを Pseudo でロードすることも可能である．

plot

項目 plot がマウスクリックされるとサブウインドウを作成し, 画面左側に実行時間 (横軸) とデータ識別番号 (縦軸) の対応関係を棒グラフで表示し, 画面右側にノード数 (横軸) と実行時間 (縦軸) の対応を点グラフで表示する. 点グラフではグラフ内にデータ識別番号をプロットする. plot コマンドではリスナのデータ格納用プールに登録されている全ての実行状況情報を表示する. サブウインドウ上の EXIT ボタンをマウスクリックすると History manipulator のトップレベルに戻る.

exit

項目 exit がマウスクリックされると History manipulator をクローズする. この時点で選択されている実行状況情報が表示対象となる. 実行状況情報が選択されていない場合は実行状況情報が選択されるまで, History manipulator はクローズできない. ただし, 実行状況情報が無い (空の) 場合はそのままクローズする.

第 6 章

システムの管理

6.1 IPL の設定

6.1.1 概要

PIMOS の IPL 方式は, 2.5 版に比べ大幅に変更された. まず, CSP から直接ロードされるコード ("*.kbn" という名前のファイル) は, PIMOS 全体から PIMOS を IPL するためのコードのみになった. この IPL コードは, PIMOS を構成するのに必要なモジュール全てをロードする.

従来は, クロスシステムである CAL で IPL でロードするファイル (*.kbn) を作成していたが, 3.0 版からは PIMOS のアンローダーで IPL でロードするファイルを生成する. このため, ユーザプログラムも設定により IPL 時に PIMOS と同時にロードすることができる.

6.1.2 IPL でロードするファイルの設定

IPL でロードするファイルは, FEP の >sys>csp>PIMOS.DIR の下にある "pimos.ipl" というファイルに記述されている. このファイルには, 行単位で IPL がロードすべきファイルのフルパス名が書かれている. 行の先頭文字がパーセント記号 (%) である場合は, コメント行とみなされる.

ロードはこの pimos.ipl に書かれた順におこなわれる. 同じモジュールが複数のファイルに存在する場合には, 後からロードされたものが有効になる. したがって, (PIMOS に) パッチを当てるには, 最後の行にパッチファイル名を追加すれば良い.

【例】 pimos.ipl の例

```
% pimos 3.0
>sys>csp>PIMOS.DIR>mpsi_bios.sav
%>sys>csp>PIMOS.DIR>ppim_bios.sav     !注意!ロードされない
>sys>csp>PIMOS.DIR>debugger.sav
>sys>csp>PIMOS.DIR>library.sav
>sys>csp>PIMOS.DIR>utility.sav
>sys>csp>PIMOS.DIR>kernel.sav
>sys>csp>PIMOS.DIR>server.sav
>sys>csp>PIMOS.DIR>shell.sav
>sys>csp>PIMOS.DIR>macro.sav
% Load Patach Modules                  !パッチモジュールのロード
>sys>csp>PIMOS.DIR>patch0.sav
% Load Demo Modules                    !ユーザモジュールのロード
>sys>user>FGCS>demo>demo.sav
```

6.1.3 ユーザプログラムの IPL でのロード

ユーザが開発したプログラム (モジュールの集合) を IPL 時に PIMOS と同じにロードするように設定することも可能である. この場合は, pimos.ipl 中の最後にユーザーのプログラムをアンロードしたファイル名を追加するだけで良い.

ただし, このようにしてロードされたモジュールは全て, パッケージ pimos にロードされる. また, モジュールのタイムスタンプ情報は全て失われるので, セルフコンパイラにおける make コマンド (操作編 3 章 コンパイラ 42頁を参照のこと) においては, これらのモジュールに対して無効となる.

また, IPL 時には未定義述語のチェックはおこなわれないので事前に十分チェックする必要がある.

【注意】unload:package/2 で生成したファイルは IPL でロードできない. unload:go/2 または, コンパイラのコマンドによりアンロードすること. (操作編 3 章 コンパイラ 49頁を参照のこと)

6.1.4 IPL の異常

以下に, IPL が CSP に出力する主な異常メッセージとその原因について述べる.

- **Unable to open IPL list file**

  pimos:pimos.ipl というファイルがオープンできない, あるいは見つからない. 原因としては, そのようなファイルが存在しないと考えられる.

- **cannot open file** *FileName*

  指定されたファイルがオープンできない. 原因としては, pimos:pimos.ipl ファイルの指定に誤りがあることが考えられる.

- **Undefined Module Found** *ModuleName*

  リンク中に未定義モジュールが検出された.

  【注意】未定義述語の検出はおこなっていないため, 未定義述語を含むようなモジュールをロードすると, 実行時に例外が報告されることがある.

- **CheckSum Error**

  ロードすべきファイルが壊れている. 原因としては, ファイル転送中にファイルが破壊されたと考えられる.

  【注意】アンロードファイルを FTP で転送する場合には必ず binary (あるいは image) モードで転送すること. さもないと破壊される.

- IPL が例外あるいはメモリ不足などで異常終了した場合

  アンロードファイルが異常で, IPL が暴走したことが考えられる. ロードするファイルのサイズをオリジナル (コピー元) とチェックしてサイズが異なるようなら, 転送 (コピー) で破壊されたことが考えられるので, 再度コピーし直してみる.

6.1.5 CSP による限定トレースのかけ方

IPL でロードしたモジュールに CSP の限定トレースやスパイをかける必要がある場合は, 事前に以下の述語を用いて, シンボルファイルを生成する必要がある. この述語は PIMOS のシェルあるいはリスナから起動する.

csp_atom_table_utility:save(FileName)

FileName (ファイル名ストリング) で指定されたファイルに, その時点までに生成されたアトムの情報を書き出す.

この述語の実行が終了したら, CSP のインスペクトメニューから “Load Symbol Table” を選択し, ファイル名を入力する. 以後, この情報を用いることでスパイの設定やメモリのダンプがシンボリックにおこなえるようになる.

このシンボルファイルの内容は, (IPL でロードされたものに限れば) IPL の順序に変更がない限り同じものになるので, 毎回作り直す必要はない. しかし, ユーザが PIMOS 上でコンパイルやロードしたモジュールに対して限定トレースをかける場合には, この限りではない.

6.2 起動とシャットダウン

6.2.1 起動方法

PIMOS の立ち上げ手順を以下に示す. PIMOS では, システム定義サーバ (399頁) が立ち上がった後, ユーザタスクファイル (6.3 節 (211 頁) を参照) に従って初期タスク (システムの立ち上げ時に最初に起動されるタスクのこと. ログインハンドラが典型的例である.) が起動されるので, そちらも合わせて読まれたい.

【参考】本マニュアルは, マルチ PSI システムを例として記述してある. 他システムの立ち上げ (およびシャットダウン) 方法もほぼ同様と考えてよい.

1. まず, マルチ PSI システムを立ち上げる. pimos を実行する.

   これらの環境設定及び CSP の操作方法については, 『マルチ PSI/V2 コンソールシステム (CSP) 操作説明書』を参照のされたい.

2. ユーザタスクファイルに従って初期タスクが起動される. 多くの場合既定の設定に従ってログインハンドラプログラム (6.4 節 (212 頁)) が起動される. 以後, このプログラムが起動されたこととして話を進める.

3. 以下の通りマルチ PSI システムの CSP ウインドウ中に, IPL のロード情報の出力後, ログインがスタートしたことを知らせるメッセージ が表示される.

```
PE00(00) @ DC> Link Modules ...
PE00(00) @ DC> PIMOS booting ...
PE00(00) @ DC> V 3.000 ...
PE00(00) @ DC> SCSI UNITS FOUND
PE00(00) @ DC> (  INTo) 00 00000007
PE00(00) @ DC> (  INTo) 00 00000000
PE00(00) @ DC> ***** xxx LOGIN SERVICE STARTS
PE00(00) @ DC> (  INTo) 00 00000000
```

4. ユーザがログインをして通常はログイン用のシェルを起動し, PIMOS の利用を開始する. ログインの方法については, 第 1 章 (6 頁) を参照されたい.

【注意】ユーザタスクファイルが存在しないときは, ハングアップするおそれがある.

6.2.2 シャットダウン方法

1. シェルから
   ```
   Shell>> shutdown↩
   ```
   と入力する (リスナから shutdown:go を実行する [1] を実行する).

2. ウインドウが表示され, シャットダウンの意思を再度確認するので
   ```
   Really(yes/no) ?yes↩
   ```
   と答える.

   【注意】もし, シャットダウン処理を中止するならば no と答えること.

3. この操作により, PIMOS の下で起動された全てのタスクが終了し, マルチ PSI システムのコンソールに以下のような PIMOS が起動されてから消費した資源量などが表示される.

```
PE00(00) @ DC> Shutdown started
PE00(00) @ DC> login Server End
PE00(00) @ DC> Statistics in PIMOS:
PE00(00) @ DC> (  INTo) 00 00000000
PE00(00) @ DC> (  INTo) 00 000954C2
PE00(00) @ DC> PIMOS END********
```

【注意】シャットダウン後, システムが完全に終了するか否かはシステムの種類によることである.

6.3 タスク環境・ユーザ定義サーバの設定

ユーザは PIMOS 立ち上げ時に任意のタスクの起動を指定できる. このタスクは, 多くの場合ユーザ定義サーバタスク (プログラム編 11.3 節サーバの定義方法 (11.3 頁) を参照のこと) を起動することができる. 以下, その方法について述べる.

6.3.1 タスク環境の設定方法

PIMOS が立ち上がる際, ブート FEP 上のタスク環境ファイル

```
pimos:pimos.conf
```

の内容に従いタスクを起動する.

【参考】ユーザ定義サーバは, このファイルに記述し, 起動するとよい.
タスク環境ファイルには, 次の項の並びを記述する.

system(Fep, Node, Name, Goal): 指定されたタスクを起動する. 各引数の意味は次の通りである.

Fep: タスクのデフォルト FEP 番号 (integer) を指定. そのタスクの環境変数 task:io_node にこの値が設定される.

Node: タスクの初期ゴールが動作するノード番号 (integer) を指定.

Name: タスク名 (string) を指定.

[1] リリース時のユーザ定義ファイル (1 節 (215 頁) を参照) にあるように, このプログラムはユーザ名 shutdown でログインすればシェルが存在しなくても起動できる.

Goal: “module 名: 述語” の形式で, タスクの初期ゴールを指定する. パッケージは pimos である.

【注意】述語の引数に誤りがあったばあいにはタスクは起動されない. また, 引数中に変数があってもいけない.

【参考】ここで起動するすべてのタスクにバグがあると暴走状態になる場合がある. 任意の時点で強制終了するためにログインハンドラ (次節参照) を起動しておくとよい.

6.3.2 リリース時のタスク環境ファイルの内容

```
system(0, 0, "login", login:go([{0,local,0}],[],off).
```

意味: 初期タスクとしてログインハンドラ
`login:go([{0,local,0}],[],off)` を FEP を 0 番, ノードを 0 番, タスク名 ”login” として起動する.

6.4 ログインハンドラ

ログインハンドラは, ログイン用初期化ファイルの内容に従ってユーザデバイスを初期化し, ログイン処理を行う. ログイン用初期化ファイルは, ユーザ定義ファイルおよびグループ定義ファイルからなる.

6.4.1 ログインハンドラの起動

PIMOS のログインハンドラを起動するためには, 次のパラメータを指定する.

- ログイン要求受付
- ログイン区分
- 利用記録

ログインハンドラの起動方法は次の通り.

`login:go(ログイン要求受付, ログイン区分, 利用記録)`

なお一般ユーザのログインの手順は第 1 章を参照されたい.

【注意】本プログラムを同時に 2 つ起動することはできない.

起動方法および各パラメータについて説明する.

6.4.1.1 ログイン要求受付パラメータ

PIMOS は, ログイン要求を受け付けるための特別な受け口を起動し, ユーザのログインを待つ.
本パラメータは, ログイン受け付けのためのパラメータであり, 以下の項のリストである.

{Fep, Mode, Node}

各要素は以下のとおりである.

Fep: ログイン要求を受け付ける `Fep` 番号.

Mode: ログイン要求の受け付け方. 以下の 3 通り.

local: Fep 自身からのログイン要求を受け付ける. Fep から直接 PIMOS を利用する場合に指定する.

tcpip: Fep を gateway として各 I/O ノードから TCP/IP ネットワーク経由で PIMOS を利用する場合に指定する。

lia: Fep を gateway として各 I/O ノードから psinet ネットワーク経由で PIMOS を利用する場合に指定する。

Node: ログイン要求受け付けルーチンを起動するノード番号である.

6.4.1.2 ログイン区分パラメータ

マルチ PSI を複数ユーザで同時に利用すると, 同じノードを利用するために計測ができない. しかし, 開発時には, 複数ユーザで利用したい等, の諸問題がある.

この問題を解消するために PIMOS では荘園 (プログラミング編 7.5 節 (290 頁) 参照) の空間分割機能を利用したログイン機能を提供する. 本機能では以下のような設定ができる.

- 全ノードを番号順にいくつかの区分に分割する. この区分単位でユーザはノードを利用することとなるが, ユーザはこの区分を意識せずにプログラムを動かすことができる.
- 区分毎に次の利用モードを設定する.
  - シングルユーザ

    本モードの区分は, ただひとりのユーザしかログインできない. 計測時等他のユーザにできるだけ邪魔をされたくない場合に設定する.

    【注意】一部 アトムデータベース等 OS のサービスを利用する場合には, 他のユーザと若干の競合が起こる.
  - マルチユーザ

    本モードの区分は, 複数ユーザの利用を許す. プログラム開発等に利用する場合に設定する.
- 区分が異なるユーザでも, 実際的なプログラムでは入出力処理を行うために FEP 利用の競合が起こり, 思ったような結果がもたらされない場合がある. そのために各区分に利用可能な FEP が指定できる.

ログイン区分指定パラメータは, 以下の項のリストで表現される.

```
{Nodes, Mode, FEP_List}
```

もしくは

```
{Nodes, Mode, Fep_List} * Times
```

である. 各要素は以下のとおりである.

【注意】もしログイン区分指定パラメータが空リストならば, 簡易ログイン機能としてハンドラを起動したことになる.

Nodes: 利用するノードの数

Mode: 利用モード

single: シングルユーザモード

multiple: マルチユーザモード

FEP_List: 本区分へのログイン受け付け処理可能な FEP の FEP 番号のリスト

Times: 同定義の数 (同じ定義の繰り返し記述を避けるために利用する)

【注意】指定の数と実ノード数が一致してない場合, 実ノード数が不足の場合にはログインハンドラは異常終了し, 余った場合には, 残りをマルチユーザモードのログイン区分とする.

6.4.1.3 利用記録パラメータ

ログイン / ログアウト情報をファイル (利用記録ファイルと呼ぶ) に保存する機能利用の可否を指定するパラメータである. 以下の通り.

on 利用記録を採る. 結果は `pimos:pimos.log` に保存される. ファイルのバージョンは PIMOS 立ち上げ時に 1 から順に append open 可能なファイルとなる.

記録を採るのは

- ログインした時
- ログアウトした時
- ログインに失敗した時

である.

off 利用記録を採らない.

6.4.2 ログイン用初期化ファイル

ユーザは, ログイン時に, ユーザ名とパスワードをログインハンドラに渡す. ログインハンドラは, 受け取ったユーザ名およびパスワードをユーザデバイスに問い合わせることによりログイン処理を行なう.

ログインハンドラプログラムは, ログイン用初期化ファイルの内容に従ってユーザデバイスとグループデバイスを初期化する. ログイン用初期化ファイルは, ユーザ定義ファイルとグループ定義ファイルからなる.

6.4.2.1 ユーザ定義ファイル

ファイル名は `pimos:pimos.logins` で, 以下の形式の項の並びから成る.

{User, Pass, UID, MaxPrio, Dir, Node, Prog, Com}

User: ユーザ名ストリング.

【注意】本ファイル中のユーザ名は唯一でなくてはいけない.

Pass: パスワードストリング.

【注意】初期状態は空ストリングである. シェルコマンド change_password で変更する. なおパスワードはファイル上では encryption して表現される.

【注意】コマンド change_password の実行後, そのログインシェルを終了するとユーザ定義ファイルは更新される.

UID: ユーザ ID (非負整数).

【注意】本ファイル中で同一のユーザ ID は認められない. なおユーザ ID 0 は特権ユーザを示すので通常のユーザに設定してはいけない.

MaxPrio: Prog で指定したプログラムタスクの最大優先度.

【注意】0 から 4096 までの整数のみ指定できる.

Dir: 初期ディレクトリ名ストリング

【注意】通常は * を指定し, ログイン時に利用した I/O ノード上のユーザのホームディレクトリとする.

Node: 起動プログラムの動作するノード番号である. * であればノード 0 を示す.

【注意】ノード指定付きログイン機能を利用する場合には各区分のノード 0 が基準になる.

Prog: ログイン時に起動するプログラムである. 以下の形式である.

```
package:: module: predicate
```

package: パッケージ名アトム.

module: モジュール名アトム.

predicate: 述語.

【注意】引数中に変数は記述できない.

Com: コメントを表すストリングのリストである.

【注意】コマンド change_password の実行後, そのログインシェルを終了すると本ファイルは更新される.

6.4.2.2 グループ定義ファイル

ファイル名は, pimos:pimos.group であり, 以下の形式の項の並びからなる.

{Group, GID, UserList, Com}

Group: グループ名ストリング.

【注意】本ファイル中で同一のグループ名は認めない.

GID: グループ ID . 非負整数.

【注意】本ファイル中で同一のグループ ID は認めない. なおグループ ID 0 は特権グループを示す.

UserList: グループ員のユーザ名ストリングのリスト.

Com: コメントを表すストリングのリストである. 特に意味はない.

6.4.2.3 リリース時のログイン用初期化ファイルの内容

1. ユーザ定義ファイル

```
{"super_pimos",  "",   0, 4095, *, *, pimos::shell:login, []}.
{"shutdown",     "",  10, 4095, *, *, pimos::shutdown:go, []}.
{"me",           "", 101, 2000, *, *, pimos::shell:login, []}.
{"you",          "", 102, 3000, *, *,
                        pimos::shell:login(xx,xx,xx,xx,xx), []}.
```

2. グループ定義ファイル

```
{"super_pimos",   0, ["super_pimos", "shutdown"], []}.
{"user",          1, ["me", "you"], []}.
```

6.5 システム定義サーバの登録方法

システム定義サーバ (プログラミング編 第 11 章 資源管理・実行管理 (398 頁) を参照のこと) の登録方法について述べる.

システム定義サーバは, 資源木 (プログラム編 11.1.6 節参照) の最上位に位置するタスクである. このようなタスクは非常に責任が重く, 正常に終了することしか許されないものであり, PIMOS の一部といってよいものである. そのためシステム定義サーバを作るユーザは PIMOS を熟知してなくてはいけない.

ここでは PIMOS にシステム定義サーバを付加する方法を説明する.

6.5.1 システム定義サーバの外部仕様

PIMOS は直接システム定義サーバを起動するため, システム定義サーバモジュールの呼び出し口が統一されている. 次の通りである.

【注意】もちろん public 宣言されている必要がある.

- **go(In, ^Out, Abort)**

  In: アトム ok が unify される. サーバタスクはこの変数が unify されるまで停止してなくてはいけない. システム定義サーバは定義された順に逐次起動される.

  Out: アトム ok を unify する. 次のサーバタスクを起動してよい時期, 例えばサーバ名を登録し終った時, この変数を unify すればよい.

  Abort: この変数に何らかの値 (通常はアトム abort) が設定されたならばサーバタスクの全ゴールは正しく終了しなくてはいけない.

6.5.2 システム定義サーバの登録

システム定義サーバを登録するには, 次のモジュールを作り, サーバプログラムモジュールと本モジュールの unload 形式のファイルを作り, それを ipl ファイル (6.1 節参照) に登録する必要がある.

system_defined_servers:modules(^ServersList)

述語 modules/1 の引数にモジュール定数とタスク名文字列を要素とする 2 要素ベクタのリストを返す.

例

```
:- module system_defined_servers.
:- with_macro pimos.
:- public modules/1.

%
% List = [{ServerModule,ServerNameString},...]
%
modules(List):-
```

```
true|
List = [{module#my_server, "my_server"},
        {module#your_server, "your_server"}].
```

6.5.3 PIMOS 起動時に関するエラーメッセージ

PIMOS 起動時に出力されるおそれのあるメッセージについて説明する．なおログインハンドラについては第 6 章 (208 頁) , ネットワーク設定に関しては, 付録 A をそれぞれ参照されたい．

6.5.4 CSP に表示されるエラー

1. Invalid LOGIN parameter:

   原因 ログインハンドラのパラメータが不正である．

   対処 タスク環境ファイル `pimos:pimos.conf` を正しく修正する．

2. Invalid Fep of Request parameter:

3. Invalid LAN of Request parameter:

4. Invalid Node of Request parameter:

5. Invalid Partition parameters:

   原因 ログインハンドラのログイン要求受付け方法パラメータリストの第 k 番目が不正である．

   対処 タスク環境ファイル `pimos:pimos.conf` 中のログインハンドラの第 1 引数を修正する．

6. Invalid Partition parameter

   原因 ログインハンドラのログイン区分指定のパラメータが不正である．

   対処 タスク環境ファイル `pimos:pimos.conf` 中のログインハンドラの第 2 引数を修正する．

7. Invalid Logging parameter

   原因 ログインハンドラの利用記録パラメータが不正である．

   対処 タスク環境ファイル `pimos:pimos.conf` 中のログインハンドラの第 3 引数を修正する．

8. CANNOT make some devices

9. CANNOT make a timer

10. CANNOT get a general request

11. CANNOT make a login server

    原因 FEP 上に PIMOS のデバイスが作成できない．

    - 通常 SIMPOS のパッケージ mpsicsp の内容がリリース時と異なっている．

    対処 SIMPOS のパッケージ mpsicsp の内容を確認する．

12. CANNOT make a login replier

13. ERROR : initiate a login replier

    原因 ログイン要求処理用 SIMPOS デバイスが作成できない.

    - SIMPOS のパッケージ mpsicsp の状況がリリース時と異なっている.
    - 既にログインハンドラが起動されている.
    - ネットワークが不正状態である.

    対処

    - SIMPOS のパッケージ mpsicsp の状況を確認する.
    - 既にログインハンドラが起動されてないか確認する.
    - ネットワークの状況を確認する.

14. ERROR : initiate a login server (xxxx)

15. ERROR : login server

    原因 ログイン要求処理用デバイスが不正状態である.

    - ネットワークが不正状態である.

    対処 ネットワークの状況を確認する.

16. ERROR : pimos.logins. Retry(1/0)

    原因 ログイン初期化用ファイルのユーザ定義ファイルが不正である.

    対処 ユーザ定義ファイル `pimos:pimos.logins` を修正し, CSP に 1 を入力して初期化をやり直す.

17. ERROR : pimos.groups. Retry(1/0)

    原因 ログイン初期化ファイルのグループ定義ファイルが不正である.

    対処 グループ定義ファイル `pimos:pimos.groups` を修正し, CSP に 1 を入力して初期化をやり直す.

18. CANNOT open pimos.logins. Retry(1/0)

    原因 ログイン初期化ファイルのログインユーザファイルがないかオープンできない.

    対処 `pimos:pimos.logins` を作成もしくは可読状態にして, CSP に 1 を入力して初期化をやり直す.

19. CANNOT open pimos.groups. Retry(1/0)

    原因 ログイン初期化ファイルのグループ定義ファイルがオープンできない.

    対処 グループ定義ファイル `pimos:pimos.groups` を作成もしくは可読状態にして,CSP に 1 を入力して初期化をやり直す.

20. ERROR : nth-term in pimos.logins.

原因 ユーザ定義ファイル pimos:pimos.logins の中で表示の次の行に表示される数値 (16 進数) 番目の項に誤りがある.

21. ERROR : nth-term in pimos.groups.

原因 pimos:pimos.groups の中で表示の次の行に表示される数値 (16 進数) 番目の項に誤りがある.

付録 A

ネットワーク環境の設定

この章ではユーザがネットワークを利用してリモートサイト上の PSI から PIMOS にログインする場合に必要とするネットワーク環境の設定方法について述べる.

現在, PIMOS では, 次のネットワークプロトコルをサポートしている.

- TCP/IP
- LIA network

A.1 TCP/IP

1. SIMPOS 上のファイル >sys>tcpip>net_config.dat を次のように変更する.

   変更前

```
 ......
@spawn
  @@process
      5,simpos##talk_input;
      6,simpos##talk_output;
      8,simpos##x_server;
      9,simpos##remote_printer;
   2000,spawn;
 ......
```

   変更後

```
 ......
@spawn
  @@process
      5,simpos##talk_input;
      6,simpos##talk_output;
      8,simpos##x_server;
      9,simpos##remote_printer;
   2000,spawn;
   2010,pimos_remote_login_server;
 ......
```

   【注意】SIMPOS の仕様は変更になっても 2010,pimos_remote_login_server; を設定すればよい.

2. TCP/IP を起動し直す.(PSI を再起動する.)

【注意】この他, 利用する Multi-PSI に接続している FEP のネットワークアドレスが設定されている必要がある. 具体的には, >sys>tcpip>host.dat の内容を確認する. 詳細は, 「小型化 PSI/SIMPOS TCP/IP ネットワークシステム 説明書 第5章 TCP/IP ネットワーク運用管理」を参照のこと)

A.2 LIA-net

1. SIMPOS 上のファイル >sys>net>process>mylia.dat を次のように変更する.

```
@process
```

と書かれている次の行に,

FEP のノード名::pimos_remote_login_server,*Net_No*,*LIA_No*,*Terminal_No*,40;

の書式で, ネットワークの設定を追加する. *Net_No*,*LIA_No*,*Terminal_No* は,
ファイル>sys>net>process>nclia.dat の *FEP* のノード名と同じものを使用すること.

【注意】なお, ファイル >sys>net>process>mylia.dat が, 存在しないときは, 例を参考にして, 新規に作成する必要がある.

2. LIA-net を起動し直す. (通常は, PSI を再起動する.)

【注意】この他, 利用する Multi-PSI に接続している FEP のネットワークアドレスが設定されている必要がある. 詳細は, 「小型化 PSI/SIMPOS PSI ネットワークシステム 説明書 第5章 PSI ネットワーク運用管理」を参照のこと)

設定の例

自分の PSI から, ノード番号 icpsi108 を FEP としてリモートログインするための設定例について述べる.

ファイル >sys>net>process>nclia.dat に, 次のように記述されているとする.

```
 ......
icpsi108::spawn0,89,7C,01,00;
icpsi108::spawn ,89,7C,01,01;
 ......
```

すると, *Net_No*,*LIA_No*,*TerminaL_No* は, それぞれ, 次のようになる.

Net_No: 89

LIA_No: 7C

TerminaL_No: 01

これを用いて, >sys>net>process>mylia.dat に, 次のように記述する.

```
@process
    icpsi108::pimos_remote_login_server,89,7C,01,40;
     ......
```

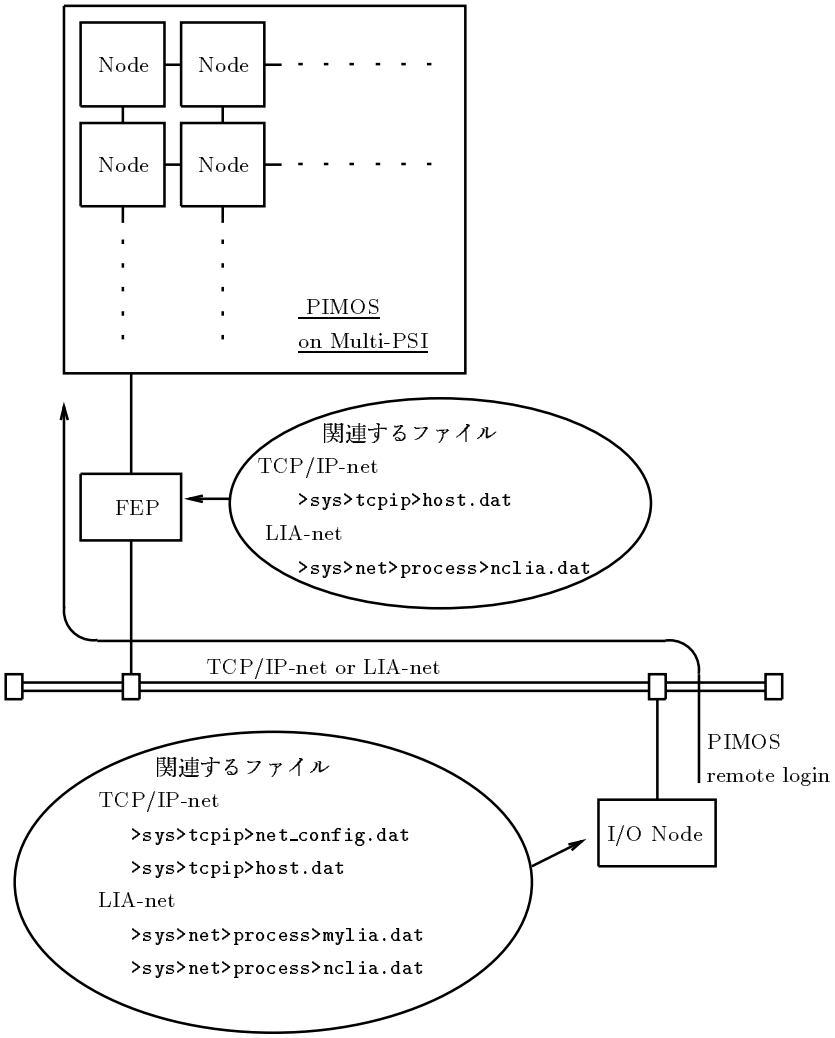

図 A.1: ネットワークの設定に関連するファイル

付録 B

PIMOS のバージョンを知るには

PIMOS 開発者グループに PIMOS のバグに関するクレームなどに関する問い合わせを行う場合, 使用している PIMOS のバージョンが重要な情報となる. 以下では, PIMOS バージョン情報を獲得する方法に付いて述べる.

B.1 CSP に表示するバージョン

PIMOS は, 複数のソフトウェアモジュールから構成されており, 各ソフトウェアモジュールには個々のバージョン情報[1]が存在する. これらのバージョン情報は, 大きく分けて次の 4 種類の情報が存在する.

1. IPL
2. KL0/KL1 処理系
3. クロスコンパイラ (以下, CAL と記す)
4. コンソールシステム (CSP), または, スーパバイザプロセッサ (SVP) 管理プログラム [2](以下, 総称して CSP と記す)
5. PIMOS 本体
6. ParaGraph 表示プログラム
7. FEP 管理プログラム (以下, 単に FEP と記す)

上述のバージョン情報の内, FEP を除き[3], PIMOS 立ち上げ時に CSP 画面上にバージョン情報を表示する.

このうち, IPL, CAL, KL0/KL1 処理系, CSP, ParaGraph 表示プログラムのバージョンについては, CSP のコマンド version によって表示される.

【例 1】 Multi-PSI, PIM/m の表示例.

```
$version
          PIMOS/ CAL/ CSP/  KL1 /  KL0 /ParaG
 Version : 3000/0193/0332/012100/019200/3000
$
```

[1] 通常は番号で表現される

[2] Multi-PSI, PIM/m の CSP については,「マルチ PSI/V2 コンソールシステム操作説明書 (第 3 版」を参照のこと.)

[3] 次版以降で FEP のバージョン情報についても, PIMOS 立ち上げ時に CSP 画面上に表示する予定である.

PIMOS 本体のバージョンはさらに細分化されており, IPL 時に, 設定されたプログラムのバージョンを表示する. [4] これらの情報は, "**IPL loading**" と "**Link Modules ...**" というメッセージに挟まれて CSP に表示される.

【参考 1】PIMOS 照会書の用紙には, これ等のバージョン情報を書く欄があるので, PIMOS 照会書を書く場合には忘れずに書き留めること.

【参考 2】通常, バグが発生して PIMOS がハングした場合, CSP に重要な情報が表示されている可能性が高いので, CSP の画面情報を保存しておくことを勧める.

【例 2】Multi-PSI, PIM/m の表示例.

```
PE00(00) @ DC> IPL loading
PE00(00) @ DC> BIOS    : 2990 : 31-Oct-91 :
PE00(00) @ DC> Debugger: 2990 : 31-Oct-91 :
PE00(00) @ DC> Library : 2990 : 31-Oct-91 :
PE00(00) @ DC> Utility : 2990 : 31-Oct-91 :
PE00(00) @ DC> Kernel  : 2990 : 31-Oct-91 :
PE00(00) @ DC> Server  : 2990 : 31-Oct-91 :
PE00(00) @ DC> Shell   : 2990 : 31-Oct-91 :
PE00(00) @ DC> Macro   : 2990 : 31-Oct-91 :
PE00(00) @ DC> patch1  : 2990 : 01-Dec-91 : BIOS patch
PE00(00) @ DC> Link Modules ...
```

B.2 バージョンを獲得するためのユーティリティ

PIMOS 本体のバージョン情報を知るには, 前述の CSP 画面に表示されるものを確認する他に, PIMOS のシェルなどからユーティリティプログラムを用いる方法も有る. このユーティリティプログラムの使用方法は次の通り.

ipl_versions

IPL したプログラムのバージョン情報を標準出力に表示する. 表示形式は, 次の通りである.

"IPL ファイル名"
{ 空白 } "バージョン情報"

IPL ファイルにバージョン情報が付加されていない場合には, バージョン情報は表示されず, ファイル名のみを表示する. もしバージョン情報がある場合は, IPL ファイルの次の行に, 字下げをして表示する.

B.3 FEP プログラムのバージョン

SIMPOS の デバッガで以下のメソッドを呼び出すことにより,FEP プログラムのバージョンを知ることができる.

```
:pimfep_version(#pimfep##pimos_scsi_front_end,^Version,^Date,^Comment)
```

Version: FEP プログラムのバージョン (16 ビットストリング).

[4] IPL の設定方法に付いては「第 6.1 節 IPL の設定」(208頁) を参照のこと.

Date: FEP プログラムのパッケージテンプレートを作成した日付 (16 ビットストリング).

Comment: コメント (16 ビットストリング).

【注意】このメソッドを呼ぶ時には,pimfep##pimos_scsi_front_end がロードされている必要がある.

【参考】デバッガからでなく,ESP のプログラム中から呼び出すことも可能である.

PIMOS マニュアル
(第 3.0 版)

— プログラミング編 —

ICOT PIMOS 開発グループ
1991 年 10 月

(財) 新世代コンピュータ技術開発機構

はじめに

このマニュアルは並列推論マシンのためのオペレーティングシステム PIMOS 3.0 版の使い方を解説するものです. PIMOS の機能の充実にともなって説明すべきことがらも増えてきたことから, マニュアルは「操作編」と「プログラミング編」の二分冊となっています. 本編 (プログラミング編) では, PIMOS を利用する際のプログラムインタフェースについて解説します. 操作法については操作編を御覧ください. なお, ページ番号は両編を通して振られており, 目次, 索引も全体を通したものが両編共に付けられています.

PIMOS は並列論理型言語 KL1 で記述されており, PIMOS の下で動く利用者のプログラムも KL1 で記述することを前提としています. KL1 は Flat GHC を基礎にさまざまな拡張を施した言語です. プログラミング編には KL1 言語の Flat GHC に対する拡張部分の仕様の概略を収めてありますが, KL1 によるプログラミング技法については, 別途 KL1 プログラミングの教科書である『KL1 プログラミング 入門編 / 初級編 / 中級編』 (ICOT 第 4 研究室) などで勉強されるのが適当です.

第 7 章

KL1 の言語仕様

PIMOS が動作する計算機システムでは, 並列論理型核言語 KL1 をソフトウェアとハードウェアのインタフェースとして採用しており, PIMOS 自身も応用プログラムも, すべて KL1 で (直接, または, より上位の言語によって間接的に) 記述される.

KL1 は並列論理型言語 GHC のサブセットである Flat GHC (以下 FGHC) を基礎とし, 種々の拡張を施したものである. 主な拡張点は, 記述を容易にするための拡張シンタクス, 処理効率向上のための組込みデータ型や組込み述語機能, 例外処理などのメタレベル制御のための荘園機構である.

ここでは原則として KL1 と FGHC の相違点のみを記述する. FGHC については他の文献を参照されたい.

7.1 入出力モード表示の読み方

PIMOS の機能は, PIMOS で提供している述語を呼び出したり, PIMOS へメッセージを送ることによって使うことができる. このとき, 述語やメッセージに使う引数には, 利用者側で設定するもの (入力) と, PIMOS あるいは KL1 言語機能の側で設定するもの (出力) の 2 つがある. 本書では, 引数の入出力関係を分かり易くするための表記を使って記述してある. この入出力関係を示す表記をモード表示と呼ぶ.

モード表示の原則は次の通り.

- モード表示は対象となる項と, その一つ外側の項を具体化するのが同じ側かどうかを示す.
- 両者が同じ場合には特別の表示を行わない. 両者が異なる場合, 対象となる項の前に “^” を付加してその旨を表す.
- なお, 組込述語 unbound や メッセージ putt の第一引数のように, どちらが具体化するか分からないものは “?” をつける.
- 変数に, 組込述語 merge/2 (285 頁参照) の入力ストリームがユニファイされる場合, 対象となる変数の後に “*” をつける.

【例 1】 normal(^Abort, Attention, ID)
デバイスに対する reset メッセージの結果を表す. この項のトップレベルを具体化するのは, PIMOS である. 引数の Abort はユーザ, Attention, ID については PIMOS が具体化する. したがって, Abort だけが一つ上のレベルと具体化するものが異なるので, “^” がつき, それ以外はつかない.

【例 2】 open(Pathname, read(^Result))
ファイルマネージャに対するファイルの open メッセージを表す. このメッセージでは, Pathname や read(_) という構造を具体化するのは, メッセージを送るユーザの側なので, “^” はつけない. Result は PIMOS 側が具体化するので “^” を付ける.

【例 3】 add(Integer1, Integer2, ^NewInteger)
この組込述語の例では, トップレベルの具体化は組込述語を呼ぶ側で行うと考えられる. 入力変数である Integer1, Integer2 は同じ側で具体化するので “^” はつけず, NewInteger は組込述語の中で具体化するので “^” を付ける.

【例 4】 unbound(?X, ^Result)
この組込述語の例では, この述語が呼ばれた時点で, X が未定義変数かどうかのチェックを行う. 上述の例同様, トップレベルは組込述語を呼ぶ側で行うと考えられるが, X については, どちらが具体化するわけでもない. したがって,“?” を付ける.

【例 5】 pool:keyed_bag(S*)
Keyed bag のプロセスを生成する述語である. 変数 S にはプロセスへのストリームがユニファイされる. このストリームは, マージャ付きで提供される. したがって, “*” を付ける. この表記のあるストリームを使用する場合, 通常ユーザが新たにマージャを付ける必要はない.

7.2 シンタクス

KL1 は FGHC に対してさまざまな拡張機能を持つ．それに対応してシンタクスも拡張されている．ここではそういった拡張点を中心に述べる．

シンタクス上の拡張の主な点は

- モジュール構造
- 非対称な節の記述
- 実行優先度指定
- 実行ノード指定
- マクロ記法

である．このうちマクロ記法については，7.3 マクロ記法 (232 頁) の項に述べる．

7.2.1 モジュール構造

KL1 のプログラムは**モジュール**と呼ばれる単位に分割して記述する．

7.2.1.1 モジュールの定義

モジュールの定義はそのモジュールの名前を宣言する

```
:- module モジュール名.
```

で始まる．そのモジュール内の述語で他のモジュールから呼出すことがある述語は

```
:- public 述語名/引数個数, 述語名/引数個数, ・・・ .
```

と宣言する．この public 宣言は :- module の宣言と 最初の述語定義の間に複数回記述できる．同一の述語に関する複数の節は連続して記述しなくてはならない．分割して記述すると，同述語名，同引数個数であっても別の述語定義として扱われ，コンパイル時に多重定義のエラーになる．

7.2.1.2 モジュール間に渡る呼出し

同一モジュール内の述語の呼出しは FGHC と同様である．モジュール間の呼出し形式はゴールの前にモジュール名とコロンを付けて“モジュール名: ゴール”のように記述する．

【例】モジュール間に渡る呼出し

```
pool:bag(S)
```

組込述語名と同名の述語を定義し，その述語を同じモジュールから呼び出すとユーザ定義述語を呼び出す．

【例】組込述語と同名の述語

```
:- module test.
:- public go/2.

add(X, Y, Z) :- true | ...
go(X, Y) :- true | add(X, 1, Y).          % 上に定義した add/3 の呼び出し
```

【参考】陽に組込述語を呼び出すためには builtin: をゴールに付加すれば良い.

【注意】Builtin という名前のモジュールは実際には存在せず, predicate_to_code の引数などに渡したり, PIMOS にモジュール情報を尋ねたりすることはできない.

【例】組込述語の明白な指定

```
:- module test.
:- public go/2.

add(X, Y, Z) :- true | ...
go(X, Y) :- true | builtin:add(X, 1, Y).  % 組込述語 add/3 の呼び出し
```

7.2.2 非対称な節の扱い

GHC では節の記述の順序は意味を持たない. これは KL1 でも原則として同様である. しかし, ひとつの述語を構成する複数の節を, 対等でなく扱いたい場合がある.

そのひとつは, 節の間で実行の優先順位を付けたい場合である. 複数の節が選択可能である時 (いずれもリダクションに必要なデータがそろっており, ガード部の実行が成功する時), 節間に優先度があれば優先度の高い方の節を選ぶ. これを実現するのが節の優先関係指定 (alternatively) の機能である.

いまひとつは, 否定の簡便な記述法である. 節の選択についての条件はガード部に完全に記述すればよいのであるが, 往々にして「この場合とそれ以外」という分け方をしたくなることがある. これを簡便に記述するためのものが KL1 否定の略記法 (otherwise) である.

7.2.2.1 節の優先関係指定 (alternatively)

述語の定義中, 節と節の間に "alternatively." と書く. これにより alternatively の前の節 (群) は, その後の節 (群) より高い優先度を持つことを宣言する. すなわち, alternatively 以前の節の全てがヘッドユニフィケーションまたはガード部の失敗により選択できないことがわかっているか, あるいは, リダクションに必要なデータが揃わないために**まだ**選択できない状態でのみ, alternatively 以降の節 (群) を選択することがある.

データが揃っているかどうかは実行しているノード[1]の上だけで検査する. つまり, リダクションに必要なデータが他のノードにある場合, 他のノードまでアクセスすれば中断せずに実行できる節が alternatively より前にあっても, alternatively 以降にある同じノード内のデータだけで実行できるような節 (群) を実行してしまう場合もある. この場合も中断の原因になったデータの他ノードへのアクセス自体は開始するので, それ以降同じデータを待ち合わせれば, いつかはそれを用いる節が実行されることになる.

一般に alternatively で指定する節間の優先度は, 必ず守られるとは限らない. したがって, alternatively を含む正しいプログラムは alternatively がなくても正しいプログラムでなくてはならない. 換言すれば, alternatively はプログラムの意味に関わるものではなく, 効率向上のための指針である.

【例】優先度つきマージャ

```
    pm([], Y, Z) :- Z = Y.
    pm([W|X], Y, WZ) :- WZ = [W|Z], pm(X, Y, Z).
alternatively.
    pm(X, [], Z) :- Z = X.
    pm(X, [W|Y], WZ) :- WZ = [W|Z], pm(X, Y, Z).
```

[1] KL1 の負荷分散指定単位. 7.2.7 ノード指定の節 (229 頁) 参照.

7.2.2.2 否定の略記 (otherwise)

述語の定義中の節と節の間に “otherwise.” と書くことにより, otherwise 以前に書かれた節の全てが選択できない (ガード部が失敗する) ことがわかった場合のみ otherwise 以降の節 (群) を選択候補とすることを指定できる. 優先関係指定 (alternatively) とは異なり otherwise 以前の節で中断しているものがある限りは otherwise 以降の節は候補としない.

これは otherwise 以降の節すべてのガード部に, otherwise 以前の節のガード条件全ての否定を挿入したのと同様の効果になる.

```
    foo([X|XX]) :- X=a | pa(X,XX).
    foo([X|XX]) :- X=b | pb(X,XX).
    ...
otherwise.
    foo(X) :- true | q(X).
    ...
```

【注意】KL1 ではガード部の実行は左から右の順番に行なう. このような順序づけをしても, 成功する節は必ず成功する. しかし, ガードに複数の条件があると, この順序づけのために最初の条件の検査で実行を中断してしまい, 他の条件の失敗の検出にまで至らない場合がある. たとえば以下のようなプログラムを `foo(_, 1)` で呼び出すと, 最初の節で第一引数の `a` とのユニフィケーションで中断し第二引数のユニフィケーションに進まないため, その失敗は検出しないまま中断し, otherwise 以降の第二節には実行が進まない.

```
    foo(a, 0) :- true | bar.
otherwise.
    foo(X, Y) :- true | baz(X, Y).
```

7.2.3 ガード内 OR

ガード内では ';'/2 を記述することにより, いずれか一方が成立すれば良いような条件 (“OR”) を記述できる.

【例】以下の 2 つのコードは同じ意味になる.

```
foo(X) :- atom(X) | bar.
foo(X) :- integer(X) | bar.
```

```
foo(X) :- (atom(X); integer(X)) | bar.
```

この, ';'/2 は 1 つのガード中に何回出現しても良いし, ネストしても良い.

【注意】この記法は, PDSS, CAL では使用できない.

【注意】現在のコンパイラのガード内 OR 機能は, ソースコードレベルで実際に上記例の様に展開することにより実現している. ボディを 1 つにまとめる等のコードサイズを小さくするための最適化は行なっていない.

7.2.4 その他の指示語

コンパイラのパーサで有効な指示語として

- オペレータ指定指示語 (:- op/3)
- ファイル読み込み指示語 (:- include/1)

がある.

【注意】この 2 つの指示語は PDSS, CAL では使用できない.

7.2.4.1 オペレータ指定指示語

以下の指示語をソースプログラム中に記述することにより, ユーザ定義のオペレータを, 指定することができる.

```
:- op(Prec, Type, OpName).
```

Prec : 優先度を表す整数 (1 〜 1200)

Type : 演算子結合型を表すアトム (xfx, xfy, yfx, yfy, fx, fy, xf, yf)

OpName : オペレータアトム

7.2.4.2 ファイル読み込み指示語

以下の指示語をソースプログラム中に記述することにより, 他のファイルを読み込むことができる.

```
:- include(ファイル名).
```

“ファイル名” は読み込むファイルの名前を文字列で指定する.

【参考 1】PIMOS のセルフコンパイラではコンパイラに指定された path より読み込むファイルを決定する. 詳細はコンパイラの環境変数についての記述 (51 頁, 3.1.4) を参照のこと.

【参考 2】読みこまれたファイル内で指定されているオペレータ指定は, 読みだし側のファイル (:- include が書かれている方のファイル) でも, 有効になる.

【例】以下のような内容のファイル,”myop.h” を考える.

```
:- op(500, xfx, my_operator).
```

このファイルを取り込む時に, アトム, my_operator の扱いは, 以下のようになる.

```
foo(X) :- true |
    X = a my_operator b.  % 文法エラー.
.....
....
:- include("myop.h").     % myop.h の読み込み.
.....
...
bar(X) :- true |
    X = a my_operator b.  % 正しく読み込める.
```

【注意】この指示語は PDSS, CAL では使用できない.

7.2.5 データの表記

KL1 のプログラム / データの表記法は原則としてはエジンバラ Prolog と同様の, 演算子順位文法に基づくものである.

ここでは一般ユーザが KL1 のプログラムで扱える / 扱って意味があるものだけを述べる. なお, 以下の記述中で『日本語文字』とは, ASCII 文字セットに対応する文字がないような文字 (仮名, 漢字など) を意味する.

変数 英大文字または下線 ("_") で始まる, 英数字, 日本語文字, または下線の並び. ひとつの節の中の同じ名前の変数は, 同じ変数を表す. ただし, 下線のみからなる変数はその範囲内に複数個あらわれても, 各々独立した変数とみなす.

整数 原則として 10 進数として表記する. 2 進から 36 進で記述する事もできる. 詳しくは, マクロ記法の項を参照. 符号は整数に含まれる.

浮動小数点数 浮動小数点数は,

[符号] 数字 + 小数点 数字 + [e または E [指数部符号] 数字 +]

という形式で表現される. ここで, [ ] 内はオプションを, 数字 + は 1 文字以上の数字の並びを意味する. 途中に空白を含んではならない. 整数と同様に符号は浮動小数点数に含まれる.

【参考】マルチ PSI では浮動小数点数の内部表現は IEEE 準拠の単精度 (32 bits) を採用している.

【注意 1】現在ソーステキスト中に ± 0.0 を記述することはできるが, 無限大を表す浮動小数点数 ± Infinity や非数 (not a number: NAN) を記述することができない.

【注意 2】浮動小数点数どうしのユニフィケーションは内部表現のビットパターンの比較で行われる. したがって, 表示した時に同じに見えても, ユニフィケーションが失敗する場合がある. 一般に, 浮動小数点数どうしのユニフィケーションは行うべきでない.

アトム エジンバラ Prolog とほぼ同様で,

1. 英小文字または日本語文字で始まる, 英数字, 日本語文字または下線の並び

   【注意】特殊文字は含んではいけない.
2. 次の特殊文字の並び: ～＋－＊／＼＾＜＞＝：． ？@＃＄＆
3. ふたつの引用符 ("'") で両側を囲った文字列. 引用符を含む場合は引用符を 2 回続けて書く.
4. ！ ； []

ベクタ 原則的には, 中括弧対 ({}) の中に要素をコンマで区切って記述する. 別にエジンバラ Prolog と同様のファンクタ形式 (第 0 要素がアトムであるようなベクタの表記の別法) があり, ファンクタ記号が第 0 要素のアトムを, 引数は第 1 要素以降の要素を指定する. 例えば `f(a)` と `{f,a}` は同じものである.

【注意】要素数が 0 の場合の `{}` もベクタである.

リスト エジンバラ Prolog と同様, 鉤括弧 ([]) の中に要素をコンマで区切って記述する. リストの終端がアトム [] 以外の場合に縦線 (|) を区切りに用いる点も同様である.

ストリング デフォルトのビット幅の要素をもつストリングはダブルクォートの中に要素の文字を書いて, `"abc"` または `string#"abc"` のように表記する. ダブルクォート自身を要素としたい場合は, ダブルクォートをふたつ続けて書けば, ひとつのダブルクォートを要素として指定したことになる.

マルチ PSI の場合, デフォルトは 16 ビットで, JIS 漢字コード系を用いる. 8 ビットの ASCII コードストリングは `ascii#"abc"` のように表記する.

【参考】この他に 1 ビットおよび 32 ビットの要素を持つストリングもあるが, 現在のところ定数としての表記法はない (実行時に動的に生成し利用することはできる).

表 7.1: KL1 のデータ表記例

変数	…… A, B, C, _, _abc
整数	…… 123, 16#"ACE", 8#"37"
浮動小数点数	…… 1.23, 1.0e10, 3.0E-30, -2.0
アトム	…… abc, 'ABC'
リスト	…… [1,2,3], [1|X]
ベクタ	……{a,Y,b}, f(X), {}
ストリング	…… "abc", "", string#"abc"
モジュール	…… module#foo, module#bar

【注意】KL1 ではストリングは独立したデータ型であり, エジンバラ Prolog のような文字コードを要素とするリストの略記法ではない.

【注意】ストリングどうしのユニフィケーションは要素サイズ, 長さが同じで, 双方の全ての要素が同じである場合に成功する. これは空ストリングについても同様で, 16 ビットの空ストリングと 8 ビットの空ストリングはユニファイできない.

モジュール プログラムの記述単位であるモジュールに対応するオブジェクトコードは, モジュール型のデータとして取り扱うことができる. モジュールは module# モジュール名 のような形式で表記する.

【注意】ここに述べた表記法の内, 文字 # を用いた表記は基本的なシンタクスではなくマクロ (後述) である. このため, プログラム中に記述すればここに述べたような意味を持つが, 単に読み込んだだけでは # を第 0 要素とするベクタとなる.
KL1 のデータ表記の例を表 7.1 に掲げる.

7.2.6 実行優先度指定

KL1 にはゴールの実行優先度 (以下, ここでは単に優先度と言う) をゴールごとに指定する機能がある. 複数のゴールが同一のノード[2]内でリダクション可能である場合, 原則として優先度の高いゴールから先にリダクションを行う.

【注意 1】マルチ PSI では組込述語に優先度は指定できない.

【注意 2】優先度は, ノード内でのみ有効であり, ノード間にまたがっては保証されない. 将来負荷分散をある程度自動化する可能性がある. この際にはどのゴールが同じノードにあるかはユーザからは必ずしもわからなくなるので, 優先度による実行順序は一般に保証されないと考えた方が良い. 実行順序を保証したければ, ガード部の実行に必要とされるデータの依存関係によって行い, 優先度は効率向上のための指針としてのみ用いるべきである.

優先度には論理優先度と物理優先度があり, ゴールはそれぞれ自身の論理優先度を持つ. 処理系内にはあらかじめ指定されたレベル数の物理優先度があり, スケジューラは論理優先度を物理優先度に変換してから用いる. 従って, 物理優先度の範囲が小さい場合は, ユーザが (論理) 優先度を変えても物理優先度が変わるとは限らない. 荘園 (後述) が持つ優先度の上限 / 下限も論理優先度である.

ユーザはゴールの優先度を指定する場合その親ゴールの論理優先度やそのゴールが所属する荘園の上限 / 下限との相対値で指定する. 前者を「所属荘園内自己相対指定」, 後者を「所属荘園内割合指定」と呼ぶ.

[2] KL1 の負荷分散指定単位. 7.2.7 ノード指定の節 (229 頁) 参照.

7.2.6.1 所属荘園内割合指定

指定されたゴールが所属している荘園の論理優先度の上限 / 下限で決まる範囲内の割合で指定する方法であり, 次のような形式で指定する.

Goal @ priority(割合)

このときゴール Goal の論理優先度は,

$$\text{下限値} + (\text{上限値} - \text{下限値}) \times \frac{\text{割合}}{4096} \qquad (0 \leq \text{割合} \leq 4096)$$

で決まる.

【参考】これは PIMOS の以前の版にあった

Goal @ priority(*, 割合)

という形式と同じ意味である. 整合性維持のためこの形式も有効だが, 新形式による記述を推奨する.

7.2.6.2 所属荘園内自己相対指定

この指定方法は呼び出すゴールの論理優先度との相対値で指定する方法である. この場合も荘園の論理優先度の上限 / 下限を越えることはできない. 次のような形式で書く.

Goal @ relative_priority(割合)

このときゴール Goal の論理優先度は, 呼び出すゴールの論理優先度を Cp とすると,

$$Cp + (\text{上限値} - Cp) \times \frac{|\text{割合}|}{4096} \qquad (0 \leq \text{割合} \leq 4096)$$

または,

$$Cp - (Cp - \text{下限値}) \times \frac{|\text{割合}|}{4096} \qquad (-4096 \leq \text{割合} < 0)$$

で決まる.

【参考】これは PIMOS の以前の版にあった

Goal @ priority($, 割合)

という形式と同じ意味である. 整合性維持のためこの形式も有効だが, 新形式による記述を推奨する.

【例】次のプログラムで Goal の論理優先度が Cp とすると, ゴール q の論理優先度 Np は図 7.1に示すように計算される.

```
Goal:      ?- p.
Clause:    p :- true | q @ relative_priority(-100).
```

7.2.7 ノード指定

KL1 ではゴールごとに実行するノードを指定することができる. クラスタのように共有メモリで結合された 1 つ以上のプロセッサの集合のことで, マルチ PSI の場合には 1 ノード 1 プロセッサということになる. 以下のような形式でノード番号により指定する.

Goal @ node(ノード番号)

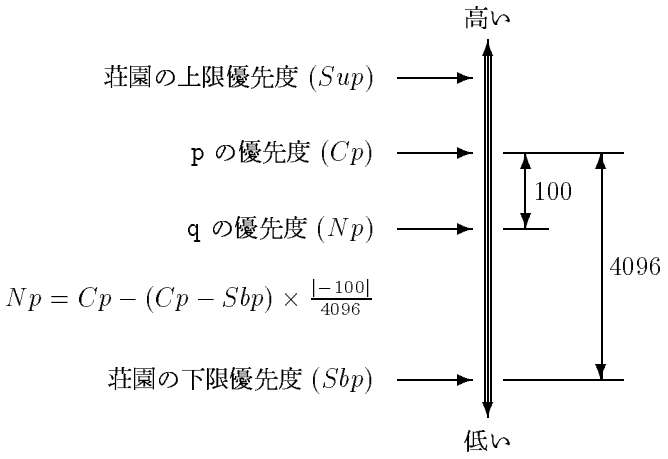

図 7.1: 実行優先度の算出方式

特にノードを指定しないゴール呼び出しは, 親ゴール (そのゴールを呼び出したゴール) と同じノードで実行される.

【注意】マルチ PSI では組込述語にノード指定はできない.

【注意】マルチ PSI では現在外部モジュールで定義されている述語に優先度指定やノード指定して, 直接呼び出すことができない. このため, ソースプログラム中にそのような記述がなされている時には, コンパイラが以下のような中間述語を自動的に生成する.

```
ソースプログラム中の記述:
   ..., module:goal@node(N), ...
コンパイラでの解釈:
   ..., '$$goal/0_1'@node(N), ...

   '$$goal/0_1' :- true | module:goal.
```

【注意】同一ゴールにノードと優先度を同時に指定した時には, コンパイラにより以下のように展開される.

```
ソースプログラム中の記述:
   ..., goal@priority(1024)@node(10), ...
コンパイラでの解釈:
   ..., '$$goal/0_0'@node(10), ...

   '$$goal/0_0' :- true | goal@priority(1024).
```

各指定 (上記例では `priority(1024)` と `node(10)`) の展開される順は, それらがノード指定であるか優先度指定であるかにはよらずその記述された順のみによって決定され, 再右のものから順に「実行」されるようなコードが出力される. また, 同種類の指定が複数回同一ゴールになされている時もコンパイラは単に上記のように複数述語に展開する.

ソースプログラム中の記述:

```
   ..., goal@node(0)@node(1)@...@node(10), ...
```

コンパイラでの解釈:

```
   ..., '$$goal/0_0'@node(10), ...

   '$$goal/0_0' :- true | '$$goal/0_1'@node(10).
   '$$goal/0_1' :- true | ...
   ...
   '$$goal/0_9' :- true | '$$goal/0_10'@node(1).
   '$$goal/0_10' :- true | goal@node(0).
```

7.3 マクロ記法

7.3.1 概要

KL1 ではプログラムの記述性向上のためにマクロ展開機能を用意している. マクロは大略以下のように分類することができる.

システム定義のマクロ： あらかじめ, PIMOS の提供するマクロ展開器, コンパイラで用意されているマクロ. 以下のものがある.

- 定数記述
- ユニフィケーション
- 数式マクロ
- 条件分岐
- 暗黙の引数
- 組込述語に展開されるマクロ
- 条件コンパイル

ユーザ定義のマクロ： ユーザが独自の定義を行なうことができるマクロ. 上記システム定義のマクロにも, ユーザ定義マクロと同じ機構を用いて定義されているものがある.

なお, 以下の説明において, 組込述語に展開されるマクロがあるが, 組込述語の機能については 7.4 組込述語 (264 頁) の項を参照のこと.

【注意】セルフコンパイラでは上に挙げたシステム定義のマクロのうち, 以下のものについては, 7.3.4 (245頁) に解説するユーザ定義マクロの機能を用いて, マクロモジュール `pimos` の定義を用いる.

- 定数記述
- ユニフィケーション
- 数式マクロ
- 組込述語に展開されるマクロ

主に以下のような拡張 / 相違があるが, これ以外は 2.0 版以前のマクロ定義と同様の仕様である.

- これらのマクロは, 種類によらず, 統一的にトップダウンにマクロ展開する
- 数式マクロで使用されていたマクロの抑制記法 (‘(項), ‘‘(項)) は, これら全てのマクロの展開を抑制する
- ユーザはこれらのマクロの一部 / 全部を再定義できる

【注意】PDSS, CAL では, 組込述語に展開されるマクロはユーザ定義マクロの機構を用いて定義されている. それを除いたシステム定義のマクロはユーザ定義のマクロと別の枠組で定義されている.

7.3.2 マクロ展開の抑制

ユーザ定義マクロ, および, システム定義のマクロでもユーザ定義マクロの機構によって定義されているマクロについては, 以下の記法を用いて展開を抑制することができる.

- ‘‘(項)
  項中のマクロ展開は全て抑制され, 項そのものを意味する.

- ‘(項)
  項が構造体であった場合, トップレベルの展開のみが抑制され, より深いレベルはマクロ展開の対象となる.

条件分岐, 暗黙の引数, 条件コンパイルのマクロは, 展開機構が異なるので, この方法で展開を抑制することはできない.

【注意】PDSS, CAL ではシステム定義のマクロの展開は抑制できない.

7.3.3 システム定義のマクロ

7.3.3.1 定数記述

これらのマクロは定数を生成する為に用いられる.

底 #"文字列 "
文字列を底で指定した基数に基づく整数値に変換したものに展開する. 底としては 2 から 36 までが許される. 文字列中の文字は, 基数に応じて順に “0”, “1”, ..., “9”, “a”/“A”, “b”/“B”, ..., “z”/“Z” を使用する. 大文字と小文字の区別はしない.

string#"文字列 "
文字列に対するデフォルトタイプの文字コード (JIS 漢字コード, PDSS では ASCII コード) のストリングになる.

ascii#"文字列 "
文字列に対する ASCII コードからなる 8 ビットストリングになる.

#"文字 "
文字に対するデフォルトタイプの文字コード (JIS 漢字コード, PDSS では ASCII コード) になる.

c#"文字 "
文字を表す ASCII コードになる.

ascii# 文字
文字は一文字の名前を持つアトムである (例: `ascii#'['`). その文字を表す ASCII コードになる.

【注意】将来の PIM ではサポートしない可能性がある.

key#lf
改行を表す ASCII コード (10) になる.

key#cr
復帰を表す ASCII コード (13) になる.

wrap# 項
入出力ユーティリティのバッファ (9.2.1, 338 頁) へのメッセージ getwt/1 で得られる, wrap 項 (9.2.2, 348 頁参照) に変換する. 但し, 項中の変数は変換しない.

【例】項の変換例

```
foo          →  atom(foo)
1            →  integer(1)
[foo,1|A]    →  list([atom(foo), integer(1)|A])
foo(1, A)    →  vector({atom(foo), integer(A), A})
```

`macro#module_name`
現在コンパイル中のモジュール名アトムに展開される.

`macro#predicate`
現在コンパイル中の述語名 / アリティ / 述語内での節の順 (0 origin の整数) なる項に展開される.

【例】モジュール名, 述語名などを得るマクロ

展開前:
```
:- module test.
...
foo(1, X) :- true | X = macro#module_name.
foo(2, X) :- true | X = macro#predicate.
```

展開後:
```
:- module test.
...
foo(1, X) :- true | X = test.
foo(2, X) :- true | X = foo/2/1.
```

`macro#switch(Param, ParamList, Default)`
条件コンパイルパラメタ Param の値により, 展開される項を指定する. 詳細は, 7.3.3.6 項 (245 頁) 参照のこと.

【注意】macro#module_name, macro#predicate, macro#switch は CAL, PDSS では使用できない.

7.3.3.2 ユニフィケーション

左辺 = 右辺
左辺と右辺のユニフィケーションに展開される. ガード部, ボディ部とも使用可能.

左辺 \= 右辺
diff(左辺, 右辺) に展開される. ガード部のみ使用可能.

【注意】現在のマルチ PSI では diff/2 は構造体を引数にできない.

7.3.3.3 数式マクロ

ユニフィケーション

左辺 := 右辺
左辺と右辺のユニフィケーションに展開される. ただし, 右辺に対して整数演算のマクロ展開が行われるので, 左辺にユニファイされるのは演算結果となる. ガード部, ボディ部とも使用可能. Prolog の is に相当する.

左辺 $:= 右辺
左辺と右辺のユニフィケーションに展開される. ただし, 右辺に対して浮動小数点数演算のマクロ展開が行われるので, 左辺にユニファイされるのは演算結果となる. ガード部, ボディ部とも使用可能. Prolog の is に相当する.

【参考】多倍長整数演算について同様のマクロの導入を予定しており, オペレータとして ^= を用いる予定である.

表 7.2: 算術比較マクロ

優先度	演算子	展開後のパターン
700	`X < Y`	less_than(X,Y)
	`X > Y`	less_than(Y,X)
	`X =< Y`	not_less_than(Y,X)
	`X >= Y`	not_less_than(X,Y)
	`X =:= Y`	equal(X,Y)
	`X =\= Y`	not_equal(X,Y)
700	`X $< Y`	floating_point_less_than(X,Y)
	`X $> Y`	floating_point_less_than(Y,X)
	`X $=< Y`	floating_point_not_less_than(Y,X)
	`X $>= Y`	floating_point_not_less_than(X,Y)
	`X $=:= Y`	floating_point_equal(X,Y)
	`X $=\= Y`	floating_point_not_equal(X,Y)

算術比較

算術比較の為のマクロは >, <, =:= 等の演算子を使ったもので, ガードの組込述語の代わりに記述することができる. 組込述語と違い, マクロの両辺に対して算術演算のマクロ展開が行われるので, 比較されるのはそれぞれの演算結果となる.

比較演算子としては表 7.2 に掲げるものが用意されている.

【参考】多倍長整数演算について同様のマクロの導入を予定しており, オペレータとして ^< などを用いる予定である.

算術演算

算術演算の為のマクロは +, -, *, / 等の演算子を使ったもので, 展開は以下の部分で行われる.

- :=, $:= マクロの右辺 (Prolog における is に相当). 演算結果が左辺にユニファイされる. := が整数用, $:= が浮動小数点数用.
- 算術比較のマクロの両辺. それぞれの演算結果が比較される. $ なしが整数用, $ 付きが浮動小数点数用.
- 暗黙の引数マクロ (後述) の <=, $<= の右辺. 演算結果が左辺で指定される引数にセットされる. := が整数用, $:= が浮動小数点数用.
- ~(式), $~(式) により明示的に展開を指示した場合. 演算結果が ~(式), $~(式) に置き換えられる. ~(式) が整数用, $~(式) が浮動小数点数用.

  【例】 p(~(X+Y+1)) が add(X,Y,A), add(A,1,B), p(B) に展開される.

  【参考】多倍長整数演算について同様のマクロの導入を予定しており, オペレータとして ^~ を用いる予定である.

なお, マクロ展開時に計算可能な場合 (入力引数がすべて定数の場合) には, マクロ展開時に計算を行なう.

演算子としては表 7.3 に掲げるものが用意されている. 優先度の数値が小さいほど強く結合するので, 他の順で演算を行いたい場合には括弧 ( ) で囲むことにより指示する.

表 7.3: 算術演算マクロ

優先度	演算子	展開後のパターン	生成される組込述語	
			整数式内	浮動小数点数式内
500	X + Y	Z	add(X,Y,Z)	floating_point_add(X,Y,Z)
	X - Y	Z	subtract(X,Y,Z)	floating_point_subtract(X,Y,Z)
	- X	Z	subtract(0,X,Z)	
	X \/ Y	Z	or(X,Y,Z)	
	X /\ Y	Z	and(X,Y,Z)	
	X xor Y	Z	exclusive_or(X,Y,Z)	
400	X * Y	Z	multiply(X,Y,Z)	floating_point_multiply(X,Y,Z)
	X / Y	Z	divide(X,Y,Z)	floating_point_divide(X,Y,Z)
	X << Y	Z	shift_left(X,Y,Z)	
	X >> Y	Z	shift_right(X,Y,Z)	
300	X mod Y	Z	modulo(X,Y,Z)	
項の形で	\(X)	Z	complement(X,Z)	
	int(F)	I	floating_point_to_integer(F,I)	
	float(I)	F		integer_to_floating_point(I,F)

【注意】\(X) は将来の PIM では exclusive_or(X,-1,Z) に変更されるかもしれない.

7.3.3.4 条件分岐

ひとつの節の定義の中での条件分岐を記述する記法である. 条件分岐は

ガード -> ボディ

の形式を, セミコロンで区切っていくつでも書ける. ここで**ガード**に記述できるものは, 節のガード部に記述できるもの, **ボディ**に記述できるものは節のボディ部に記述できるものと同じである.

セミコロンで区切るものの中には, 上述の形式の他に alternatively または otherwise があってもよい. その意味は節に対するものと同様である.

【例】条件分岐マクロ

```
foo(X,Y) :- true |
    ( X=:=0 -> p(Y,Z);
      X > 0 -> q(Y,Z);
    otherwise;
      true  -> r(Y,Z) ),
    s(X,Z).
```

このプログラムは以下のようなプログラムと等価である.

```
foo(X,Y) :- true |
    '$foo/2/0'(X,Y,Z),
    s(X,Z).

'$foo/2/0'(X,Y,Z) :- X=:=0 | p(Y,Z).
'$foo/2/0'(X,Y,Z) :- X > 0 | q(Y,Z).
otherwise.
'$foo/2/0'(X,Y,Z) :- true  | r(Y,Z).
```

【注意】現在のコンパイラのプリプロセサは, 実際に上記のような展開を行っている. 述語 '$foo/2/0' はマクロ展開を行うプリプロセサによって新たに生成された述語である. 生成する述語の名前は必ず $ で始まる (一般にユーザは $ で始まるアトムを使用しない方が良い).

このマクロでは, ';'/2 の部分が述語として括り出されるが, この述語に引数として渡される変数は, ';'/2 の "外部" にある変数と, "内部" にある変数の "積集合" である.

【例】以下の例を考える.

```
p(A, B, C, D) :- true |
   q(A, D),
   ( A =:= 0 -> r1(B, E), r2(E, C);
   otherwise;
     true -> r3(B, E), r(E, C)).
```

である. この時

- "外部" にある変数: A, B, C, D
- "内部" にある変数: A, B, C, E

従って, 受け渡される変数は, A, B, C となり, 以下のように展開される.

```
p(A, B, C, D) :- true | q(A, D), '$p/2/0'(A, B, C).

'$p/2/0'(A, B, C) :- A =:= 0 | r1(B, E), r2(E, C).
otherwise.
'$p/2/0'(A, B, C) :- true | r3(B, E), r4(E, C).
```

【注意】CAL, PDSS では条件分岐式はネストしてはならないし, 同じ節の中にふたつ以上の条件分岐式を書くこともできない.

7.3.3.5 暗黙の引数

KL1 には大域的な変数のような概念はないため, アクセスしたいデータはすべて引数として持たねばならない. このことにはプログラム部分からアクセスする可能性のある範囲が明確である, という利点を持つ半面, 記述が繁雑になりやすい欠点がある. たとえばエラーを報告するためにはそのためのストリームを持たねばならないが, すべての述語にこの引数を持たせ, プロセスを分岐するごとに各プロセスのエラー報告ストリームをマージするように記述するのは, プログラムの繁雑にし, かえって見通しを悪くする恐れがある. そこで, 多くの述語が共通して持つ引数を, 明にではなく暗黙の内に記述するためのマクロ機能が用意してある.

暗黙の引数には, モジュール全体に有効な (グローバルな) 宣言と, 一部分だけに有効な (ローカルな) 宣言がある. それぞれ以下のような形式で宣言を行う.

```
:- implicit 引数名 : タイプ { , 引数名 : タイプ , ...}.

:- local_implicit 引数名 : タイプ { , 引数名 : タイプ, ...}.
```

「引数名」は暗黙の引数を参照する際に用いる名前であり, アトムとして指定する. 「タイプ」はその引数の種類を指定するもので, それに従って述語定義中の暗黙の引数に対する操作の展開の仕方が決められる. 現在提供されているタイプには shared, stream, oldnew, string の 4 種類がある.

Shared 分岐するゴールに原則として同じデータが渡される. 各ゴールが共通に参照する値を渡すのに便利である.

Stream 分岐するゴールに, 一本のストリームにマージしていく, 分流したストリームを渡す. 各ゴールからの結果を, 結果が得られたものから順不同に集めるのに便利である.

Oldnew 展開によってふたつの引数となり, 分岐するゴールがひとつのデータの更新前後の両版を, 順次受け渡していくのに便利である.

String oldnew とほぼ同様. 構造体の場合, 要素を文字列として (oldnew はベクタとして) アクセスする点が異なる.

なお, グローバルな暗黙の引数の宣言は :- module の後で :- local_implicit 宣言または, 最初の述語定義より前に一回だけ書くことができ, モジュールの途中で変更することはできない.

ローカルな暗黙の引数の宣言は何回行ってもよい. この場合には, 以前のものは無効になり新たな宣言が有効になる. また, 引数を指定しないで,

```
:- local_implicit.
```

とだけ書いた場合には, それ以降ローカルな暗黙の引数が無いことを意味する. なお, implicit 宣言と local_implicit 宣言で同じ名前の引数を宣言することはできない.

暗黙の引数の展開を行うか, 行わないかの指定は, 節単位で行える. ある述語が暗黙の引数を持つ場合 (展開が必要) には, その節定義の:- の代わりに --> を用いる. --> で定義された節の述語には宣言されている引数が, 宣言された順に, 明に記述されている引数の前に付加される. すなわち, 引数は:

1. グローバルな暗黙の引数
2. ローカルな暗黙の引数
3. 明示された引数

の順に並べられる.

【例】暗黙引数宣言の有効範囲

```
:- module test.
:- public ...
:- implicit a:oldnew, b:shared.
   %% ここでは, 暗黙の引数は a, b のみ.
:- local_implicit d:oldnew.
   %% ここでは, 暗黙の引数は a, b, d.
:- local_implicit d:shared, e:stream.
   %% ここでは, 暗黙の引数は a, b, d, e.
   %% 引数名 d のタイプは shared に変わる点に注意.
:- local_implicit.
   %% ここでは, 暗黙の引数は a, b のみ.
```

暗黙の引数を名前で参照する場合には, 引数名の直前に前置演算子 & を付ければ良い. また中置演算子 <=, $<= 及び <<= を使った記法は引数の値を更新 (あるいはユニファイ) するためものである. ベクタやストリングの要素に対する更新 (あるいは参照) のためには “&引数名 (ポジション)” という記法が用意されている. ここでポジションは対象としているベクタやストリングの要素番号を意味する. 詳細は暗黙引数のタイプにより異なる. 以下, 個々のタイプごとにその意味及び使用例を示す.

Shared 型 引数

この型の引数は, ボディ部の述語間で同じ名前の引数を持つ場合, それらの引数は同一の値を共有する. 引数名で参照している引数の値を変更したい場合は <=, $<= を使って

```
&引数名  <= 新しい値
&引数名 $<= 新しい値
```

とする. こうすると, その節中のこれ以降 (すなわち, 節定義中で右あるいは下) で参照する値はここで更新された値になる. なお, <= の右辺は整数演算のマクロ展開が, $<= の右辺は浮動小数点数演算のマクロ展開が行われる.

【例】カウンタを引き渡していく場合

```
宣言:    :- implicit counter:shared.

展開前:  p --> true | q, r.
展開後:  p(Cnt) :- true | q(Cnt), r(Cnt).

展開前:  p --> true | q, &counter <= &counter + 1, r.
展開後:  p(Cnt) :- true | q(Cnt), add(Cnt,1,Cnt1), q(Cnt1).
```

引数の値がベクタである場合の記法としては

要素の更新

&引数名 (ポジション) <= 新しい要素
&引数名 (ポジション) $<= 新しい要素

要素の参照 (<=, $<= の左辺以外に現れた場合.)

&引数名 (ポジション)

がある. 更新はボディ部のみで可能で, 組込述語 set_vector_element/5 に展開する. また参照はその記述の出現場所により, ガード部では vector_element/3 を, ボディ部では set_vector_element/5 を用いる.

```
宣言:    :- implicit table:shared.

展開前:  p(Index, New) --> true |
             q(&table(3)), &table(Index) <= New, r.
展開後:  p(Table, Index, New) :- true |
             set_vector_element(Table, 3, Elem, Elem, Table1),
             q(Table1, Elem),
             set_vector_element(Table1, Index, _, New, Table2),
             r(Table2).
```

Stream 型 引数

ボディ部の述語間で同じ名前のこの型の暗黙引数を持つ場合, それらの引数はマージされ, ヘッド部の対応する引数に接続される. ストリームに要素を流す場合には次の様に, 流したい要素をリストで繋げ <<= の右辺に書く.

&引数名 <<= [要素 1, 要素 2, …]

【例】ストリームを引き渡していく場合

```
宣言:    :- implicit window:stream.

展開前:  p --> true | q, r.
展開後:  p(Win) :- true | Win = {In1, In2}, q(In1), r(In2).

展開前:  p --> true | &window <<= [putb("gazonk")], r.
展開後:  p(Win) :- true | Win = [putb("gazonk")|Win1], r(Win1).
```

Stream 型の暗黙引数に対する参照 (&引数名) は, 元のストリームにマージしていくような一本のストリームを表す.

【例】Stream 型の暗黙引数の明示的参照

```
宣言:     :- implicit window:stream.

展開前:   p --> q, {{ r(&window), s(&window) }}.
展開後:   p(W) :- true | W = {W1, W2, W3}, q(W1), r(W2), s(W3).
```

【注意】暗黙の引数の機構では, マージャは自動的には挿入されない. ユーザが必要な個所に (普通は最初に一回だけ) 明示する必要がある (7.4.19 ストリームサポート, 285頁参照).

Oldnew 型 引数

この型はふたつの引数の組を表すもので, Prolog の DCG (definite clause grammars) 機能に似ている. DCG ではふたつの引数は difference list を表すが, KL1 では更新前後のデータを表すような使い方などもできる点, 暗黙引数がひと組に限られない点が拡張されている.

KL1 では, 更新可能なテーブルとしてベクタを使ったような場合, 効率の面からベクタに対する参照数が常に 1 になるようにプログラムすることが多く, この型の暗黙引数はこのような用法に有用である.

Difference list として用いる場合には stream 型と同様, 以下のリストに要素を加える記法が用意されている.

```
&引数名 <<= [要素1, 要素2, … ]
```

引数の値が整数である場合には, shared 型と同様 <=, $<= を使って

```
&引数名  <= 新しい値
&引数名 $<= 新しい値
```

とすることもできる. なお, <= の右辺は整数演算のマクロ展開が, $<= の右辺は浮動小数点数演算のマクロ展開が行われる. また, 引数の値がベクタである場合の記法としては

要素の更新 (1)

```
&引数名(ポジション)  <= 新しい要素
&引数名(ポジション) $<= 新しい要素
```

要素の参照 (<=, $<= の左辺以外に現れた場合)

```
&引数名(ポジション)
```

要素の更新 (2)

```
&引数名(ポジション) <<= [要素1, 要素2, ...]
```

がある. 更新 (1, 2 共) はボディ部のみで可能で, 組込述語 set_vector_element/5 に展開する. また参照はその記述の出現場所により, ガード部では vector_element/3 を, ボディ部では set_vector_element/5 を用いる. 同一の名前を持つ引数間の接続関係は DCG と同様, 左から右, 上から下に順番に接続される. 更新 (1) と (2) の違いは (1) ではベクタのポジション番目を <=, $<= の右辺で置き換えるのに対し, (2) ではベクタのポジション番目が difference list の末尾と考え <<= の右辺のリストの要素を追加し, 新たな末尾をベクタのポジション番目にセットする, という点である.

この他, oldnew 型には現在の old 値を参照する為の次のような記法もある (カウンタをカウントアップしつつ適当な時点でその値を参照する, といった場合に便利).

&引数名(old)

【例】以下のような oldnew 型の暗黙引数宣言がされている場合の, さまざまな展開例を示す.

```
宣言:    :- implicit mutter:oldnew.
```

通常の受渡し:

```
展開前:  p --> true | q, r.
展開後:  p(Old,New) :- true | q(Old,Mid), r(Mid,New).
```

更新:

```
展開前:  p(X) --> true | q, &mutter <= X, r.
展開後:  p(Old,New,X) :- true | q(Old,_), r(X,New).
```

参照と更新:

```
展開前:  p --> true | q, &mutter <= &mutter+1, r.
展開後:  p(Old,New) :- true |
             q(Old,Mid1), add(Mid1,1,Mid2), r(Mid2,New).
```

差分リストとしての更新:

```
展開前:  p --> true | q(X), &mutter <<= [X], r.
展開後:  p(Old,New) :- true | q(Old,Mid1,X), Mid1=[X|Mid2], r(Mid2,New).
```

要素の参照:

```
展開前:  p --> true | r(&mutter(3)).
展開後:  p(Old,New) :- true |
             set_vector_element(Old,3,Elem,Elem,Mid), r(Mid,New,Elem).
```

要素の更新:

```
展開前:  p --> true | q(X), &mutter(3) <= X, r.
展開後:  p(Old,New) :- true |
             q(Old,Mid1,X),
             set_vector_element(Mid1,3,_,X,Mid2), r(Mid2,New).
```

要素の差分リストとしての更新:

```
展開前:  p --> true | &mutter(2) <<= [d_n,gee,gosh], r.
展開後:  p(Old,New) :- true |
             set_vector_element(Old,2,[d_n,gee,gosh|Cdr],Cdr,Mid),
             r(Mid,New).
```

Old 値の参照:

```
展開前:  p(X) --> true |
             X = [&mutter(old)|XX], &mutter <= &mutter+1, p(XX).
展開後:  p(Old,New,X) :- true |
             X=[Old|XX], add(Old,1,Mid), p(Mid,New,XX).
```

String 型 引数

この型は基本的に oldnew 型と同じであるが, 使用する組込述語が string_element/3 及び set_string_element/4 である点が異なる. すなわち, ベクタではなくストリングを用いる場合用の oldnew 型である.

終了処理の自動生成

暗黙の引数を持つ節が, ボディ部に展開対象となるゴール呼び出しを含まない場合には, 宣言されている引数の型に応じて次の処理を行う.

shared 型　なにもしない.
stream 型　アトム [] とユニファイ.
oldnew 型　Old と New をユニファイ.
string 型　Old と New をユニファイ.

暗黙の引数の展開制御

暗黙の引数を持つと宣言した節のボディ部から暗黙の引数を持たない述語を呼び出すためには, それらのゴールを二重の中括弧対 ({{ と}}) で囲う. これにより, 自動的な引数展開が抑制される.

```
:- module test.
:- public go/0.
:- implicit input:stream, output:oldnew, counter:shared.
...
  loop(_) --> &counter =< 0 | true.
otherwise.
  loop(A) --> true |
    &counter <= &counter-1, &input <<= [getc(X)],
    {{ check(X, &output, &counter) }},
    loop(A).
...
```

上の例では, input, output, counter の 3 個の暗黙引数がグローバルに宣言され, それぞれの型は stream, oldnew, shared である. 上記のような宣言を行うと, ネックが --> で書かれた節には全て 4 個の暗黙引数 (oldnew 型はひとつで 2 個に対応) があるものとしてマクロ展開時に変換される. 例えば上記の述語定義 loop は以下のように展開される.

```
  loop(In, Oh,Ot, Cnt, A) :- Cnt =< 0 | In=[], Oh=Ot.
otherwise.
  loop(In, Oh,Ot, Cnt, A) :- true |
    Cnt1 := Cnt-1, In=[getc(X)|In1],
    check(X, Oh,Om, Cnt1),
    loop(In1, Om,Ot, Cnt1, A).
```

{{ と }} で囲まれた述語 check/4 は暗黙の引数を持たないように展開されている. このような述語に暗黙の引数の値を渡したい場合は, 上記のように明に &引数名 を用いなければならない. Oldnew 型の output は, ふたつの引数に展開されていることに注意.

【注意】暗黙の引数には, その引数のタイプに関係なくどのような値を持たせてもよい. マクロの展開系はその記述に合わせて単純に展開するだけであり, 明らかに誤った記述 (例えば同一の引数に対してリストをユニファイしかつ要素の更新を行った) に対しても, ほとんどの場合プログラマに対して注意すら促さない.

7.3.3.6　条件コンパイル機能

KL1 の処理系 (現在開発中のものも含めて) の間には, オペレーティングシステムの用いるような低レベル機能を中心に, 細かい相違点が生じてきた. 処理系の相違によるソースコードの多重化を防ぎ, 移植性を向上させるため, 条件コンパイル機能を導入した.

ソースコード中 [3], または, コンパイラのトップレベルより与えられた条件コンパイルパラメタの値により, ソースコード中の指定箇所を有効 / 無効にすることができる. 言語 C のプリプロセッサ cpp の #define, #ifdef 等に 相当する働きをする.

【注意】条件コンパイル機能は, PDSS, CAL では使用できない.

【参考 1】この条件コンパイル機能は標準マクロプリプロセッサ (kl1_macro_expander) が実現している.

【参考 2】この条件コンパイル機能は, KL1 プログラム中でも, マクロ定義ファイル中でも利用できる.

パラメタの定義

パラメタは, 条件コンパイル機能の “条件づけ” の対象となる項である. 現在はアトムのみ認められる. パラメタは以下の方法により, その値と共に定義される. 値は任意の, 変数を含まない項である.

- ソースコード中に以下の節を記述すると, その節以降でパラメタは有効になる.

  ```
  :- define(パラメタ, 値).
  ```

  【参考】この定義は, :- include (7.2.4.2, 226頁 参照) で取り込まれるファイル中にあっても良い.

- セルフコンパイラのトップレベルで, 以下のコマンドを打ち込むことにより, パラメタは定義される (3.1.3.4, 47 頁参照).

  ```
  !define パラメタ 値
  ```

  これ以降にコンパイルされるソースプログラム中の先頭に上述の宣言を行なったのと同じ効果になる.

パラメタの “無効化”

パラメタは以下の方法により無効にすることができる.

- ソースコード中に以下の節を記述すると, その節以下でパラメタは無効になる.

  ```
  :- undefine(パラメタ).
  ```

  【参考】定義されていないパラメタを無効化しようとしても, 警告を発するだけである.

- セルフコンパイラのトップレベルで以下のコマンドを打ち込むことにより !define の効果を無効にすることができる. つまり, KL1 プログラムに :- define 節を記述すると同等の効果を生ずることを停止する (3.1.3.4, 47 頁 参照).

  ```
  ! undefine パラメタ
  ```

条件コンパイルマクロ

パラメタの状態により, ソースコード中, コンパイルの対象とする部分を決定するための指示節を記述することができる.

この節は大まかに 2 通りに分類することができる.

- :- if_XXX 〜 :- else 〜 :- end_if.
- :- switch 〜 :- case 〜 :- end_switch.

[3] Include するファイルを含む. 7.2.4.2 (226頁) 参照.

真偽による条件

真偽値による条件コンパイルを行なう :- if_XXX 系の指示節として, 以下のものが用意されている. いずれも, 以下の規則が成り立つ. また, :- else はなくてもよい.

- :- if_XXX で指定された条件が真であれば, 次の :- else, または :- end_if (その前に :- else がない場合) までが有効になる.
- :- if_XXX で指定された条件が偽であれば, :- else から :- end_if までが有効になる. :- else がなければ, :- end_if までは削除される.

具体的には以下のような記述が可能である.

:- if_equal(パラメタ, 比較値) : パラメタの値と比較値が, 単一化可能であれば真, さもなくば偽.

:- if_not_equal(パラメタ, 比較値) : パラメタの値と比較値が, 単一化可能であれば偽, さもなくば真.

:- if_defined(パラメタ) : パラメタが定義済みであれば真, さもなくば偽.

:- if_not_defined(パラメタ) : パラメタが定義済みであれば偽, さもなくば真.

【例】以下の定義がなされていた場合の展開例を示す.

```
:- define(machine, mpsi).
```

```
展開前: :- if_equal(machine, mpsi).
          p1(X) :- true | X=mpsi.
        :- else.
          p1(X) :- true | X=pim.
        :- end_if.

展開後:   p1(X) :- true | X=mpsi.
```

```
展開前: :- if_not_equal(machine, pim).
          p2(X) :- true | X=pim.
        :- else.
          p2(X) :- true | X=mpsi.
        :- end_if.

展開後:   p2(X) :- true | X=mpsi.
```

```
展開前: :- if_defined(machine).
          p3(X) :- true | X=defined.
        :- else.
          p3(X) :- true | X=undefined.
        :- end_if.

展開後:   p3(X) :- true | X=defined.
```

```
展開前: :- if_not_defined(machine).
          p4(X) :- true | X=undefined.
        :- else.
          p4(X) :- true | X=defined.
        :- end_if.
展開後:   p4(X) :- true | X=defined.
```

値による分岐

:- switch(パラメタ) は, パラメタの値と, :- case(比較値 $_1$, 比較値 $_2$ ... 比較値 $_N$) の比較値 $_{1...N}$ のいずれかが単一化可能であれば, 次の :- case または :- end_switch までが有効となる. 単一化可能な比較値を含む case が複数あれば, 全て展開の対象となる.

デフォルト条件を表すために :- case(...) のかわりに :- default を記述しても良い. これは展開候補となった case が 1 つもない時のみ展開対象となる.

【例】Switch による条件コンパイル

```
:- switch(int).
:- case(0).
  foo(X) :- X = zero.
:- case(1,2,3,4,5,6,7,8,9).
  foo(X) :- X = little_integer.
:- default.
  foo(X) :- X = big_integer.
:- end_switch.
```

パラメタに応じたマクロ展開

ユーザ定義マクロの機能 (245頁, 7.3.4 参照) に説明するユーザ定義マクロの機能を利用して, 展開結果がパラメタの値によって変わるようなマクロが, マクロモジュール pimos に用意されている. このマクロは KL1 プログラムの任意の項を書ける場所に書いて良い.

```
macro#switch(Param, ValueList, Default)
```

Param : 対象とするパラメタ

ValueList : 値と, その値にマッチした時の展開後の項の二要素を持つベクタのリスト

[{ 値 $_1$, 項 $_1$}, { 値 $_2$, 項 $_2$}, ...,{ 値 $_N$, 項 $_N$}]

Default : ValueList 内の どの値ともマッチしなかった時に展開される項.

【例】パラメタに応じたマクロ展開

```
          :- define(machine, pim).
展開前:  foo(X) :- true |
             X=macro#switch(machine, [{mpsi,10}, {pim, 20}], 30).
展開後:  foo(X) :- true | X=20.
```

【注意】このマクロは PDSS, CAL では利用できない.

7.3.4 ユーザ定義のマクロ

ユーザ定義のマクロは, 現在大別して 2 種類用意されている.

- PIMOS 2.0 版以前で利用できたマクロ. このマクロは CAL, PDSS, セルフコンパイラで用いることができ, 単純な項の置換えのみをサポートする. 以降, **旧仕様マクロ** と称する.

  旧仕様マクロは整合性維持のために用意しているもので, 今後の保守や機能拡張は新仕様マクロが中心になる. 新規のマクロ定義は新仕様に従うことが望ましい.

- PIMOS 2.5 版以上で利用できるマクロ. このマクロはセルフコンパイラでしか利用できないが, 項の置換え以外の様々な展開規則を記述することができる. 以降, **新仕様マクロ**と称する.

ここでは新仕様マクロについて述べる. 旧仕様マクロについては 7.3.5 節 (262 頁) を参照されたい.

KL1 のマクロ展開機能では, ソースプログラム中に現れるマクロ定義中に指定された展開対象の入力パターンとマッチするパターンについて, 以下のような操作を行なうよう指示できる.

置換 入力パターンを別のパターンに置き換える. この際, 元のパターン中の一部を取り出して, 展開後のパターン中に埋め込むこともできる.

挿入 入力パターンの周辺 (たとえば, パターンがゴールの引数中に現れた場合なら, その引数の前後や, その引数を含むゴールの前後など) に, 任意のパターンを挿入する (括り出し機能). この場合も元のパターン中の一部を埋め込める.

こうしたプログラムからプログラムへの変換操作をマクロ展開と呼ぶ. マクロ展開はコンパイル時に行なうので, プログラムをマクロを用いて記述しても, 展開結果を直接記述しても, 実行時の速度に変わりはない.

【注意】PIMOS ではシャープ記号 (#) を用いた以下のような形式のマクロを標準に用いている. このため, ユーザはこれらの形式のマクロを定義しない方が良い.

```
fep#???              shoen#???           builtin#???
pimos_tag#???        shell_tag#???       listener_tag#???
exception_code#???   exception_tag#???
```

ここで ??? は何らかのタームである.

7.3.4.1 マクロ展開機能の利用手順

ユーザ定義のマクロを使うには, まずマクロを定義, 登録し, 次にこれを使ってプログラムをコンパイル, あるいは解釈実行する. マクロの定義は以下のような手順で行なう.

1. ユーザは, マクロ記述言語でマクロ定義ファイルを記述し, マクロコンパイラにかける.
2. マクロコンパイラは, マクロ定義ファイルの内容を調べて, 定義されたマクロ展開を実行する KL1 プログラムを生成する.
3. 生成された KL1 プログラムをコンパイルし, PIMOS のモジュール管理プロセスに登録する.

このうち 2, 3は, セルフコンパイラのプリプロセサ機能により自動的に行なわれるので, 通常ユーザは意識する必要はない.

このようにして登録したマクロ定義の利用したコンパイルには, マクロ展開を施したいプログラム中に登録したマクロモジュール名を含む:

```
:- with_macro 適用マクロモジュール名, ... .
```

のような宣言を書いて, これをコンパイルすれば良い.

リスナでの解釈実行にマクロ展開を用いたい場合には, change_macro を用いる (65 頁, 4.1.2.4節参照).

マクロ展開は, KL1 のプログラムとして登録されているマクロ展開プログラムを dynamic link により呼び出すことにより行なわれる. このようにして呼び出されたマクロ展開プロセスは, コンパイラ, リスナ, その他のアプリケーションプログラムのサブプロセスとして動作し, 必要な展開を行なった後に, 結果がコンパイルあるいは解釈実行される.

7.3.4.2 マクロ展開の順序

マクロ展開にあたっては, 前もって定義されているマクロ展開規則に対象となる入力項を展開できるような規則があるかどうか調べ, あればその規則にしたがった展開を行なう. もし適用できる規則が複数あれば, どれかひとつを適用する (どれを適用するかは保証されない).

項の展開は, 展開結果に対してさらなる展開ができなくなるまで繰り返し行なう. それ以上の展開ができなくなった時, 展開結果が構造体であれば, その各要素に対して同様に再帰的に展開を試みる. つまり, マクロ展開は構造の浅い方から深い方へ (外側から内側へ) とトップダウンに行なわれる.

【例】 `wrap#(16#"FFFF")` なる項は,

```
        wrap#(16#"FFFF")
   →    vector({atom((#)),integer(16), string("FFFF")})
```

と展開される.

16 進数を表しているように見える“`16#"FFFF"`”という部分は, wrap# という外側のマクロの展開の結果, 形が変わってしまうので, 16 進数としての展開は起きない.

7.3.4.3 マクロ展開規則

マクロ規則は以下のような記法で記述される項の集まりである.

```
<マクロ規則>        ::=  <マクロ規則述語>
                     |   <マクロ規則述語> <マクロ規則>
<マクロ規則述語>    ::=  <マクロ規則節> '.'
                     |   <マクロ規則節> '!' <マクロ規則述語>
```

以下に各部分のシンタクスと働きを詳述する.

マクロ規則節

マクロ規則節は

展開前のパターン `=>` 展開後のパターン規則 `:-` ガード `|` ボディ

または,

`otherwise`

のように記述する.

最初の形式は, 概略「入力パターンが展開前のパターンとマッチし, ガード条件を満たすならば, ボディを実行した上で, 展開後のパターン規則を適用して展開を行なえ」という意味である (各部の詳細は後述する).

【参考】ガード, ボディは以下のような記法により省略することができる. 省略した時にはどちらも 'true' と書かれているものとして扱われる.

% ガードの省略
展開前のパターン `=>` 展開後のパターン `:-` ボディ `.`
% ガード / ボディの省略
展開前のパターン `=>` 展開後のパターン.

後者の otherwise は, これ以前のマクロ規則節が適用できない場合に限って, これ以降のマクロ規則節を適用して良いことを意味する.

【注意】'otherwise' 節は述語の最初または最後の節として記述することはできない. また, 連続して 'otherwise' 節を記述することはできない.

展開前のパターン

どのようなパターンを展開対象とするかの概略を定めるもので, この項が入力項とパターンマッチした時に, 引続きガードの実行が試みられ, ガードも成功するような規則が展開に用いられる. つまり, 展開前のパターンは通常の KL1 プログラムでいうヘッドに相当する. 以下のいずれかでなくてはならない.

- wrap された項 (9.2.2, 348 頁参照)
- アトム begin, end

アトム begin, end は特殊なパターンであり, 各々, ひとつのソースファイルの展開の開始時, 終了時に選択される (詳細 は 254頁 参照).

【参考】マクロを定義するマクロモジュールの先頭で :- with_macro pimos と宣言するか, または pimos マクロモジュールを継承するマクロモジュールを :- with_macro 宣言で指定することにより wrap# マクロを用いることができ, wrap の手間を削減することができる (233 頁参照).

【例】以下の 2 つは, pimos マクロモジュールを用いる時, 同じ意味を持つ.

```
vector({atom((#)), atom(macro), atom(mymacro)}) => ...
wrap#(macro#mymacro) => ...
```

ガード

入力項とマクロ展開前のパターンがマッチした場合に, 左から右に実行されるゴール列. 通常の KL1 のガード組込述語の他, 展開対象がどのようなものでどこに現れたかを知るために, 以下のような疑似述語を記述することができる.

- 展開対象の種別

  as_a_clause :: G

  入力項が節レベルの項の時に成功

  as_a_goal :: G

  入力項がゴールレベルの項の時に成功

  as_an_argument :: G

  入力項がゴール引数レベルの項の時に成功

- 展開対象の出現個所

  in_the_head :: G

  入力項がヘッド内にある時に成功

  in_the_guard :: G

  入力項がガード内にある時に成功

in_the_body :: G

入力項がボディ内にある時に成功

これらの出現個所に関する疑似述語は, 入力項がゴールより深いレベルのものである場合のみ成功する (節レベルである時には成功しない).

パターンがマッチし, ガードに書かれたゴールが全て成功したマクロ定義節ひとつを適用して, 展開を行なう.

展開後のパターン

展開後のパターンの記法は以下の通り

```
<展開後のパターン>   ::=   <置き換え項> <括りだし規則>
<置き換え項>         ::=   <wrap された項>
                     |     <wrap された項のリスト>
<括りだし規則>       ::=   [ 'when' <括りだし項>] [ 'where' <括りだし項 >]
                     |     [ 'where' <括りだし項 >] [ 'when' <括りだし項>]
```

入力項は「マクロ展開後の項」と置き換えられる. このとき, 「括り出し」が行なわれることがある (251 頁).

置き換え項が単に wrap された項であれば, 入力項をそのままの位置で 1 対 1 に置き換える. 置き換え項が wrap された項のリストである時には, 入力項をそのままの位置で, リストの要素を同じレベルの項としてカンマで並べたものに置き換える.

【例】1 つの節を複数の節に置き換える.

```
マクロ定義: wrap#(foo :- true | true) =>
                  [wrap#(bar1 :- true | true),
                   wrap#(bar2 :- true | true)]
                :- true | true.
展開前: foo :- true | true.
展開後: bar1 :- true | true.
        bar2 :- true | true.
```

【例】以下の 2 つのマクロは, ゴールレベルの項に対して展開される時, 同じ結果になる.

```
wrap#goal => wrap#(goal1,goal2) :- true | true.
wrap#goal => [wrap#goal1, wrap#goal2] :- true | true.
```

【例】置き換え項として [ ] を指定すれば, 入力項を消去することができる.

```
wrap#(foo) => [] :- true | true.
```

【注意】現在のマクロ展開機構では, 引数レベルの項に対してマクロ展開した時に, 展開後の項がリストであった場合は, エラーとなる.

ボディ

マクロ規則のボディ部では, 組込述語を含む通常の KL1 の述語を呼び出せ, 展開結果の生成などを行なえる.

ボディ部の述語呼び出しは, 以下のような「暗黙の引数宣言」が行なわれているものとして引数の展開を行なう.

```
:- local_implicit info:shared, stream:oldnew.
```

それぞれの引数に渡されるのは以下のようなものである.

info 現在マクロ展開を行なっている節についての情報を持つベクタ. 今のところ以下の 2 種の情報が得られる.

構造体レベル 現在展開中の項の構造体レベルを `&info` の第 0 要素に格納されているアトムで表す. 出現し得るアトムは

- `clause` : 節レベル
- `goal` : ゴールレベル
- `argument` : 引数レベル

構造体位置 現在展開中の項の位置を `&info` の第 1 要素に格納されているアトムで表す. 出現し得るアトムは

- `head` : ヘッド
- `guard` : ガード
- `body` : ボディ
- `clause` : 以上のいずれでもない. すなわち構造体レベルが clause である

【注意】このベクタは現在 2 要素だが, 将来要素が追加される可能性がある.

stream マクロ処理系が提供するプロセスと通信するためのストリーム. 以下のようなメッセージを利用できる.

error(FormatString, ArgList) 引数は putf/2 にそのまま渡され, 標準メッセージ出力に出力される. マクロ展開時にこのメッセージを 1 つでも流すと, マクロ展開に誤りがあったものと判断し, 展開結果はコンパイルの後に捨てられ, モジュールとして PIMOS に登録されることはない. マクロ展開時にこのメッセージの投げられた回数はコンパイラによりカウントされ, 標準出力に報告される.

warning(FormatString, ArgList) 引数は putf/2 にそのまま渡され, 標準メッセージ出力に出力される. それ以外には何もしない. マクロ展開時にこのメッセージの投げられた回数はコンパイラによりカウントされ, 標準出力に報告される.

info(FormatString, ArgList) 引数は putf/2 にそのまま渡され, 標準出力に出力される. それ以外には何もしない.

new_variable(^Var) 展開対象になっている節に含まれない, ユニークな wrap された変数が Var にユニファイされる.

new_atom(String, ^Atom) 16bit 文字列 String を印字名として持つアトムが Atom にユニファイされる. String に対応した Atom がなければ, 新たに生成される.

pool:keyed_bag で許されるメッセージ 暗黙の引数 stream で 表されるストリームには pool:keyed_bag で定義されるプールのプロセスが接続されており, そこにメッセージを流せば, マクロ展開に必要な任意の値の登録・参照ができる (8.3.5節「Keyed bag」 309頁参照). このプロセスの内部状態は, 展開対象となるプログラムの module ごとにシステム 初期化されるが, その他の内部状態変化に関しては完全にユーザのマクロ定義に委ねられている.

この stream マクロ引数は oldnew 型で宣言されているため, そのメッセージの送出される順は, マクロを展開する (マクロ展開規則を適用する) 順序通りになる.

以上の暗黙の引数に関しては, 通常の暗黙の引数マクロで用いられる記法により, 抑制 / 部分的な使用ができる (242 頁).

ボディで呼び出されるユーザ述語は, 明示的な指定がなければその時コンパイラに指定されたパッケージ (通常 default package) にある `:- macro_module` で指定された module にある述語を呼び出す. 明示的に指定する場合は, 通常の KL1 の呼び出し記法 (すなわち パッケージ名::モジュール名:ゴール) に従う.

マクロ規則述語

マクロ規則述語は, 概念的に 1 つのマクロパターンについて書かれているマクロ規則節の集合である.

【例】Otherwise を含むマクロ規則述語

```
マクロ定義:
   vector({atom(integerOrNot), integer(_)}) => atom(yes)
      :- true | true !
  otherwise !
   vector({atom(integerOrNot), _}) => atom(no).
適用例:
  展開前: integerOrNot(1)       展開後: yes
  展開前: integerOrNot([])      展開後: no
```

【参考】「マクロ規則述語」の概念は otherwise 機能を持ち込むために導入された. Otherwise を含まない「マクロ規則述語」は単なる「マクロ規則節」の並びと同じコードにコンパイルされる.

【例】以下の 2 つのマクロ記述は同一のコードにコンパイルされる.

- ふたつの規則節からなるマクロ規則述語ひとつによる定義.

```
vector({atom(foo), X}) => atom(bigger)
    :- X=integer(Int), Int > 1000 | true !
vector({atom(foo), X}) => atom(less)
    :- X=integer(Int), Int =< 1000 | true.
```

- ひとつの規則節からなるマクロ規則述語ふたつによる定義.

```
vector({atom(foo), X}) => atom(bigger)
  :- X=integer(Int), Int > 1000 | true.
vector({atom(foo), X}) => atom(less)
  :- X=integer(Int), Int =< 1000 | true.
```

括り出しマクロ規則

括り出しマクロ規則とは, 例えば

```
...,a(~(X+1)),... ⇒ ...,X1:=X+1,a(X1),...
```

といったように, マクロ展開の結果をマクロ対象項のあった位置と異なる構造体レベルに出現させるために用いる記法である.

括り出しマクロの記法　括り出しマクロのために以下のようなオペレータを用いる.
when:　この後に続く項を, マクロ展開の対象となった項の左方 (前) に挿入する
where:　この後に続く項を, マクロ展開の対象となった項の右方 (後) に挿入する
この括り出しマクロ用のオペレータの後に続く括り出される項は, wrap された項でなくてはならない.

【例】括り出しの例

```
マクロ展開規則: wrap#a => wrap#b when wrap#g1 where wrap#g2.
展開前: foo(a)
展開後: g1, foo(b), g2
```

但し, ヘッド中の引数を展開する場合には, 以下のルールを適用する.
when:　この後に続く項を, ガードの右から (末尾に) 挿入する
where:　この後に続く項を, ガードの左から (先頭に) 挿入する

括り出しのレベル　括り出し機能は, ゴール引数レベル / ゴールレベルの項にのみ適用可能である. 括り出される項は必ず, ゴールレベル[4]の項になる. 節レベルの項に対して括り出しの記法を持つ定義で展開を行なおうとすると, エラーを生ずる.

【参考】以下の 2 つの規則は, ゴールレベルの項に対して適用された時には同じ働きをする.

```
wrap#(macro#a) => wrap#(macro#b, macro#c).
wrap#(macro#a) => wrap#(macro#b) where wrap#(macro#c).
```

節レベルの展開の前処理

マクロ展開時に, マクロ展開器は, まず, 節レベルの項に対して以下のような標準形に展開してから, マクロ展開を行なう.

ヘッド :- ガード | ボディ.

例えば, 展開の対象となる KL1 ソースプログラム中に節レベルで

```
foo.
```

と書かれていても, まず,

```
foo :- true | true.
```

と, ガード / ボディを補ってからマクロ展開を行なう. このため, アトム foo を展開前パターンとするようなマクロ展開規則は, 節レベルの展開に用いられることはない.

展開後のパターン中の変数

展開後のパターンはそのままコンパイラに渡るため, 未定義変数を含まないグラウンド値でなくてはならず, 変数は全て wrap された形 (variable(文字列) または variable(整数)) になっている必要がある. もし, wrap されていない変数がコンパイラに渡ると, コンパイラの入力フィルタが永久に中断する (GC 時に deadlock として報告される).

展開前のパターンに単一化される項は展開対象のソースプログラム中の項なので, 変数はすべて wrap 済みであるが, 展開後のパターン中にのみしか出現しない変数はそのままでは具体化されていない変数となってしまい, 適当な wrap された変数と単一化してやらねばならない.

この問題に対してマクロ言語処理系では, wrap された変数の自動的な生成と, マクロ定義中での明示的な生成の, 両手段を用意している.

[4] ガード内のゴール引数であればガードゴール, ボディないのゴール引数であればボディゴール. また, ヘッド引数の場合は, ガードゴールに展開される.

自動生成 展開後のパターン中にのみ存在し, 展開前のパターン / ガード / ボディいずれにも出現しない変数は, マクロ言語処理系で自動的にユニークな wrap 済み変数に置き換えるコードを出す.

【例】

```
wrap#a => wrap#(b(X), c(X)).
```

変数 X は展開後パターン中でのみ存在するので, マクロ言語処理系で X を wrapped 項と unify するようなコードを出力する.

【参考】この規則があるため, 旧仕様ユーザ定義マクロの記法は, 展開対象 / 展開後パターンを wrap 項とするのみで使用することができる.

【例】旧仕様から新仕様への変換例

- 旧仕様記法

```
builtin#set_substring(S1,P1,L,S2,NewS1)
  => builtin_utility:copy_string_elements(S1,P1,L,S2,0,NewS1,_).
```

- 新仕様記法

```
% 無名変数 _ は自動的に wrap 項に具体化される.
wrap#builtin#set_substring(S1,P1,L,S2,NewS1)
  => wrap#builtin_utility:
        copy_string_elements(S1,P1,L,S2,0,NewS1,_).
```

【参考 1】現在のコンパイラでは, 旧仕様マクロ定義記法で書かれたマクロ定義ファイルもコンパイルすることができる.

【参考 2】旧仕様マクロ定義記法で書かれたマクロ定義ファイルは, マクロコンバータプログラムを用いることにより新仕様マクロ定義記法で書かれたマクロモジュールに変換することができる (263 頁参照).

明示的な生成 マクロ展開規則から呼び出されるユーザ定義の KL1 述語で, 明示的に wrap した変数を作ってユニファイすることもできる. ユニークな wrap した変数を得るには, 暗黙の引数 &stream に new_variable/1 メッセージを送れば良い.

【例】以下のマクロ定義は上にあげた例と等価である.

```
% 変数 ExpandedTerm はボディに出現するため
% マクロ言語処理系では自動的には変数を作らない.
wrap#a => ExpandedTerm
  :- true | pred(ExpandedTerm).

% &stream に new_variable/1 メッセージを投げる.
pred(ExpandedTerm) --> true |
  &stream <<= [new_variable(X)],
  ExpandedTerm = wrap#(b(X),c(X)).

% 暗黙の引数の展開を行なうと, 以下のようになる.
pred(_, Stream0, Stream, ExpandedTerm) :- true |
  Stream0=[new_variable(X)|Stream],
  ExpandedTerm = wrap#(b(X),c(X)).
```

特殊な展開前のパターン

Wrap されていない, アトム begin, end を展開前のパターンとして記述すると, ファイル毎のマクロ展開の開始時 / 終了時に展開を試みる.

このパターンは各々, 同一のマクロモジュール内で複数回出現してはならない. また, 展開された結果は無視され, 展開後のプログラムに取り込まれることはない. すなわち, &stream に対して行なわれた操作のみが有効となる.

【例】マクロ展開のために必要なプールプロセス (以下の例では keyed_sorted_set) を, マクロ展開の開始時に生成し, マクロで提供する keyed_bag に登録する. マクロ展開の終了時には, そのプールへのストリームを閉じる (ストリームを閉じないと deadlock となる).

```
begin => [] :- true |
  pool:keyed_sorted_set(Stream),
  &stream <<= [put(my_pool, Stream)].
end => [] :- true |
  &stream <<= [get(my_pool, [])].
```

なお, 継承が行なわれている場合には, 以下の順で begin/end 特殊パターンの展開を試みる.

begin : “子” から “親” の順で展開を試みる.

end : “親” から “子” の順で展開を試みる.

マクロ定義ファイル中の KL1 述語定義

マクロ定義ファイル中には KL1 の述語も定義することができる. このマクロ定義ファイル中に記述された KL1 述語については, 同一のファイルに書かれたマクロ定義中のボディから, 特にパッケージ名 / モジュール名の指定をせずに呼び出すことができる.

【注意】マクロ定義ファイル中に定義されている KL1 プログラムに対しては, 同一ファイル内で宣言されているマクロは適用されない. 現在のところ, そのプログラムに対して適用されるマクロを同一ファイル中に記述することはできない[5].

このマクロ定義ファイル中に書かれた KL1 述語群は, マクロモジュール宣言 (:- macro_module) で指定されたアトムを名前とする module の一部となる. つまり, マクロをコンパイルした結果生成される KL1 述語と同じモジュールに入る. また, このモジュールは defualt package に登録される. これらの述語は陽にユーザが宣言しない限り local 述語となる (陽に宣言するためには :- public 宣言を記述すれば良い).

7.3.4.4 マクロモジュールの構成

PIMOS/ マクロプリプロセッサに 1 単位として扱われる, マクロ展開規則の集合をマクロモジュールと呼ぶ. ひとつのマクロモジュールは, 実際に展開に用いる前に KL1 に変換され, ひとつの KL1 のモジュールとして扱われる.

マクロモジュールのソースプログラムは以下の 2 部分よりなり, ソースプログラム中にもこの順に書かれている必要がある.

- 宣言部
- マクロ規則記述部

[5] 将来的には導入したいが, その時でも両者を共存させることは難しいと思われる.

マクロ規則記述部には, マクロ展開規則 (247 頁, 7.3.4.3 節参照) の他に, 通常の KL1 プログラムを記述することができる. これらはマクロ定義のボディから呼び出し, マクロ展開のための補助的な役割を負わせることができる (249 頁参照).

以下, 本節ではマクロモジュールの宣言部について詳述する.

マクロモジュール宣言

マクロモジュールは PIMOS には通常の KL1 のモジュールとして扱われるが, 本宣言はそのモジュール名アトムを指定する. この宣言はマクロ定義ファイルの先頭で行なわれる必要がある. マクロモジュール宣言は以下のような記法を用いる.

```
:- macro_module 'マクロモジュール名'.
```

マクロの定義は, この宣言で指定されたアトムをモジュール名とする KL1 プログラムを PIMOS に登録することによって行なわれる.

適用マクロモジュール宣言

マクロモジュールのソースプログラムを KL1 プログラムに変換する際にも, マクロ展開を行なうことができる. そのマクロ展開時に用いるマクロモジュール名を, この宣言で指定する.

```
:- with_macro '適用マクロモジュール名', '適用マクロモジュール名', ...
```

マクロ展開時には, ここで並べられた順 (左 → 右) にマクロモジュールを適用する.

適用マクロモジュール宣言がないマクロモジュールの KL1 プログラムへの変換には, デフォルトとしてマクロモジュール pimos を用いる. マクロ展開をいっさい適用したくない時には

```
:- with_macro.
```

と記述する.

【注意】現在の仕様ではこの宣言は 1 マクロモジュール内 1 つしか記述できない.

【注意】同一のマクロモジュール内で記述されているマクロは, そのマクロモジュール自身を KL1 に展開する過程で使うことはできない.

【例】同一マクロモジュール中にあるマクロ展開規則は適用されない.

```
:- macro_module test.
:- with_macro pimos.

% macro#this を that に展開するマクロ
wrap#(macro#this) => atom(that)
    :- true | true.
...
% macro#this は展開されない
f(macro#this) :- true | true.
```

【例】以下のように複数マクロモジュール / 別ファイルに分割すれば, マクロ展開規則を適用できる.

```
:- macro_module macro_test.
:- with_macro pimos.

% macro#this を that に展開するマクロ
vector({atom((#)), atom(macro), atom(this)}) => atom(that)
    :- true | true.
```

別のファイル:

```
:- module test.
:- with_macro macro_test.

% f(that) :- true | true. に展開される
f(macro#this) :- true | true.
```

継承マクロモジュール宣言

展開対象の項が当該マクロモジュール中の対象パターンに 1 つもマッチングしない時に, 他のマクロモジュール中の規則を用いて展開を試みるように指示できる. これをマクロモジュールの継承という. 継承マクロモジュール宣言は, 引続き試みられるべきマクロモジュールを宣言する.

```
:- inherit_macro '継承マクロモジュール名'.
```

マクロの継承は単一継承だけが可能で, ひとつのマクロモジュール宣言 1 つに対して継承マクロモジュール宣言を複数書くことはできない. この宣言が存在しないマクロモジュールはデフォルトで pimos を継承する. いかなるマクロモジュールも継承させたくない場合は, 以下のように宣言する.

```
:- inherit_macro.
```

宣言部にはこれらの宣言の他に public 宣言などを, 通常の KL1 モジュールと同様に記述することができる.

7.3.4.5 KL1 プログラムへのコンパイル

以上に述べたマクロ規則は, 以下のように KL1 述語へコンパイルされる.

コンパイル後のヘッド コンパイル後のヘッドは以下のような項 (expand_macro/6) になる.

```
:- local_implicit info:shared, stream:oldnew.
expand_macro(MacroCode, InputPattern, ^OutputPattern)
```

暗黙の引数があるので, 実際には 6 引数になる.

- `info`, `stream`: 7.3.4.3(249 頁) 参照
- `MacroCode`: 展開対象のプログラムの中で直接 `:- with_macro` 宣言で参照されているマクロモジュールの expand_macro/6 述語へのポインタ (マクロ展開を繰り返す際の出発点)
- `InputPattern`: "展開前のパターン" として書かれた項
- `OutputPattern`: マクロ展開後のパターンがユニファイされる出力引数

コンパイル後のガード マクロ規則中にガードに書かれた述語群は, そのまま展開後の節のガードになる. 但し, 7.3.4.3 節 (248 頁) に説明した疑似述語については, 以下のように展開される.

```
as_a_clause     vector_element(&info, 0, clause)
as_a_goal       vector_element(&info, 0, goal)
as_an_argument  vector_element(&info, 0, argument)
in_the_head     vector_element(&info, 1, head)
in_the_guard    vector_element(&info, 1, guard)
in_the_body     vector_element(&info, 1, body)
```

コンパイル後のボディ ユーザの書いたボディが暗黙の引数宣言

```
:- local_implicit info:shared, stream:oldnew.
```

がなされているように展開される. また, 展開後パターン中で when, where で指定されている括り出される項は, 適当なメッセージで &stream に送出される.

展開後パターンにのみ出現する変数については

```
&stream <<= [new_variable(変数),.....]
```

なるゴールを自動的に付加することにより, マクロ展開時に具体化を行なう.

展開後の項の取り扱いについては, 指定された項が wrap 項である場合と, wrap されていないリスト (または空リスト) の場合とで異なる.

- 展開後の項として wrap 項を指定した場合
  引数 MacroCode (展開対象のプログラムに直接 :- with_macro 宣言で参照されているマクロモジュールの expand_macro/6 述語へのポインタ, 256 頁参照) で示されるコードを, 以下のような引数で再帰的に呼ぶ.

  ```
  {&info, &stream, MacroCode, マクロ展開後パターン, ^OutputPattern}
  ```

  この呼び出しによって, 展開できる限りマクロ展開を繰り返すことになる.

- 展開後の項として, リストまた空リストを指定した場合
  出力の変数 (上記 OutputPattern) に展開後の項がそのままユニファイされる (その後, 呼びだし側がリストの要素について繰り返し展開を行なう).

継承, 展開終了のための節 全てのマクロ規則に対して以上の様なコンパイルがなされた後に, 以下のような otherwise をともなった節が出力される

- 継承するマクロがある場合

  ```
  otherwise.
    expand_macro(MacroCode, InputPattern, OutputPattern) -->
      true |
      '継承マクロモジュール名':
         expand_macro(MacroCode, InputPattern, OutputPattern).
  ```

  これは, 当該マクロモジュールに適当な展開規則がなかった場合に, 継承するマクロモジュールで展開を試みることに対応する.

- 継承するマクロがない場合

  ```
  otherwise.
    expand_macro(_, InputPattern, OutputPattern) --> true |
      InputPattern = OutputPattern.
  ```

  これは, 当該マクロモジュールに適当な展開規則がなく, 継承するモジュールもない場合には, 入力項がそのまま展開結果になることに対応する.

【注意】上述のように, マクロの継承は静的なリンクで実現されている. したがって, 継承関係上位のマクロモジュールを更新した時には, 再リンカによりマクロモジュール間のリンクも更新する必要がある (3.2 再リンカ, 55 頁参照).

【例】以下のように宣言されている 2 つのマクロモジュールを考える.

1. 上位のモジュール

```
:- macro_module god.
:- inherit_macro .
...
```

2. 下位のモジュール

```
:- macro_module human.
:- inherit_macro god.
...
```

この 2 つのモジュールの内 god モジュールのみを更新しても, human モジュールは更新以前の got モジュールを参照している. このため, human モジュールより継承によって参照されるモジュールの内容は, 更新以前のものである.

このリンクを新しくするためには, シェルなどから,

```
relink(human, god)
```

と打ち込み, 参照関係を更新する必要がある.

7.3.4.6 旧仕様マクロの取り扱い

PIMOS 2.0 版以前, および, クロスコンパイラで利用できる旧仕様マクロ記法 (7.3.5 節, 262 頁) は, 新仕様マクロコンパイラでも使用できる.

マクロコンパイラは, ":- user_macro マクロモジュール" なる宣言を見つけると, それ以降については旧仕様マクロ定義記法で記述されていると判断する. すなわち, 以下のような変換を行なった後に本章で説明した方法によりコンパイルを行なう.

- :- user_macro は, :- macro_module に置き換えられる. すなわち, :- user_macro で指定されたアトムを名前とするモジュールを生成する.
- 以下の 2 つの宣言を付け加える.

  :- with_macro pimos.

  :- inherit_macro pimos.

- '=>'/2 の引数を wrap する.
- 上記以外については何もしない.

また, 変換プログラム (263 頁) を用いることにより, 新仕様マクロ定義記法に置き換え, ファイルに保存することも可能である.

7.3.4.7 マクロ関連ツール

現在, マクロを使用するにあたり, 以下のようなツールを使用できる.

- マクロ定義記法変換プログラム
- マクロ継承関係チェッカ

マクロ定義記法変換プログラム 以前「ユーザ定義マクロ」として暫定的に提供されて来た, 旧仕様の記法で書かれているマクロ定義を, 本章で解説した新仕様マクロ記法に変換するプログラムである. 詳細は 262 頁, 7.3.5 節を参照.

マクロ継承関係チェッカ マクロモジュール間の継承関係を標準出力に表示する．継承関係にループがある時にはその旨表示する．

macro(EntryModule)

EntryModule にはマクロの継承関係を調べたいもっとも低位のマクロモジュール名をアトムで指定する．

【例】

```
SHELL> macro(mymacro)
mymacro --> pimos
```

上の表示は, mymacro モジュールは pimos モジュールを継承していることを表す．

7.3.4.8 マクロ定義の例

pimos.mac

pimos.mac 中で定義されているマクロの一部をあげる．

```
:- macro_module pimos.
% マクロ展開は行なわない.
:- with_macro .
% マクロの継承は行なわない.
:- inherit_macro .

%
% atomic データから atomic データへの変換
%    このファイル自身についてはマクロ展開を行なうことができないため
%    展開前のパターン中にはマクロは記述できない.
%    展開後のパターンは，このマクロ定義が参照される時に
%    動的 / 再帰的に展開されるので pimos.mac 中で定義されているマクロを
%    使用しても構わない.

% pimos_tag#task -> 16#"100"
vector({atom((#)), atom(pimos_tag), atom(task)}) =>
        vector({atom((#)), integer(16), string("100")}).
% pimos_tag#pimos -> 16#"200"
vector({atom((#)), atom(pimos_tag), atom(pimos)}) =>
        vector({atom((#)), integer(16), string("200")}).
%
% 展開パターン中に変数が出る場合.
%
% builtin#set_substring(S1,P1,S2,NewS1)
%   => string(S2,L,_,S22),
%     builtin_utility:copy_string_elements(S1,P1,L,S22,0,NewS1,_).
vector({atom(#}, atom(builtin),
        vector({atom(set_substring), S1, P1, S2, NewS1})}) =>
    vector({atom((',')),
            vector({atom(string), S2, L, _, S22}),
```

```
            vector({atom((:)), atom(builtin_utility),
                   vector({atom(copy_string_elements),
                           S1, P1, L, S22, integer(0), NewS1, _})})}) .
%
% 算術演算マクロ
%
% 引数レベルで ~(算術演算) が出現したら，新しい変数 Z に置き換え
% '相当する組込'(~X, ~Y, Z) を括り出す.
% 引数の，~X, ~Y は再帰的に変換される.
% 算術演算以外のものが ~(...) 中にあった時には，~(...) 中のものに変換する.
vector({atom((~)), vector({atom(+), X, Y})}) => Z
        when vector({atom(add), vector({atom((~)), X}),
                             vector({atom((~)), Y}), Z})
  :- as_an_argument | true !
vector({atom((~)), vector({atom(-), X, Y})}) => Z
       when vector({atom(subtract), vector({atom((~)), X}),
                                  vector({atom((~)), Y}), Z})
  :- as_an_argument | true !
vector({atom((~)), vector({atom(int), X})}) => Z
       when vector({atom(floating_point_to_integer),
% floating 計算に変換
                    vector({atom(($~)), X}), Z})
  :-as_an_argument | true !
otherwise !
vector({atom((~)), X}) => X :- as_an_argument | true.
%
% ゴールトップレベルにある
%      X := 算術演算
% を展開する.
%
vector({atom((:=)), X, Exp}) =>
    vector({atom((=)), X, vector({atom((~)), Exp})})
  :- as_a_goal | true.
```

標準入力, 標準出力に変換されるマクロ

マクロ定義

```
  :- macro_module stdio.
  :- with_macro pimos.
  :- inherit_macro pimos.

  wrap#(stdio#stdin) => Stream
      when wrap#(shoen:raise(pimos_tag#shell, get_std_in, Stream))
      :- as_an_argument, in_the_body | true.
  wrap#(stdio#stdout) => Stream
      when wrap#(shoen:raise(pimos_tag#shell, get_std_out, Stream))
      :- as_an_argument, in_the_body | true.
```

使用例

ソースプログラム:

```
:- module stdio_test.
:- public go/0.
:- with_macro stdio.

go :- true |
    stdio#stdin = [getwt(Command)|StdIn],
    commandLoop(Command, StdIn, [], stdio#stdout, []).
...
```

展開結果:

```
:- module stdio_test.
:- public go/0.

go :- true |
    shoen:raise(16384, get_std_in, GenVar0),
    GenVar0=[getwt(Command)|StdIn],
    shoen:raise(16384, get_std_out, GenVar1),
    commandLoop(Command, StdIn, [], GenVar1, []).
...
```

算術計算

マクロ定義

```
:- macro_module arithmetic.
:- with_macro pimos.
:- inherit_macro pimos.

wrap#($~(sin(X))) => Y
%% 先に wrap# が展開されるため $~ はマクロコンパイル時には展開されない.
    when wrap#(arithmetic_library:sin($~(X), Y))
    :- in_the_body | true .

% 定数であればマクロ展開時に計算
wrap#(sin(X)) => floating_point(Y)
    :- X = floating_point(Float), as_an_argument, in_the_body |
  {{arithmetic_library:sin(Float, Y)}} !
otherwise !
wrap#(sin(X)) => Y
    when wrap#(arithmetic_library:sin(($~(X)), Y))
    :- as_an_argument, in_the_body | true.
```

使用例

ソースプログラム:

```
:- module arith.
:- public calc/2.
:- with_macro arithmetic.

calc(X, Y) :- true | Y $:= sin(X) * sin(X).
```

展開結果:

```
:- module arith.
:- public calc/2.

calc(X, Y) :- true |
  arithmetic_library:sin(X, GenVar0),
  arithmetic_library:sin(X, GenVar1),
  floating_point_multiply(GenVar0, GenVar1, GenVar2),
  Y = GenVar2.
```

7.3.5 旧仕様マクロ

【注意】本節に述べるマクロ機能は, PIMOS 2.0 版までで用意していた機能であり, 2.5 版以降では 7.3.4 (245 頁) に述べる新仕様のマクロに機能的に包含されている. 新規のマクロ定義は新仕様に従うことが望ましい. 旧仕様のマクロを新仕様に変換するためのプログラムも用意されている (263 頁).

旧仕様マクロでは, ソースプログラム中のあるパターンを別のパターンに置き換えるマクロを定義することができる. このマクロを定義するためには, 以下のような形式のマクロ定義ファイルを作らなければならない.

```
:- user_macro マクロ定義名.
マクロ名 1# マクロ名 2  => 展開後の形式.
...
```

なお, 左辺のパターンは必ず xxx#yyy のように # を使ったもので, マクロ名 1 はアトムでなければならない.

【例】

```
:- user_macro io.
   io_status#normal      => 0.
   io_status#read_error  => 1.
   io_status#write_error => 2.
   io_command#read(X)    => io(read,X).
   io_command#write(X)   => io(write,X).
```

このマクロをソースプログラム中で使用する場合には, モジュール定義の先頭で以下のように宣言しなければならない.

```
:- with_macro マクロ定義名.
```

ここで, マクロ定義名はアトムである.
ソースプログラム中に => の左側のパターンを書くと, コンパイル時に右側のパターンに展開される.

【参考 1】マルチ PSI 用のクロスコンパイラシステム (CAL) を使用する場合は, マクロを使用している KL1 のソースプログラムをコンパイルする前に, マクロ定義ファイルをロード (LMac コマンド) しなければならない. 詳しくは, CAL の説明書を参照のこと.

【参考 2】セルフコンパイラでは, マクロ定義名はマクロの展開方法が記述してある KL1 のモジュールであると解釈し参照される. 詳細は 7.3.4 節 (245頁) を参照のこと.

【参考 3】PIMOS 3.0 版以降の PIMOS が提供する新仕様マクロ機能では, この旧仕様のマクロも扱うことができる.

【注意】PIMOS 2.1 版のセルフコンパイラで提供していた環境変数 (macro:auto_initialize, macro:initial_file) によりマクロファイルを読み込む機能は, 新仕様マクロによる機能拡張のため不要になったので, 廃止された.

旧仕様のマクロを新仕様に変換するプログラム convert_macro が用意されている. このプログラムは以下の 2 点について変換を行なう.

- `:- user_macro` を `:- macro_module` に変換する.
- 展開前のパターン / 展開後のパターンを wrap する.

convert_macro(OldFile, NewFile)

OldFile: 旧仕様記法で記述されたマクロファイル名を 16bit ストリングで入力する.

NewFile: 新仕様記法に変換されたマクロファイルを出力すべきファイル名を 16 bit ストリングで入力する.

【注意】以前の版とモジュール名が変更された.

7.4 組込述語

ここでは マルチ PSI および, PIM/m で用意されているな組込述語について述べるが, 特に断りがないものは他の PIM の仕様も共通である. 各組込述語は次の形式で示されている.

$$\underline{\mathbf{vector(X,\ \hat{}Size)}} :: \underline{\mathbf{G}}$$

↑ 呼出し形式　　↑ 記述可能な場所

記述可能な場所: 組込述語はその性質により記述できる場所が制限されている. 記述可能な場所が **G** であればガード部でのみ記述できる述語を意味し, **B** であればボディ部でのみ記述可能であることを意味する. また, **GB** はガード部でもボディ部でも記述可能であることを意味する.

引数のモード: ^ の付いている引数はユニファイされる引数 (以下では出力引数と呼ぶこともある) であることを意味する. それ以外の引数は入力引数である. 入力引数が具体化されていない場合, 組込述語の出現場所がガード部ならそのクローズを用いた実行は中断され, 他のクローズが試される. ボディ部なら, その組込述語の実行が中断される (同じボディ部中の他のゴールの実行には影響を及ぼさない).

「失敗 / 例外」とあるものはガード部であれば失敗, ボディ部であれば例外として扱われることを意味する.

なお, 算術演算については, 直接組込述語で記述せずとも, 簡略に記述できるマクロが用意されている. 詳細は 7.3.3.3 数式マクロ (234 頁) の項を参照のこと.

7.4.1 タイプチェック

wait(X) :: G

X のトップレベルが具体化していれば成功

中断: X のトップレベルが未定義の場合

失敗: なし

atom(X) :: G

X がアトムであれば成功

中断: X が未定義の場合

失敗: X がアトム以外の場合

integer(X) :: G

X が整数であれば成功

中断: X が未定義の場合

失敗: X が整数以外の場合

floating_point(X) :: G

X が浮動小数点数であれば成功

中断: X が未定義の場合

失敗: X が浮動小数点数以外の場合

list(X) :: G

X がリストであれば成功

中断: X が未定義の場合

失敗: X がリスト以外の場合

vector(X, ^Size) :: G

ベクタ X のサイズを Size とユニファイする

中断: X が未定義の場合

失敗: X がベクタ以外の場合

string(X, ^Size, ^ElementSize) :: G

ストリング X のサイズを Size とユニファイし要素サイズを ElementSize とユニファイする

中断: X が未定義の場合

失敗: X がストリング以外の場合

module(X, ^Size) :: G

モジュール X のサイズを Size とユニファイする

中断: X が未定義

失敗: X がモジュール以外の場合

code(X, ^Arity, ^Module, ^Position) :: G

コード X の引数個数を Arity とユニファイし, そのコードが属するモジュールを Module とユニファイし, モジュールの先頭からの位置を Position とユニファイする.

中断: X が未定義の場合

失敗: X がコード以外の場合

unbound(X, ^Result) :: B

X が現時点 (すなわち unbound/2 が実行された時点) で未定義変数ならば, Result に {Id, Address, NewX} なるベクタをユニファイする. ここで, Id は変数ごとにつく識別子, Adress2 は変数のアドレス, NewX は X をユニファイしたものである. X が定義済みの場合は Result に一要素のベクタ {NewX} をユニファイする.

中断, 例外: なし

【注意】並列言語である KL1 では逐次型言語の Prolog などと異なり, 変数が未定義のままであることを認識することには本来意味がない. 未定義であると認識した直後に (並行して走る別のプロセスにより) 既に未定義ではなくなっているかも知れないからである. したがって, 本述語の使用には十分に注意を払う必要がある.

7.4.2 diff

diff(X, Y) :: G

X と Y が同じ値でないことが確定すれば成功.

【注意】X または Y が未定義変数以外のアトミックな項 (1 ワードで表現されるデータ, すなわち整数, アトム, または空ベクタ) でなければならない.

中断: X または Y が未定義の場合

失敗: X と Y がまったく同じ値であることが確定した場合

通常は以下のマクロ表記を用いる.

```
X \= Y  =>  diff(X, Y).
```

PIM/m や マルチ PSI では

```
p(X, X) :- q(X).
p(X, Y) :- X\=Y | r(X, Y).
```

のようなプログラムは:

```
    p(X, X) :- q(X).
otherwise.
    p(X, Y) :- r(X, Y).
```

と書くことが望ましい.

PIM/m マルチ PSI 以外では逆に otherwise を使わない方が最適なコードの出るものがあり, アトミックな項以外でも使えるものがある.

また, X または Y が未定義変数またはアトミックな項でなければ, CSP ウインドウに Illegal DIFF !!! : なるメッセージがでる. そこで s を入力すると success し, f を入力すると fail し, b を入力すると, break して CSP のトップレベルにもどる.

7.4.3 整数の比較

less_than(Integer1, Integer2) :: G

Integer1 < Integer2 であれば成功

中断: Integer1 または Integer2 が未定義の場合

失敗:

- Integer1 または integer 2 が整数でない場合
- 上記成功条件を満たさない場合

通常は以下のマクロ表記を用いる.

```
X < Y  =>  less_than(X, Y).
X > Y  =>  less_than(Y, X).
```

not_less_than(Integer1, Integer2) :: G

Integer1 ≥ Integer2 であれば成功

中断: Integer1 または Integer2 が未定義の場合

失敗:

- Integer1 または integer 2 が整数でない場合
- 上記成功条件を満たさない場合

通常は以下のマクロ表記を用いる.

```
X >= Y  =>  not_less_than(X, Y).
X =< Y  =>  not_less_than(Y, X).
```

equal(Integer1, Integer2) :: G

Integer1 と Integer2 が等しければ成功

中断: Integer1 または Integer2 が未定義の場合

失敗:

- Integer1 または integer2 が整数でない場合
- 上記成功条件を満たさない場合

通常は以下のマクロ表記を用いる.

```
X =:= Y  =>  equal(X, Y).
```

not_equal(Integer1, Integer2) :: G

Integer1 と Integer2 が等しくなければ成功

中断: Integer1 または Integer2 が未定義の場合

失敗:

- Integer1 または integer 2 が整数でない場合
- 上記成功条件を満たさない場合

通常は以下のマクロ表記を用いる.

```
X =\= Y  =>  not_equal(X, Y).
```

7.4.4 整数の演算

add(Integer1, Integer2, ^NewInteger) :: GB

Integer1 に Integer2 を加えた値を NewInteger にユニファイする

中断: Integer1 または Integer2 が未定義の場合

失敗 / 例外:

- Integer1 または integer 2 が整数でない場合
- オーバフローした場合

通常は以下のマクロ表記を用いる.

```
Z := X + Y  =>  add(X, Y, Z).
```

subtract(Integer1, Integer2, ^NewInteger) :: GB

Integer1 から Integer2 を引いた値を NewInteger にユニファイする

中断: Integer1 または Integer2 が未定義の場合

失敗 / 例外:

- Integer1 または integer 2 が整数でない場合
- オーバフローした場合

通常は以下のマクロ表記を用いる.

```
Z := X - Y  =>  subtract(X, Y, Z).
Z := - X    =>  subtract(0, X, Z).
```

multiply(Integer1, Integer2, ^NewInteger) :: GB

Integer1 に Integer2 を掛けた値を NewInteger にユニファイする

中断: Integer1 または Integer2 が未定義の場合

失敗 / 例外:

- Integer1 または integer2 が整数でない場合
- オーバフローした場合

通常は以下のマクロ表記を用いる.

```
Z := X * Y  =>  multiply(X, Y, Z).
```

divide(Integer1, Integer2, ^NewInteger) :: GB

Integer1 を Integer2 で割った商を NewInteger にユニファイする

中断: Integer1 または Integer2 が未定義の場合

失敗 / 例外:

- Integer1 または Integer2 が 整数でない場合
- Integer2 が 0 である場合
- オーバフローした場合[6]

通常は以下のマクロ表記を用いる.

```
Z := X / Y  =>  divide(X, Y, Z).
```

modulo(Integer1, Integer2, ^NewInteger) :: GB

Integer1 を Integer2 で割った余りを NewInteger にユニファイする

中断: Integer1 または Integer2 が未定義の場合

失敗 / 例外:

- Integer1 または Integer2 が 整数でない場合
- Integer2 が 0 である場合
- オーバフローした場合

通常は以下のマクロ表記を用いる.

```
Z := X mod Y  =>  modulo(X, Y, Z).
```

[6] 負の絶対値最大数を -1 で割った場合にオーバフローが起きる.

shift_left(Integer, ShiftWidth, ^NewInteger) :: GB

Integer を ShiftWidth ビットだけ左に論理シフトした結果を NewInteger とユニファイする

中断: Integer または ShiftWidth が未定義の場合

失敗 / 例外:

- Integer が整数でない場合
- $0 \leq$ Shiftwidth ≤ 31 の整数でない場合

通常は以下のマクロ表記を用いる.

```
Z := X << Y  =>  shift_left(X, Y, Z).
```

shift_right(Integer, ShiftWidth, ^NewInteger) :: GB

Integer を ShiftWidth ビットだけ右に論理シフトした結果を NewInteger とユニファイする

中断: Integer または Shiftwidth が未定義の場合

失敗 / 例外:

- Integer が 整数でない場合
- $0 \leq$ Shiftwidth ≤ 31 の整数でない場合

通常は以下のマクロ表記を用いる.

```
Z := X >> Y  =>  shift_right(X, Y, Z).
```

and(Integer1, Integer2, ^NewInteger) :: GB

Integer1 と Integer2 の各ビットの論理積をとり結果を NewInteger にユニファイする.

中断: Integer1 または Integer2 が未定義の場合

失敗 / 例外:

- Integer1 または Integer2 が 整数でない場合

通常は以下のマクロ表記を用いる.

```
Z := X /\ Y  =>  and(X, Y, Z).
```

or(Integer1, Integer2, ^NewInteger) :: GB

Integer1 と Integer2 の各ビットの論理和をとり結果を NewInteger にユニファイする.

中断: Integer1 または Integer2 が未定義の場合

失敗 / 例外:

- Integer1 または Integer2 が 整数でない場合

通常は以下のマクロ表記を用いる.

```
Z := X \/ Y  =>  or(X, Y, Z).
```

exclusive_or(Integer1, Integer2, ^NewInteger) :: GB

Integer1 と Integer2 の各ビットの排他的論理和をとり結果を NewInteger にユニファイする.

中断: Integer1 または Integer2 が未定義の場合

失敗 / 例外:

- Integer1 または Integer2 が 整数でない場合

通常は以下のマクロ表記を用いる.

```
Z := X xor Y  =>  exclusive_or(X, Y, Z).
```

complement(Integer, ^NewInteger) :: GB

Integer の各ビットの反転した値を NewInteger にユニファイする

中断: Integer が未定義の場合

失敗 / 例外:

- Integer が整数でない場合

通常は以下のマクロ表記を用いる.

```
Y := \(X)  =>  complement(X, Y).
```

【注意】この述語は現在ボディ部に使うと, -1 と exclusive_or をとるのと同じである.

7.4.5 浮動小数点数の比較

floating_point_less_than(Float1, Float2) :: G

Float1 $<$ Float2 であれば成功

中断: Float1 または Float2 が未定義の場合

失敗:

- Float1 または Float2 が浮動小数点数でない場合
- 上記成功条件を満たさない場合

通常は以下のマクロ表記を用いる.

```
X $< Y  =>  floating_point_less_than(X, Y).
X $> Y  =>  floating_point_less_than(Y, X).
```

floating_point_not_less_than(Float1, Float2) :: G

Float1 $\geq$ Float2 であれば成功

中断: Float1 または Float2 が未定義の場合

失敗:

- Float1 または Float2 が浮動小数点数でない場合
- 上記成功条件を満たさない場合

通常は以下のマクロ表記を用いる.

```
X $>= Y  =>  floating_point_not_less_than(X, Y).
X $=< Y  =>  floating_point_not_less_than(Y, X).
```

floating_point_equal(Float1, Float2) :: G

Float1 = Float2 であれば成功

中断: Float1 または Float2 が未定義の場合

失敗:

- Float1 または Float2 が浮動小数点数でない場合
- 上記成功条件を満たさない場合

通常は以下のマクロ表記を用いる.

```
X $=:= Y  =>  floating_point_equal(X, Y).
```

floating_point_not_equal(Float1, Float2) :: G

Float1 と Float2 が等しくない場合

中断: Float1 または Float2 が未定義の場合

失敗:

- Float1 または Float2 が浮動小数点数でない場合
- 上記成功条件を満たさない場合

通常は以下のマクロ表記を用いる.

```
X $=\= Y  =>  floating_point_not_equal(X, Y).
```

7.4.6 浮動小数点数の演算

floating_point_add(Float1, Float2, ^NewFloat) :: GB

Float1 に Float2 を加えた値を NewFloat にユニファイする. オーバフローした時は NewFloat に無限大をユニファイする.

中断: Float1 または Float2 が未定義の場合

失敗 / 例外:

- Float1 または Float2 が浮動小数点数でない場合
- $(+\infty)+(-\infty)$ または $(-\infty)+(+\infty)$ を実行した場合

通常は以下のマクロ表記を用いる.

```
Z $:= X + Y  =>  floating_point_add(X, Y, Z).
```

floating_point_subtract(Float1, Float2, ^NewFloat) :: GB

Float1 から Float2 を引いた値を NewFloat にユニファイする. オーバフローした時は NewFloat に無限大をユニファイする.

中断: Float1 または Float2 が未定義の場合

失敗 / 例外:

- Float1 または Float2 が浮動小数点数でない場合
- $(+\infty)-(+\infty)$ または $(-\infty)-(-\infty)$ を実行した場合

通常は以下のマクロ表記を用いる.

```
Z $:= X - Y  =>  floating_point_subtract(X, Y, Z).
```

floating_point_multiply(Float1, Float2, ^NewFloat) :: GB

Float1 に Float2 を掛けた値を NewFloat にユニファイする. オーバフローした時は NewFloat に無限大をユニファイする.

中断: Float1 または Float2 が未定義の場合

失敗 / 例外:

- Float1 または Float2 が浮動小数点数でない場合
- $(\pm 0.0) * (\pm\infty)$ または $(\pm\infty) * (\pm 0.0)$ を実行した場合

通常は以下のマクロ表記を用いる.

```
Z $:= X * Y  =>  floating_point_multiply(X, Y, Z).
```

floating_point_divide(Float1, Float2, ^NewFloat) :: GB

Float1 を Float2 で割った商を NewFloat にユニファイする. オーバフローした時は NewFloat に無限大をユニファイする. Float1 が $\pm$ 0.0 でなく Float2 が $\pm$ 0.0 ならば NewFloat に無限大をユニファイする.

中断: Float1 または Float2 が未定義

失敗 / 例外:

- Float1 または Float2 が浮動小数点数でない場合
- $(\pm 0.0)/(\pm 0.0)$ または $(\pm\infty)/(\pm\infty)$ を実行した場合

通常は以下のマクロ表記を用いる.

```
Z $:= X / Y  =>  floating_point_divide(X, Y, Z).
```

7.4.7 整数と浮動小数点数の変換

floating_point_to_integer(Float, ^Integer) :: GB

Float を整数に変換 (小数点以下切り捨て) して Integer にユニファイする

中断: Float が未定義の場合

失敗 / 例外:

- Float が浮動小数点数でない場合
- オーバフローの場合

通常は以下のマクロ表記を用いる.

```
Y := int(X)  =>  floating_point_to_integer(X, Y).
```

integer_to_floating_point(Integer, ^Float) :: GB

Integer を浮動小数点数に変換して Float にユニファイする.

中断: Integer が未定義の場合

失敗 / 例外: Integer が 整数でない場合

通常は以下のマクロ表記を用いる.

```
Y $:= float(X)  =>  integer_to_floating_point(X, Y).
```

7.4.8 整数の倍精度演算

add_extended(X, Y, ^Zh, ^Zl) :: B

X, Y を 32 ビット符号なし整数とみなして, X に Y を加えた値で下位 32 ビットを Zl に, 桁上がり分を Zh にユニファイする.

中断: X または Y が未定義の場合

例外: X または Y が整数でない場合

subtract_extended(X, Y, ^Borrow, ^Z) :: B

X, Y を 32 ビット符号なし整数とみなして, X から Y を引いた結果が非負 Z に絶対値, Borrow に 0 を, 負ならば Z には 2^{32} の補数をとったものを, Borrow には 1 をユニファイする.

中断: X または Y が未定義の場合

例外: X または Y が整数でない場合

multiply_extended(X, Y, ^Zh, ^Zl) :: B

X, Y を 32 ビット符号なし整数とみなして, X と Y を掛けた結果で下位 32 ビットを Zl に, 上位 32 ビットを Zh にユニファイする.

中断: X または Y が未定義の場合

例外: X または Y が整数でない場合

devide_extended(Xh, Xl, Y, ^Z, ^Mod) :: B

Xh, Xl, Y を 32 ビット符号なし整数とみなして, Xh を上位 32 ビット, Xl を下位 32 ビットとみなした 64 ビット符号なし整数を Y で割った商を Z に, 余りを Mod にユニファイする.

中断: Xh, Xl または Y が未定義の場合

例外:

- Xh, Xl または Y が整数でない場合
- 演算結果が 32 ビット整数に表せない場合
- Y が 0 の場合

7.4.9 多倍長整数の比較

以下の組込述語の引数としての多倍長整数の表現形式は 32 ビットストリングで, これを符号のない整数値を表すものとみなして扱う. MSB が第 0 要素で 0 サプレスである.

bignum_less_than(Bignum1, Bignum2) :: G

多倍長整数と Bignum1, Bignum2 について, Bignum1 < Bignum2 であれば成功.

中断: Bignum1 または Bignum2 が未定義の場合

失敗:

- Bignum1 または Bignum2 が 32 ビットのストリングの多倍長整数形式以外の場合
- 成功条件を満たさない場合

bignum_not_less_than(Bignum1, Bignum2) :: G

多倍長整数と Bignum1, Bignum2 について, Bignum1 ≥ Bignum2 であれば成功.

中断: Bignum1 または Bignum2 が未定義の場合

失敗:

- Bignum1 または Bignum2 が 32 ビットのストリングの多倍長整数形式以外の場合
- 成功条件を満たさない場合

7.4.10 多倍長整数演算

以下の組込述語の引数としての多倍長整数の表現形式は 32 ビットストリングで, これを符号のない整数値を表すものとみなして扱う. MSB が第 0 要素で 0 サプレスである.

add_bignum(BigX, BigY, ^BigZ) :: B

BigX, BigY の加算を行い, その結果を BigZ にユニファイする.

中断: BigX または BigY が 未定義の場合

例外: BigX または BigY が 32 ビットストリングの多倍長整数形式以外の場合

subtract_bignum(BigX, BigY, ^Borrow ^BigZ) :: B

BigX から BigY の減算を行い, その結果の絶対値を BigZ にユニファイする. 結果が非負の場合は, Borrow に 0 を, 負の場合は 1 をユニファイする.

中断: BigX または BigY が 未定義の場合

例外: BigX または BigY が 32 ビットストリングの多倍長整数形式以外の場合

multiply_bignum_by_word(BigX, Y, ^BigZ) :: B

BigX と整数 Y を 32 ビット符号なし整数とみなして, 掛けた結果を BigZ にユニファイする.

中断: BigX または Y が未定義の場合

例外:

- BigX が 32 ビットストリングの多倍長整数形式以外の場合
- Y が整数でない場合

【参考】ここに述べた多倍長整数の比較・演算のための基本的な組込述語を用いて, 多倍長整数に対する種々の演算を実現するモジュールの提供を予定しており, 現在準備中である. モジュール名は bignum となる. また, 多倍長整数に対する算術式のためのマクロも提供予定で, これらのマクロには “^:=”, “^<” などのオペレータを定義して用いる予定である.

7.4.11 ベクタ関係

vector(X, ^Size, ^NewVector) :: B

ベクタ X のサイズを Size とユニファイし X を NewVector にユニファイする.

中断: X が未定義の場合

例外: X がベクタ以外の場合

new_vector(^Vector, Size) :: B

要素数 が Size であるようなベクタを新たに生成し Vector とユニファイする. 生成したベクタの要素はすべて整数 0 で初期化される.

中断: Size が未定義の場合

例外: Size が非負整数でない場合

vector_element(Vector, Position, ^Element) :: G

ベクタ Vector の Position 番目の要素を Element とユニファイする.

中断: Vector または Position が未定義の場合

失敗:

- Vector が ベクタ以外の場合
- Position が非負整数でないか, 要素数以上である場合

vector_element(Vector, Position, ^Element, ^NewVector) :: B

ベクタ Vector の Position 番目の要素を Element とユニファイするとともに Vector を NewVector にユニファイする.

中断: Vector または Position が未定義の場合

例外:

- Vector が ベクタ以外の場合
- Position が非負整数でないか, 要素数以上である場合

set_vector_element(Vector, Position, ^OldElem, NewElem, ^NewVect) :: B

ベクタ Vector の Position 番目の要素を OldElem とユニファイし, Vector の Position 番目を NewElem と置き換えた新しいベクタを生成し NewVect とユニファイする.

中断: Vector または Position が未定義の場合

例外:

- Vector が ベクタ以外の場合
- Position が非負整数でないか, 要素数以上である場合

【参考】 NewElem は定義済みである必要はなく, この組込述語の実行時点では未定義変数であっても良い.

7.4.12 ストリングの比較

ストリングの比較の方法は, 要素のビット幅が異なればビット幅の大きい方が大. 要素のビット幅が同じなら, 先頭要素から順に, 正整数扱いで大小比較していく. 比較の途中でいずれかのストリングの末尾に達した場合, 要素数の多いものが大となる.

string_less_than(String1, String2) :: G

String1 < String2 であれば成功

中断: String1 または String2 が未定義の場合

失敗:

- String1 または String2 がストリングでない場合
- 成功条件を満たさない場合

string_not_less_than(String1, String2) :: G

String1 ≥ String2 であれば成功

中断: String1 または String2 が未定義の場合

失敗:

- String1 または String2 がストリングでない場合
- 成功条件を満たさない場合

7.4.13 ストリング操作

string(X, ^Size, ^ElementSize, ^NewString) :: B

ストリング X のサイズを Size とユニファイし要素のビット幅を ElementSize とユニファイするとともに X を NewString にユニファイする.

中断: X が未定義の場合

例外: X がストリング以外の場合

new_string(^String, Size, ElementSize) :: B

要素数 Size, 要素のビット幅が ElementSize のストリングを新たに生成し String とユニファイする. 生成したストリングの要素は全て整数 0 で初期化される.

中断: Size または, ElementSize が未定義の場合

例外:

- Size が非負整数の場合
- ElementSize が 1, 8, 16, 32 以外の整数の場合

string_element(String, Position, ^Element) :: G

String の Position 番目の要素を整数として Element とユニファイする.

中断: String または Position が未定義の場合

失敗:

- String がストリング以外の場合
- Position が非負整数以外または String の要素数以上の場合

string_element(String, Position, ^Element, ^NewString) :: B

String の Position 番目の要素を Element とユニファイするとともに String を NewString とユニファイする.

中断: String または Position が未定義の場合

例外:

- String がストリング以外の場合
- Position が非負整数以外または String の要素数以上の場合

set_string_element(String, Position, NewElement, ^NewString) :: B

String の Position 番目の要素を NewElement 置き換えた新たなストリングを生成し NewString とユニファイする.

中断: String, Position, または NewElement が未定義の場合

例外:

- String がストリング以外の場合
- Position が非負整数以外または String の要素数以上の場合
- NewElement が String の要素に収まる範囲の正の整数以外の場合 (ただし, 32 ビットストリングでは整数以外の場合)

change_element_size(String, NewElemSize, ^NewString) :: B

String の要素のビット幅を NewElemSize に変換したものを ^NewString にユニファイする. NewElemSize は 1, 8, 16, 32 のいずれかである.

ビット幅を大きくする場合は String の複数の要素が NewString の 1 要素に, 小さくする場合は String の 1 要素が NewString の複数の要素に対応づけられる. 複数要素と 1 要素の対応では, 要素番号の小さいものが上位ビットを占めるようになる. たとえば, String が 8 ビットストリングで NewElemSize が 16 なら, String の要素 0 が NewString の要素 0 の上位 8 ビットに, String の要素 1 が NewString の要素 1 の下位 8 ビットになる.

要素のビット幅の小さいものから大きいものに変換する時に, String の要素数が NewElemSize の要素の境界に一致しない場合は, 要素数を NewElemSize の境界に一致するように切り上げ, 末尾に追加した要素は 0 詰めする.

中断: String または NewElemSize が未定義の場合

例外:

- String がストリングでない場合
- NewElemSize が 1, 8, 16, 32 以外の場合

7.4.14 ストリングの要素の検索

search_character(String, Start, End, CharacterCode, ^Position, ^NewString) :: B

String の Start から End まで (要素 End を含む) の範囲で CharacterCode と同一の文字を探し, 最初に見つかった要素位置を Position にユニファイし, NewString に String をユニファイする. End < Start の場合は, 後ろから探す. Start から End の間に CharacterCode と同一の文字がなかった場合, Position には -1 をユニファイする.

中断: String, Start, End または CharacterCode が未定義の場合

例外:

- String がストリング以外の場合
- Start が 0 ≤ Start < ストリングの要素数 の整数でない場合
- End が 0 ≤ End < ストリングの要素数 の整数でない場合
- CharacterCode が String の要素に収まる範囲の正の整数以外の場合 (ただし, 32 ビットストリングでは整数以外の場合)

search_characters
(String, Start, End, Mask, Table, ^Position, ^NewString, ^NewTable) :: B

String 中の Start から End まで (要素 End を含む) の範囲の各要素で Table を引いた値と Mask (整数) との AND をとった結果が non-zero のものを探し, その要素位置を Position に ユニファイし, NewString は String と同じものを, NewTable は Table と同じものをユニファイする.

Table は 次のいずれかである.

- String が 8 ビットの時は長さが 256, 要素のビット幅が 8 ビットのストリング.
- String が 16 ビットの時は長さが 65536, 要素のビット幅が 8 ビットのストリング.

【注意】String は 8 ビットまたは 16 ビットのみである.

中断: String, Start, End, Mask, または Table が未定義の場合

例外:

- String が 8 ビットまたは 16 ビットのストリング以外の場合
- Start が 0 ≤ Start < ストリングの要素数 の整数でない場合
- End が 0 ≤ End < ストリングの要素数 の整数でない場合
- Mask が整数でない場合
- Table が 8 ビットストリングでない場合

7.4.15 汎用ストリング演算

binary_string_operation
(Op, String1, Position1, Length, Position2, String2, ^NewString1, ^NewString2) :: B

String1 の Position1 番目から Length 個の要素と, String2 の Position2 番目から Length 個の要素との間で, Op で指定された演算 (表 7.4) を行い, その結果で String1 の Postion1 番目から Length 個の要素を置き換えてたものを NewString1 にユニファイする. NewString2 には String2 をそのままユニファイする.

中断: Op, String1, Position1, Length, Position2 または String2 が未定義の場合

例外:

- Op が 0 ≤ Op ≤ 15 の整数でない場合
- String1, String2 がストリングでない場合
- String1, String2 が同じビット幅のストリングでない場合
- Postion1, Position2, Length が非負整数でない場合
- Position1 + Length > (String1 の要素数) の場合
- Position2 + Length > (String2 の要素数) の場合

【注意】Op の値が String2, Position2 を必要としない場合でも String2, Position2 のチェックを行う

shifting_string_operation
(Op, String1, Position1, Length, Position2, ^NewString1) :: B

String1 の Position1 番目から Length 個の要素と, String1 の Position2 番目から Length 個の要素との間で, Op で指定された演算 (表 7.4) を行い, その結果で String1 の Postion1 番目から Length 個の要素を置き換えてたものを NewString1 にユニファイする.

中断: Op, String1, Position1, Length または Position2 が未定義の場合

例外:

- Op が $0 \leq \mathrm{Op} \leq 15$ の整数でない場合
- String1 がストリングでない場合
- Postion1, Position2, Length が非負整数でない場合
- Position1 + Length > (String1 の要素数) の場合
- Position2 + Length > (String2 の要素数) の場合

【注意】Op の値が Position2 を必要としない場合でも Position2 のチェックを行う

unary_string_operation(Op, String1, Position1, Length, ^NewString1) :: B

String1 の Position1 番目から Length 個の要素と同士で Op で指定された演算 (表 7.4) を行い, その結果で String1 の Postion1 番目から Length 個の要素を置き換えてたものを NewString1 にユニファイする.

中断: Op, String1, Position1 または Length が未定義の場合

例外:

- Op が $0 \leq \mathrm{Op} \leq 15$ の整数でない場合
- String1 がストリングでない場合
- Postion1, Length が非負整数でない場合
- Position1 + Length > (String1 の要素数) の場合

【参考 1】引数 Op の指定には, 表 7.4 に掲げたマクロを用いるのが便利である. これらのマクロはマクロモジュール pimos に定義されている.

【参考 2】次節に述べるストリングマクロユーティリティは, 本節に述べた組込述語を用いるようになっている.

7.4.16 ストリングマクロユーティリティ

以下の述語は, マクロで記述することにより, PIMOS が提供するストリング関係のユーティリティを呼び出せる. 以下のマクロはマクロモジュール pimos に定義されている.

builtin#substring(String, Position, Length, ^SubString, ^NewString) :: B

String の Position 番目から長さ Length 分をコピーし新たなストリングを生成し SubString とユニファイする. また, String と NewString をユニファイする.

中断: String, Position または Length が未定義の場合

例外:

- String がストリング以外の場合
- Position が $0 \leq \mathrm{Position} \leq$ (String の要素数) の整数でない場合
- Length が $0 \leq \mathrm{Length} \leq$ (String の要素数) − Position の整数でない場合

表 7.4: 汎用ストリング演算表

番号	マクロ	入力 (S1,S2) に対する出力
0	string_operation#clear	0
1	string_operatin#and	S1 AND S2
2	string_operation#s2_and_not_s1	(not S1) AND S2
3	string_operation#s2	S2
4	string_operation#s1_and_not_s2	S1 and (not S2)
5	string_operation#s1	S1
6	string_operation#xor	S1 XOR S2
7	string_operation#or	S1 OR S2
8	string_oprtation#nor	S1 NOR S2
9	string_operation#not_xor	S1 not XOR S2
10	string_operation#not_s1	not S1
11	string_operation#s2_or_not_s1	(not S1)O R S2
12	string_operation#not_s2	not S2
13	string_operation#s1_or_not_s2	S1 OR (not S2)
14	string_operation#nand	S1 NAND S2
15	string_operation#set	1

builtin#set_substring(String, Position, SubString, ^NewString) :: B

String の Position 番目からを SubString で示されたストリングで置き換えたストリングを生成し NewString とユニファイする.

中断: String, Position または SubString が未定義の場合

例外:

- String, SubString がストリング以外の場合
- String, SubString が同じビット幅でない場合
- Position が 0 ≤ Position ≤ (String の要素数) – (SubString の要素数) の整数でない場合

builtin#set_substring(String, Position, Length, SubString, ^NewString) :: B

String の Position 番目からを Length 個の要素を SubString で示されたストリングの 0 番目から Length 個の要素で置き換えたストリングを生成し NewString とユニファイする.

中断: String, Position, Length または SubString が未定義の場合

例外:

- String, SubString がストリング以外の場合
- String, SubString が同じビット幅でない場合
- Position が 0 ≤ Position ≤ (String の要素数) の整数でない場合
- Length が 0 ≤ Length ≤ (SubString の要素数) の整数でない場合
- Length が Length ≤ String の要素数 – Position の整数でない場合

builtin#set_substring
(String, Position, Length, SubString, ^NewString1, ^NewString2) :: B

String の Position 番目からを Length 個の要素を SubString の で示されたストリングの 0 番目から Length 個の要素で置き換えたストリングを生成し NewString1 とユニファイする. NewString2 はそのまま String をユニファイする.

中断: String, Position, Length または SubString が未定義の場合

例外:

- String, SubString がストリング以外の場合
- String, SubString が同じビット幅でない場合
- Position が $0 \leq$ Position $\leq$ (String の要素数) の整数でない場合
- Length が $0 \leq Length \leq$ (SubString の要素数) の整数でない場合
- Length が Length $\leq$ (String の要素数) − Position の整数でない場合

builtin#copy_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

String1 の Position 番目からを Length 個の要素を String2 の Position2 番目から Length 個の要素で置き換えたストリングを生成し NewString1 とユニファイする. NewString2 はそのまま String2 をユニファイする.

中断: String1, String2, Position1, Position2 または Length が未定義の場合

例外:

- String1, String2 がストリング以外の場合
- String1, String2 が同じビット幅でない場合
- Position1 が $0 \leq$ Position1 $\leq$ (String1 の要素数) の整数でない場合
- Position2 が $0 \leq$ Position2 $\leq$ (String2 の要素数) の整数でない場合
- Length が $0 \leq$ Length $\leq$ (String1 の要素数) − Position1 の整数でない場合
- Length が $0 \leq$ Length $\leq$ (String2 の要素数) − Position2 の整数でない場合

builtin#or_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#xor_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#and_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#nand_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#nor_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

builtin#not_xor_string_elements
(String1, Position1, Length, String2, Position2, ^NewString1, ^NewString2) :: B

String1 の Position 番目からを Length 個の各要素を String2 の Position2 番目から Length 個の各要素と, それぞれ or, xor, and, nand, nor, xor の否定の演算を施した結果で置き換えたストリングを生成し, NewString1 とユニファイする. NewString2 はそのまま String2 をユニファイする.

中断: String1, String2, Position1, Position2 または Length が未定義の場合

例外:

- String1, String2 がストリング以外の場合
- String1, String2 が同じビット幅でない場合
- Position1 が $0 \leq$ Position1 $\leq$ (String1 の要素数) の整数でない場合
- Position2 が $0 \leq$ Position2 $\leq$ (String2 の要素数) の整数でない場合
- Length が $0 \leq$ Length $\leq$ (String1 の要素数) − Position1 の整数でない場合
- Length が $0 \leq$ Length $\leq$ (String2 の要素数) − Position2 の整数でない場合

builtin#complement_string_elements(String, Position, Length, ^NewString) :: B

String の Position 番目からを Length 個の各要素の complement をした結果で置き換えたストリングを生成し NewString とユニファイする.

中断: String, Position または Length が未定義の場合

例外:

- String がストリング以外の場合
- Position が $0 \leq$ Position $\leq$ (String の要素数) の整数でない場合
- Length が $0 \leq$ Length $\leq$ (String の要素数) − Position1 の整数でない場合

builtin#move_string_elements
(String, Position, Length, ShiftCount, ^NewString) :: B

String の Position 番目からを Length 個の各要素を ShiftCount だけシフトしたストリングを生成し NewString とユニファイする. $0 <$ ShiftCount では右シフト. $0 >$ ShiftCount では左シフト.

中断: String, Position, Length, または ShiftCount が未定義の場合

例外:

- String がストリング以外の場合
- Position が $0 \leq$ Position $\leq$ (String の要素数) − ShiftCount の整数でない場合
- Length が $0 \leq$ Length $\leq$ (String の要素数) − Position1 − ShiftCount の整数でない場合
- Shiftcount が非負整数でない場合

builtin#string_tail(String, Position, ^NewString) :: B

String の Position 番目から後ろの ストリングを NewString とユニファイする.

中断: String, または Position が未定義の場合

例外:

- String がストリング以外の場合
- Position が $0 \leq$ Position $\leq$ (String の要素数) の整数でない場合

builtin#append_string(String1, String2, ^NewString) :: B

String1 の内容の後に String2 の内容を繋げた新たなストリングを生成し NewString とユニファイする.

中断: String1, String2 が未定義の場合

例外:

- String1, String2 がストリング以外の場合
- String1, String2 が同じビット幅でない場合

builtin#append_string([String1, String2, ..., StringN], ^NewString) :: B

String1 から StringN の内容を繋げた新たなストリングを生成し NewString とユニファイする.

中断: String1 から StringN のうちひとつでも未定義の場合

例外:

- String1 から StringN のうちひとつでもストリング以外の場合
- String1 から StringN の全部が同じビット幅でない場合
- 第一引数がリスト以外, または空リストの場合

【参考】2.5 版では, 第一引数が空リストの場合に, 長さ 0 の 16 ビットストリングが NewString とユニファイされていた.

7.4.17 アトム関係

new_atom(^Atom) :: B

新たにアトムを生成し, Atom とユニファイする.

中断, 例外: なし

【注意】PIMOS/KL1 ではアトムと名前文字列の対応づけを行うのは PIMOS の仕事であり, この組込述語で生成されるアトムは, 名前の文字列との対応を持たない. 名前を持つアトムを作りたい場合には後述のアトム名データベース (10.1 アトムの管理, 385 頁) を用いること.

7.4.18 コード / モジュール関係

コード及び, モジュールに関しては 各 PIM およびマルチ PSI において機種依存の部分が多いので以下の組込述語の使用には注意すること.

module_element(Module, Position, ^Element) :: B

Module の Position 番目の要素を Element とユニファイする.

中断: Module, または Position が未定義の場合

例外:

- Module がモジュールでない場合
- Position が $0 \leq$ Position $<$ (Module の要素数) の整数でない場合

predicate_to_code(Module, Name, Arity, ^Code) :: B

モジュール Module, 述語名 Name, 引数個数 Arity であるコードを Code とユニファイする. モジュール内に対応する述語が存在しない (定義されていないかパブリック宣言されていない) 場合には, {`Module, Name, Arity`} という 3 要素ベクタを Code とユニファイする.

中断: Module, Name または Arity が未定義の場合

例外:

- Module がモジュール以外の場合
- Name がアトム以外の場合
- Arity が非負整数以外の場合

code_to_predicate(Code, ^Module, ^Name, ^Arity, ^Info) :: B

Code の属するモジュールを Module に, 述語名アトムを Name に, 引数個数を Arity に, ソフト用の 1 ワードの情報 (<Code のアドレス −3> 番地の内容) を Info にユニファイする.

中断: Code が未定義の場合

例外: Code がコード以外の場合

【注意】本述語の仕様は機種依存で, この仕様は マルチ PSI, PIM/m のものである.

apply(Code, Args) :: B

Code に対応する述語を引数環境 Args で呼び出す.

中断: Code または Args が未定義の場合

例外:

- Code がコード以外の場合
- Code がスパイ対象述語の呼びだし先コードの場合
- Args がベクタ以外の場合
- Code の引数個数と Args の要素数が不一致の場合

apply_tracing(Code, Args, ID) :: B

Code に対応する述語を引数環境 Args でトレース対象ゴールとして呼び出す. ID はこのゴールを識別するためのもの (任意のデータ型が許される) で, トレース例外発生時に返される.

中断: Code または Args が未定義の場合

例外:

- Code がコード以外の場合
- Args がベクタ以外の場合
- Code の引数個数と Args の要素数が不一致の場合
- 組込述語 apply_spying による実行の中で呼ばれた場合

apply_spying(Code, Args, Spied_code_vector, ID) :: B

Code に対応する述語を引数環境 Args でスパイ対象ゴールとして呼び出す. ID はこのゴールを識別するためのもの (任意のデータ型が許される) で, スパイ例外発生時に返される. Spied_code_vector はスパイ対象述語を指定するためのベクタで, モジュール名アトム, 述語名アトム, 引数個数の繰り返し列を要素として持つ. モジュール名アトム, 述語名アトムのワイルドカードは整数 0, 引数個数のワイルドカードは [] (NIL) である. Spied_code_vector は以下のようなフォーマットを持つ.

```
{ Mod1, Pred1, Arity1, Mod2, Pred2, Arity2, ... , ModN, PredN, ArityN }
```

中断: Code, Args, Spied_code_vector または ID が未定義の場合

例外:

- Code がコード以外の場合
- Code がスパイ対象述語の呼びだし先コードの場合
- Args がベクタ以外の場合
- Code の引数個数と Args の要素数が不一致の場合
- Spied_code_vector の要素が未定義である場合
- Spied_code_vector の形式が正しくない場合
- 組込述語 apply_spying による実行の中で呼ばれた場合

7.4.19 ストリームサポート

merge(In, ^Out) :: B

マージャを生成し, そのマージャに対する入力ストリームを In に, 出力ストリームを Out にユニファイする. マージャはベクタを受け取るとベクタ内の全ての要素を新たに入力ストリームとして加える.

中断: In が未定義の場合

例外: In がベクタまたはリスト以外の場合

マージャの論理的な定義は以下に示すような 2 から無限の引数まで持つような述語として定義できる.

```
merge([], O) :- true | O=[].
merge([A|I], O) :- true | O=[A|NO], merge(I, NO).
merge({}, O) :- true | O=[].
merge({I}, O) :- true | merge(I, O).
merge({I1, I2}, O) :- true | merge(I1, I2, O).
merge({I1, I2, I3}, O) :- true | merge(I1, I2, I3, O).
                  ・・・
merge([], I2, O) :- true | merge(I2, O)
merge(I1, [], O) :- true | merge(I1, O).
merge([A|I1], I2, O) :- true | O=[A|NO], merge(I1, I2, NO).
merge(I1, [A|I2], O) :- true | O=[A|NO], merge(I1, I2, NO).
merge({}, I2, O) :- true | merge(I2, O).
merge(I1, {}, O) :- true | merge(I1, O).
merge({I3, I4}, I2, O) :- true | merge(I3, I4, I2, O).
merge({I3, I4, I5}, I2, O) :- true | merge(I3, I4, I5, I2, O).
                  ・・・
                  ・・・
```

7.4.20 特殊入出力

【注意】ここに述べる入出力用の組込述語は, デバッグを容易にするために低レベルで用意するものであり, 機能 / 性能の両面で本格的な入出力としての利用には適さないものである. また, 機能は一部機種依存である.

read_console(^X) :: G

CSP ウインドウから入力したデータを読み込み, X とユニファイする. 入力の方法は, タグ部 [空白] 値部で, 各々 16 進数で入力する. タグ部を省略した場合は, 整数とみなされる.

中断, 失敗: なし

【注意】入力待ちの時はすべてのノードが停止する. PIM/m, マルチ PSI 以外では仕様が異なる可能性がある.

display_console(X) :: G

CSP ウインドウに X の現在の値を書き出し, 改行する. X が 8 ビットストリングの場合にはそれを ASCII ストリングと見なし, 16 ビットストリングの場合にはそれを JIS ストリングと見なして CSP

ウインドウにそのストリングの内容を書き出す. X がベクタの場合には各要素の内容を CSP ウインドウに書き出す. それ以外のデータ型の場合はタグ部と値部を表示する.

中断: X が未定義の場合

失敗: なし

【注意】出力中はすべてのノードが停止する. PIM/m, マルチ PSI 以外では仕様が異なる可能性がある.

get_console(^X) :: G

CSP ウインドウから入力した文字を一文字を読み込みその ASCII コードを X とユニファイする.

【注意】入力待ちの時はすべてのノードが停止する.

put_console(X) :: G

X が整数の場合にはそれを ASCII コードと見なして CSP ウインドウに 1 文字書き出す. 改行はしない. X が 8 ビットストリングの場合にはそれを ASCII ストリングと見なして CSP ウインドウにそのストリングの内容を書き出す. 改行はしない. X がそれ以外の型の場合は動作を保証しない.

中断: X が未定義の場合

失敗: なし

【注意】出力中はすべてのノードが停止する. PIM/m, マルチ PSI 以外では仕様が異なる可能性がある.

7.4.21 その他

current_priority(^Priority, ^Min, ^Max) :: B

本述語の論理優先度を Priority とユニファイし, 本述語が属する荘園で許されている最低優先度を Min と, 最高優先度を Max とユニファイする.

中断, 例外: なし

calculate_priority_from_current(X, ^Y) :: B

自己相対的優先度 X を指定すると, X に対応する論理的優先度を本述語の論理優先度から計算して Y とユニファイする.

中断: X が未定義の場合

例外: X が $-4096 \leq X \leq 4096$ の整数でない場合

calculate_priority_from_minimum(X, ^Y) :: B

所属荘園内相対的優先度 X を指定すると, X に対応する論理的優先度を本述語の属する荘園の論理優先度範囲から計算して Y とユニファイする.

中断: X が未定義の場合

例外: X が $0 \leq X \leq 4096$ の整数でない場合

current_node(^NodeNumber, ^TotalNode) :: B

本述語が属するノード番号を NodeNumber とユニファイし, 合計のノード数を TotalNode とユニファイする. ノード番号は 0 から TotalNode − 1 までふられている.

中断, 例外: なし

get_heap_size(^Size, ^Used) :: B

本述語が属するノードのメモリ利用状況を知る. Size は利用可能なメモリのサイズ, Used は現在の使用量が整数値としてユニファイされる. 単位はシステム依存.

中断, 例外: なし

【参考】 Multi プサイ, PIM/m では単位はワード (ポインタや整数値がひとつ入るメモリ量) である.

hash(X, ^Value, ^NewX) :: B

X をもとに整数 (ハッシュ値) を生成し Value とユニファイし, X と NewX をユニファイする. ここで X は任意のタイプで良い.

X のデータ型により, 以下のように整数を生成する.

整数: 値部 (そのまま)

アトム: 値部 (アトム番号)

その他のアトミックデータ: 値部

リスト: −1

ベクタ: ベクタの長さ

ストリング: ストリングの長さと先頭要素最終要素から計算

その他: 0

中断: X が未定義の場合

例外: なし

【注意】 ハッシュ関数の返す値は将来仕様が変更される可能性がある. 基本原則として, 一回の実行中に同じデータについて (呼び出し時に未定義部分を含むような構造体であっても) 何回ハッシュ関数を計算しても (計算が終了する限りは) 同じハッシュ値を返す, ということは保証する (変数の具体化によって終了しなかったハッシュが終了するようになることはあっても, ハッシュ値が変わることはない).

【参考】 PIMOS の標準のハッシュ関数は, この組込述語の直接の結果ではなく, ユーティリティ hasher で提供するハッシュ関数である (8.2 「ハッシング」 305頁参照).

system_timer(^X1, ^X2) :: G

本述語の実行された プロセッサ が立ち上がった時間からの経過時間を 1/16 msec 単位で 上位 16 ビットを X1 に, 下位 32 ビットを X2 にユニファイする.

中断, 失敗: なし

【注意】本述語はハードウェアに依存した仕様であり, PIM/m, マルチ PSI 以外では, 単位が機種依存となる. また, タイマを持たないモデルもある. このため, 通常は 9.2.3 節 (350 頁) に述べるユーティリティを通して用いることが望ましい.

7.4.22 参照数管理について

現在の KL1 の言語処理系では MRB と呼ばれる方式でデータの参照数を管理して, インクリメンタルなガーベジコレクションなどに役立てている. この管理が行なわれているため, 組込述語の表層上の仕様を見る限り新たなデータ領域の割り付けが必要になりそうに見える場合でも, 実際には割り付けを必要としない場合が少なくない.

たとえば組込述語は set_vector_element/5 は, 言語の表層の仕様では元のベクタと要素がひとつだけ異なるような新たなベクタを生成する必要があるように見える. しかし, 参照数管理によって元のベクタが他には使われないことがわかる場合には, 元のベクタを書き換えてしまうので, メモリを割り付けずに済む. これは, メモリの消費量を削減するだけではなく, 要素のコピーのための手間がなくなるので, 計算の手間のオーダまで変える効果がある. このことは set_string_element, binary_string_operation, unary_string_operation, shifting_string_operation など, 構造体の一部の要素を書き換えたような別の構造体を作る組込述語, およびこれらに展開されるマクロについても同様である.

MRB は余分なメモリアクセスを伴わないため, 並列処理時に効率的な参照管理方式だが, いったんデータへの参照パスが複数になってしまうと, そのうちひとつを除いて全部が捨てられてもそれを検出することができない. このため, 本質的にひとつの参照パスで済むようなデータについては, 参照パスを単一に保つようなコーディングスタイルをとることが効率上望ましい.

以下の例では変数 Table への参照パスが, いったん vector_element/4 に渡されるものと, set_vector_element/5 に渡されるもののふたつになっている (vector_element の方は要素参照後にすぐ捨てているが). このため, set_vector_element による更新のコストはベクタの大きさに比例する程度になってしまう.

```
increment_element(Table, K, NewTable) :- true |
    vector_element(Table, K, Elem, _),
    NewElem := Elem + 1,
    set_vector_element(Table, K, _, NewElem, NewTable).
```

これを以下のように記述すれば参照パスが単一に保たれるので, ベクタの大きさによらず一定の手間で実行できる.

```
increment_element(Table, K, NewTable) :- true |
    set_vector_element(Table, K, Elem, NewElem, NewTable),
    NewElem := Elem + 1.
```

このプログラムでは set_vector_element の実行時には NewElem の値はまだ決まっていないことになるが, 足し算の結果が入るべき変数 NewElem が NewTable の要素に入るので, 問題なく動作する.

直接組込述語を呼んでいるのではなくても, 同様である. たとえば compute という述語はテーブルを用いてなんらかの計算を行ない, update という述語はその結果をテーブルに格納したような新しいテーブルを作るものとする. これを:

```
do(Table, NewTable) :- true |
    compute(Table, Values),
    update(Table, Values, NewTable).
```

のように記述すると, 変数 Table への参照が複数になり, 効率上の問題が出てくる. これを:

```
do(Table, NewTable) :- true |
    compute(Table, Values, Table1),
    update(Table1, Values, NewTable).
```

のように, compute が読み終ったテーブルを返し, それを update に渡すように記述すれば, テーブルへの参照パスはひとつだけに保てる.

この参照数をひとつに保つようなコーディングは, 頻繁に更新する構造体については特に推奨される. しかし, 常に墨守しなければならないわけではない. 参照数をひとつに保つことは並列性の制限につながる (ひとつの述語が読み終って返すまで, 次の述語は読むことができない). 更新の頻度や構造体の大きさと, 並列性を制限する程度とを勘案して, 複数参照にしても並列度を上げるべきか, 単一参照を守るべきかを決めることが望ましい.

7.5 荘園機能

PIMOS で用意するモジュール shoen は, KL1 の提供する荘園機能を使いやすい形でユーザに提供する. このモジュールは, KL1 の提供する機能の内, ユーザがどのように利用しても問題を生じないような機能だけを見せる, という役割も負う.

【参考】PIMOS では荘園機能に入出力機器などの管理機能を付加したタスクの機能を提供している. タスクについては 第 11 章 資源管理・実行管理 (p.398) を参照のこと.

7.5.1 概要

荘園は KL1 プログラムの実行のある部分 (荘園トップレベルのゴールと, そのゴールから派生した子孫ゴール群) をまとめたようなものである. 荘園の中にまた荘園を作ることもできるので, 一般に荘園は木構造をなす. 荘園は以下のような機能を持つ.

実行の制御: 荘園内のゴールの実行 (以下, 単に「荘園の実行」という) の開始 / 中断 / 再開 / 放棄を制御する.

【注意】並列処理を効率良く行うため, これらの操作は必ずしも即座に行われるとは限らない, という仕様になっている.

資源管理: 荘園の実行で消費する計算時間やメモリ領域などの計算機資源 (以下, この節では単に「資源」という) を管理する. 具体的には, 資源消費量が設定した上限に近づくと, それを報告する. 実際に上限を越えると実行を中断する. 資源消費の上限値は後から増やすこともでき (いったん与えたものを減らすことはできない), 資源消費が上限を越えたことによって中断している時に上限値を増やせば, 荘園の実行は再開する.

PIMOS の管理する資源と対比して, KL1 言語の荘園機能によって管理される資源を特に「言語定義資源」と呼ぶこともある.

【注意】現在管理されている資源消費量はリダクション数に対応するもの (ゴールを作ることに対して 1, そのリダクションを終了することに対して 1, 計 1 リダクションについて 2) だけである. 将来, 計算時間, メモリ消費量, ノード間の通信量なども管理するように改良する可能性がある.

例外処理: 荘園の実行中に生じた例外事項の処理を行う. 実行の失敗 (ゴールに対応する候補節がない場合や, ユニフィケーションの失敗) も例外として扱われる.

各荘園はどのような例外を処理するかを決めるマスクパターン (整数値) を持つ. 各例外事象も種類に応じて整数値のタグを持ち, 例外タグとマスクパターンの論理積が 0 でないような最も内側の親荘園に例外が報告される.

例外時にはどのような例外が生じたかを示す情報を報告すると共に, 例外を生じたゴールの代わりに実行するゴールのコード (述語に対応) と引数ベクタを指定するための変数も, 例外処理者に渡す. これらを具体化することによって, 例外を生じたゴールはコードと引数ベクタで指定されるゴールに置き換えられる. 荘園内にはこれらの変数の具体化を待っているゴールが存在するので, 例外処理を行わない内に荘園の実行が正常終了することはない.

7.5.2 荘園の生成

shoen:execute(Code, ArgV, MinPrio, MaxPrio, ExcepMask, Control, ^Report)

荘園を生成する.

生成する荘園のトップレベルのゴールは Code(トップレベルの述語に対応するコード型データ) と ArgV(トップレベルゴールの引数ベクタ) で指定する.

生成した荘園の実行制御はストリーム Control から行い, 実行状態の報告はストリーム Report からなされる. 制御ストリームと報告ストリームについてはそれぞれ 第 7.5.3 節 (p.291), 第 7.5.4節 (p.292) を参照のこと.

荘園内のゴールの優先度範囲は MinPrio と MaxPrio で指定する. これらは 0 から 4096 までの範囲の整数値で, 0 が外側の荘園内での最小優先度, 4096 がこの生成用の述語を呼び出したゴールの優先度に対応する.

ExcepMask は処理する例外事象を決めるマスクパターンである. 例外タグについては 第 7.5.6 節 (p.301) を参照のこと.

【注意】生成した荘園の中で PIMOS が管理する資源 (OS 定義資源) を利用する場合, それら資源の後始末を正しく行わなければならない (後始末を正しく行うとは, 使用している資源へのストリームを閉じて正常に終了させることである. 従って, abort により荘園内の実行を放棄したような場合は正しく後始末を行ったとは言わない). 資源の後始末が正しく行われなかった場合, 以降 PIMOS の正常な動作は保証されない. ユーザが生成した荘園の中で入出力機器などの OS 定義資源を利用する場合は, タスク機能を利用しなければならない. このため, 将来, モジュール shoen の使用は禁止するかもしれない.

shoen:profile(Code, ArgV, MinPrio, MaxPrio, ExcepMask, Control, ^Report)

プロファイル機能付きの荘園を生成する.

各引数は `shoen:execute/7` に与えるものと同じである. プロファイル機能とは, 生成した荘園の中で実行されるゴールが, 各述語について何回実行したかを計測するものである.

生成した荘園の実行制御はストリーム Control から行い, 実行状態の報告はストリーム Report からなされる. 荘園の実行途中で計測情報を収集することも可能である. プロファイルに関するメッセージについては 第 7.5.3 節 (p.291), 第 7.5.4 節 (p.292) を参照のこと.

【注意 1】プロファイル機能付きの荘園を生成した後で, 計測を止めることはできない. また, プロファイル機能付きの荘園の元で (子) 荘園を生成した場合, 子荘園は自動的にプロファイル機能付きの荘園となる. 但し, 計測された情報の報告は親の荘園にのみ行われる.

【注意 2】`shoen:execute/7` の場合と同様に, 生成したプロファイル機能付き荘園の中で PIMOS が管理する資源 (OS 定義資源) を利用する場合は, プロファイル機能付きのタスクを利用しなければならない. プロファイル機能付きタスクについては 第 11 章 資源管理・実行管理 (p.398) を参照のこと.

7.5.3 制御ストリーム

荘園の制御ストリームはユーザから荘園へ向かうメッセージストリームである. 以下のようなメッセージによって荘園の実行や資源消費などを制御することができる.

【参考】制御ストリームに流すメッセージには, 必ず, それに対応する受取り確認メッセージが存在する.

start

荘園の実行を開始する.

作成直後の荘園は停止状態にあり, このメッセージを送ることによって始めて実行可能状態に入る. このメッセージは後述の stop メッセージによって中断した実行を再開する場合にも用いる.

stop

荘園の実行を中断する.

上述の start メッセージによって実行を再開することができる.

abort

荘園の実行を放棄する.

このメッセージにより実行を放棄した場合は, 後に実行を再開することはできない.

statistics

荘園内の実行による資源消費量の報告を指示する.

報告は報告ストリームからなされる.

add_resource(Amount_h, Amount_l)

荘園の実行に資源を与える.

資源消費量の上限値に, 上位 32 ビットが Amount_h, 下位 32 ビットが Amount_l で表現された値を加える.

allow_resource_report

資源消費量の上限が近づいた際の報告を行って良いことを指示する.

通常は報告を行う状態にあるが, いったん上限が接近したことを報告すると, それ以降は報告しない状態に入る. その後このメッセージを受け取ると, また報告する状態に入る. これは余分な報告が数多くなされることを防ぐための機構である.

profile

プロファイル機能付きの荘園に, 途中までのプロファイル情報の集計を要求する.

要求が受け付けられたかどうかは, 報告ストリームからの受取り確認メッセージで確認できる.

7.5.4 報告ストリーム

荘園の報告ストリームは荘園からユーザに向かうメッセージストリームである. 報告ストリームには以下の二種類のメッセージが流れる.

事象報告メッセージ: 荘園の実行にともなって生じた事象を報告するためのメッセージ.

受取り確認メッセージ: 制御ストリームからのメッセージを受け取った時に流すメッセージで, これらと事象報告メッセージのストリーム中の順序によって, 事象報告メッセージに報告した事象と制御メッセージの処理の順序を知ることができる. 例えば, 資源消費上限の追加 (add_resource) と資源消費の上限接近 (resource_low) の前後関係で, さらに追加が必要なのかどうかを知ることができる.

terminated

荘園の終了を報告する.

荘園内のすべてのゴールのリダクションが終了したことによる正常終了か強制的な異常終了かは, 報告ストリーム中に後述の aborted メッセージが先行しているかどうかで知ることができる. このメッセージを報告した後に荘園内のゴールのリダクションやユニフィケーションを行うことはない.

resource_low

資源消費量の上限値が近づいたことを報告する.

【注意】どの程度近づいたらこの報告がなされるかは処理系に依存する.

statistics(Info_h, Info_l)

report 消費資源量の集計報告.

制御メッセージ statistics を受け取ると集計を開始するが, 開始の際には statistics_started(後述) を返し, 集計が完了するとこのメッセージを送る. Info_h は荘園内 (子孫荘園を含む) でこれまでに消費した資源量の合計を表す整数値 (現在の仕様ではリダクション数の二倍) の上位 32 ビットで, Info_l は下位 32 ビットである.

荘園の実行が終了した後に statistics を送ると, このメッセージは正確な値を返す. 荘園が動作中の場合, 報告される消費量は不正確であるが, 実際の消費量を上回ることはないこと, 複数回の集計を行う場合結果は単調増加になることが保証されている.

return_resource(Amount_h, Amount_l)

資源消費量の上限値を与え過ぎたことを報告する.

Amount_h は資源量の上位 32 ビット, Amount_l は下位 32 ビットである. 荘園の実行が終了した時に消費されなかった資源量がこのメッセージによって報告される. また, 実行途中でも消費可能資源量 (資源消費量の上限値からこれまで消費した資源量を引いた量) が既定値 (PIM/m やマルチ PSI では 2^{63}) を越えた場合でも報告される.

started

stopped

aborted

resource_added

resource_report_allowed

statistics_started

それぞれ制御メッセージ start, stop, abort, add_resource, allow_resource_report, statistics に対応する受取り確認メッセージである.

【注意】これらのメッセージの順番により処理の順序を知ることはできるが, 処理の進み具合を知ることはできない. 例えば, stopped メッセージが報告されたからと言って, その時点で荘園内の実行が既に中断されているとは限らない.

profiling_started

profiling_rejected

それぞれ制御メッセージ profile に対応する受取り確認メッセージである.

正当な要求であれば profiling_started が報告される. 不当であれば profiling_rejected を報告する. 不当な場合とは次のような場合である.

1. 既に荘園の実行が終了している場合や, abort されている時.
2. 前回発行した profile メッセージによる途中情報の収集が終了していない時.

3. 実行している荘園の親にあたる荘園もプロファイル機能付きの荘園である場合.
4. profile メッセージを送られた荘園がプロファイル機能付きの荘園でなかった場合.

profiling_completed

制御メッセージ profile に対応するプロファイル情報の収集が完了したことを報告する.

profiling_information(Node, Type, Depth, Data)

プロファイル情報の報告を行う. 報告される情報は次の通り.

Node: ノード情報. 例外メッセージの Node と同じである.

Type: データのタイプ. 次のものがある.

0: 里親の終了により報告されたもの[7]

1: 途中収集要求 profile により報告されたもの

2: カウンタがオーバフローしそうになって報告されたもの[8]

Depth: 荘園のネストの深さ

Data: カウント情報

【注意】カウント情報 Data の形式は, 種々の PIM で異なる可能性がある.

【参考】PIM/m やマルチ PSI では, カウント情報 Data の形式は, 次のようにネストした 5 要素のベクタである.

ユーザ定義述語: { `Code, 0, Dequeue_count, Suspend_count, NextData` }

【注意】リリンク (relink) が不完全である場合, つまり, あるモジュールでは古いままのモジュールを指している場合は, 同じパッケージ名, モジュール名, 述語名, アリティの述語が複数別々に報告される.

組込述語: { `Module, Offset, Dequeue_count, Suspend_count, NextData` }
組込述語はサスペンドしたものだけが報告される[9]. Suspend_count は常に 0.

【注意 1】マージャの実行が中断した場合は, プロファイル情報として報告されない.

【注意 2】一つのボディ部の中で同じ組込述語を複数呼び出している場合, それらの組込み述語の実行が中断した場合には別々に報告される.

【参考】特殊な組込述語が報告されることがある.

enqueue_goal_to_node : ノード指定付きゴールがサスペンドした時.

enqueue_goal_with_priority : 実行優先度指定付きゴールがサスペンドした時. 次の述語も同時に報告される.

calculate_priority_from_minimum: 所属荘園内割合指定

calculate_priority_from_current: 所属荘園内自己相対指定

[7] ゴールがそのゴールを呼び出したゴールとは別のノードで実行される時 (例えばユーザが負荷分散指定を行った時など), そのゴールの転送先プロセサには新たに荘園が生成され, 転送されたゴールはその荘園の中で実行される. この荘園のことを里親荘園と呼ぶ. 所謂, 里子と里親の関係である. 転送されたゴールは里子に相当すると考えて良い.

[8] PIM/m やマルチ PSI では処理系のカウンタが 16#"7FFFFFFF" になったとき.

[9] 組込述語に関しては Dequeue_count だけが報告される. これはその組込述語がサスペンドした回数に対応している (組込述語のサスペンドは 1 リダクションとしてカウントされる).

特殊な命令: { Opecode, 0, Dequeue_count, Suspend_count, NextData }
Suspend_count は常に 0.

【参考】特殊な命令には次のようなものがある.

apply: 以下の場合がある.

1. タスク (荘園) を生成してゴールを実行する時.
2. ノード指定付きゴールがサスペンドした時.
3. 実行優先度指定付きゴールがサスペンドした時.

unify: ボディユニフィケーション. 色々な場合がある. 例えば, マージャの入力引数同士のユニフィケーションが行われたような場合である.

unify_vector: ベクタ同士のボディユニフィケーション. 要素の数だけ報告される. 次のようなプログラムが実行されると考えて良い.

```
unify_vector(0,V1,V2) :-
  vector_element(V1,0,E1), vector_element(V2,0,E2) | E1=E2.
unify_vector(N,V1,V2) :- N>0 |
    set_vector_element(V1,N,OE1,[],NV1),
    set_vector_element(V2,N,OE2,[],NV2),
    OE1=OE2, N1 := N-1,
    unify_vector(N1,NV1,NV2).
```

unify_list: リスト同士のボディユニフィケーション. 要素の数だけ報告される. 次のようなプログラムが実行されると考えて良い.

```
unify_list([X1|X2],[Y1|Y2]) :- true | X1=Y1, X2=Y2.
```

データの終了: [ ]

なお, データ中の Dequeue_count, Suspend_count は次に示す通り.

Dequeue_count: 各述語が実行を開始 (再開) した回数[10]

Suspend_count: 各述語が実行を中断した回数[11]

従って, ゴールの実際のリダクション数は次の計算式より求まる.

```
(リダクション数) = (Dequeue_count) - (Suspend_count)
```

【参考 1】制御メッセージ profile に対応するプロファイル情報は, 受けとり確認メッセージ profiling_started が報告されてから profiling_completed が報告されるまでの間に報告される.

【参考 2】プロファイル情報のカウントは, 各荘園毎, 各ノード毎に行われる. プロファイル情報が報告されると, カウンタは全て 0 にクリアされるため, 荘園全体での実行回数を知りたい場合は, 荘園終了までに報告された全部の情報を加算する必要がある.

【参考 3】カウント情報 Data は後述するプロファイル情報変換ユーティリティを利用して, 処理し易い形式に変換することができる. プロファイル情報変換ユーティリティについては 第 7.5.7.2 節 (p.302) を参照のこと.

[10] dequeue 系の命令, execute 系の命令を実行した回数

[11] suspend 命令を実行した回数

exception(Kind, Info, ^NewCode, ^NewArgV)

荘園の実行中に生じた例外事象を報告する.

Kind は例外の種類を示すアトムである. Info は例外に関する情報のベクタ. NewCode, NewArgV は例外を起こしたゴールに代わって実行すべきゴールを指定するためのコード, 引数ベクタに対応する変数である.

以下に, 各例外の種類 (Kind) に対する例外情報 (Info) の形式を示す. 例外の種類の一覧は 表 7.5「例外タグのビット位置」(p.301) を参照のこと. 例外情報の内容は例外の種類により異なるが, 次にあげる項目は共通である.

Node: 例外を起こしたゴールが存在していたノードの情報. 以下の形式である.

{ノード番号, プロセサ番号}

ノード番号は処理系から報告されるクラスタ番号を相対値に変換したもの, プロセサ番号はノード番号で指示されるクラスタ内のプロセサの番号である[12].

Predicate: 例外を起こした述語の情報とその述語を呼び出している述語の情報. Code と Builtin がある.

Code: コード型データ. KL1 定義述語. コード関連の組込述語 (predicate_to_code や apply など) で操作できる. 別に提供する例外処理用ユーティリティを利用するとモジュール名や述語名などの述語情報を得ることができる. 例外処理用ユーティリティについては 第 7.5.7.1 節 (p.302) を参照のこと.

Builtin: 例外を発生した組込述語の情報. 以下のものがある.

{Code, Caller}: KL1 定義の組込述語 (ライブラリ)

{Name, Caller}: ファームウエア定義の組込述語

{Name, Caller1, Caller2}: ライブラリから呼ばれたファームウエア定義の組込述語. Caller1 は組込述語 Name を呼び出しており, Caller2 は Caller1 を呼び出している. つまり, Caller1 はライブラリに関する情報であり, Caller2 はユーザ定義述語に関する情報である.

Name: 組込述語名アトム.

Caller: {Module, Offset}

Module と Offset は, 例外を発生した述語が, どの KL1 定義述語から呼び出されたものかを示す. 具体的には, Module はその述語を呼び出しているモジュール, Offset はモジュール内におけるその述語の位置, つまり, モジュール先頭からの相対位置を指す.

【参考】別に提供する例外処理用ユーティリティを利用するとモジュール名や述語名などの述語情報を得ることができる. 例外処理用ユーティリティについては 第 7.5.7.1 節 (p.302) を参照のこと.

ArgV: 例外を起こした述語の引数情報ベクタ.

ArgPos: 例外を起こした述語の引数のうち原因となった引数の位置を示す情報. 1 オリジンの番号である.

例外事象の一覧

【参考】それぞれの例外の発生例は 操作編 第 4.1.8.3 節 (p.149) を参照のこと.

[12] プロセサの構成上 PIM/m やマルチ PSI ではプロセサ番号は常に 0 になる.

illegal_input: 入力引数型不正

- {Node, Builtin, ArgV, ArgPos}

第 ArgPos 引数が不正である ($1 \leq ArgPos$).

【参考 1】PIM/m やマルチ PSI では, 実行プライオリティに不正な値を指定した場合 (例えばアトム), Name には次の組込述語名が返される.

calculate_priority_from_minimum : 所属荘園内割合指定

calculate_priority_from_current : 所属荘園内自己相対指定

また, ArgV の第 1 引数 (第 0 要素) はユーザが指定した優先度の割合である.

【参考 2】PIM/m やマルチ PSI では, 実行ノードの指定に不正な値を指定した場合 (例えばアトム), Name には次の組込述語名が返される.

enqueue_goal_to_node : Goal @ node(Node) に対応する.

また, ArgV の第 1 引数 (第 0 要素) はユーザが指定したノード番号, 第 2 引数 (第 1 要素) はノード指定がなされた述語の呼び出し先コードである.

range_overflow: 入力引数範囲不正

- {Node, Builtin, ArgV}
- {Node, Builtin, ArgV, ArgPos}

引数が正しい値の範囲にない.

【参考 1】実行プライオリティに不正な値を指定した場合, Name には次の組込述語名が返される.

calculate_priority_from_minimum : 所属荘園内割合指定

calculate_priority_from_current : 所属荘園内自己相対指定

また, ArgV の第 1 引数 (第 0 要素) はユーザが指定した優先度の割合である.

【参考 2】実行ノードの指定に不正な値を指定した場合, Name には次の組込述語名が返される.

enqueue_goal_to_node: Goal @ node(Node) に対応する.

また, ArgV の第 1 引数 (第 0 要素) はユーザが指定したノード番号, 第 2 引数 (第 1 要素) はノード指定がなされた述語の呼び出し先コードである.

out_of_bounds: インデックス範囲不正

- {Node, Builtin, ArgV}
- {Node, Builtin, ArgV, ArgPos}

構造体のインデクス (第 ArgPos 引数) が範囲を逸脱している.

integer_overflow: 整数演算桁溢れ

- {Node, Builtin, ArgV}

integer_zero_division: ゼロによる除算

- {Node, Builtin, ArgV}

invalid_floating_point_operation: 浮動小数点無効演算

- {Node, Builtin, ArgV}

【参考】以下のような演算を行った場合に無効となる.

```
(+infinity) + (-infinity)  or   (-infinity) + (+infinity)
(-infinity) - (-infinity)  or   (+infinity) - (+infinity)
 (± 0.0)   * (± infinity)  or  (± infinity) *  (± 0.0)
 (± 0.0)   /   (± 0.0)     or  (± infinity) / (± infinity)
```

arity_mismatch: 高階述語の引数個数不整合こすうふせいごう

- {Node, Builtin, ArgV}

undefined_module: 未定義モジュールの呼び出し

- {Node, UndefModInfo, ArgV}

UndefModInfo は未定義モジュール述語の情報. 以下の形式である.

```
{{Package, ModName, Pred, Arity}, Caller}  % Caller と違うパッケージ
{         {ModName, Pred, Arity}, Caller}  % Caller と同じパッケージ
```

Package, ModName, Pred, Arity はそれぞれ未定義モジュール述語のパッケージ名アトム, モジュール名アトム, 述語名アトム, アリティである. また, ArgV は未定議モジュール述語に与えた引数である.

undefined_predicate: 未定義述語の呼び出し

- {Node, UndefPredInfo, ArgV}

UndefPredInfo は未定義述語の情報. 以下の形式である.

```
{{Module, Pred, Arity}, Caller}
```

Module, Pred, Arity はそれぞれ未定義述語のモジュール, 述語名アトム, アリティである. また, ArgV は未定議述語に与えた引数である.

illegal_merger_input: マージャの不正入力

- {Node, Builtin, Input, NewInput}

Input は不正入力データ, NewInput は復帰用の新しいマージャ入力変数. 報告される組込述語名は必ず merge である.

reduction_failure: 全候補節のガードの失敗

- {Node, Code, ArgV}
- {Node, {Code, Caller}, ArgV}

Code は失敗した述語のコード.

unification_failure: ボディユニフィケーションの失敗

- {Node, DataX, DataY}
- {Node, DataX, DataY, Caller}

DataX, DataY はユニフィケーションしようとした二つのデータ.

【参考】DataX と DataY のトップレベルは必ず具体化されている (そうでなければユニフィケーションは失敗することはない) が, 構造体の場合は要素が具体化されているとは限らない.

【注意】構造体同士のユニフィケーションは要素ごとにばらばらに分解されて行われる. このため, たとえば [a, b, c] と [a, b, d] のユニフィケーションの失敗は c と d とのユニフィケーションの失敗として報告される.

deadlock: デッドロック (永久中断) の発生

- {Node, Predicate, ArgV, Type}
- {Node, Predicate, ArgV, Type, Collector}

Type はデッドロック検出の種類を示す整数. Collector はデッドロックを引き起こす原因を作ったゴール (所謂殺人者) の情報である. 具体的にはデッドロックを実時間 GC で検出した場合に付加される情報で, デッドロックゴールが具体化を待っていた HOOK 変数を回収しようとした述語の情報である. 形式は Predicate と同じである. Type には現在以下のものがある.

0: 一括 GC により検出された

1: 一括 GC によりトレース実行中のゴールが検出された

2: 一括 GC によりスパイ実行中のゴールが検出された

10: HOOKo(MGHOKo) 変数を回収しようとした (collect_value 命令)

11: HOOKo(MGHOKo) 変数にサスペンドしようとした (suspend 命令)

12: HOOKo(MGHOKo) 変数同士をユニファイしようとした (unify 命令)

13: VOID 変数にサスペンドしようとした (suspend 命令)

14: VOID 変数と HOOKo 変数をユニファイしようとした (unify 命令)

15: HOOKo 変数に対して

その他の整数: その他のパターン

【参考】デッドロックのタイプと検出例については 操作編 第 4.1.5.3 節 (p.80) を参照のこと.

merger_deadlock: マージャデッドロックの発生

- {Node, Builtin, Inputs, Type}
- {Node, Builtin, Inputs, Type, Collector}

Type はデッドロック検出の種類を示す整数. Collector はデッドロックを引き起こす原因を作ったゴールの情報である. Inputs は捨てられたマージャ入力引数を格納したベクタ. ベクタのサイズは捨てられた入力引数の数に対応し, ベクタの各要素は未定義変数である. この変数に任意のストリームをつなげたり, [] で閉じることが可能である.

trace: トレース例外の検出

- {Node, TraceIDList, SubGoalInfoList}
- {Node, TraceIDList, SubGoalInfoList, DebugInfo}

TraceIDList はゴールを識別するための識別子のリストで, 識別子は組込述語 apply_tracing の第 3 引数に指定したものである. SubGoalInfoList はサブゴール情報 SubGoalInfo のリストである. DebugInfo はコンパイラが付加したデバック情報であり, 現在はコミットしたクローズの番号 (正整数) が報告される.

【注意】将来, DebugInfo には別の情報が付加される可能性がある.

【参考】トレース時に, ゴールがコミットしたクローズの番号を出力させることができる. クローズ番号の出力方法は 操作編 第 4.1.7.5 節 (p.124) を参照のこと.

【参考】SubGoalInfo は以下の形式をしている.

{KlbName, Code, ArgV}: ユーザ定義述語

{KlbName, Code, ArgV, Priority}: 優先度指定

{KlbName, Code, ArgV, ClsOrProc}: クラスタまたはプロセサ指定

{unify, {DataX, DataY}}: ボディユニフィケーション

{Name, ArgV}: ボディ組込述語

KlbName は処理系の命令名アトム (KL1-B 命令) である. KL1-B 命令は機種依存であるため (特にプラグマ指定付きのもの), トレーサの開発者は各種処理系およびこれらの命令の意味を良く知っていなければならない. ここでは PIM/m (マルチ PSI) における命令の種類を記す.

{**execute, Code, ArgV**}: 同一モジュール内ユーザ定義述語[13]

{**execute_external, Code, ArgV**}: 外部モジュールユーザ定義述語[14]

{**enqueue, Code, ArgV**}: 同一モジュール内ユーザ定義述語

{**enqueue_with_priority, Code, ArgV, {Prio, Min, Max}**}:
優先度指定付き同一モジュール内ユーザ定義述語. {Prio, Min, Max} はゴールに指定された論理優先度を表し. Prio はユーザが指定した論理優先度[15], Min, Max はそのゴールが所属する荘園の最低および最高優先度である.

{**enqueue_to_processor, Code, ArgV, Node**}: ノード指定付き同一モジュール内ユーザ定義述語. Node はユーザが指定したノードの番号である.

{**enqueue_external, Code, ArgV**}: 外部モジュールユーザ定義述語

{**unify, {DataX, DataY}**}: ボディユニフィケーション. DataX, DataY はユニフィケーション対象の 2 つのデータである.

{**Name, ArgV**}: ボディ組込述語

spy: スパイ例外の検出

- {Node, SpyID, SubGoalInfo, Caller}

SpyID はゴールを識別するための識別子であり, 組込述語 apply_spying の第 4 引数に指定したものである. SubGoalInfo はトレース例外の場合と同じ形式のサブゴール情報である.

【注意 1】但し, enqueue_with_priority 命令の引数として報告される優先度は, トレース例外の場合の {Prio,Min,Max} の 3 要素ベクタではなく, Prio のみである.

【注意 2】スパイ例外は apply などの高階述語の引数として与えられたコードがスパイ対象の述語である場合には, その高階述語の名前と引数が SubGoalInfo に報告される[16].

raised: 積極的な例外の発生

- {Node, Info, Data}

述語 shoen:raise/3 により意図的に例外が起こされた. Info と Data は組込述語 raise の引数に与えられたもの. 積極的な例外の生起の仕方については 第 7.5.5 節 (p.300) を参照のこと.

etc: 上記以外の例外の発生

- {ExpID, ExpInfo}

ExpID は例外の種類を表す識別子, ExpInfo は例外情報である (変数が含まれている可能性がある).

【参考】この例外は, 当面発生されない.

7.5.5 例外の生起

例外事象は組込述語などで生ずる以外に, 以下の述語を用いて積極的に生起することもできる.

shoen:raise(Tag, Info, Data)

例外を生起する. Tag は例外のタグである. 例外タグについては第 7.5.6 節 (p.301) を参照のこと.

例外の生起は Info が構造体の場合, その要素も, 要素がまた構造体の場合, そのまた要素もすべて完全に具体化されるまで待たされる. Data に関してはこのような制限はない. これにより, 例外処理側

[13] サブゴールの中で最初に実行されるゴール

[14] サブゴールの中で最初に実行されるゴール

[15] 未定義変数である. Prio は組込述語 calculate_priority_from_XXX の実行により具体化される.

[16] apply, apply_tracing, apply_spying, create_shoen, create_profiling_shoen がある.

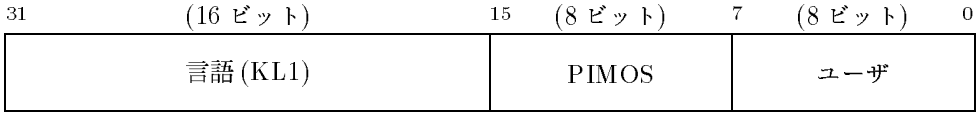

図 7.2: 例外タグのビット位置割り当て

表 7.5: 例外タグのビット位置

例外の種類	ビット位置	意味
illegal_input	16	入力引数型不正
range_overflow	17	入力引数範囲不正
out_of_bounds	18	インデクス範囲不正
integer_overflow	19	整数演算桁あふれ
integer_zero_division	20	ゼロによる除算
invalid_floating_point_operation	21	浮動小数点数無効演算
arity_mismatch	22	引数個数不整合
undefined_module	23	存在しないモジュールの呼びだし
undefined_predicate	23	存在しない述語の呼びだし
illegal_merger_input	24	マージャ入力不正
reduction_failure	25	全候補節の失敗
unification_failure	26	ユニフィケーション失敗
deadlock	27	デッドロック検出
merger_deadlock	28	マージャデッドロック検出
trace	29	トレース
spy	29	スパイ
(reserved)	30	
etc	31	その他

が Info の内容だけを調べている限り, 具体化されていないことによって例外処理が中断することはないようになっている.

【参考 1】 Data は例外処理側で内容を調べないで, 直接 NewArgV の要素として raise の代わりに実行されるゴールの引数とするのが普通の使い方である.

【参考 2】 shoen:raise/3 により積極的に発生された例外は, その引数として与えた例外タグ Tag と荘園生成時に与える例外処理を決定するマスクパターンの論理積が 0 でないような最も内側の荘園の報告ストリームに **raised** の例外として報告される (第 7.5.4 節 p.300 参照のこと).

7.5.6 例外タグ

例外タグのビット位置は図 7.2 に示すように言語で定義するもの, PIMOS が定義するもの, ユーザが定義できるものの 3 つの領域に分けられている. ユーザプログラム同士で例外 (raise) を用いて通信する時の例外タグのビット位置は, 0 から 7 までででなければならない.

言語定義の例外事象に対する例外タグのビット位置を表 7.5 に示す.

【参考】 raised については, raise を呼び出す時に例外タグを指定するので, 例外タグのビット位置は決まっていない.

【注意】この例外タグのビット位置は将来変更される可能性があるので, 荘園 (またはタスク) のマスクパターンの指定にはできるだけ PIMOS で用意しているマクロを使用した方が良い. このマクロは, “exception_tag# 例外事象名” とすることにより, 各例外に対応するマスクパターンに変換される. 複数の例外事象に対するマスクパターンは, このマクロ表記を用いてそれぞれの論理和を取れば良い.

7.5.7 例外処理用ユーティリティ

7.5.7.1 述語情報の取り出し

例外事象の報告時に返される Info の Caller または Code から, 以下の述語を用いることによりその例外事象がどの述語で起きたかを知ることができる.

module:get_predicate_info(CallerOrCode,^Package,^ModName,^Pred,^Arity,^Info)

第一引数に Caller({Module,Offset}) 情報を与えた場合は, モジュール Module 内の Offset 位置にある命令 (例外を発生した組込述語) を呼び出している KL1 定義述語の情報を取り出す. 得られる情報はパッケージ名アトム (Package), モジュール名アトム (ModName), 述語名アトム (Pred), アリティ (Arity) である.

また, 第一引数にコード (Code) を指定した場合は, そのコードが属するモジュールのパッケージ名, モジュール名, 述語名, アリティを返す.

いずれの場合でも最後の引数 (Info) には, コンパイラが埋め込んだ情報 (現在は整数ゼロが返されるが、将来はソースファイルの相当する位置を返すようになる可能性がある) が返される.

module:get_module_name(Module,^Package,^ModName)

モジュール (Module) より, パッケージ名アトム (Package), モジュール名アトム (ModName) を返す.

module:get_module_name(Module,^ModSpec)

モジュール (Module) より, パッケージ名付きモジュール名 (ModSpec) を返す. ModSpec の形式は {::, Package, ModName} である.

7.5.7.2 荘園プロファイル情報変換ユーティリティ

exception_table:translate_profiling_data(Data,^NewData)

profiling_information(Node,Type,Depth,Data) の Data を以下の形式に変換する.

- {Predicate, Dequeue_count, Suspend_count}
  ユーザ定義述語 または ライブラリ または 組込述語. Predicate の形式は例外情報中の形式と同じ.
- {Name, Dequeue_count, Suspend_count}
  特殊な命令. Name は命令名アトム.

第 8 章

ユーティリティ

PIMOS では, PIMOS, 応用プログラムの別を問わず広く利用できる共通のユーティリティとして, 以下の機能を提供する.

- 任意の KL1 データの間の大小比較
- 任意の KL1 データに対するハッシュ関数の計算
- 任意のデータの格納 / 取り出しを行うデータベース (プール)
- 任意のデータ集合のソート / マージを行うフィルタ (ソーティング / マージング用フィルタ)
- 文字データの格納 / 取り出しを行うデータベース (無限長バッファ)
- 文字データのブロッキング / デブロッキングを行うフィルタ (バッファリングフィルタ)

これらのユーティリティは, 2 種類のうちのいずれかの利用形式をとる. ひとつは PIMOS が提供するモジュールの述語の呼び出しによるものであり, もうひとつはそのような述語を呼び出すことによって得られるストリームへメッセージを送るものである. 本章では入出力ユーティリティを除くその他のユーティリティについて述べる. 入出力ユーティリティについてはプログラミング編 9.2節「入出力ユーティリティ」(338頁) を参照.

【注意】なお, ここに記述する述語の呼出しによって得られるストリームのうち, あらかじめ組込述語のマージャ (プログラミング編 7.4.19節 組込述語「ストリームサポート」(285頁) を参照) が挿入されているものには, ストリームを表す変数の右側に'*' が付加されている. 表示の読み方について詳しくは, プログラミング編 7.1節「モード表示の読み方」(222頁) を参照.

8.1 比較

任意の KL1 データを一意に比較できる機構を提供する.

8.1.1 標準の比較ルーチン

comparator:sort(X, Y, ^S, ^L, ^Swapped)

X と Y の大小関係が決まるまで待ち, 小さい方を S に, 大きい方を L に返す. 両者が等しい場合は, X を S に, Y を L に返す (このような性質を stable であるという). また, X が Y より大きかった場合は Swapped に yes を, さもなければ no を返す. 大小関係の定義については後述.

comparator:descending_sort(X, Y, ^L, ^S, ^Swapped)

X と Y の大小関係が決まるまで待ち, 大きい方を L に, 小さい方を S に返す. 両者が等しい場合は, X を L に, Y を S に返す. また, X が Y より小さかった場合は Swapped に yes を, さもなければ no を返す. 大小関係の定義については後述.

8.1.2 標準比較ルーチンでのデータの大小関係

上記の述語呼び出しによって決定されるデータの大小関係は, 以下のように定義されている.

比較される両引数のデータ型が等しい場合, その大小関係は各データ型に従って以下のように定義される:

整数: 通常の符号つき整数の大小関係.

浮動小数点数: 通常の符号つき浮動小数点数の大小関係.

アトム: アトム番号の大小関係.

文字列: 要素のビット長が異なる場合はその大小関係. ビット長が同じ場合, 辞書式順序[1] の大小関係.

リスト: Car の大小関係. Car が等しければ, Cdr の大小関係.

ベクタ: 要素数が異なる場合はその大小関係. 要素数が等しければ第一要素, それも等しければ第二要素, 等々の大小関係.

モジュール: 要素数が異なる場合はその大小関係. 要素数が等しければ第一要素, それも等しければ第二要素, 等々の大小関係.

コード: そのコードの属するモジュールの大小関係. モジュールが等しければオフセットの大小関係.

その他: データのタグ部の大小関係. タグ部が等しければ値部の大小関係.

比較される両引数のデータ型が異なる場合, その大小関係はデータ型間の大小関係として, 以下の順に定義される:

整数 < **浮動小数点数** < **アトム** < **文字列** < **リスト** < **ベクタ** < **モジュール** < **コード** < **その他**

ただし, 整数と浮動小数点数の比較は, その具体的数値の大小関係を比較することによって決定する. 数値が等しい場合は, 整数, 浮動小数点数の順とする.

【注意】データ型間の大小関係は, 将来変更される可能性がある.

8.1.3 比較ルーチンの指定方法

多くのプールでは, 要素間の大小関係に基づいて要素の格納・取り出しを行う. 比較に用いるルーチンはプログラム中に指定することもでき, その場合, 以下のいずれかを用いる.

ascending (アトム): 標準の昇順比較ルーチン comparator:sort/5 (8.1.1節「標準の比較ルーチン」 303頁参照)

descending (アトム): 標準の降順比較ルーチン comparator:descending_sort/5 (8.1.1節「標準の比較ルーチン」 304頁参照)

Code(コード): ユーザ定義比較ルーチン. ただし Code は標準の比較ルーチン comparator:sort/5 (8.1.1節「標準の比較ルーチン」 303頁参照) と同一インタフェースでなければならない.

[1] Lexicographic order.

8.2 ハッシング

標準のハッシュ関数を提供する.

hasher:hash(X, ^H, ^Y)

X を適当なハッシュ関数でハッシュして, その値を H (非負整数) に返す. また, Y に X をそのまま返す.

【注意】ハッシュ関数の仕様は将来変更される可能性が大きいので, あえて記述しないが, 組込み述語 hash (プログラミング編 7.4.21節 組込述語「その他」(287頁) を参照) のものよりも構造体の内部までハッシュ関数計算に用いるようになっている.

8.3 プール

任意の KL1 データを格納しておける仕組みを提供する. 探索のキーや比較に使うものを除き, 格納するデータは未定義状態でも良い.

プールには大きく分けて, データ自身を探索のキーとして格納する「キーなしプール」と, データをキー付きで格納する「キー付きプール」の 2 種類がある. このうち, キーなしプールには以下の 4 種類のプールがある:

Bag: 最も基本的なプール. 8.3.1節「Bag」(306頁) 参照.

Stack: bag と同様だが, データの追加及び取り出しを LIFO で行う. 8.3.2節「Stack」(307頁) 参照.

Queue: bag と同様だが, データの追加及び取り出しを FIFO で行う. 8.3.3節「Queue」(307頁) 参照.

Sorted bag: bag と同様だが, データを「小さい順」に取り出す. 8.3.4節「Sorted bag」(308頁) 参照.

データをキー付きで格納し, そのキーを用いてデータの取り出しや参照を行うプールは, 以下のように大きく 2 種類に分けられる. ひとつはデータに対してキーをただひとつしかつけられないもの, もうひとつは, データに対して複数のキーをつけることのできるものである. 前者を「キー付きプール」, 後者を「複数キー付きプール」と呼ぶ. これらのプールには以下の種類がある:

Keyed bag: 最も基本的なキー付きプール. 8.3.5節「Keyed bag」(309頁) 参照.

Keyed set: keyed bag と同様だが, キーの重複を許さない. 8.3.6節「Keyed set」(311頁) 参照.

Keyed sorted bag: keyed bag と同様だが, データを「小さい順」に取り出す. 8.3.7節「Keyed sorted bag」(312頁) 参照.

Keyed sorted set: keyed bag と同様だが, キーの重複を許さず, データを「小さい順」に取り出す. 8.3.8節「Keyed sorted set」(313頁) 参照.

Multiply keyed bag: 複数キー付きプール. 8.4節「Multiply keyed bag」(315頁) 参照.

これらのプールは, それぞれ指定された述語を呼び出すことによってプロセスを起動し, それらに対してストリームを介したメッセージ通信を行うことにより操作する. 以下に各プールのプロセスの生成方法及びメッセージプロトコルを示す.

【注意】以下のメッセージプロトコルの記述で, プロトコル中の変数に付加されている記号 ^, ? 等は, 7.1節のモード表示方法に従うものである. プールでは一般に探索のキーや比較に用いるものを除き, 格納データは未定義状態でもかまわないので, そのようなデータの格納及び取り出しでは, モード表示は ? となる. 詳しくは, プログラミング編 7.1節「モード表示の読み方」(222頁) 参照.

8.3.1 Bag

入れる, 出すという基本機能しかない基本的なキーなしのプールである. 要素を参照するには取り出すしかなく, 中に残したまま参照する方法はない. 以下の述語を呼び出してプロセスを生成し, メッセージを送って操作する.

pool:bag(S*)

新しい bag を作り, そこへのストリーム S を得る.

Bag に送ることのできるメッセージは以下の通り.

empty(^YorN)

Bag が空かどうかを問い合わせる. Bag が空の場合, YorN を yes に, そうでない場合, no に具体化する.

put(X)

Bag に要素 X を追加する.

get(^X)

Bag から要素をひとつ取り出し X に返す. Bag 内のどの要素が取れてくるかはわからない. 取り出すと bag からなくなる. Bag が空の場合は失敗する.

get_if_any(^X)

Bag に要素があればひとつ取り出し, X に { 値 } の形式で返す. Bag 内のどの要素が取れてくるかはわからない. 取り出すと bag からなくなる. Bag が空の場合は X に {} を返す.

get_all(^O)

Bag の要素を全て取り出し O に返す. O の形式は要素のリストである. リストに返される要素の順は特に決まっていない. Bag が空の場合は [] を返す.

carbon_copy(^O)

Bag の要素を全部参照し, O に返す. O の形式は要素のリストである. リストに返される要素の順は特に決まっていない. Bag が空の場合は [] を返す.

【注意】このメッセージの実行により参照された要素の MRB は黒となる. 白のまま参照したいユーザは, 取り出し操作 (get/1 など) と格納操作 (put/1) の 2 操作を用いるのがよい.

do(Stream*)

Stream にメッセージを要素とするリストを渡すと, Stream 内のメッセージは連続して処理される.

【参考】ひとつのプールに対して複数のプロセスがアクセスする場合, そのうちのひとつのプロセスからの一連の操作を連続して処理したい場合に有効である.

8.3.2 Stack

基本的には bag と同様のキーなしのプールである. ただし, 要素の取り出される順が, 要素を追加した順とは逆順 (LIFO) となる点が異なる. 以下の述語を呼び出してプロセスを生成し, メッセージを送って操作する.

pool:stack(S*)

新しい stack を作り, そこへのストリーム S を得る.

Stack に送ることのできるメッセージは以下の通り, 全て bag と同様である. ただし, 要素をひとつ取り出す際には最も新しく追加されたものから順に取り出される. Stack 内の全要素を取り出したり参照したりする際には, 最も新しく追加されたものから順に返される.

empty(^YorN) (306頁参照)

put(X) (306頁参照)

get(^X) (306頁参照)

get_if_any(^X) (306頁参照)

get_all(^O) (306頁参照)

carbon_copy(^O) (306頁参照)

do(Stream*) (306頁参照)

8.3.3 Queue

基本的には bag と同様のキーなしのプールである. ただし, 要素の取り出される順が, 要素を追加した通りの順 (FIFO) となる点が異なる. 以下の述語を呼び出してプロセスを生成し, メッセージを送って操作する.

pool:queue(S*)

新しい queue を作り, そこへのストリーム S を得る.

Queue に送ることのできるメッセージは以下の通り, 全て bag と同様である. ただし, 要素をひとつ取り出す際には先に追加されたものから順に取り出される. Queue 内の全要素を取り出したり参照したりする際には, 先に追加されたものから順に返される.

empty(^YorN) (306頁参照)

put(X) (306頁参照)

get(^X) (306頁参照)

get_if_any(^X) (306頁参照)

get_all(^O) (306頁参照)

carbon_copy(^O) (306頁参照)

do(Stream*) (306頁参照)

8.3.4 Sorted bag

基本的には bag と同様のキーなしのプールである. ただし, 要素の取り出しは指定された比較ルーチンに従って「小さい順」となる点が異なる. 比較ルーチンが stable ならば (比較結果が等しいものの順序は入れ替えないようなものなら), 大小関係で等しいものについてはその順を保存して取り出される.

【注意】将来, sorted bag における探索アルゴリズムは並列性の高いものに変更される可能性がある. それに伴い, 格納要素の MRB は黒となる可能性が高い.

以下の述語を呼び出してプロセスを生成し, メッセージを送って操作する.

pool:sorted_bag(S*)

新しい sorted bag を作り, そこへのストリーム S を得る. 要素の比較には標準の昇順比較ルーチン comparator:sort/5(8.1.1節「標準の比較ルーチン」 303頁参照) を用いる.

pool:sorted_bag(Comparator, S*)

新しい sorted bag を作り, そこへのストリーム S を得る. 要素の比較には Comparator に指定された比較ルーチンを用いる. Comparator の指定方法は 8.1.3節「比較ルーチンの指定方法」 (304頁) を参照のこと.

Sorted bag に送ることのできるメッセージは以下の通り. ただし, 要素をひとつ取り出す際にはその時点での “最小” のものから順に取り出される. Sorted bag 内の全要素を取り出したり参照したりする際には, “小さい順” に返される.

empty(^YorN) (306頁参照)

put(X) (306頁参照)

get(^X) (306頁参照)

get_if_any(^X) (306頁参照)

get_all(^O) (306頁参照)

carbon_copy(^O) (306頁参照)

do(Stream*) (306頁参照)

get_max_if_any(^X)

Sorted bag に要素があればそのうち最も大きいものをひとつ取り出し, X に { 値 } の形式で返す. 比較が stable で, 最も大きいものが複数存在する場合は, 入れた順に取り出す. 取り出すと sorted bag からなくなる. Sorted bag が空の場合は X に {} を返す.

get_min_if_any(^X)

Sorted bag に要素があればそのうち最も小さいものをひとつ取り出し, X に { 値 } の形式で返す. 比較が stable で, 最も小さいものが複数存在する場合は, 入れた順に取り出す. 取り出すと sorted bag からなくなる. Sorted bag が空の場合は X に {} を返す.

8.3.5 Keyed bag

任意の KL1 データをキー付きで格納し, キーをもとに検索できる仕組みを提供する, 基本的なキー付きプールである. ハッシュ表により実現されている. ハッシュ表の大きさの初期値は規定値或いは生成時に指定される引数によって決定されるが, 要素数が増えハッシュ表が混雑してくると自動的に表を大きくするようになっている. 以下の述語を呼び出してプロセスを生成し, メッセージを送って操作する.

pool:keyed_bag(S*)

新しい keyed bag をを作り, そこへのストリーム S を得る. 要素のハッシュには標準のハッシュ関数 hasher:hash/3 (8.2節「ハッシング」305頁参照) を用いる. ハッシュ表の大きさの初期値は 1 である.

pool:keyed_bag(S*, Size)

新しい keyed bag をを作り, そこへのストリーム S を得る. 要素のハッシュには標準のハッシュ関数 hasher:hash/3 (8.2節「ハッシング」305頁参照) を用いる. ハッシュ表の大きさの初期値を Size(非負整数) とする.

pool:keyed_bag(Hasher, S*, Size)

新しい keyed bag をを作り, そこへのストリーム S を得る. 要素のハッシュには Hasher (コード) を用いる. ただし, Hasher は標準のハッシュ関数 hasher:hash/3 (8.2節「ハッシング」305頁参照) と同一インタフェースでなければならない. ハッシュ表の大きさの初期値を Size(非負整数) とする.

Keyed bag に送ることのできるメッセージは以下の通り.

empty(^YorN)

Keyed bag が空かどうかを問い合わせる. 空の場合, YorN を yes に, そうでない場合, no に具体化する.

empty(Key, ^YorN)

Keyed bag の中にキーが Key の要素があるかどうかを問い合わせる. 対応する要素がない場合 YorN を yes に, そうでない場合 no に具体化する.

put(Key, ?X)

Keyed bag にキーが Key, 値が X の要素を追加する.

get(Key, ?X)

Keyed bag からキーが Key の要素をひとつ取り出し, X に返す. 同じキーのものが複数ある場合は, どれが取れてくるかはわからない. 取り出すと keyed bag からなくなる. 対応する要素がない場合は失敗する.

get_if_any(Key, ^X)

Keyed bag にキーが Key の要素があればひとつ取り出し, X に { ? 値 } の形式で返す. 同じキーのものが複数ある場合は, どれが取れてくるかはわからない. 取り出すと keyed bag からなくなる. 対応する要素がない場合は X に {} を返す.

get_all(^O)

Keyed bag の要素を全部取り出し, O に返す. O の形式は { キー, ? 値 } のリストである. Keyed bag が空の場合は [] を返す.

get_all(Key, ^O)

Keyed bag からキーが Key の要素を全部取り出し, O に返す. O の形式は値のリストである. 対応する要素がない場合は [] を返す.

carbon_copy(^O)

Keyed bag の要素を全部参照し, O に返す. O の形式は { キー, ? 値 } のリストである. Keyed bag が空の場合は [] を返す.

【注意】このメッセージの実行により参照されたキーと値の MRB は黒となる. 白のまま参照したいユーザは, 取り出し操作と格納操作の 2 操作 (get/2, put/2 など) を用いるのがよい.

carbon_copy(Key, ^O)

Keyed bag からキーが Key の要素を全部参照し, O に返す. O の形式は値のリストである. 対応する要素がない場合は [] を返す.

【注意】このメッセージの実行により参照された値の MRB は黒となる. 白のまま参照したいユーザは, 取り出し操作と格納操作の 2 操作 (get/2, put/2 など) を用いるのがよい.

get_and_put(Key, ?X, ?Y)

Keyed bag からキーが Key の要素を取りだして X に返し, 同じ 位置に新しい要素 Y を格納する. 対応する要素がない場合は失敗する.

get_if_any_and_put(Key, ^X, ?Y)

Keyed bag にキーが Key の要素があればひとつ取り出し, X に { ? 値 } の形式で返す. また, 同じ位置に新しい要素 Y を格納する. 対応する要素がない場合は X に {} を返し, Y を新たに追加する.

get_and_put_if_any(Key, ^X, ?Y, ^Y1)

Keyed bag にキーが Key の要素があればひとつ取り出し, X に { ? 値 } の形式で返す. また, 同じ位置に新しい要素 Y を格納し, Y1 に {} を返す.

対応する要素がない場合は X に {} を返す. また, Y1 に { ?Y } を返し, keyed bag に対しては何も行わない.

get_load_factor(^A)

メッセージ受信時のハッシュ表のロードファクタを A に返す. ロードファクタは, ハッシュ表全体に対する使用中のエントリ数の割合を示す. 0 以上 1 以下の浮動小数点数で表される.

【注意】現在ハッシュ表は, ロードファクタが 0.75 を越えた時点で自動的に 2 倍の大きさにするようになっている.

get_average_chaining_length(^L)

メッセージ受信時のハッシュ表内の要素の平均連鎖長を L に返す. ハッシュ表の衝突処理は, 分離連鎖法 (separate chaining) によって実現されている. これにより同一ハッシュ値を持つ要素 (同類語) は同一の連鎖につながれる. 平均連鎖長は, ハッシュテーブル内に存在する (長さ 1 以上の) 連鎖の平均長を示す. 1.0 以上の浮動小数点数で表される. 特にハッシュテーブルが空の場合, L には 0.0 を返す.

【参考】ハッシュ表内の衝突の度合いを知る目安として使用するとよい. 例えば, ロードファクタの値が小さいにもかかわらず平均連鎖長の値が大きい場合には, 表内で衝突が多く起こっている可能性が高い. このような場合には, キーを変更するか, ハッシュ関数を再定義して用いるのが望ましい.

do(Stream*)

Stream にメッセージを要素とするリストを渡すと, Stream 内のメッセージは連続して処理される.

【参考】ひとつのプールに対して複数のプロセスがアクセスする場合, そのうちのひとつのプロセスからの一連の操作を連続して処理したい場合に有効である.

8.3.6 Keyed set

基本的には keyed bag と同様のキー付きプールである. ただし, キーの重複が許されない点が異なる. ハッシュ表により実現されている. ハッシュ表の大きさの初期値は規定値或いは生成時に指定される引数によって決定されるが, 要素数が増えハッシュ表が混雑してくると自動的に表を大きくするようになっている. 以下の述語を呼び出してプロセスを生成し, メッセージを送って操作する.

pool:keyed_set(S*)

新しい keyed set ををを作り, そこへのストリーム S を得る. 要素のハッシュには標準のハッシュ関数 hasher:hash/3 (8.2節「ハッシング」 305頁参照) を用いる. ハッシュ表の大きさの初期値は 1 である.

pool:keyed_set(S*, Size)

新しい keyed set ををを作り, そこへのストリーム S を得る. 要素のハッシュには標準のハッシュ関数 hasher:hash/3 (8.2節「ハッシング」 305頁参照) を用いる. ハッシュ表の大きさの初期値を Size(非負整数) とする.

pool:keyed_set(Hasher, S*, Size)

新しい keyed bag ををを作り, そこへのストリーム S を得る. 要素のハッシュには Hasher (コード) を用いる. ただし, Hasher は標準のハッシュ関数 hasher:hash/3 (8.2節「ハッシング」 305頁参照) と同一インタフェースでなければならない. ハッシュ表の大きさの初期値を Size(非負整数) とする.

Keyed set に送ることのできるメッセージは以下の通り.

empty(^YorN) (309頁参照)

empty(Key, ^YorN) (309頁参照)

put(Key, ?X, ^OldX)

Keyed set にキーが Key, 値が X の要素を追加する. キーが Key のものが既にあれば更新され, OldX に前の値が一要素ベクタ {? 値 } の形で返される. キーが Key のものがなければ OldX には空ベクタ {} が返される.

get(Key, ?X) (309頁参照)

get_if_any(Key, ?X) (309頁参照)

get_all(^O) (310頁参照)

get_all(Key, ^O) (310頁参照)

carbon_copy(^O) (310頁参照)

carbon_copy(Key, ^O) (310頁参照)

get_and_put(Key, ?X, ?Y) (310頁参照)

get_if_any_and_put(Key, ^X, ?Y) (310頁参照)

get_and_put_if_any(Key, ^X, ?Y, ^Y1) (310頁参照)

get_load_factor(^A) (310頁参照)

get_average_chaining_length(^L) (311頁参照)

8.3.7 Keyed sorted bag

基本的には keyed bag と同様のキー付きプールである. ただし, 要素の取り出しは指定された比較ルーチンに従ってキーの 「小さい順」 となる点が異なる.

【注意】将来, keyed sorted bag における探索アルゴリズムは並列性の高いものに変更される可能性がある. それに伴い, 格納要素のキーの MRB は黒となる可能性が高い. 以下の述語を呼び出してプロセスを生成し, メッセージを送って操作する.

pool:keyed_sorted_bag(S*)

新しい keyed sorted bag を作り, そこへのストリーム S を得る. 要素の比較には標準の昇順比較ルーチン comparator:sort/5(8.1.1節「標準の比較ルーチン」 303頁参照) を用いる.

pool:keyed_sorted_bag(Comparator, S*)

新しい sorted bag を作り, そこへのストリーム S を得る. 要素の比較には Comparator に指定された比較ルーチンを用いる. Comparator の指定方法は 8.1.3節 「比較ルーチンの指定方法」 (304頁) を参照のこと.

Keyed sorted bag に送ることのできるメッセージは以下の通り. ただし, 要素をひとつ取り出す際にはその時点でキーが “最小” のものから順に取り出される. 比較が stable で同じキーのものが複数ある場合は, 先に追加されたものから順に取り出される. Keyed sorted bag 内の全要素を取り出したり参照したりする際には, “小さい順” に返される.

empty(^YorN) (309頁参照)

empty(Key, ^YorN) (309頁参照)

put(Key, ?X) (309頁参照)

get(Key, ?X) (309頁参照)

get_if_any(Key, ?X) (309頁参照)

get_all(^O) (310頁参照)

get_all(Key, ^O) (310頁参照)

carbon_copy(^O) (310頁参照)

carbon_copy(Key, ^O) (310頁参照)

get_and_put(Key, ?X, ?Y) (310頁参照)

get_if_any_and_put(Key, ^X, ?Y) (310頁参照)

get_and_put_if_any(Key, ^X, ?Y, ^Y1) (310頁参照)

get_max_if_any(^Element)

Keyed sorted bag に要素があればそのうち最も大きいキーを持つものをひとつ取り出し, Element に { キー, ? 値 } の形式で返す. 比較が stable で, 最も大きいキーを持つ要素が複数ある場合は, 入れた順に取り出す. 取り出すと keyed sorted bag からなくなる. Keyed sorted bag が空の場合は Element に {} を返す.

get_min_if_any(^Element)

Keyed sorted bag に要素があればそのうち最も小さいキーを持つものをひとつ取り出し, Element に { キー, ? 値 } の形式で返す. 比較が stable で, 最も小さいキーを持つ要素が複数ある場合は, 入れた順に取り出す. 取り出すと keyed sorted bag からなくなる. Keyed sorted bag が空の場合は Element に {} を返す.

8.3.8 Keyed sorted set

基本的には keyed sorted bag と同様のキー付きプールである. ただし, キーの重複が許されない点が異なる. 同一キーの要素を追加すると, そのキーの元の要素は自動的に削除される.

【注意】将来, keyed sorted set における探索アルゴリズムは並列性の高いものに変更される可能性がある. それに伴い, 格納要素のキーの MRB は黒となる可能性が高い. 以下の述語を呼び出してプロセスを生成し, メッセージを送って操作する.

pool:keyed_sorted_set(S*)

新しい keyed sorted set を作り, そこへのストリーム S を得る. 要素の比較には標準の昇順比較ルーチン comparator:sort/5(8.1.1節「標準の比較ルーチン」 303頁参照) を用いる.

pool:keyed_sorted_set(Comparator, S*)

新しい sorted bag を作り, そこへのストリーム S を得る. 要素の比較には Comparator に指定された比較ルーチンを用いる. Comparator の指定方法は 8.1.3節 「比較ルーチンの指定方法」 (304頁) を参照のこと.

Keyed sorted set に送ることのできるメッセージは以下の通り. ただし, 要素をひとつ取り出す際にはその時点でキーが “最小” のものから順に取り出される. Keyed sorted set 内の全要素を取り出したり参照したりする際には, “小さい順” に返される.

empty(^YorN) (309頁参照)

empty(Key, ^YorN) (309頁参照)

put(Key, ?X, ^OldX) (312頁参照)

get(Key, ?X) (309頁参照)

get_if_any(Key, ?X) (309頁参照)

get_all(^O) (310頁参照)

get_all(Key, ^O) (310頁参照)

carbon_copy(^O) (310頁参照)

carbon_copy(Key, ^O) (310頁参照)

get_and_put(Key, ?X, ?Y) (310頁参照)

get_if_any_and_put(Key, ^X, ?Y) (310頁参照)

get_and_put_if_any(Key, ^X, ?Y, ^Y1) (310頁参照)

get_max_if_any(^Element) (313頁参照)

get_min_if_any(^Element) (313頁参照)

8.3.9 キーなし及びただひとつのキー付きプールが表示するエラーメッセージ

キーなしプール及びただひとつのキー付きプールは, 定義されていないメッセージを受信した場合や存在しない要素を取り出そうとした場合などにエラー報告を行う. 報告の内容及び形式を以下に示す. なお, シェル上でエラーが報告される場合の表示形式について, 詳しくは操作編 2.1.5.2節 シェルの「エラーメッセージ」(25, 26頁) を, リスナ上でエラーが報告される場合の表示形式について, 詳しくは操作編 4.1.8.3節 リスナの「エラーメッセージ」【例 39】(156頁) を参照.

プールが, 定義されていないメッセージを受信した場合

エラー ID: 使用したプール名

エラー情報: illegal_message_protocol

エラーデータ: エラーを生じたメッセージ

【例 1】リスナ上で実行中にエラーが検出された場合の報告例

```
[1] pool:keyed_bag([unknown(X)|In]).
  Error>> keyed_bag
   Info > illegal_message_protocol
   Data > unknown(X)
   Node > 0
 !! Exception (h for help):
```

【例 2】シェル上で実行中にエラーが検出された場合の報告例

```
Shell>> del:go
 Error reported !!
  Job-number : 2
  Node-number: 0
  Error ID   : keyed_bag
  Error Info : illegal_message_protocol
  Error Data : unknown(A)
 Job 2 was aborted .
Shell>>
```

プールから, 存在しない要素を取り出そうとした場合

エラー ID: 使用したプール名

エラー情報: tried_to_get_nonexistent_element

エラーデータ: エラーを生じたメッセージ

【例】リスナ上で実行中にエラーが検出された場合の報告例

```
[1] pool:keyed_bag([get(key,X)]).
  Error>> keyed_bag
   Info > tried_to_get_nonexistent_element
   Data > get(key,X)
   Node > 0
 !! Exception (h for help):
```

8.4 複数キー付きのプール (multiply keyed bag)

任意の KL1 データを複数個のキーによって格納し, これらのキー索引をもとに検索を行う仕組みを提供する. このプールを multiply keyed bag[2] と呼ぶ.

8.4.1 概要

Multiply keyed bag の作成: 指定された述語を呼び出すことによってプロセスを起動し, メッセージ通信によって操作する. 要素はすべて { 属性 (0), 属性 (1), ···, 属性 (n-1) } の形式のベクタとする. 属性として任意の KL1 データを設定することができる. 索引となる属性を除き, 未定義状態でも良い. 作成時に形式と, どの属性についてどういう索引を付けるのかを指定する.

機能: 機能として提供するのは, 全要素取り出し等の器全体に対する従来と同内容の操作, 条件を記述することによって行う要素の取り出し等の操作である. 条件は, and, or, not を用いて「どの索引でキーが何々」と記述する. 要素を取り出す / 参照する際には, 要素中の必要な属性を指定し, それだけを得ることができる. 格納されている要素のうちある属性が「最大 / 最小」の要素を取り出すには, 属性を指定してその属性の「最大 / 最小」キーを求めた後, それをもとに要素自身の取り出し / 参照等を行う. 従来のプールと同様, 範囲検索の指定はできない. また, 複数ノードでの実行は考えていない.

索引: 各索引はプロセスである. ハッシュ表と木構造の 2 種類の索引を作成することができる. 索引には要素自身ではなく, 要素に一意に与えられる識別番号が格納される. 要素は主プロセスが管理するデータファイルに識別番号によって格納される. 要素のうち索引指定された属性の MRB は黒となる.

[2] multiply は形容詞 multiple の副詞.

データファイル: データファイルはベクタで実現される．要素の識別番号はすなわちベクタ内での格納位置を表す．従って大きさ n のベクタに格納可能な総要素数は n である．一杯になると自動的に大きさが 2 倍のベクタにすべての要素を格納し直し，索引を作り直す作業を行う．

要素の削除: 要素が削除されるとベクタ内の要素自身は捨てられ，格納されていた場所には代わりに 0 が置かれる．ただし，この空き領域はこのままでは再利用することはできない．他の索引からのちに参照される可能性があるからである．要素の削除時に，索引の修理は行われない．

パック: 削除済み要素のための空き領域を再利用するためには，パックと呼ばれる操作を行う．パックは，データファイル内の空き領域を詰め，索引を作り直す操作である．既定ではデータファイルの大きさの半分を削除済み要素のための空き領域が占めた時，パックが行われる．メッセージによって明にパック操作を行うこともできる．パッキング比率 (自動的に行われるパックのタイミング，すなわち削除済み要素のための空き領域がデータファイルの x 割を占めたらパックする) は，プール作成時に指定することができる．

要素の入れ替え: 削除済み要素のための空き領域をむやみに増加させないために，要素の入れ替え操作は重要である．get と put の 2 操作を用いる場合では空き領域が 1 つできる．また，パック操作では索引はすべて新しく作り直される．get_and_put の操作を用いれば，空き領域を作ることなく最小限の索引修理で要素の更新を行うことができる．

要素の部分集合を対象とする検索: ある条件を満たす要素をすべて取り出し，のちにそれらの要素に対してさらに条件付き探索を行って絞り込むというようなことを記述したいかもしれないが，とりあえず現版ではサポートしない．

8.4.2 使用方法

以下の述語を呼び出してプロセスを生成し，メッセージを送って操作する．

pool:multiply_keyed_bag(DataFormat, S*)

新しい multiply keyed bag を作り，そこへのストリーム S を得る．DataFormat は要素の格納形式を表すベクタである．データファイルの大きさの初期値は 1，パッキング比率は 0.5 である．

pool:multiply_keyed_bag(DataFormat, Init, S*)

新しい multiply keyed bag を作り，そこへのストリーム S を得る．DataFormat は要素の格納形式を表すベクタである．Init にデータファイルのための初期化情報を指定する．

要素の格納形式 DataFormat の指定方法 **DataFormat** は要素の格納形式を表すベクタである．ベクタの各要素が各属性に対応する．各要素は以下のいずれかの形式で記述しなければならない．

{ **属性名 (アトム), 索引指定パラメタ (アトム)**}
{ **属性名 (アトム), 索引指定パラメタ (アトム), 索引初期化情報 (リスト)**}
属性名 (アトム)

属性名 (アトム) はその属性に付けられた名前であり，索引を指定する場合はこの属性名を用いる．すべての属性に必ず指定しなければならない．また，複数個の属性に対して同一名を付けてはいけない．

索引指定パラメタ (アトム) は，ordered, hashed または none のいずれかを指定する．それぞれパラメタに従って次のような索引が用意される．

ordered: 木構造で実現された (二分探索用) 索引

hashed: ハッシュ表で実現された索引

none: 索引を作らない

通常, 索引を作成しない場合は上記のいずれのパラメタも指定する必要はない. その場合, 後にその属性に対して条件付きの検索を行う場合は, 線形検索となる. 線形検索の際の比較ルーチンを特に指定したいユーザは, パラメタ none を用いるとよい.

索引初期化情報 はリストで与える. リストの要素として以下のものを記述することができる. 省略された場合は既定値が用いられる. **Dataformat** で指定された索引の種類と無関係なパラメタが指定された場合, 及び不正なパラメタが指定された場合は, プールの生成に失敗する.

二分探索用索引に設定可能なパラメタ

comparator(Comp): 比較ルーチンとして Comp を指定する. 指定方法は 8.1.3節「比較ルーチンの指定方法」(304頁) を参照のこと.

ハッシュ表索引に設定可能なパラメタ

index_size(Size): ハッシュ表の大きさの初期値を Size(正整数) に設定する.

hasher(Hasher): ハッシュ関数として Hasher(コード) を指定する.

comparator(Comp): 比較ルーチンとして Comp を指定する. 指定方法は 8.1.3節「比較ルーチンの指定方法」(304頁) を参照のこと.

索引を作成しない場合に設定可能なパラメタ

comparator(Comp): 比較ルーチンとして Comp を指定する. 指定方法は 8.1.3節「比較ルーチンの指定方法」(304頁) を参照のこと.

規定値

比較ルーチン: PIMOS ユーティリティが提供する標準の昇順比較ルーチン (8.1.1節「標準の比較ルーチン」303頁参照)

ハッシュ関数: PIMOS ユーティリティが提供する標準のハッシュ関数 (8.2「ハッシング」305頁参照)

ハッシュテーブルの大きさの初期値: データファイルの大きさの初期値

【例】{ 氏名, 年齢, 住所 } という形式の要素を格納する場合の指定例. 氏名欄にはハッシュテーブルによる索引, 年齢欄には二分探索による索引をつける. ハッシュテーブルの初期サイズは 64 とする. また, 住所欄には特に索引を付けない.

```
DataFormat = {name(hashed,IdxInit), age(ordered), address},
IdxInit = [index_size(64)].
```

データファイルの初期化情報 Init の指定方法 データファイルに関する初期化情報はリストかまたは正整数で与える. リストで与える場合, 要素として以下のものを記述することができる. 省略された場合はそれぞれ既定値が用いられる. 不正なパラメタが指定された場合は, プールの生成に失敗する.

size(Size): データファイルの大きさの初期値を Size(正整数) に設定する.

packing_ratio(P): パッキング比率を P(実数, 0<P<1) に設定する.

初期化情報を正整数で与えた場合, その値がデータファイルの大きさの初期値となる.

8.4.2.1 要素の検索

検索のための条件の記述方法

条件は, 以下の形式に従って記述する.

and(条件 1, 条件 2) 条件 1 と条件 2 の両方を満たすものを検索対象とする.
条件 1∧ 条件 2 と記述してもよい.

or(条件 1, 条件 2) 条件 1 か条件 2 のいずれかを満たすものを検索対象とする.
条件 1∨ 条件 2 と記述してもよい.

not(条件) 条件を満たさないものを検索対象とする.
\(条件) と記述してもよい.

equal(属性名, キー) 属性名がキーであるものを検索対象とする.
属性名 = キー と記述してもよい.

not_equal(属性名, キー) 属性名がキーでないものを検索対象とする.
属性名 \ = キー と記述してもよい.

複数条件による要素の検索

複数の属性に対する条件検索は, おのおの全要素を対象に並列に行われ, それぞれの条件を満たす要素の候補が得られた後, 集合演算によって最終的な結果が返される.

【参考】索引のない属性についての条件検索と索引付き属性についての条件検索の and が取られる場合には, 将来, 索引付き属性についての結果得られた候補に対して線形探索が行われるようになる可能性がある[3].

8.4.2.2 要素の部分取り出し / 部分参照の指定

要素の取り出し / 参照を行う場合, 要素全体ではなく部分的に必要な属性のみを得ることができる. これによってメモリ効率は幾分悪くなるが, 特に参照時に, 不必要な属性の MRB を黒くすることを防ぐことができる[4].

要素の部分取り出し / 部分参照を行うには, 以下の形式に従って必要な属性を指定する. 形式は, $m(0 < m)$ 要素のベクタとする. ベクタの各要素には属性名を記述する. 属性名はアトムかその格納位置とし, 指定の順は自由である. 同一属性を含んではいけない.

【注意】同一属性に対する 2 回目以降の取り出しでは, 正常動作は保証されない. また, この点に関するエラーは検出されない.

8.4.2.3 属性の指定方法

通常, 属性を指定するにはプール作成時に指定した属性名 (アトム) を用いるが, この場合, 属性名 (アトム) と「要素中のその属性の位置」との対応を取るための処理を必要とする. そこで, 特に効率を重視するユーザのために, 属性名として直接「要素中のその属性の位置 (正整数)」を指定することも許す. これによって, ユーザインタフェースは少々低下するが, 実行時の計算量を幾分減少させることが可能である.

8.4.2.4 メッセージプロトコル

Multiply keyed bag に送ることのできるメッセージは以下の通りである. ここでは大きく 2 つのグループに分割して示す. 基本的なメッセージについては 318頁「基本メッセージ」を, 要素の部分取り出し / 部分参照に関するメッセージについては 320頁「要素の部分取り出し / 部分参照に関するメッセージ」を参照のこと.

基本メッセージ

[3] 条件検索は, 各条件毎に並列に行う方が効率が良い場合と, 絞り込みながら逐次に検索する方が効率が良い場合がある. これは索引の大きさや条件のきつさにも依存するが, 索引がなく線形探索を行う場合等はたぶん後者の方がよいだろう.

[4] 索引指定されている属性の MRB は作成時から黒.

empty(^YorN)

器が空かどうかを問い合わせる. 器が空の場合, YorN を yes に, そうでない場合, no に具体化する.

empty(C, ^YorN)

器の中に条件 C を満たす要素があるかどうかを問い合わせる. 条件を満たす要素がない場合 YorN を yes に, そうでない場合 no に具体化する.

put(X)

器に要素 X を追加する. 不正な形式の要素を追加しようとした場合, 失敗する.

get(C, ^X)

器から条件 C を満たす要素をひとつ取り出し, X に返す. 条件を満たす要素が複数ある場合, どれが取れてくるかは用いる索引に依存する. 取り出すと器からなくなる. 条件を満たす要素がない場合は失敗する.

get_if_any(C, ^X)

器から条件 C を満たす要素をひとつ取り出し, X に返す. 条件を満たす要素が複数ある場合, どれが取れてくるかは用いる索引に依存する. 取り出すと器からなくなる. 条件を満たす要素がない場合は X に {} を返す.

max_if_any(Attr_name, ^Key)

器に存在する要素のうち, 属性名 Attr_name のキーが最大のものをひとつ選び, そのキーを参照して Key に { キー } の形式で返す. 器が空の場合は Key に {} を返す.

【注意】指定された属性に対して二分探索用索引が用意されていない場合, 全要素に対して線形探索が行われる. 特に比較ルーチンが指定されていない場合は規定のものが用いられる.

min_if_any(Attr_name, ^Key)

器に存在する要素のうち, 属性名 Attr_name のキーが最小のものをひとつ選び, そのキーを参照して Key に { キー } の形式で返す. 器が空の場合は Key に {} を返す.

【注意】指定された属性に対して二分探索用索引が用意されていない場合, 全要素に対して線形探索が行われる. 特に比較ルーチンが指定されていない場合は規定のものが用いられる.

get_all(^A)

器の要素を全部取り出し, A に返す. A の形式は要素のリストである. 器が空の場合は [] を返す.

get_all(C, ^A)

器から条件 C を満たす要素を全部取り出し, A に返す. A の形式は要素のリストである. 条件を満たす要素がない場合は [] を返す. C として, 一般の条件記述の他に属性名 (アトム) も許す. この場合, 指定された属性に用意されている索引順での全要素取り出しを意味する (索引がない場合は指定された属性によってソートされた順に取り出される. ソートに用いられる比較ルーチンは, 特に指定されている場合それが用いられ, 指定されていない場合は PIMOS が提供する標準のものが用いられる).

carbon_copy(^A)

器の要素を全部参照し, A に返す. A の形式は要素のリストである. 器が空の場合は [] を返す.

【注意】このメッセージの実行により参照された要素の MRB は黒となる.

carbon_copy(C, ^A)

器から条件 C を満たす要素を全部参照し, A に返す. A の形式は要素のリストである. 条件を満たす要素がない場合は [] を返す. C として, 一般の条件記述の他に属性名 (アトム) も許す. この場合, 指定された属性に用意されている索引順での全要素参照を意味する (索引がない場合は指定された属性によってソートされた順に取り出される. ソートに用いられる比較ルーチンは, 特に指定されている場合それが用いられ, 指定されていない場合は PIMOS が提供する標準のものが用いられる).

【注意】このメッセージの実行により参照された要素の MRB は黒となる.

get_and_put(C, ^X, X1)

器から条件 C を満たす要素を取り出して X に返し, 同じ位置に新しい要素 X1 を格納する. 条件を満たす要素がない場合は失敗する. このメッセージによって要素を置き換えるに当たり, 必要な場合は索引の修理を行う.

get_if_any_and_put(C, ^X, X1)

器に条件 C を満たす要素があれば取り出し, X に返す. また, 同じ位置に新しい要素 X1 を格納する. 条件を満たす要素がない場合は X に {} を返し, X1 を新たに格納する. このメッセージによって要素を置き換えるに当たり, 必要な場合は索引の修理を行う.

get_and_put_if_any(C, ^X, X1, ^X2)

器に条件 C を満たす要素があれば取り出し, X に返す. また, 同じ位置に新しい要素 X1 を格納し, X2 に {} を返す. 条件を満たす要素がない場合は X に {} を返す. また, X2 に X1 を返し, 器に対しては何も行わない. このメッセージによって要素を置き換えるに当たり, 必要な場合は索引の修理を行う.

do(Stream*)

Stream にメッセージを要素とするリストを渡すと, Stream 内のメッセージは連続して処理される.

【参考】ひとつのプールに対して複数のプロセスがアクセスする場合, そのうちのひとつのプロセスからの一連の操作を連続して処理したい場合に有効である.

pack

要素の削除後の空き領域を詰め, 索引を作り直す.

要素の部分取り出し / 部分参照に関するメッセージ

getp(C, P, ^X)

器から条件 C を満たす要素をひとつ取り出し, その要素中の P に指定された属性の内容を X に返す. 条件を満たす要素が複数ある場合, どれが取れてくるかは用いる索引に依存する. 取り出すと器からなくなる. 条件を満たす要素がない場合は失敗する.

getp_if_any(C, P, ^X)

器から条件 C を満たす要素をひとつ取り出し, その要素中の P に指定された属性の内容を X に返す. 条件を満たす要素が複数ある場合, どれが取れてくるかは用いる索引に依存する. 取り出すと器からなくなる. 条件を満たす要素がない場合は X に {} を返す.

getp_all(P, ^A)

器の要素を全部取り出し, それぞれの要素中の P に指定された属性の内容を A に返す. A の形式はそれらのリストである. 器が空の場合は [] を返す.

getp_all(C, P, ^A)

器から条件 C を満たす要素を全部取り出し, それぞれの要素中の P に指定された属性の内容を A に返す. A の形式はそれらのリストである. 条件を満たす要素がない場合は [] を返す. C として, 一般の条件記述の他に属性名 (アトム) も許す. この場合, 指定された属性に用意されている索引順での全要素取り出しを意味する (索引がない場合は指定された属性によってソートされた順に取り出される. ソートに用いられる比較ルーチンは, 特に指定されている場合それが用いられ, 指定されていない場合は PIMOS が提供する標準のものが用いられる).

carbon_copyp(P, ^A)

器の要素を全部参照し, それぞれの要素中の P に指定された属性の内容を A に返す. A の形式はそれらのリストである. 器が空の場合は [] を返す.

【注意】このメッセージの実行により参照された属性の MRB は黒となる.

carbon_copyp(C, P, ^A)

器から条件 C を満たす要素を全部参照し, P に指定された属性の内容を A に返す. A の形式はそれらのリストである. 条件を満たす要素がない場合は [] を返す. C として, 一般の条件記述の他に属性名 (アトム) も許す. この場合, 指定された属性に用意されている索引順での全要素参照を意味する (索引がない場合は指定された属性によってソートされた順に取り出される. ソートに用いられる比較ルーチンは, 特に指定されている場合それが用いられ, 指定されていない場合は PIMOS が提供する標準のものが用いられる).

【注意】このメッセージの実行により参照された属性の MRB は黒となる.

getp_and_put(C, P, ^X, X1)

器から条件 C を満たす要素ひとつを選び, その要素中の P に指定された属性の内容を X に返す. さらに, それらの属性を X1 に与えられた内容で置き換える. 条件を満たす要素がない, 或いは X1 に不正な形式の値が与えられた場合は失敗する. このメッセージによって要素の属性を置き換えるに当たり, 必要な場合は索引の修理を行う.

getp_and_put_if_any(C, P, ^X, X1, ^X2)

器に条件 C を満たす要素があればひとつ選び, その要素中の P に指定された属性の内容を X に返す. さらに, それらの属性を X1 に与えられた内容で置き換え, X2 に {} を返す. 条件を満たす要素がない場合は X に {} を返す. また, X2 に X1 を返し, 器に対しては何も行わない.

このメッセージによって要素の属性を置き換えるに当たり, 必要な場合は索引の修理を行う.

8.4.3 例

【例 1】魚に関する multiply keyed bag の作成

格納要素の形式は { 英名, 和名, 体長, 目, 腹びれ, 胸びれ, 脂ひれ, 背びれ, 海水 / 淡水 } とし, 英名及び和名にはハッシュテーブル索引, 体長には二分探索索引を付ける.

```
initialize(In) :- true |
        pool:multiply_keyed_bag(Fmt,128,In),
        Fmt={name(hashed),jname(hashed),length(ordered),
                eye,ventral_fin,pectoral_fin,abura_fin,dorsal_fin,habitat}.
```

【例 2】要素の格納[5]

```
put_data(In,In1) :- true |
        In=[put({ell,         unagi,      60, 0,0,1,0,1,f}),
            put({cutlass_fish,tachiuo,  150, 0,0,1,0,1,s}),
            put({rubripes,    torafugu,   70, 0,0,1,0,1,s}),
            put({filefish,    kawahagi,   30, 0,0,1,0,1,s}),
                        ......
            put({sardine,     maiwashi,   36, 0,1,0,0,1,s}),
            put({herring,     nishin,     30, 0,1,0,0,1,s}),
            put({salmon,      sake,       70, 0,1,0,1,1,s}),
            put({rainbow_trout,nijimasu, 40, 0,1,0,1,1,f}),
            put({hemiramph,   sayori,     40, 0,1,1,0,1,s}),
            put({rabbitfish,  aigo,       40, 0,{2,2,3},0,0,1,s}),
            put({red_sea_bream,madai,     60, 0,{2,1,5},0,0,1,s}),
            put({sea_bass,    suzuki,     90, 0,{2,1,5},0,0,2,s}),
            put({oceanic_bonito,katsuo, 100, 0,{2,1,5},1,0,2,s}),
                        ......
            put({flatfish,    hirame,     70, 1,3,1,0,1,s})|In1].
```

【例 3】barracuda のデータを取り出す

```
        In=[get(name=barracuda,X)|In1].
```

結果として X=barracuda(akakamasu,50,0,1,1,0,2,s) が返される.

以下のような記述も可.

```
        In=[get(0=barracuda,X)|In1].
```

【例 4】目が頭の片側に並ぶ海水魚をひとつ取り出し, 英名, 和名, 体長のみを得る

```
        In=[getp(C,P,X)|In1],
        C=(eye=1)/\(habitat=s),
        P={name,jname,length}.
```

[5] 格納内容は以下を意味する.

目 **0:** 頭の左右両側にある, **1:** 頭の片側に並ぶ.

腹びれ **0:** なし, **1:** 胸びれより後方にある, {**2**,**M**,**N**}**:** 腹びれは胸びれとほぼ同じ位置の下部にあり, M 本の棘と N 本の軟条を持つ. **3:** 胸びれより前方にある, **4:** 吸盤に変形している.

胸びれ **0:** 体側下部, **1:** 体側中, 上部に位置する.

脂ひれ **0:** なし, **1:** あり.

背びれ 背びれの数.

海水 / 淡水 **s:** 海水, **f:** 淡水.

結果として X=right_eyed_flounder(magarei,55) が返される.

【例 5】目が頭の左右両側, 腹びれが胸びれより後方にあり脂ひれを持たない魚の英名と体長を参照

```
In=[carbon_copyp(C,P,A)|In1],
C=(eye=0)/\(ventral_fin=1)/\(abura_fin=0),
P={name,length}.
```

結果として A=[sardine(36),herring(30),carp(60),loach(18),hemiramph(40), ...] が返される.

【例 6】背びれの数が複数の魚を取り出し, 英名と背びれの数を得る

```
In=[getp_all(C,P,A)|In1],
C=(dorsal_fin\=0)/\(dorsal_fin\=1),
P={name,dorsal_fin}.
```

結果として A = [barracuda(2), gray_mullet(2), sea_bass(2), horse_mackerel(2), yellowtail(2), codfish(3), goby(2), ...] が返される.
条件は (背びれの数が 2 または 3 とわかっているならば) 以下のように記述してもよい.

```
C=(dorsal_fin=2)\/(dorsal_fin=3)
```

【例 7】腹びれは胸びれとほぼ同じ位置の下部にあって 1 本の棘があり, 胸びれが体側下部にある魚を取り出し, 英名と腹びれの情報及び背びれの数を得る

```
In=[getp_all(C,P,A)|In1],
C=((ventral_fin={2,1,10})\/(ventral_fin={2,1,5}))/\(pectoral_fin=0),
P={name,ventral_fin,dorsal_fin}.
```

結果として A=[alfonsino({2,1,10},1),red_sea_bream({2,1,5},1),sea_bass({2,1,5},2), horse_mackerel({2,1,5},2),yellowtail({2,1,5},2), ...] が返される.

【例 8】最も体長が長い魚を取り出す

```
In=[max_if_any(length,{Key}),get(length=Key,X)|In1].
```

結果として X=bluefin_tuna(maguro,200,0,2,1,5,1,0,2,s) が返される.

【例 9】すべてのデータを体長順に取り出す

```
In=[get_all(length,A)|In1].
```

結果として A=[ayu(ayu,15,0,1,0,1,1,f),loach(dojo,18,0,1,0,0,1,f), ...] が返される.

【例 10】いわなのデータの間違いを修正する (いわなは淡水魚)

```
In=[getp_and_put(jname=iwana,{habitat},OldAttr,NewAttr),
        carbon_copy(jname=iwana,A)|In1],
NewAttr={f}.
```

結果として OldAttr={s}, A=[char(iwana,25,0,1,0,1,1,f)] が返される.

8.4.4 Multiply keyed bag が表示するエラーメッセージ

Multiply keyed bag は, 定義されていないメッセージを受信した場合や存在しない要素を取り出そうとした場合などにエラー報告を行う. 以下に multiply keyed bag が表示するエラーメッセージと幾つかの例を示す. なお, シェル上でエラーが報告される場合の表示形式について, 詳しくは操作編 2.1.5.2節 シェ

ルの「エラーメッセージ」(25, 26頁) を, リスナ上でエラーが報告される場合の表示形式について, 詳しくは操作編 4.1.8.3節 リスナの「エラーメッセージ」【例 39】(156頁) を参照.

【参考】これらのエラーメッセージのうち一部を除く殆どはエラー検出時に保持している情報を出力するというもので, エラーメッセージ出力のために不必要な情報を蓄えるということは行っていない. これは効率の低下を防ぐためであるが, これによってプログラムのデバッグに幾分支障が出る恐れもあり, 将来, エラーの詳細内容まで報告する版を提供する可能性もある.

illegal_initiation_parameter	: 初期化パラメタが不正	(例 1-2)
illegal_data_format	: 格納形式が不正	(例 3)
illegal_attribute_name	: 属性名が不正	(例 4,11)
doubly_defined_attribute_name	: 既に定義済みの属性名を指定しようとした	(例 5)
illegal_index_type	: 索引指定パラメタが不正	(例 6)
illegal_initiation_parameter_for_ordered_index	: 二分探索用索引初期化パラメタが不正	(例 7)
illegal_initiation_parameter_for_hashed_index	: ハッシュテーブル索引初期化パラメタが不正	(例 8-9)
illegal_condition	: 条件記述が不正	(例 10)
illegal_message_protocol	: メッセージプロトコルが不正	(例 12)
tried_to_get_nonexistent_element	: 存在しない要素を取り出そうとした	(例 13)
illegal_format_for_partial_reference	: 部分参照形式が不正	(例 14)
illegal_data_format in Message	: メッセージ Message の格納要素の形式が不正	(例 15-16)

【例 1】初期化パラメタが不正

```
[1] pool:multiply_keyed_bag({a,b,c},-1,[]).
  Error>> multiply_keyed_bag
   Info > illegal_initiation_parameter
   Data > -1
   Node > 0
 !! Exception (h for help):
 no.
```

【例 2】初期化パラメタが不正

```
[2] pool:multiply_keyed_bag({a,b},[packing_ratio(0.0)],[]).
  Error>> multiply_keyed_bag
   Info > illegal_initiation_parameter
   Data > packing_ratio( +0.0 )
   Node > 0
 !! Exception (h for help):
 no.
```

【例 3】格納形式が不正

```
[3] pool:multiply_keyed_bag(a,[]).
  Error>> multiply_keyed_bag
   Info > illegal_data_format
   Data > a
   Node > 0
 !! Exception (h for help):
 no.
```

【例 4】属性名が不正

```
[4] pool:multiply_keyed_bag({1},[]).
  Error>> multiply_keyed_bag
   Info > illegal_attribute_name
   Data > 1
   Node > 0
 !! Exception (h for help):
 no.
```

【例 5】既に定義済みの属性名を指定しようとした

```
[5] pool:multiply_keyed_bag({a,a},[]).
  Error>> multiply_keyed_bag
   Info > doubly_defined_attribute_name
   Data > a
   Node > 0
 !! Exception (h for help):
 no.
```

【例 6】索引指定パラメタが不正

```
[6] pool:multiply_keyed_bag({a(wan)},[]).
  Error>> multiply_keyed_bag
   Info > illegal_index_type
   Data > wan
   Node > 0
 !! Exception (h for help):
 no.
```

【例 7】二分探索用索引初期化パラメタが不正

```
[7] pool:multiply_keyed_bag({a(ordered,{a})},[]).
  Error>> multiply_keyed_bag
   Info > illegal_initiation_parameter_for_ordered_index
   Data > {a}
   Node > 0
 !! Exception (h for help):
 no.
```

【例 8】ハッシュテーブル索引初期化パラメタが不正

```
[8] pool:multiply_keyed_bag({a(hashed,[a])},[]).
  Error>> multiply_keyed_bag
   Info > illegal_initiation_parameter_for_hashed_index
   Data > a
   Node > 0
 !! Exception (h for help):
 no.
```

【例 9】ハッシュテーブル索引初期化パラメタが不正

```
[9] pool:multiply_keyed_bag({a(hashed,[hasher(c)])},[]).
  Error>> multiply_keyed_bag
   Info > illegal_initiation_parameter_for_hashed_index
   Data > hasher(c)
   Node > 0
 !! Exception (h for help):
 no.
```

【例 10】条件記述が不正

```
[10] pool:multiply_keyed_bag({a},[empty(a,YorN)]).
  Error>> multiply_keyed_bag
   Info > illegal_condition
   Data > a
   Node > 0
 !! Exception (h for help):
 no.
```

【例 11】属性名が不正

```
[11] pool:multiply_keyed_bag({a},[empty((a=1)/\(b=1),YorN)]).
  Error>> multiply_keyed_bag
   Info > illegal_attribute_name
   Data > b
   Node > 0
 !! Exception (h for help):
 no.
```

【例 12】メッセージプロトコルが不正

```
[12] pool:multiply_keyed_bag({a},[unknown(_,_)]).
  Error>> multiply_keyed_bag
   Info > illegal_message_protocol
   Data > unknown(A,B)
   Node > 0
 !! Exception (h for help):
 no.
```

【例 13】存在しない要素を取り出そうとした

```
[13] pool:multiply_keyed_bag({a},[get(a=1,X)]).
  Error>> multiply_keyed_bag
   Info > tried_to_get_nonexistent_element
   Data > get(a=1,X)
   Node > 0
 !! Exception (h for help):
 no.
```

【例 14】部分参照形式が不正

```
[14] pool:multiply_keyed_bag({a,b},[getp(a=1,[b],X)]).
  Error>> multiply_keyed_bag
   Info > illegal_format_for_partial_reference
   Data > [b]
   Node > 0
 !! Exception (h for help):
 no.
```

【例 15】メッセージ put において格納要素の形式が不正

```
[15] pool:multiply_keyed_bag({a,b,c},[put({1})]).
  Error>> multiply_keyed_bag
   Info > illegal_data_format in put
   Data > {1}
   Node > 0
 !! Exception (h for help):
 no.
```

【例 16】メッセージ getp_and_put において格納要素の形式が不正

```
[16] pool:multiply_keyed_bag({a,b,c},[put({1,1,1}),getp_and_put(a=1,{b},X,{2,2})]).
  Error>> multiply_keyed_bag
   Info > illegal_data_format in getp_and_put
   Data > {2,2}
   Node > 0
 !! Exception (h for help):
 no.
```

8.5 プールへのメッセージ一覧

めっせーじいちらん

表 8.1: プールへのメッセージ一覧

(1)bag, (2)stack, (3)queue, (4)sorted_bag, (5)keyed_bag, (6)keyed_set,
(7)keyed_sorted_bag, (8)keyed_sorted_set, (9)multiply_keyed_bag

メッセージ名	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
empty(^YorN)	○	○	○	○	○	○	○	○	○
empty(K,^YorN)	×	×	×	×	○	○	○	○	○
put(X)	○	○	○	○	×	×	×	×	○
put(K,^X)	×	×	×	×	○	×	○	×	×
put(K,X,^OldX)	×	×	×	×	×	○	×	○	×
get(^X)	○	○	○	○	×	×	×	×	×
get(K,^X)	×	×	×	×	○	○	○	○	○
getp(C,P,^X)	×	×	×	×	×	×	×	×	○
get_if_any(^X)	○	○	○	○	×	×	×	×	×
get_if_any(K,^X)	×	×	×	×	○	○	○	○	○
getp_if_any(C,P,^X)	×	×	×	×	×	×	×	×	○
get_all(^O)	○	○	○	○	○	○	○	○	○
get_all(K,^O)	×	×	×	×	○	○	○	○	○
getp_all(P,^O)	×	×	×	×	×	×	×	×	○
getp_all(C,P,^O)	×	×	×	×	×	×	×	×	○
carbon_copy(^O)	○	○	○	○	○	○	○	○	○
carbon_copy(K,^O)	×	×	×	×	○	○	○	○	○
carbon_copyp(P,^O)	×	×	×	×	×	×	×	×	○
carbon_copyp(C,P,^O)	×	×	×	×	×	×	×	×	○
get_and_put(K,X,Y)	×	×	×	×	○	○	○	○	○
getp_and_put(C,P,^X,Y)	×	×	×	×	×	×	×	×	○
get_if_any_and_put(K,^X,Y)	×	×	×	×	○	○	○	○	○
get_and_put_if_any(K,^X,Y,^Y1)	×	×	×	×	○	○	○	○	○
getp_and_put_if_any(C,P,^X,Y,^Z)	×	×	×	×	×	×	×	×	○
get_max_if_any(^X)	×	×	×	○	×	×	○	○	×
get_min_if_any(^X)	×	×	×	○	×	×	○	○	×
max_if_any(A,^K)	×	×	×	×	×	×	×	×	○
min_if_any(A,^K)	×	×	×	×	×	×	×	×	○
get_load_factor(^A)	×	×	×	×	○	○	×	×	×
get_average_chaining_length(^L)	×	×	×	×	○	○	×	×	×
pack	×	×	×	×	×	×	×	×	○
do(Stream*)	○	○	○	○	○	○	○	○	○

【参考】シェルやリスナから

usage(プール名アトム)

を実行すると, 各プールへのメッセージ一覧を表示する. なお, プール名を **pool** と指定するとプール名の一覧を表示することができる.

【例】リスナで stack のメッセージ一覧を表示する

```
[1] usage(stack).
Usage :
  pool:stack(S*)
Messages :
  empty(^YorN)
  put(Data)
  get(^Data)
  get_if_any(^X)  , X={Data};{}
  get_all(^O)     , O=[Data, .. ]
  carbon_copy(^O) , O=[Data, .. ]
  do(S*)

[2]
```

8.6 ソーティング用フィルタ

任意の KL1 データ集合をソートする機能を提供する.

ソータは, 与えられた比較機構に従ってデータ集合を順番に並べ変えるものである. PIMOS ではフィルタ形式で提供する. すなわち, 入力ストリームとして KL1 データ列を与えることにより, 出力ストリームからソートされたデータ列を得る. 入力ストリームを閉じると, 初めて出力ストリームに出力を開始する. 比較が stable なら, ソーティングも stable になる.

【参考】ソータの実現にはマージソートを用いた.

【注意】データプロトコル中の変数に付加されている記号 ^, ? 等は, 7.1節のモード表示方法に従うものである. ソーティング / マージング用フィルタでは一般に比較に用いるものを除き, データは未定義状態でもかまわないので, そのようなデータの入力及び結果の出力では, モード表示は ? となる. 詳しくは, プログラミング編 7.1節「モード表示の読み方」(222頁) 参照.

8.6.1 キーなしのソーティング用フィルタ

通常のソータである. 以下の述語を呼び出してフィルタを生成する.

sorter:sort(In*, ^Out)

キーなしのソーティング用フィルタを生成する. 要素の比較には標準の昇順比較ルーチン comparator:sort/5(8.1.1節「標準の比較ルーチン」 303頁) を用いる.

sorter:sort(In*, Comparator, ^Out)

キーなしのソーティング用フィルタを生成する. 要素の比較には Comparator に指定された比較ルーチンを用いる. Comparator の指定方法は 8.1.3節 「比較ルーチンの指定方法」(304頁) を参照のこと.

8.6.2 キー付きのソーティング用フィルタ

入力ストリームとして {Key, ?Data} の形式で表された KL1 データの列を与えることにより, キー部だけに着目してソートした結果を出力ストリームに返す. 出力ストリームは, やはり {Key, ?Data} の形式のデータの並びである. 以下の述語を呼び出してフィルタを生成する.

sorter:key_sort(In*, ^Out)

キー付きのソーティング用フィルタを生成する. 要素の比較には標準の昇順比較ルーチン comparator:sort/5 (8.1.1節「標準の比較ルーチン」 303頁) を用いる.

sorter:key_sort(In*, Comparator, ^Out)

キー付きのソーティング用フィルタを生成する． 要素の比較には Comparator に指定された比較ルーチンを用いる．Comparator の指定方法は 8.1.3節 「比較ルーチンの指定方法」(304頁) を参照のこと．

8.7 マージング用フィルタ

任意の KL1 データ集合のマージ機能を提供する．

マージャは，2 本のソート済みのデータ列から 1 本のソートされたデータ列を生成するものである．PIMOS ではフィルタ形式で提供する．すなわち，ソート済みの 2 本の KL1 データ列をそれぞれ 2 本の入力ストリームとして与えることにより，出力ストリームとしてソートされた 1 本のデータ列を得る．比較が stable なら，ソーティングも stable になる．

【注意】ここに述べるマージャは，組込述語の (内容の如何に関わらず複数のストリームをマージする) マージャ (プログラミング編 7.4.19節 組込述語「ストリームサポート」(285頁) を参照) とは別のものである

8.7.1 キーなしのマージング用フィルタ

通常のマージャである．以下の述語を呼び出してフィルタを生成する．

sorter:merge(In1, In2, ^Out)

キーなしのマージング用フィルタを生成する．要素の比較には標準の昇順比較ルーチン comparator:sort/5(8.1.1節「標準の比較ルーチン」 303頁) を用いる．

sorter:merge(In1, In2, Comparator, ^Out)

キーなしのマージング用フィルタを生成する． 要素の比較には Comparator に指定された比較ルーチンを用いる．Comparator の指定方法は 8.1.3節 「比較ルーチンの指定方法」(304頁) を参照のこと．

8.7.2 キー付きのマージング用フィルタ

キー部だけに着目してマージした結果を出力ストリームに返す．入出力ストリームは，{Key, ?Data} の形式のデータの並びである．以下の述語を呼び出してフィルタを生成する．

sorter:key_merge(In1, In2, ^Out)

キー付きのマージング用フィルタを生成する．要素の比較には標準の昇順比較ルーチン comparator:sort/5 (8.1.1節「標準の比較ルーチン」 303頁) を用いる．

sorter:key_merge(In1, In2, Comparator, ^Out)

キー付きのマージング用フィルタを生成する．要素の比較には Comparator に 指定された比較ルーチンを用いる．Comparator の指定方法は 8.1.3節 「比較ルーチンの指定方法」(304頁) を参照のこと．

8.8 ソーティング / マージング用フィルタが表示するエラーメッセージ

これらのフィルタでは, 入力データの形式が不正の場合などにエラー報告を行う. 報告の内容及び形式を以下に示す. なお, シェル上でエラーが報告される場合の表示形式について, 詳しくは操作編 2.1.5.2節 シェルの「エラーメッセージ」(25, 26頁) を, リスナ上でエラーが報告される場合の表示形式について, 詳しくは操作編 4.1.8.3節 リスナの「エラーメッセージ」【例 39】(156頁) を参照.

キー付きのソーティング用フィルタ, マージング用フィルタへの入力ストリームに不正な形式のデータを与えた場合

エラー ID: 使用した述語名 (key_sort または key_merge のいずれか)

エラー情報: illegal_data_protocol

エラーデータ: エラーを生じたデータ

【例 1】リスナ上で実行中にエラーが検出された場合の報告例

```
[1] sorter:key_sort([a],Out).
 Error>> key_sort
  Info > illegal_data_protocol
  Data > a
  Node > 0
!! Exception (h for help):
```

【注意】このエラー報告は比較を行おうとして検出されたものに関してのみ報告されるものであり, データ列全体に渡って必ずしも検出されるとは限らない. 例えば以下の例のように, もはや比較を行う必要のないデータ列に対しては検出されることはない. これはユーティリティが正しく使用された場合の効率を落とさないようにするためである.

【例 2】

```
[2] sorter:key_merge([],[key],Out).
 Out = [key]
 yes.

[3]
```

キー付きのマージング用フィルタへの入力ストリーム用引数が不正 (リスト以外のデータを与えた) の場合

エラー ID: key_merge

エラー情報: illegal_protocol

エラーデータ: エラーを生じたデータ

【例 1】

```
[1] sorter:key_merge(a,[{key,data}],Out).
 Error>> key_merge
  Info > illegal_protocol
  Data > a
  Node > 0
!! Exception (h for help):
```

【注意】このエラー報告は比較を行おうとして検出されたものに関してのみ報告されるものであり, データ列全体に渡って必ずしも検出されるとは限らない. 例えば以下の例のように, もはや

比較を行う必要のない場合には検出されることはない. これはユーティリティが正しく使用された場合の効率を落とさないようにするためである.

【例 2】

```
[2] sorter:key_merge(a,[],Out).
 Out = a
 yes.

[3]
```

【注意】Key_merge を除くユーティリティへの入力ストリームには組込述語のマージャが挿入されているため, 引数が不正だった場合, ユニフィケーションの失敗となる.

第 9 章

入出力管理

本章では, 入出力のしかたについて述べる.

PIMOS の入出力資源には何があるか

PIMOS は, ウインドウ, ファイル, タイマ, ストリング入出力デバイスという 4 種類の入出力資源を提供している. それぞれの特徴は以下の通りである.

ウインドウ: 文字の入出力をすることができる. 入力を途中でアボートすることができる. ウインドウの大きさ, 位置, フォントやタイトルなどを変更することも可能である.

ファイル: 文字の入出力をすることができる. ディレクトリ操作も用意している.

タイマ: 現在の日付と時刻を知ることができる. 目覚まし時計やストップウォッチも持っている. またそれらを途中でアボートすることも可能である.

ストリング入出力デバイス: ユーザが FEP 上に ESP を用いて定義したデバイスを PIMOS 上で操作するためのデバイスである. 詳しくは 9.3.8節「ストリング入出力デバイス」 (371頁) 参照.

PIMOS の入出力資源を使うには

PIMOS が提供する入出力資源は, すべてプロセスでできている. 入出力資源を利用するには, まず, それぞれの入出力資源へのストリームを決められた方法で獲得し, それらのストリームに対してメッセージを送る, という形式になっている. ユーザは, これらの手順を正しく踏まなければならない.

これらの基本的な一連の操作のやりかたについて, 9.1節 「入出力資源の獲得」 (333頁) でわかりやすく説明する. なお, 9.1節は, 入出力資源を獲得するための全体の流れをわかりやすく説明することに重点をおいている. 一連の操作の中で行われる個々の操作については, マニュアル中のそれぞれ対応する箇所に詳細な仕様が記述してあるので, そこを個別に参照するのがよい.

入出力のためのユーティリティ

入出力資源自体は, いわば何も飾りのついていない生のデバイスである. 有限長の 16 ビット長文字列の入出力を始めとするデバイス依存の処理のみを行うことができる. 入力文字列を適当な文法に従って構文解析したり, 任意の KL1 データを文字列として出力する機能はない.

そこで PIMOS では, 入出力を容易に効率よく実現するためのユーティリティを, フィルタの形式で提供する. フィルタには, それぞれの用途に合わせて数種類がある. これらのフィルタは, 内部に演算子表を持っており, それに基づいてエジンバラ Prolog と同様の演算子順位文法によるデータの文字列表現の構文解析, 及び KL1 データから文字列表現の生成を行う.

また, 入出力によって生じる通信オーバヘッドを軽減するための, ブロッキング / デブロッキング機能も持つ. すなわち, 入力が必要になった時にある程度の文字数をまとめて入力して内部に溜めておき,

要求に応じて少しずつ返す, 或いは, ある程度の文字数のデータが溜まるまで実際には出力せずに内部に溜めておき, 溜まったらまとめて出力する, というようなことである.

ユーザは, 獲得した生のデバイスへの出力ストリームの手前にフィルタを挿入するだけで, これらの機能を利用することができる. このような入出力に関するユーティリティの使用方法について 9.2節「入出力ユーティリティ」 (338頁) で説明する.

入出力デバイスの制御

デバイスには, 通常の入出力メッセージを処理する他に, それらのメッセージの処理中にも PIMOS からの要求, 或いはユーザからの割り込みによりそれらを中断する機構が必要である. PIMOS が提供するすべてのデバイスは, 通常の入出力処理を行うための通信路であるストリームの他に, 2 本の制御ラインを持っている. ひとつは PIMOS からの中断要求を受け付けるためのアボートライン, もうひとつはユーザからの割り込みを PIMOS に通知するためのアテンションラインである. このようなデバイスの制御方法, 及び生のデバイスを操作するためのメッセージプロトコルなどについて 9.3節「入出力制御」 (354頁) で説明する.

標準の入出力デバイス

プログラムを記述する時, 入出力デバイスの種類にかかわらずある決まったメッセージプロトコルで操作できるような入出力デバイスがあると便利である. PIMOS にはタスク単位で, 標準入出力資源と呼ばれる抽象的な入出力デバイスが用意されている. 通常の入出力デバイスと同様に, まず標準入出力資源へのストリームを獲得し, そのストリームに対して決まったメッセージを送ることによって入出力を行う. 標準入出力資源について 9.4節 「標準入出力」 (375頁), その使用方法については 9.4.2節 「標準入出力を利用して入出力を行う方法」 (376頁) で説明する.

標準入出力資源として具体的にどのような入出力デバイスが割り当てられるかは, 各タスクの起動時に設定される. 例えば, シェルやリスナからプログラムを実行する場合には, シェルやリスナが, ユーザの入力するコマンドにしたがって, それぞれの場合に応じて設定する. 詳しくは 操作編 2.1節 シェルの基本操作「ユーザゴールの起動」 (15頁), 或いは 4.1.4節 リスナの「標準入出力」 (78頁) 参照.

一方, ユーザプログラム自身がタスクに対して標準入出力資源を割り当てて使用する場合の, 割り当て方やその制御方法については, 9.4.3節 「標準入出力をタスクに割り当てそれを管理するには」 (379頁) で説明する.

9.1 入出力資源の獲得

PIMOS が提供する入出力デバイスを利用するには, まず入出力デバイス自身を獲得しなければならない. 本節では, 入出力デバイスを獲得するために必要な, 基本的な一連の操作のやりかたを説明する.

なお, 本節は, 入出力資源を獲得するための全体の流れをわかりやすく説明することに重点をおいている. 一連の操作の中で行われる個々の操作については, マニュアル中のそれぞれ対応する箇所に詳細な仕様が記述してあるので, そこを個別に参照するのがよい.

9.1.1 まず, ジェネラルリクエストデバイスの獲得

ジェネラルリクエストデバイスとは, PIMOS がユーザに提供する様々なサービスを行うデバイスである. ユーザが PIMOS の入出力デバイスを利用する際にも, まずジェネラルリクエストデバイスを獲得し, このデバイスに要求を出さなければならない. ジェネラルリクエストデバイスを獲得するには, 以下にあげる荘園の例外生起用述語 raise を用いる.

```
shoen:raise(pimos_tag#task, general_request, GeneralRequestDevice)
```

この述語を実行すると，実行したタスクが持つジェネラルリクエストデバイスへのストリームを GeneralRequestDevice に得ることができる．

【注意】この述語呼び出しを記述する module の先頭にて次の宣言を行なう必要がある．

```
:- with_macro pimos.
```

【参考】ジェネラルリクエストデバイスについて, 詳しくは 11.1.9節 「ジェネラルリクエストデバイス」(403頁) を, 荘園の例外生起用述語 raise について, 詳しくは 7.5.5節 「例外の生起」(300頁) 参照.

9.1.2 次に，リクエストデバイスの獲得

PIMOS のデバイスは, その役割機能に応じて階層構造を成している．デバイスを利用するには, 上位階層を占めるデバイスから順に獲得していかなければならない．デバイスの階層構造について, 詳しくは 9.3.1節 「デバイスの階層」(354頁) 参照.

前節の手順で獲得したジェネラルリクエストデバイスに対して, 以下のメッセージを送ると, それぞれ, ウインドウを生成するためのデバイス, ファイルをオープンするためのデバイス, タイマを生成するためのデバイス, ストリング入出力デバイスを生成するためのデバイスを獲得することができる．これらのデバイスをそれぞれ, **ウインドウリクエストデバイス**, **ファイルリクエストデバイス**, **タイマリクエストデバイス**, **ストリング入出力リクエストデバイス** と呼ぶ.

なお, メッセージの説明は, 7.1節のモード表示にしたがって記述してある．詳しくは, プログラミング編 7.1節 「モード表示の読み方」(222頁) を参照.

window(^Result)

ジェネラルリクエストデバイスへのストリームを獲得したタスクに設定されている環境変数 **task: io_node** の値に指定されている FEP 上にウインドウリクエストデバイスを生成して, アボートライン, アテンションラインを張る.

生成に成功した場合, Result に **normal(^Request, ^Abort, Attention)** を返す．Request は生成されたウインドウリクエストデバイスへのストリーム, Abort, Attention はそれぞれウインドウリクエストデバイスのアボートライン, アテンションラインである．生成に失敗した場合, Result に **abnormal** を返す.

file(^Result)

ジェネラルリクエストデバイスへのストリームを獲得したタスクに設定されている環境変数 **task: io_node** の値に指定されている FEP 上にファイルリクエストデバイスを生成して, アボートライン, アテンションラインを張る.

生成に成功した場合, Result に **normal(^Request, ^Abort, Attention)** を返す．Request は生成されたファイルリクエストデバイスへのストリーム, Abort, Attention はそれぞれファイルリクエストデバイスのアボートライン, アテンションラインである．生成に失敗した場合, Result に **abnormal** を返す.

timer(^Result)

ジェネラルリクエストデバイスへのストリームを獲得したタスクに設定されている環境変数 **task: io_node** の値に指定されている FEP 上にタイマリクエストデバイスを生成して, アボートライン, アテンションラインを張る.

生成に成功した場合, Result に **normal(^Request, ^Abort, Attention)** を返す. Request は生成されたタイマリクエストデバイスへのストリーム, Abort, Attention はそれぞれタイマリクエストデバイスのアボートライン, アテンションラインである．生成に失敗した場合, Result に **abnormal** を返す.

string_io(^Result)

ジェネラルリクエストデバイスへのストリームを獲得したタスクに設定されている環境変数 **task: io_node** の値に指定されている FEP 上にストリング入出力リクエストデバイスを生成して, アボートライン, アテンションラインを張る.

生成に成功した場合, Result に **normal(^Request, ^Abort, Attention)** を返す. Request は生成されたストリング入出力リクエストデバイスへのストリーム, Abort, Attention はそれぞれストリング入出力リクエストデバイスのアボートライン, アテンションラインである. 生成に失敗した場合, Result に **abnormal** を返す.

【参考 1】アボートライン, アテンションラインについては プログラミング編 9.3.3節 「デバイスのアボート / アテンション処理」(356頁) を参照.

【参考 2】既定以外の FEP や FEP を介してネットワークで接続されたリモートサイトにおけるマシン上の入出力資源を利用することもできる. このような資源を獲得するためのジェネラルリクエストデバイスについての詳細は, プログラミング編 11.1.9節「ジェネラルリクエストデバイス」(403頁) 参照.

9.1.3 そして, デバイスの獲得

以上のようにして獲得したそれぞれのリクエストデバイスに対してメッセージを送ると, ウインドウやファイルなどの生のデバイスを獲得することができる. 以下に幾つかのプログラム例をあげる.

【参考】各リクエストデバイスに送ることのできるメッセージはそれぞれ,
9.3.5.1節「ウインドウリクエストデバイス」(359頁),
9.3.6.1節「ファイルリクエストデバイス」(363頁),
9.3.7.1節「タイマリクエストデバイス」(368頁),
9.3.8.1節「ストリング入出力リクエストデバイス」(371頁) を参照.

【例 1】ウインドウを生成する

```
get_window_device(Device) :- true |
        shoen:raise(pimos_tag#task,general_request,GRS),
        GRS=[window(Result)],
        wait_window_request_device(Device,Result).

wait_window_request_device(Device,normal(WRS,_,_)) :- true |
        WRS=[create(Result)],
        wait_window_device(Device,Result).
otherwise.
wait_window_request_device(Device,Status) :- true | Device=Status.

wait_window_device(Device,normal(WS,Abort,Att)) :- true |
        Device=normal(WS1,Abort,Att),
        initiate(WS,WS1).
otherwise.
wait_window_device(Device,Status) :- true | Device=Status.

% 本当は Status も見た方が良い
initiate(WS,WS1) :- true |
        WS=[activate(_),reshape(mouse,mouse,_)|WS1].
```

【例 2】ファイルをオープンする

```
get_file_device(Pathname,Mode,Device) :- true |
        shoen:raise(pimos_tag#task,general_request,GRS),
        GRS=[file(Result)],
        wait_file_request_device(Pathname,Mode,Device,Result).

wait_file_request_device(Pathname,Mode,Device,normal(FRS,_,_)) :- true |
        FRS=[open(Pathname,{Mode,Device})].
otherwise.
wait_file_request_device(_,_,Device,Result) :- true | Device=Result.
```

9.1.4 入出力をする

以上のようにして入出力デバイスを獲得することができたら, デバイスへのストリームに対して入出力メッセージを送れば, 入出力を行うことができる.

【参考】各デバイスに送ることのできるメッセージはそれぞれ,
9.3.5.2節「ウインドウ」(360頁),
9.3.6.3節「ファイル」(365頁),
9.3.6.4節「ディレクトリ」(367頁),
9.3.7.2節「タイマ」(369頁),
9.3.8.2節「ストリング入出力デバイス」(371頁) を参照.

【例】ウインドウや入力ファイルから 256 文字入力する

```
input(DS,DS1,String) :- true |                  % DS, DS1 はデバイスへのストリーム
        DS=[getb(256,Result)|DS1],
        wait_input(Result,String).

wait_input(normal(String),String1) :- true | String=String1.
otherwise.
wait_input(_,String) :- true | String="".
```

リスナでの実行例 (入力ファイルから 256 文字入力する)

```
[10] test:get_file_device("login.com",read,normal(DS,_,_)),test:input(DS,[],String) .
 DS = [getb(256,normal("   % login?? "))]
 String = "   % login?? "
```

9.1.5 入出力ユーティリティのフィルタをつけて入出力をする

以上のような手順を踏めば, 正しくデバイスを操作することができる. ただし, この場合は生のデバイスをそのまま操作することとなる.

そこで通常, 一般のユーザは, 容易に効率良く入出力を行うために, デバイスへの出力ストリームの手前に入出力ユーティリティのフィルタを挿入して入出力を行うのが望ましい. フィルタには, 入力用, 出力用, 入出力両用の 3 種類がある. それぞれの用途に応じて使い分けるのがよい. 入出力ユーティリティについて, 詳しくは 9.2節「入出力ユーティリティ」(338頁) 参照. フィルタの挿入は以下のようにして行う.

【例 1】ウインドウデバイスに入出力ユーティリティのフィルタをつける

```
get_user_window_device(Window) :- true |
        get_window_device(Device),
        wait_device(Window,Device).

wait_device(Window,normal(WS,_,_)) :- true |
        buffer:interaction_filter(Window,WS).
otherwise.
wait_device(_,Device) :- true |          % デバイス生成に失敗したらエラーをあげる
        shoen:raise(pimos_tag#error,
            '$PIMOS$'(user_window,"failed_in_window_creation"),Device).
```

【参考】上の例にあるようにエラー報告を記述しておけば, 実行時にエラーが生じた場合, シェルやリスナが標準的な形式でエラーメッセージを表示してくれる. ユーザプログラムからのエラー報告の方法について, 詳しくは 11.5節 「標準に提供するエラー報告方法」 (428頁) 参照. また, 具体的にどのようにエラーが表示されるかについては 操作編 2.1.5.2節 シェルの「エラーメッセージ」 (25, 26頁), または 4.1.8.3節 リスナの「エラーメッセージ」【例 39】 (156頁) を参照.

【例 2】入出力ユーティリティのフィルタをつけたウインドウから 1 行入力する

```
input_line(Win,Win1,Line) :- true |
        Win=[prompt("input_line> "),getl(Line)|Win1].
```

リスナでの実行例

```
[10] test:get_user_window_device(Win),test:input_line(Win,[],Line).
 Line = "abc"
 Win = [prompt("input_line> "),getl("abc")]
```

【例 3】入出力ユーティリティのフィルタをつけたウインドウから項を 1 つ入力する

```
input_term(Win,Win1,Term) :- true |
        Win=[prompt("input_term> "),gett(Term)|Win1].
```

リスナでの実行例

```
[11] test: get_user_window_device(Win),test:input_term(Win,[],Term).
 Term = term
 Win = [prompt("input_term> "),gett(term)]
```

【例 4】入出力ユーティリティのフィルタをつけたウインドウから項をラップして入力する

```
input_wrapped_term(Win,Win1,Result) :- true |
        Win=[prompt("input_wrapped_term> "),getwt(Result)|Win1].
```

リスナでの実行例

```
[12] test: get_user_window_device(Win),test:input_wrapped_term(Win,[],Result).
 Result = normal(atom(wrapped_term))
 Win = [prompt("input_wrapped_term> "),getwt(normal(atom(wrapped_term)))]
```

【参考 1】変数を変数として扱いたい場合には, 項を **ラップした (wrap)** 状態で取り扱わなければならない. **ラップされた項** とは, 通常の項にそのデータ型にしたがって皮をかぶせたものである. ラップされた項の入出力には getwt/1, putwt/1 などのメッセージを用いる. 詳しくは 9.2.2節 「通常の項と wrap された項」 (348頁) を参照.

【参考 2】PIMOS のウインドウとファイルに対する具体的な文字の入出力メッセージは, 同一形式となっている. そのため, 入出力ユーティリティのフィルタは, 具体的なデバイスの種類を知ることな

く入出力を行なうことができるようになっている．したがって，ユーザが入出力ユーティリティのフィルタを挿入する際に，デバイスの種類を気にする必要はない．実際，上の 4 つの例題プログラムは，ファイルに対しても有効である．
また，入出力ユーティリティのフィルタは，具体的な入出力メッセージのみを解釈し，それ以外の知らないメッセージはデバイスに素通しする．したがって，ウインドウ属性設定メッセージなどのデバイス依存のメッセージもそのまま入出力ユーティリティのフィルタを介してデバイスに送ることができる．

9.2 入出力ユーティリティ

本節では，入出力に関して PIMOS が提供するユーティリティについて述べる．

なお，ここに記述する述語の呼出しによって得られるストリームには，すべてあらかじめ組込みのマージャ (第 7.4.19 節参照) が挿入されている．

9.2.1 バッファ

PIMOS は，通信オーバヘッドを軽減するためのブロッキング / デブロッキングを行うバッファリングフィルタ (以下，単にフィルタと呼ぶこともある) と，任意量の文字データを格納するための無限長バッファ機能を提供する．

バッファリングフィルタは，ユーザから入出力デバイスへつながるストリームの中間に挿入されるフィルタで，入力用，出力用，入出力両用の三種類がある．入力用フィルタは入力が必要になった時にある程度の文字数をまとめて入力して内部に溜めておき，要求に応じて少しずつ返してくれる．出力用フィルタはある程度の文字数のデータが溜まるまで内部にデータを溜めておき，まとめて出力する．入出力両用フィルタは両方の機能を持つ．これらのフィルタは，標準入出力ストリームを獲得する際に自動的に要求ストリームに挿入される (第 9.4.2.2 節参照)．

無限長バッファはユーザから入出力両方のメッセージを受け付ける．内部にいくらでも伸びうる FIFO バッファを持っており，出力したデータはここに溜め込まれる．入力要求があるとバッファの先頭から順に返される．

バッファリングフィルタと無限長バッファは，基本的には共通のインタフェースを備え，同じプログラムの入出力先を切り替えることによって両者を使い分けることができる．また，両者は内部に演算子の表 (オペレータブール) を保持しており，これに基づいてエジンバラ Prolog と同様の演算子順位文法によるデータの文字列表現の構文解析 (入力の場合) や，文字列表現の生成 (出力の場合) を行う機能を持つ．

なお，PIMOS で扱う文字入出力は原則として JIS 16 ビットコード系を用い，改行は文字コード 10 (十進) で表される．

9.2.1.1 共通プロトコル

フィルタと無限長バッファに共通するプロトコルについて述べる．

【注意】以下の記述中で入力の終わりやアボート状態に関する記述はフィルタに関するもので，無限長バッファでは入力の終わりになることもアボート状態になることもない．空の無限長バッファに getc 等の入力要求を送ると，別のメッセージによりバッファが空でなくなるまで入力要求処理は中断する．

入力用メッセージプロトコル

getc(^C)

次の一文字のコードと C とをユニファイする．入力の終わりならば -1 を，デバイスがアボート状態になった場合は -2 をユニファイする．

getl(^String)

次の一行を 16 ビットストリング として String とユニファイする. 入力の終わりならば -1 を, デバイスがアボート状態になった場合は -2 をユニファイする.

getb(Size, ^Block)

次の Size 文字を 16 ビットストリングとして Block とユニファイする. Size 文字読み込まないうちに入力の終わりになった場合はそこまでを 16 ビットストリングに変換する. 既に入力の終わりまで来ていたならば -1 を, デバイスがアボート状態になった場合は -2 をユニファイする.

getwt(^Result)

フルストップ (後に空白, タブまたは改行が続くピリオド) まで読み込み, その文字列をオペレータプール (346ページ参照) の指定に基づいて構文解析する. 解析が成功した場合, Result は `normal(WT)` とユニファイされる. ここで, WT は wrap された項である.

入力の終わりに来ていたならば, Result は `normal(end_of_file)` とまたフルストップのみの入力の場合は `normal(empty)` とユニファイされる.

入力項中に誤りがある場合は, Result は `abnormal(ErrorInfo)` とユニファイされる. `ErrorInfo` の内容は以下の通り.

error({TL,Ps}) 入力トークン列 TL の Ps 番目に誤りがある.

error({TL,ErrorReason,Ps}) 入力トークン列 TL の Ps 番目のトークンに誤りがある. ErrorReason はその理由を表すアトムで以下のものがある.

newline_occurred_in_string 文字列の途中に改行があった.

newline_occurred_in_quoted_atom クォートで囲ったアトムの途中に改行があった.

integer_over_flow 読み込み中に整数演算あふれが生じた. 32 ビットで表現しきれない値の整数値定数を入力した時に生じる.

atom_device_abnormal アトム名からアトムへの変換に失敗した (たぶん PIMOS のバグ).

ambiguous({TL,Ps}) 入力トークン列 TL の Ps 番目のトークンには複数の解釈の仕方がある.

【参考 1】 Wrap された項については第 9.2.2 節通常の項と wrap された項 (348ページ) を参照のこと.

【参考 2】 ErrorInfo を `print_error(ErrorInfo)` (342 ページ参照) に渡すことにより, 誤りの内容を表示することができる.

【参考 3】後述する gett は getwt を用いて実現されている.

【注意】エラーの示し方については, 将来改訂する予定がある.

gett(^T)

gett(^T,NVP)

gett(^T,^VP,NVP)

フルストップ (後に空白, タブまたは改行が続くピリオド) まで読み込み, その文字列をオペレータプール の指定に基づいて構文解析する. VP は解析時に使用される変数プール (348ページ参照) への

ストリームで, 解析の結果更新された変数プールへのストリームが NVP である. VP が指定されない場合は空の変数プールが生成され使われ, NVP が指定されない場合は変数プールは終了する.

解析が成功した場合, T は得られた (通常の) 項とユニファイされる. 入力の終わりに来ていたならばアトム `end_of_file` が具体化される.

【注意】フルストップのみの入力は読み飛ばされる.

入力項中に誤りがある場合は, T は次の何れかに具体化される. それぞれ getwt/1 の ErrorInfo に対応している.

$error({TL,Ps}) .

$error({TL,ErrorReason,Ps}) .

$ambiguous({TL,Ps}) .

skip(C)

文字コード C が出現するか入力の終わりになるまで読み飛ばす. 直後の getc(C) では C の次に続く文字コードがユニファイされる. 読み飛ばしている途中で入力の終わりになった場合, 直後の getc(C) では C と -1 がユニファイされる.

ungetc(C)

内部バッファに文字コード C を書き戻す. 直後の getc(C) ではこのコードが読める. Ungetc は連続して何回でもでき, 最後の ungetc に指定した文字から順に読める.

prompt(String)

対話的な入出力デバイスにつながっている場合は putb(String) と同じ, そうでなければ無視される.

【参考】Prompt は, ユーザの入力を促すプロンプトを表示するためのメッセージで, 入出力先デバイスの種類に依存せずプログラムを記述するためのものである. prompt/1 をウインドウに対して発行した場合, putb/1 と同様に動作するが, ファイルに対して発行した場合は何もしない.

出力用メッセージプロトコル

putc(C)

文字コード C で示される文字を書き出す.

putl(String)

String で示される文字列を書き出し, 改行を行う.

putb(String)

String で示される文字列を書き出す.

putwt(WT, PDepth,PLength, PSLength,PALength)

putwt(WT, PDepth,PLength, PSLength)

putwt(WT, PDepth,PLength)

putwt(WT)

オペレータプールに基づき, wrap された項 WT を演算子順位文法を用いて書き出す. 構造体の深さが PDepth を越える部分は “&”, 長さが PLength, PSLength, PALength を越える部分については “...” が書き出される (PDepth, PLength, PSLength, PALength が指定されない時はデフォルトの値が使われる (343 ページ参照)).

【注意 1】項の終わりのフルストップはこのメッセージや以下の putwtq, putt, puttq メッセージでは出力されない.

【注意 2】WT が wrap されていなければ, 出力は中断 (未定義変数を含んでいる場合) されるかフェイル (正しく wrap していない場合) する.

【参考】モジュール型は, “module$パッケージ名::モジュール名 $モジュールサイズ”, コード型は, “code$パッケージ名::モジュール名:述語名 / 引数個数 $ モジュールサイズ”, 出力イメージのない (アトムテーブルに登録されていない) アトムは, “atom$ アトム番号” という形式で出力される.

putwtq(WT, PDepth,PLength, PSLength,PALength)

putwtq(WT, PDepth,PLength, PSLength)

putwtq(WT, PDepth,PLength)

putwtq(WT)

Wrap された項 WT を適当にクォート付けして書き出す. Putwtq で出力された項は直後にフルストップを出力すれば getwt, gett 等で読み込むことができる.

【注意】出力した項を後で gett, getwt などで読み込みたい時は putwtq, puttq を使用する必要がある. これらは gett, getwt で誤りなく読み込めるように適宜アトムにクォートをつけたり, トークンの間にブランクを挿入する.

putt(T, PDepth,PLength, PSLength,PALength)

putt(T, PDepth,PLength, PSLength)

putt(T, PDepth,PLength)

putt(T)

オペレータプールに基づき, 項 T を演算子順位文法を用いて書き出す.

【注意】T が具体化されていない変数を含む場合は, 出力を中断する. これは以下の puttq メッセージでも同様である.

puttq(T, PDepth,PLength, PSLength,PALength)

puttq(T, PDepth,PLength, PSLength)

puttq(T, PDepth,PLength)

puttq(T)

項 T を適当にクォート付けして書き出す. Puttq で出力された項は直後にフルストップを出力すれば getwt, gett 等で読み込むことができる.

nl

改行を行う.

tab(N)

空白コードを N 個書き出す.

putf(FormatString, ArgumentList)

ArgumentList を書式 FormatString に従って出力する. ArgumentList は引数リストで, 引数を取る書式項目についてはこのリストの先頭から順に引数を取る.

書式はストリングで与え, ~ で始まる書式項目以外の文字はそのまま出力される. 書式項目の一般形は ~NC で, C は書式のタイプを表し, N は正整数でオプショナルである.

~N**c** (Character) 引数を文字コードと解釈し, その文字を N 回出力する. N のデフォルトは 1.

~N**s** (String) 引数のストリングの先頭から N 文字を出力する. N のデフォルトは引数ストリングの長さ.

~N**b** (Blank) N 個の空白を出力する. N のデフォルトは 1.

~N**t** (Term) 引数の項を putt と同じ形式で N 回出力する. Nのデフォルトは 1.

~N**T** (Term) 引数の項を puttq と同じ形式で N 回出力する. Nのデフォルトは 1.

~N**w** (Term) 引数の項を putwt と同じ形式で N 回出力する. Nのデフォルトは 1.

~N**W** (Term) 引数の項を putwtq と同じ形式で N 回出力する. Nのデフォルトは 1.

~N**n** (Newline) N 個の改行を出力する. N のデフォルトは 1.

~N~ 文字 "~" 自身を N 個出力する. N のデフォルトは 1.

【注意】FormatString で指定された書式項目数に対して Argument が足りない場合はフェイルする. また, Argument が多い場合はその分無視される.

print_error(ErrorInfo)

Getwt/1 の異常終了時に返される `abnormal(ErrorInfo)` の ErrorInfo を引数として渡すことによりエラー情報が表示される.

flush(^Status)

バッファに溜まったデータを 入出力先デバイスに書き出す. 出力が完了すると (FEP 上のデバイスとつながっている場合は, バッファの内容が FEP に送られ, FEP が出力動作の完了を通知してくると) Status にアトム normal がユニファイされる. 無限長バッファの場合は, このメッセージ以前に発行された出力メッセージの処理が全て終わった後に Status にアトム normal を返す.

フィルタでは I/O デバイスとの通信回数を減らすために入出力データのブロッキングを行っている. このため出力メッセージを送っただけではフィルタのバッファに蓄えられているだけで, I/O デバイスには送られないことがある. バッファが I/O デバイスに送られるのは以下の場合である.

- バッファがいっぱいになったとき
- flush/1, putl/1 メッセージを受け取ったとき
- ストリームを閉じた時

オペレータプール用メッセージプロトコル

プロトコル gett, putt などに用いる演算子の操作を行うプロトコル (詳しくは第 9.2.1.6 節 (346 ページ) を参照).

add_op(Prec, Type, OpName)

優先順位 Prec, 型 Type の演算子 OpName (アトム) をオペレータプールに追加する.

remove_op(Prec, Type, OpName)

優先順位 Prec, 型 Type の演算子 OpName(アトム) をオペレータプールから削除する. 登録されてなければ何もしない.

operator(OpName, ^Def)

演算子 OpName の優先順位と型の組のリストを Def とユニファイする. 組は "{ 優先順位, 型 }" の形式である. OpName が登録されていなければ Def は [] とユニファイされる.

制御メッセージプロトコル

do(Stream*)

Stream は親のストリームと同じプロトコルのメッセージを要素とするリスト (必ず閉じられるべきストリーム) である. Stream 内のメッセージは連続して処理される.

メッセージストリームに多くのストリームがマージされる時, 各ストリーム内でのメッセージの順序は保証されるが, マージされたストリーム群のどのストリームのメッセージが先に処理されるかは保証がない. そこで, 連続して処理することを保証したい場合 (たとえば, 一連の文字列を出力して改行するまでは, 別の出力を間に入れたくない場合など) には, この do メッセージを用いて処理の連続性を保証する必要がある.

【参考】`S = [do(S1)|S2]` は `append(S1,S2, S)` と同じ意味である.

【参考】メッセージ do の引数リストの中にまた do があってもよい.

replace_op_pool(OldOpPool, ^NewOpPool)

オペレータプールを OldOpPool から NewOpPool に変更する.

change_op_pool(^NewOpPool)

`replace_op_pool([], NewOpPool)` と同じ.

print_depth(PDepth)

print_length(PLength)

print_string_length(PSLength)

print_atom_length(PALength)

構造体出力時の構造体の深さ, 長さ, ストリングの長さ, アトム名の長さの制限のデフォルト値をそれぞれ, PDepth, PLength, PSLength, PALength に変える. 制限値の初期値は構造体の深さ 4, 長さ 10, ストリングの長さ 20, アトム名の長さ 30 である.

print_depth(^OPDepth,NPDepth)

print_length(^OPLength,NPLength)

print_string_length(^OPSLength,NPSLength)

print_atom_length(^OPALength,NPALength)

構造体出力時の構造体の深さ, 長さ, ストリングの長さ, アトム名の長さの制限のデフォルト値をそれぞれ, OPDepth, OPLength, OPSLength, OPALength から NPDepth, NPLength, NPSLength, NPALength に変える.

アボート / アテンションメッセージプロトコル

第 9.3.3 節デバイスのアボート / アテンション処理 (356 ページ) において述べられるプロトコル.

reset(^Result)

resend(ID, ^Result)

cancel(ID, ^Result)

next_attention(^Result)

【注意】バッファは一度アボート状態になると, 終了までアボート状態のままである.

9.2.1.2 入力用フィルタ

作成

buffer:input_filter(Init, In*, ^Out)

初期設定 Init の入力用フィルタを作る. Init は以下の要素からなるリストである. フィルタは In (ユーザ側) からコマンドプロトコルを読み, Out (デバイス側) にコマンドプロトコルを出す.

size(Size) 内部バッファの大きさを Size とする. デフォルトは 490 である.

atom_table(^AtomDev) アトムデータベース (385 頁参照) を AtomDev とする. AtomDev はアトムデータベースへのストリーム. デフォルトは `atom_table:cache_table(AtomDev)` である.

operator_pool(^OpPool) オペレータプール (第 9.2.1.6 節参照) を OpPool とする. OpPool はオペレータプールへのストリーム. デフォルトは `operator:pool(OpPool)` である.

buffer:input_filter(In*, ^Out)

`buffer:input_filter([], In,Out)` と同じ.

ユーザ (In) 側プロトコル

入力用 (338 ページ参照), オペレータプール用 (342 ページ参照), 制御用及びアボート / アテンションのメッセージプロトコル (344 ページ参照) を受け付ける.

【注意】知らないプロトコルの場合は素通しする. つまりデバイス側にそのまま流す.

デバイス (Out) 側プロトコル

getb(Size, ^Status)

prompt(String, ^Status)

9.2.1.3 出力用フィルタ

作成

buffer:output_filter(Init, In*, ^Out)

初期設定 Init の出力用フィルタを作る. Init の仕様は入力用フィルタと同じ. フィルタは In (ユーザ側) からコマンドメッセージを読み, Out (デバイス側) にコマンドメッセージを出す.

buffer:output_filter(In*, ^Out)

`buffer:output_filter([], In,Out)` と同じ.

ユーザ (In) 側プロトコル

出力用 (340 ページ参照), オペレータプール用 (342 ページ参照), 制御用及びアボート / アテンションのメッセージプロトコル (344 ページ参照) を受け付ける.

【注意】知らないプロトコルの場合は素通しする. つまりデバイス側にそのまま流す.

デバイス (Out) 側プロトコル

putb(String, ^Status)

flush(^Status)

9.2.1.4 入出力両用フィルタ

作成

buffer:interaction_filter(Init, In*, ^Out)

初期設定 Init の入出力両用フィルタを作る. Init の仕様は入力用フィルタと同じ. フィルタは In (ユーザ側) からコマンドメッセージを読み, Out (デバイス側) にコマンドメッセージを出す.

buffer:interaction_filter(In*, ^Out)

`buffer:interaction_filter([], In,Out)` と同じ.

ユーザ (In) 側プロトコル

入力用 (338 ページ参照), 出力用 (340 ページ参照), オペレータプール用 (342 ページ参照), 制御用及びアボート / アテンションのメッセージプロトコル (344 ページ参照) を受け付ける.

【注意】知らないプロトコルの場合は素通しする. つまりデバイス側にそのまま流す.

デバイス (Out) 側プロトコル

getb(Size, ^Status)

prompt(String, ^Status)

putb(String, ^Status)

flush(^Status)

9.2.1.5 無限長バッファ

作成

buffer:character_pool(Init, In*)

初期設定 Init の無限長バッファを作る. Init の仕様は入力用フィルタと同じ. バッファへの書込み, バッファからの読出し共に In からのコマンドメッセージによって行う.

buffer:character_pool(In*)

`buffer:character_pool([], In)` と同じ.

プロトコル

入力用 (338 ページ参照), 出力用 (340 ページ参照), オペレータプール用 (342 ページ参照) を受け付ける.

【注意】知らないプロトコルの場合はフェイルする.

9.2.1.6 オペレータプール

オペレータ順位文法によるパース, アンパースに用いる演算子のプールである. 演算子は型と優先順位を持つ. 型は, 演算子が前置 (prefix), 間置 (infix), 後置 (postfix) のどれか, 及び同じ優先順位の場合の結合の仕方を示すもので, fx, fy, xfx, yfx, xfy, xf, yf という種類がある. 優先順位は, 1 以上 1200 以下の整数で表され, 演算子の結合力の強さを表す (値が小さいほど結合力が強い).

【注意】同じアトムを複数の型の演算子として登録することもできる.

作成

operator:pool(In*)

標準の演算子を登録済みのオペレータプールを作る.

標準に用意している演算子を表 9.1 に掲げる.

operator:empty_pool(In*)

何の演算子も登録されていない空のオペレータプールを作る.

プロトコル

オペレータプール用 (342 ページ参照) を受け付ける.

表 9.1: 標準の演算子

優先度	種類	演算子
1200	xfx, fx	`:-`
	xfx	`-->`
1150	fx	`public`, `implicit`, `local_implicit`, `with_macro`
1100	xfy	`;`, `|`
1090	xfx	`=>`
1050	xfy	`->`
1000	xfy	`,`
800	xfx	`:`
700	xfx	`=`, `\=`, `==`, `=:=`, `=\=`, `<`, `>`, `=<`, `>=`, `:=`, `<=`, `<<=`, `$=:=`, `$=\=`, `$<`, `$>`, `$=<`, `$>=`, `$:=`, `$<=`
	xfy	`@`
500	yfx,fx	`+`, `-`
	yfx	`/\`, `\/`, `xor`
400	yfx	`*`, `/`, `<<`, `>>`
300	xfy	`**`
	xfx	`mod`
200	fx	`&`
150	xf	`++`, `--`
100	xfx,fx	`#`
90	xfx	`::`
80	fx	`module`

9.2.1.7 変数プール

変数プールは変数名 (ストリング), 変数の番地と変数との対応表である.

作成

variable:pool2(In*)

空の変数表を作る.

プロトコル

count(^N)

現在登録されている変数名の数と N をユニファイする.

unify

同じ変数名で登録された変数をユニファイする.

reset

同じ変数名で登録された変数をユニファイした後, 変数表を初期化する.

【参考】変数プールは終了処理として自分自身に reset を送る.

put_var(Name, ?V)

変数名 Name で変数 V を登録する.

get_name({PE,ADR}, ^Result)

Result と {N,Name} をユニファイする. ここで N 及び Name はデレフ先がプロセッサ番号 PE, アドレス ADR である変数の変数番号と変数名である. 変数番号は変数表で一意につけられた整数である.

get_value(Name, ^Val)

変数名 Name で登録されている変数の値が Value の時, Val を {Value} とユニファイする. 変数が登録されていない時は, {} とユニファイする.

【注意】変数のあるプロセッサとアドレスはユニフィケーション及び GC によって変わり得る. 変数プールはユニフィケーションによってアドレスが変わったのか, GC によってアドレスが変わったのか知ることができないので違う変数を同じものと見なすこともある.

9.2.2 通常の項と wrap された項

並列論理型言語である KL1 では, 逐次型の Prolog などと異なり, 変数がまだ変数のままであることを認識することには, 本来意味がない. なぜなら, 変数であると認識した直後には (並行して走る別のプロセスにより) 既に変数ではなくなっているかも知れないからである. したがって, 変数を変数として認識する必要のあるメタレベルのプログラム (例えばデバッガなど) を作成する場合はそれなりの工夫が必要となる. そのための一つの考え方が, 通常の項と wrap された項の二つを考えることである. これは従来使われてきた変数を基底項と見なす 'フリーズ' という考えの拡張である.

通常の項は KL1 で普通に扱われる項である. Wrap された項というのは通常の項にそのデータ型にしたがって皮を被せたものである. 例えば, " X + 1 " は通常の項としては { +, X, 1} , Wrap された項としては vector({atom(+),variable("X"),integer(1)}) と内部表現される. Wrap された項というのは, ソフトレベルでデータにタグを付けたものと見なすことができる.

皮の被せ方は以下の規則による.

アトム atom(Atom)

整数 integer(Integer)

浮動小数点数 floating_point(FloatingPoint)

変数 variable(変数名) : 変数名は (通常の)16 ビットストリング. 変数名は変数をテキスト入力した場合はその変数名ストリング, 実行中に作られた変数についてはシステムで適当に作られたストリングである (第 9.2.2.1 節参照).

ストリング string(String)

リスト list(WrappedList)

ベクタ vector(WrappedVector)

モジュール module(Module)

コード code(Code)

その他 unknown(Unknown)

通常の項と wrap された項の変換を行う述語が以下に述べる wrap と unwrap である.

9.2.2.1 Wrap

Wrap は通常の項を wrap された項に変換する述語である.

variable:wrap(T, ^Result, D,L)

variable:wrap(T, ^Result, D,L, NVP)

variable:wrap(T, ^Result, D,L, ^VP,NVP)

通常項 T を深さ D 長さ L の範囲内で wrap する. Result と {NewT, WT} をユニファイする. ここで NewT は T にユニファイされ, WT は T を wrap した項にユニファイされる.

変数プール VP が指定されたらそれを用い, 指定されなければ空の変数プールを用いる. 変数プールに登録されていない変数には適当に変数名が付けられる. Wrap によって更新された変数プールが NVP である. NVP が指定されていない場合は使われた変数プールは終了する.

variable:wrap_address(T, ^Result, D,L)

変数の扱い方以外は variable:wrap/4 と同じ. 変数は variable({PE,ADR}) と wrap される. ここで PE, ADR は変数があるプロセッサ番号とアドレスである.

【参考】これは, デバッグ等で変数のアドレス表示が欲しいときになどに有効である.

【例】Wrap_address で得られた wrap された項 WT を putwt(WT) することにより変数のアドレス表示を得ることができる.

【注意】Wrap はある時点での未定義変数を置き換えるものである. したがって, 同じ項に対して wrap した結果がいつも等しいとは限らない. この機能はメタレベルの処理を必要とする特殊なプログラムの中でのみ用いるべきもので, 通常のプログラムでの使用には適さない.

9.2.2.2 Unwrap

Unwrap は wrap された項を通常の項に変換する述語である.

variable:unwrap(WT, ^Result)

variable:unwrap(WT, ^Result, NVP)

variable:unwrap(WT, ^Result, ^VP,NVP)

Wrap された項 WT を unwrap する. Result は {`NewWT, T`} とユニファイされる. ここで NewWT は WT と, T は unwrap した項とユニファイされる.

変数プール VP が指定されたらそれを用い, 指定されなければ空の変数プールを用いる. Wrap によって更新された変数プールが NVP である. NVP が指定されていない場合は使われた変数プールは終了する.

【注意】 wrap_address で得られた wrap された項は unwrap することはできない.

9.2.3 タイマに関するユーティリティ

PIMOS のタイマは, 日付及び時刻を獲得する機能を持つ (プログラミング編 9.3.7.2節「タイマ」 369頁参照). これらの値はそれぞれ, 1901 年 1 月 1 日から本日までの日数 (正整数) および, 本日 0 時 0 分 0 秒から現在までの時間単位数 (単位の既定値はミリ秒, 正整数) で返される. 本節では, これらの値から日付及び時刻を算出するユーティリティ, 及び与えられた時間単位数の単位変換を行うユーティリティについて述べる.

【注意 1】 以下, 「日数」はすべて 1901 年 1 月 1 日から数えられた値とする.

【注意 2】 時間単位数から時刻を算出するユーティリティでは, 時間単位数はミリ秒を単位として表されていなければならない. それ以外の単位で表された値は, まず時間単位変換ユーティリティ (353頁) で単位をミリ秒に変換する必要がある.

【例】 タイマから日数と時間数を得る

```
% go(Date,Time) :- true |
%         shoen:raise(pimos_tag#task,general_request,GRS),
%         GRS=[timer(Result)],
%         wait_timer_request_device(Date,Time,Result).
%
% wait_timer_request_device(Date,Time,normal(TRS,_,_)) :- true |
%         TRS=[create(Result)],
%         wait_timer_device(Date,Time,Result).
% otherwise.
% wait_timer_request_device(Date,Time,_) :- true | Date=0,Time=0.
%
% wait_timer_device(Date,Time,normal(TS,_,_)) :- true |
%         TS=[get_date_time(Result)],
%         wait_date_time(Date,Time,Result).
% otherwise.
% wait_timer_device(Date,Time,_) :- true | Date=0,Time=0.
%
% wait_date_time(Date,Time,normal(D,T)) :- true | D=Date,T=Time.
% otherwise.
% wait_date_time(Date,Time,_) :- true | Date=0,Time=0.

[1] timer_test:go(Today,Now).                    % 上のゴールを実行すると
^Now = 53800039
^Today = 33166

[2]
```

9.2.3.1 日付に関するユーティリティ

timer_utility:get_date_string(Date_count, ^String)

日数 Date_count(正整数) から, 日付を表す文字列を生成し, String に返す.

【例】

```
[10] v.
^Today = 33166

[11] timer_utility:get_date_string(Today,String).
 String = "22-Oct-1991"
```

timer_utility:get_date_value(Date_count, ^Year, ^Month, ^Day, ^Week)

日数 Date_count(正整数) から日付を算出し, 年, 月, 日, 曜日をそれぞれ Year, Month, Day, Week に返す. Year は年を表す正整数, Month は 1 から 12 までの正整数, Day は 1 から 31 までの正整数, Week は 0(日曜日) から 6(土曜日) の正整数である.

【例】

```
[12] v.
^Today = 33166

[13] timer_utility:get_date_value(Today,Year,Month,Day,Week).
 Day = 22
 Month = 10
 Week = 2
 Year = 1991
```

timer_utility:get_date_count(Year, Month, Day, ^Date_count)

1901 年 1 月 1 日から Year, Month, Day で表される日付までの日数を算出し, Date_count(正整数) に返す.

【例】

```
[14] timer_utility:get_date_count(1992,6,1,^FGCS).
^FGCS = 33389
```

timer_utility:month(Mon_index, ^Name, ^AbbrName, ^Days)

Mon_index に月を表す 1 から 12 までの正整数を与えると, Name に月の名称 (文字列), AbbrName に省略名称 (文字列), Days にその月の日数 (正整数) を返す.

【例】

```
[15] timer_utility:month(10,Name,AbbrName,Days).
 AbbrName = "Oct"
 Days = 31
 Name = "October"
```

timer_utility:day(Day_index, ^Name, ^AbbrName, ^JapaneseName)

Day_index に曜日を表す 0 から 6 までの正整数を与えると, Name に曜日の名称 (文字列), AbbrName に省略名称 (文字列), JapaneseName に日本語による省略名称 (文字列) を返す.

【例】

```
[16] timer_utility:day(2,Name,AbbrName,JapaneseName).
 AbbrName = "Tue"
 JapaneseName = "火"
 Name = "Tuesday"
```

9.2.3.2 時刻に関するユーティリティ

timer_utility:get_time_string(Time_count, ^String)

ミリ秒数 Time_count(正整数) から, 時刻を表す文字列を生成し, String に返す.

【例】

```
[17] v.
^Now = 53800039

[18] timer_utility:get_time_string(Now,String).
 String = "14:56:40"
```

timer_utility:get_time_value(Time_count, ^Hour, ^Min, ^Sec, ^Millisec)

ミリ秒数 Time_count(正整数) から時刻を算出し, 時, 分, 秒, ミリ秒をそれぞれ Hour, Min, Sec, Millisec に返す. Hour は 0 から 23 までの正整数, Min 及び Sec は 0 から 59 までの正整数, Millisec は 0 から 999 までの正整数である.

【例】

```
[19] v.
^Now = 53800039

[20] timer_utility:get_time_value(Now,Hour,Min,Sec,Millisec).
 Hour = 14
 Millisec = 39
 Min = 56
 Sec = 40
```

timer_utility:get_time_count(Hour, Min, Sec, Millisec, ^Time_count)

0 時 0 分 0 秒から Hour, Min, Sec, Millisec で表される時刻までのミリ秒数を算出し, Time_count (正整数) に返す.

【例】

```
[21] timer_utility:get_time_count(12,0,0,0,Noon).
 Noon = 43200000
```

9.2.3.3 日付と時刻の両方を扱うユーティリティ

timer_utility:add_date_time(Date1, Time1, Date2, Time2, ^Date, ^Time)

日数及びミリ秒数の対で表される時刻と時間の加算, 或いは時間どうしの加算を行う.

Date1 (正整数) と Time1 (正整数) の対で表される時刻 (或いは時間) に Date2 (正整数) と Time2 (正整数) の対で表される時間を加え, 結果を Date (正整数) と Time (正整数) の対に返す.

【例】一週間後の日付を求める

```
[22] v.
^Today = 33166

[23] timer_utility:add_date_time(Today,0,7,0,_D,_),
timer_utility:get_date_string(_D,NextWeek).
 NextWeek = "29-Oct-1991"
```

timer_utility:subtract_date_time(Date1, Time1, Date2, Time2, ^Date, ^Time)

日数及びミリ秒数の対で表される時刻から時間の減算, 或いは時間どうしの減算を行う.

Date1 (正整数) と Time1 (正整数) の対で表される時刻 (或いは時間) から Date2 (正整数) と Time2 (正整数) の対で表される時間を減じ, 結果を Date (正整数) と Time (正整数) の対に返す.

【例】FGCS まであと 222 日と 9 時間 3 分.

```
[24] v.
^FGCS = 33389
^Now = 53800039
^Today = 33166

[25] timer_utility:subtract_date_time(FGCS,0,Today,Now,Days,_T),
timer_utility:get_time_string(_T,Time).
 Days = 222
 Time = "09:03:19"
```

9.2.3.4 時間単位変換ユーティリティ

time_unit_translator:trans({Unit, Count}, {NUnit, ^NCount})

時間単位 Unit (アトム) で表された時間単位数 Count (正整数) を時間単位 NUnit (アトム) に変換し, NCount に返す. 時間単位として以下のものを指定することができる:

sec: 秒
msec: ミリ秒
msec16: 16 分の 1 ミリ秒

【参考】時間単位の種類は, 将来追加される可能性がある.

【注意】変換結果が整数で表せない大きな数となる場合, -1 を返す. 将来, 多倍長整数に関する演算モジュール (bignum) の提供と同時に改修予定.

【例】

```
[1] time_unit_translator:trans(msec(1000),sec(X)).
 X = 1
```

9.3 入出力制御

PIMOS が提供する入出力デバイスは, 通常の入出力処理を行うためのストリームの他に, 2 本の制御ラインを持っている. ひとつは PIMOS からの中断要求を受け付けるためのアボートライン, もうひとつはユーザからの割り込みを PIMOS に通知するためのアテンションラインである. 本節では, デバイスの制御方法, 及び生のデバイスを操作するためのメッセージプロトコルについて説明する.

9.3.1 デバイスの階層

PIMOS が提供する入出力デバイスは, それぞれの役割機能に応じて階層構造を成している. その最上位に位置するのがジェネラルリクエストデバイスである. ジェネラルリクエストデバイスは, PIMOS がユーザに提供する様々なサービスを行うデバイスである. このデバイスに対して, 入出力資源を要求すると, 入出力デバイスを生成して提供する役割を持つデバイスを獲得することができる. 入出力デバイスを生成して提供するデバイスを **リクエストデバイス** と呼ぶ. デバイスの階層で, ジェネラルリクエストデバイスの一段下に位置するデバイスである. さらにリクエストデバイスの一段下に, ウインドウやファイルなどの具体的な入出力資源を持ち, 入出力処理を行う **デバイス** がある.

リクエストデバイス

ジェネラルリクエストデバイスの一段下の階層に位置するデバイスである. リクエストデバイスは, ウインドウ, ファイルなどの具体的な入出力デバイスを生成してユーザに提供する役割機能を持つ. リクエストデバイスは, デバイスのタイプごとに, ウインドウリクエストデバイス, ファイルリクエストデバイス, タイマリクエストデバイス, ストリング入出力リクエストデバイスの 4 種類がある.

各リクエストデバイスにストリームを通じてメッセージを送信することにより, それぞれ具体的なデバイスを生成し, それへのストリームを獲得する. 各リクエストデバイスの獲得については 9.1節 「入出力資源の獲得」 (333頁) を参照. また, リクエストデバイスにはアボートライン, アテンションラインが設けられており, アボート / アテンション処理を行うことができる. 詳しくは 9.3.3節「デバイスのアボート / アテンション処理」 (356頁) 参照.

【参考】リクエストデバイスをアボートするとは, リクエストデバイスが受信したメッセージに対応する処理をアボートするということである. すなわち, ウインドウの生成, ファイルのオープンなどをアボートする, 或いは, それらのメッセージを実行せずに放棄することに当たる.

デバイス

リクエストデバイスの一段下の階層に位置する. タイプごとにそれぞれのリクエストデバイスによって生成され, 提供される. PIMOS には, ウインドウ, ファイル, タイマ, ストリング入出力デバイスの 4 種類がある. デバイスはそれぞれ具体的な入出力資源を持ち, ストリームを介したメッセージ通信によって入出力等の処理を行う. また, 各デバイスにはアボートライン, アテンションラインが設けられており, アボート / アテンション処理を行うことができる. 詳しくは 9.3.3節「デバイスのアボート / アテンション処理」(356頁) 参照.

9.3.2 デバイスとの通信方式

各デバイスとのインタフェースは, ストリームを介したメッセージ通信である. 通常の入出力は, デバイスの持つ入出力用ストリームにメッセージを送ることによって行う. 入出力メッセージの形式は, ある規則に基づいて決められている. デバイスはさらに, OS からの要求, またはユーザからの割り込みにより入出力を中断するためのラインを 2 本持っている.

入出力メッセージの形式

入出力をするには, 各デバイスへ入出力メッセージを流す.

各メッセージは, メッセージ名, 0 個以上の入力用引数, 1 個の出力用引数から成る. デバイスは受け取ったメッセージのメッセージ名と入力用引数に応じて入出力を実行し, 結果を出力用引数に返す. 出力用引数には, 以下の規則にしたがって値が返される:

メッセージに対応する処理が正常終了した場合で,

その処理が成功したか失敗したかのみを返すような場合
出力用引数に **normal** (アトム) を返す.

その処理の結果として得られる値を返すような場合
出力用引数に, 第 0 要素目を **normal** (アトム) とするようなベクタを返す. 第 1 要素以降に処理の結果得られた値がはいっている.

メッセージに対応する処理が正常終了しなかった場合
出力用引数にその理由を返す. 以下のいずれかの値を取る:

abnormal (アトム) : 処理に失敗した

aborted (アトム) : 処理を放棄した

disconnected (アトム) : ネットワークを介したリモートサイトでの入出力資源の利用中に, 処理を行おうとしてネットワークが切れていることが検出された

【注意】現在, 一旦ネットワークが切れていることが検出されたデバイスに対して, 入出力処理を再開することはできない. ネットワークの復帰後, 新しいデバイスを新たに獲得し直す必要がある.

PIMOS の入出力資源に対して送ることのできるメッセージの説明は, 7.1節のモード表示にしたがって記述してある. 詳しくは, プログラミング編 7.1節「モード表示の読み方」(222頁) を参照. また, メッセージ中の出力用引数には, 以下の変数名を用いた.

^**Status:** メッセージに対応する処理が正常終了した場合に, その処理が成功したか失敗したかのみを返すようなメッセージの出力用引数を表す

^**Result:** メッセージに対応する処理が正常終了した場合に, その処理の結果として得られる値を一緒に返すようなメッセージの出力用引数を表す

アボートラインとアテンションライン

デバイスには, 通常のメッセージを処理する他に, それらのメッセージの処理中にも OS からの要求, またはユーザからの割り込みによりそれらを中断する機構が必要である.

デバイスは, 通常, ストリームを通じて受信したメッセージを受信順に処理していく. 処理中のメッセージの中断を指示するためには, ストリーム中に溜まっている未処理のメッセージより先にデバイスに通知する必要があるので, このストリームを割り込み通知用に用いることはできない.

そこで, PIMOS が提供するすべてのデバイスには, 通常のメッセージを受信するためのストリームの他に, アボートライン, アテンションラインと呼ばれる通信路を設けている. アボートラインは, ユーザまたは OS がデバイスに対して処理の中断を指示するためのものであり, アテンションラインは, デバイスに割込みがはいったことをユーザまたは OS に通知するためのものである. デバイスのアボート / アテンション処理について詳しくは, 次節「デバイスのアボート / アテンション処理」を参照.

9.3.3 デバイスのアボート / アテンション処理

すべてのデバイスは, 通常の入出力メッセージを送信するためのストリームの他に, OS からの中断要求を受け付けるためのアボートライン, ユーザからの割り込みを OS に通知するためのアテンションラインを持つ. これらのストリーム, ラインは, デバイス生成時に張られる.

2 本のラインは変数で実現されている. OS またはユーザは, アボートラインの変数をアトム **abort** に具体化することにより, 当該デバイスが処理中のメッセージを中断させることができる. また, ユーザがキーボードから **^C** を入力することにより, アテンションラインの変数が デバイスによって **attention** に具体化され, OS またはユーザに割り込みを通知することができる (現在はウインドウデバイスのみ有効).

デバイスには 3 つの状態がある.

- アボートされた状態
- 通常のメッセージ待ち状態
- メッセージに対する処理の終了待ち状態

アボートライン, アテンションラインは変数で実現されているため, 一度使用されたラインは再使用できない. 従って abort が送られた後のデバイスはアボートラインが張られていない状態となる. この状態をデバイスがアボートされた状態と呼ぶ. これによって中断されたメッセージ, 及びこの状態の時に受信した通常の入出力メッセージは, すべて実行されずにデバイス内に格納される. デバイスを再び通常のメッセージ待ち状態にするには, 新しいアボートライン, アテンションラインを張り直す必要がある. ストリームを通じて以下のメッセージを流すことによってラインの張り直しを行う.

reset(^Result)

新しいアボートライン, アテンションラインを張る.

メッセージが正常終了した場合, 新しいライン Abort, Attention を張り, 中断メッセージ列の識別番号 ID を Result に **normal(^Abort, Attention, ID)** の形式で返す. 識別番号 ID は, デバイスがアボートされてからこのメッセージの到着直前までの間に, デバイスがアボート状態だったために実行されずにデバイス内に格納された保留メッセージ列を指すものである.

ラインが既に張られている時に reset メッセージを受信した場合, デバイスは中断要求が来たとみなして既存のアボートラインを abort に具体化し, 新しいラインを張り直す. メッセージの成功を待ってアテンションラインは新しいものに張りかえられ, 古いアテンションラインはアトム [] に具体化される. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

デバイスがアボート状態になると, それ以降上述の reset を受信するまでのメッセージ列は, 実行されずにデバイス内に格納され, 保留状態になる. デバイスが通常のメッセージ待ち状態になった後, 次のメッセージによって, このように保留されたメッセージ列に対して, 再試行あるいは実行放棄を指示でき

る. 再試行 / 放棄は, デバイスがアボートされた状態となった 1 期間を単位として指示することができる.

いずれのメッセージも送られない場合, 格納されたメッセージはデバイスの終了時に実行を放棄される.

resend(ID, ^Status)

識別番号 ID の保留メッセージ列を再試行する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

cancel(ID, ^Status)

識別番号 ID の保留メッセージ列の処理を放棄する. この時, 処理を放棄されたメッセージの出力用引数は **aborted** に具体化される. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

ユーザから割り込みが通知されたが, 処理の中断を行わない場合には, 以下のメッセージを用いて, アテンションラインのみを新しく張り直すことができる.

next_attention(^Result)

新しいアテンションラインを張る.

メッセージが正常終了した場合, Result に **normal(Attention)** を返し, 新しいアテンションライン Attention を張る.

既にラインが張られている時に next_attention メッセージが送られた場合, メッセージの成功を待って新しいラインに張りかえられる. 古いラインは アトム [] に具体化される. メッセージが失敗した場合は, 古いラインが有効である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

なお, 張られているアテンションラインはデバイスの終了時にアトム [] に具体化される.

9.3.3.1 ウインドウデバイスのアボート処理

ウインドウデバイスでは, ウインドウから入力されたが, まだユーザに返していない文字列 (入力残留文字列) を管理している. 入力残留文字列について, 詳しくは ウインドウからの getb (360節 ウインドウへの「入出力メッセージ」360頁) を参照. 従って, ウインドウデバイスへのアボートに関するメッセージは, 上記の動作の他に入力残留文字列に対する以下のような処理を行う.

reset(^Result)

入力残留文字列を空文字列に初期化する. 実行の結果として返される識別番号 ID は, 中断メッセージ列と, アボートされた時点でウインドウデバイスが管理していた入力残留文字列の両者を指す. resend, cancel メッセージにこの ID を指定すると, 中断メッセージ列だけでなく入力残留文字列の回復, 解放も行う. いずれのメッセージも送られない場合, 格納された入力残留文字列はウインドウデバイスの終了時に解放される.

resend(ID, ^Status)

識別番号 ID の入力残留文字列を回復する. 既に入力残留文字列が存在する場合はこれを捨て, 識別番号 ID の入力残留文字列を採用する. ただし, 正しくない識別番号が指定された場合は, 既にあった方の入力残留文字列が有効である.

cancel(ID, ^Status)

識別番号 ID の入力残留文字列を解放する.

9.3.4 入出力デバイスのデバイス名

ユーザが利用中の入出力デバイスは, PIMOS の資源管理機構によって管理されている. ユーザは, PIMOS に入出力デバイスの資源状態を問い合わせることができる. 資源状態の表示に用いられる **デバイス名** は以下のようになっている. また, デバイス名と共に表示されるコメントは, それぞれ以下のような内容を表している.

ジェネラルリクエストデバイス

デバイス名: general_request

コメント: "general_request"

ウインドウリクエストデバイス

デバイス名: window_request

コメント: "window_request"

ウインドウデバイス

デバイス名: window

コメント: window, 或いは生成時に指定したバッファ名文字列

ファイルリクエストデバイス

デバイス名: file_request

コメント: "file_request"

入力ファイルデバイス

デバイス名: read_file

コメント: ファイル名文字列

出力ファイルデバイス

デバイス名: write_file

コメント: ファイル名文字列

追加ファイルデバイス

デバイス名: append_file

コメント: ファイル名文字列

ディレクトリデバイス

デバイス名: directory

コメント: ディレクトリ名文字列

タイマリクエストデバイス

デバイス名: timer_request

コメント: "timer_request"

タイマデバイスで, FEP のタイマを用いている場合

デバイス名: fep_timer

コメント: fep

タイマデバイスで, システムタイマを用いている場合

デバイス名: system_timer

コメント: {absolute_id(ABS_ID), relative_id(REL_ID)}
ABS_ID はシステムタイマのある絶対ノード番号, REL_ID は相対ノード番号である.

ストリング入出力リクエストデバイス

デバイス名: string_io_request

コメント: "string_io_request"

ストリング入出力デバイス

デバイス名: string_io

コメント: デバイスを定義している ESP プログラムのパッケージ名 ## クラス名文字列

【参考】これらのデバイス名及びコメントは, シェルやリスナからシェルユーティリティの **rs** を実行するか, またはリスナで実行中ゴールのタスクの資源状態を問い合わせれば, 表示することができる. シェルユーティリティ **rs** について, 詳しくは 2.2節 「シェルユーティリティプログラム」 (31頁), リスナでの, 実行中ゴールのタスクの資源状態の問い合わせについて, 詳しくは 4.1.2.7節 「実行の中断」 (70頁) 参照.

9.3.5 ウインドウ

PIMOS が提供するウインドウは, 主に以下のような機能を持っている.

- 文字の入出力
- ウインドウを生成すると SIMPOS の pmacs バッファが割り当てられる
- ウインドウの大きさは, あるフォントにおける文字数, 行数か, あるいはマウスで設定する
- ウインドウの位置は, ウインドウの左上端の位置を, スクリーン上のドット数か, あるいはマウスで設定する
- 入出力処理の途中でも, OS からの要求またはユーザからの割り込みによって処理を放棄することができる

ウインドウを利用して入出力を行うには, デバイスの階層にしたがって, まずジェネラルリクエストデバイスからウインドウリクエストデバイスを生成し, それからウインドウデバイスを生成する.

9.3.5.1 ウインドウリクエストデバイス

ウインドウリクエストデバイスは, ウインドウデバイスを生成し, 提供するデバイスである. ウインドウ生成に関する補助情報を得ることもできる.

ウインドウリクエストデバイスには, 他のデバイスと同様, アボートライン, アテンションラインが設けられている. ウインドウリクエストデバイスをアボートするとは, ウインドウ生成処理などをアボートする, 或いは, ウインドウ生成要求メッセージなどのメッセージに対する処理を行なわずに放棄することに当たる. ウインドウリクエストデバイスのアテンションラインは特に意味を持たない. 制御方法やアボート / アテンション処理関連メッセージについて, 詳しくは 9.3.3節 「デバイスのアボート / アテンション処理」 (356頁) 参照.

ウインドウリクエストデバイスに送ることのできる入出力関連メッセージは以下の通り.

create(Buffer_name, ^Result)

ウインドウデバイスを生成し, アボートライン, アテンションラインを張る. ウインドウ生成時に, テキストバッファ名 Buffer_name (文字列) で指定されるテキストバッファを割り当てる.

メッセージが正常終了した場合, Result に **normal(^Window, ^Abort, Attention)** を返す. Window は生成されたウインドウデバイスへのストリーム, Abort, Attention はそれぞれアボートライン, アテンションラインである.

指定されたバッファが存在しない場合, 新しいバッファがその名前で生成され, ウインドウに割り当てられる. 異常終了時の **Result** の値については 9.3.2節 「入出力メッセージの形式」 (355頁) 参照.

create(^Result)

ウインドウデバイスを生成し, アボートライン, アテンションラインを張る. ウインドウ生成時に新たなテキストバッファをデフォルトの名前で生成し割当てる.

メッセージが正常終了した場合, Result に **normal(^Window, ^Abort, Attention)** を返す. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

get_max_size(X, Y, Font_pathname, ^Result)

フォントをパス名 Font_pathname (文字列) で指定し, スクリーン上の座標 X, Y をドット数で指定すると, そこをウインドウの左上端と仮定した場合に生成することができるウインドウの最大の大きさを, 指定されたフォントでの文字数, 行数で返す.

メッセージが正常終了した場合, Result に **normal(Characters, Lines)** を返す. Characters, Lines はそれぞれ指定されたフォントにおける文字数, 行数である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

9.3.5.2 ウインドウ

ウインドウデバイスは, 具体的なウインドウを管理し, そこへの入出力処理を行なうデバイスである. アボートライン, アテンションラインを持っており, 入出力処理中にアボート / アテンション処理を行なうことができる. 制御方法及びアボート / アテンション処理関連メッセージ について, 詳しくは 9.3.3節「デバイスのアボート / アテンション処理」(356頁) 参照.

ウインドウデバイスに送ることのできる入出力関連メッセージは, 大きく分けて, 入出力メッセージ, ウインドウ属性設定メッセージ, ウインドウ属性問い合わせメッセージ の 3 種類があり, それぞれ以下の通りである.

入出力メッセージ

getb(Size, ^Result)

改行コードまでの文字列を入力する. ただし, 入力する最大文字列長 Size を指定しなければならない. 入力された文字列長がそれを越える場合は, 後ろを切り捨てた Size 長の文字列を返す. 切り捨てられた残りの文字列 (入力残留文字列と呼ぶ) はデバイス内で管理され, getb の実行時に既に入力残留文字列が存在する場合, ウインドウに対して入力要求を出さずに, 入力残留文字列のうち指定されたサイズ長の部分文字列を結果として返す.

メッセージが正常終了した場合, Result に **normal(String)** を返す. String は上に述べた仕様に従って返される文字列である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【注意】**アボートと入力残留文字列:** ウインドウデバイスのアボート処理の際, 入力残留文字列は中断メッセージ列と同様に取り扱われる. すなわち, ウインドウデバイスへの reset, resend, cancel メッセージにおける ID は, 中断メッセージ列と入力残留文字列の両者を指す識別番号である. 詳しくは, 9.3.3.1節「ウインドウデバイスのアボート処理」(357頁), 及び 9.3.3節「デバイスのアボート / アテンション処理」(356頁) 参照.

【参考】本来, ウインドウから 1 行入力する場合には, その行がいかに長くとも改行コードまでの文字列を返すようにしたいところである. PIMOS では, 入力用メッセージプロトコルをファイルなどと統一するために, このような形式とした.

putb(String, ^Status)

ウインドウに文字列 String を出力する. 文字列の長さに制限はない. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

prompt(String, ^Status)

ウインドウに文字列 String を出力する. 文字列の長さに制限はない. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【参考】 prompt/2 は, ユーザの入力を促すプロンプトを表示するためのものである. 本来, ウインドウへのプロンプトの表示は putb/2 を用いれば充分であるが, 入出力先デバイスの種類に依存せずプログラムを記述したい場合もあるので, 別に用意した. prompt/2 をウインドウに対して発行した場合, putb/2 と同様に動作するが, ファイルに対して発行した場合は何もしない.

ungetb(String, ^Status)

ウインドウの入力開始位置を移動させずに文字列 String を出力する. 文字列の長さに制限はない. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【参考】 ungetb/2 は, 後に行われるユーザの入力のガイドラインを表示するためのものである. ファイルに対して発行した場合, ガイドラインは全て削除されたと見なされ, 何もしない.

flush(^Status)

最新のバッファの内容をただちにウインドウに出力する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

beep(^Status)

ベルを鳴らす. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

ウインドウ属性設定メッセージ

set_size(Size, ^Status)

ウインドウの大きさを設定する.

Size が char(Characters, Lines) の形式で指定された場合, 文字数 Characters, 行数 Lines の大きさに設定する. Size が mouse と指定された場合, マウスを用いて大きさを設定する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【注意】指定するサイズは, PIMOS が生成可能なウインドウの最小サイズより大きくなくてはならない.

set_position(Position, ^Status)

スクリーン上のウインドウの位置を設定する.

Position が at(X, Y) の形式で指定された場合, ウインドウの左上端の位置をスクリーン上のドット座標 (X, Y) に設定する. Position が mouse と指定された場合, マウスを用いて位置を設定する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

reshape(Position, Size, ^Status)

スクリーン上のウインドウの位置, 大きさを設定する.

Position が at(X, Y) の形式で指定された場合, ウインドウの左上端の位置をスクリーン上のドット座標 (X, Y) に設定する. Position が mouse と指定された場合, マウスを用いて位置を設定する.

Size が char(Characters, Lines) の形式で指定された場合, 文字数 Characters, 行数 Lines の大きさに設定する. Size が mouse と指定された場合, マウスを用いて大きさを設定する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

set_title(String, ^Status)

タイトルを String (文字列) に設定する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

set_font(Pathname, ^Status)

フォントをパス名 Pathname (文字列) で設定する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【注意】指定するフォントは, その設定後もウインドウの大きさを, PIMOS が生成可能なウインドウの最小サイズより大きく保てるものでなければならない.

select_buffer(Buffer_name, ^Status)

テキストバッファを名前 Buffer_name (文字列) で設定する. 指定されたバッファが存在しない場合, 新しいバッファをその名前で生成し, ウインドウに割り当てる. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

activate(^Status)

ウインドウをスクリーン上に表示する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

deactivate(^Status)

ウインドウのスクリーンへの表示をやめる. ウインドウの画面イメージは保存されており, メッセージ activate(^Status) を送ることによりにより, 再びスクリーンに表示することができる. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

show(^Status)

ウインドウが他のウインドウによって隠されていない状態にする. ただし, ウインドウがスクリーン上に表示されていない時 (deactivate 状態) は, メッセージ activate(^Status) と同じ働きをする. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

hide(^Status)

ウインドウを他のウインドウ中の最下層に移動する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

clear(^Status)

ウインドウのテキストバッファの内容を消去する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

ウインドウ属性問い合わせメッセージ

get_char_size(^Result)

ウインドウの大きさを文字数, 行数で返す.

メッセージが正常終了した場合, Result に **normal(Characters, Lines)** を返す. Characters, Lines はそれぞれ文字数, 行数である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

get_position(^Result)

ウインドウの左上端の位置をスクリーン上のドット座標で返す.

メッセージが正常終了した場合, Result に **normal(X, Y)** を返す. X, Y はスクリーン上のドット座標 (X, Y) を表す. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

get_title(^Result)

タイトルを返す.

メッセージが正常終了した場合, Result に **normal(String)** を返す. String がタイトル (文字列) である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

get_font(^Result)

フォントをパス名で返す.

メッセージが正常終了した場合, Result に **normal(Pathname)** を返す. Pathname がフォントのパス名 (文字列) である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

get_buffer_name(^Result)

そのウインドウに指定されているテキストバッファの名前を返す.

メッセージが正常終了した場合, Result に **normal(Buffer_name)** を返す. Buffer_name がバッファ名 (文字列) である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

9.3.6 ファイルとディレクトリ

PIMOS が提供するファイルは, 主に以下のような機能を持っている.

- ファイルには 3 種類のモード **read**, **write**, **append** がある. ファイルはいずれかのモードを指定してオープンする. **read** は入力モード, **write** は新ファイルに出力するモード, **append** はもし指定したファイルがあればそのファイルに追加出力するモードである.
- 文字の入出力
- ディレクトリに対する幾つかの操作を行うことができる

ファイルに対して入出力を行うには, デバイスの階層にしたがって, まずジェネラルリクエストデバイスからファイルリクエストデバイスを生成し, それからそれぞれのモードでファイルデバイスを生成する.

9.3.6.1 ファイルリクエストデバイス

ファイルリクエストデバイスは, ファイルデバイス, ディレクトリデバイスを生成し, 提供するデバイスである. これらのデバイスを生成するにはファイルやディレクトリのパス名を指定する必要がある. ファイルリクエストデバイスには **カレントディレクトリ** という概念があり, これを用いてカレントディ

レクトリからの相対指定でパス名を表すことができる．パス名の解釈及び指定方法について，詳しくは 9.3.6.2節「パス名の解釈」(365頁) 参照.

ファイルリクエストデバイスには，他のデバイスと同様，アボートライン，アテンションラインが設けられている．ファイルリクエストデバイスをアボートするとは，ファイルのオープン処理などをアボートする，或いは，ファイルのオープン要求メッセージなどのメッセージに対する処理を行なわずに放棄することに当たる．ファイルリクエストデバイスのアテンションラインは特に意味を持たない．制御方法やアボート / アテンション処理関連メッセージについて，詳しくは 9.3.3節「デバイスのアボート / アテンション処理」(356頁) 参照.

ファイルリクエストデバイスに送ることのできる入出力関連メッセージは以下の通り.

open(Pathname, read(^Result))

パス名 Pathname (文字列) で指定されるファイルデバイスを入力モードで生成し，アボートライン，アテンションラインを張る．メッセージが正常終了した場合，Result に **normal(^File, ^Abort, Attention)** を返す．File は生成されたファイルデバイスへのストリーム，Abort, Attention はそれぞれアボートライン，アテンションラインである.

指定されたファイルが存在しない場合，Result には **abnormal** を返す．それ以外の異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

open(Pathname, write(^Result))

パス名 Pathname (文字列) で指定されるファイルデバイスを出力モードで生成し，アボートライン，アテンションラインを張る.

メッセージが正常終了した場合，Result に **normal(^File, ^Abort, Attention)** を返す.

指定されたファイルが存在しない場合，新規作成オープンする．既存のファイルの場合，同一名でバージョンアップされたファイルが作成オープンされる．異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

open(Pathname, append(^Result))

パス名 Pathname (文字列) で指定されるファイルデバイスを追加モードで生成し，アボートライン，アテンションラインを張る.

メッセージが正常終了した場合，Result に **normal(^File, ^Abort, Attention)** を返す.

指定されたファイルが存在しない場合，新規作成オープンする．既存のファイルの場合，そのファイルをオープンする．異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

directory(Pathname, ^Result)

パス名 Pathname (文字列) で指定されるディレクトリデバイスを生成し，アボートライン，アテンションラインを張る.

メッセージが正常終了した場合，Result に **normal(^Directory, ^Abort, Attention)** を返す.

指定されたディレクトリが存在しない場合，Result には **abnormal** を返す．それ以外の異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

set_current_directory(Pathname, ^Status)

パス名 Pathname (文字列) で指定されるディレクトリをカレントディレクトリとする．**Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

current_directory(^Result)

カレントディレクトリの値 (文字列) を返す.

メッセージが正常終了した場合, Result に **normal(Pathname)** を返す. Pathname がカレントディレクトリ (文字列) である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

9.3.6.2 パス名の解釈

ファイルデバイス, およびディレクトリデバイスを獲得するときには, ファイル, ディレクトリを識別するためのパス名が必要である. このときの指定には次の 2 つの方法がある.

- 絶対パス名指定
- 相対パス名指定

PIMOS で扱うディレクトリは, SIMPOS と同様に木構造のディレクトリである. 前述の絶対パス名指定は, ファイルリクエストを要求する時に指定した FEP のルートからのパス名を全て指定する方法であり, 相対パス名指定は, ファイルリクエストを要求する時に指定した FEP のあるディレクトリから, パス名を指定する方法である. 相対パス指定に使用するディレクトリのことをカレントディレクトリと呼ぶ.

ファイルデバイス, ディレクトリデバイスとカレントディレクトリの関係を次に示す.

- ファイルリクエストデバイス獲得時のカレントディレクトリは, タスクの環境変数 task:directory の値 (文字列) が使用される.
- ファイルリクエストデバイス獲得後のカレントディレクトリは, デバイス自身が持つ. カレントディレクトリの変更は, ファイルリクエストデバイス毎に可能である.

PIMOS では, 絶対パス名と相対パス名を次のように取り扱う.

- 与えられたパス名が '>' で始まるものは, 絶対パス名.
- パス名中に '::' を含むものは, 絶対パス名.
- そうでなければ, カレントディレクトリの値と, 与えられたパス名を文字列としてアペンドしたパス名. このとき, カレントディレクトリの最後が,' >' で終わっていなければ, '>' を補って, アペンドする.

9.3.6.3 ファイル

ファイルデバイスは, 具体的なファイルを管理し, そこへの入出力処理を行なうデバイスである.

ファイルデバイスには, 他のデバイスと同様, アボートライン, アテンションラインが設けられている. ファイルデバイスをアボートするとは, ファイルへの入出力処理などをアボートする, 或いは, ファイルへの入出力メッセージに対する処理を行なわずに放棄することに当たる. ファイルデバイスのアテンションラインは特に意味を持たない. 制御方法やアボート / アテンション処理関連メッセージについて, 詳しくは 9.3.3節 「デバイスのアボート / アテンション処理」(356頁) 参照.

ファイルデバイスに送ることのできる入出力関連メッセージは, モードごとに異なる. 以下に, どのモードでオープンしたファイルに対しても共通に送ることのできるメッセージと, 入力ファイルに対してのみ送ることのできるメッセージ, 出力または追加ファイルに対してのみ送ることのできるメッセージを順にあげる.

各ファイルに共通のメッセージ

pathname(^Result)

そのファイルの完全パス名を返す.

メッセージが正常終了した場合, Result に **normal(Full_pathname)** を返す. Full_pathname が完全パス名 (文字列) である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

attributes(^Result)

そのファイルの属性を返す.

メッセージが正常終了した場合, Result に **normal(Owner, UpdateUser, CDate, CTime, MDate, MTime, LinkCnt, FSize, RSize)** を返す. それぞれ, Owner は所有者 (文字列), UpdateUser は更新者 (文字列), CDate は生成日 (文字列), CTime は生成時間 (文字列), MDate は更新日 (文字列), MTime は更新時間 (文字列), LinkCnt はリンク数 (整数), FSize はファイルサイズ (整数), RSize はレコードサイズ (整数) を表す. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

入力ファイルへのメッセージ

getb(Size, ^Result)

ファイルからサイズ長 Size の文字列を入力する.

メッセージが正常終了した場合, Result に **normal(String)** を返す. String が入力された文字列である.

ファイルから取り出せるデータがサイズ長に満たない (途中でファイルの終わりになった) 場合, 有効な長さ分だけを返す. 次回の入力 に対して Result には **end_of_file** を, それ以降の入力に対しては **abnormal** を返す. それ以外の異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【注意】入力対象とするファイルは 16 ビット長で整合されている必要がある.

prompt(String, ^Status)

何もしない.

ungetb(String, ^Status)

何もしない.

end_of_file(^Result)

入力の終了を問い合わせる.

メッセージが正常終了した場合, Result に **normal(Ans)** を返す. 入力が終了していれば, Ans は yes に, そうでなければ no に具体化される. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

出力ファイル, 追加ファイルへのメッセージ

putb(String, ^Status)

ファイルに文字列 String を出力する. 文字列の長さに制限はない.

【注意】String の要素ビット長は 16 でなければならない. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

flush(^Status)

最新のバッファの内容をただちにファイルに出力する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

9.3.6.4 ディレクトリ

ディレクトリデバイスは, 指定されたディレクトリに対するディレクトリ操作を行なうデバイスである.

ディレクトリデバイスには, 他のデバイスと同様, アボートライン, アテンションラインが設けられている. ディレクトリデバイスをアボートするとは, ディレクトリ操作をアボートする, 或いは, ディレクトリ操作メッセージに対する処理を行なわずに放棄することに当たる. ディレクトリデバイスのアテンションラインは特に意味を持たない. 制御方法やアボート / アテンション処理関連メッセージについて, 詳しくは 9.3.3節「デバイスのアボート / アテンション処理」(356頁) 参照.

ディレクトリデバイスに送ることのできるディレクトリ操作メッセージは以下の通り.

ディレクトリへのメッセージ

pathname(^Result)

そのディレクトリの完全パス名を返す.

メッセージが正常終了した場合, Result に **normal(Full_pathname)** を返す. Full_pathname が完全パス名 (文字列) である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

listing(Wildcard, ^Result)

ワイルドカード Wildcard (文字列) に適合するファイルのファイル名を返す.

メッセージが正常終了した場合, Result に **normal(Filename)** を返す. Filename はファイル名のリストである.

適合するファイルがない場合, またはワイルドカード指定が正しくない場合, Result には **abnormal** を返す. それ以外の異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【注意】ワイルドカードにディレクトリ名を含めた指定をすることはできない. (以下同様.)

delete(Wildcard, ^Status)

ワイルドカード Wildcard (文字列) に適合する名前のファイルを削除する.

適合するファイルがない場合, またはワイルドカード指定が正しくない場合, Status には **abnormal** を返す. それ以外の異常終了時の **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

undelete(Wildcard, ^Status)

削除されているファイルで, 名前がワイルドカード Wildcard (文字列) に適合するものを回復する.

適合するファイルがない場合も, Status には **normal** を返す. ワイルドカード指定が正しくない場合は **abnormal** を返す. それ以外の異常終了時の **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

deleted(Wildcard, ^Result)

削除されたファイルで, ワイルドカード Wildcard (文字列) に適合するもののファイル名を返す.

メッセージが正常終了した場合, Result に **normal(Filename)** を返す. Filename はファイル名のリストである.

適合するファイルがない場合, またはワイルドカード指定が正しくない場合, Result には **abnormal** を返す. それ以外の異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

purge(Wildcard, ^Status)

ファイル名がワイルドカード Wildcard (文字列) に適合するファイルのうち, 最大バージョン番号を持つものだけを残し, それ以外のものを一括抹消する. Wildcard にバージョン番号部分の指定はできない. このメッセージが送られた時点で削除状態のファイルは抹消される.

適合するファイルがない場合も, Status には **normal** を返す. ワイルドカード指定が正しくない場合は **abnormal** を返す. それ以外の異常終了時の **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

expunge(^Status)

削除状態のファイルを一括抹消する. **Status** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

9.3.7 タイマ

PIMOS が提供するタイマは, 主に以下のような機能を持っている.

- 現在の日付と時刻を知ることができる
- 指定時刻になったら知らせるような目覚まし時計の機能がある
- 指定時間を経過したら知らせるようなストップウォッチの機能がある
- 目覚まし時計やストップウォッチは途中で処理を放棄することができる

タイマを利用するには, デバイスの階層にしたがって, まずジェネラルリクエストデバイスからタイマリクエストデバイスを生成し, それからタイマデバイスを生成する.

9.3.7.1 タイマリクエストデバイス

タイマリクエストデバイスは, タイマデバイスを生成し, 提供するデバイスである. PIM やマルチ PSI など, 本体にシステムタイマを持つ機種では, システムタイマによって実現されたタイマデバイスを獲得することができる. その他の機種では従来通り FEP 上のタイマを用いる.

タイマリクエストデバイスには, 他のデバイスと同様, アボートライン, アテンションラインが設けられている. タイマリクエストデバイスをアボートするとは, タイマデバイスの生成処理をアボートする, 或いは, タイマデバイスの生成要求メッセージに対する処理を行なわずに放棄することに当たる. タイマリクエストデバイスのアテンションラインは特に意味を持たない. 制御方法やアボート / アテンション処理関連メッセージについて, 詳しくは 9.3.3節 「デバイスのアボート / アテンション処理」(356頁) 参照.

タイマリクエストデバイスに送ることのできる入出力関連メッセージは以下の通り.

【注意】本体のシステムタイマの時間単位及び精度などの仕様は, 機種ごとに異なる. そこでユーザプログラムが機種依存となることを避けるために, PIMOS が, 時間単位の統一を行うようになっている. 時間単位の既定値はミリ秒であるが, ユーザがタイマデバイスにメッセージを送ることによって変更することができる. 詳しくは 9.3.7.2節「タイマ」 (369頁) 参照. また, 時間単位数を表す数値の単位変換ユーティリティについては 9.2.3.4節 「タイマに関するユーティリティ」(353頁) 参照.

create(^Result)

タイマデバイスを生成し, アボートライン, アテンションラインを張る.

メッセージが正常終了した場合, Result に **normal(^Timer, ^Abort, Attention)** を返す. Timer は生成されたタイマデバイスへのストリーム, Abort, Attention はそれぞれアボートライン,

アテンションラインである. システムタイマを持つマシン上で実行された場合, ジェネラルリクエストデバイスを獲得したノードにあるシステムタイマを用いる. システムタイマのないマシン上で実行された場合, ジェネラルリクエストデバイスの持つFEP(或いはI/Oノード)上のFEPタイマを用いる.

このメッセージによって生成されるタイマデバイスの時間単位の既定値はミリ秒である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

create(Node, ^Result)

Node に指定されたタイマを用いるタイマデバイスを生成し, アボートライン, アテンションラインを張る.

システムタイマを持つマシン上で実行される場合, Node に各ノード番号 (非負整数) を指定すると, 指定ノードにあるシステムタイマを用いるタイマデバイスを生成する. また, **fep** (アトム) を指定すると, ジェネラルリクエストデバイスの持つFEP(或いはI/Oノード)上のFEPタイマを用いるタイマデバイスを生成する.

システムタイマのないマシン上で実行される場合, Node には **fep** (アトム) のみが有効である.

メッセージが正常終了した場合, Result に **normal(^Timer, ^Abort, Attention)** を返す. Timer は生成されたタイマデバイスへのストリーム, Abort, Attention はそれぞれアボートライン, アテンションラインである. このメッセージによって生成されるタイマデバイスの時間単位の既定値はミリ秒である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

9.3.7.2 タイマ

タイマデバイスは, 内部に具体的な時計やストップウォッチを持ち, ユーザにタイマ機能を提供するデバイスである. 具体的な時計として本体のシステムタイマが割り当てられている場合には, PIMOS のシステムタイマ管理プロセスに要求を出すようになっている.

タイマデバイスには, 他のデバイスと同様, アボートライン, アテンションラインが設けられている. タイマデバイスをアボートするとは, タイマに仕掛けられている目覚し時計やストップウォッチをアボートする, 或いは, タイマ操作メッセージに対する処理を行なわずに放棄することに当たる. タイマデバイスのアテンションラインは特に意味を持たない. 制御方法やアボート / アテンション処理関連メッセージについて, 詳しくは 9.3.3節 「デバイスのアボート / アテンション処理」 (356頁) 参照.

タイマデバイスに送ることのできるタイマ操作メッセージは以下の通り.

get_date_time(^Result)

1901年1月1日から本日までの日数及び, 本日0時0分0秒から現在までの時間単位数を返す.

メッセージが正常終了した場合, Result に **normal(^Date_count, ^Time_count)** を返す. それぞれ, Date_count は日数を表す正整数, Time_count は時間単位数を表す正整数である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【参考 1】時間単位数とは, 設定された時間単位での時間数をいう (タイマデバイスには, デバイス毎に時間の単位を設定することができる). 例えばあるタイマデバイスの時間単位がミリ秒ならば, 時間単位数はミリ秒数であらわされた値である.

【参考 2p】得られる時間単位数を日付及び時刻に変換するユーティリティについては, プログラミング編 9.2.3節「タイマに関するユーティリティ」 (350頁) 参照.

get_count(^Result)

0時0分から現在までの時間単位数を返す.

メッセージが正常終了した場合, Result に **normal(Count)** を返す. Count は時間単位数を表す正整数である. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

on_at(Count, ^Result)

設定された時間単位数で時刻 Count (正整数) を指定することにより, その時刻になると通知する.

メッセージが正常終了した場合, Result に **normal(Now)** を返す. 指定された時刻になった時 Now が **wake_up** に具体化される. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

on_after(Count, ^Result)

設定された時間単位数で時間 Count (正整数) を指定することにより, その時間が経過すると通知する.

メッセージが正常終了した場合, Result に **normal(Now)** を返す. 指定された時間が経過した時 Now が wake_up に具体化される. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

where(^Result)

このデバイスが使用するシステムタイマのノード番号を返す.

メッセージが正常終了した場合, Result に **normal(Node)** を返す. Node はノード番号を表す非負整数である. FEP のタイマを使用している場合には **fep** (アトム) を返す. 異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

get_time_unit(^Result)

時間単位を返す.

メッセージが正常終了した場合, Result に **normal(Unit)** を返す. Unit は時間単位を表すアトムで, 以下の種類がある:

sec: 秒
msec: ミリ秒
msec16: 16 分の 1 ミリ秒

異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【参考】時間単位の種類は, 将来追加される可能性がある.

set_time_unit(Unit, ^Status)

時間単位を Unit に設定する.

Unit は時間単位を表すアトムで, 以下のものが有効である:

sec: 秒
msec: ミリ秒
msec16: 16 分の 1 ミリ秒

Status の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【参考】時間単位の種類は, 将来追加される可能性がある.

9.3.8 ストリング入出力デバイス

ストリング入出力デバイスは, ユーザが FEP 上に ESP を用いて定義したデバイスを PIMOS 上で操作するためのデバイスである. ESP で定義されているパッケージ名及びクラス名をそのデバイス名として指定してストリング入出力デバイスを生成する.

ストリング入出力デバイスには, **io(Command, ToString, ^Result)** という入出力処理用のメッセージがただひとつだけ用意されている. **Command** にコマンド名を指定し, **ToString** に処理を行うのに必要なデータを指定してメッセージを送ると, ユーザが定義した ESP プログラムの中の対応するメソッドが呼び出されるようになっている. **ToString** は, 16 ビット長文字列ならばその中に何を詰めてもよい.

メッセージに対応する処理が正常終了すると, Result に **normal(FromString)** が返される. FromString には処理の結果として得られた情報が 16 ビット長文字列の中に詰め込まれている. 何を詰め込むかは, ユーザの意図にしたがって ESP で自由に定義すればよい.

ESP でのデバイスの定義方法については, プログラミング編 C章「ストリング入出力デバイスの作成方法」(430頁) 参照.

ストリング入出力デバイスを利用するには, デバイスの階層にしたがって, まずジェネラルリクエストデバイスからストリング入出力リクエストデバイスを生成し, それからストリング入出力デバイスを生成する.

9.3.8.1 ストリング入出力リクエストデバイス

ストリング入出力リクエストデバイスは, ストリング入出力デバイスを生成してユーザに提供するデバイスである.

ストリング入出力リクエストデバイスには, 他のデバイスと同様, アボートライン, アテンションラインが設けられている. ストリング入出力リクエストデバイスをアボートするとは, ストリング入出力デバイスの生成処理をアボートする, 或いは, ストリング入出力デバイスの生成要求メッセージに対する処理を行なわずに放棄することに当たる. ストリング入出力リクエストデバイスのアテンションラインは特に意味を持たない. 制御方法やアボート / アテンション処理関連メッセージについて, 詳しくは 9.3.3節「デバイスのアボート / アテンション処理」(356頁) 参照.

ストリング入出力リクエストデバイスに送ることのできる入出力関連メッセージは以下の通り.

create(DeviceName, ^Result)

デバイス名 DeviceName (文字列) で指定されるストリング入出力デバイスを生成し, アボートライン, アテンションラインを張る.

メッセージが正常終了した場合, Result に **normal(^StringIO, ^Abort, Attention)** を返す. StringIO は生成されたストリング入出力デバイスへのストリーム, Abort, Attention はそれぞれアボートライン, アテンションラインである.

指定された ESP のデバイス (クラス) が存在しない場合, Result に **abnormal** を返す. それ以外の異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

【参考】デバイス名は通常, " SIMPOS のパッケージ名 ## クラス名" と指定する.

9.3.8.2 ストリング入出力デバイス

ストリング入出力デバイスは, ユーザが定義した ESP のデバイスへの入出力操作を行うデバイスである. アボートライン, アテンションラインを持っており, ユーザが ESP プログラム中にアボート, アテンション処理を記述しておけば, 入出力処理中にアボート / アテンション処理を行なうことができる. PIMOS 上のストリング入出力デバイスの制御方法及びアボート / アテンション処理関連メッセージ について, 詳しくは 9.3.3節「デバイスのアボート / アテンション処理」(356頁) 参照.

ストリング入出力デバイスに送ることのできる入出力関連メッセージは以下の通り.

io(Command, ToFepString, ^Result)

Command (整数) で指定される入出力命令を ToFepString (文字列) で指定されるデータをもとに実行する.

メッセージが正常終了した場合, Result に **normal(FromFepString)** を返す. FromFepString (文字列) は, 実行結果である.

指定された入出力命令が存在しない場合, または実行した結果, 正常終了しなかった場合, Result には **abnormal** を返す. それ以外の異常終了時の **Result** の値については 9.3.2節「入出力メッセージの形式」(355頁) 参照.

9.3.9 デバイスへのメッセージ一覧

表 9.2: PIMOS のデバイスへのメッセージ一覧

ジェネラルリクエストデバイスへの入出力メッセージ:

メッセージ	意味	参照頁
window(^Result)	ウインドウリクエストデバイスを生成, Result=normal(^WinReq,^Abort,Att)	334
file(^Result)	ファイルリクエストデバイスを生成, Result = normal(^FilReq, ^Abort, Att)	334
timer(^Result)	タイマリクエストデバイスを生成, Result = normal(^TimReq, ^Abort, Att)	334
string_io(^Result)	ストリング入出力リクエストデバイスを生成, Result=normal(^StrioReq,^Abort,Att)	335

ウインドウリクエストデバイス:

reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357

create(Buffername, ^Result)	ウインドウ生成, Result = normal(^Window, ^Abort, Att)	359
create(^Result)	ウインドウ生成, Result = normal(^Window, ^Abort, Att)	360
get_max_size(X,Y,Pathname,^Result)	ウインドウの最大の大きさを得る, Result=normal(Chars,Lines)	360

ウインドウデバイス:

reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356, 357
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357, 357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357, 357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357

getb(Size, ^Result)	改行コードまでの文字列を入力, Result = normal(String)	360
putb(String, ^Status)	文字列 String を出力	361
prompt(String, ^Status)	文字列 String を出力	361
ungetb(String, ^Status)	入力開始位置を移動させずに文字列 String を出力	361
flush(^Status)	最新のバッファの内容をただちにウインドウに出力	361
beep(^Status)	ベルを鳴らす	361

set_size(Size, ^Status)	ウインドウの大きさを設定	361
set_position(Position, ^Status)	ウインドウの位置を設定	361
reshape(Position, Size, ^Status)	ウインドウの位置, 大きさを設定	362
set_title(String, ^Status)	タイトルを String (文字列) に設定	362
set_font(Pathname, ^Status)	フォントをパス名 Pathname (文字列) で設定	362
select_buffer(Buffername, ^Status)	テキストバッファを名前 Buffername (文字列) で設定	362
activate(^Status)	ウインドウをスクリーン上に表示	362
deactivate(^Status)	ウインドウのスクリーンへの表示をやめる	362
show(^Status)	ウインドウが他のウインドウによって隠されていない状態にする	362
hide(^Status)	ウインドウを他のウインドウ中の最下層に移動	362
clear(^Status)	テキストバッファの内容を消去	362

get_char_size(^Result)	ウインドウの大きさを文字数, 行数で返す, Result = normal(Chars, Lines)	363
get_position(^Result)	ウインドウの左上端の位置をドット座標で返す, Result = normal(X, Y)	363
get_title(^Result)	タイトルを返す, Result = normal(String)	363
get_font(^Result)	フォントをパス名で返す, Result = normal(Pathname)	363
get_buffer_name(^Result)	テキストバッファの名前を返す, Result = normal(Buffername)	363

ファイルリクエストデバイス:

メッセージ	意味	参照頁
reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357
open(Pathname, read(^Result))	入力ファイルをオープン Result = normal(^File, ^Abort, Att)	364
open(Pathname, write(^Result))	出力ファイルをオープン Result = normal(^File, ^Abort, Att)	364
open(Pathname, append(^Result))	追加ファイルをオープン Result = normal(^File, ^Abort, Att)	364
directory(Pathname, ^Result)	ディレクトリデバイスを生成 Result = normal(^Dir, ^Abort, Att)	364
set_current_directory(Pathname,^Status)	カレントディレクトリを設定	364
current_directory(^Result)	カレントディレクトリを返す, Result = normal(Pathname)	365

ファイルデバイス:

メッセージ	意味	参照頁
reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357
pathname(^Result)	完全パス名を返す, Result = normal(Pathname)	365
attributes(^Result)	ファイルの属性を返す, Result = normal(Owner, UpdateUser, Date, Time, Date1, Time1, LinkCnt, FileSize, RecSize)	366
getb(Size, ^Result)	サイズ長 Size の文字列を入力, Result = normal(String)	366
prompt(String, ^Status)	何もしない	366
ungetb(String, ^Status)	何もしない	366
end_of_file(^Result)	入力の終了を問い合わせる, Result = normal(Ans)	366
putb(String, ^Status)	文字列 String を出力	366
flush(^Status)	バッファの内容をただちにファイルに出力	367

ディレクトリデバイス:

メッセージ	意味	参照頁
reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357
pathname(^Result)	完全パス名を返す, Result = normal(Pathname)	367
listing(Wildcard, ^Result)	ファイル名を返す, Result = normal([Pathname, ...])	367
delete(Wildcard, ^Status)	ファイルを削除	367
undelete(Wildcard, ^Status)	削除されているファイルを回復	367
deleted(Wildcard, ^Result)	削除されたファイル名を返す, Result = normal([Pathname, ...])	367
purge(Wildcard, ^Status)	最大バージョン以外のファイルを一括抹消	368
expunge(^Status)	削除状態のファイルを一括抹消	368

タイマリクエストデバイス:

メッセージ	意味	参照頁
reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357
create(^Result)	タイマを生成, Result = normal(^Timer, ^Abort, Att)	368
create(Node, ^Result)	Node にタイマを生成, Result = normal(^Timer, ^Abort, Att)	369

タイマデバイス:

reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357
get_date_time(^Result)	日数, 時間数を返す, Result = normal(Date, Time)	369
get_count(^Result)	時間数を返す, Result = normal(Time)	369
on_at(Count, ^Result)	指定時刻になると通知, Result = normal(WakeUp)	370
on_after(Count, ^Result)	指定時間を経過すると通知, Result = normal(WakeUp)	370
where(^Result)	使用するタイマのノード番号を返す, Result = normal(Node)	370
get_time_unit(^Result)	時間単位を返す, Result = normal(Unit)	370
set_time_unit(Unit, ^Status)	時間単位を設定	370

ストリング入出力リクエストデバイス:

reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357
create(DeviceName, ^Result)	ストリング入出力デバイスを生成, Result = normal(^StringIO, ^Abort, Att)	371

ストリング入出力デバイス:

reset(^Result)	アボート, アテンションラインを張る, Result = normal(^Abort, Att, ID)	356
resend(ID, ^Status)	識別番号 ID のメッセージ列を再試行	357
cancel(ID, ^Status)	識別番号 ID のメッセージ列を放棄	357
next_attention(^Result)	アテンションラインを張る, Result = normal(Att)	357
io(Command, String, ^Result)	指定入出力命令を String をもとに実行, Result = normal(FromFepString)	372

9.4 標準入出力

9.4.1 標準入出力機能とは

9.4.1.1 資源木との関係

資源木には, すべてのタスクに対して「標準入力」「標準出力」「標準インタラクション」の 3 つの入出力用ストリームを登録する仕組みが組み込まれている. この仕組みを利用して一般のユーザタスクに標準的な入出力を行うストリームをあらかじめ与えておき, それらストリームをタスク間で共有したり, あるいは使用するタスクを切り替えたりするなどのサービスを行う仕組みを標準入出力機能と呼ぶ.

資源木は, タスクに対して標準入出力を保持する機能を提供しているだけなので, タスク間でのストリームの共有やそれらの排他制御機構は別に用意する必要がある. その機能を提供しているのが標準入出力機能 (標準入出力サーバ) である.

9.4.1.2 入出力ストリームの共有と排他制御

タスク間でのストリームの共有とは, 一つの資源ストリームを同時に複数のタスクからアクセスし, 入出力動作を行うことをいう. 共有は, 親タスクが利用している標準入出力ストリームを子タスクに継承させて, 子タスクがそれを利用する場合に発生する.

もしもストリームを共有しているタスク同士が, 同時に勝手に入出力動作を行うと, そのストリームを使って入出力を行っているユーザプログラムが予期しない結果 (入出力動作) が起こる場合がある.

たとえば複数のタスクが同時に一つのウインドウにメッセージを出力した場合, ウインドウを見ているユーザには, 個々のメッセージがどのタスクの出力したものかが分からなくなる. また逆に同時に入力を要求した場合には, ウインドウから入力した結果が, どのユーザプログラムに送られるのかが分からなくなる.

このような状況を避けるためには, 共有しているストリームを流れるメッセージの交通整理を行い, 「一つの共有ストリームは, ある決められた期間の間, 同時にはたかだか一つのタスクからの入出力メッセージしか受け付けない」ように排他制御を行う必要がある. この排他制御機構つまり共有ストリームを利用するタスクの切り替えを行う機構を提供しているのが標準入出力機能である.

9.4.1.3 入出力ストリームの割り当てと切り替え

具体的に各タスクに標準入出力としてどのようなストリームを割り当てるかは, タスクを起動する際に指定する必要がある. また割り当てたストリームの排他制御, つまり切り替え動作をいつ行うかを制御する必要もある. それらの管理を行うのが, 一般ユーザプログラムの制御や管理を行うプログラム (シェルやリスナ) である. ユーザプログラムを実行するタスクを起動するに当たっては, そのタスクに標準入出力ストリームを割り当て, それらが共有されている場合には, ユーザの要求 (アテンションなど) に従って切り替えを行うことが必要となる.

9.4.1.4 入出力ストリームを使った入出力動作

シェルやリスナが起動する一般的なユーザプログラムは, あらかじめ決められた入出力用のプロトコル (標準入出力プロトコル) を使って, ウインドウやファイルなどの資源に簡単にアクセスすることができる.

9.4.2 標準入出力を利用して入出力を行う方法

一般のユーザプログラムが, そのタスクに割り当てられている標準入出力ストリームを獲得する方法には二通りある.

一つはユーザプログラムが標準入出力ストリームを介したアテンションの制御を行わない (シェルやリスナに任せる) 場合で, ごく一般的な獲得方法である.

他の一つはユーザプログラムが標準入出力ストリームを介したアテンションの制御を行う場合である. これは特殊な場合で, シェルやリスナなどはこの機能を使って子タスクの制御を行うこととなる.

標準入出力ストリームは, すべて述語 shoen:raise/3 によって獲得する. 獲得に際しては, 以下の点に注意する必要がある.

- 同じ標準入出力ストリームを複数回獲得しても構わない. ただしその場合には, それらのストリームは単にマージされているだけであり, 排他制御は行わない. それゆえマージした結果, 実際の資源に送られるメッセージの順序は保証されない.
- 標準入力用ストリーム, 標準出力用ストリーム, 標準インタラクション用ストリームのいずれかを用いる場合は標準入出力用ストリームを用いてはならない (逆も同様) . もしも行った場合には, アテンション処理時の動作は, 保証されない.
- 入出力ストリームを獲得する場合は, `with_macro pimos.` をプログラムに記述することが必要である.

9.4.2.1 アテンションの制御を行わない場合 (一般のプログラム)

ストリームの獲得

アテンションの制御を行わない方法で獲得すると, 各標準入出力ストリームには, 通信オーバヘッドを軽減するためのブロッキング／デブロッキングなどの機能を持つバッファリングフィルタ (入出力バッファ) が, 獲得時に自動的に挿入される. 従って, 入出力バッファが提供しているメッセージプロトコルを使用して資源にアクセスすることとなる.

1. 標準入力用ストリームの獲得

   shoen:raise(pimos_tag#task, get_std(in), STD_IN)

2. 標準出力用ストリームの獲得

   shoen:raise(pimos_tag#task, get_std(out), STD_OUT)

3. 標準インタラクションストリームの獲得

   shoen:raise(pimos_tag#task, get_std(inter), STD_INTER)

通信プロトコル

1. 標準入力 (STD_IN)
   `buffer:input_filter` が提供しているメッセージプロトコル
2. 標準出力 (STD_OUT)
   `buffer:output_filter` が提供しているメッセージプロトコル
3. 標準インタラクション (STD_INTER)
   `buffer:interaction_filter` が提供しているメッセージプロトコル

なおメッセージプロトコルの詳細については, 第 9.2.1 節バッファ (338頁) を参照されたい.

9.4.2.2 アテンションの制御を行う場合 (特殊なプログラム)

ストリームの獲得

`std_in`, `std_out`, `std_inter` を個別に獲得する代わりに, それらの機能を併合した `std_io` を利用せねばならない.

アテンションの制御を行う方法で獲得すると, 入出力バッファが挿入されないままの状態の標準入出力ストリームが得られる. このストリームには, `reset/1` などの特殊なメッセージを送ることができる.

1. 標準入出力用ストリームの獲得

   shoen:raise(pimos_tag#task, get_std(io), STD_IO)

ストリームを獲得した直後の状態では, アテンションの制御は親タスクに委ねられている. アテンションを実際にユーザタスク (`STD_IO` を獲得したタスク) で制御するためには, `STD_IO` に `reset/1` メッセージを送り, アテンション変数を獲得する必要がある. そうすることで, アテンションの制御 (というか処理) 権を獲得することができる.

アテンション変数を獲得した状態でアテンションが発生すると, 獲得したアテンション変数にアトム `attention` がユニファイされる. またもしもこの状態で入出力が終了する (入出力ストリームが閉じられる) とアテンション変数には `[]` がユニファイされる.

一度獲得したアテンション制御権は, 決して放棄することはできない. しかしアテンションが発生したことを親タスクに通知することは可能である. 入出力ストリームに, switch(Status) というメッセージを流すと, 親タスクにその結果が報知される. どのようにして報知されるかは, 第9.4.3.3項 (381頁) で述べる.

通信プロトコル

1. 標準入力ストリームへ送られるメッセージ

   getb(Size, ^StringWithStatus)

   prompt(String, ^Status)

   ungetb(String, ^Status)

2. 標準出力ストリームへ送られるメッセージ

   putb(String, ^Status)

   flush(^Status)

3. 標準インタラクションストリームへ送られるメッセージ

   igetb(Size, ^StringWithStatus)

   iprompt(String, ^Status)

   iungetb(String, ^Status)

   iputb(String, ^Status)

   iflush(^Status)

   なお各メッセージの詳細は, 第9.3節 (354頁) 入出力制御の説明を参照されたい.

   【注意】igetb/2, iprompt/2, iputb/2, iflush/1 の各メッセージの機能は, 送出先が標準インタラクションであること以外は getb/2, prompt/2, putb/2, flush/1 の各メッセージと同等である.

4. アボート / アテンション制御用メッセージ

   reset(^AbortAttentionAndIdWithStatus)

   ユーザが自身でアテンション処理を行うつもりで std_io を獲得した場合でも, この reset/1 メッセージを送るまでに発生したアテンションは, シェルが処理を行う. すなわちユーザは, reset/1 を送ることによって始めてアテンションの制御権を得る. その後は, reset を含む以下の 5 つのメッセージを用いて, ユーザが所望のアテンション処理を行うことができる.

   【注意】一度獲得したアテンション制御権は, 決して放棄することはできない.

   switch(^Status)

   アテンションを検出したユーザが, 自分自身のジョブを強制的にバックグラウンドに移行させるためのメッセージである.

next_attention(^AttentionWithStatus)

resend(Id, ^Status)

cancel(Id, ^Status)

なお, reset/1, next_attention/1, resend/2, cancel/2 の各メッセージの詳細は, 第9.3節 (354頁) 入出力制御の説明を参照されたい.

9.4.2.3 旧版との互換性

上記の使用方法に関しては, レイズの形式を除き, 上位互換である. レイズの形式の相違点をまとめると次のようになる.

- 標準入力用ストリームの獲得

  shoen:raise(pimos_tag#task, get_std(in), STD_IN) (新)

  shoen:raise(pimos_tag#shell, get_std_in, STD_IN) (旧)

- 標準出力用ストリームの獲得

  shoen:raise(pimos_tag#task, get_std(out), STD_OUT) (新)

  shoen:raise(pimos_tag#shell, get_std_out, STD_OUT) (旧)

- 標準インタラクションストリームの獲得

  shoen:raise(pimos_tag#task, get_std(inter), STD_INTER) (新)

  shoen:raise(pimos_tag#shell, get_std_inter, STD_INTER) (旧)

- 標準入出力ストリームの獲得

  shoen:raise(pimos_tag#task, get_std(io), STD_IO) (新)

  shoen:raise(pimos_tag#shell, get_std_io, STD_IO) (旧)

旧形式でのレイズでストリームを獲得することも可能である. ただし, その場合には, 新形式でレイズをやり直すゴールを挿入し, 再度レイズを行うので, 獲得に時間がかかる. なるべく新形式でレイズを行うことを推奨する.

9.4.3 標準入出力をタスクに割り当てそれを管理するには

タスクが実際に標準入出力を利用するためには, そのタスクの起動時にあらかじめ標準入出力を指定しておく必要がある. 従って一般のユーザプログラムをタスクとして起動するプログラム (シェルやリスナなど) は, 子タスクの起動に際して, その子タスクに標準入出力を設定しなければならない.

9.4.3.1 タスクに標準入出力を割り当てる方法

標準入出力の割り当ては, タスクを起動する時点で行う. 具体的には, `general_request` ストリームに流すタスク起動メッセージ `execute` 中で指定する. `execute` (第 11.1.9.2 節, 404 頁参照) の引き数の中に `stdio` というパラメタがあるが, これを用いて標準入出力を指定する.

stdio([{Type, RSC}, {Type, RSC}, ...], ^Result)

ここで Type は, 指定する標準入出力の種類を表し, 次のいずれかのアトムでなければならない.

in : 標準入力となる資源を指定する

out : 標準出力となる資源を指定する

inter : 標準インタラクションとなる資源を指定する

in_out : 標準入力および標準出力となる資源を指定する

in_inter : 標準入力および標準インタラクションとなる資源を指定する

out_inter : 標準出力および標準インタラクションとなる資源を指定する

in_out_inter : 標準入力および標準出力および標準インタラクションとなる資源を指定する

また RSC は, {Server, InInfo, OutInfo} なるベクタであり, Server は, PIMOS に登録されているサーバ名を表すアトム (window_request, file_request など) である. InInfo は, 資源の様々な初期化パラメタであり, 該当するサーバに送られる. サーバは標準入出力資源を設定した後, OutInfo を具体化する.

現在 PIMOS が提供している標準入出力機能では, InInfo は次のいずれかの形式で記述する.

1. window_request の場合
   - ウインドウタイトル文字列
   - { ウインドウタイトル文字列 }
   - { ウインドウタイトル文字列, At, Char }
   - { ウインドウタイトル文字列, At, Char, フォント名文字列 }

     ここで At および Char は, ウインドウの位置および大きさの指定である.

   (a) At の指定方法

   at(X, Y) : ウインドウ左上端の座標位置を指定

   mouse : マウスを使って指定

   (b) Char の指定方法

   char(X, Y) : ウインドウの大きさをキャラクタ数で指定

   mouse : マウスを使って指定

2. file_request の場合
   - ファイル名文字列
   - { ^ ファイル名文字列 }

【注意】この指定形式は将来拡張される予定である.

正常に設定できた場合, Result には normal(ID, Info) の形式の値に具体化される. ID は標準入出力の切替え機能を使う際に用いる識別子, Info は親タスクの標準入出力の継承状況についての情報である.

指定方法や返される値の詳細については, 第 11.1.9.2 節の記述 (405 頁) も参照されたい.

【例】例えば具体的には, 次のように指定する.

```
.....
shoen:raise(pimos_tag#task, general_request, GR),
GR = [execute(CODE, ArgVector, [stdio(S_PARAM, S_Result)], Result),
    S_PARAM = [in({window_request,
                   {"Test1-input",0,300,20,5,"font:test_11"}, OUT1}),
```

```
                out({window_request,
                     {"Test1-output",0,500,20,5,"font:test_11"}, OUT2}),
                inter({window_request,
                     {"Test1-inter",0,700,20,5,"font:test_11"}, OUT3})],
.....
```

execute で標準入出力を指定すると, 標準入出力に番号が与えられ, execute の引き数に返される. 上記の例で言うと, S_Result に normal(ID, Info) として返される ID である. いくつかの子タスクを同時に起動した場合に, どの子タスクに共有している資源のアクセスを許可するかは, この番号で指定する.

ただしこの番号と一緒に返される標準入出力の継承情報がすべて未定義変数を含まないグラウンド値になるまでは, この番号を使って資源の制御を行うことはできない. もしも仮に行った場合には, そのメッセージが無視されることがある.

9.4.3.2 タスクに割り当てた標準入出力を制御する方法

子タスクに与えた (共有させた) 標準入出力の制御は, 次のようにして制御用ストリームを獲得し, そこに制御メッセージを流すことで行う.

- 標準入出力制御用ストリームの獲得

  shoen:raise(pimos_tag#task, get_std(switch), STD_SW)

子タスクを生成し, 親タスクの標準入出力ストリームを子タスクに与えた (共有させた) だけでは, 子タスクは, そのストリームを使用することはできない. 子タスクがメッセージを送ることは自由であるが, 親タスクが子タスクにストリームの利用権限を与えるまで, 子タスクが送りだしたメッセージは処理されずに待たされることになる.

【注意 1】標準入出力制御用ストリームを獲得するまでは, この限りではなく, すべての子タスクは, 自由に親タスクの標準入出力ストリームを利用して入出力動作を行うことができる. つまり利用権限は制限されない. 子タスクが利用権限を制限されるのは, 親タスクが標準入出力制御用ストリームを確保した場合だけである.

【注意 2】親タスクが, 標準入出力制御用ストリームを獲得する以前に作った子タスクも親タスクが標準入出力制御用ストリームを獲得した時点で, その制御下におかれる. そのためそれ以後は, 親タスクから利用権限を改めて与えられるまで, 標準入出力の利用を制限される.

9.4.3.3 標準入出力の制御

親タスクが STD_SW を獲得したとしても, そのまま何もしなければ状況は変わらない. 子タスクにストリームを利用させるためのメッセージを送る必要がある. それが attach/2, attach/3 である.

attach メッセージを送り, メッセージの中の引き数 Status がバインドされてから, 引き数 Detect がバインドされるまでの間, attach で指定した ID を持つ子タスクに排他的に標準入出力ストリームを利用させることができる.

ある子タスクに与えた標準入出力の利用権限が返却されることは, Detect に値がユニファイされることで親タスクに報告される. 例えば引き数 Detect にアトム detected がユニファイされたとすると, それは指定した子タスクがアテンションを検出したため, 標準入出力の利用を中断し親タスクに制御を返したことを表す. また reset メッセージによって制御権を獲得した子タスクが, switch メッセージを送った場合にも同様の結果が得られる. この状態では, 標準入出力ストリームはどのタスクにもつながれていない.

アテンションなどによらず, 親タスクが detach/1 を制御用ストリームに送っても同じ結果が得られる. ただしその場合には, Detect には, アトム disconnected が返される.

一つの標準入出力ストリームが, 実際には複数の資源につながれている場合がある. 例えば親タスクの標準入力がファイルA, 標準出力がファイルB, 標準インタラクションがウインドウCだったとした時に, すべての標準入出力を子タスクに継承させたとすると, attach メッセージは, それら 3 つの資源 (正確には資源の排他制御を管理しているプロセス) に送られる. それら 3 つの資源に送った attach メッセージが, それぞれ異なる結果を返すことも有り得る. そのような場合には, ある優先順位に従ってそれらの結果が採択され, ユーザが送った attach メッセージの結果として返される. 以下に示す Status と Detect の結果に番号を付けてあるが, これがその優先順位である.

1. 接続メッセージ

   標準入出力は, 生成後この attach を行うまで, フロート状態 (どこにも繋がれていない状態) である.

   attach(ID, ^Status, ^Detect)

   ID で指定した標準入出力を enable にする. それ以外は, すべて disable となる.

   attach(^Status, ^Detect)

   最後に生成した標準入出力を enable にする. それ以外は, すべて disable となる.

   - ID = 非負整数
   - Status =
     - 1.abnormal: 切り替えに失敗した (デバイス異常)
     - 2.non_existent: (指定した) ID に対応する標準入出力が存在しない
     - 3.normal: 切り替えに成功した
   - Detect =
     - 1.error: 標準入出力のエラー
     - 2.detected: attention 検出 | switch 検出
     - 3.non_existent: (指定した) ID に対応する標準入出力が存在しない
     - 4.deserted: 標準入出力ストリーム破棄
     - 5.disconnected: detach 検出
     - 6.closed: 標準入出力ストリームのクローズ. 以降は何を送っても closed

2. 切り離しメッセージ

   detach(^Status)

   すべての標準入出力を disable にする.

   - Status = normal (| abnormal)

   detach(ID, ^Status)

   ID で指定した標準入出力が enable だったら disable にする.

   - ID = 非負整数
   - Status =
     - 1.abnormal: 切り替えに失敗した
     - 2.non_existent: 指定した端子が存在しない
     - 3.non_execute: 指定した端子は disable だった
     - 4.normal: 切り替えに成功した

9.4.4 標準メッセージ出力

シェルの管理下で起動するプログラムからは, 資源木と標準入出力機能が提供している, 標準入力ストリーム, 標準出力ストリーム, 標準インタラクションストリームの他に, 標準メッセージ出力と呼ぶ入出力ストリームを利用することができる. なお標準メッセージ出力の働きについては, 操作編第 2.1章 (12頁) シェルを参照されたい.

ストリームの獲得

標準メッセージ出力ストリームには, 通信オーバヘッドを軽減するためのブロッキング / デブロッキングなどの機能を持つバッファリングフィルタが, 獲得時に自動的に挿入される (9.2節 (338頁) 入出力ユーティリティ参照).

1. 標準メッセージ出力用ストリームの獲得

   shoen:raise(pimos_tag#shell, get_std_mes, MesStream)

通信プロトコル

1. 標準メッセージ出力 (STD_MES)
   `buffer:output_filter` が提供しているメッセージプロトコル

9.4.5 標準入出力を使ったプログラムの作成例

以下に, 標準入出力を使ったプログラムの作成について簡単な例をいくつか示す. なお標準メッセージ出力を使う場合には, シェルの管理下で起動する必要がある.

Tee プログラム

標準入力の内容を標準出力にコピーすると同時に, 標準メッセージ出力にもコピーする.

```
:- with_macro pimos.
:- module tee.
:- public go/0.

go :- true |
    shoen:raise(pimos_tag#task, get_std(in), In),
    shoen:raise(pimos_tag#task, get_std(out), Out),
    shoen:raise(pimos_tag#shell, get_std_mes, Mes),
    In = [prompt(string#"TEE>"), getl(Line) | NIn],
    loop(Line, NIn, Out, Mes).

loop(Line, In, Out, Mes) :- true |
    ( string(Line, L, _), L > 0 ->
        In = [prompt(string#"TEE>"), getl(NLine) | NIn],
        Out = [putl(Line), flush(_) | NOut],
        Mes = [putl(Line), flush(_) | NMes],
        loop(NLine, NIn, NOut, NMes) ;
    otherwise ;
      true -> In = [],
        Out = [flush(Stat1) | NOut], waitStatus(Stat1, NOut, []),
```

```
        Mes = [flush(Stat2) | NMes], waitStatus(Stat2, NMes, []) ).

waitStatus(Stat, In, Out) :- wait(Stat) | In = Out.
```

Echo プログラム

引数の項を標準出力に書き出す.

```
:- with_macro pimos.
:- module echo.
:- public go/1.

go(Arg) :- true |
    shoen:raise(pimos_tag#task, get_std(out), Out),
    Out = [putt(Arg), nl, flush(Stat) | NOut],
    waitStatus(Stat, NOut, []).

waitStatus(Stat, In, Out) :- wait(Stat) | In = Out.
```

第 10 章

プログラムの管理

PIMOS はプログラムに関する情報 (アトム, モジュール) をデータベースとして管理している. これらのデータベースへのアクセスは, 入出力機器へのアクセスと同様, PIMOS に依頼しておこなう. また, これらのデータベースへのアクセスパスは PIMOS の定義する資源として管理される.

一方, PIMOS が提供するコンパイラを始めとするモジュールの生成・管理支援プログラムについても, ユーザプログラムからの呼び出し述語が提供されている.

10.1 アトムの管理

10.1.1 アトムキャッシュデータベース

アトムとその印字名との対応関係を格納するデータベースをアクセスするユーティリティである.

PIMOS ではアトムとその印字名との対応関係は一元的に管理されているため, 基本的には常に単一のデータベースにアクセスする必要が生じる. しかし, これではアクセスの集中を招き, 効率上のボトルネックになり得る. そこで, 何回もアクセスする必要がある場合はノード (プロセッサまたはクラスタ) 毎にデータベースのキャッシュへのアクセスだけで済ませることができるようになっている.

この節内の各述語の **Status** 引数に返される値は, 特に記さない限り以下のような意味を持つ.

normal(Object): 正常終了. 取り出された情報が Object にユニファイされる.

abnormal: 異常終了.

non_existent(Info): 対応するアトムが存在しない. Info には, 何を取り出そうとしたかの情報が返される.

10.1.1.1 生成

atom_table:cache_table(AtomDev)

アトムデバイスへのストリームを獲得し, そこへのストリームを AtomDev に返す.

10.1.1.2 述語によるアクセス

一回だけのアクセスの場合のために以下の述語が用意されている.

atom_table:get_atom(AtomName, ^Status)

AtomName (ストリング) で指定されたアトム名に相当するアトムを Status の Object に返す.

atom_table:get_atom_name(Atom, ^Status)

Atom (アトム) に対応する印字名 (ストリング) を Status の Object に返す.

atom_table:intern(AtomName, ^Status)

AtomName (ストリング) で指定された印字名を持つアトムを作りそれを Status の Object に返す. もし既に登録済であれば, 対応するアトムを返す.

【注意】アクセス回数が多い場合にこれらの述語を用いると非常に効率が悪い.

10.1.2 アトムデバイス

アトムとその印字名との対応関係を格納するデータベースである. PIMOS の初期化時点で, 各ノード (プロセッサまたはクラスタ) にキャッシュプロセスを生成し, ストリーム獲得要求があったノードのキャッシュプロセスのストリームを返す.

以下, この節内のメッセージの **Status** 引数に返される値は, 特に記さない限り以下のような意味を持つ.

normal(Object): 正常終了. 取り出された情報が Object にユニファイされる.

abnormal: 異常終了.

non_existent(Info): 対応するアトムが存在しない. Info には, 何を取り出そうとしたかの情報が返される.

10.1.2.1 生成

アトムリクエストデバイスに以下のメッセージを送ることで生成される.

atom(^Status)

アトムデバイスを生成し, 成功すれば Status に **normal(AtomDev,^Abort,Attension)** を返し, 失敗すれば **abnormal** を返す. ここで, 他のデバイス同様, Abort はアボートライン, Attention はアテンションラインを示す. AtomDev は生成されたアトムデバイスへのストリームである.

10.1.2.2 プロトコル

get_atom(AtomName, ^Status)

AtomName (ストリング) で指定されたアトム名に相当するアトムを Status の Object に返す.

get_atom_name(Atom, ^Status)

Atom (アトム) に対応する印字名 (ストリング) を Status の Object に返す.

intern(AtomName, ^Status)

AtomName (ストリング) で指定された印字名を持つアトムを作りそれを Status の Object に返す. もし既に登録済であれば対応するアトムを返す.

10.2 モジュールの管理

10.2.1 モジュールキャッシュデータベース

実行可能コードの実体であるモジュールと, その名前 (アトム) の対応づけを行うデータベースにアクセスするユーティリティである.

PIMOS では, 多重モジュール名空間を提供するために, パッケージの機能がある. このため以下に示すデータベースへのアクセスの指定において, パッケージが陽に指定されない場合は, 環境変数

task:package に指定されているデフォールトパッケージが指定されたものとみなされる. PIMOS におけるパッケージの詳細については, 後述するパッケージの節を参照のこと.

パッケージの継承は, 以下に示すプロトコル中のメッセージや述語で有効である. モジュールの更新, 削除などではパッケージの継承機能は無効である.

【参考】あるユーザのパッケージ me で, pimos から継承されたモジュール (例: pimos::buffer など) は, コードを獲得する時に me::buffer と指定できるが, 削除する時は, 明示的に pimos::buffer と指定する必要が有る. (ただし, この例ではプロテクションのため, 特権ユーザ super_pimos のみが削除可能である.)

get_module/2, get_module/3, get_code/4, get_code/5 (メッセージ)

dynamic_link/3 (述語)

以下, この節内の各述語やメッセージの **Status** 引数に返される値は, 特に記さない限り以下のような意味を持つ.

normal: 正常終了.

normal(Object): 正常終了. 取り出された情報が Object にユニファイされる.

abnormal: 異常終了.

non_existent: 対応するモジュールが存在しない.

protection_violation: 保護されているモジュール (PIMOS のモジュールなど) を更新しようとした.

また, 各述語やメッセージの ModuleSpec (入力) 引数の値は, **Package::ModuleName**(ベクタ表記では, {**(::),Package,ModuleName**}) というような 3 要素ベクタ (複合項), あるいは単に ModuleName (アトム) である. ここで, Package はパッケージ名 (アトム), ModuleName はモジュール名 (アトム) である.

【注意】このモジュール名にキャッシュという単語が用いられているが, 実際にはキャッシュ機構は組み込まれていない. モジュールは一般に変更されるため一貫性を保証することが難しいからである. 将来はキャッシュが導入されることも考えられる.

10.2.1.1 生成

module_table:cache_table(Stream)

モジュールデータベースのキャッシュを作りそこへのストリームを Stream に返す.

10.2.1.2 プロトコル

set_default_package(Package, ^Status)

デフォールトパッケージを設定する. このメッセージ以降, 陽にパッケージが指定されない場合, Package で指定されたアトムがデフォールトパッケージと見なされる. Status については 387 頁参照のこと.

【注意】このメッセージ処理の段階では, 指定されたパッケージの存在確認はおこなわれない. したがって, 存在しないパッケージを指定した場合は, 実際のアクセス時点でエラーとなる.

get_default_package(^Status)

設定されているデフォールトパッケージを返す. Status については 387 頁参照のこと.

define_package(Package, Super, Protection, ^Status)

Package (アトム) を名前とするパッケージを新規に生成する. Super (アトム) は親パッケージ名で, 既存のパッケージ名でなければならない. Protection (整数) は保護モードの設定である. 親パッケージ名と新規生成するパッケージ名は同じであってはならない. Status については 387 頁参照のこと). (保護モードについては, プログラミング編 11.1 章 資源の獲得 399頁を参照のこと)

【注意】パッケージの新規生成は特権ユーザである super_pimos のみが可能である. 新規パッケージの生成方法については, 後述するパッケージの節参照のこと.

remove_package(Package, ^Status)

既存のパッケージ Package (アトム) を削除する. Status については 387 頁参照のこと.

【注意】パッケージの削除は特権ユーザである super_pimos のみが可能である.

protect_package(Package, Protection, ^Status)

既存のパッケージ Package (アトム) の保護を変更する. Protection は設定する新しい保護モード (整数) である. (保護モードについては, プログラミング編 11.1 章 資源の獲得 399頁を参照のこと) Status については 387 頁参照のこと.

get_super_package(Package, ^Status)

パッケージ Package (アトム) の親パッケージ名 (アトム) を取り出す. Status については 387 頁参照のこと.

get_package_info(Package, ^Status)

パッケージ Package (アトム) の保護情報を取り出す. (保護モードについては, プログラミング編 11.1 章 資源の獲得 399頁を参照のこと) Status の Object には {{**OwnerID, GroupID**}, **Protection**} という形式の複合項が返される. ここで, OwnerID はこのパッケージのオーナ ID (整数), GroupID はこのパッケージのグループ ID (整数), Protection はこのパッケージの保護モード (整数) である. Status については 387 頁参照のこと.

get_all_packages(^Status)

現在登録されている全てのパッケージ情報を取り出す. ここで, Status の Object には 3 要素ベクタ {Package, Status1, Status2} を要素とするリストが返される. ここで, Status1 には親パッケージの情報が, Status2 には保護に関する情報が, それぞれ返される.

Package: パッケージ名 (アトム)

Status1: get_super_package/2 の Status と同じ

Status2: get_package_info/2 の Status と同じ

get_module(Package, ModuleName, ^Status)

Package (アトム), ModuleName (アトム) で指定された名前のモジュールを Status の Object に返す. Status については 387 頁参照のこと.

get_module(ModuleSpec, ^Status)

ModuleSpec で指定されたモジュールを Status の Object に返す. Status については 387 頁, ModuleSpec については 387 頁参照のこと.

get_module_in_the_package(Package, ModuleName, ^Status)

Package (アトム), ModuleName (アトム) で指定された名前のモジュールを Status の Object に返す. Status については 387 頁参照のこと. 親パッケージの探索 (パッケージ継承) はおこなわれない.

get_module_in_the_package(ModuleSpec, ^Status)

ModuleSpec で指定されたモジュールを Status の Object に返す. 詳細は, get_module_in_the_package/3 と同じ. Status については 387 頁, ModuleSpec については 387 頁参照のこと.

get_code(Package, ModuleName, Predicate, Arity, ^Status)

パッケージ Package (アトム), モジュール名 ModuleName (アトム), 述語名 Predicate (アトム), 引数個数 Arity (ゼロ以上の整数) で指定されるコードを Status の Object に返す. モジュールは存在するがそのモジュールに対応する述語が存在しない場合は **non_existent(code(^Module, ^Predicate, ^Arity))** という形式のベクタを Status 返す. Status については 387 頁参照のこと.

【参考】述語が存在しない場合にこのようなベクタを返すのは, 返された結果をそのまま組込み述語 apply の引数に渡した場合, 例外情報中にモジュール名などの情報が渡されるようにするためである.

get_code(ModuleSpec, Predicate, Arity, ^Status)

ModuleSpec で指定されたモジュールのコードを返す. 詳細は, get_code/5 と同じ. Status については 387 頁, ModuleSpec については 387 頁参照のこと.

update_module(Package, ModuleName, Module, ^Status)

Package (アトム), ModuleName (アトム) で指定されたモジュールを新しいモジュール Module に更新する. ModuleName に相当するモジュールがない場合は, モジュールを単に追加する. Status については 387 頁参照のこと.

【注意 1】更新しようとするモジュールと同名モジュールが上位パッケージに存在する場合は, 単に指定されたパッケージに新規モジュールを登録する.

【注意 2】データベースの更新はモジュール自体の更新を意味しない. 既にデータベースから取り出されたモジュールはデータベースを更新しても元のままであるし, 既に実行中のゴールの呼び先は (後述のダイナミックリンク機構を用いて呼出しているのでない限り) 変更されることはない.

update_module(ModuleSpec, Module, ^Status)

ModuleSpec で指定されたモジュールを新しいモジュール Module に更新する. 詳細は, updated_module/4 と同じ. Status については 387 頁, ModuleSpec については 387 頁参照のこと.

remove_module(Package, ModuleName, ^Status)

Package (アトム), ModuleName (アトム) で指定されたモジュールをデータベースから抹消する. Status については 387 頁参照のこと.

【注意】データベースからの抹消はモジュール自体の抹消を意味しない. データベースからは消されても, 既にモジュールオブジェクトを持っていればそれを使い続けることができる. また, 既に実行中のゴールがこの抹消によって実行すべきコードを失うことはない.

remove_module(ModuleSpec, ^Status)

ModuleSpec で指定されたモジュールをデータベースから抹消する. 詳細は, remove_module/3 と同じ. Status については 387 頁, ModuleSpec については 387 頁参照のこと.

get_all_modules(Package, ^Status)

現在パッケージ Package (アトム) 下にある全てのモジュールを返す. Status の Object には, モジュール名アトム, モジュール, モジュールの大きさをそれぞれ要素とした 3 要素ベクタを要素とするリストが返される. Status については 387 頁参照のこと.

get_all_modules(^Status)

現在管理下にある全てのモジュールを返す. Status の Object にはパッケージ名アトムとそのパッケージにメッセージ get_all_modules/2 を送った結果を対とする 2 要素ベクタを要素とするリストが返される. Status については 387 頁参照のこと.

10.2.1.3 ダイナミックリンク

通常は呼出しのオーバヘッドを軽減するために, 実行可能コードどうしはリンク時にスタティックに結合する. このため, 後から呼出し先モジュールを更新しても既にリンク済みの呼び出し元には影響を与えない. したがって, モジュールの更新を実行に反映させるには, 呼び出し元もリンクし直す必要がある.

しかし, プログラム開発中など, 頻繁にモジュールを更新する場合には, 実行時のオーバヘッドがあってもリンクし直さなくてよいことのメリットの方が大きい場合がある. このような場合には以下の述語を用いて, 実行時に名前からモジュールを探して動的に結合する方式 (ダイナミックリンク) を用いるのが適当である.

module_table:dynamic_link(ModuleSpec, Predicate, Args)

ModuleSpec で指定されるモジュールの述語 Predicate (アトム) に引数ベクタ Args を (組込み述語 apply を用いて) 呼び出す. このモジュールの探索においてパッケージの継承は有効である. ModuleSpec については 387 頁参照のこと.

指定されたモジュールが存在しない場合は raise が発生する.

10.2.2 モジュールデバイス

実行可能コードの表現形であるコードモジュールと, その名前 (アトム) の対応づけを行うデバイスである. このデバイスは基本的に前述したモジュールキャッシュデータベースと同様の機能を提供するが, デフォールトパッケージの処理がおこなわれないことや, 将来への機能拡張から仕様変更される可能性が高いため, 一般ユーザは, 前述したモジュールキャッシュデータベースを用いるべきである.

パッケージの継承は, プロトコル **get_module/2, get_module/3, get_code/4, get_code/5** 中のメッセージで有効である. モジュールの更新, 削除などではパッケージの継承機能は無効である.

以下, この節内の各述語やメッセージの **Status** 引数に返される値は, 特に記さない限り以下のような意味を持つ.

normal: 正常終了.

normal(Object): 正常終了. 取り出された情報が Object にユニファイされる.

abnormal: 異常終了.

non_existent: 対応するモジュールが存在しない.

protection_violation: 保護されているモジュール (PIMOS のモジュールなど) を更新しようとした.

10.2.2.1 生成

モジュールリクエストデバイスに以下のメッセージを送ることで生成される.

module(^Status)

モジュールデバイスを生成する. 成功すれば Status に **normal(ModuleDev, ^Abort, Attention)** を返し, 失敗すれば **abnormal** を返す. ここで, 他のデバイス同様, Abort はアボートライン, Attention はアテンションラインを示す. ModuleDev は生成されたモジュールデバイスへのストリームである.

10.2.2.2 プロトコル

define_package(Package, Super, Protection, ^Status)

Package (アトム) を名前とするパッケージを新規に生成する. Super (アトム) は親パッケージ名で, 既存のパッケージ名でなければならない. Protection (整数) は保護モードの設定である. Status については 390 頁参照のこと. (保護モードについては, プログラミング編 11.1 章 資源の獲得 399頁を参照のこと)

【注意】パッケージの新規生成は特権ユーザである super_pimos のみが可能である. 新規パッケージの生成方法については, 後述するパッケージの節参照のこと.

remove_package(Package, ^Status)

既存のパッケージ Package (アトム) を削除する. Status については 390 頁参照のこと.

【注意】パッケージの削除は特権ユーザである super_pimos のみが可能である.

protect_package(Package, Protection, ^Status)

既存のパッケージ Package (アトム) の保護を変更する. Protection は設定する新しい保護モード (整数) である. Status については 390 頁参照のこと. (保護モードについては, プログラミング編 11.1 章 資源の獲得 410頁を参照のこと)

get_super_package(Package, ^Status)

パッケージ Package (アトム) の親パッケージ名 (アトム) を取り出す. Status については 390 頁参照のこと.

get_package_info(Package, ^Status)

パッケージ Package (アトム) の保護情報を取り出す. Status の Object には {{OwnerID, GroupID}, Protection} という形式の複合項が返される. ここで, OwnerID はこのパッケージのオーナー ID (整数), GroupID はこのパッケージのグループ ID (整数), Protection はこのパッケージの保護モード (整数) である. Status については 390 頁参照のこと. (保護モードについては, プログラミング編 11.1 章 資源の獲得 399頁を参照のこと)

get_all_packages(^Status)

現在登録されている全てのパッケージ情報を取り出す. ここで, Status の Object には 3 要素ベクタ {Package, Status1, Status2} を要素とするリストが返される. ここで, Status1 には親パッケージの情報が, Status2 には保護に関する情報が, それぞれ返される. (プログラミング編 11.1 章 資源の獲得 399頁を参照のこと)

Package: パッケージ名

Status1: get_super_package/2 の Status と同じ

Status2: get_package_info/2 の Status と同じ

get_module(Package, ModuleName, ^Status)

Package (アトム), ModuleName (アトム) で指定された名前のモジュールを Status の Object に返す. Status については 390 頁参照のこと.

get_module_in_the_package(Package, ModuleName, ^Status)

Package (アトム), ModuleName (アトム) で指定された名前のモジュールを Status の Object に返す. 親パッケージの探索 (パッケージ継承) はおこなわれない. Status については 390 頁参照のこと.

get_code(Package, ModuleName, Predicate, Arity, ^Status)

パッケージ Package (アトム), モジュール名 ModuleName (アトム), 述語名 Predicate (アトム), 引数個数 Arity (正整数) で指定されるコードを Status の Object に返す. モジュールは存在するがそのモジュールに対応する述語が存在しない場合は **non_existent(code(^Module, ^Predicate, ^Arity))** という形式のベクタを返す. Status については 390 頁参照のこと.

【参考】述語が存在しない場合にこのようなベクタを返すのは, 返された結果をそのまま組込み述語 apply の引数に渡した場合, 例外情報中にモジュール名などの情報が渡されるようにするためである.

update_module(Package, ModuleName, Module, ^Status)

Package (アトム), ModuleName (アトム) で指定されたモジュールを新しいモジュール Module に更新する. Name に相当するモジュールがない場合は, モジュールを単に追加する. Status については 390 頁参照のこと.

【注意 1】更新しようとするモジュールと同名モジュールが上位パッケージに存在する場合は, 単に指定されたパッケージに新規モジュールを登録する.

【注意 2】データベースの更新はモジュール自体の更新を意味しない. 既にデータベースから取り出されたモジュールはデータベースを更新しても元のままであるし, 既に実行中のゴールの呼び先は (ダイナミックリンク機構を用いて呼出しているのでない限り) 変更されることはない.

remove_module(Package, ModuleName, ^Status)

Package (アトム), ModuleName (アトム) で指定されたモジュールをデータベースから抹消する. Status については 390 頁参照のこと.

【注意】データベースからの抹消はモジュール自体の抹消を意味しない. データベースからは消されても, 既にモジュールオブジェクトを持っていればそれを使い続けることができる. また, 既に実行中のゴールがこの抹消によって実行すべきコードを失うことはない.

get_all_modules(Package, ^Status)

現在パッケージ Package (アトム) 下にある全てのモジュールを返す. Status の Object には, モジュール名アトム, モジュール, モジュールの大きさをそれぞれ要素とした 3 要素ベクタを要素とするリストが返される. Status については 390 頁参照のこと.

10.2.2.3 ユーティリティ

モジュールに関するユーティリティである. 用意されている述語を以下に説明する.

module:get_module_name(Module,^Name)

Module (モジュール) の名前 (ModuleName) とそのモジュールが属しているパッケージ (Package) を **Package::ModuleName**(ベクタ表記では, {**(::)**,**Package**,**ModuleName**}) という形式の複合項にして返す.

module:get_module_name(Module,^Package,^ModuleName)

Module (モジュール) の名前 (ModuleName) とそのモジュールが属しているパッケージ (Package) を返す.

module:get_attributes(Module,^Status)

Module (モジュール) に設定されているタイムスタンプ情報を取り出して引数 Status に返す. Status には, タイムスタンプ情報が設定されている場合は, **normal({CDT, LDT, SourceFile, SDT, Depends})** という複合項が返される. ここで, CDT はモジュールの生成日時, LDT はリンク日時, SourceFile はソースファイル名 (ストリング), SDT はソースファイルの更新日時, Depends はこのモジュールの生成が依存するファイル名ストリングとそのファイルの更新日時を対とする 2 要素ベクタのリストである. また, 日時とは, 日数とミリ秒単位の時間 (整数) を対とする 2 要素ベクタである. (日時については, プログラミング編 9.2 章 入出力ユーティリティ 338頁を参照のこと) タイムスタンプが設定されていない場合には, non_existent が Status に返される.

module:get_predicate_info(Info,Package,ModuleName,Predicate,Arity,SoftInfo)

例外でファームウェアから報告される情報 (Info) から, 例外が発生した場所のパッケージ名アトム (Package), モジュール名アトム (ModuleName), 述語名アトム (Predicate), 引数個数 (Arity), そしてコンパイラが埋め込んだ情報 (SoftInfo) を返す.

【注意】現在のセルフコンパイラでは, 上記の SoftInfo に対応する部分には整数ゼロしか埋め込んでいない.

10.2.3 パッケージ

PIMOS におけるパッケージシステムとはモジュール名の多重空間を提供する機能のことであり, パッケージはユーザにおけるモジュール名空間を限定する環境である. PIMOS におけるパッケージは以下に示す目的で用いられている.

- 個々のユーザにおけるモジュール名空間を分離する
- 多重プログラミング環境を提供し, プログラム開発を効率化する

10.2.3.1 継承

PIMOS のパッケージシステムは, 単一継承機能を持つため, 多くのパッケージは完全に独立でなく親パッケージのモジュール群を継承している. この継承は, 指定されたパッケージに指定されたモジュールが存在しなかった場合に, 順次親パッケージを探索することで実現されている. このためモジュール参照におけるパッケージの指定とは, モジュール探索における探索の開始点を指定していることになる. このパッケージの継承により, 同じ名前のモジュールが異なるパッケージに存在する場合は, 指定されたパッケージにより近いモジュールが参照される.

10.2.3.2 組込パッケージ

以下に PIMOS 立ち上げ時から既に存在するパッケージを説明する.

root パッケージ継承におけるルートパッケージである. このパッケージは便宜的あるいは形式的なパッケージであり, このパッケージに含まれるモジュールは常に空である. このパッケージにモジュールを登録することはできない.

pimos PIMOS を構成するモジュールが含まれるパッケージである. 正確には IPL でロードされるモジュールがここに入る. root パッケージを親パッケージとする.

user 個々のユーザパッケージ (ユーザ名と同じ名前のパッケージ) の直接の親に当たるパッケージである. pimos パッケージを親パッケージとする.

work 環境変数 task:package の値が不正であったり存在しない場合は, この work パッケージがデフォールトパッケージとなる. pimos パッケージを親パッケージとする.

privilege 特権ユーザである super_pimos のみが参照可能なパッケージである. 特別に保護される必要のあるモジュールのみこのパッケージに登録される. 一般ユーザはこのパッケージをアクセスすることはできない. pimos パッケージを親パッケージとする.

10.2.3.3 ユーティリティ

現在パッケージに関連するユーティリティモジュールには, 以下に示す公開述語が用意されている.

package:define(Package, Super)

Super (アトム) で指定されたパッケージを親とし, Package (アトム) をパッケージ名とするパッケージを生成する. 保護モードには, 711(8 進数) が設定される.

モジュールの生成に失敗したならば, シェルの標準 (メッセージ) 出力にメッセージが出力される.

package:define(Package, Super, Protect)

Super (アトム) で指定されたパッケージを親とし, Package (アトム) をパッケージ名とするパッケージを生成する. Protect はこの新規パッケージの保護モード (整数) である. (保護モードについては, プログラミング編 11.1 章 資源の獲得 399頁を参照のこと)

モジュールの生成に失敗したならば, シェルの標準 (メッセージ) 出力にメッセージが出力される.

package:define(Package, Super, Protect, Status)

Super (アトム) で指定されたパッケージを親とし, Package (アトム) をパッケージ名とするパッケージを生成する. Protect はこの新規パッケージの保護モード (整数) である. (保護モードについては, プログラミング編 11.1 章 資源の獲得 410頁を参照のこと)

引数 **Status** には, 処理が正常に終了したならばアトム **normal**, そうでなければアトム **abnormal** が返される.

10.3 ライブラリ

以下に, コンパイラ, 再リンカ, ローダ, アンローダのプログラムからの呼び出し述語について説明する. (操作編 3 章 コンパイラ, 再リンカ, ローダ, アンローダ 38頁も参照のこと)

以下の説明において, モジュールの指定とは, モジュール名アトムまたは **Package::ModuleName** (ベクタ表記では, {**(::), Package, ModuleName**}) という複合項を指す. また, 引数 STDOUT とはシェルの標準出力へのストリーム, STDMES とはシェルの標準メッセージ出力へのストリームである. これらのストリームは, 必ず対応するストリームに接続すること. そうでなければデッドロックする.

また, 引数 Status には, 処理が正常に終了したならばアトム normal, そうでなければアトム abnormal が返される.

10.3.1 コンパイラの呼び出し

コンパイラをプログラムから呼び出すために以下の述語が用意されている.

compile:compile(FileNames,^STDOUT,^STDMES,^Status)

FileNames (ファイル名ストリングのリスト) で指定されたファイルをコンパイルする. コンパイルの表示メッセージは, シェルの標準出力 (STDOUT), 標準メッセージ出力 (STDMES) にそれぞれ出力される.

この述語呼び出しにおいて, コンパイルパラメタファイルである “compile.param” の内容あるいは環境変数の値が各種コンパイルパラメタの設定に用いられる.

コンパイルが正常に終了したならば, **Status** 変数に **normal** が, そうでなければ **abnormal** が返される.

10.3.2 再リンカの呼び出し

再リンカをプログラムから呼び出すために以下の述語が用意されている.

また, これらの再リンクで生成されたモジュールは全てデフォールトパッケージ (環境変数 task:package の値) に登録される. しかし, 環境変数 compile:original_package の値がアトム on であったなら, 再リンクの対象となったモジュールが存在するパッケージと同じパッケージに登録される.

relink:do(ModuleNames,^STDOUT,^STDMES,^Status)

ModuleNames (モジュール指定のリスト) で指定されたモジュールを再リンクする. 再リンクが正常に終了したならば, **Status** 変数に **normal** が, そうでなければ **abnormal** が返される.

relink:do(TopModName,GoalModNames,^STDOUT,^STDMES,^Status)

TopModName で指定されたモジュールをトップモジュールとし, GoalModuleNames で指定されたモジュールに至る参照関係に関与する全てのモジュールを再リンクする. このモジュールの探索において, 環境変数 compile:working_packages で指定されたパッケージのみを探索範囲とする. この環境変数の値がアトム ('*') であった場合は, 全てのパッケージが探索範囲となる. 再リンクが正常に終了したならば, **Status** 変数に **normal** が, そうでなければ **abnormal** が返される.

relink:package(Package,^STDOUT,^STDMES,^Status)

Package (アトム) で指定されたパッケージに含まれる全てのモジュールを再リンクする. 再リンクが正常に終了したならば, **Status** 変数に **normal** が, そうでなければ **abnormal** が返される.

10.3.3 ローダの呼び出し

ローダをプログラムから呼び出すために以下の述語が用意されている.

ローダで生成されたモジュールは全てデフォールトパッケージ (環境変数 task:package の値) に登録される. しかし, 環境変数 compile:original_package の値がアトム on であったなら, アンロード時にモジュールが存在したパッケージと同じパッケージに登録される.

load:do(FileNames,^STDOUT,^STDMES,^Status)

FileNames (ファイル名ストリングのリスト) で指定されたファイルを全てロードする．ロードが正常に終了したならば, **Status** 変数に **normal** が, そうでなければ **abnormal** が返される．

環境変数 **compile:distribution** の設定によりファイル単位での負荷分散が指定できる．また, ファイルの探索は, 環境変数 **compile:path** の設定に従う．

10.3.4 アンローダの呼び出し

アンローダをプログラムから呼び出すために以下の述語が用意されている．

unload:do(ModuleNames,FileName,^STDOUT,^STDMES,^Status)

ModuleNames (モジュール指定のリスト) で指定された全てのモジュールを FileName (ファイル名ストリング) で指定されたファイルにアンロードする．アンロードが正常に終了したならば, **Status** 変数に **normal(FullPathName)** が, そうでなければ **abnormal** が返される．FullPathName は, アンロードファイルのフルパス名 (ストリング) である．

ファイルの拡張子は “sav” がデフォールトの設定になっている．

10.4 プログラミング例

【例】指定されたファイルをコンパイル, 再リンク, アンロードする

```
cru(FileNames,Status) :- list(FileNames) |
    null_device(Null0),       % ダミーの標準出力デバイス
    merge({Null1,Null2,Null3},Null0),
    compile:compile(FileNames,Null1,Null2,CompStatus),  % コンパイル
    module_table:cache_table(ModTbl0),
    ModTbl0 = [get_default_package(PackageStatus)|ModTbl1],
    relink(CompStatus,PackageStatus,ModTbl1,Null3,Status).
otherwise.
cru(FileNameList,Status) :- true | Status = abnormal.

relink(normal,normal(Package),ModTbl0,Null0,Status) :- true |
    Null0 = {Null1,Null2,Null3},
    relink:package(Package,Null1,Null2,RelinkStatus),
    % デフォールトパッケージ内にある全てのパッケージを再リンク
    get_all_modules(RelinkStatus,Package,ModTbl0,Null3,Status).
otherwise.
relink(_,_,ModTbl0,Null0,Status) :- true |
    ModTbl0 = [],
    Null0 = [],
    Status = abnormal.

% 必ず再リンクの終了を確認してからモジュールを取り出していることに注意！
get_all_modules(normal,Package,ModTbl0,Null0,Status) :- true |
    ModTbl0 = [get_all_modules(Package,AllModStatus)],
    convert(AllModStatus,Null0,Status).
otherwise.
get_all_modules(_,Package,ModTbl0,Null0,Status) :- true |
```

```
        ModTbl0 = [],
        Null0 = [],
        Status = abnormal.

    convert(normal(AllMods),Null0,Status) :- list(AllMods) |
        Null0 = {Null1,Null2},
        convert_0(AllMods,Modules),
        unload:do(Modules,"allmodules",Null1,Null2,Status).
        % カレントディレクトリの allmodules.sav というファイルにアンロード
    otherwise.
    convert(_,Null0,Status) :- true |
        Null0 = [],
        Status = abnormal.

    % モジュール名のリストに変換する
    convert_0([{Name,Module,Size}|Rest],Modules0) :- true |
        Modules0 = [Name|Modules1],
        convert_0(Rest,Modules1).
    convert_0([],Modules0) :- true | Modules0 = [].

    % 出力なので, flush/1 メッセージのみ注目すれば OK
    null_device([flush(Done)|Next]) :- true |
        Done = normal,
        null_device(Next).
    null_device([]) :- true | true.
    otherwise.
    null_device([DontCareMessage|Next]) :- true |
        null_device(Next).
```

第 11 章

資源管理・実行管理

PIMOS が提供するサービスの主体を OS 定義資源と呼ぶ (これに対比して, KL1 の実行自体に必要な計算時間やメモリ領域などの資源を言語定義資源と呼ぶ). PIMOS は 2 種類の OS 定義資源をもつ. ひとつは PIMOS 全体のベースとなる資源であり、タスクと呼ばれるプログラムの実行主体およびジェネラルリクエストデバイスと呼ばれるユーザサービスの主体, そしてタスクに必ず 1 つ存在する標準入出力と呼ばれる資源である. もうひとつはサーバと呼ばれるプログラムにより動的に定義された資源であり, 入出力やその他のユーザサービスを行う OS 定義資源である. 後者の資源を提供するサーバプログラムとして, ウインドウやファイルなどの OS のサービスに相当する機能をもつサーバ (システム定義サーバと呼ぶ) とユーザが自由に定義したユーザ定義サーバの 2 種類のサーバが存在する.

本章では, 以下の項目に関するプログラム記述方法について説明する.

11.1 資源の獲得と管理

ジェネラルリクエストデバイスを中心とした資源の全般と資源の獲得方法 (タスクの起動方法も含む)

11.2 タスク

タスクの制御方法

11.3 サーバの定義

サーバの定義方法 (標準入出力サーバも含む)

11.4 ユーザの管理

ユーザ管理資源の利用法

なお, 標準入出力資源は 9.4 節 (375 頁), 入出力資源は 9.1 節 (333 頁) に詳細が述べられている.

11.1 資源の獲得と管理

OS 定義資源の提供するサービスの利用は, ユーザが獲得した各資源へのリクエストストリームにメッセージを送ることによって行う. この節ではリクエストストリームの獲得方法を述べる.

11.1.1 種類

OS 定義資源の種類は以下の通りである.

ジェネラルリクエストデバイス: タスクやその他の資源利用ためにデバイスを作成するメッセージを処理し, 資源のリクエストストリームを提供する. また PIMOS への問い合わせなども本デバイスを利用する.

タスク: KL1 プログラムの実行管理単位であると同時に OS 定義資源の管理単位でもある. PIMOS ではタスクを OS 定義資源として扱う. タスクは OS 定義資源として実行の中止や言語定義資源の消費状況等, タスクに対するメッセージの処理を行なう.

標準入出力資源: タスクに対して設定された標準入出力を行うための資源である. 一般の資源を仮想化した資源であり, タスクを生成する親タスクにより設定させる. ジェネラルリクエストデバイスから獲得する.

システム定義サーバにより提供される資源: PIMOS がもつ資源サービス表 (11.1.2 節 (400 頁) 参照) に登録されたユーザサービスを行う資源である. PIMOS が, ユーザプログラムのためにあらかじめ登録した資源は次の通りである.

- window_request
- file_request
- timer_request
- string_io_request

  【参考】詳細は, 第 9 章 (332 頁) を参照せよ.
- atom_request
- module_request

  【参考】詳細は, 第 10 章 (385 頁) を参照せよ.
- user_request

  【参考】詳細は, 11.4 節 (424 頁) を参照せよ.

上述の資源からさらに獲られる資源は次の通りである.

- window
- file
- directory
- timer
- string_io
- atom
- module
- user

- group
- login

サーバの資源サービス表への登録および利用は, ジェネラルリクエストデバイスを通して行われる.

11.1.2 資源サービス表

PIMOS では資源サービス表と呼ばれる管理表を用意し, 資源の利用サービスを行う. ユーザプログラム (クライアントと呼ぶ) は, 本表にある名前アトムを指定することでサーバが提供する資源への通信路 (多くの場合ストリーム) を獲得する.

資源サービス表には, 各サーバ単位に次の要素が登録されている.

名前アトム: 登録名

保護情報: オーナ, グループオーナ および利用権に関する情報

制御ストリーム: 資源サービス表とサーバタスク間の通信路

PIMOS はクライアントから資源獲得要求があった時,

1. 指定された名前に対応した資源のクライアント側のタスクに対する利用権を確認
2. 制御ストリームを通じてサーバタスクとクライアントとの通信路を獲得
3. クライアント側へのサーバとの通信路の提供

の順で処理を行う.

11.1.3 I/O ノード

PIMOS では FEP や FEP を介してネットワークで接続されたリモートサイトにおけるマシン上の資源を利用することができる. ジェネラルリクエストデバイスを通してユーザが利用できる装置 (例えば計算機) を総称して I/O ノードと呼ぶ. ネットワークが存在しない FEP を利用するユーザは, FEP が I/O ノードとなってその上の資源を利用することになる.

I/O ノード上の装置を利用する場合には, 必ず FEP もしくはリモートサイトの OS 上のログインユーザ名およびパスワードを必要とする.

11.1.4 タスク

タスクは荘園と同様の KL1 プログラムの実行管理単位であると同時に, OS 定義資源の管理単位である.

タスクにはユーザ ID が設定されている. 資源の保護にはこの情報が利用される.

タスクは生成されると必ずひとつの標準入出力資源と I/O ノードの既定値をもつ. 通常ユーザはこのリクエストストリームを OS から提供を受けて, 各装置を利用する. なおこのリクエストストリームが接続している I/O ノードの名称は環境変数 (11.1.7 節 (401 頁) を参照) task:io_node に設定されている.

タスクは KL1 の荘園機構を用いて実現されており, 荘園に対して可能な制御や状態の監視は, ほぼ同様にタスクに対しても行うことができる. タスクと単なる荘園の差異は, タスク中で獲得した資源はタスクの終了時に自動的に解放されるのに対し, 荘園に関してはこのような機構が存在しないこと, タスクは資源として管理されているので PIMOS を通じてすべてのタスクについての実行状況などの問合せができるのに対して, 荘園に関しては PIMOS は感知しないこと, などである.

また PIMOS には荘園機構を利用してプロファイル情報 (7.5 節 (290 頁) を参照) を生成する特別なタスクも用意している. 詳しくは 11.1.9.2 節 (403 頁) を, タスクに関しての詳細は 11.2 節 (410 頁) をそれぞれ参照されたい.

11.1.5 標準入出力資源

プログラムを記述する上で既定の入出力があると便利である．PIMOS にはタスク単位で標準入出力資源と呼ばれる抽象的な入出力装置が用意されている．標準入出力資源は, 親タスクによるタスク起動時, タスク起動のパラメータとして設定される．自分で設定を変更することはできない．

標準入出力資源は, 実際には資源サービス表に登録されている資源をアクセスするフィルタ的なプロセスであり, 標準入出力資源の設定時には標準入出力資源とすべき資源のサーバ情報を指定することとなる．

タスクが子タスクを生成する時, 標準入出力資源は子タスクに継承される．詳しくは 9.4 節標準入出力 (375頁) を参照されよ．

なお, 資源木 (次節) 最上位のタスク (システム定義サーバのタスク) は, 疑似的な標準入出力資源しかもたない．

11.1.6 資源木

PIMOS ではタスクを含みすべての OS 定義資源はひとつの木構造の下に管理されている．

あるタスクで OS 定義資源を獲得すると, 獲得した資源はそのタスクの子資源となる．タスクを生成するとそれも子資源になるので, 一般に資源は木構造をなすことになる．

ひとつのタスクの子資源はすべて異なる識別子 (子資源識別子, 正整数) を与えられる．資源が解放されても, ひとつのタスクの子資源に同じ識別子を再び用いることはない．したがって, すべての資源は資源木の根から始めて, 各レベルでの子資源識別子を並べたリスト (絶対資源識別子) で一意に指定することができる．

特定の資源に関する問合せなどを行う場合, 自分のタスクからの相対的な位置で指定するのが便利なことが多い．そこで, 通常以下のようなリスト形式による相対的な資源指定 (相対資源識別子) を用いることがある．

[]: 自分自身.

[N | 相対資源識別子]: N で指定される資源から, 再帰的に相対資源識別子で指定されるような資源．ここで N の意味は以下の通り．

0: 資源木の根. すなわち, トップレベルのタスク.

n ($n \geq 1$): 識別子 n を持つ直接の子資源.

-1: 自分自身.

$-n$ ($n \geq 1$): $n-1$ 段先祖の資源.

*** :** 直接の子資源すべて.

*** :** 自分の子孫資源すべて. すなわち, `[]`, `[*]`, `[*,*]`, `[*,*,*]`, ... などを合わせたもの.

11.1.7 環境変数

PIMOS はタスク単位に環境変数プールをもつ．環境変数プールとはプログラムを実行するための情報参照表のようなものである．

利用者は任意の時点で任意の環境変数 (いわば key のようなもの) に対し任意のグラウンド値 (未定義変数を含まない値) を設定 / 参照できる．タスク生成時点での環境変数値は, その時点の親タスクの環境変数値がそのまま受け継がれる．その後の環境変数値は両者独立に変更される．なお, 全タスクはデフォルトで環境変数 task:io_node をもつ．

PIMOS の提供するさまざまなユーティリティは, それぞれ対応する環境変数の値にしたがってその振る舞いを変える．これを用いて, 環境変数の設定の仕方によって, タスクごとに呼出すユーティリティの必要な振る舞いを引き出すことができる．

【参考】本マニュアルの索引に PIMOS が使用する環境変数の一覧がある．

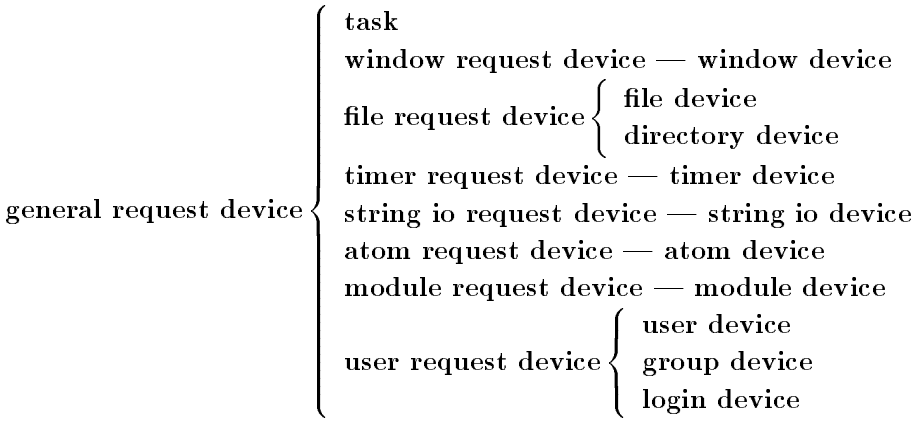

図 11.1: システム定義サーバが提供する資源の階層

11.1.8 システム定義サーバが提供する資源の獲得階層

システム定義サーバが提供する資源を階層からみた資源の獲得階層で説明する.

ジェネラルリクエストデバイスの獲得から始めて階層的に獲得していく必要がある. 例えばウインドウの場合, まずジェネラルリクエストデバイスにメッセージを送ることによってウインドウリクエストサーバへのストリームを獲得し, 次いでそこにメッセージを送ることによって具体的なウインドウへのストリームを獲得する (このとき実際にウインドウを生成する). そしてこのウインドウへのストリームにメッセージを送って具体的なウインドウ入出力を行うことになる.

システム定義サーバが提供する資源の階層を図 11.1 に示す.

ジェネラルリクエストデバイスへのストリーム獲得方法と, そこから階層的に行う資源獲得のためのメッセージプロトコルについては, 本章および関連する各章に個別に述べる.

11.1.9 ジェネラルリクエストデバイス

11.1.9.1 獲得法

荘園の例外生起用述語 raise を利用する.

shoen:raise(pimos_tag#task, general_request, GeneralRequestDevice)

タスクがもつジェネラルリクエストデバイスへのストリームを GeneralRequestDevice に獲得する.

【注意】この述語呼び出しを記述する module の先頭にて次の宣言を行なう必要がある.

```
:- with_macro pimos.
```

shoen:raise(pimos_tag#task, general_request(IONode, User, Password), GeneralRequestDevice)

PIMOS の装置が用意された I/O ノード IONode を既定値としたジェネラルリクエストデバイスへのストリームを IONode 上の OS に対するユーザ名およびパスワードで GeneralRequestDevice に獲得する. パスワードを間違えたときには, このストリームに送ったすべてのメッセージが abnormal となる. なお接続の確認には後述するメッセージ connnection を利用するとよい.

IONode: 接続するネットワークサイトの I/O ノード名ストリングである. もし Fep に接続する場合には Fep 番号である.

User: I/O ノード の OS におけるユーザ名ストリングである.

Password: ユーザ名に対応したパスワードストリングである.

shoen:raise(pimos_tag#task, general_request(Fep, IONode, User, Password), GeneralRequestDevice)

指定された Fep を通して I/O ノードへ接続する以外は同様である. IONode が Fep 番号であった場合にはこの指定は無効である.

Fep: I/O ノードに接続する Fep 番号である.

11.1.9.2 プロトコル

request(Name, Cancel, ^Result)

資源サービス表 (400 頁参照) に登録されているアトム Name で指定した資源を獲得する. 獲得できた場合には **Result** に normal(ServerVar) が返る. ServerVar のプロトコルについては各サーバの規定に従う. もし指定されないか, 保護モードで利用できなかった場合には, Result に abnormal が返る. 獲得処理中にサーバ側タスクが終了してしまった場合には, aborted が返る. また要求処理中にこの処理をキャンセルする場合には, Cancel に任意の値をユニファイする. すると Result に aborted が返る.

request_anyway(Name, Cancel, ^Result)

上述の `request(Name, Cancel, ^Result)` の機能に加えて, 指定された名前のサーバの未登録時に登録されるまで待つ機能をもつ。

window(^Result)

file(^Result)

timer(^Result)

string_io(^Result)

atom(^Result)

module(^Result)

user(^Result)

PIMOS が用意したシステム定義サーバへのストリームを獲得する. `request_anyway(Name, Cancel, ^Result)` のバリエーションである. 各種リクエストデバイスの獲得を行う. Result には獲得に成功すれば **normal(^Request, ^Abort, Attention)**, 失敗すれば **abnormal** が返る. Request がリクエストデバイスへのストリームである. なお, Abort, Attention については入出力制御 (第 9 章 (332 頁)) を参照のされたい.

register(Name, ProtectionMode, ^Result)

資源サービス表に新たに資源を登録する (サービスの登録とも呼ぶ). 詳細は 11.3.3.1 節 (418 頁) を参照されたい.

【注意】本コマンドは特権ユーザのみ許可される.

reset(Name, ^Result)

Name で指定されたサービスを受け持つサーバに attention 情報を送る. 結果 **Result** は次の通りである.

normal(^NewAttention): attention 情報が送られた.

abnormal: attention 情報の送出が拒否された.

aborted: 対象となるサーバタスクが終了処理にある.

【注意】本コマンドはサービス享受可能なそれぞれのユーザにのみ許される.

execute(Code, Argv, Param, ^Result)

タスクの生成を行う. 引数は以下の通りで, 多くの引数の意味は荘園の生成時に与えるもの (290 頁を参照) と同様である.

Code: タスクのトップレベルゴールの述語のコード.

Argv: トップレベルゴールの引数ベクタ.

Param: 様々なタスク生成に関するパラメータをもつリストである. リストの要素は次の通りである.

resource(HI, LOW): 初期資源量である. 省略値は 0 である.

ionode(IONode): I/O ノード名もしくは FEP 番号である. 省略値は親タスクと同値である.

remote(RUser, RPass): I/O ノード名に対応する I/O ノードのユーザ名, パスワードである. 省略値は親タスクと同値である.

name(Name): タスク名である. 省略値は空文字列である.

prio(Min, Max): 優先度の範囲である. 省略値は最小値が 0, 最大値が 4096 である.

tag(Tag): 例外処理用のタグである. 省略値は pimos_tag#task である.

【注意】ユーザの指定よらず, 荘園のタグにはシステム呼び出し用のタグビット (pimos_tag#task) が必ず立つ.

node(Start, Nodes): 開始ノードとノード数である. 省略値は親タスクと同様である.

uid(Uid): タスクのユーザ ID を Uid とする. スーパユーザのみ指定が可能である.

user(UName, Pass): タスクのユーザ名およびそのパスワードを与える. もし正しいのならば対応するユーザ ID がタスクに設定される. パスワードが違った場合にはタスク生成は失敗する.

stdio(STDIO, ^STDResult): 標準入出力を指定する. パラメータの意味は次の通りである.

STDIO: 新たに標準入出力として生成するデバイスを指定するためのパラメータリストである. ここで指定しない標準入出力は親 (タスク起動者) の標準入出力を継承する. もしすべての標準入出力を継承する場合には空リスト ([]) となる. `{Type,RSC}` を要素とする.

Type: 生成する標準入出力の種類である

in: 標準入力として

out: 標準出力として

inter: 標準インタラクションとして

in_out: 標準入力、標準出力として

in_inter: 標準入力、標準インタラクションとして

out_inter: 標準出力、標準インタラクションとして

in_out_inter: すべての標準入出力として

RSC: 対象となるサーバとそのパラメータ `{Server,InInfo,^OutInfo}` である.

Server: サーバ名アトム

InInfo: 標準入出力生成のためのパラメータ, 未定義変数を含まないグラウンド値である (ファイル名など).

OutInfo: サーバからの結果である. 未定義変数を含まないグラウンド値である. ここに値が設定されれば対応する標準入出力資源の設定が終了したこととなる.

【注意】**Server** に対応するサーバが未登録の場合には, **OutInfo** に [] が返る. この場合空の標準入出力資源が設定され, ユーザが利用すると返答が返らないためデッドロックが起こる.

STDResult: 以下の通りである.

normal(ID,Info): 正常終了である.

ID は親の標準入出力に対するこのタスクの標準入出力の識別子であり, 非負整数である. すべての標準入出力を新規に生成した場合は無意味である. なお, 親タスクは自分の標準入出力のスイッチを獲得して, 本 **ID** を指定することによって子タスクの標準入出力が利用可能状態となる.

パラメータ **Info** は, 親の標準入出力を継承したことを確認するための情報リストである。次の要素のリストである. もし, 親の標準入出力を一つも継承しなかった場合は [] となる. どのような要素のリストににになるかは, 親の標準入出力の状況による. なお, **STDInfo** は未定義変数を含まないグラウンド値である.

in(^STDInfo)

out(^STDInfo)

inter(^STDInfo)

in_out(^STDInfo)
in_inter(^STDInfo)
out_inter(^STDInfo)
in_out_inter(^STDInfo)

【注意】**Info** が返って来たタイミングが標準入出力が設定された時である. よって **Info** を待たずに switch を本タスクの標準入出力に切替えることができるとは限らない.

abnormal:
aborted: 親タスクが終了するタイミングで本コマンドを送った場合や、重複したパラメータ指定などを行なった場合に起こる.

Result: タスク生成状態である. 以下のいずれかが返される.

normal(^Control, Report): 正常にタスクが生成できた. 引数の Control はタスクの制御ストリーム, Report はタスクの報告ストリームである.

abnormal: タスク生成に失敗した.

execute(Code, Argv, Name, MinPrio, MaxPrio, Tag, ^Result)

タスクの生成を行う. 引数は以下の通りで, 多くの引数は荘園の生成時に与えるものと同様である.

Code: タスクのトップレベルゴールの述語のコード.

Argv: トップレベルゴールの引数ベクタ.

Name: タスクの名称文字列. 状態問合せなどの際に返されるのでコメントに用いると良い.

MinPrio: タスクの荘園としての最小優先度.

MaxPrio: タスクの荘園としての最大優先度.

Tag: 例外用タグ (整数値, 7.5.6 節, 301 頁参照).

Result: タスク生成状態である. 以下のいずれかが返される.

normal(^Control, Report): 正常にタスクが生成できた. 引数の Control はタスクの制御ストリーム, Report はタスクの報告ストリームである.

abnormal: タスク生成に失敗した.

【注意】タスクの実行初期状態は resource_low (計算機資源 0) である.

execute(Code, Argv, Name, MinPrio, MaxPrio, Tag, InitialResoc, ^Result)

タスクの生成を行う. 引数は InitialResoc を除いて, 上述の execute/7 と同様である.

InitialResoc: タスク起動時の初期計算機資源であり, 形式は 2 要素ベクタ {**HighResoc, LowResoc**} である. HighResoc は荘園の初期言語定義資源上位 32 ビット, LowResoc は下位 32 ビット である (言語定義資源については 7.5 節 (290 頁) を参照).

profile(Code, Argv, Param, ^Result)

profile(Code, Argv, Name, MinPrio, MaxPrio, Tag, ^Result)

profile(Code, Argv, Name, MinPrio, MaxPrio, Tag, InitialResoc, ^Result)

プロファイル機構付きタスクの生成を行う. 引数は execute と同様である. プロファイル情報の収集法については 7.5 節 (290) および, 11.2 節 (410 頁) を参照されたい.

connection(^Result)

I/O ノード上のデバイスが生成できることを確認する. 引数 Result には **normal(Node)** (正常状態であり, 接続先 I/O ノード名が **Node** である) または **abnormal** (異常状態) が返される.

suicide(^Result)

タスク (そこで生成したタスクではない. 通常このメッセージを送るタスク自身) を強制終了させる. 引数 Result には **normal** (正常終了) または **abnormal** (異常終了) が返される.

【参考】正常に強制終了できた場合, タスクはこの Result を見ることなく終了するのが普通である.

abort(ResourceId, ^Result)

ResourceId で指定したタスクを強制終了させる.

ResourceId: 資源識別子

Result: 処理結果. **normal** (正常終了) または **abnormal** (異常終了).

number_of_feps(^Result)

現在接続されている FEP の数を知る. Result に処理結果が以下のいずれかが返る.

normal(Feps): 正常終了. 引数に FEP 数を返す.

abnormal: 異常終了

setenv(Name, Value, ^Result)

環境変数に値を設定する.

Name: 環境変数名. 未定義変数を含まないグラウンド値でなくてはいけない.

Value: 値. 未定義変数を含まないグラウンド値でなくてはいけない.

Result: 処理結果. 結果は **normal** または **abnormal** で それぞれ正常終了および異常終了である.

getenv(Name, ^Result)

環境変数値を求める.

Name: 環境変数名. 未定義変数を含まないグラウンド値でなくてはいけない.

Result: 処理結果

normal(Value): 正常終了. 引数に値を返す.

abnormal 指定された環境変数がない.

unsetenv(Name, ^Result)

環境変数値を削除する.

Name: 環境変数名. 未定義変数を含まないグラウンド値でなくてはいけない.

Result: 処理結果

normal: 正常終了. 環境変数名がない場合にもこの結果が返る.

abnormal 異常終了.

getenv(^Result)

すべての環境変数とその値を返す. Result は処理結果で, 以下のいずれかが返される.

normal(NameValueList): 正常終了. NameValueList には, もし環境変数がない場合には空リスト, あれば各々に対応して {**Name, Value**} の形式のベクタを要素に持つリストを返す. ここで Name は環境変数名, Value はその値である.

abnormal 異常終了.

id(^Result)

自タスクの資源識別子の問合せ. Result は処理結果で, 以下のいずれかが返される.

normal(ResourceId): 正常終了. ResourceId に自タスクの子資源識別子 (正整数, 親タスクの子資源の中での識別子) が返される.

abnormal: 異常終了

absolute_id(^Result)

自タスクの絶対資源識別子の問合せ. Result は処理結果で, 以下のいずれかが返される.

normal(AbsoluteResourceId): 正常終了. AbsoluteResourceId にトップタスクからみた資源階層に沿った代々の子資源識別子のリストが返される.

abnormal: 異常終了.

status(ResourceIdList, ResourceType, Verbosity, ^Result)

タスク状態の問合せ. 引数は以下の通り.

ResourceIdList: 問合せ対象タスク. 相対資源識別子で指定する.

ResourceType: 問合せ対象資源種類. 現在指定可能な資源の種類は * (すべての資源) または task (タスクのみ) である.

Verbosity: 情報の詳細度. 各対象資源種類によって指定可能な詳細度が異なる. 現在は task に対しては detail, * については brief のみが指定可能である.

Result: 問合せの結果. 以下の通りである.

normal(Info): 正常終了. 引数 Info には対象資源 (複数かも知れない) の状態を表すベクタのリストが返される (詳細は後述). ただし資源識別子の指定によっては, 指定経路の資源 ID のみの情報も含まれる場合がある.

abnormal: 異常終了.

資源状態を表すベクタの形式は, 対象資源の種類によって異なる.

- **対象資源種類が task でかつ情報詳細度が detail の場合:**

  {Id, Name, TaskStatus, Limit, Consume, Children}

  もしくは

  {Id, Children}

  各要素は以下のようなものである.

  Id: 相対資源識別子.

  Name: タスク名 (生成時に与えた文字列)

TaskStatus: タスクの状態. {**Cond, Stops**} という形式のベクタが返される. ただし状態不明 (例えば, 親荘園が停止中である場合) であるときには unknown が返される.
Cond はタスクの実行状態を表し, 以下のもののいずれかである.

normal: 正常状態 (資源の不足ではない状態のこと)

resource_low: 資源不足状態

terminating: 終了処理中

shut_terminating: 親タスクの強制終了による終了処理中状態

Stops はタスク停止理由のリスト. タスクが停止していない場合には [] である. 各要素は **stop(Reason)** の形式で, **Reason** は停止理由を表すアトムである.

Limit: 言語定義資源消費量の上限値を表す 2 要素ベクタである.

Consume: 現在までの言語定義資源消費量 (-1 のときは, 不明) を表わす要素が整数である 2 要素ベクタである.

Children: 子タスクの情報も獲得した場合, すべての対象子タスクに対する同様の情報のベクタをリストにしたもの.

- **対象資源種類が * でかつ情報詳細度が brief の場合：**

  {Id, Type, TypeInfo, Children}

  もしくは

  {Id, Children}

  各要素の意味は以下の通り.

  Id: 資源識別子.

  Type: 資源種類.

  Name: 資源種類に付随する情報.

  Children: 資源の情報も獲得した場合, すべての対象資源に対する同様の情報のベクタをリストにしたもの.

11.1.10 標準入出力資源の獲得

タスクが標準入出力資源を利用するには, まず標準入出力資源へのストリームを獲得する必要がある. 以下にその方法を説明する. なお子タスクへの標準入出力資源の設定方法は 11.1.9.2 節 (405 頁), 標準入出力の詳細およびプロトコルについては 9.4 節 (375 頁) を参照されたい.

11.1.10.1 獲得方法

荘園の例外生起用述語 raise を用いる.

shoen:raise(pimos_tag#task, stdio(Type), Stream)

標準入出力資源へのストリームを獲得する. 標準入出力へのストリームは 5 種類あり, その種別を述語の第 2 引数のパラメータ **Type** で指定する. **Type** で指定できるのは以下の通りである.

in: 標準入力ストリーム

out: 標準出力ストリーム

inter: 標準インタラクションストリーム

io: 標準入出力ストリーム

switch: 標準入出力制御ストリーム

11.1.11 資源の保護モード

PIMOS では複数ユーザ環境での資源の利用に備え, 資源の保護モードを用意している. 各資源に, この保護モードを付けることにより, 資源に対するアクセスの範囲をユーザごとに制限できる. なお, 保護モードは, その意味を整数で表現し, 獲得した資源に対して適当なプロトコルを送ることによって設定できる.

【注意】現在この保護モードが導入されているのはパッケージのみである (10.2.1.2節参照).

11.1.11.1 資源の持ち主

保護モードが導入されている資源にはその持ち主であるユーザ (オーナと呼ぶ) と持ち主であるグループ (グループオーナと呼ぶ) が存在する. PIMOS のユーザがこの資源を利用するとき, ユーザ自身とユーザが属するグループの権利により, その資源が利用可能である.

11.1.11.2 保護モード

保護モードは 9 bits の整数で表現される. 各 bit が 1 ならばそのモードは有効となる. 各 bit のモードを説明する.

0-bit: 一般ユーザの実行権

1-bit: 一般ユーザの書込権

2-bit: 一般ユーザの読出権

3-bit: グループオーナのグループに属するユーザの実行権

4-bit: グループオーナのグループに属するユーザの書込権

5-bit: グループオーナのグループに属するユーザの読出権

6-bit: オーナの実行権

7-bit: オーナの書込権

8-bit: オーナの読出権

11.2 タスク

タスクは PIMOS におけるプログラム実行の管理単位である.

タスクは他の PIMOS 定義資源と異なり能動的に動作し非同期的な通信を必要とするため, 制御 / 報告の二本のストリームを持つ. 制御ストリームはユーザからタスクへ, 報告ストリームは逆にタスクからユーザへのメッセージストリームである.

なおプロファイル情報用メッセージはプロファイル情報生成用タスクのみ有効である.

11.2.1 タスク制御ストリームへのメッセージ

stop

タスクを停止する. 停止したタスクの再起動は, 対応する報告ストリームメッセージ中の変数 (後述) により行うことができる.

allow_resource_report

資源不足報告を許可する. いったん資源不足を報告すると, その後このメッセージを受け取るまでは再び報告しない.

statistics

資源消費量の報告を要求する. 報告は対応する報告ストリームからのメッセージとして受け取る.

abort(ResourceId, ^Result)

ResourceId で指定したタスクを強制終了させる. 指定は, 直接の制御対象タスクからの相対資源識別子の形式である.

ResourceId: 資源識別子

Result: 処理結果

normal: 正常終了

abnormal: 異常終了

abort(^Result)

直接の制御対象タスクを強制終了させる. `abort([], ^Result)` と同じ. 引数 Result は以下の値をとる.

normal: 正常終了

abnormal: 異常終了

add_resource(HighResoc, LowResoc)

タスク内の言語定義資源消費量の上限値を追加する. HighResoc は荘園の初期計算機資源上位 32 ビット, LowResoc は下位 32 ビット である (計算機資源については 7.5 節 (290 頁) を参照).

id(^Result)

absolute_id(^Result)

status(ResourceIdList, ResourceType, Verbosity, ^Result)

対象タスクが制御対象タスクであることの他は, ジェネラルリクエストデバイスへのメッセージに対する処理と同様である (11.1.9.2 節 (403 頁) 参照).

profile

プロファイル情報収集を開始する. プロファイル情報生成タスクのみ有効である.

11.2.2 タスク報告ストリームからのメッセージ

statistics_started

resource_report_allowed

resource_added

resource_low

aborted

terminated

exception(Node, ExcCode, ExcInfo, ^NewCode, ^NewArgv)

return_resource(High_Resoc, Low_Resoc)

profiling_information(Pe, Type, Depth, Data)

profiling_started

profiling_completed

profiling_rejected

statistics(High_resoc, Low_resoc)

上記すべて荘園の報告ストリームの仕様に準拠 (7.5.4 (292 頁) 参照).

stopped(^ReasonAtom, ^CancelVar)

タスクの停止処理を開始したことを示す. 初期の停止理由 (stop reason) は none である. ReasonAtom に停止理由 atom を unify すれば, 停止理由を変更できる. これはタスクの状態問合せ時に停止理由として返されるものである. 停止状態を解除するためには, CancelVar を任意のデータで具体化すればよい.

11.3 サーバの実現方法

PIMOS のサーバタスクの実現方法について説明する. なお標準入出力処理機能の実現方法については 11.3.4 節 (424 頁) を, 資源の利用方法については 11.1 節 (399 頁) をそれぞれ参照されたい.

11.3.1 クライアントサーバ機能の概要

従来の PIMOS では, タスク間通信機能は 荘園の raise の機構を用いた親子タスク間の通信に限定されていた. このため, タスクの階層が親子間でない場合, タスク間通信を実現することが困難であった. たとえば, シェルで実現するパイプのような兄弟タスク間での通信は, 親タスクが通信のためのストリームを管理する必要があった.

さらに, 不特定タスク間での通信を実現するためは, PIMOS のトップレベルのタスクが全タスクを幾重の階層にまたがって管理する必要が生じる.

新しく導入したクライアントサーバ機能では, サーバの名前を PIMOS 資源管理部に登録することで, タスクの階層に依存することの無いタスク間通信機構を提供している.

本節では, クライアントサーバ機能の特徴を述べる.

11.3.1.1 クラスインスタンス

PIMOS のクライアントサーバ機能は, クラスインスタンスの関係を用いると説明し易い.

- サーバはクラスに相当する.
- PIMOS が直接管理するのは, クラス名に相当するサーバの名前だけである.
- 階層的なデバイスの階層, たとえば, ファイルサーバ → ファイルのような階層は, それぞれ, クラス → インスタンス に相当する. この階層は, クラス → インスタンス (クラス) → インスタンスのように, 任意の段数ネストできる.
- クライアントとサーバの間の通信路は OS を経由しない. このため, OS へのクライアント側からの要求は, サーバへの通信路を確保するときだけで済む.

11.3.1.2 保護の機能

タスク間通信時, クライアントの責任によるユニフィケーションエラーやデッドロックに対する保護機能は, サーバ側で実現する. PIMOS は, クライアントサーバ間のストリームの保護は行なわない.

ただし, サーバ側でクライアントタスクが終了した時に, その終了を知らせるためや, 通信の保護のためのフィルタプログラムの挿入の枠組は提供する.

これにより, サーバ側で提供していたデバイスの後始末や, サーバ側に残っている未処理の変数を処分することができる.

11.3.2 利用方法

本節では, PIMOS で提供するクライアントサーバ機能の概要を手順を追って説明する.

11.3.2.1 サービスの登録 (サーバ側)

PIMOS に対してアクセスは, 従来通り, raise によって, ジェネラルリクエストデバイス (GRD) への通信路を獲得し, GRD のストリームにプロトコル register/3 を発行することで行なう (11.3.3.1 節参照). クライアントサーバ機能を使用するためには, まずサーバ側から, PIMOS の資源サービス表に自分のサービス名を登録することが必要である.(図 11.2 参照) このサービス表への登録は, サーバへの通信路を登録することである.

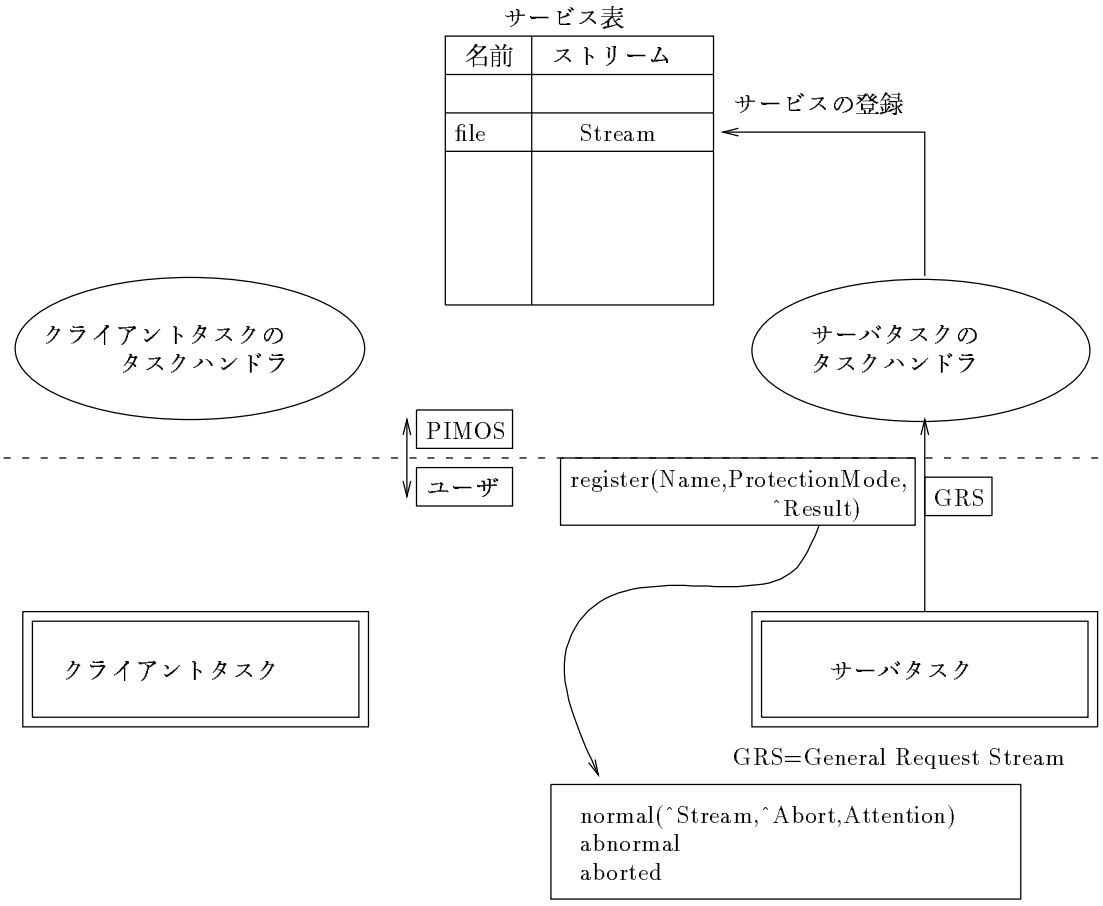

図 11.2: サービスの登録

11.3.2.2 資源の獲得 (クライアント側)

また, クライアント側では, サーバを利用するために, やはり, GRD に対して, サーバへの通信路をとるためのプロトコル request/3 を発行する. (図 11.3参照)

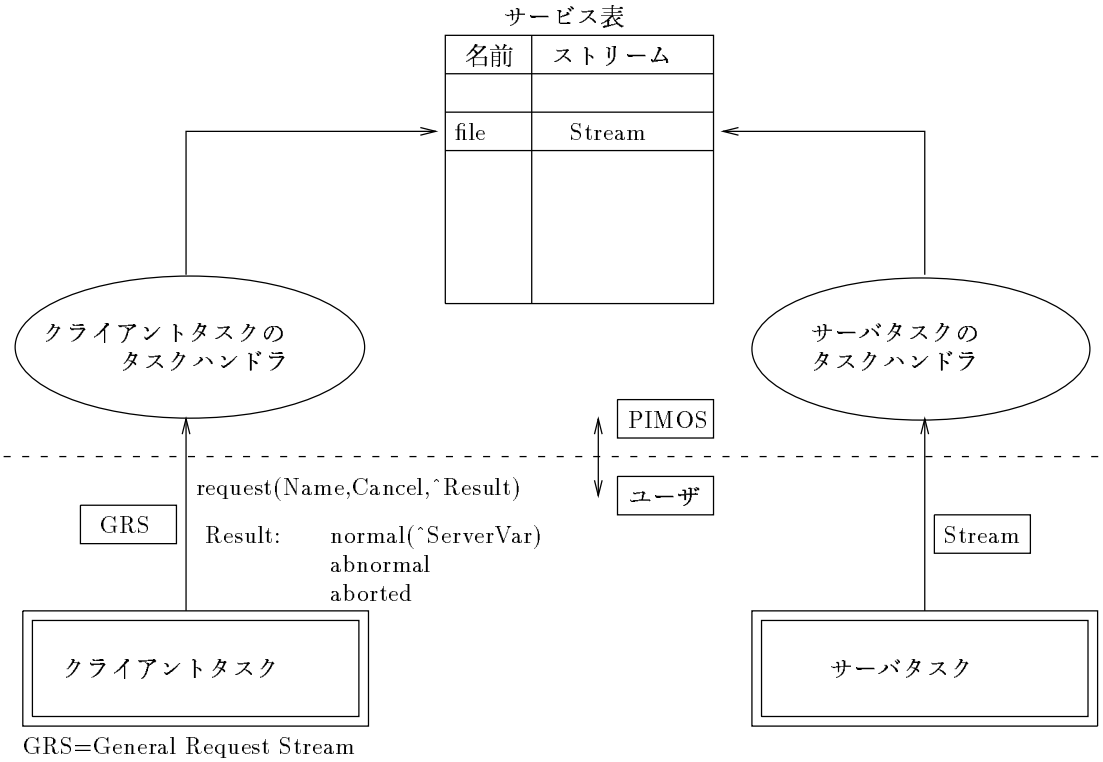

図 11.3: 資源の獲得

11.3.2.3 資源要求の受付開始 (サーバ側)

クライアントからの要求を満たすために, サーバ側からサービス表に対して, 資源要求の受付プロトコル get_request/3 を発行する. このプロトコルは, クライアント側で発行したプロトコル request/3 と 1 対 1 に対応しており, 両者のプロトコルが成立すると, クライアントサーバ間の通信路が確保される.

また, クライアントの失敗によって, サーバ側が失敗しないように, この通信路に保護フィルタを入れる必要がある. これを防ぐため保護フィルタのコードを, get_request/3 で与えることができる. (図 11.4参照)

このときサーバ側で作るデバイスは, 通常, 従来の PIMOS のリクエストデバイスのように, 単にインスタンスを生成するだけのデバイスとすることが望ましい. このように獲得階層を複数設けることにより, OS への集中的なアクセスを軽減することができる.

サーバ側でクライアントの通信路を管理するために, クライアントタスクの終了を通知して貰う機構を用意した. 具体的には, クライアントタスクが終了した場合, get_request で獲得した Abort 変数に aborted が具体化される. このため, サーバ側では, Abort 変数とクライアントへの通信路を管理する必要がある.

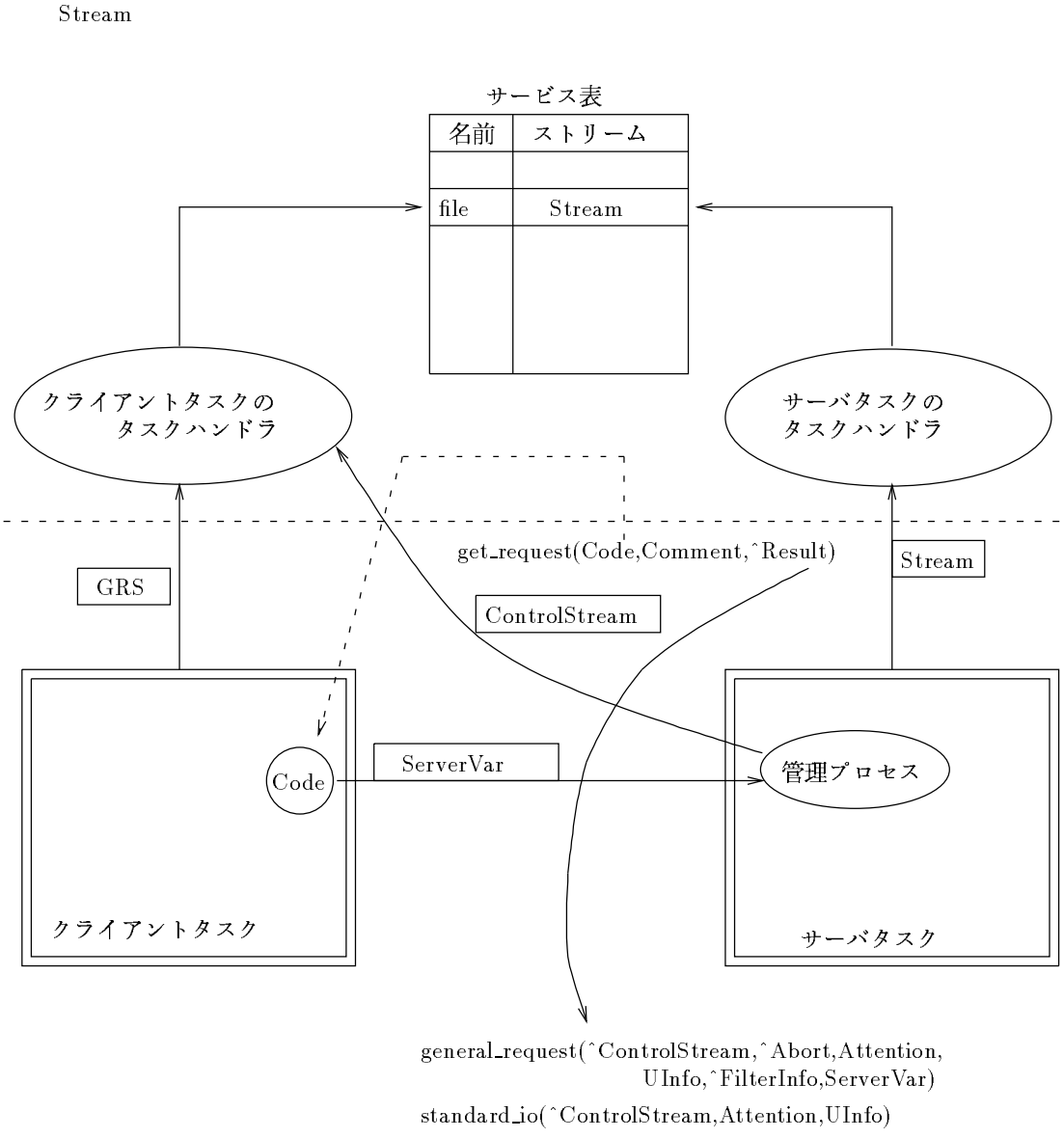

図 11.4: 資源要求の受付開始

11.3.2.4 子資源の生成

サーバがさらに下の階層のデバイスを子資源として生成した場合, クライアントサーバ間の通信路は, その分だけ増加する. その新しく加わった通信路をサーバ側で管理するため, get_request/3 で獲得した ServerControl というストリームに, add_resource/3 というプロトコルを設けている (図 11.5 参照). add_resource/3 で与える, Type, Name はコメント情報であり, 対応する資源を強制的に終了したい場合には, Abort 変数に aborted が具体化する.

11.3.3 クライアント・サーバ機能による資源の提供と獲得

カーネルによってユーザに提供する資源はタスクとジェネラルリクエストデバイス, 標準入出力資源だけである。その他の例えばウインドウやファイル等の資源はサーバと呼ばれるタスクによって実現され, ユーザタスクはサーバタスクとの間でタスク間通信を行なうことによってサービスを受けることとなる. なお, サーバはユーザタスクにより定義することも可能である.

一般に資源を利用することは, サーバと呼ばれるタスクとクライアントと呼ばれるタスクの間で通信することとなる。

ユーザが資源を利用するためには, サーバへのアクセス路を何らかの方法で獲得しなくてはならない. そのためカーネルは資源サービス表と呼ばれるものを用意し, サーバがこの表に自分のサービスを登録し, ユーザがこの表からサーバへのアクセス路を獲得することで資源を利用することができる.

【注意】サーバとタスクの関係は 1:1 である必要はなく, 複数のサーバをひとつのタスクで実現するのも, 複数のタスクでひとつのサーバを実現するのも作成する側の自由である.

11.3.3.1 サービスの登録方法

サービスの登録はジェネラルリクエストデバイスの次のコマンドによって行うことができる. なお本コマンドは特権ユーザのみ許可される.

register(Name, ProtectionMode, ^Result)

Name: サービス名アトム (登録名である)

ProtectionMode: 3 要素のベクタからなる次の保護情報である.

{Uid, Gid, Mode}

Uid: 資源のオーナのユーザ ID

Gid: 資源のグループオーナのグループ ID

Mode: 整数である。下位 3 bit の上から次の意味を割当てる.

- オーナの利用権
- グループオーナの利用権
- その他のユーザの利用権

それぞれ 1 であれば許可, 0 であれば不許可である.

Result: 次の通りである.

abnormal: 登録に失敗した.

aborted: タスクが終了直前である.

normal(^Stream, ^Abort, Attention): 登録に成功した.

Stream: サービス処理のためのストリームである.

Abort: Stream の利用を強制終了するためのアボート変数である.

Attention: ユーザからサーバへアテンションを告知するための変数である.

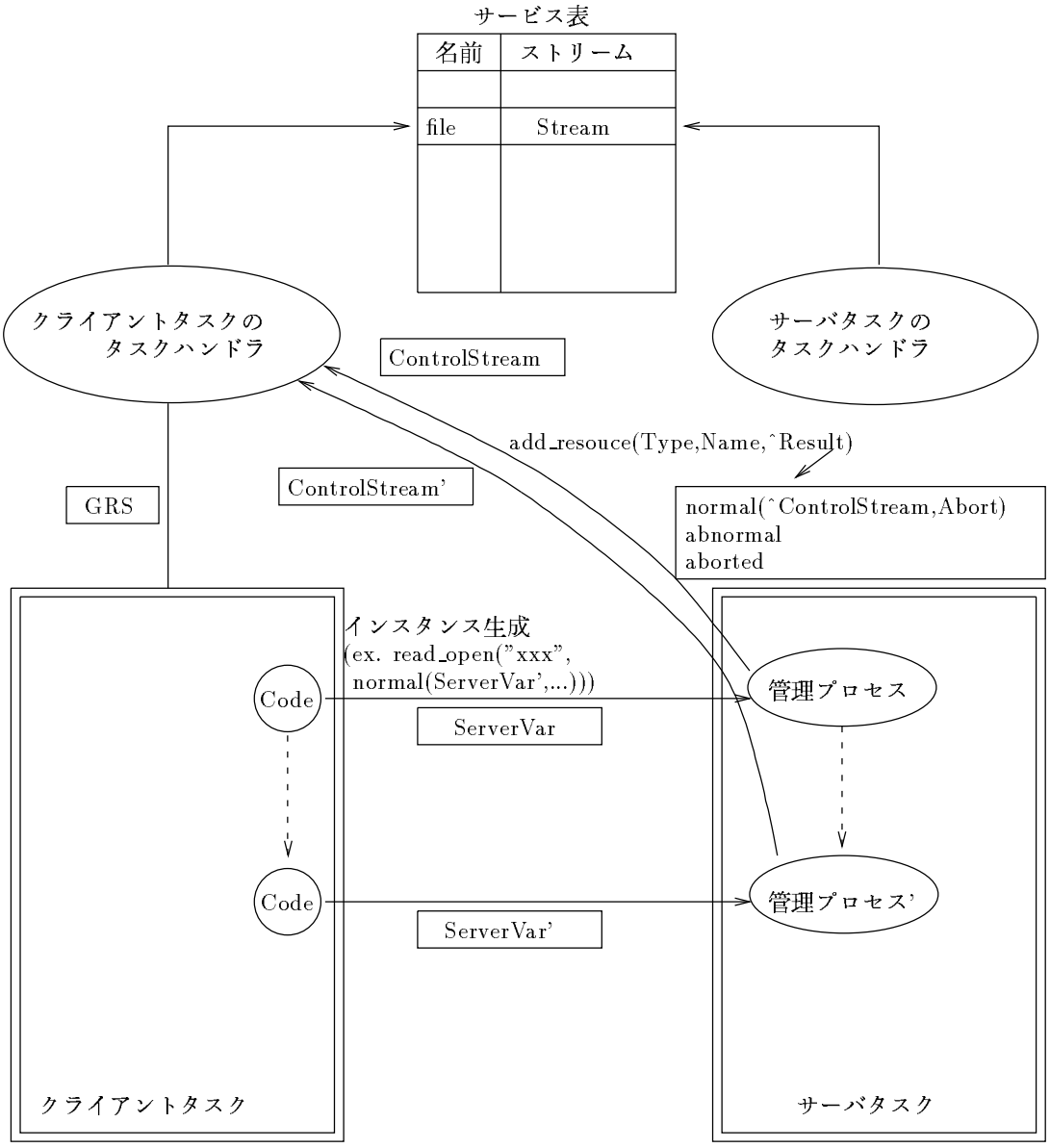
サービス表
名前
ストリーム
file
Stream
クライアントタスクの
タスクハンドラ
サーバタスクの
タスクハンドラ
ControlStream
add_resouce(Type,Name,^Result)
ControlStream'
GRS
normal(^ControlStream,Abort)
abnormal
aborted
インスタンス生成
(ex. read_open("xxx",
normal(ServerVar',...)))
Code
管理プロセス
ServerVar
Code
管理プロセス'
ServerVar'
クライアントタスク
サーバタスク

図 11.5: 子資源の生成

【注意】**Result** の結果については次節で詳細を説明する.

本コマンドによりサービス表のサービス名がリザーブされる.

11.3.3.2 サービス要求の受付

前節の register/3 の結果, すなわちサービス表に正しく名前が登録される結果として

normal(^Stream, ^Abort, Attention)

が返ることは前に述べた.

サーバは **Stream** ストリームを利用してサービスを行なう. 逆にこのストリームを閉じることによってサービスを終了する.

この **Stream** ストリームに次のメッセージを送ることで, ユーザに対するサービスの受け付け体制を採る.

get_request(Code, Comment, ^Request)

Code: サーバをユーザから保護するための保護フィルタコードである. 保護フィルタとは, ユーザのプロトコルエラー (特に `abnormal = normal` のようなユーザがサーバの処理結果をまたずに値を設定してしまったようなユニフィケーションのエラー) やユーザが値を設定しないために起こるサーバのデッドロックを避けるためのプロセスである.

サーバが起動するサービスプロセスとユーザとの通信路の間に入り, ユーザタスク内のフィルタプロセスとして動作する. **Code** は 3 引数の述語である. PIMOS は **Code** に特別な引数をあたえ, ユーザタスク内で次のように起動される.

apply(Code,{FilterInfo, UserVar, ServerVar})

Code: 保護フィルタコード

FilterInfo: サーバが指定した情報 (後述)

UserVar: ユーザ側通信用変数

ServerVar: サーバ側通信用変数

Comment: サーバをユーザが獲得するとユーザタスクの資源木 (11.1.6 節 (401 頁)) にその旨が登録される. その時情報としてサービス名と並んでこの Comment が登録される. データ型は未定義変数を含まないグラウンド値であれば何でもよいが, 慣習として文字列とする.

Request: サービス表よりユーザからのサービス要求メッセージが返る. 次の 2 種類のメッセージである. 説明は後述する.

- **general_request(^ControlStream, ^Abort, Attention, UInfo, ^FilterInfo, ServerVar)**
  通常の資源獲得要求である. 次節で説明する.
- **standard_io(ControlStream, Attention, UInfo)**
  標準入出力資源としての獲得要求である. PIMOS が直接送る. 次々節で説明する.
- **aborted**
  サーバの処理を強制終了した. 前節で説明した register/3 の引数 **Abort** に何らかの値を設定してサーバを資源サービス表から強制解除した場合に返ってくる.

サービスの終了時にはメッセージ register で獲得した Stream ストリームを閉じる. また, get_request を既に送出してしまった後にこのメッセージをキャンセルして終了させるには, register で獲得した **Abort** に何かの値を設定すればよい. キャンセルして再開することはできない.

Attention には、サーバにユーザから送られた異常通知が返る。通知はユーザがジェネラルリクエストデバイスコマンド (403 頁) `reset(Name, Status)` を送った時,

attention(NewAttention)

が返り attention の通知と共に次の reset 通知受信のための変数が返る.
サービス終了時には [] が返り, 終了の確認結果となる.

11.3.3.3 ジェネラルリクエストデバイスからの要求

メッセージ get_request の返答として, ユーザがジェネラルリクエストデバイスによって資源獲得を要求したとき次のメッセージが返る.

general_request(^ControlStream, ^Abort, Attention, UInfo, ^FilterInfo, ?Var)

各引数を説明する.

ControlStream: クライアントタスクのタスクハンドラ上に要求に対応して, デバイスを管理するプロセスが形成される. このプロセスへの制御ストリームであり, サーバ制御ストリームとも呼ぶ. 通常このデバイスをユーザが使い終った時, 閉じるべきものである. 次のメッセージを送ることができる.

add_resource(Type, Comment, Result)

新たにクライアントタスクのタスクハンドラにデバイス利用を登録する. 例えばリクエストデバイスがデバイスを生成するときに本メッセージを送り, クライアントに新デバイスを登録する. 詳細は 11.3.3.6 節 (423 頁) で述べる.

absolute_id(^Result)

本デバイスに対応する資源絶対 ID を求める. Result には次の値が返る.

- **normal(AbsoluteID)**
- **abnormal**
- **aborted**

change_comment(Com, ^Result)

本デバイスのコメント情報を変更する.

Com: 新しいコメント情報である. 一般に未定義変数を含まないグラウンド値を許すが, 文字列とする場合が多い.

Result: 次の通りである.

- normal
- abnormal
- aborted

Abort: ControlStream に送ったメッセージを解除するためのアボート変数である.

【注意】本変数を具対化した後, 必ず ControlStream を閉じること.

Attention: 本デバイスを獲得したクライアントタスクが終了した場合に, PIMOS より設定されるバルブ変数である.

UInfo: クライアントタスクの情報である. 詳細は 423 頁を参照されたい.

FilterInfo: 保護フィルタコードの第 1 引数に渡される情報である. 未定義変数を含まないグラウンド値である必要がある.

Var: クライアントと実際に通信するための変数である. 保護フィルタコードの第 3 引数に渡される.

11.3.3.4 標準入出力資源の生成要求

メッセージ get_request の返答として, 以下の標準入出力を設定するメッセージが送られる.

standard_io(ControlStream, Attention, UInfo)

標準入出力資源としての獲得要求である. タスク生成時にカーネルから直接送られる.

【参考】本メッセージは, 標準サーバを利用すれば処理を記述する必要はない. 11.3.4 節 (424 頁) を参照されたい.

ControlStream: 標準入出力資源サーバ制御ストリーム.

【注意】前節のメッセージ general request と入出力モードが異なる.
このストリームのメッセージプロトコルは以下の通りである.

open(STDInfo, switch(^ContSW, ^AbSW), io(^ContIO, ^AbIO))

標準入出力資源を生成する. 引数の説明は次の通りである.

STDInfo: 次のベクタである.

{**Type, IInfo, ^OInfo** }

Type: 獲得方法の種類である.

in, out, inter, in_out, in_inter, out_inter, in_out_inter

IInfo: 獲得時のパラメータである. 未定義変数を含まないグラウンド値である.

OInfo: 獲得結果を返すパラメータである. 未定義変数を含まないグラウンド値である. エラーである場合には, [] が返る.

ContSW: クライアントタスクの OS 内の管理プロセス上に要求に対応して標準入出力のスィッチを管理するプロセスが形成される. このプロセスへの制御ストリームである.

AbSW: 同様に ContSW に対応したアボート変数である.

ContIO: クライアントタスクの OS 内の管理プロセス上に要求に対応して標準入出力の I/O ストリームを管理するプロセスが形成される. このプロセスへの制御ストリームである.

AbIO: 同様に ContIO に対応したアボート変数である.

request(Type, Node, Communication, FilterInfo)

ユーザからの標準入出力関係のストリーム要求である.

Type: 標準入出力の獲得タイプ (`in, out, inter, io, switch`)

Node: ノード番号

Communication: ユーザの入出力用ストリーム. 保護フィルタコードの第 3 引数

FilterInfo: 保護フィルタコードへの情報. 保護フィルタコードの第 1 引数

make_branch(SubControl, CShut,ID, UserInfo)

標準入出力資源を子タスクに分け与える.

SubControl: 分岐後の標準入出力資源サーバ制御ストリーム. 分岐前とは次のメッセージのみ異なる.

open(Type, Status, switch(^ContSW, ^AbSW), io(^ContIO, ^AbIO))

Type は 獲得のタイプで

in, out, inter, in_out, in_inter, out_inter, in_out_inter

である. 他のパラメータは上述の同コマンドと同様である.

ID: 分岐に対する識別子である。swtich で切替える時に用いる.

CShut: 階層的に生成されたクライアントタスクの終了を通知するための変数である. 終了時に適当な値が設定される.

UserInfo: 標準入出力を利用するタスクの情報である. 次節で説明する.

Attention: 標準入出力資源設定対象のクライアントタスクが終了した場合に適当な値が設定される変数である.

UInfo: 標準入出力設定対象のクライアントタスクのユーザ情報. 次節で説明する.

11.3.3.5 サーバが獲得するユーザ情報

サーバが資源獲得要求を受けると同時に獲得するクライアントタスクの情報を説明する. 形式は以下の通りである.

{ Uid, Gids, Remote, Node }

Uid: ユーザ ID である.

Gids: Uid に対応した グループ ID のリストである.

Remote: タスクのもつリモート関係の情報である. サーバの種類により内容が異なる.

- システムサーバの場合:
  `{Fep, IONode, RUser, RPass, RInfo}`
  Fep: FEP 番号
  IONode: I/O ノード名もしくは FEP 番号
  RUser: リモートユーザ名もしくは 0
  RPass: リモートユーザのパスワードもしくは 0
  RInfo: リモート情報、現在は I/O ノード上のディレクトリ
- 一般のサーバの場合:
  `{Fep, IONode}`

11.3.3.6 クライアントタスクへの資源の利用登録

PIMOS では, ユーザとサーバが通信中に存在する制御ストリームを通じて, サーバが新たなユーザに新たな資源を与えることが可能である. 例えば file_request サーバは, ユーザが新たに file を利用することを資源木に反映させることで, 一般に利用者が使っている資源を確認することが可能である (11.1 節 (399 頁) を参考にせよ).

ここでは, サーバがもつ制御ストリームを通じて上述の処理を実現する次のコマンドを説明する.

add_resource(Type, Comment, ^Result)

Type: デバイス名アトム

Comment: Type に対するコメントである. 普通は文字列を指定する.

Result: 以下の通りである.

normal(^ControlStream, ^Abort): クライアントタスクの管理プロセス上に新しく登録したデバイスに対応するプロセスが形成された. 引数は以下の通り.

ControlStream: デバイスに対応するプロセスへの制御ストリームである. (11.3.3.4 節 (422 頁) を説明を参照されよ.)

Abort: このストリームへ送ったメッセージを解除するためのアボート変数である.

【注意】本変数を具体化した後, ControlStream を閉じる必要がある.

aborted: サーバタスクが終了処理中である.

abnormal: サーバが作れなかった.
どちらもサーバタスクが終了する状況の場合に起こる。

11.3.3.7 資源の獲得

資源の獲得方法については 11.1 節 (399 頁) を, 標準入出力資源の獲得方法については 9.4 節 (375 頁) を, 子タスクへの標準入出力の設定方法については 11.1 節 (399 頁) をそれぞれ参照されたい.

標準入出力機能をもったサーバの簡単な作り方については次節を参照せよ.

11.3.4 一般のサーバに標準入出力機能を持たせる方法

資源サーバは, 資源木が規定するプロトコルに従って PIMOS と通信を行う必要がある. もしもそのサーバが提供するストリームをさらに標準入出力を行うことができるストリームとしたければ, サーバプログラムから次の述語を呼び出し, 標準入出力サービスを行う機能をサーバに付加する必要がある.

stdio_server:go(GR_IN, GR_OUT)

この手順に従って標準入出力サービスを行う標準入出力サーバを呼び出す例を以下に示す.

```
shoen:raise(pimos_tag#task,general_request,GR),
stdio_server:go(GR,GROut),                    <========
GROut = [register(window_request,{0,0,3},Res)],
```

既存の標準入出力サーバ (stdio_server) を利用せず, 独自に標準入出力サーバを作成し利用することは全く差し支えない. ただしその場合には, PIMOS が規定する標準入出力用のメッセージを処理する必要がある.

具体的には standard_io/3 を正しく処理しなければならない.

11.4 ユーザ情報の管理

ここでは, PIMOS が管理するユーザ情報とそのアクセス方法について述べる.

11.4.1 ユーザとグループの識別子

ユーザ情報の中でも最も重要なのがユーザ ID とグループ ID である. 以下で説明する.

ユーザ ID: PIMOS では各ユーザに非負整数を割り当て, ユーザの識別子としている. この識別子のことをユーザ ID と呼ぶ.

特にユーザ ID 0 のユーザを 特権ユーザ と呼び, システムを管理する特権が与えられている.

グループ ID: PIMOS では資源の保護モード (11.1.11 節 (410 頁)) のためにユーザをグループ分け (重複を許す) する枠組を提供している. ユーザグループに対して非負整数を割り当てそのグループの識別子とする. この識別子のことをグループ ID と呼ぶ.

特にグループ ID 0 のグループを 特権グループ と呼び, 特権ユーザが必ず属するグループとしている.

11.4.2 ユーザ情報のデータベース

PIMOS で管理するユーザ情報には, ユーザ名, パスワードや, グループ名, 現在ログインしているユーザなどの情報がある. これらのユーザ情報はデータベースとして管理される. これらのデータベースへのアクセスも, 入出力機器やプログラム情報を管理するデータベースへのアクセスと同様に, PIMOS に依頼して行う. また, データベースへのアクセスパスも, 他の資源同様に PIMOS の定義する資源として管理される.

PIMOS では, ユーザ情報のデータベースを管理するため, 次のデバイスを資源として提供している.

ユーザデバイス PIMOS にログイン可能なユーザの情報を管理する. 具体的には, ユーザ名, パスワード, ユーザ ID, ログイン時のトップレベルゴール (通常, シェル), カレントディレクトリを管理する.

グループデバイス ユーザデバイスで定義されたユーザのグループに関する情報を管理する. 具体的には, グループ名, グループ ID, グループに属するユーザの ID を管理する. PIMOS は上述のユーザデバイスの情報と合わせて, 提供する資源の保護モードをチェックする (11.1.11節 (410 頁), 10.2.1.2 節 (388 頁) 参照).

ログインデバイス 現在, PIMOS にログインしているユーザの情報を管理する. 具体的には, ユーザ名とログイン ID である.

なお, ユーザ情報のデータベースをアクセスするためのプロトコルは, 特権ユーザ (ID:0) でなければ, 使用できないものがある.

11.4.2.1 ユーザリクエストデバイス

ユーザリクエストデバイスは,

```
shoen:raise(pimos_tag#task, general_request, GR)
```

としてジェネラルリクエストデバイスへのストリームを獲得し, 他の資源同様コマンドを送ることによって, 獲得することができる.

ユーザリクエストデバイスには, 次のコマンドを送ることが可能である.

コマンドプロトコル

create(user(^Result))

ユーザデバイスを生成する. 正常に生成できた場合には, `Result` に normal(UserDeviceStream, Abort, ^Attention) を返し, そうでない場合には, abnormal を返す.

create(group(^Result))

グループデバイスを生成する. 正常に生成できた場合には, `Result` に
normal(GroupDeviceStream, Abort, ^Attention) を返し, そうでない場合には, abnormal を返す.

create(login(^Result))

ログインデバイスを生成する. 正常に生成できた場合には, Result に normal(LoginDeviceStream, Abort, ^Attention) を返し, そうでない場合には abnormal を返す.

11.4.2.2 ユーザデバイス

コマンドプロトコル

set_up(UsersList, ^Result)

リスト UsersList の各要素をユーザ情報として登録し, 成功なら Result に normal, 失敗なら abnormal を返す.
UsersList の形式は次の通り.

```
UsersList ::= [UserInfo...]
UserInfo ::= {UserName,Password,UID,Priority,Directory,Node,EntryPoint,CommentList}
EntryPoint ::= Package::Module:Predicate
CommentList ::= [Comment...]

UserName, Password, Directory : 16 ビットストリング (Password は暗号化済み)
Comment : 未定義変数を含まないグラウンド値
UID : 整数
Module, Predicate : アトム
```

【注意】このプロトコルはスーパユーザ (UID: 0) のみが使用可能なプロトコルである. このプロトコルを一般ユーザが使用した場合, protection_violation が Result に返る.

authenticate(Username, Password, ^Result)

Username, Password で示される (それぞれ 16 ビットストリング) ユーザ名パスワードをもつユーザがいるかどうか調べ, 成功すれば Result に normal を返し, 失敗なら abnormal を返す. パスワードを暗号化する必要はない.

get_uid(UserName, ^Result)

UserName で示されるユーザの識別子 UID を獲得する. 成功すれば Result に normal(UID) を返し, 失敗なら abnormal を返す.

get_info(UID, ^Result)

UID で示されるユーザの情報を獲得する. 成功すれば Result に normal({UserName, Priority, Directory, Node, EntryPoint, CommentList}) を返し, 失敗なら abnormal を返す.

status(UID, ^Result)

UID で示されるユーザの参照情報を獲得する. 成功すれば Result に normal(RefCount) を返し, 失敗なら abnormal を返す. RefCount は実行した get_info の回数を表す. status 実行後, 対象とする RefCount は 0 クリアされる.

get_all_info(^Result)

ユーザデバイスに登録されている全てのユーザの情報を獲得する. 成功すれば Result に normal(UserList) を返し, 失敗なら abnormal を返す.

【注意】このプロトコルはスーパユーザ (UID: 0) のみが使用可能なプロトコルである. このプロトコルを一般ユーザが使用した場合, abnormal が Result に返る.

change_password(UID, OldPassword, NewPassword,^Result)

UID で示されるユーザ ID を持つユーザのパスワードを OldPassword から NewPassword に変更する. 成功なら normal, 失敗なら abnormal を Result に返す. ユーザ ID が登録されていなければ失敗. 登録されているパスワードと OldPassword が一致しなければ失敗.

11.4.2.3 グループデバイス

コマンドプロトコル

set_up(GroupsList, ^Result)

リスト GroupsList の各要素をユーザ情報として登録し, 成功なら Result に normal, 失敗なら abnormal を返す.
GroupsList の形式は次の通り.

```
GroupsList ::= [GroupInfo...]
GroupInfo ::= {GroupName,GID,[UID...],CommentList}
CommentList ::= [Comment...]

GroupName, UserName : 16 ビットストリング (Password は暗号化済み)
Comment : 未定義変数を含まないグラウンド値
GID : 整数
UID : 整数
```

【注意】このプロトコルはスーパユーザ (UID: 0) のみが使用可能なプロトコルである. このプロトコルを一般ユーザが使用した場合, protection_violation が Result に返る.

get_gid(GroupName, ^Result)

GroupName で示される (16 ビットストリング) グループ名に相当するグループ識別番号 (GID) を調べ, 成功すれば Result に normal(GID) を返し, 失敗なら abnormal を返す.

get_gids(UID, ^Result)

UID で示される (16 ビットストリング) ユーザ名が属するグループ識別番号 (GID) を調べ, 成功すれば Result に normal([GID...]) を返し, 失敗なら abnormal を返す.
結果の GID のリストには GID が昇順で並ぶ. 特に GID のリストの先頭要素, つまりユーザのの持つ最小の GID のことを 代表グループ **ID** と呼ぶ.

get_users(GID, ^Result)

GID で示されるグループに属するユーザを調べ, 成功すれば Result に normal([UID...]) を返し, 失敗なら abnormal を返す.

get_all_info(^Result)

グループデバイスに登録されている全てのユーザの情報を獲得する. 成功すれば Result に normal(GroupList) を返し, 失敗なら abnormal を返す.

【注意】このプロトコルはスーパユーザ (UID: 0) のみが使用可能なプロトコルである．このプロトコルを一般ユーザが使用した場合, abnormal が Result に返る．

11.4.2.4 ログインデバイス

コマンドプロトコル

login(UserName, Info, ^Result)

UserName で示されるユーザ名のユーザをユーザ情報 Info とともに登録し, 成功なら Result に normal(LID), 失敗なら abnormal を返す.

【注意】このプロトコルはスーパユーザ (UID: 0) のみが使用可能なプロトコルである．このプロトコルを一般ユーザが使用した場合, protection_violation が Result に返る．

logout(LID, ^Result)

LID で示されるユーザの情報を削除し, 成功なら Result に normal, 失敗なら abnormal を返す.

【注意】このプロトコルはスーパユーザ (UID: 0) のみが使用可能なプロトコルである．このプロトコルを一般ユーザが使用した場合, abnormal が Result に返る．

get_info(LID, ^Result)

LID で示されるユーザの情報を獲得する．成功すれば Result に normal({UserName, [Partition, Date, Time, IONode, IONodeUser, Network]}) を返し, 失敗なら abnormal を返す.

get_login(UserName, ^Result)

UserName で示されるユーザの情報を獲得する．成功すれば Result に normal([LID...]) を返し, 失敗なら abnormal を返す.

get_all_info(^Result)

ログインデバイスに登録されている全てのユーザの情報を獲得する．成功すれば Result に normal(LoginList) を返し, 失敗なら abnormal を返す.

【注意】このプロトコルはスーパユーザ (UID: 0) のみが使用可能なプロトコルである．このプロトコルを一般ユーザが使用した場合, abnormal が Result に返る．

11.5 標準に提供するエラー報告方法

PIMOS では, ユーザ・プログラムからのエラー報告を統一的に扱うため, その報告形式に関して標準的な枠組を提供している．この枠組を用いることによって, シェルやリスナなどで表示するエラー・メッセージの形式が統一されて分かり易くなったり, エラー表示ルーチンを書く手間が軽減できるなどの利点がある．以下に, 標準的なエラー報告をユーザプログラムで実現する方法について述べる．

11.5.1 エラー報告の形式

プログラム中から, 標準的な形式のエラー報告を行うためには, 次の形式で書く．

```
shoen:raise(pimos_tag#error, '$PIMOS$'(Error,Info), Data)
```

各引数の形式は, 次の通り.[1]

Error : グラウンド値でなければならない. バッファの putt メッセージを用いて出力する値である. 通常はエラーを起こした (検出した) デバイス名やユーティリティ名を与える.

Info : ストリング (文字列) でなければならない. バッファの putb メッセージを用いて出力する. 通常はエラーの内容を与える.

Data : 任意の項を指定できる. 変数が含まれていてもよい. シェルやリスナはこの項をラップ (wrap) した後, バッファの putwt メッセージで出力する. 通常はエラーとなったメッセージやメッセージの引数を与える.

【参考】ここで説明したエラー報告の表示形式は, 次の部分を参照すること.

シェル: 第 2.1.5.3 節

リスナ: 第 4.1.8.3 節

[1] バッファのメッセージに付いては, 「9.2.1 バッファ」の節 (338頁〜) を参照.

付録 C

FEP 上のストリング入出力デバイスの作成方法

ストリング入出力デバイスとは, ユーザが FEP 上に ESP を用いて定義したデバイスを, PIMOS 上で操作するインタフェースを提供するものである. ここでは, ESP による任意のストリング入出力デバイスの定義方法について説明する. 以下で述べるパッケージ名などは, 全て SIMPOS 上で用いられる ESP のパッケージである.

ストリング入出力のデバイス名は, 一般的に "パッケージ名 ## クラス名" である. パッケージ名は省略できない (省略した場合にはデバイスが生成された時にデフォルトになっていたパッケージとなるので, 保証できない) ものとする. 従って, ユーザはこのデバイス名に対応したパッケージ名とクラス名で, ESP プログラムを定義する必要が有る.

このデバイスのプログラムを作成するに当たっては, 以下のような規約に従う必要があるが, この規約はユーザの定義したストリング入出力デバイスのプログラム (と, もし継承があればその継承したクラスも含む) についてのみ有効であり, ストリング入出力デバイスのプログラム中から呼ばれる別の ESP プログラムについてはユーザが任意に定義してかまわない.

【参考】これらのクラスは 1 度テンプレートをセーブしておけば, デバイスを生成する時に自動ロードされる.

継承するクラス: PIMOS のデバイスとしての機能 (KL1 と KL0 のデータ変換など) を得るために, クラス pimfep##as_pimos_string_io_device を継承する.

継承してはならないクラス: 既に継承しているので, simpos##as_program を継承してはならない.

また simpos##as_resource は SIMPOS の規則により simpos##as_program を継承しているクラスで継承することができない. したがって, このクラスを pimfep##as_pimos_string_io_device を継承したクラスで継承してはならない.

ウインドウ, ファイルなどのクラスは, simpos##as_resource を継承しているので, 同様に継承してはならない.

定義してはならないメソッド / スロット: 以下に示すメソッド及びスロットは, クラス pimfep##as_pimos_string_io_device で既に定義されているので, ユーザは定義してはならない.

【注意】pimfep##as_pimos_string_io_device は, simpos##as_program を継承しているので, これを継承するクラスで定義してはいけないメソッド及びスロットについても, 定義してはならない (詳細は, 小型化 PSI/SIMPOS プログラミング説明書 (I) を参照のこと).

クラスメソッド: :stream/2, :process/4

インスタンスメソッド: :abort_loop/1, :loop_main/1, :reply_message/2, :set_device_id/2, :set_creation_status_stream/2, :set_output_stream/2, :set_input_stream/2, :set_close_flag/2, :device_id/2, :creation_status_stream/2, :output_stream/2, :input_stream/2, :aborted/1, :attention/1, :disconnected/1, :is_closed/1, :close/3, :initiate/1, :goal/1, :trace_string_io/1, :abort_string_io/2, :do_io/6, :set_pimfep_trace_flag/2, :pimfep_trace_flag/2, :io/5, :exit/1

インスタンス属性スロット: なし.

定義するメソッド: 以下のインスタンスメソッドを定義する.

:io(Instance, Command, FromMulPString, ^ToMulPString)

引数は以下の通り.

Instance: デバイスのインスタンスオブジェクト.

Command: 整数. 入出力命令の種類を示す ID.

FromMulPString: 16 ビットストリング. 並列マシン本体 (KL1 側) から送ったデータ. 通常, 入出力命令を実行するために必要なデータが入る.

ToMulPString: 16 ビットストリング. 入出力命令実行後, 並列マシン本体 (KL1 側) へ送るデータ. 通常, 入出力命令を実行した結果をいれる.

このメソッドは, Command で指定される入出力命令を実行するように定義する. 実際の入出力命令が何らかの原因で失敗した場合には, このメソッドを失敗させることにより,KL1 側の入出力メッセージの Status が abnormal になる。

また, 入出力命令の内容によっては KL1 側にデータを送る必要がない場合, ToMulPString は未定義変数のままで良い.

初期化: デバイスの初期化は, デバイス (インスタンス) の生成時に行うのが一般的である. しかし, ストリング入出力デバイスの初期化は, 一般の入出力命令のひとつとして定義しなければならない.

【参考】これは, 次の理由による.

- デバイス生成時に初期化を行うと, FEP 上の PIMOS デバイスを管理する機構が働かなくなるので, 異常処理や終了処理が保証できなくなる.
- デバイス生成時に初期化を行なうと, 生成に失敗した時に, 指定されたデバイスが生成できなかったためなのか, 生成はできたが, 初期化に失敗したためなのかが分かり難くなる.

アボート処理: ストリング入出力デバイスは他の PIMOS デバイスと同様に KL1 プログラムからアボートすることが可能である. ストリング入出力デバイスをアボートすると現在実行中の :io/4 がアボートされるが, アボート処理にあたるものは特に定義していないので, ユーザがアボート時にストリング入出力デバイス上で何か処理したい場合には, :device_abort(Instance) を定義することができる.

終了処理: ストリング入出力デバイスのプロセスは, KL1 側のストリームが閉じられることにより終了する. ユーザが終了前に何か処理したい場合には, :pimos_user_exit(Instance) を定義することができる.

デバッグ: 定義したストリング入出力デバイスのプログラムをデバッグする際には, プログラム中で :trace_string_io(Instance) を呼び出すことにより, 呼び出した時点から SIMPOS のデバッガを用いて当該プログラムをトレースすることができる.

【注意 1】本機能では, ユーザの定義したメソッドのみでなくデバイスのプログラム全体がトレースされてしまう. したがって本機能の使用にあったては, 本来のデバイスの処理を妨げることのないよう十分注意をする必要がある (SIMPOS のスパイ機能を使って, ユーザの定義したメソッドのみをトレースすることが望ましい).

【注意 2】本機能は SIMPOS 7.1 版以降でのみ使用が可能で有る (7.1 版より前の版では, 正常に動作しない場合が有る).

【注意 3】本機能は PIMOS をリモートの I／O ノードから使用している際には, 使うことができない.

プログラミング例

以下にデバイスプログラムの例を示す．このプログラムは，メニューウインドウからコマンドを入力して，別ウインドウにエコーバックする例である．

FEP 上の ESP プログラム

```
use_package(my_device_package).
class string_io_example has
nature
        pimfep##as_pimos_string_io_device ;
instance
attribute
        echo_window, read_menu ;

% デバイスの初期化コマンドを受ける．
:io(Obj, 0, FromMulPString, ToMulPString) :- !,

    %プログラムのトレースを開始する．
    :trace_string_io(Obj),
    %

    :init(Obj, FromMulPString, ToMulPString) ;
%%
:io(Obj, 1, FromMulPString, ToMulPString) :- !,
    :read_from_menu(Obj, FromMulPString, ToMulPString) ;
:io(Obj, 2, FromMulPString, ToMulPString) :- !,
    :write_to_window(Obj, FromMulPString, ToMulPString) ;

% 実際の初期化処理
:init(Obj, Input, "") :-
    ItemsList = [
        {"test", common,0,_},      {"analysis",common,1,_},
        {"manipulate",common,2,_}, {"check",common,3,_},
        {"exit",common,4,_}],
    :create(#menu, [title("MENU"),items_list(ItemsList)], Menu),
    :activate(Menu), Obj!read_menu := Menu,
    string_element(Input, 0, Sizex),
    string_element(Input, 1, Sizey),
    string_element(Input, 2, Positionx),
    string_element(Input, 3, Positiony),
    :create(#pmacs_window,
        [size(Sizex, Sizey),position(Positionx, Positiony),
         title("ECHO WINDOW")], W),
    :activate(W), Obj!echo_window := W ;
%%

:read_from_menu(Obj, Input, String) :-
    W = Obj!echo_window,
```

```
    :putl(W, Input), :flash(W),
    :read(Obj!read_menu, Type),
    new_string(String, 1, 16),
    set_string_element(String, 0, Type) ;

:write_to_window(Obj, Input, "") :- !,
    W = Obj!echo_window,
    :putl(W, Input), :flash(W) ;

% 終了処理
:pimos_user_exit(Obj) :- !,
    W = Obj!echo_window,
    :putl(W, "Terminate Device"), :flash(W) ;
%%

end.
```

本体上の KL1 プログラム

```
:- module string_io_example.
:- public go/0.
:- with_macro pimos.

go :- true |
    shoen:raise(pimos_tag#task, general_request, GRD),
    GRD = [string_io(STGIO)],
    go1(STGIO).

go1(normal(STGIO,_,_)) :- true |
    STGIO = [create(string#"my_device_package##string_io_example",STGIOR)],
    go2(STGIOR).

go2(normal(STGIO,_,_)) :- true |
    new_string(S, 4, 16),
    set_string_element(S, 0, 400, S1),
    set_string_element(S1, 1, 190, S2),
    set_string_element(S2, 2, 180, S3),
    set_string_element(S3, 3, 0, S4),
    STGIO = [io(0,S4,FromFEP)|STGIOR],
    read_menu(FromFEP, STGIOR).

read_menu(normal(_), STGIO) :- true |
    STGIO = [io(1,string#"PLEASE SELECT MENU",FromFEP)|STGIOR],
    execute(FromFEP, STGIOR).

execute(normal(FromFEP), STGIO) :- true |
    string_element(FromFEP, 0, Type, _),
    processing(Type, STGIO).
```

```
processing(0, STGIO) :- true |          % "test"
    STGIO = [io(2, string#"test start ... end", FromFEP)|STGIO2],
    read_menu(FromFEP, STGIO2).
processing(1, STGIO) :- true |          % "analysis"
    STGIO = [io(2, string#"analysis start ... end", FromFEP)|STGIO2],
    read_menu(FromFEP, STGIO2).
processing(2, STGIO) :- true |          % "manipulate"
    STGIO = [io(2, string#"manipulate start ... end", FromFEP)|STGIO2],
    read_menu(FromFEP, STGIO2).
processing(3, STGIO) :- true |          % "check"
    STGIO = [io(2, string#"check start ... end", FromFEP)|STGIO2],
    read_menu(FromFEP, STGIO2).
processing(4, STGIO) :- true |          % "exit"
    STGIO = [io(2, string#"SESSION END", _)].
```

付録 D

プリプロセッサの作成方法

D.1 概要

プリプロセッサとは, KL1 言語をベースとした上位言語の開発を容易にするためのセルフコンパイラの機構である. ユーザは, 以下に述べる規則に従ってモジュールを作成し, そのモジュールをセルフコンパイラに教えることで, 簡単にプリプロセッサを付加できるようになっている. (付加の方法については, 操作編 3 章 コンパイラ 45頁を参照のこと)

【注意 1】以下に述べる仕様の多くの部分は 2.5 版とは異なっているが, 実際には 3.0 版のプリプロセッサ機能は 2.5 版対応のプリプロセッサでも使用可能である. しかし, この場合 3.0 版から導入されたソースファイル上での位置の表示機能は無効となる.

【注意 2】ユーザが定義したプリプロセッサにバグあるいは何らかの異常が生じた場合は例外が発生し, その結果としてセルフコンパイラが落ちる可能性がある.

【参考】3.0 版からベリファイア機能が付加された. これは, UNIX の C 言語における lint プログラムに相当するもので, ユーザが定義した言語の静的な解析・検証をおこなうことを目的とする. PIMOS のセルフコンパイラにおいては, この機能を一般的に「ベリファイア」と呼ぶこととし, プリプロセッサ同様, ユーザが定義したベリファイアをセルフコンパイラに組み込むことができる.

D.2 プリプロセッサの機構

- セルフコンパイラにおけるコンパイルでは以下の手順でプリプロセッサを割り当てている.

  1. パーサの割り当て

     オープンしたファイルの拡張子から, 対応するパーサを探して見つかれば, そのパーサとファイルのストリームを接続する. もし対応するパーサが見つからなければ, 標準パーサである kl1_standard_preprocessor を割り当てる.

  2. プリプロセッサの割り当て

     オープンしたファイルの拡張子から, 対応するパーサを探してプリプロセッサを割り当てる. もし, 対応するプリプロセッサが見つからなければ KL1 言語と思って対応するプリプロセッサを探す.

  3. プリプロセッサの実行

     パーサの出力とプリプロセッサを接続し, プリプロセッサを実行する. 同時に, プリプロセッサの設定からプリプロセッサの出力言語を取り出して, その出力言語が “pkl1” でない場合は再度プリプロセッサを割り当てる (プリプロセッサの連鎖). 出力言語が “pkl1” であり, かつプリプロセッサが正常に終了しているならば, KL1 コンパイラを起動する.

4. 表示メッセージのソート

   パーサ, プリプロセッサがそれぞれ出力する表示メッセージは, 実行の正常終了を確認後ソートされる. もし, 異常終了していたならば, その処理より後方のプリプロセッサの処理結果は捨てられる. このソートにおいて, パーサに比較述語が定義されているならば, その定義されている比較述語によりソートされる. そうでなければ, モジュール kl1_standard_preprocessor で定義されている比較述語が用いられる.

   【参考】表示メッセージを位置情報でソートするのは, プリプロセッサの後段においても表示されるメッセージはソースファイルの順序に出力されて欲しいからである.

5. 表示メッセージの出力

   パーサに message/4 という述語が定義されているならば, ソートが完了した表示メッセージ中の位置に関する情報をどのように表示するかはこの述語により決定される. そのような述語が定義されていない場合は, このようなフィルタリング処理はおこなわれない.

- また, `!translate` コマンドでは, 以下のような手順になる.

1. パーサの割り当て

   コンパイルと同じ.

2. プリプロセッサの割り当て

   コンパイルと同じ.

3. プリプロセッサの実行

   プリプロセッサの連鎖まではコンパイルと同じであるが, 連鎖が途切れたか, 出力言語が “pkl1” になった時点で, 出力ファイルをオープンし, 対応する言語のアンパーサが見つかればそのパーサとプリプロセッサの出力を結んで実行する. そうでなければ, 標準のアンパーサである kl1_standard_preprocessor のアンパーサが割り当てられる.

4. 表示メッセージのソート

   コンパイルと同じ.

5. 表示メッセージの出力

   コンパイルと同じ.

- 一方, `!verify` コマンドでは以下のような手順になる.

1. パーサの割り当て

   コンパイルと同じ.

2. ベリファイアの割り当て

   対応する言語にベリファイアが設定されていれば, そのベリファイアにパーサの出力を接続して終了. そうでなければ, コンパイルと同様に対応するプリプロセッサを割り当てる.

3. プリプロセッサの実行

   プリプロセッサの連鎖まではコンパイルと同じであるが, 連鎖が途切れたか, 出力言語が “pkl1” になってもベリファイアがひとつも設定されていなければ, エラーとして終了する.

4. ファイルにまたがるベリファイ

   指定されたファイルに対し全てのベリファイの割り当てが終了して, ベリファイアに inter_file_verify/3 という述語が定義されているならば, ファイルをまたがるようなベリファイ処理をこの述語に対し実行する.

5. 表示メッセージのソート

   コンパイルを同じ.

6. 表示メッセージの出力
   コンパイルと同じ.

以下のようにプリプロセッサを設定した場合において, コンパイルや translate, verify コマンドにおけるパイプライン構成をそれぞれ図示する.

```
COMPILE> !preprocessor lm2 kl1 my_preproc
COMPILE> !parser lm2 my_parser
COMPILE> !checker lm2 me_verifier
```

- **KL1** 言語のコンパイルの場合

```
[kl1 ソースファイル] → kl1_standard_preprocessor (パーサ)
                            ↓ [言語 kl1]
                       kl1_macro_expander (マクロ展開プリプロセッサ)
                            ↓ [言語 pkl1]
                       kl1 セルフコンパイラ → [モジュール]
```

- **LM2** 言語のコンパイルの場合

```
[lm2 ソースファイル] → my_parser
                            ↓ [言語 lm2]
                       my_preproc
                            ↓ [言語 kl1]
                       kl1_macro_expander (マクロ展開プリプロセッサ)
                            ↓ [言語 pkl1]
                       kl1 セルフコンパイラ → [モジュール]
```

- **KL1** 言語の translate の場合

```
[kl1 ソースファイル] → kl1_standard_preprocessor
                            ↓ [言語 kl1]
                       kl1_macro_expander (マクロ展開プリプロセッサ)
                            ↓ [言語 pkl1]
                       kl1_standard_preprocessor → [pkl1 ソースファイル]
                        (アンパーサ)
```

- **LM2** 言語の translate の場合

```
[lm2 ソースファイル] → my_parser
                            ↓ [言語 lm2]
                       my_preproc
                            ↓ [言語 kl1]
                       kl1_macro_expander (マクロ展開プリプロセッサ)
                            ↓ [言語 pkl1]
                       kl1_standard_preprocessor → [pkl1 ソースファイル]
                        (アンパーサ)
```

- **KL1** 言語の verify の場合

```
[kl1 ソースファイル] → kl1_standard_preprocessor (パーサ)
                            ↓ [言語 kl1]
                       kl1_macro_expander (マクロ展開プリプロセッサ)
                            ↓ [言語 pkl1]
                       kl1_standard_preprocessor (ベリファイア)
```

- **LM2** 言語の verify の場合

```
[lm2 ソースファイル] → my_parser
                              ↓ [言語 lm2]
                          my_verifier
```

以下の節において説明の重複を避けるため, 共通な変数の説明を以下に述べる.

- DevIn,DevOut

  DevIn は入力デバイス (ファイル) へのストリームである. バッファは挿入されていない.

  DevOut は出力デバイス (ファイル) へのストリームである. DevIn と同様, バッファは挿入されていない.

- In,Out

  プリプロセッサ間の受け渡しをするリスト. In はプリプロセッサへのリスト, Out はプリプロセッサから出力されるリスト. In は前段のパーサあるいはプリプロセッサからの出力でもあり, Out は後段のプリプロセッサあるいはアンパーサへの入力でもある.

  このリストの要素は, {**(-),WrappedTerm,PosiotionInfo**} という形式のベクタである. ここで, WrappedTerm はバッファから getwt されたラップされた項である. 一方, PositionInfo は, {**Position,Info**} という 2 要素ベクタで, Position はソースファイル中の位置を示す項, Info はパーサが任意に埋め込むことができる項である. Position はリストであり, ソースファイル名ストリング, ソースファイル中の位置を示す整数がこの順序で並んでいる. もし, ソースファイルが include のように他のファイルを含んでいるような場合は, このリストの後ろに, include ファイルのファイル名ストリングとそこにおける位置を示す整数の 2 要素が続く. 以下, include が深くなれば, このリストもそれにしたがって深くなる.

  【参考】ソースファイルの位置を示す整数の単位は, ファイルの先頭からの項, 行, 文字などが考えられる.

  【注意】2.5 版のプリプロセッサ仕様では, このリストの要素は単に WrappedTerm である.

- VP

  コンパイラオプション格納プールへのストリームである. このストリームに対するプロトコルは「プリプロセッサ作成ツール」の節 (441 頁) を参照のこと. なお, マージャが挿入されている.

- MESG

  メッセージ出力のためのストリームである. このストリームに対するプロトコルは「プリプロセッサ作成ツール」の節 (443 頁) を参照のこと.

- MESGnew,MESGold

  メッセージ出力のための差分ストリームである. このストリームに対するプロトコルは「プリプロセッサ作成ツール」の節 (443 頁) を参照のこと.

- Status

  パーサやプリプロセッサなどの処理の正常終了あるいは異常終了を通知するための変数である. normal は正常終了を意味し, abnormal は異常終了を意味する.

  【注意】表示メッセージにおいてエラー (プロトコルにおける error/2, error/3 メッセージ) を出力した場合は, 必ずこの変数を abnormal にすること.

D.3 パーサ, プリプロセッサ, アンパーサ, ベリファイアに共通な述語

Preprocessor:describe(^VP, ^MESGold, MESGnew)

このモジュールに有効なオプション設定を表示するための述語である. この述語により, 実際のオプション設定を確認することができる. セルフコンパイラの `!describe` コマンドで呼び出される.

D.4 パーサモジュールの作成方法

- 必須公開述語

  Parser:parse(DevIn, ^Out, ^VP, ^MESG, ^Status)

  構文解析を記述する.

  【注意】2.5 版では, parse 述語は 6 引数であったが, 3.0 版から 5 引数に変更された. なお, 2.5 版の parse/6 もそのまま使用可能である.

- オプショナルな公開述語

  Parser:sort(MessageX, MessageY, Small, Large Swapped)

  表示メッセージを位置情報によりソートするための公開述語である. (この述語の仕様は, プログラミング編 8 章 ユーティリティ 303頁を参照のこと) MessageX, MessageY は MESG ストリームを流れるメッセージであり, そこに含まれる位置情報からソートする. 位置情報が含まれていないメッセージも存在することに注意を要する. 位置情報が含まれていないメッセージは, 位置情報が含まれているメッセージ全てに先行するものとする.

  MessageX, MessageY から, 位置情報を取り出すには, **kl1_library_message: position(Message, ^Position)** という述語を呼び出せば良い. メッセージの詳細は, 後述する「ライブラリメッセージ出力」の節 (444 頁) を参照のこと.

  Parser:message(Mode, MESGin, ^VP*, ^MESGout)

  表示メッセージをモードによりどこまで表示するかを決めるフィルタである. 主に, 位置情報にパーサが付加した任意の項をどのように表示するかという処理を記述する. Mode は, セルフコンパイラの !message コマンド (操作編 3 章 コンパイラ 47頁を参照のこと) で指定されたアトム silent または verbose が具体化される. MESGin は, 表示メッセージのストリームであり, MESGout は表示メッセージのフィルタを通した出力である.

【参考】パーサ定義の枠組

```
:- module my_parser.
:- public parse/5.
:- public sort/5, message/4.    % オプショナルな公開述語

parse(In,Out,VP,MESG,Status) :- true |
    ....

sort(X,Y,Small,Large,Swapped) :- true |
    ....

message(Mode,MESGin,VP,MESGout) :- true |
    ....
```

D.5 プリプロセッサモジュールの作成方法

プリプロセッサモジュールは, 以下に示す公開述語を持つ必要がある.

Preprocessor:translate(In, ^Out, ^VP, ^MESG, ^Status)

言語の変換を記述する.

【注意】2.5 版では, translate 述語は 6 引数であったが, 3.0 版から 5 引数に変更された. なお, 2.5 版の translate/6 もそのまま使用可能である.

【注意】2.5 版では translate/2 という公開述語を定義する必要があったが, 3.0 版からは不要になった.

【参考】プリプロセッサ定義の枠組

```
:- module my_preproc.
:- public translate/5.

%% translate(In,Out) :- true | In = "abc", Out = "kl1". %% 3.0 版から不要

translate(In,Out,VP,MESG,Status) :- true |
    ....
```

D.6 アンパーサモジュールの作成方法

アンパーサモジュールには, 以下に示す公開述語が定義されていなければならない.

Unparser:unparse(In, ^DevOut, ^VP, ^MESG, ^Status)

In はそれまでのプリプロセッサで変換された wrap された項のリスト, Out は出力デバイスへのストリームである. バッファは挿入されていない.

【注意】2.5 版では, unparse 述語は 6 引数であったが, 3.0 版から 5 引数に変更された. なお, 2.5 版の unparse/6 もそのまま使用可能である.

【参考】アンパーサ定義の枠組

```
:- module my_unparser.
:- public unparse/5.

unparse(In,Out,VP,MESG,Status) :- true |
    ....
```

D.7 ベリファイアモジュールの作成方法

- 必須公開述語

  Verifier:verify(In, ^IFC, ^VP, ^MESG)

  In から入力される項のリストから, 静的な解析をおこない, その結果を MESG に出力する. ここで, IFC とはファイル間のチェックをおこなうためのストリームであり, ファイルをまたがるようなベリファイをおこなうための述語 (後述) inter_file_verify/3 に十分な情報を提供するためのものである. このストリームのプロトコル仕様は任意である. ただし, ファイル間チェックを行なわないならば必ず [] をバインドすること. そうでなければデッドロックする.

- オプショナルな公開述語

Verifier:inter_file_verify(IFCs, ^VP, ^MESG)

ファイルをまたがった静的なベリファイをおこなうための述語である. IFCs は [{**Messages, Sorter**}, **...**] という構造であり, Messages は, 上記の verify/4 の第2引数, Sorter は表示メッセージをソートするための比較述語 (sort/5) コードである.

このソートは, 引き数 Messages に位置情報が入っていた場合に, それに対する表示メッセージをソートするためのものである. 表示メッセージをソートするかしないかは, するとしてもどのようにソートするかは, この述語に任される. また, 与えられた比較述語の利用もこの述語の仕様による.

【注意】ベリファイアが対応する言語よりも上位のプリプロセッサが存在する可能性に注意 (位置情報の形式, 比較述語の仕様が異なることも考慮すること) のこと.

【参考】ベリファイア定義の枠組

```
:- module my_verifier.
:- public verify/4.
:- public inter_file_verify/3.    % オプショナルな公開述語

verify(In,IFC,VP,MESG) :- true |
    ....

inter_file_verify(IFCs,VP,MESG) :- true |
    ....
```

D.8 プリプロセッサ作成ツール

D.8.1 KL1 標準プリプロセッサ

モジュール kl1_standard_preprocessor には, ユーザーのプリプロセッサ開発の助けとなるように以下の述語が提供されている.

kl1_standard_preprocessor:parse_with_op_pool(DevIn, ^Out, ^OpPool, ^VP, ^MESG, ^Status)

KL1 の標準オペレータとは異なるパーサを作りたい時に用いる述語である. 引数 OpPool は入力デバイスからの項の読み込みにおけるバッファが用いるオペレータプールへのストリームである.

【例】KL1 の標準とは異なるオペレータを持つパーサの定義

```
parse(In,Out,VP,MESG,Status) :- true |
    kl1_standard_preprocessor:
        parse_with_op_pool(In,Out,MyOpPool,VP,MESG,Status),
    operator:empty_pool(MyOpPool0),    % 空のオペレータプールを生成する
    MyOpPool0 = [add_op(...), add_op(...), ... |MyOpPool].
    % 望みのオペレータを定義する
```

D.8.2 コンパイラオプション格納プール

コンパイラオプション格納プールは以下に示すメッセージを受け付ける. このプールへのストリームにはマージャが挿入されている.

get_atom_table(AtomTable*)

アトムテーブルへのストリームを返す. なおこのストリームにはマージャが挿入されている. (このストリームのプロトコルについては, プログラミング編 10 章 プログラム管理 386頁を参照のこと)

get_directory(Directory)

現在設定されているカレントディレクトリ (ストリング) を返す.

get_file_request(FileRequest*)

ファイルリクエストデバイスへのストリームを返す. デフォールトディレクトリの設定はこのファイルリクエストデバイスに対して行われているため, プリプロセッサ内でのソースファイル以外のファイルをオープンする場合にはこのストリームを用いること. (このストリームのプロトコルについては, プログラミング編 9.1 章 入出力資源の獲得 363頁を参照のこと)

get_module_table(ModuleTable*)

モジュールテーブルへのストリームを返す. なおこのストリームにはマージャが挿入されている. (このストリームのプロトコルについては, プログラミング編 10 章 プログラム管理 387頁を参照のこと)

get_package(DefaultPackage)

現在設定されているデフォールトパッケージ (アトム) を返す.

input_language(InputLanguage)

現在コンパイル中のファイルの言語名 (ストリング) を返す.

source_filename(FileName)

現在コンパイル中のファイル名 (ストリング) を返す.

add_dependency(FileName,DateTime)

現在コンパイル中のモジュールが依存するファイルのファイル名とファイルの更新日時 (日付に関する整数とその日のミリ秒時間を対とする 2 要素ベクタ) を設定する. この設定をおこなうと, このコンパイルで生成されたモジュールに対し, セルフコンパイラの `!make` コマンドにおいてこのファイルの更新日時をチェックするようになる.

do(Stream*)

Stream にメッセージを流すと連続して処理される. このストリームには, マージャが挿入されている.

get(Key,Status)

プリプロセッサ用の変数プールへのアクセスプロトコルである. Key は KL1 の任意の項 (変数を除く) である. Status には, 環境変数 compile:language_options に設定された値, あるいはインタラクティブコンパイルの `set` コマンドで設定された値が normal(Values) という項の Values に返される. 対応するキーが存在しない場合には, non_existent が返される.

【注意】 2.5 版では, Key は 16 ビットストリングであったが, 3.0 版からは任意の基底項が可能になった.

D.8.3 ライブラリメッセージ出力

コンパイラに代表される PIMOS のライブラリで利用されているプロセスである. メッセージは自動的に shell の標準出力 (このストリームのプロトコルについては, プログラミング編 9.4 章 標準入出力 377 頁を参照のこと) あるいは標準メッセージ出力 (同じ章の 383 頁を参照のこと) に振り分けられる. また, エラー, 警告のカウントもおこなっておりその出力も可能である.

以下に, このプロセスが受け付けるメッセージを示す, これらのメッセージ中に表れる変数で, FormatString はバッファユーティリティにおける書式付き出力 putf の書式と同じであり, Args はその書式における引数リストである.

info(FormatString,Args)

シェルの標準メッセージ出力へそのままの形でメッセージが出力される.

info(FormatString,Args,PosInfo)

シェルの標準メッセージ出力へ位置情報を出力した後は, info/2 と同じ.

message(FormatString,Args)

シェルの標準出力へそのままの形でメッセージが出力される.

message(FormatString,Args,PosInfgo)

シェルの標準出力へ位置情報を出力した後は, message/2 と同じ.

warning(FormatString,Args)

"!WARNING!" という文字列が行頭に付加された形のメッセージが標準出力に出力される。なお、このメッセージはワーニングとしてカウントされる.

warning(FormatString,Args,PosInfo)

"!WARNING!" という文字列の直後に位置情報を出力した後は, warning/2 と同じ.

error(FormatString,Args)

"!ERROR!" という文字列が行頭に付加された形のメッセージが標準出力に出力される。なお、このメッセージはエラーとしてカウントされる.

error(FormatString,Args,PosInfo)

"!ERROR!" という文字列の直後に位置情報を出力した後は, error/2 と同じ.

syntax_error(ErrorInfo)

シンタックスエラー報告のためのメッセージである。標準出力へ出力される.

syntax_error(ErrorInfo,PosInfo)

位置情報を出力した後は, syntax_error/1 と同じ.

stdout(Messages)

シェルの標準出力に対する do(Messages) と同じ.

stdout(Messages,PosInfo)

位置情報を出力した後は, stdout/1 と同じ.

stdmsg(Messages)

シェルの標準メッセージ出力に対する do(Messages) と同じ.

stdmsg(Messages,PosInfo)

位置情報を出力した後は, stdmsg/1 と同じ.

flush

標準出力および標準メッセージ出力ともに flush される. 基本的にバッファに対する flush/1 と同じ.

【注意】PIMOS 3.0 版から表示メッセージをソートするようになったため, 実質的な効果は期待できない.

また, パーサモジュールにおける sort/5 述語を定義するために, 以下の述語も用意されている.

kl1_library_message:position(Message,^Position)

ライブラリメッセージ出力に対するメッセージから, 位置情報を取り出すための述語である. 引数 Message はメッセージであり, Position がそのメッセージに含まれる位置 (整数) である. 位置情報が含まれていないメッセージの場合は, アトム [ ] が返される.

D.9 プログラム例

【例】プリプロセッサのテストスタブ

```
% InFile : 入力ファイルへのストリーム
% OutFile : 出力ファイルへのストリーム
test(InFile,OutFile) :- true |
    kl1_library_message:message(MESG0),
    kl1_library_variable_pool:compile(VP0,MESG0,MESG1),
    merge({VP1,VP2,VP3},VP0),
    module_table:cache_table([get_code(my_parser,sort,5,SortCode)]),
    sorter:sort(MESG2,SortCode,MESG1),
    merge({MESG3,MESG4,MESG5},MESG2),
    my_parser:parse(InFile,List0,VP1,MESG3,ParseStatus),
    my_preproc:translate(List0,List1,VP2,MESG4,PreproStatus),
    my_unparser:unparse(List1,OutFile,VP3,MESG5,UnparseStatus).
```

【注意】正確には, モジュール kl1_library_variable_pool は, プリプロセッサでいうところのコンパイラオプション格納プールではないため, do/1, add_dependency/2, source_filename/1, input_language/1 のメッセージは受け付けない.

付録 E

プロトコル定義

E.1 概要

ユーザプログラムの PIMOS に対する要求はストリームにメッセージを送ることにより伝えられる. PIMOS ではこのメッセージのプロトコルを定義するために専用の定義言語 (以下プロトコル定義言語という) を用いており, PIMOS の一部はこの言語によるプロトコル定義から自動生成している.

プロトコル定義言語について簡単に説明したのち, 実際のプロトコル定義を示す.

E.2 プロトコル定義言語

E.2.1 文法

プロトコル定義言語の文法を以下に示す.

```
 definition ::= name[(formals)] = [inheritance] body.

inheritance ::= {super &}
      super ::= invocation
       body ::= [sign] (invocation | {enumeration}) [error]
 invocation ::= name [(actuals)] | parameter
enumeration ::= pattern [error] {; pattern [error]}
    pattern ::= atom [(actuals)]
    formals ::= parameter {, parameter}
    actuals ::= body {, body}
  parameter ::= name
       sign ::= + | -
      error ::= # [atom] | ## [atom]

  |      どちらか 1 つを選択する
{ }    0 個以上繰り返す
[ ]    省略可能
```

E.2.2 システム定義プロトコル

atom, integer, string, code, module, ground, any, list(p), vector(p), vector([p1,p2,···,pN]), stream(p) をシステム定義のプロトコルとして用意する.

atom, integer, string, code, module は, そのプロトコルが atom, integer, string, code, module の各データタイプであること, ground はそのデータが完全に具体化されていることを意味する. any は変

数も含めて任意のデータであることを示す.

stream(p), list(p) は, そのプロトコルがそれぞれストリームあるいはリストであることを意味し, かつその要素は, p で指定されたプロトコルに従うことを意味する. ベクタに関しては vector(p), vector([p1,p2,···,pN]) の 2 通りの記法がある. vector(p) はサイズ不定のベクタで, 各要素のプロトコルが p であるという定義である. また vector([p1,p2,···,pN]) はサイズ N のベクタで, その各要素のプロトコルはそれぞれ p1, p2, ···, pN であるという定義である.

E.2.3 文法の説明

上記の文法を簡単に説明する.

E.2.3.1 プロトコル定義

プロトコルは,

プロトコル名 = プロトコル定義

という形式で定義する. プロトコル定義としては, メッセージパターンあるいは参照したいプロトコル名を記述できる. メッセージパターンは, { } でくくって記述する. メッセージパターンの間の区切りには, ';' を使用する.

E.2.3.2 継承

プロトコル名を指定することにより, プロトコルの多重継承ができる.

E.2.3.3 符号

プロトコルの値の設定者をしめすために符号をもちいる. 符号の既定値は '+' である. '−' を指定するとそのプロトコルを参照しているプロトコルと設定者が異なるプロトコルとなる.

E.2.3.4 パラメータ付きプロトコル定義

name(formals) = [inheritance] body.

の形式でパラメータつきのプロトコルをユーザが定義することができる. このプロトコルは, 参照時にパラメータに値を受け取る. 右辺にこの値を渡して展開したものが実際の定義となる.

E.2.3.5 エラー情報指定

プロトコル定義のあとに, ユーザプログラムのそのプロトコルに間違いがあったときに保護フィルタが出力するエラー情報を指定できる. この指定には 2 レベルありそれ以前に指定されていた情報にプラスされるエラー情報 (#[atom]) とそれ以前に指定されていた情報をリセットして新たに指定されるエラー情報 (##[atom]) がある.

エラー情報が省略されて '#' あるいは '##' のみ指定したときは, その定義のプロトコル名あるいはパターンの 'アトム / アリティ' を規定値として設定する.

E.3 プロトコル定義の読み方

PIMOS で提供するメッセージプロトコル定義のはじめの部分を見ながら読み方の説明をする.

```
genreq_filter = stream(genreq_command).                (1)

genreq_command = ask_task_command & {                  (2)
        execute(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# ,initrsc# ,-task)# ;
        execute(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# , -task)# ;
        execute_remote(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#,initrsc#, fepnodeno#,ruser#,rpassword#,-task)# ;
        execute_remote(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#, fepnodeno#,ruser#,rpassword#,-task)# ;
        execute_super(modulename#,predname#,arity#,argv#,taskname#,
                minprio#,maxprio#,tag#,initrsc#,-task)#;
        execute_super(modulename#,predname#,arity#, argv#, taskname#,
                minprio#, maxprio#, tag#, -task)# ;
        execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#,initrsc#,userid#,-task)# ;
        execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#, user#, -task)#;
        execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#,initrsc#,user#,fepnodeno#,ruser#,rpassword#,-task)# ;
        execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
                tag#, user#, fepnodeno#,ruser#,rpassword#,-task)# ;

        suicide(-status)# ;
        abort(rscpath# , -status)# ;
        number_of_feps(-status(integer))# ;

        setenv(envname# , value#, -status)# ;
        getenv(envname# , -status(value))# ;
        getenv(-status(envs))# ;
        unsetenv(envname, -status)# ;

        window(-rscstream(winreq_command)##window_request)# ;
        file(-rscstream(filereq_command)##file_request)# ;
        timer(-rscstream(timerreq_command)##timer_request)# ;
        string_io(-rscstream(strioreq_command)##string_io)# ;
        module(-rscstream(modulereq_command)##module_request)# ;
        atom(-rscstream(atomreq_command)##atom_request)#;
        user(-rscstream(userreq_command)##user_request)#

}.

ask_task_command = {                                   (3)
        id(-status(integer))# ;
        absolute_id(-status(rscpath))# ;
        status(rscpath# , rsckind# , verbosity# , -status(rscinfo))#
}.
```

(1) で示しているのがトップレベルのプロトコルの定義である. genreq_filter は, 資源木のトップレベルである general_request_device へのストリームを定義している. このストリームを介して general_request_device にメッセージを送ることができる. 右辺の stream(genreq_command) は, このストリームに送ることのできるメッセージは, genreq_command というプロトコル名で定義されたものであることをしめす.

genreq_command の定義は, ask_task_command というプロトコル名の定義を継承している. { } 内に記述されている構造と, ask_task_command として定義された 3 つの構造が実際のメッセージである. メッセージはメッセージ名 (アトムで記述する) と各引数に対するプロトコルの定義で定義する. メッセージは, 引数を持たないアトムでもよい.

メッセージの例として

- setenv(envname# , value# , -status)#
- window(-rscstream(winreq_command)##window_request)#

の 2 つを考える.

setenv(envname# , value# , -status)#

は, 環境変数に値を設定するメッセージで, 3 つの引数をもつ. envname, value に値を設定して送ると, status に値が返ることが符号により定義してある. envname, value, status については終わりのほうに, 次のように定義してある.

```
envname = ground.
value = ground.

status = {
        normal ;
        abnormal ;
        aborted
}.
```

envname, value の定義はどちらも ground であるので, 設定する値は, ground なデータであればよいことがわかる. status は normal, abnormal, aborted のいずれかのアトムと定義されている.

setenv(envname# , value# , -status)#

の # はエラー情報を加えることを指定している. エラー情報の指定がそれぞれ省略してあるので規定値として 'envname', 'value','setenv/3' が設定される.

window(-rscstream(winreq_command)##window_request)#

は, window_request_stream を獲得するためのものである. このメッセージを送ると引数に値が返される. この値は rscstream(winreq_command) と定義されている. ユーザ定義のパラメタつきプロトコルである. rscstream( ) の定義はつぎのようである.

```
rscstream(p) = {
        normal(-stream(p), -abort, attention) ;
        abnormal ;
        aborted
}.
```

rscstream(p) の 'p' がパラメータである. 参照するときにパラメータに値をわたす. この例では winreq_command である. そこで, window(-rscstream(winreq_command)) に返される値は,

表 E.1: メッセージプロトコルと送り先デバイスの対応

メッセージの送り先	プロトコル名
general_request_device	genreq_command
window_request_device	winreq_command
file_request_device	filereq_command
timer_request_device	timerreq_command
string_io_request_device	strioreq_command
module_request_device	modulereq_command
atom_request_device	atomreq_command
window_device	window_command
file_device	rfile_command/wfile_command/dir_command
timer_device	timer_command
string_io_device	strio_command
module_device	module_command
atom_device	atom_command
task	control_command/report_command

- normal(-stream(winreq_command), -abort, attention)
- abnormal
- aborted

のいずれかということになる.

normal( ) の第一引数は,window_request_stream , 第二引数はアボートライン, 第三引数はアテンションラインである. window_request_stream に流せるメッセージは, winreq_command というプロトコル名で定義されているものである. 第一引数, 第二引数に '–' がついているのは, normal( ) が PIMOS からユーザに返されるのに対し, その引数の window_request_stream およびアボートラインは, ユーザから PIMOS にむけて値の設定されるものであることを示している.

window(-rscstream(winreq_command)##window_request)#

では, 獲得した window_request_stream に対して, それまでに設定されていた情報をすべてクリアして 'window_request' というエラー情報を設定する. このようにして, 資源木のレベルごとにエラー情報を与えるようにしている. このようにして指定されたエラー情報は, 保護フィルタがメッセージプロトコルの誤りを発見したときにレイズして報告される.

このようにして, プロトコル定義を読むことにより正確なメッセージプロトコルがわかる.

メッセージの定義を調べるときの参考に, メッセージの送り先と, この定義中でのプロトコル名との対応を表 E.1に記す.

E.4 プロトコル定義

```
%%%
```

```
%         (2) 環境変数の問い合わせ及び設定
%         (3) 各デバイスへの request stream の獲得
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

genreq_filter = stream(genreq_command).

genreq_command = ask_task_command & {
        execute(execode# , argv# , initrsc_or_list# ,-task)# ;
        execute_super(modulename# , predname#,arity#,argv# ,
                      initrsc_or_list# ,-task)# ;
        profile(execode# , argv# , initrsc_or_list# ,-task)# ;
        execute(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# ,initrsc_or_list# ,-task)# ;
        execute(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# , -task)# ;
        profile(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# ,initrsc# ,-task)# ;
        profile(execode# , argv# , taskname# , minprio# , maxprio# ,
                tag# , -task)# ;
  execute_remote(execode#, argv#, taskname#, minprio#, maxprio#,
                 tag#,initrsc#,
                 fepnodeno#,ruser#,rpassword#,-task)# ;
  execute_remote(execode#, argv#, taskname#, minprio#, maxprio#,
                 tag#, fepnodeno#,ruser#,rpassword#,-task)# ;
  execute_super(modulename#,predname#,arity#,argv#,taskname#,
                minprio#,maxprio#,tag#,initrsc#,-task)#;
  execute_super(modulename#,predname#,arity#, argv#, taskname#,
                minprio#, maxprio#, tag#, -task)# ;
  execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
               tag#,initrsc#,userid#,-task)# ;
  execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
               tag#, user#, -task)#;
  execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
               tag#,initrsc#,user#,
               fepnodeno#,ruser#,rpassword#,-task)# ;
  execute_user(execode#, argv#, taskname#, minprio#, maxprio#,
               tag#, user#,
               fepnodeno#,ruser#,rpassword#,-task)# ;
  execute_user_with_proc(execode#, argv#, taskname#, minprio#, maxprio#,
               integer,integer,tag#,initrsc#,userid#,-task)# ;

        suicide(-status)# ;
        abort(rscpath# , -status)# ;
        number_of_feps(-status(integer))# ;

        setenv(envname# , value#, -status)# ;
        getenv(envname# , -status(value))# ;
        getenv(-status(envs))# ;
        unsetenv(envname, -status)# ;
```

```
        get_puid(-status(uid))# ;
        connection(-status(node_name))#;

        register(atom#, vector([uid,gid,integer]),
                 -register_status##register)# ;
        request(atom#, any, -status_any0##request)# ;
        request_anyway(atom#, any, -status_any0##request_anyway)# ;

        window(any)# ;
        file(any)# ;
        timer(any)# ;
        string_io(any)# ;
        module(any)# ;
        atom(any)# ;
        user(any)#
}.

status_any0 = {abnormal#; aborted#; normal(any)#}.

initrsc_or_list = vector([integer,integer]) &
                  list(task_init_command).

task_init_command =
     { resource(integer,integer);
       stdio(list(stdio_param),-status(integer,list(stdio_result)));
       io_node(ground);
       remote(string,string);
       uid(integer);
       user(string,string);
       name(string);
       node(integer,integer);
       prio(integer,integer);
       tag(integer)}.

stdio_param = { in(stdio_param_body)#;
                out(stdio_param_body)#;
                inter(stdio_param_body)#;
                in_out(stdio_param_body)#;
                in_inter(stdio_param_body)#;
                out_inter(stdio_param_body)#;
                in_out_inter(stdio_param_body)# }.

stdio_param_body = vector([atom,ground,-ground]).

stdio_result = { in(status)#;
                 out(status)#;
                 inter(status)#;
                 in_out(status)#;
```

```
                in_inter(status)#;
                out_inter(status)#;
                in_out_inter(status)# }.

register_att = { []#;
                 attention(register_att)#}.

register_command =
     { get_request(code,ground,-request_out)# }.

request_out =
     {general_request(-stream(server_control)##server_control,
                      any, ground, ground,-ground,any);
      standard_io(stream(std_control)##std_control,atom,ground);
      abnormal;
      aborted}.

register_status = {abnormal;
                   aborted;
                   normal(-stream(register_command),-any,register_att)}.

server_control = {add_resource(atom,ground,-server_control_status)#;
                  change_comment(ground,-status);
                  absolute_id(-status(list(integer)))}.

server_control_status =
     {abnormal; aborted; normal(-stream(server_control),any)#}.

std_control =
    {open(vector([atom,ground,-ground]),
          {switch(-stream(server_control),any)#},
          {io(-stream(server_control),any)})#;
     request(any,integer,any,-ground)#;
     make_branch(stream(std_subcontrol),any,ground,ground)#}.

std_subcontrol =
    {open(vector([atom,-ground]),
          {switch(-stream(server_control),any)#},
          {io(-stream(server_control),any)})#;
     request(any,integer,any,-ground)#;
     make_branch(stream(std_subcontrol),any,ground,ground)#}.


ask_task_command = {
        id(-status(integer))# ;
        absolute_id(-status(rscpath))# ;
        status(rscpath# , rsckind# , verbosity# , -status(rscinfo))#
}.
```

```
task = {
        normal( -control##control_stream, -report##report_stream ) ;
        abnormal ;
        aborted ;
        disconnected
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  アボート, アテンションラインに対するメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

aa_command = {
        reset(-status(-abort, attention# , msgid# ))# ;
        next_attention( -status(attention))# ;
        resend( msgid# , -status)# ;
        cancel( msgid# , -status)#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Control Stream のメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


control = stream(control_command).

control_command = ask_task_command & aa_command & {
        stop ;
        allow_resource_report ;
        statistics ;
        profile ;
        abort(rscpath# , -status)# ;
        abort(-status)# ;
        add_resource(hrsc# ,lrsc# )#
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Report Stream のメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

report = -stream(report_command).

report_command = {
```

```
        statistics_started ;
        resource_report_allowed ;
        resource_added ;
        resource_low ;
        aborted ;
        terminated ;
        exception(exckind, excinfo, -ncode# , -nargv# )# ;
        statistics(hstat,lstat) ;
        return_resource(hrsc,lrsc) ;
        profiling_information(profile_node, profile_type,
                              profile_depth, profile_data) ;
        profiling_started ;
        profiling_rejected ;
        profiling_completed ;
        stopped( -reason_stream# , -cancel# )#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% windwow に対するメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


winreq_filter = -rscstream(winreq_command).

winreq_command = aa_command & {
        create( -rscstream(window_command)##window)# ;
        create( buffername# , -rscstream(window_command)##window)# ;
        get_max_size( x# , y# , pathname# , -status(charnum, linenum))#
}.

window_command = input_command & output_command & aa_command & {
        beep( -status)# ;

        set_size( winsize# , -status)# ;
        set_position( winpos# , -status)# ;
        set_title( title# , -status)# ;
        reshape( winpos# , winsize# , -status)# ;
        set_font( pathname# , -status)# ;
        select_buffer( buffername# , -status)# ;

        activate( -status)# ;
        deactivate( -status)# ;
        show( -status)# ;
        hide( -status)# ;
        clear( -status)# ;
        get_char_size( -status(charnum, linenum))# ;
```

```
        get_position( -status(x,y))# ;
        get_title( -status(title))# ;
        get_font( -status(pathname))# ;
        get_buffer_name( -status(buffername))#
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% file に対するメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

filereq_filter = -rscstream(filereq_command).


filereq_command = aa_command & {
        open( pathname# ,
        {read(-rscstream(rfile_command)##read_file)# ;
                write(-rscstream(wfile_command)##write_file)# ;
                append(-rscstream(wfile_command)##append_file)# }#
         file_mode )# ;
        directory( pathname# , -rscstream(dir_command)##directory)# ;
        current_directory(-status(string))# ;
        set_current_directory(string#pathname, -status)#
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% file に対するメッセージ共通部
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cfile_command = {
        pathname(-status(pathname))# ;
        attributes(-file_attributes)#
}.

file_attributes = {
        normal(owner,updateuser,cdate,ctime,mdate,mtime,
               linkcnt,fsize,rsize) ;
        abnormal ;
        aborted ;
        disconnected
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 入力 file に対するのメッセージ
```

```
%
%%%
```

```
        get_date_time( -status(date_count, time_count))# ;
        where( -status(timer_id))# ;
        set_time_unit( time_unit#, -status)# ;
        get_time_unit( -status(time_unit)#)
}.

time_unit = { sec ; msec ; msec16 ; usec }.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% string_io に対するのメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


string_io_req_filter = -rscstream(strioreq_command).
strioreq_command = aa_command & {
        create( devicename# , -rscstream(strio_command)##string_io)#
}.


strio_command = aa_command & {
        io( io_command# , tofepstring# , -status(fromfepstring))#
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% module に対するメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


modulereq_filter = -rscstream(modulereq_command).
modulereq_command = aa_command & {
        create( -rscstream(module_command)##module_device)#
}.


module_command = aa_command & {
        define_package(packagename# , packagename# ,
                        protection# , -retrieval_status)# ;
        set_package_owner(packagename# , owner_info# , -retrieval_status)# ;
        protect_package(packagename# , protection# , -retrieval_status)# ;
        get_package_info(packagename# , -retrieval_status(protectionInfo))# ;
        get_super_package(packagename# , -retrieval_status(packagename))# ;
        get_all_packages(-packageInfoList)# ;
        remove_package(packagename# , -retrieval_status_1)# ;
        get_module_info(packagename# , modulename# , -module_status_1)# ;
        get_module(packagename# , modulename# , -module_status_2)# ;
```

```
        get_module_in_the_package(packagename#,
                                  modulename#, -module_status_2)# ;
        get_all_modules(packagename# , -module_status_3)# ;
        get_all_modules(-module_status_4)# ;
        get_code(packagename# , modulename# , predname# , arity# ,
                 -module_status_5 )# ;
        update_module(packagename# , modulename# , module#update_module,
                      -retrieval_status_1 )# ;
        remove_module(packagename# , modulename# , -retrieval_status_1)
}.

packageInfoList = {
        normal( list( packageInfo ) ) ;
        protection_violation ;
        abnormal }.

owner_info = vector([uid,gid]).

packageInfo = vector( [ packagename, packageInfo1, packageInfo2 ] ).
packageInfo1 = retrieval_status( packagename ).
packageInfo2 = retrieval_status( ground ).
packageInfo3 = vector( [ packagename, module_status_3 ] ).

module_status_1 = {
        normal( packagename, module, moduleSize ) ;
        non_existent ;
        protection_violation ;
        abnormal }.

module_status_2 = {
        normal( module ) ;
        non_existent ;
        protection_violation ;
        abnormal }.

module_status_3 = {
        normal( moduleInfoList ) ;
        non_existent ;
        protection_violation ;

        abnormal }.

module_status_4 = {
        normal( list( packageInfo3 ) ) ;
        protection_violation ;
        abnormal }.

module_status_5 = {
        normal( code ) ;
```

```
        non_existent ;
        non_existent( ground ) ;
        protection_violation ;
        abnormal }.

retrieval_status = {
        normal ;
        protection_violation ;
        abnormal ;
        aborted ;
        non_existent }.

retrieval_status(x) = {
        normal(x) ;
        protection_violation ;
        abnormal ;
        aborted ;
        non_existent }.

retrieval_status_1 = {
        normal ;
        protection_violation ;
        abnormal ;
        aborted ;
        non_existent( ground ) }.

protectionInfo = vector( [ ground, protection ]) .
moduleInfoList = list( moduleInfo ).
moduleInfo = vector( [ modulename, module, size ] ).
moduleSize = integer.
userid = ground.
groupid = ground.
size = integer.
protection = integer.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% atom に対するメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


atomreq_filter = -rscstream(atomreq_command).
atomreq_command = aa_command & {
        create( -rscstream(atom_command)##atom_device)#
}.
```

```
atom_command = aa_command & {
        get_atom(atomname# , -retrieval_status(atom,dkey))# ;
        get_atom_name(atom# , -retrieval_status(atomname,dkey))# ;
        intern(atomname# , -status(atom))# ;
        get_all_atoms(-status(atominfo))#
}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% user device に対するのメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

userreq_filter = -rscstream(userreq_command).
userreq_command = aa_command & {
        create({user(-rscstream(user_command)##user_device);
                group(-rscstream(group_command)##group_device);
                login(-rscstream(login_command)##login_device)}#user_mode)# }.


user_command = aa_command & {
        set_up(list(user_all_info),-status)#;
        authenticate(user_name#,password#,-status)#;
        get_uid(user_name#,-status(uid))#;
        get_info(uid#,-status(user_info))#;
        status(uid#,-status(change_count))#;
        get_all_info(-status(list(user_all_info)))#;
        change_password(uid#,old_password#,new_password#,-status)#
}.

old_password = password.
new_password = password.

user_name = string.
password = string.
uid = integer.
priority = integer.
directory_name = string & {*}.
entry_point = {':'({'::'(packagename,modulename)},predicate)}.

predicate = atom & vector(ground).
comment_list = list(string).
pe_number = integer & {*}.

change_count = integer.

user_all_info = vector([user_name,password,uid,priority,
                    directory_name,pe_number,entry_point,comment_list]).
user_info = vector([user_name,priority,directory_name,
```

```
                    pe_number,entry_point,comment_list]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% group device へのメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

group_command = aa_command & {
        set_up(list(group_info),-status)#;
        get_gid(group_name#,-status(gid))#;
        get_gids(uid#,-status(list(gid)))#;
        get_users(gid#,-status(group_members))#;
        get_all_info(-status(list(group_info)))#
}.

gid = integer.
group_name = string.
group_members = list(uid).
group_info = vector([group_name,gid,group_members,comment_list]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% login device へのメッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

login_command = aa_command & {
        login(user_name#,login_info#,-status(lid))#;
        logout(lid#,-status)#;
        get_info(lid#,-status(login_info))#;
        get_login(user_name#,-status(list(lid)))#;
        get_all_info(-status(list(vector([user_name,lid,login_info]))))#
}.

lid = integer.
login_info = ground.
node_name = fepnodeno.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% device への共通メッセージ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

input_command = {
        getb( blocksize# , -eofstatus(string))# ;
```

```
        ungetb( string#ungetb , -status)# ;
        prompt( string#prompt, -status)#
}.

output_command = {
        putb( string#put_data, -status)# ;
        flush( -status)#
}.



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% メッセージの各要素をチェックする共通部
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rscpath = {[rscid | rscpath];
            *               ;
            []
         }.
rscid = integer & { * }.
rsckind = rscname & {
        * }.
rscname = atom.
verbosity = {
        brief ;
        detail
}.
reason_stream = stream(atom).
rscinfo = list(rscinfo2).
rscinfo2 = brief_info & other_brief_info & detail_info & other_detail_info.
brief_info = vector([rscid, rscname, ground, bchildren]).
other_brief_info = vector([rscid,bchildren]).
both_brief_info = brief_info & other_brief_info.
detail_info = vector([rscid, ground, task_status, limit, consume, dchildren]).
other_detail_info = vector([rscid,dchildren]).
both_detail_info = detail_info & other_detail_info.
task_status = vector([atom, stops]).
stops = list(stopreason).
stopreason = {stop(atom)}.
binfo2 = brief_info & other_brief_info.
dinfo2 = detail_info & other_detail_info.
bchildren = list(binfo2).
dchildren = list(dinfo2).
msgid = integer.
winpos = {
        mouse ;
        at(x,y)
```

```
}.
winsize = {
        mouse ;
        char(charnum, linenum)
}.
now = {wake_up; aborted}.
attention = {attention ; []}.

atominfo = list(atominfo2).
atominfo2 = vector([atom,string]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 基本プロトコル
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


excargv = any.
cancel = ground.
hstat = integer.
lstat = integer.
nargv = any.
ncode = any.
exccode = any.
excinfo = vector(any).
exckind = atom.
term1 = any.
term2 = any.
rtype = any.
info = any.
execode = code.
argv = vector(any).
taskname = string.
minprio = integer.
maxprio = integer.
tag = integer.
hrsc = integer.
lrsc = integer.

envname = ground.
value = ground.

abort = any.

buffername = string.
x = integer.
y = integer.
pathname = string.
```

```
devicename = string.

fepno = integer.
blocksize = integer.
charnum = integer.
linenum = integer.
title = string.
filename = string.
wildcard = string.
owner = string.
updateuser = string.
cdate = string.
ctime = string.
mdate = string.
mtime = string.
linkcnt = integer.
fsize = integer.
rsize = integer.
count = integer.
date_count = integer.
time_count = integer.
io_command = integer.
tofepstring = string.
fromfepstring = string.
modulename = atom.
packagename = atom.
predname = atom.
arity = integer.
dkey = atom & string.
atomname = string.

limit = vector([hrsc,lrsc]).
consume = vector([hrsc,lrsc]) & {unknown}.
initrsc = vector([hrsc,lrsc]).
name = string.
eof = {yes ;
       no
}.

envs = list(env).

env = vector([envname,value]).

user = integer.
ruser = string.
rpassword = string.
fepnodeno = integer & string.
nodeno = string.
```

```
profile_node = vector(integer).
profile_type = integer.
profile_depth = integer.
profile_data = ground.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% パラメタライズしたプロトコル
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rscstream(p) = {
        normal(-stream(p), -abort, attention) ;
        abnormal ;
        aborted ;
        disconnected
}.

status = {
        normal ;
        abnormal ;
        aborted ;
        disconnected
}.

status(p) = {
        normal(p) ;
        abnormal ;
        aborted ;
        disconnected
}.

status(x,y) = {
        normal(x,y) ;
        abnormal ;
        aborted ;
        disconnected
}.
status(x,y,z) = {
        normal(x,y,z) ;
        abnormal ;
        aborted ;
        disconnected
}.

retrieval_status(x,y) = {
        normal(x) ;
        protection_violation ;
```

```
        abnormal ;
        aborted ;
        non_existent(y)
}.
eofstatus(p) = {
        normal(p) ;
        abnormal ;
        aborted ;
        end_of_file ;
        disconnected
}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 標準入出力
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

stdwin_filter = stream(window_command).
stdrfile_filter = stream(rfile_command).
stdwfile_filter = stream(wfile_command).
stdafile_filter = stream(wfile_command).

stdstring_io_filter = stream(strio_command).

std_in_filter = stream( std_in_command ).
std_out_filter = stream( std_out_command ).
std_inter_filter = stream( std_inter_command ).
std_io_filter = stream( std_io_command ).
std_switch_filter = stream( std_switch_command ).

std_in_command = aa_command & input_command .
std_out_command = aa_command & output_command .
std_inter_command = aa_command & interaction_command
    & stdio_special_command .
std_io_command = aa_command & input_command & output_command
    & interaction_command & stdio_special_command .
std_switch_command = {
    attach( switch_no#, -attach_status#, -detect# )# ;
    attach( -attach_status#, -detect# )# ;
    detach( switch_no#, -detach_status# )# ;
    detach( -detach_status# )# ;
    kill( switch_no_with_all#, -kill_status# )# ;
    kill( -kill_status# )#
}.

switch_no = integer.
switch_no_with_all = integer & { all }.
```

```
attach_status = atom.
detect = atom.
detach_status = atom.
kill_status = atom.

interaction_command = {
    igetb( blocksize#, -eofstatus( string ) )# ;
    iungetb( string#ungetb , -status)# ;
    iprompt( string#prompt, -status )# ;
    iputb( string#put_data, -status )# ;
    iflush( -status )#
}.

stdio_special_command = {
    switch( -status )# ;
    am_I_there( -status)#
}.
```

付録 F

サーバプログラムの記述例

F.1 概要

サーバプログラムの記述例を示す.

ここでは, ホワイトボードと呼ばれるユーザが使うデータプールをサービスするサーバを作成する. ホワイトボードの機能は次の通りである.

- ホワイトボードはユーザ単位, グループ単位で存在する.
- 登録できるデータはアトムをキーとした文字列とする.
- 各ホワイトボードに対してキーの一覧を見ることはできるが, 基本的にはデータを入れ, 出す操作のみサポートする

ホワイトボードの使用例を以下に示す.

```
shoen:raise(pimos_tag#task, general_request, GR),
GR = [request(whiteboard, Cancel, WRResult)],

%   サーバ名は whiteboard である.
%

Result = normal(WSStream),

%   Result として aborted も abnormal もくることがあるが
%   ここでは省略
%   WSStream にリクエストデバイスが獲得できたことになる

WSStream = [user(UResult)],

%   ここではユーザ単位のホワイトボードを利用する.

UResult = normal(UStream),

%   獲得成功

UStream = [put(example,"ここは文字列を書きます",Status)],

%   一つ登録した
```

```
%   Status == normal となるでしょう
```

F.2 サーバプログラム例

```
:- module whiteboard_server.
:- with_macro pimos.
:- public go/0, filter/3.

%   呼びだし口は whiteboard_server:go である.
%   保護フィルタコードのために filter/3 も
%   public 宣言する.

go:-
    true|
        shoen:raise(pimos_tag#task,general_request,GR),
%
%       サーバ名:        whiteboard
%       オーナ:          特権ユーザ
%       グループオーナ: 特権グループ
%       保護モード:      すべてのユーザが利用可能
%
%             サーバを登録する
%
        GR = [register(whiteboard,{0,0,2#"111"},Res)],
        wait_register(Res).

wait_register(normal(Control,Abort,Attention)):-
    true|
%
%    登録に成功したので
%        Control:   制御ストリーム
%        Abort:     登録を緊急に削除することを通知するための変数
%        Attention: 異常通知用変数である.  ここでは用いない.
%%%   ignore reset Attention
%
%       ホワイトボードプロセス群を保持するためのプールである.
%
        pool:keyed_set(Pool),
        merge(PoolIn,Pool),
%
%       保護フィルタコードを取り出す
%
        predicate_to_code(module#whiteboard_server,filter,3,Code),
%
%       ユーザの処理受け付けを開始する.
%       ユーザの利用時, タスクの資源木に "whiteboard" として
%       登録される.
```

```
%
        Control = [get_request(Code,"whiteboard",Res)|Next],
        wait_get_request(Res,Next,Code,Abort,PoolIn).
wait_register(abnormal):- true | true.
wait_register(aborted):- true | true.

%
%       ユーザからの処理要求がひとつ来た.
%

wait_get_request(general_request(Control,Abort,Attention,
                                                UInfo,FilterInfo,Var),
                                    Next,Code,ServerAbort,Pool):-
    true|
%
%       まず次の処理受け付けを可能にする
%
        Next = [get_request(Code,"whiteboard",Res)|NextN],
        wait_get_request(Res,NextN,Code,ServerAbort,Pool1),
%
%       ユーザの情報 UInfo からユーザ ID と属するグループの
%       ID の列，およびユーザが処理要求を行なったノード番号
%       (全荘園を通じた絶対的なノード番号) を獲得する.
%
        set_vector_element(UInfo,0,Uid,0,UInfo1),
        set_vector_element(UInfo1,1,Gids,0,_),
%
%       各ユーザが利用できるグループ ID をユーザタスク内で
%       チェックするために保護フィルタ側にユーザが属する
%       グループの ID 列を渡す
%
        FilterInfo = Gids,
%
        Pool = {Pool1,Pool2},
        merge(ControlIn,Control),
%
%       リクエストデバイスとしての処理ルーチンを起動する.
%
        rhandler(Var,Pool2,Attention,
                    ControlIn,Abort,ServerAbort,Uid).
%
%       サーバ自身が処理を中止する場合，ホワイトボードプロセスを
%       終了させる.
%
wait_get_request(aborted,Next,_,_,Pool):-
    true|
        Next = [],
        abort_pool(Pool).
wait_get_request(abnormal,Next,_,_,Pool):-
```

```
    true|
        Next = [],
        abort_pool(Pool).

%
%       リクエストデバイスの処理
%
rhandler(User,Pool,Attention,Control,Abort,ServerAbort,_):-
%
%       サーバの処理中止の通知がきた場合
%
    wait(ServerAbort)|
%
%       既に行なっているユーザサービスの後処理，すなわち
%       すべての要求に対して aborted を返す．
%       ユーザタスク内で動作する保護フィルタプロセスを
%       デッドロックさせないためのサービスである．
        abort_rhandler(User,Attention),
        Pool = [],
        Abort = abort,
        Control = [].
%
rhandler(_,Pool,Attention,Control,Abort,_,_):-
%
%      クライアントタスクが終了してしまった場合
&
    wait(Attention)|
        Pool = [],
        Abort = abort,
        Control = [].
alternatively.
%
%      ユーザ単位のホワイトボードを獲得する．
%
rhandler([user(Result)|Next],Pool,Attention,
                    Control,Abort,ServerAbort,Uid):-
    true|
        Pool = [put(uid(Uid),NewUID,UIDV)|Pool1],
%
%       ユーザタスクの資源木に user_board の名前，Uid を情報として
%       ユーザ単位のホワイトボードの利用を登録する．
%
        Control = [add_resource(user_board,Uid,Status)|ControlN],
%
%       ユーザの要求に獲得成功を返す．
%
        Result = normal(User),
%
%       Uid に対応したプールの獲得 / 生成後ホワイトボードの
```

```
%       処理を行なう.
%
        exist(UIDV,NewUID,Status,User,Attention,ServerAbort),
%
%       次の処理を行なう.
%
        rhandler(Next,Pool1,Attention,ControlN,Abort,ServerAbort,Uid).
rhandler([group(Gid,Result)|Next],Pool,Attention,
                    Control,Abort,ServerAbort,Uid):-
%
%       グループ単位のホワイトボードの処理である.
%
    true|
        Pool = [put(gid(Gid),NewGID,GIDV)|Pool1],
        Control = [add_resource(group_board,Gid,Status)|ControlN],
        Result = normal(User),
        exist(GIDV,NewGID,Status,User,Attention,ServerAbort),
        rhandler(Next,Pool1,Attention,ControlN,Abort,ServerAbort,Uid).
rhandler([],Pool,_,Control,_,_,_):-
%
%       ユーザがストリームを閉じたので処理終了
%
    true|
        Pool = [],
        Control = [].

exist(IDV,NewID,Status,User,Attention,ServerAbort):-
%
%       登録したホワイトボードを取り出す
%
    vector_element(IDV,0,ID)|
        ID = {IDPool,NewID},
%
%       資源木への登録処理結果をまつ
%
        wait_add_resource(Status,User,IDPool,Attention,ServerAbort).
otherwise.
exist(_,NewID,Status,User,Attention,ServerAbort):-
%
%       ホワイトボードが存在しなかったので，生成する
%
    true|
        pool:keyed_bag(BAG),
        merge(BAGIN,BAG),
        BAGIN = {IDPool,NewID},
        wait_add_resource(Status,User,IDPool,Attention,ServerAbort).

wait_add_resource(normal(Control,Abort),User,
                    UID,Attention,ServerAbort):-
```

```
%
%       登録が成功したので制御ストリームを獲得できた.
%
    true|
        handler(User,UID,Attention,Abort,ServerAbort,Control).
otherwise.
wait_add_resource(_,User,UID,Attention,_):-
%
%       資源木への登録に失敗したのは，ユーザタスク終了直前で
%       あったためなので，こちらもユーザサービスを終了する.
%
    true|
        UID = [],
        abort_handler(User,Attention).

%
% handler(User,UID,Attention,Abort,ServerAbort,Control)
%
%       ユーザサービスの本体部である.
%
handler(_,UID,Attention,_,_,Control):-
%
%       ユーザタスクが終了した.
%
    wait(Attention)|
        UID = [],
        Control = [].
handler(User,UID,Attention,Abort,_,Control):-
    wait(Abort)|
%
%       サーバがこのユーザへの処理を中止した.
%
        abort_handler(User,Attention),
        UID = [],
        Control = [].
handler(User,UID,Attention,_,ServerAbort,Control):-
    wait(ServerAbort)|
%
%       サーバ自身が処理全体を中止した.
%
        abort_handler(User,Attention),
        UID = [],
        Control = [].
alternatively.
handler([],UID,_,_,_,Control):-
%
%       ユーザがストリームを閉じたので処理終了
%
    true|
```

```
        UID = [],
        Control = [].
handler([get(Key,Result)|Next],UID,Attention,
                            Abort,ServerAbort,Control):-
%
%       キーに対応した文字列を取り出す
%
    true|
        UID = [get_if_any(Key,X)|UIDN],
        wait_get_if_any(X,Result),
        handler(Next,UIDN,Attention,Abort,ServerAbort,Control).
handler([put(Key,X,Result)|Next],UID,Attention,
                            Abort,ServerAbort,Control):-
%
%       データを登録する
%
    true|
        UID = [put(Key,X)|UIDN],
        Result = normal,
        handler(Next,UIDN,Attention,Abort,ServerAbort,Control).
handler([get_keys(Result)|Next],UID,Attention,
                            Abort,ServerAbort,Control):-
    true|
%
%       キー一覧を返す
%
        UID = [do([get_all(ALL)|PUTALL])|UIDN],
        Result = normal(Keys),
        get_keys(ALL,Keys,PUTALL),
        handler(Next,UIDN,Attention,Abort,ServerAbort,Control).
handler([get_all(Result)|Next],UID,Attention,
                            Abort,ServerAbort,Control):-
%
%       すべてのデータを取り出す
%
    true|
        UID = [get_all(ALL)|UIDN],
        Result = normal(ALL),
        handler(Next,UIDN,Attention,Abort,ServerAbort,Control).

wait_get_if_any(V,Result):-
    vector_element(V,0,X)|
        Result = normal(X).
otherwise.
wait_get_if_any(_,Result):-
    true|
        Result = abnormal.

get_keys([],Keys,PUTALL):-
```

```
    true|
        Keys = [],
        PUTALL = [].
get_keys([{Key,Value}|Next],Keys,PUTALL):-
    true|
        Keys = [Key|KNext],
        PUTALL = [put(Key,Value)|PUTALLN],
        get_keys(Next,KNext,PUTALLN).

abort_pool(Pool):-
    true|
        Pool = [get_all(ALL)],
        abort_all_pool(ALL).

abort_all_pool([]):-
    true|
        true.
abort_all_pool([{_,IDP}|Next]):-
    true|
        IDP = [],
        abort_all_pool(Next).

abort_rhandler(_,Attention):-
    wait(Attention)|
        true.
otherwise.
abort_rhandler([],_):-
    true|
        true.
abort_rhandler([user(Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_rhandler(Next,Attention).
abort_rhandler([group(_,Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_rhandler(Next,Attention).

abort_handler(_,Attention):-
    wait(Attention)|
        true.
otherwise.
abort_handler([],_):-
    true|
        true.
abort_handler([get(_,Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_handler(Next,Attention).
```

```
abort_handler([put(_,_,Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_handler(Next,Attention).
abort_handler([get_keys(Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_handler(Next,Attention).
abort_handler([get_all(Result)|Next],Attention):-
    true|
        Result = aborted,
        abort_handler(Next,Attention).

%
%  以下は保護フィルタコードである.
%
%  保護フィルタは,
%        ユーザの入力はすべて揃うまでサーバへ送らない
%        ユーザへの結果出力は返せるものは全部返す
%  方針で記述する.
%
%   filter(Gids,User,Server)
%      Gids:   ユーザの Gid の列上述のサーバによって送られたもの
%      User:   ユーザの入力変数
%      Server: サーバへの出力変数

filter(_,[],Server):-
    true|
        Server = [].
filter(Gids,[user(Result)|User],Server):-
    true|
        Server = [user(SResult)|ServerN],
        wait_cfilter(SResult,Result),
        filter(Gids,User,ServerN).
filter(Gids,[group(Gid,Result)|User],Server):-
    true|
        member(Gid,Gids,GidsN,Yes),
        wait_gid(Yes,Gid,Server,ServerN,Result),
        filter(GidsN,User,ServerN).

wait_gid(yes,Gid,Server,ServerN,Result):-
    true|
        Server = [group(Gid,SResult)|ServerN],
        wait_cfilter(SResult,Result).
otherwise.
wait_gid(_,_,Server,ServerN,Result):-
    true|
        Server = ServerN,
        Result = abnormal.
```

```
member(_,[],GidsN,Yes):-
    true|
        GidsN = [],
        Yes = no.
member(Gid,[Gid|Next],GidsN,Yes):-
    true|
        GidsN = [Gid|Next],
        Yes = yes.
otherwise.
member(Gid,[Gid2|Next],GidsN,Yes):-
    true|
        GidsN = [Gid2|GidsNN],
        member(Gid,Next,GidsNN,Yes).

wait_cfilter(normal(Server),Result):-
    true|
        Result = normal(User),
        cfilter(User,Server).
otherwise.
wait_cfilter(SResult,Result):-
    true|
        Result = SResult.

cfilter([],Server):-
    true|
        Server = [].
cfilter([get(Key,Result)|User],Server):-
    atom(Key)|
        Server = [get(Key,SResult)|ServerN],
        wait_get_cfilter(SResult,Result),
        cfilter(User,ServerN).
cfilter([put(Key,Value,Result)|User],Server):-
    atom(Key),
    string(Value,_,_)|
        Server = [put(Key,Value,SResult)|ServerN],
        wait_put_cfilter(SResult,Result),
        cfilter(User,ServerN).
cfilter([get_keys(Result)|User],Server):-
    true|
        Server = [get_keys(SResult)|ServerN],
        wait_get_keys(SResult,Result),
        cfilter(User,ServerN).
cfilter([get_all(Result)|User],Server):-
    true|
        Server = [get_all(SResult)|ServerN],
        wait_get_all(SResult,Result),
        cfilter(User,ServerN).
```

```
wait_get_cfilter(normal(Value),Result):-
    true|
        Result = normal(UValue),
        wait_value(Value,UValue).
otherwise.
wait_get_cfilter(SResult,Result):-
    true|
        Result = SResult.

wait_put_cfilter(SResult,Result):-
    wait(SResult)|
        Result = SResult.

wait_get_keys(normal(Keys),Result):-
    true|
        Result = normal(UKeys),
        wait_get_keys2(Keys,UKeys).
otherwise.
wait_get_keys(SResult,Result):-
    true|
        Result = SResult.

wait_get_keys2([],UALL):-
    true|
        UALL = [].
wait_get_keys2([E|ALL],UALL):-
    true|
        UALL = [UE|UALLN],
        wait_value(E,UE),
        wait_get_keys2(ALL,UALLN).

wait_get_all(normal(ALL),Result):-
    true|
        Result = normal(UALL),
        wait_get_all2(ALL,UALL).
otherwise.
wait_get_all(SResult,Result):-
    true|
        Result = SResult.

wait_get_all2([],UALL):-
    true|
        UALL = [].
wait_get_all2([E|ALL],UALL):-
    true|
        UALL = [UE|UALLN],
        wait_get_all3(E,UE),
        wait_get_all2(ALL,UALLN).
```

```
wait_get_all3(E,UE):-
    vector_element(E,0,Key),
    vector_element(E,1,Value)|
        new_vector(UE0,2),
        wait_value(Key,UKey),
        wait_value(Value,UValue),
        set_vector_element(UE0,0,_,UKey,UE1),
        set_vector_element(UE1,1,_,UValue,UE).

wait_value(Value,UValue):-
    wait(Value)|
        UValue = Value.
```

索引

Symbols

! イベント番号 (シェル コマンド) **14**
! 文字列 1 (シェル コマンド) **14**
! (リスナ コマンド) 64, **81, 82**
!! (コンパイラ コマンド) **38**
!! (シェル コマンド) **14**
!! (リスナ コマンド) 64, **82**
!Relink (コンパイラ コマンド) **43**
!Unload (コンパイラ コマンド) **44**
!alias (コンパイラ コマンド) **44**
!auto_link (コンパイラ コマンド) **39**
!cd (コンパイラ コマンド) **38**
!compile (コンパイラ コマンド) **37**
!default_extension (コンパイラ コマンド) 34, **38**
!define (コンパイラ コマンド) **42**
!describe (コンパイラ コマンド) **42**, 42
!distribute (コンパイラ コマンド) 34, **41**, 47, 53
!exit (コンパイラ コマンド) **45**
!get (コンパイラ コマンド) **42**
!help (コンパイラ コマンド) **45**
!keep (コンパイラ コマンド) 34, **45**
!klb (コンパイラ コマンド) **37**
!link (コンパイラ コマンド) 39, **39**
!link_error (コンパイラ コマンド) **39**
!load (コンパイラ コマンド) **43**
!ls (コンパイラ コマンド) **38**
!make (コンパイラ コマンド) **37**
!message (コンパイラ コマンド) **43**
!original_package (コンパイラ コマンド) **39**, 48
!package (コンパイラ コマンド) **40**
!parallel (コンパイラ コマンド) 34, **43**
!parser (コンパイラ コマンド) **40**
!path (コンパイラ コマンド) **38**
!preprocess (コンパイラ コマンド) **40**
!pwd (コンパイラ コマンド) **39**
!relink (コンパイラ コマンド) **44**
!revert (コンパイラ コマンド) **45**
!set (コンパイラ コマンド) 42, **43**
!show (コンパイラ コマンド) **45**
!take (コンパイラ コマンド) 35, **46**
!top_modules (コンパイラ コマンド) **40**, 44
!translate (コンパイラ コマンド) **41**
!unalias (コンパイラ コマンド) **46**
!undefine (コンパイラ コマンド) **42**
!unload (コンパイラ コマンド) **44**
!unparser (コンパイラ コマンド) **41**
!verifier (コンパイラ コマンド) **41**
!verify (コンパイラ コマンド) **41**
!working_packages (コンパイラ コマンド) 40, **44**, 44
; 63
& (パラグラフ コマンド) **199**
^ 71

A

a_little_bigger (インスペクタ コマンド) **163**
abnormal → 入出力メッセージ形式
abort 67, → デバイスアボート
abort (ジェネラルリクエストへのメッセージ) **407**
abort (タスクへのメッセージ) **411**
abort (荘園へのメッセージ) **292**
aborted → 入出力メッセージ形式
aborted (タスクからのメッセージ) **412**
aborted (荘園からのメッセージ) **293**
absolute_id (サーバへのメッセージ) **421**
absolute_id (ジェネラルリクエストへのメッセージ) **408**
absolute_id (タスクへのメッセージ) **411**
activate (ウインドウへのメッセージ) **362**
add (組込述語) **267**
add_bignum (組込述語) **274**
add_date_time (timer_utility の述語) **353**
add_dependency (コンパイラオプション格納プールへのメッセージ) **442**
add_extended (組込述語) **273**
add_op (オペレータプールへのメッセージ) **343**
add_op (リスナ コマンド) **88**
add_resource (サーバへのメッセージ) **421**, **424**
add_resource (タスクへのメッセージ) **411**
add_resource (荘園へのメッセージ) **292**
alias (シェル コマンド) **16**
alias (リスナ コマンド) **85, 86**
alias_expansion (リスナ コマンド) **86**

allow_resource_report (タスクへのメッセージ) **410**
allow_resource_report (荘園へのメッセージ) **292**
alternatively 224
amonitor (パラグラフ コマンド) **182**
and (組込述語) **269**
and_string_elements (疑似組込述語) **281**
append mode 73
append_string (疑似組込述語) **282, 283**
apply (組込述語) **284**, 390
apply_spying (組込述語) **284**
apply_tracing (組込述語) **284**
argument vector 296
arithmetics 267, 271, 273, 274
arity_mismatch 150, 298
as_a_clause (疑似組込述語) **248**, 256
as_a_goal (疑似組込述語) **248**, 256
as_an_argument (疑似組込述語) **248**, 256
atom 283
  cache database 385
  database 385
  device 386
  request device 386
atom (ジェネラルリクエストへのメッセージ) **404**
atom (メッセージ) **386**
atom (組込述語) **264**
atom_table (モジュール)
  :cache_table (述語) **385**
  :get_atom (述語) **385**
  :get_atom_name (述語) **385**
  :intern (述語) **386**
attach (標準入出力制御へのメッセージ) 381, **382**
attention → デバイスアテンション
attention key 65
attributes (ファイルへのメッセージ) **366**
attributes (リスナ コマンド) **101**
authenticate (ユーザデバイスへのメッセージ) **426**

B

back (シェル コマンド) **12**
background 68
bag (pool の述語) **306**
beep (ウインドウへのメッセージ) **361**
begin/end パターン 254
big [*bigger*] (インスペクタ コマンド) **162**
bigger (インスペクタ コマンド) **162**
bignum 273, 274
  compare 273
  operation 274
bignum_less_than (組込述語) **273**
bignum_not_less_than (組込述語) **274**
binary_string_operation (組込述語) **278**, 288
break 67
buffer (モジュール)
  :character_pool (述語) **346**
  :input_filter (述語) **344**, 377
  :interaction_filter (述語) **345**, 377
  :output_filter (述語) **345**, 377, 383
built-in predicate → 組込述語

C

cache_table (atom_table の述語) **385**
cache_table (module の述語) 396
cache_table (module_table の述語) **387**
calc [*calculator*] (リスナ コマンド) **111**
calculate_priority_from_current (組込述語) **286**
calculate_priority_from_minimum (組込述語) **286**
calculator (リスナ コマンド) **111**
caller **296**, 302
cancel (ウインドウへのメッセージ) **357**, 357
cancel (デバイスへのメッセージ) **357**
cancel (メッセージ) **344**
cancel (標準入出力へのメッセージ) **379**
cannot open file 147
carbon_copy (キーなしプールへのメッセージ) **306**, 307, 308
carbon_copy (キー付きプールへのメッセージ) **310**, 312–314
carbon_copy (multiply_keyed_bag へのメッセージ) **320**
carbon_copyp (multiply_keyed_bag へのメッセージ) **321**
:- case 245
cat (シェル コマンド) **29**
cc [*change_condition*] (インスペクタ コマンド) **177**
cd (シェル コマンド) **14, 15**
cd (リスナ コマンド) **98, 99**
change_comment (サーバへのメッセージ) **421**
change_condition (インスペクタ コマンド) **177**
change_element_size (組込述語) **277**
change_macro (リスナ コマンド) **89**, 246
change_nodes (リスナ コマンド) **90**
change_op_pool (フィルタへのメッセージ) **343**
change_password (シェル コマンド) **29**
change_password (ユーザデバイスへのメッセージ) **427**
character_pool (buffer の述語) **346**
clause number 117, **119**, 299

clear (ウインドウへのメッセージ) **362**
cluster 296
cm [*conditional_monitor*] (インスペクタ コマンド) **168**
cmf [*conditional_monitor_and_fork*] (インスペクタ コマンド) **168, 169**
code 283
code (組込述語) **265**
code_to_predicate (組込述語) **284**
collect_value 76
com [*commands*] (インスペクタ コマンド) **161, 162**
com [*commands*] (リスナ コマンド) **87**
commands (インスペクタ コマンド) **161, 162**
commands (リスナ コマンド) **87**
comparator (モジュール)
:descending_sort (述語) **304**
:sort (述語) **303**
compare
bignum 273
floating point 270
integer 266
string 275
compile
batch — 35
conditional — 42
conditional-parameter 42
interactive — 35
compile (シェル コマンド) **33**
compile (モジュール)
:compile (述語) **395**, 396
compile (環境変数)
:auto_link **46**
:default_file_extension **47**
:distribution **47**
:language_options **47**
:link_error **47**
:ls_coulmn **48**
:message **48**
:more_prompt **48**
:original_package **48**
:parallel **48**
:parsers **48**
:path **48**
:preprocessors **48**
:prompt **48**
:top_modules **48**
:unparsers **48**
:verifiers **49**
:working_packages **49**
compile (compile の述語) **395**, 396
compiler **33**
environment variable 46
initiation 34
parameter file 34
self — 33
syntax 36
complement (組込述語) **270**
complement_string_elements (疑似組込述語) **282**
con (リスナ コマンド) **100**
conditional_monitor (インスペクタ コマンド) **168**, 174
conditional_monitor_and_fork (インスペクタ コマンド) **168, 169**, 174
connection (ジェネラルリクエストへのメッセージ) **407**
Control-C → アテンションキー, → デバイスアテンション
convert_macro (シェル コマンド) **263**
copy_string_elements (疑似組込述語) **281**
count (変数ブールへのメッセージ) **348**
create (ウインドウリクエストへのメッセージ) **359, 360**
create (ストリング入出力リクエストへのメッセージ) **371**
create (タイマリクエストへのメッセージ) **368, 369**
create (ユーザリクエストデバイスへのメッセージ) **425, 426**
csp_atom_table_utility (モジュール)
:save (述語) **205**
current_directory (ファイルリクエストへのメッセージ) **365**
current_node (組込述語) **287**
current_priority (組込述語) **286**

D

da [*display_ancestor*] (リスナ コマンド) **107**
date (リスナ コマンド) **87**
day (timer_utility の述語) **352**
deactivate (ウインドウへのメッセージ) **362**
deadlock 298, 299
detection 65, **74**
debug
information 299
def (リスナ コマンド) **100**
:- default 245
default package 39
default_module (リスナ コマンド) **90**
default_package (リスナ コマンド) **90**
:- define 243
define (package の述語) **394**

define_package (モジュールデータベースへのメッセージ) **388**
define_package (モジュールデバイスへのメッセージ) **391**
delete (ディレクトリへのメッセージ) **367**
delete (リスナ コマンド) **100**
deleted (ディレクトリへのメッセージ) **367**
deleted (リスナ コマンド) **100**
descending_sort (comparator の述語) **304**
describe (Preprocessor の述語) **438**
detach (標準入出力制御へのメッセージ) 381, **382**
devide_extended (組込述語) **273**
dfm [*default_module*] (リスナ コマンド) **90**
dfp [*default_package*] (リスナ コマンド) **90**
diff 266
diff (組込述語) **266**
directory (ファイルリクエストへのメッセージ) **364**
dirs (シェル コマンド) **15**
dirs (リスナ コマンド) **99**
disconnected → 入出力メッセージ形式
disp [*display*] (リスナ コマンド) **86**
display (リスナ コマンド) **86**
display_ancestor (リスナ コマンド) **107**, 138
display_console (組込述語) **285**
display_variables (リスナ コマンド) 70, 73, **97**
divide (組込述語) **268**
do (キーなしプールへのメッセージ) **306**, 307, 308
do (キー付きプールへのメッセージ) **311**
do (コンパイラオプション格納プールへのメッセージ) **442**
do (シェル コマンド) **16**
do (フィルタへのメッセージ) **343**
do (load の述語) **396**
do (multiply_keyed_bag へのメッセージ) **320**
do (relink の述語) **395**
do (unload の述語) **396**, 396
do_it (パラグラフ コマンド) **198**
down (インスペクタ コマンド) **165**
dv [*display_variables*] (リスナ コマンド) **97**
dynamic_link (module_table の述語) **390**

E

echo (シェル コマンド) **30**
:- else 244
empty (キーなしプールへのメッセージ) **306**, 308
empty (キー付きプールへのメッセージ) **309**, 311, 312, 314
empty (multiply_keyed_bag へのメッセージ) **319**
empty_pool (operator の述語) **346**
:- end_if 244
end_of_file 366
end_of_file (入力ファイルへのメッセージ) **366**
:- end_switch 245
env [*environment*] (インスペクタ コマンド) **172**
env [*environment*] (リスナ コマンド) **92, 93**
environment (インスペクタ コマンド) **172**
environment (リスナ コマンド) **92, 93**
equal (組込述語) **267**
error 151
error (ライブラリメッセージ出力へのメッセージ) **443**
error_queue_length (リスナ コマンド) **86**, 144
errors (リスナ コマンド) **86**, 144
etc 300
ex [*exit*] (インスペクタ コマンド) **162**
ex [*exit*] (リスナ コマンド) **88**
exception **296**, 300
  arity_mismatch 298
  deadlock 298
  etc 300
  handling **290**
    utility **302**
  illegal_input 297
  illegal_merger_input 298
  integer_overflow 297
  integer_zero_division 297
  invalid_floating_point_operation 297
  mask 290, 291, 302
  merger_deadlock 299
  out_of_bounds 297
  raised 300
  range_overflow 297
  reduction failure 298
  spy 284, 300
  tag 290, 291
    allocation 301
    bit position 301
  trace 284, 299
  undefined_module 298
  undefined_predicate 298
  unification failure 298
exception (タスクからのメッセージ) **412**
exception (荘園からのメッセージ) **296**
exception_table (モジュール)
  :translate_profiling_data (述語) **302**
exclusive_or (組込述語) **270**
execute (ジェネラルリクエストへのメッセージ) 379, 381, **404**, **406**
execute (shoen の述語) 120, **290**
execution

control 290
time 70, 88
exit (インスペクタ コマンド) **162**
exit (シェル コマンド) **12**
exit (パラグラフ コマンド) **200**, **202**
exit (リスナ コマンド) 67, 68, 70, **88**, 141
exit 用 (矩形) ボタン (パラグラフ コマンド) **198**
expunge (ディレクトリへのメッセージ) **368**
expunge (リスナ コマンド) **101**

F

f [*fork*] (インスペクタ コマンド) **162**
failure 147, 298
fgt [*forget*] (リスナ コマンド) **95, 96**
file (ジェネラルリクエストへのメッセージ) **334**, **404**
floating point
compare 270
conversion to integer 272
operation 271
floating_point (組込述語) **264**
floating_point_add (組込述語) **271**
floating_point_divide (組込述語) **272**
floating_point_equal (組込述語) **271**
floating_point_less_than (組込述語) **270**
floating_point_multiply (組込述語) **272**
floating_point_not_equal (組込述語) **271**
floating_point_not_less_than (組込述語) **270**
floating_point_subtract (組込述語) **271**
floating_point_to_integer (組込述語) **272**
flush (ウインドウへのメッセージ) **361**
flush (ライブラリメッセージ出力へのメッセージ) **444**
flush (出力用フィルタからのメッセージ) **345, 346**
flush (出力用フィルタへのメッセージ) **342**
flush (出力, 追加ファイルへのメッセージ) **367**
flush (標準入出力へのメッセージ) **378**
fore (シェル コマンド) **12**
forget (リスナ コマンド) 73, **95, 96**
fork (インスペクタ コマンド) 154, **162**
foster parent 294

G

garbage collector → *ガーベジコレクタ*
GC → *ガーベジコレクタ*
gc (シェル コマンド) **27**
general_request (get_request コマンドからのメッセージ) **421**
get (キーなしプールへのメッセージ) **306**, 307, 308
get (キー付きプールへのメッセージ) **309**, 312–314
get (コンパイラオプション格納プールへのメッセージ) **442**
get (シェル コマンド) **13, 14**, **16**, **22–24**
get (multiply_keyed_bag へのメッセージ) **319**
get_all (キーなしプールへのメッセージ) **306**, 307, 308
get_all (キー付きプールへのメッセージ) **310**, 312–314
get_all (multiply_keyed_bag へのメッセージ) **319**
get_all_info (グループデバイスへのメッセージ) **427**
get_all_info (ユーザデバイスへのメッセージ) **426**
get_all_info (ログインデバイスへのメッセージ) **428**
get_all_modules (モジュールデータベースへのメッセージ) **390**
get_all_modules (モジュールデバイスへのメッセージ) **392**
get_all_packages (モジュールデータベースへのメッセージ) **388**
get_all_packages (モジュールデバイスへのメッセージ) **391**
get_and_put (キー付きプールへのメッセージ) **310**, 312–314
get_and_put (multiply_keyed_bag へのメッセージ) **320**
get_and_put_if_any (キー付きプールへのメッセージ) **310**, 312–314
get_and_put_if_any (multiply_keyed_bag へのメッセージ) **320**
get_atom (アトムデバイスへのメッセージ) **386**
get_atom (atom_table の述語) **385**
get_atom_name (アトムデバイスへのメッセージ) **386**
get_atom_name (atom_table の述語) **385**
get_atom_table (コンパイラオプション格納プールへのメッセージ) **442**
get_attributes (module の述語) **393**
get_average_chaining_length (キー付きプール (ハッシュ表) へのメッセージ) **311**, 312
get_buffer_name (ウインドウへのメッセージ) **363**
get_char_size (ウインドウへのメッセージ) **363**
get_code (モジュールデータベースへのメッセージ) **389**
get_code (モジュールデバイスへのメッセージ) **392**
get_console (組込述語) **286**
get_count (タイマへのメッセージ) **369**
get_date_count (timer_utility の述語) **352**
get_date_string (timer_utility の述語) **351**
get_date_time (タイマへのメッセージ) **369**
get_date_value (timer_utility の述語) **351**
get_default_package (モジュールデータベースへのメッセージ) **387**
get_directory (コンパイラオプション格納プールへのメッセージ) **442**

get_file_request (コンパイラオプション格納プールへのメッセージ) **442**
get_font (ウインドウへのメッセージ) **363**
get_gid (グループデバイスへのメッセージ) **427**
get_gids (グループデバイスへのメッセージ) **427**
get_heap_size (組込述語) **287**
get_if_any (キーなしプールへのメッセージ) **306**, 307, 308
get_if_any (キー付きプールへのメッセージ) **309**, 312–314
get_if_any (multiply_keyed_bag へのメッセージ) **319**
get_if_any_and_put (キー付きプールへのメッセージ) **310**, 312–314
get_if_any_and_put (multiply_keyed_bag へのメッセージ) **320**
get_info (ユーザデバイスへのメッセージ) **426**
get_info (ログインデバイスへのメッセージ) **428**
get_load_factor (キー付きプール (ハッシュ表) へのメッセージ) **310**, 312
get_login (ログインデバイスへのメッセージ) **428**
get_max_if_any (キー付きプール (木構造) へのメッセージ) **313**, 314
get_max_if_any (sorted_bag へのメッセージ) **308**
get_max_size (ウインドウリクエストへのメッセージ) **360**
get_min_if_any (キー付きプール (木構造) へのメッセージ) **313**, 314
get_min_if_any (sorted_bag へのメッセージ) **308**
get_module (モジュールデバイスへのメッセージ) **392**
get_module_in_the_package (モジュールデータベースへのメッセージ) **389**
get_module_in_the_package (モジュールデバイスへのメッセージ) **392**
get_module_name (module の述語) **302**, **393**
get_module_table (コンパイラオプション格納プールへのメッセージ) **442**
get_name (変数プールへのメッセージ) **348**
get_package (コンパイラオプション格納プールへのメッセージ) **442**
get_package_info (モジュールデータベースへのメッセージ) **388**
get_package_info (モジュールデバイスへのメッセージ) **391**
get_position (ウインドウへのメッセージ) **363**
get_predicate_info (module の述語) **302**, **393**
get_request (register コマンドへのメッセージ) **420**
get_super_package (モジュールデータベースへのメッセージ) **388**
get_super_package (モジュールデバイスへのメッセージ) **391**
get_time_count (timer_utility の述語) **353**
get_time_string (timer_utility の述語) **352**
get_time_unit (タイマへのメッセージ) **370**
get_time_value (timer_utility の述語) **352**
get_title (ウインドウへのメッセージ) **363**
get_uid (ユーザデバイスへのメッセージ) **426**
get_users (グループデバイスへのメッセージ) **427**
get_value (変数プールへのメッセージ) **348**
getb (ウインドウへのメッセージ) **360**
getb (入力ファイルへのメッセージ) **366**
getb (入力用フィルタからのメッセージ) **345, 346**
getb (入力用フィルタへのメッセージ) **339**
getb (標準入出力へのメッセージ) **378**
getc (入力用フィルタへのメッセージ) **338**
getenv (ジェネラルリクエストへのメッセージ) **407, 408**
getenv (シェル コマンド) **23**
getenv (リスナ コマンド) **94**
getl (入力用フィルタへのメッセージ) **339**
getp (multiply_keyed_bag へのメッセージ) **320**
getp_all (multiply_keyed_bag へのメッセージ) **321**
getp_and_put (multiply_keyed_bag へのメッセージ) **321**
getp_and_put_if_any (multiply_keyed_bag へのメッセージ) **321**
getp_if_any (multiply_keyed_bag へのメッセージ) **321**
gett (入力用フィルタへのメッセージ) **339**
getwt (入力用フィルタへのメッセージ) **339**
go (stdio_server の述語) **424**
Goal @ node (リスナ コマンド) 63, **81**
Goal @ priority (リスナ コマンド) 63, **81**
Goal @ relative_priority (リスナ コマンド) **81**
Goal1, Goal2 (リスナ コマンド) 63, **81**
Goal1; Goal2 (リスナ コマンド) 63, **81**
grep (シェル コマンド) **30, 31**

H

h, he [*help*] (インスペクタ コマンド) **161**
hash 287, 305
hash (hasher の述語) **305**
hash (組込述語) **287**
hasher (モジュール)
:hash (述語) **305**
he [*help*] (リスナ コマンド) **87**
help (インスペクタ コマンド) **161**
help (シェル コマンド) **13**
help (リスナ コマンド) **87**

hide (ウインドウへのメッセージ) **362**
his [*history*] (リスナ コマンド) **83**
History (パラグラフ コマンド) **198**
history (インスペクタ コマンド) **166**
history (シェル コマンド) **13**
history (リスナ コマンド) 64, **83**
history manipulator **193**
history_length (リスナ コマンド) 64, **83**
hl [*history_length*] (リスナ コマンド) **83**
HOOKo 76–79

I

i [*inspect*] (インスペクタ コマンド) **162**
I/O ノード 400, **400**, 403, 407
id (ジェネラルリクエストへのメッセージ) **408**
id (タスクへのメッセージ) **411**
idle 75
:- if_defined 244
:- if_equal 244
:- if_not_defined 244
:- if_not_equal 244
iflush (標準入出力へのメッセージ) **378**
igetb (標準入出力へのメッセージ) **378**
illegal
  argument 146
  goal 146
  input 146
  macro module 144
  specification 147
illegal_input 149, 297
illegal_merger_input 147, 298
in (リスナ コマンド) **111**
in_the_body (疑似組込述語) **249**, 256
in_the_guard (疑似組込述語) **248**, 256
in_the_head (疑似組込述語) **248**, 256
include 226
info (ライブラリメッセージ出力へのメッセージ) **443**
info (リスナ コマンド) **101**
:- inherit_macro 256
input output 285
input_filter (buffer の述語) **344**, 377
input_language (コンパイラオプション格納プールへのメッセージ) **442**
ins [*inspect*] (リスナ コマンド) **107**
inspect (インスペクタ コマンド) **162**
inspect (シェル コマンド) **22**
inspect (リスナ コマンド) **107**
inspector (環境変数)
  :initial_window_size **156**
  :print_string_mode **156**
  :window_size_increment **156**
  :window_size_increment_l **156**
integer
  compare 266
  conversion to floating point 272
  double precision operation 273
  operation 267
integer (組込述語) **264**
integer_overflow 150, 297
integer_to_floating_point (組込述語) **272**
integer_zero_division 150, 297
inter (リスナ コマンド) **112**
inter_file_verify (Verifier の述語) **441**
interaction_filter (buffer の述語) **345**, 377
intern (アトムデバイスへのメッセージ) **386**
intern (atom_table の述語) **386**
interruption 65
interval (リスナ コマンド) **111**
invalid_floating_point_operation 150, 297
io (ストリング入出力へのメッセージ) **372**
IPL 102, 143, **203**
ipl_versions (シェル コマンド) **219**
iprompt (標準入出力へのメッセージ) **378**
iputb (標準入出力へのメッセージ) **378**
iungetb (標準入出力へのメッセージ) **378**

K

key_merge (sorter の述語) **329**
key_sort (sorter の述語) **328, 329**
keyed_bag (pool の述語) **309**
keyed_sorted_bag (pool の述語) **312**
keyed_sorted_set (pool の述語) **313**
kill (シェル コマンド) **12**
kl1_library_message (モジュール)
  :position (述語) **444**
kl1_standard_preprocessor (モジュール)
  :parse_with_op_pool (述語) **441**

L

l [*list*] (リスナ コマンド) **98**
language defined resource 290
last_error (シェル コマンド) **22**
last_error (リスナ コマンド) **86**, 144
lbig [*a_little_bigger*] (インスペクタ コマンド) **163**
lc (シェル コマンド) **31**
len [*length*] (インスペクタ コマンド) **166**
length (インスペクタ コマンド) **166**
less_than (組込述語) **266**

library 394
list (インスペクタ コマンド) **172, 173**
list (リスナ コマンド) 67, 72, **98**, 136, 143
list (組込述語) **265**
listen [*listener*] (シェル コマンド) **58**
listener **57**
listener (シェル コマンド) **58**
listener.com 70
listener:listener (シェル コマンド) **59**
listing (ディレクトリへのメッセージ) **367**
listing (リスナ コマンド) **100**
load **53**
load (シェル コマンド) 39, 43, **53**
load (パラグラフ コマンド) **201**
load (モジュール)
  :do (述語) **396**
loader **53**
local predicate 62, 146
login (ログインデバイスへのメッセージ) **428**
login.com 1
logout (ログインデバイスへのメッセージ) **428**
ls (シェル コマンド) **31**
ls [*list*] (インスペクタ コマンド) **172, 173**
ls [*listing*] (リスナ コマンド) **100**

M

m [*monitor*] (インスペクタ コマンド) **167**
m [*monitor*] (リスナ コマンド) **108**
macro (シェル コマンド) **259**
:- macro_module 255
make_branch (標準入出力資源サーバ制御ストリームへのメッセージ) **423**
mask 290, 291, 302
max_if_any (multiply_keyed_bag へのメッセージ) **319**
me (インスペクタ コマンド) **163**
merge 285
merge (sorter の述語) **329**
merge (組込述語) **285**, 396
merger 285
  deadlock 76, 299
merger_deadlock 299
message (ライブラリメッセージ出力へのメッセージ) **443**
message (Parser の述語) **439**
mf [*monitor_and_fork*] (インスペクタ コマンド) **167, 168**
mf [*monitor_and_fork*] (リスナ コマンド) **108**
MGHOKo 76, 78–80
min_if_any (multiply_keyed_bag へのメッセージ) **319**
module 283
  database 386
  device 390
  name 302
  request device 391
  utility 54, **393**
module (ジェネラルリクエストへのメッセージ) **404**
module (シェル コマンド) **54**
module (モジュールデバイスへのメッセージ) **391**
module (モジュール)
  :cache_table (述語) 396
  :get_attributes (述語) **393**
  :get_module_name (述語) **302**, **393**
  :get_predicate_info (述語) **302**, **393**
module (組込述語) **265**
module:info (シェル コマンド) **54**
module:remove (シェル コマンド) **55**
module:where (シェル コマンド) **55**
module_element (組込述語) **283**
module_table (モジュール)
  :cache_table (述語) **387**
  :dynamic_link (述語) **390**
modules (system_defined_servers の述語) **211**
ModuleSpec ! Local (リスナ コマンド) 62, **81**
ModuleSpec : Public (リスナ コマンド) 61, **81**
modulo (組込述語) **268**
monitor (インスペクタ コマンド) **167**, 174
monitor (リスナ コマンド) **108**
monitor (環境変数)
  :query_mode **175**
  :silent_mode **176**
  :stream_auto_mode **175**
monitor_and_fork (インスペクタ コマンド) **167, 168**, 174
monitor_and_fork (リスナ コマンド) **108**
monitored_terms (インスペクタ コマンド) **170**, **178**
month (timer_utility の述語) **352**
move_string_elements (疑似組込述語) **282**
MRB 31, 288
  GC 74
mt [*monitored_terms*] (インスペクタ コマンド) **170, 178**
multiply (組込述語) **268**
multiply_bignum_by_word (組込述語) **274**
multiply_extended (組込述語) **273**
multiply_keyed_bag (pool の述語) **316**

N

n [*name*] (インスペクタ コマンド) **172**

name (インスペクタ コマンド) **172**
nand_string_elements (疑似組込述語) **281**
network set up 215
  LIA-net **216**
  TCP/IP **215**
new
  argument 290
  goal 290
new_atom (組込述語) **283**
new_string (組込述語) **276**
new_vector (組込述語) **275**
new_win (パラグラフ コマンド) **198**
next_attention (デバイスへのメッセージ) **357**
next_attention (メッセージ) **344**
next_attention (標準入出力へのメッセージ) **379**
nl (出力用フィルタへのメッセージ) **341**
node 287
  information 294, **296**
nonexistent
  command 146
  local predicate 146
  macro module 144
  module 144
  package 144
  public predicate 146
nor_string_elements (疑似組込述語) **281**
normal → 入出力メッセージ形式
nospy (リスナ コマンド) **105**, 134
nostatistics (リスナ コマンド) 70, **89**
not_equal (組込述語) **267**
not_less_than (組込述語) **267**
not_xor_string_elements (疑似組込述語) **281**
notrace (リスナ コマンド) **103**, **107**, 115, 135
notrace_all (リスナ コマンド) **107**, 135
nst [*nostatistics*] (リスナ コマンド) **89**
ntr [*notrace*] (リスナ コマンド) **103**, **107**
ntr_all [*notrace_all*] (リスナ コマンド) **107**
Num (パラグラフ コマンド) **199**
number (リスナ コマンド) **111**
number_of_feps (ジェネラルリクエストへのメッセージ) **407**

O

offset 296
on_after (タイマへのメッセージ) **370**
on_at (タイマへのメッセージ) **370**
op 226
op [*operator*] (リスナ コマンド) **88**
open (ファイルリクエストへのメッセージ) **364**
open (標準入出力資源サーバ制御ストリームからのメッセージ) **422**
open (標準入出力資源サーバ制御ストリームへのメッセージ) **423**
operation
  bignum 274
  double precision 273
  floating point 271
  integer 267
  string 278
operator (オペレータブールへのメッセージ) **343**
operator (モジュール)
  :empty_pool (述語) **346**
  :pool (述語) **346**
operator (リスナ コマンド) **88**
or (組込述語) **269**
or_string_elements (疑似組込述語) **281**
OS 定義資源 291, **398**
  タイプ 67
otherwise 225
out (リスナ コマンド) **112**
out_of_bounds 150, 297
output_filter (buffer の述語) **345**, 377, 383
Overall What × When (パラグラフ コマンド) **195**
Overall What × Where (パラグラフ コマンド) **196**

P

pack (multiply_keyed_bag へのメッセージ) 316, **320**
package 393
  builtin — 394
  inheritance 393
  utility 53, 394
package (モジュール)
  :define (述語) **394**
package (relink の述語) **395**, 396
package:define (シェル コマンド) **53**
package:listing (シェル コマンド) **54**
package:protect (シェル コマンド) **54**
pal [*print_atom_length*] (インスペクタ コマンド) **171**
pal [*print_atom_length*] (リスナ コマンド) **91**
ParaGraph **185**
paragraph (リスナ コマンド) **111**
parse (Parser の述語) **439**
parse_with_op_pool (kl1_standard_preprocessor の述語) **441**
Parser (モジュール)
  :message (述語) **439**
  :parse (述語) **439**
  :sort (述語) **439**

pathname (ディレクトリへのメッセージ) **367**
pathname (ファイルへのメッセージ) **365**
pd [*print_depth*] (インスペクタ コマンド) **170**
pd [*print_depth*] (リスナ コマンド) **91**
performance meter **181**
perpetual suspension 298, 299
pg [*paragraph*] (リスナ コマンド) **111**
pimos.com 25
pimos.conf 206
pimos.group 210
pimos.ipl 203
pimos.log 209
pimos.logins 209
pl [*print_length*] (インスペクタ コマンド) **171**
pl [*print_length*] (リスナ コマンド) **91**
plot (パラグラフ コマンド) **202**
pm [*print_mode*] (インスペクタ コマンド) **172**
pmeter (パラグラフ コマンド) **182**
pmeter (環境変数)
  :display **183**
  :display_node **184**
  :display_password **184**
  :display_username **184**
  :gateway_node **184**
  :interval **183**
  :item **184**
  :timer_node **183**
  :timer_password **183**
  :timer_username **183**
pool (モジュール)
  :bag (述語) **306**
  :keyed_bag (述語) **309**
  :keyed_set (述語) **311**
  :keyed_sorted_bag (述語) **312**
  :keyed_sorted_set (述語) **313**
  :multiply_keyed_bag (述語) **316**
  :queue (述語) **307**
  :sorted_bag (述語) **308**
  :stack (述語) **307**
pool (operator の述語) **346**
pool2 (variable の述語) **348**
popd (シェル コマンド) **15**
popd (リスナ コマンド) **99**
position (kl1_library_message の述語) **444**
predicate
  information 302
predicate_to_code (組込述語) **283**
predicates (リスナ コマンド) **101**
Preprocessor (モジュール)
  :describe (述語) **438**
  :translate (述語) **440**
preprocessor **35**
  chain 40
print_atom_length (インスペクタ コマンド) 154, **171**
print_atom_length (リスナ コマンド) **91**, 97, 137, 143
print_atom_length (出力用フィルタへのメッセージ) **343, 344**
print_depth (インスペクタ コマンド) 154, **170**
print_depth (リスナ コマンド) **91**, 97, 137, 143
print_depth (出力用フィルタへのメッセージ) **343, 344**
print_error (メッセージ) 339
print_error (出力用フィルタへのメッセージ) **342**
print_length (インスペクタ コマンド) 154, **171**
print_length (リスナ コマンド) **91**, 97, 137, 143
print_length (出力用フィルタへのメッセージ) **343, 344**
print_mode (インスペクタ コマンド) 154, **172**
print_string_length (インスペクタ コマンド) 154, **171**
print_string_length (リスナ コマンド) **91**, 97, 137, 143
print_string_length (出力用フィルタへのメッセージ) **343, 344**
print_string_mode (インスペクタ コマンド) 154, **171**
print_variable_mode (リスナ コマンド) **91**, 97, 137, 143
printenv (リスナ コマンド) **93**
priority 228, 286, 291
profile (ジェネラルリクエストへのメッセージ) **406**
profile (タスクへのメッセージ) **411**
profile (リスナ コマンド) 70, **108**
profile (荘園へのメッセージ) **292**
profile (shoen の述語) 120, **291**
profile_msg (リスナ コマンド) 70, **110**
profile_node (リスナ コマンド) 70, **110**
profiling_completed (タスクからのメッセージ) **412**
profiling_completed (荘園からのメッセージ) **294**
profiling_information (タスクからのメッセージ) **412**
profiling_information (荘園からのメッセージ) **294**
profiling_rejected (タスクからのメッセージ) **412**
profiling_rejected (荘園からのメッセージ) **293**
profiling_started (タスクからのメッセージ) **412**
profiling_started (荘園からのメッセージ) **293**
prompt (ウインドウへのメッセージ) **361**

prompt (リスナ コマンド) **84**
prompt (入力ファイルへのメッセージ) **366**
prompt (入力用フィルタからのメッセージ) **345, 346**
prompt (入力用フィルタへのメッセージ) **340**
prompt (標準入出力へのメッセージ) **378**
protect_package (モジュールデータベースへのメッセージ) **388**
protect_package (モジュールデバイスへのメッセージ) **391**
protection violation 144
protocol 445
  definition language 445
    grammer 445
    system-defined protocol 445
protocol
  — of PIMOS 449
psl [*print_string_length*] (インスペクタ コマンド) **171**
psl [*print_string_length*] (リスナ コマンド) **91**
psm [*print_string_mode*] (インスペクタ コマンド) **171**
pu [*push_root*] (インスペクタ コマンド) **167**
public predicate 61, 146
purge (ディレクトリへのメッセージ) **368**
purge (リスナ コマンド) **101**
push_root (インスペクタ コマンド) **167**
pushd (シェル コマンド) **15**
pushd (リスナ コマンド) **99**
put (キーなしプールへのメッセージ) **306**, 307, 308
put (キー付きプールへのメッセージ) **309**, 313
put (重複を許さないキー付きプールへのメッセージ) **312**, 314
put (multiply_keyed_bag へのメッセージ) **319**
put_console (組込述語) **286**
put_var (変数プールへのメッセージ) **348**
putb (ウインドウへのメッセージ) **361**
putb (出力用フィルタからのメッセージ) **345, 346**
putb (出力用フィルタへのメッセージ) **340**
putb (出力, 追加ファイルへのメッセージ) **366**
putb (標準入出力へのメッセージ) **378**
putc (出力用フィルタへのメッセージ) **340**
putf (出力用フィルタへのメッセージ) **342**
putl (出力用フィルタへのメッセージ) **340**
putt (出力用フィルタへのメッセージ) **341**
puttq (出力用フィルタへのメッセージ) **341**
putwt (出力用フィルタへのメッセージ) **340**
putwtq (出力用フィルタへのメッセージ) **341**
pv [*print_variable_mode*] (リスナ コマンド) **91**
pwd (シェル コマンド) **15**
pwd (リスナ コマンド) **99, 100**

Q

qm [*query_mode*] (インスペクタ コマンド) **169**, **178**
query (リスナ コマンド) 73, **97**
query_mode (インスペクタ コマンド) **169**, 175, **178**
queue (pool の述語) **307**

R

r (パラグラフ コマンド) **198**
raise (shoen の述語) 120, **300**, 333, **379**, **381**, **383**, **403**, **409**, 428
raised 148, 300
range_overflow 149, 150, 297
read mode 73
read_console (組込述語) **285**
reduction 70, 88, 290
  failure 298
register (ジェネラルリクエストへのメッセージ) **404**, **418**
relink **50**, 257
relink (シェル コマンド) 39, 43, 44, **50, 51**, 258
relink (モジュール)
  :do (述語) **395**
  :package (述語) **395**, 396
relink:package (シェル コマンド) **51**
relink:package(Package) (シェル コマンド) **51**
relinker **50**, 257
rem [*remember*] (リスナ コマンド) **94, 95**
remember (リスナ コマンド) **94, 95**
remove (パラグラフ コマンド) **201**
remove_module (モジュールデータベースへのメッセージ) **389, 390**
remove_module (モジュールデバイスへのメッセージ) **392**
remove_op (オペレータプールへのメッセージ) **343**
remove_op (リスナ コマンド) **88**
remove_package (モジュールデータベースへのメッセージ) **388**
remove_package (モジュールデバイスへのメッセージ) **391**
renew (パラグラフ コマンド) **201**
replace_op_pool (フィルタへのメッセージ) **343**
request (ジェネラルリクエストへのメッセージ) **403**
request (標準入出力資源サーバ制御ストリームへのメッセージ) **422**
request_anyway (ジェネラルリクエストへのメッセージ) **403**
resend (ウインドウへのメッセージ) 357, **357**
resend (デバイスへのメッセージ) **357**
resend (メッセージ) **344**

resend (標準入出力へのメッセージ) **379**
reset (ウインドウへのメッセージ) 356, **357**
reset (ジェネラルリクエストへのメッセージ) **404**
reset (デバイスへのメッセージ) **356**
reset (パラグラフ コマンド) **200**
reset (メッセージ) **344**
reset (標準入出力へのメッセージ) 377, **378**, 381
reset (変数プールへのメッセージ) **348**
reset* (パラグラフ コマンド) **200**
reshape (ウインドウへのメッセージ) **362**
resource 290
  consumption 65, 70, 88, 290, 292
  management 290
resource_added (タスクからのメッセージ) **412**
resource_added (荘園からのメッセージ) **293**
resource_low (タスクからのメッセージ) **412**
resource_low (荘園からのメッセージ) **293**
resource_report_allowed (タスクからのメッセージ) **412**
resource_report_allowed (荘園からのメッセージ) **293**
Result (パラグラフ コマンド) **197**
resume 65
return_resource (タスクからのメッセージ) **412**
return_resource (荘園からのメッセージ) **293**
rewrite_depth (リスナ コマンド) **92**
rewrite_length (リスナ コマンド) **92**
rmonitor (パラグラフ コマンド) **182**
rs (シェル コマンド) **27**
rsall (シェル コマンド) **27**
rwd [*rewrite_depth*] (リスナ コマンド) **92**
rwl [*rewrite_length*] (リスナ コマンド) **92**

S

s (パラグラフ コマンド) **198**
sam [*stream_auto_mode*] (インスペクタ コマンド) **169**, **178**
save (パラグラフ コマンド) **201**
save (csp_atom_table_utility の述語) **205**
search_character (組込述語) **277**
search_characters (組込述語) **278**
select_buffer (ウインドウへのメッセージ) **362**
seqgc (シェル コマンド) **27**
set (シェル コマンド) **13–15**, **22–24**
set (パラグラフ コマンド) **199**
set_current_directory (ファイルリクエストへのメッセージ) **364**
set_default_package (モジュールデータベースへのメッセージ) **387**
set_font (ウインドウへのメッセージ) **362**
set_position (ウインドウへのメッセージ) **361**
set_size (ウインドウへのメッセージ) **361**
set_string_element (組込述語) **277**, 288
set_substring (疑似組込述語) **280**
set_time_unit (タイマへのメッセージ) **370**
set_title (インスペクタ コマンド) **162**
set_title (ウインドウへのメッセージ) **362**
set_up (グループデバイスへのメッセージ) **427**
set_up (ユーザデバイスへのメッセージ) **426**
set_vector_element (組込述語) **275**, 288
setenv (ジェネラルリクエストへのメッセージ) **407**
setenv (シェル コマンド) **23**
setenv (リスナ コマンド) **94**
set* (パラグラフ コマンド) **199**
sf [*spy_fork*] (リスナ コマンド) **103**
sh (シェル コマンド) **13**
sh [*show*] (インスペクタ コマンド) **166**
shell (シェル コマンド) **13**
shell (パラグラフ コマンド) **199**
shell.com 25
shift_left (組込述語) **269**
shift_right (組込述語) **269**
shifting_string_operation (組込述語) **279**, 288
shoen **290**
  abort 292
  aborted 293
  add_resource 292
  allow_resource_report 292
  creation **290**
  exception 296
  profile 292
  profiling_completed 294
  profiling_information 294
  profiling_rejected 293
  profiling_started 293
  resource_added 293
  resource_low 293
  resource_report_allowed 293
  restart 291
  return_resource 293
  start 291
  started 293
  statistics 292, 293
  stop 292
  stopped 293
  terminated 292
shoen (モジュール)
  :execute (述語) 120, **290**

:profile (述語) 120, **291**
:raise (述語) 120, **300**, 333, **377**, **379**, **381**, **383**, **403**, **409**, 428
show (インスペクタ コマンド) **166**
show (ウインドウへのメッセージ) **362**
shutdown (シェル コマンド) **28**
si [*spying*] (リスナ コマンド) **105, 106**
silent_mode (インスペクタ コマンド) **169**
SIMPOS 1, 4
skip (入力用フィルタへのメッセージ) **340**
sm [*silent_mode*] (インスペクタ コマンド) **169**
sort (comparator の述語) **303**
sort (Parser の述語) **439**
sort (sorter の述語) **328**
sorted_bag (pool の述語) **308**
sorter (モジュール)
:key_merge (述語) **329**
:key_sort (述語) **328, 329**
:merge (述語) **329**
:sort (述語) **328**
source_filename (コンパイラオプション格納プールへのメッセージ) **442**
spy
exception 148, 284, 300
identifier 300
spy (リスナ コマンド) **104**, 134
spy_fork (リスナ コマンド) **103**, 115
spy_reduction (リスナ コマンド) **103**, 115
spying (リスナ コマンド) **105, 106**, 134, 135
sr [*spy_reduction*] (リスナ コマンド) **103**
st [*statistics*] (リスナ コマンド) **88, 89**
stack (pool の述語) **307**
start (荘園へのメッセージ) **291**
started (荘園からのメッセージ) **293**
statistics (タスクからのメッセージ) **412**
statistics (タスクへのメッセージ) **411**
statistics (リスナ コマンド) 70, **88, 89**
statistics (荘園からのメッセージ) **293**
statistics (荘園へのメッセージ) **292**
statistics_started (タスクからのメッセージ) **412**
statistics_started (荘園からのメッセージ) **293**
status (ジェネラルリクエストへのメッセージ) **408**
status (シェル コマンド) **12**
status (タスクへのメッセージ) **411**
status (ユーザデバイスへのメッセージ) **426**
stdio (メッセージ) **380**
stdio_server (モジュール)
:go (述語) **424**
stdmsg (ライブラリメッセージ出力へのメッセージ) **444**
stdout (ライブラリメッセージ出力へのメッセージ) **443, 444**
stop (シェル コマンド) **12**
stop (タスクへのメッセージ) **410**
stop (荘園へのメッセージ) **292**
stopped (タスクからのメッセージ) **412**
stopped (荘園からのメッセージ) **293**
stream 285
stream_auto_mode (インスペクタ コマンド) **169**, 175, **178**
string 276
compare 275
operation 278
search 277
string (組込述語) **265**, **276**
string_element (組込述語) **276, 277**
string_io (ジェネラルリクエストへのメッセージ) **335**, **404**
string_less_than (組込述語) **276**
string_not_less_than (組込述語) **276**
string_tail (疑似組込述語) **282**
su (シェル コマンド) **29**
substring (疑似組込述語) **279**
subtract (組込述語) **268**
subtract_bignum (組込述語) **274**
subtract_date_time (timer_utility の述語) **353**
subtract_extended (組込述語) **273**
suicide (ジェネラルリクエストへのメッセージ) **407**
suspend 77, 78, 294
suspend (シェル コマンド) **13**
suspend (リスナ コマンド) 59, **87**
suspension 70
:- switch 245
switch (標準入出力へのメッセージ) **378**
switch (標準入出力制御へのメッセージ) 381
syntax_error (ライブラリメッセージ出力へのメッセージ) **443**
system_defined_servers (モジュール)
:modules (述語) **211**
system_timer (組込述語) **287**

T

tab (出力用フィルタへのメッセージ) **342**
tag → exception tag
take (シェル コマンド) **24**
take (リスナ コマンド) 70, **84**
task (環境変数)
:directory **3**, **23**, **49**, **59**, 365
:fep **4**

:home **4**, **24**, **59**
:io_node **4**, **24**, **59**, 206
:package **3**, **24**, **49**, **59**, 60, 90
:user **4**
tee (シェル コマンド) **29**
terminated (タスクからのメッセージ) **412**
terminated (荘園からのメッセージ) **292**
time_unit_translator (モジュール)
:trans (述語) **353**
timer 287
timer (ジェネラルリクエストへのメッセージ) **334**, **404**
timer_utility (モジュール)
:add_date_time (述語) **353**
:day (述語) **352**
:get_date_count (述語) **352**
:get_date_string (述語) **351**
:get_date_value (述語) **351**
:get_time_count (述語) **353**
:get_time_string (述語) **352**
:get_time_value (述語) **352**
:month (述語) **352**
:subtract_date_time (述語) **353**
To (パラグラフ コマンド) **199**
tr [*trace*] (リスナ コマンド) **102**, **106**
trace
exception 148, 284, 299
identifier 299
trace (リスナ コマンド) **102**, **106**, 115, 118, 135
tracer 113
tracing (リスナ コマンド) **107**, 135
trans (time_unit_translator の述語) **353**
translate (Preprocessor の述語) **440**
translate_profiling_data (exception_table の述語) **302**
tri [*tracing*] (リスナ コマンド) **107**
ts (シェル コマンド) **28**
tsall (シェル コマンド) **28**
type check 264

U

u [*up*] (インスペクタ コマンド) **165**
unalias (シェル コマンド) **16**
unalias (リスナ コマンド) **85**
unary_string_operation (組込述語) **279**, 288
unbound (組込述語) **265**
:- undefine 243
undefined_module 148, 298
undefined_predicate 148, 298
undelete (ディレクトリへのメッセージ) **367**
undelete (リスナ コマンド) **101**
ungetb (ウインドウへのメッセージ) **361**
ungetb (入力ファイルへのメッセージ) **366**
ungetb (標準入出力へのメッセージ) **378**
ungetc (入力用フィルタへのメッセージ) **340**
unification failure 147, 298
unify 79, 80
unify (変数プールへのメッセージ) **348**
unify_list 295
unify_vector 295
unload **52**
unload (シェル コマンド) **52**
unload (モジュール)
:do (述語) 396, **396**
unload:package (シェル コマンド) **52**
unload:program (シェル コマンド) **52**
unloader **52**
unparse (Unparser の述語) **440**
Unparser (モジュール)
:unparse (述語) **440**
unsetenv (ジェネラルリクエストへのメッセージ) **407**
unsetenv (シェル コマンド) **23**
unsetenv (リスナ コマンド) **94**
unwrap 339, **348**
unwrap (variable の述語) **350**
up (インスペクタ コマンド) **165**
update_module (モジュールデータベースへのメッセージ) **389**
update_module (モジュールデバイスへのメッセージ) **392**
usage (シェル コマンド) **29**, **30**
user (ジェネラルリクエストへのメッセージ) **404**

V

v [*value*] (リスナ コマンド) **98**
value (リスナ コマンド) 67, 72, **98**, 137, 143
varchk (シェル コマンド) **32**, 74
variable (モジュール)
:pool2 (述語) **348**
:unwrap (述語) **350**
:wrap (述語) **349**
:wrap_address (述語) **349**
variable pool 71
vector 274
vector (組込述語) **265**, **274**
vector_element (組込述語) **275**
Verifier (モジュール)
:inter_file_verify (述語) **441**
:verify (述語) **440**

verify (Verifier の述語) **440**
view selector **187**

W

wait (組込述語) **264**
warning (ライブラリメッセージ出力へのメッセージ) **443**
What (パラグラフ コマンド) **198**
What × When (パラグラフ コマンド) **195**
What × Where (パラグラフ コマンド) **196**
what manipulator **192**
When × Where (パラグラフ コマンド) **197**
where (タイマへのメッセージ) **370**
who (シェル コマンド) **29**
Window (パラグラフ コマンド) **199**
window (ジェネラルリクエストへのメッセージ) **334**, **404**
:- with_macro 255
wrap 233, 248, 249, 252, 339, 340, **348**
wrap (variable の述語) **349**
wrap_address (variable の述語) **349**
write mode 73

X

xor_string_elements (疑似組込述語) **281**
xref (リスナ コマンド) **102**

あ

アテンション → デバイスアテンション, 378
アテンションキー **12**, **65**, 75, 110, 132, 133, → インスペクタの出力の中断
アテンションキー (シェル) **16**, **24**
アテンションライン → デバイスアテンション
アトム 283
　キャッシュ
　　データベース 385
　データベース 385, 404
　デバイス 386
　表記 227
　リクエストデバイス 386
アペンドモード 73
アボート → デバイスアボート, 378
アボートライン → デバイスアボート
アンパーサ 436, 438, 440
暗黙の引数 237
　展開の抑制 **242**
アンローダ **52**

い

イベント番号 **58**, 64
イベント番号 (シェル コマンド) **13**
イベント番号 (リスナ コマンド) 64, **82**
イベント番号リスト (リスナ コマンド) 64, **83**
インクリメンタル GC **74**, 76
インスペクタ 152
　解析位置の指定 159
　環境変数 155
　監視プロセス **173**
　起動 97, 107, 135, 142, 152
　コード形式 153
　コマンド 161
　　一覧 180
　　環境 — 170
　　監視 — 167
　　基本 — 161
　　調査 — 163
　　入力 153
　　変数プール操作 — 172
　　モニタ — 167
　終了 155
　出力制限値 **153**
　出力の中断 155
　入出力先の切り換え 154
　表示形式 156
　変数プール 153
　ホロフラスティング **153**
　文字印字形式 153
　文字列出力形態設定 153
　モニタ 173
インスペクタ コマンド
　a_little_bigger **163**
　big [*bigger*] **162**
　bigger **162**
　cc [*change_condition*] **177**
　change_condition **177**
　cm [*conditional_monitor*] **168**
　cmf [*conditional_monitor_and_fork*] **168, 169**
　com [*commands*] **161, 162**
　commands **161, 162**
　conditional_monitor **168**, 174
　conditional_monitor_and_fork **168, 169**, 174
　down **165**
　env [*environment*] **172**
　environment **172**
　ex [*exit*] **162**
　exit **162**
　f [*fork*] **162**

fork 154, **162**
help **161**
history **166**
i [*inspect*] **162**
inspect **162**
lbig [*a_little_bigger*] **163**
len [*length*] **166**
length **166**
list **172, 173**
ls [*list*] **172, 173**
m [*monitor*] **167**
me **163**
mf [*monitor_and_fork*] **167, 168**
monitor **167**, 174
monitor_and_fork **167, 168**, 174
monitored_terms **170**, **178**
mt [*monitored_terms*] **170**, **178**
n [*name*] **172**
name **172**
pal [*print_atom_length*] **171**
pd [*print_depth*] **170**
pl [*print_length*] **171**
pm [*print_mode*] **172**
print_atom_length 154
print_depth 154, **170**
print_length 154, **171**
print_mode 154, **172**
print_string_length 154, **171**
print_string_mode 154, **171**
psl [*print_string_length*] **171**
psm [*print_string_mode*] **171**
pu [*push_root*] **167**
push_root **167**
qm [*query_mode*] **169**, **178**
query_mode **169**, 175, **178**
sam [*stream_auto_mode*] **169**, **178**
set_title **162**
sh [*show*] **166**
show **166**
silent_mode **169**
sm [*silent_mode*] **169**
stream_auto_mode **169**, 175, **178**
u [*up*] **165**
up **165**
ピリオドリターン **173**
空 [*down*] **165**

う

ウインドウ **360**, 404
　アテンション **356**
　アテンションライン **356**
　アボート **356, 357**
　アボートライン **356, 357**
　生成 359
　入力残留文字列 357, **360**
　メッセージ **360**
　　属性設定 **361**
　　属性問い合わせ **363**
　　入出力 **360**
ウインドウリクエストデバイス **359**
　生成 334
　メッセージ **359**
受取り確認メッセージ 292, 293

え

永久中断 → デッドロック
エコー 30
エラー
　再表示 144
　処理 138
　履歴 86
エラーキュー 86, 144
　長さの設定 86
エラー報告 428
　方法 428
エラーメッセージ
　ソータ 330
　プール → プールのエラーメッセージ
　マージャ 330
　リスナ 144
演算
　ストリング 278
　整数 267
　多倍長整数 274
　倍精度 273
　浮動小数点数 271
演算子 → オペレータ, → オペレータ

お

オーナ 410
オーバフロー 297
オフセット 296
オペレータ 73, 226, 346
　削除 88, 343
　追加 88, 343
　取り出し 343
　表示 88
　標準の — 347

オペレータプール 339, 342, 346
親ゴール 118

か

ガード内 OR 225
ガーベジコレクタ 74, 115
　起動 **26**, 65, **75**
改行 233
書込権 **410**
格納 → プール
加減乗除 → 演算
型不正
　入力引数の — 297
稼働状況 181
カレントディレクトリ 365
簡易ログイン機能 208
環境変数 93, 206, **401**, 404
　compile
　　:auto_link **46**
　　:default_file_extension **47**
　　:distribution **47**
　　:language_options **47**
　　:link_error **47**
　　:ls_coulmn **48**
　　:message **48**
　　:more_prompt **48**
　　:original_package **48**
　　:parallel **48**
　　:parsers **48**
　　:path **48**
　　:preprocessors **48**
　　:prompt **48**
　　:top_modules **48**
　　:unparsers **48**
　　:verifiers **49**
　　:working_packages **49**
　inspector
　　:initial_window_size **156**
　　:print_string_mode **156**
　　:window_size_increment **156**
　　:window_size_increment_l **156**
　monitor
　　:query_mode **175**
　　:silent_mode **176**
　　:stream_auto_mode **175**
　pmeter
　　:display **183**
　　:display_node **184**
　　:display_password **184**
　　:display_username **184**
　　:gateway_node **184**
　　:interval **183**
　　:item **184**
　　:timer_node **183**
　　:timer_password **183**
　　:timer_username **183**
　task
　　:directory **3**, **23**, **49**, **59**, 365
　　:fep **4**
　　:home **4**, **24**, **59**
　　:io_node **4**, **24**, **59**, 206
　　:package **3**, **24**, **49**, **59**, 60, 90
　　:user **4**
　削除 94, 407
　参照 94, 407, 408
　設定 94, 407
　表示 93
漢字 233

き

キー付きプール 305
キーなしプール 305
疑似組込述語 248
　and_string_elements **281**
　append_string **282, 283**
　as_a_clause **248**, 256
　as_a_goal **248**, 256
　as_an_argument **248**, 256
　complement_string_elements **282**
　copy_string_elements **281**
　in_the_body **249**, 256
　in_the_guard **248**, 256
　in_the_head **248**, 256
　move_string_elements **282**
　nand_string_elements **281**
　nor_string_elements **281**
　not_xor_string_elements **281**
　or_string_elements **281**
　set_substring **280**
　string_tail **282**
　substring **279**
　xor_string_elements **281**
基数の変換 111, 233
行数 31
強制終了 407, 411
局所述語 62
　実行 81
極大ゴール 74, 76

く

組込述語 264
add **267**
add_bignum **274**
add_extended **273**
and **269**
apply **284**, 390
apply_spying **284**
apply_tracing **284**
atom **264**
bignum_less_than **273**
bignum_not_less_than **274**
binary_string_operation **278**, 288
calculate_priority_from_current **286**
calculate_priority_from_minimum **286**
change_element_size **277**
code **265**
code_to_predicate **284**
complement **270**
current_node **287**
current_priority **286**
devide_extended **273**
diff **266**
display_console **285**
divide **268**
equal **267**
exclusive_or **270**
floating_point **264**
floating_point_add **271**
floating_point_divide **272**
floating_point_equal **271**
floating_point_less_than **270**
floating_point_multiply **272**
floating_point_not_equal **271**
floating_point_not_less_than **270**
floating_point_subtract **271**
floating_point_to_integer **272**
get_console **286**
get_heap_size **287**
hash **287**
integer **264**
integer_to_floating_point **272**
less_than **266**
list **265**
merge **285**, 396
module **265**
module_element **283**
modulo **268**
multiply **268**
multiply_bignum_by_word **274**
multiply_extended **273**
new_atom **283**
new_string **276**
new_vector **275**
not_equal **267**
not_less_than **267**
or **269**
predicate_to_code **283**
put_console **286**
read_console **285**
search_character **277**
search_characters **278**
set_string_element **277**, 288
set_vector_element **275**, 288
shift_left **269**
shift_right **269**
shifting_string_operation **279**, 288
string **265**, **276**
string_element **276**, **277**
string_less_than **276**
string_not_less_than **276**
subtract **268**
subtract_bignum **274**
subtract_extended **273**
system_timer **287**
unary_string_operation **279**, 288
unbound **265**
vector **265**, **274**
vector_element **275**
wait **264**
デッドロック 76
呼び出し 62
クライアントサーバ機能 412
クラスタ 296
グループ 207, 210, 410
ID 210, **424**
オーナ 410
代表 — ID **427**
定義ファイル 209, **210**
デバイス 427
クローズ番号 117, **119**, 299

け

継承
パッケージの — 386
マクロの — 256
計測 291

時間間隔 111
言語定義資源 290, 398

こ

コード 283
ゴール
置き換え 290
起動 61
再実行 64, 82, 83
再表示 64, 81, 82
番号 117
未実行 — の一覧 67, **132**
公開述語 61
実行 81
高階述語 298
項の表示 86
子ゴール 113, 117
通し番号 117
子資源識別子 → 資源識別子
コピー 29
コンパイラ **33**
環境変数 46
初期化 34
シンタックス 36
セルフ — 33
パラメタファイル 34
コンパイラ コマンド
!! **38**
!Relink **43**
!Unload **44**
!alias **44**
!auto_link **39**
!cd **38**
!compile **37**
!default_extension 34, **38**
!define **42**
!describe **42**, 42
!distribute 34, **41**, 47, 53
!exit **45**
!get **42**
!help **45**
!keep 34, **45**
!klb **37**
!link 39, **39**
!link_error **39**
!load **43**
!ls **38**
!make **37**
!message **43**
!original_package **39**, 48
!package **40**
!parallel 34, **43**
!parser **40**
!path **38**
!preprocess **40**
!pwd **39**
!relink **44**
!revert **45**
!set 42, **43**
!show **45**
!take 35, **46**
!top_modules **40**, 44
!translate **41**
!unalias **46**
!undefine **42**
!unload **44**
!unparser **41**
!verifier **41**
!verify **41**
!working_packages 40, **44**, 44
コンパイル
インタラクティブ — 35
コマンド **37**
条件 — パラメタ 42
バッチ — 35

さ

サーバ 398, 399
実現方法 **412**
制御ストリーム **421**
サービス
開始 420
終了 420
登録 404, → サーバ, 418
再表示 137, 143
再リンカ **50**, 257
再リンク **50**, 257
サスペンション数 70, 108
サスペンド 294
里親 294
サブゴール → 子ゴール
算術演算 235, 264, → 演算
算術演算マクロ 264

し

ジェネラルリクエストデバイス 120, 398, 399, **403**
シェル **7**

印字パラメタ **15**
カレントディレクトリ **14**
環境変数 **23**
ゴールの実行 7
コマンドヒストリ **13**
コマンドファイル **24**
コマンド別名 **16**
シェル環境変数 **23**
シェル変数 **13**
初期化ファイル 25, **25**
ジョブ → ジョブ
タスク間通信 7
パイプ 7, **11**
標準入出力 7, **10**, 10, 11
シェルウインドウ 8
ログインファイル **25**, 25
シェルユーティリティプログラム **26**, 112
ガーベジコレクタ **26**
コピー **29**
資源木 **27**
シャットダウン **28**
タスク **28**
変数チェッカ **31**
モジュール使用方法 **29**
ユーザ管理 **28**
シェル コマンド
! イベント番号 **14**
! 文字列 1 **14**
!! **14**
alias **16**
back **12**
cat **29**
cd **14, 15**
change_password **29**
compile **33**
convert_macro **263**
dirs **15**
do **16**
echo **30**
exit **12**
fore **12**
gc **27**
get **13, 14**, **16**, **22–24**
getenv **23**
grep **30, 31**
help **13**
history **13**
inspect **22**
ipl_versions **219**
kill **12**
last_error **22**
lc **31**
listen [*listener*] **58**
listener **58**
listener:listener **59**
load 39, 43, **53**
ls **31**
macro **259**
module **54**
module:info **54**
module:remove **55**
module:where **55**
package:define **53**
package:listing **54**
package:protect **54**
popd **15**
pushd **15**
pwd **15**
relink 39, 43, 44, **50, 51**, 258
relink:package **51**
relink:package(Package) **51**
rs **27**
rsall **27**
seqgc **27**
set **13–15**, **22–24**
setenv **23**
sh **13**
shell **13**
shutdown **28**
status **12**
stop **12**
su **29**
suspend **13**
take **24**
tee **29**
ts **28**
tsall **28**
unalias **16**
unload **52**
unload:package **52**
unload:program **52**
unsetenv **23**
usage **29, 30**
varchk **32**, 74
who **29**
イベント番号 **13**
資源 **398**
OS 定義 — 398

与え過ぎ 293
階層 402
獲得 402, 403, 415, 418
管理 290
言語定義 — 290, 398
サービス表 **400**
識別子 **401**, 408, 411
消費量 28, 65, 70, 75, 88, 290, 293, 409, 411
上限接近 293
問い合わせ 65, 292
報告 293
追加 292
提供 418
不足
報告 293
報告許可 292
資源木 **401**
表示 27, 28
時刻 287, 352, 353
事象報告メッセージ 292–294, 296
システムタイマ 287, 368
システム定義サーバ 211, **398**, → サーバ
実行
開始 291
環境 92
起動 7, 58
計測 70, 108, 291, 292
継続 12, 65, 121, 141, 142, 291
計測結果 294
再開 → 継続
時間 65, 70, 88
終了 292
状況 12, 70
制御 290
中断 **12**, **65**, 75, 110, 127, 132, 133, 141, 292
停止 12
トレース 113
ノード指定 22, 63, 81, 118, 149, 229
放棄 12, 67, 110, 127, 140, 141, 292
保留 **129**
ゴールの一覧 67, **130**
優先度指定 23, 63, 81, 118, 149, 228, 291
ログ 119
実行権 **410**
失敗 298
自動スパイ 119, 126
自動トレース 119, 122
シャットダウン 28, 206
終了 70
述語 223, 234
モジュール間の呼び出し 223
呼び出し元 **296**, 302
例外時の — 情報 302
出力ファイル
オープン 364
メッセージ **365, 366**
出力用フィルタ 345
出力用メッセージプロトコル 340
出力量の制限 → ホロフラスティング
荘園 **290**
識別子 118
制御ストリーム 290, **291**
生成 **290**
トレース 120
報告ストリーム 290, **292**
優先度指定 291
条件コンパイル 234, 242
パラメタ 42, 243
初期実行ファイル **70**
初期タスク 205
ジョブ
アテンション **12**
エラー処理 **21**
強制終了 **12**, **21**
資源消費 **22**
実行ノード **22**
実行優先度 **23**
中断 **12**
停止 **12**
バックグラウンド **8**, 11, 12
フォアグラウンド **8**, 11, 12
新ゴール 142
シンタクス 223

す

数式 234
ストリーム 285
ストリング 276
演算 278
検索 30
比較 275
表記 227, 233
マクロ 279
ユーティリティ 279
要素サイズの変換 277
要素の検索 277

ストリング入出力 **371**, 404, 430
生成 371
メッセージ **371**
ストリング入出力リクエストデバイス **371**
生成 335
メッセージ **371**
スパイ 103, 114, 119, 124–126
一覧 105, 106, 134, 135
解除 103, 105, 133, 134
ゴール 118
識別子 300
設定 104, 133, 134
マーク 118
抑制 103
例外 284, 300

せ

制御メッセージプロトコル 343
整数
値の表示 111
演算 267
桁溢れ 297
倍精度演算 273
比較 266
表記 227
浮動小数点数への変換 272
性能評価 181
絶対資源識別子 → 資源識別子
絶対パス名 **365**
節の優先関係指定 224
節番号 117, **119**, 299
ゼロによる除算 297

そ

ソータ 328
エラーメッセージ 330
キー付き 328
キーなし 328
ソート → ソータ
相対資源識別子 66, 67, → 資源識別子
相対パス名 **365**
祖先ゴール 117
番号 117
表示形式 107, 138

た

代替ゴール 142
代表グループ ID **427**
タイプチェック 264
タイマ 287, 351–353, **369**, 404
時間単位 368
生成 368
メッセージ **369**
タイマリクエストデバイス **368**
生成 334
メッセージ **368**
タグ → 例外タグ
タスク 399, **400**, 404, 406, 410
環境ファイル 206
木構造の表示 28
識別子 118, **121**, 411
状態 66, 408, 411
問い合わせ 66
制御ストリーム 406, 410
タスク間通信 412
トレース 120
報告ストリーム 406, 412
多倍長整数
演算 274
比較 273

ち

逐次型言語 113
逐次実行 63, 81
チューニング 181
中断 292

つ

追加ファイル
オープン 364
メッセージ **365, 366**

て

停止 410, 412
ディレクトリ 31, **367**
一覧 100
移動 98–100
獲得 363
格納 99
スタック 99
内容 99
定義 100
取り出し 99
パス名 365
表示 99, 100
メッセージ **367**
デッドロック
検出 65, **74**, 298, 299

種類 75
デバイス **355**
アテンション **356**
アテンションライン **356**, 356
アボート **356**
アボートライン 356, **356**
メッセージ一覧 373
デバッグ
情報 299
デフォルトトレースモード 60, **118**, 121
一覧 107, 135
解除 107, 135
設定 106, 135
デフォルトパッケージ 39, 49, 60, 90
デフォルトモジュール 90, 104, 105
電卓 111

と

特権グループ **424**
特権ユーザ 29, 403, **424**, 425
トレーサ 113
コマンド 121
番号 117, **120**, 127, 128
トレース 113
解除 103
ゴール 117
識別子 299
実行 102
終了 127
出力 **115**
ビット 121
方式 113
マーク 117, **118**, 121
モードの反転 123, 124
抑制 103, **113**, 122
例外 284, 299

に

日時 351–353
表示 87
入出力 285, 338, 398
管理 332
資源 333
獲得 333
制御 354
アテンション 356
アテンションライン 356
アボート 356
アボートライン 356
パス名 **365**
通信方式 355
メッセージ形式 355
ユーティリティ 338
入力ファイル
オープン 364
メッセージ **365, 366**
入力用メッセージプロトコル 338
入力用フィルタ 344
入力履歴 64

ね

ネットワーク環境
設定 215
LIA-net **216**
TCP/IP **215**

の

ノード **229**, 287
稼働状況 110
稼働内容 110
指定 22, 63, 81, 118, 149, **229**
情報 294, **296**
通信状況 110
番号 117
利用範囲 90

は

パーサ 435, 436, 438, 439, 441, 444
バージョン 218
パスワード 4, 209
変更 29
バックグラウンド **8**, 11, 12, 68, 87
パッケージ 24, 36, 39, 44, 48, 49, 51–54, 251, 386, 393
組込 — 394
継承 386, 393
表示 54
ユーティリティ 53, 394
ハッシング 287, 305
バッファ 338
バッファリングフィルタ 338, 344, 345
パフォーマンスメータ **181**
パブリック宣言 61
パラグラフ **185**
起動 111
パラグラフ コマンド
& **199**
amonitor **182**

do_it **198**
exit **200**, **202**
exit 用 (矩形) ボタン **198**
History **198**
load **201**
new_win **198**
Num **199**
Overall What × When **195**
Overall What × Where **196**
plot **202**
pmeter **182**
r **198**
remove **201**
renew **201**
reset **200**
reset* **200**
Result **197**
rmonitor **182**
s **198**
save **201**
set **199**
set* **199**
shell **199**
To **199**
What **198**
What × When **195**
What × Where **196**
When × Where **197**
Window **199**
パラメタ
条件コンパイルの — 243
範囲不正
インデックスの — 297
入力引数の — 297

ひ

比較 235, 303
ストリング 275
整数 266
多倍長整数 273
浮動小数点数 270
引数
個数不整合 298
例外時の — 296
ヒストリキュー 64, **64**, 81–83
長さの設定 83
日付 351, 353
否定の略記 225
表示
オペレータの — 88
項の — 86
日時の — 87
標準演算子表 347
標準入出力 64, **73**, **375**
デバイス 73
標準メッセージ出力 **383**
利用方法 **376**, 377
資源の指定 **379**
使用例 383
ストリーム獲得 377, 379, 381, 383
制御 **381**
プロトコル 377, 378, 383
標準入出力資源 **398**, 399, 401, 422
獲得 409
設定 405
標準メッセージ出力 383
表示量の制限 → ホロフラスティング
ピリオドリターン (インスペクタ コマンド) **173**
ピリオドリターン (リスナ コマンド) 72, **96**

ふ

プール
メッセージ一覧 327
エラーメッセージ
multiply keyed bag 323
キー付きプール 314
キーなしプール 314
キー付き
keyed bag **309**
keyed set **311**
keyed sorted bag **312**
keyed sorted set **313**
キーなし
bag **306**
queue **307**
sorted bag **308**
stack **307**
複数キー付き
multiply keyed bag **315**
ファイル **365**, 404
一覧 31, 100
オープン 363
回復 101
コピー 29
削除 100
属性 101
名前の取り出し 100
パス名 365

抹消 101
メッセージ **365**
ユーティリティ 29, 30
ファイルリクエストデバイス **363**
カレントディレクトリ 365
生成 334
メッセージ **363**
フィルタ 338, 344, 345
フォーク 127, 128
スパイ 103, **114**, 125, 126
フォアグラウンド **8**, 11, 12
負荷分散 → ノード指定
複数キー付きプール 305, 315
複数ゴールの実行 63
復帰 233
浮動小数点数
演算 271
整数への変換 272
比較 270
表記 227
無効演算 297
プリプロセッサ **35**, 435, 436, 438, 439
連鎖 40
プログラム
コンパイル 33
ファイルからの読み出し 53
保存 52
プロセッサ → ノード
プロテクションフィルタ 68
プロトコル 445
PIMOS の — 449
定義言語 445
システム定義プロトコル 445
文法 445
プロファイル 108, 400, 406, 411
機能 291
収集 292
終了 294
情報 294
情報変換ユーティリティ 295, **302**
報告 294
モード 117, 119
フロントエンドプロセッサ 403, 407
プロンプト
シェルの — の表示 14
シェルの — の変更 14
リスナの — の変更 84

へ

並列言語 113
並列実行 63, 81, → ノード指定
ベクタ 274
表記 227
ボディユニフィケーション 295
別名 → alias
ベリファイア 74, 436, 438, 440
変数
値
解放 95, 96
記憶 71, 94, 95
表示 70, 73, 97
一覧 67, 72, 98, 136, 137, 143
解放 72
出力形式 91, 137, 143
チェッカ 31, 74
問い合わせ 73, 97
同一性 115
表記 227
表記法 71
プール **71**, 75, 94, **153**, 339, 348

ほ

保護モード **410**
ボディユニフィケーションの失敗 298
ホロフラスティング 343, **343**, 344
インスペクタでの — **153**
シェルでの — **15**
リスナでの — 73, 91, 92, 97, 137, 143

ま

マージ → マージャ
マージャ 285, 329
エラーメッセージ 330
キー付き 329
キーなし 329
組込述語 285
デッドロック 76, 299
不正入力 298
マクロ **232**
:- inherit_macro 256
旧仕様 — の取り扱い 258
括り出し — 251
継承 256
コンパイル規則 256
算術演算 264
システム定義の — 232
ストリング 279
定義記法変換 258

展開の抑制 60, 89, 232
モジュール 254
継承宣言 256
宣言 255
適用宣言 246, 255
変更 60, 89
ユーザ定義の — 232
マクロ展開 **232**
マルチウインドウトレース 120, 127, 128

み

未実行ゴールの一覧 67, **132**
未定義述語 141, 142, 298
未定義モジュール 141, 142, 298

む

無限長バッファ 338, 346
無効演算
浮動小数点数の — 297

も

モード 222
文字コード
表記 233
モジュール 223, 283
参照関係 102
情報表示 101, 142
使用方法の表示 29
データベース 386, 404
デバイス 390
表記 228
ユーティリティ 54, **393**
リクエストデバイス 391
モジュール間
呼び出し 223
モジュール名 223, 234
取り出し 302
モニタ
一覧 67, 136
解析位置の指定 159
環境 175
質問モード 173, **175**
ストリーム自動監視モード 173, **175**
環境変数 175
監視プロセス **173**
入出力先の変更 175
有効期限 175
起動 108, 135, 136, 174
コマンド 177
コマンド一覧 180
コマンド入力 174
終了 176
表示形式 156
モード 136

ゆ

ユーザ 207, 404
ID 209, **424**
環境初期化ファイル 3
管理 28, 209
情報の管理 424
定義サーバ 206, **398**
定義ファイル 207, 209, **209**
デバイス 424, 426
リクエストデバイス 425
ログイン — の表示 28
ユーティリティ 303
格納 305
キー付きプール 305
キーなしプール 305
木構造 312, 313
ストリング 279
ソート 328
タイマ 350
時刻 352, 353
単位変換 353
日付 351, 353
ハッシュ 305
ハッシュ表 309, 311
比較 303
プール → プール
複数キー付きプール 305, 315
マージ 329
優先度 117, 228, 286, 291
指定 23, 63, 81, 118
所属荘園内自己相対指定 229
所属荘園内割合指定 229
ユニフィケーション 234
失敗 298
ベクタ同士 295
リスト同士 295

よ

読出権 **410**

ら

ライトモード 73
ライブラリ 394

り

リードモード 73
リクエストデバイス **354**
  ウインドウ 359
  ストリング入出力 371
  タイマ 368
  ファイル 363
リスト
  表記 227
  ボディユニフィケーション 295
リスナ **57**
  ウインドウ **58**
  エラー処理 138
  エラーメッセージ 144
  オペレータ 88
  環境変数 59
  起動 58
  コマンド一覧 87
  コマンドファイル 58, 59, **70**, 84
  終了 70, 88
  初期実行ファイル 58, 59, **70**
  デフォルトトレースモード 60
  デフォルトパッケージ 60, 90
  デフォルトモジュール 90
  トップレベル **58**, 67, 68
  マクロ展開 60, 89
リスナ コマンド
  ! 64, **81, 82**
  !! 64, **82**
  add_op **88**
  alias **85, 86**
  alias_expansion **86**
  attributes **101**
  calc [*calculator*] **111**
  calculator **111**
  cd **98, 99**
  change_macro **89**, 246
  change_nodes **90**
  com [*commands*] **87**
  commands **87**
  con **100**
  da [*display_ancestor*] **107**
  date **87**
  def **100**
  default_module **90**
  default_package **90**
  delete **100**
  deleted **100**
dfm [*default_module*] **90**
dfp [*default_package*] **90**
dirs **99**
disp [*display*] **86**
display **86**
display_ancestor **107**, 138
display_variables 70, 73
dv [*display_variables*] **97**
env [*environment*] **92, 93**
environment **92, 93**
error_queue_length **86**, 144
errors **86**, 144
ex [*exit*] **88**
exit 67, 68, 70, **88**, 141
expunge **101**
fgt [*forget*] **95, 96**
forget 73, **95, 96**
getenv **94**
Goal @ node 63, **81**
Goal @ priority 63, **81**
Goal @ relative_priority **81**
Goal1, Goal2 63, **81**
Goal1; Goal2 63, **81**
he [*help*] **87**
help **87**
his [*history*] **83**
history 64, **83**
history_length 64
hl [*history_length*] **83**
in **111**
info **101**
ins [*inspect*] **107**
inspect **107**
inter **112**
interval **111**
l [*list*] **98**
last_error **86**, 144
list 67, 72, **98**, 136, 143
listing **100**
ls [*listing*] **100**
m [*monitor*] **108**
mf [*monitor_and_fork*] **108**
ModuleSpec ! Local 62, **81**
ModuleSpec : Public 61, **81**
monitor **108**
monitor_and_fork **108**
nospy 134
nostatistics 70, **89**
notrace **103**, **107**, 115, 135

notrace_all **107**, 135
nst [*nostatistics*] **89**
ntr [*notrace*] **103**, **107**
ntr_all [*notrace_all*] **107**
number **111**
op [*operator*] **88**
operator **88**
out **112**
pal [*print_atom_length*] **91**
paragraph **111**
pd [*print_depth*] **91**
pg [*paragraph*] **111**
pl [*print_length*] **91**
popd **99**
predicates **101**
print_atom_length **91**, 97, 137, 143
print_depth **91**, 97, 137, 143
print_length **91**, 97, 137, 143
print_string_length **91**, 97, 137, 143
print_variable_mode **91**, 97, 137, 143
printenv **93**
profile 70, **108**
profile_msg 70, **110**
profile_node 70, **110**
prompt **84**
psl [*print_string_length*] **91**
purge **101**
pushd **99**
pv [*print_variable_mode*] **91**
pwd **99, 100**
query 73, **97**
rem [*remember*] **94, 95**
remember **94, 95**
remove_op **88**
rewrite_depth **92**
rewrite_length **92**
rwd [*rewrite_depth*] **92**
rwl [*rewrite_length*] **92**
setenv **94**
sf [*spy_fork*] **103**
si [*spying*] **105, 106**
spy 134
spy_fork **103**, 115
spy_reduction **103**, 115
spying **105, 106**, 134, 135
sr [*spy_reduction*] **103**
st [*statistics*] **88, 89**
statistics 70, **88, 89**
suspend 59, **87**
take 70, **84**
tr [*trace*] **102**, **106**
trace **102**, **106**, 115, 118, 135
tracing **107**, 135
tri [*tracing*] **107**
unalias **85**
undelete **101**
unsetenv **94**
v [*value*] **98**
value 67, 72, **98**, 137, 143
xref **102**
イベント番号 64, **82**
イベント番号リスト 64, **83**
ピリオドリターン 72, **96**
リダクション **113**
失敗 298
数 65, 70, 88, 108, 290
問い合わせ 292
報告 293
リダクションスパイ 103, **114**, 124, 125

れ

例外 **296**
一覧 296
種類 296
処理 **290**
ユーティリティ 296, **302**
スパイ 284
生起 300
タグ 290, 291, 301
ビット位置 301
ビット位置割り当て 301
マスク 121, 290, 291, 302
トレース 284
発生 300
発生場所 **296**, 302
報告 **296**

ろ

ローカル述語 62
ローダ **53**
ロード **53**
ロギング 209
ログイン 1
簡易 — 機能 3, 208
区分 2, 208
通常の手順 1
デバイス 428
ハンドラ 207, 209

ファイル 1
論理ディレクトリ名 100
一覧 100

わ

割り込み 12, 65

空

空 [*down*] (インスペクタ コマンド) **165**