ICOT Technical Memorandom: TM-0945

TM-0945

PDSS マニュアル (Version 2.52.00)

平野　喜彦

August, 1990

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

ＰＤＳＳマニュアル

(Version 2.52.00)

1989年9月19日

新世代コンピュータ技術開発機構

第四研究室

目次

2.51.07 版 ⇒ 2.52.00 版 (1989/9/19)

1. 組込述語 unbound/2 を 2 引数に変更した。
   unbound(+X,-Rpenum,-Raddress,-NewX) → unbound(+X,-Result)

2.51.04 版 ⇒ 2.51.07 版 (1989/7/25)

1. デッドロック報告の仕様変更
   荘園のレポートストリームに出力されるデッドロックメッセージの仕様を変更した。従来は全デッドロックゴールを報告していたが、新仕様ではデッドロックの因果関係を解析し、ルートとなっているゴールだけを報告することにした。
2. 組込述語 unbound/4 の仕様変更
   組込述語 unbound/4 の第 3 引数に出力されるアドレスを絶対アドレスから、ヒープ領域の先頭からの相対アドレスに変更した。

2.51 版 ⇒ 2.51.04 版 (1989/6/6)

1. Micro PIMOS ウィンドウ / ファイルの入出力コマンド
   以下の入出力コマンドが追加 / 変更された。
   - add_op/3 で、定義済みのタイプと共存できない場合 (fx ↔ fy, xf ↔ yf, xfy ↔ xfx ↔ yfx)、古い定義を削除するようにした。
   - remove_op(Op) が追加された。
2. Micro PIMOS コマンド・インタプリタ
   - add_op/3 で、定義済みのタイプと共存できない場合 (fx ↔ fy, xf ↔ yf, xfy ↔ xfx ↔ yfx)、古い定義を削除するようにした。
   - remove_op(Op) が追加された。
   - varchk コマンドでファイル名をウィンドウの先頭に表示するようにした。
   - シェルの環境変数 plength の初期値を 20 に変更した。

2.50 版 ⇒ 2.51 版 (1989/5/19)

1. 例外メッセージの仕様を変更した。
   Multi-PSI/V2 に合わせて、例外メッセージを以下のように変更した。レポートストリームに流される例外メッセージは
   deadlock 以外の全て:

   ```
   exception(ExcpCode, Info, ^NewCode, ^NewArgv)
   ```

   deadlock:

   ```
   deadlock(ExcpCode, Info)
   ```

   の形式である。
   - ExcpCode は例外の種類を表わす正の整数。
   - Info は例外情報で、例外の種類により異なる。
   - NewCode,NewArgv には例外を起こしたゴールの代わりに実行して欲しいゴールのコードと引数をユニファイする。
2. 組込述語
   - merge/2 が変更された。マージャは荘園に属するようになり、デッドロックが検出されるようになった。

- predicate_to_code(Mod, Pred, Arity, ^Code) が追加された。
- code_to_predicate(Code, ^Mod, ^Pred, ^Arity, ^Info) が追加された。

3. Micro PIMOS のウィンドウ / ファイルの入出力コマンド

   以下の入出力用コマンドが追加 / 変更された。

   - gett(^Term, ^Status)
   - getft(^Term, ^NumberOfVariables, ^Status)
   - skip(Char)
   - putb(Buffer, Count)
   - replace_op_pool(^OldOpPool, NewOpPool)
   - change_op_pool(NewOpPool)
   - print_length/1, print_depth/1 の長さ、深さの初期値をファイルに対しては 100 、ウィンドウに対しては 10 に変更した (変更前は共に 10)。
   - operator/2 の定義の仕様を 2 要素ベクタ {Precedence, Type} のリストに変更した。

4. Micro PIMOS のコマンド・インタプリタ

   - シェルのコマンドライン (ターム) 中のマクロを展開するようにした。
   - replace_op_pool(^OldOpPool, NewOpPool), change_op_pool(NewOpPool) がシェルコマンドに追加された。

1.60 版 ⇒ 2.50 版 (1989/4/7)

1. コンパイラ関係

   KL1/Prolog コンパイラが新しい仕様のクローズ・インデキシング命令を生成するようになった。この仕様のインデキシングを行うと実行速度の向上が期待される。そのため、KL1/Prolog コンパイラを用いる場合には、インデキシング・モードがデフォールトで ON となるように変更された。また、マクロ展開を抑止するオプションが廃止された。

   なお、KL1/KL1 コンパイラでは、まだクローズ・インデキシング命令を生成しない。

2. 浮動小数点数

   浮動小数点数をサポートするようになった。PDSS では浮動小数点数として、32bit の単精度を採用した。これは、$-3.402823 \times 10^{38} \sim 3.402823 \times 10^{38}$ の値を 10 進数約 7 桁の精度で表現する。浮動小数点数はアトミック・データとして扱われ、ガード (比較, 演算)/ ボディ (演算) の組込述語が追加された。ただし、追加された組込述語の多くは Multi-PSI V2 では使用できないので注意が必要。

   なお、浮動小数点数と整数は別のデータ・タイプとして扱われ、自動的な型変換は行われない。型変換には組込述語を利用する。

   また、整数の場合と同様な数式のマクロが用意されるので、ユーザーがプログラムを書く場合は、それを利用することにより、簡潔に記述する事ができるようになっている。型変換のマクロも用意されている。

3. 組込述語

   以下の組込述語の仕様が変更された。

   - hash/3 の仕様を Multi-PSI に合わせた。
     hash(+X,+Width,-HashValue) → hash(+X,-HashValue,-NewX)
   - apply/3 を Multi-PSI に合わせ、2 引数に変更した。
     apply(+Mod,+Pred,+ArgV) → apply({+Mod,+Pred,+ArgN},+ArgV)
   - unbound/2 を Multi-PSI に合わせ、4 引数に変更した。
     unbound(+X,-Result) → unbound(+X,-Rpenum,-Raddress,-NewX)
   - make_atom/2 の名前、引数の順番を変更した。
     make_atom(+Str,-Atom) → intern_atom(-Atom,+Str)
   - new_atom(-Atom) を追加した。
   - atomic(X) が廃止された。
   - merge_in(+In1,+In2,-Out) をマクロに変更した。
     展開は、merge_in(In1,In2,Out) → Out={In1,In2} のように行われる。
   - 整数の演算でオーバーフローを検出するようにした。
   - その他一般ユーザー用でない組込述語で引数の変更等が行われた。

4. パーザーの仕様を変更した。

   - 数値 (整数、浮動小数点数) のフォーマットのチェックを厳密にした。また、入力時にオーバーフローのチェックも行うようにした。
   - 数値に付ける符号 (+/-) の扱いを変更した。
     "-1" は符号を含めて 1 つのアトミックデータとして扱い、"- 1" は {-,1} というベクタとして読むようにした。ただし、X := Y-1 のように "-1" を 1 つのアトミックデータとして扱うとシンタックスエラーになってしまう場合には、符号を分離してパーズするようにした。
   - $ を特別なエスケープ文字として扱うようにした。具体的には $ に続く英数字列は特別なデータを表すようにした。
     例えば、荘園データ等を印字した場合に、"$Shoen001" という形にし、これを読もうとした場合に、「これは再読み込みができないデータである」と正しくエラーにできるようにした。また、PDSS ではサポートしていないが、浮動小数点数の infinity 等を表現するのにも使う事ができる。

5. Micro PIMOS のコマンド・インタプリタ

   - xref コマンド

     オプションが追加 / 変更された。

6. スケジューラ

   スケジューラの機能が拡張され、ゴールをエンキューする時、乱数により、一部をレディー・キューの最後に入れる事ができるようになった。これは、マルチ PE で実行した場合の実行順序の非決定性をシミュレートし、KL1 プログラムのテストを行う場合に使われる。

   また、スケジューラの制御が立ち上げ時のオプションだけでなく、トレーサーからも制御できるようになった。

7. エミュレータ関係

   エミュレータの多くの部分が変更されたが、ユーザーの使用方法は変更されていない。変更点は主に、MRB-GC をサポートするようになった事と、D-Code の部分である。

1.50 版 ⇒ 1.60 版 (1988/10/25)

1. コンパイラ関係
   a. KL1/KL1 コンパイラが新しいバージョンになった。主な変更点としては以下の項目があり、クローズインデキシングの命令が生成できない点を除き KL1/Prolog コンパイラと同等になった。尚、コンパイルのためのコマンドは変更されていない。
      - KL1/Prolog コンパイラと同等のマクロ機能のサポート。
      - コンパイル速度の向上。
      - Structured Constant 系命令のサポート。
      - コンパイラのモジュール名を変更。全て "kl1cmp_" で始まる名前になった。
   b. KL1/Prolog コンパイラは、ストリング型データに関してだけ Structured Constant 系命令をサポートするように変更。リスト, ベクタは Structured Constant 系命令を使わずに、従来どおり命令により動的にデータを生成する。
   c. コンパイラが Structured Constant 系命令をサポートしたのに伴い Emacs ライブラリの kl1-mode における部分コンパイルの機能を使うとオブジェクトのサイズが大きくなり、"Assembler: Relative address field overflow." のエラーが発生する可能性が大きくなった。これは部分コンパイルのときに使用するツールにおいて、部分コンパイルしたオブジェクトと元のオブジェクトをマージする時に、Structured Constant の部分を単純にアペンドしている為で、もしこのエラーが発生した場合にはファイル全体を再コンパイルする必要がある。
   d. 定数表記のためのマクロが追加された。
      - `string#"文字列"`
        デフォルト・タイプのストリングになる。PDSS では 8bit ASCII ストリング。(Multi-PSI V2 では 16bit JIS 漢字。)
      - `#"文字"`
        デフォルト・タイプの文字コード (整数) になる。PDSS では 8bit ASCII コード。(Multi-PSI V2 では 16bit JIS 漢字。)
      - `c#"文字"`
        ASCII コードになる。(Multi-PSI V2 でも同じ。)
      - `key#lf`
        改行を表すコード (整数 10) になる。(Multi-PSI V2 でも同じ。)
      - `key#cr`
        復改を表すコード (整数 13) になる。(Multi-PSI V2 でも同じ。)
2. Micro PIMOS 関係
   a. compile コマンドの追加。今までの comp コマンド +load コマンド +save コマンドに相当する。
   b. varchk コマンドの拡張。ゴールの表示形式が変更できるようになった。
3. エミュレータ関係
   a. Structured Constant 系の命令の追加。Structured Constant は変数を含まないストリング, リスト, ベクタをコード領域に置き、データを動的に生成しないで使えるようにするもので、処理速度が向上する。ただし、データが共有されることになるので MRB は必ず ON となる。KL1/KL1 コンパイラでは全てのストリングとボディ部のリスト, ベクタで使用され、KL1/Prolog コンパイラではストリングだけで使用される。尚、これによりコードの形式が変更されたので、新しいコンパイラでコンパイルしたコードを古いエミュレータで使うことはできない。上位コンパチなので古いコードはそのまま使うことができる。
   b. スケジューラが拡張され、Breadth First に実行すること可能になった。これは立ち上げ時のオプション (-b) で指定する。ただし、コンパイラは Depth First 用のコードを出すので、ボディ部で複数のゴール

を呼び出した場合の実行順序は、先頭のゴールが最初で (TRO: Execute される)、残りは最後から (ソースコードと逆順) になる。また、Execute で実行できる深さを制限することができ、立ち上げ時オプションの引数で指定する。(例: -b100, デフォルトは 100) この機能はプログラムのバグ (実行順序に依存している) を検出するのに使える。

c. トレーサーが拡張され、次のスパイポイントまでのスキップ / トレースを行う時に次の 4 種類から選べるようになった。(以前は 3 種類)
PDSS ではスパイの指定にコード (述語名 / アリティ) 指定とゴール指定の 2 種類あるので、この組み合わせにより、

- コードがスパイされているものである。
- ゴールがスパイされているものである。
- コードがスパイされているものであるか、または、ゴールがスパイされているものである。
- コードがスパイされているものであり、且つ、ゴールもスパイされているものである。

d. GC 後にヒープエリアの回収量をチェックし、回収量が少ない場合に特別な処理を行うようにした。

- あまり回収できなかった時 (約 5% 以下)
ゴールスタックの一番高いプライオリティの中のゴールの順序を入れかえる。これにより、データを消費するゴールが動きアクティブなデータが減る可能性がある。
- 殆んど回収できなかった時 (GC 直後なのに GC 要求が出てしまっている)
ユーザタスクを強制的にアボートする。

e. ウィンドウの入力待ちで、ctrl-C ctrl-Z や ctrl-C ctrl-T の割り込みをかけると、read_token/4 等の組込述語がデッドロックになっていたのを修正。

f. タイマ割り込みを禁止するオプションを追加。dbx でエミュレータ自身をデバッグする時に便利。

g. 立ち上げ時オプションの指定方法を変更。 − と + による区別を止め、どちらでもデフォルトと違う状態にすることを意味することにした。(+t, -t は今までどうり区別される。)

h. 以下の点が変更され移植性が向上した。

- `alloca()` の使用を止めた。
- `sigmask()` のマクロが `<signal.h>` に定義されていない時には PDSS 側で定義するようにした。
- 大きな構造体変数の宣言で初期値を指定するのを止めた。

4. その他

a. PIMOS 用に開発されたユーティリティが使えるようになった。現在以下のものがサポートされている。

比較	どんな KL1 データでも一意に比較できる機能
ハッシング	標準のハッシュ関数
キーなしの Pool	Bag, Stack, Queue, Sorted Bag
キー付きの Pool	Keyed Bag, Keyed Set, Keyed Sorted Bag, Keyed Sorted Set

1　PDSS とは

PDSS とは “PIMOS Development Support System” の略であり、その名のとおり PIMOS の開発用に作成された KL1 システムである。PDSS はその目的から、なるべく Multi-PSI V2 System 上で実現される KL1 と互換性を保つよう留意されているが、実現方法の相違や速度等の点で一部互換性を損なっている。主な相違点と思われるものを以下に示す。

- アトムの管理などソフトで実現されると思われるものを処理系レベルで行っている。これによって、一部のアトム操作の機能が組込述語として提供されている。
- コードの管理も同様に処理系レベルで行っている。
- PDSS の資源管理機能で対象としている資源とは、リダクション数だけである。
- I/O 等いわゆるデバイス・ストリームがその形態上異なる。
- シングル・プロセッサ・システムなので処理の分散の為のプロセッサ指定機能が無い (指定しても無視される)。

PDSS のもう一つの目的は、並列プログラムの開発用ツールの提供である。このため PDSS は極力マシーンに依存しないコーディング・スタイルで記述されており、種々の UNIX の稼動するマシーンに移植する予定である。またプログラム開発ツールとして使い勝手の良いシステムを目指している。これについては PIMOS の開発とともに機能の向上が望めるものと期待している。

PDSS は主に二つの部分から成るシステムである。一つは KL1 の基本機能を実行する言語処理系であり、他方は Micro PIMOS と呼ばれる KL1 自身で書かれたユーザー・インタフェース部である。Micro PIMOS については 4 章で詳しく述べるが、一言で言えば、I/O 機能やコード管理機能をユーザーに提供する Single User, Single Task の簡易 OS である。図 1 に PDSS の全体構成のイメージを示す。

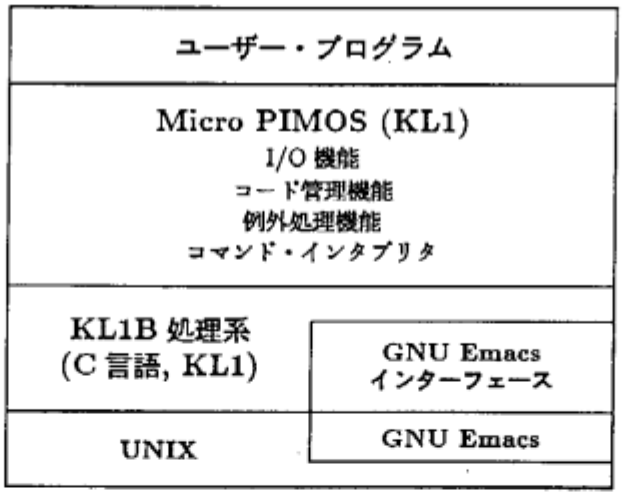

図 1: PDSS の全体構成

図 1 で、GNU Emacs インターフェース部分は GNU-Emacs と呼ばれるフルスクリーン・エディタを利用してユーザーにマルチウィンドウの環境を提供している部分である。この為のライブラリは Emacs-LISP で記述されている。

PDSS では I/O 機能やコードの操作機能は所謂デバイス・ストリームと呼ぶ特別な組込ストリームを基に構築されている。このデバイス・ストリームの機能仕様は付録 -1, 付録 -2 に示したとおりである。しかし、ユーザーはこのデバイス・ストリームを直接獲得し使用する訳ではなく、4 章で述べるように、Micro PIMOS が用意している各種のライブラリを用いてこのデバイス・ストリームの機能を使用することになる。

2　PDSS の使用方法

PDSS の基本的な操作法について解説する。PDSS の起動から、プログラムのコンパイル、実行、デバッグ、終了までの手順を簡単に示す。ここでは、操作の流れを示すことを目的としているため、システムへのコマンド等に関する解説は最小限に留めている。それらについては、後続の章で再度詳しく述べることにする。

PDSS は、単体で使用可能であるが、GNU-Emacs の下での使用も可能である。むしろ、プログラムの開発効率、実行環境を考慮すると GNU-Emacs 下で使用する方が望ましい。

以下では、PDSS 単体、GNU-Emacs 下、各々の使用形態での操作手順について解説する。

2.1　PDSS 単体での操作

KL1 プログラムの開発における一般的なサイクルは以下のようになる。

1. エディタを用いたソースプログラムの作成
2. コンパイル
3. ロード
4. 実行
5. デバッグ

PDSS 単体での操作では、コンパイル以降を PDSS 上で行なうことになる。次のコマンドを実行することによって PDSS が起動する。

pdss *return*

2.1.1　コマンド

PDSS が起動すると、プロンプト "|?-" が表示される。これに対する入力は、起動時に生成されたコマンド・インタプリタによって解釈、実行される。本章で用いるコマンドを以下に示す。その他のコマンドの詳細については、4.1.2「コマンド」を参照。

- comp(FileName,OutFileName)
  FileName で指定される属性 ".kl1" の KL1 ソース・ファイルをコンパイルし、OutFileName で指定される属性 ".asm" のファイルに出力する。
- load(FileName)
  FileName で指定される属性 ".sav" のセーブ・ファイル（これがない場合には属性 ".asm" のアセンブラ・ファイル）をコード領域にロードする。
- save(ModuleName,FileName)
  ModuleName で示されるモジュールの実行コードを、FileName で示される属性".sav" のファイルにセーブする。
- listing
  すでにロードされているモジュールの情報を表示する。
- public(ModuleName)
  ModuleName で示されるモジュールが他のモジュールに公開している述語（public 宣言されている述語）の一覧を表示する。
- halt
  PDSS を終了する。

また、下記の記法を用いることにより、1つのコマンドラインに複数のコマンドを与えることができる。

- カンマ (',')
  前後のコマンドを並列に実行する。
- セミコロン (';')
  前のコマンド群を実行してから、後のコマンド群を実行する。
- 縦線 ('|')
  前におかれたコマンドを実行後、後ろにかかれた変数の値を表示する。変数の代わりに all と書くと全ての変数が表示される。

2.1.2 操作例

ここでは、フィボナッチ数列のプログラムを例にとり、PDSS の起動からプログラムのコンパイル、実行までの操作例を示す。

まず、次のようなプログラムをエディタで作成し、ファイル名 sample.kl1 としてセーブする。

```
:-module sample.
:-public fib/2.

fib(N,R):-true|fib1(N,1,1,T),R = [1|T].

fib1(N,F1,F2,R):-F2 > N|R = [].
fib1(N,F1,F2,R):-F2 =< N|F := F1 + F2,fib1(N,F2,F,T),R = [F2|T].
```

以下は、上記のプログラムを用いた操作例である。《 》内は、ユーザの入力に関する説明である。

```
% pdss   《PDSS の起動》
***** PDSS-KL1 V2.51.00 (Thu May 25 17:38:28 GMT+9:00 1989) *****
.....................................................

*********************************************
**** Micro PIMOS (version 2.5) ************
*********************************************
World is /pdss/pdss2.5/doc
Total heap size is 200000 words
Total code size is 100632 bytes
*********************************************

| ?- comp("sample","sample").  《コンパイル。ファイル sample.asm が生成される。》
"sample" : Compiling ... fib/2,fib1/4,END.
done

[fib/2]
success
yes.
| ?- load("sample").   《ファイル sample.asm をコード領域にロードする。》
"sample" : Loading ... done
```

```
Module name : sample

yes.
| ?- listing.   《ロードされているモジュールの表示》
---------------------------------
 Module_Name  Trace  Saved  SpyPN
---------------------------------
 sample              ?      0
---------------------------------

yes.
| ?- public(sample).   《モジュール sample 内で public 宣言されている述語の一覧表示》
fib/2, END.

yes.
| ?- sample:fib(10,L)|L.   《プログラムの実行》
L = [1,1,2,3,5,8]

yes.
| ?- save(sample,"sample").   《モジュール sample の実行コードを sample.sav にセーブ》
"sample" : Saving ... done

yes.
| ?- halt.   《PDSS の終了》
```

2.2　GNU-Emacs 下での操作

ここでは、おもに Emacs 特有の操作について説明する。その他の PDSS 操作については前節を参照されたい。なお、Emacs での PDSS 用コマンドの詳細は 付録-9 を参照。

PDSS の起動

まず、Emacs を起動する。PDSS は Emacs から次のコマンドを入力することにより起動される。

Meta-X pdss *return*

するとウィンドウが以下の 2 つに分割される。

- PDSS-SHELL
  PDSS のトップレベルのウィンドウである。プログラムのロードや、実行を行なうためのコマンドを入力する。前節の PDSS 単体での操作例は、このウィンドウ上で同じように実行できる。
- PDSS-CONSOLE
  プログラム実行中のエラーを表示したり、トレース実行をしたりするウィンドウである。

プログラムの作成とコンパイル

エディット用のバッファを生成して、そこでプログラムを作成する。このとき、ファイル名には拡張子 ".kl1" を付けること。このプログラムをコンパイルするには、そのウィンドウ上で次のコマンドを入力する。

Ctrl-C Ctrl-C

すると PDSS-COMPILER ウィンドウが現われて、コンパイルが実行される。エラーが検出された場合には、このウィンドウにメッセージが表示される。その場合は、プログラムを修正した後、再度コンパイルする。正常にコンパイルされた時には、次のようなメッセージが表示される。

PDSS KL1 Compiler: Success

プログラムのロードと実行

すべて PDSS-SHELL ウィンドウで行なう。前節と同様に load(FileName) によりプログラムをロードし、実行する。実行中に発生したエラーは PDSS-CONSOLE に表示される。

PDSS の終了

PDSS-SHELL から halt. コマンドを入力することにより終了する。

2.3　プログラムのデバッグ

PDSS では、トレーサによる実行過程のトレース、及びデッドロック情報等を用いてデバッグを行う。ここでは、主としてトレース機能を用いたデバッグの過程を示す。

2.3.1　トレーサ

PDSS が提供するトレース機能は、実行されるゴールに注目したトレースである。トレーサはゴールが次のような状態になった時、そのゴールに関する情報を表示する。この表示が行なわれるタイミングをトレースポイントという。

- ゴール呼び出し
- 変数の具体化を待つための中断
- 中断からの復帰
- ゴールの失敗
- スワップ・アウト
  (割り込み、又はより高いプライオリティがスケジュールされたことによる実行の中断)

KL1 のトレースの方法としては「コードに注目したトレース」と実行中の「ゴールに注目したトレース」の 2 通りが考えられる。

コードに注目したトレースとは、トレースしたいコードが呼び出された時にトレースを行うものであり、モジュールごとにトレース・モードを指定できる。以下ではこれをコード・トレースと呼ぶ。また、更に細かく、特定の述語にだけ注目してトレースを行うこともでき、これをコードのスパイと呼ぶ。

ゴールに注目したトレースとは、生成された各ゴールごとにその子孫のゴール (すなわちそのゴールの実行によって生成されるゴール) をトレースするか、あるいはトレースしないかを指定するものである。以下ではこれをゴール・トレースと呼ぶ。また、特に指定したゴールの子孫だけをトレースするという指定もでき、これをゴールのスパイと呼ぶ。

例を考えてみよう。以下のプログラムで p(X) がゴール・トレースの状態にあり p(Y) がその状態になかったとすると、p(X) から呼び出される q(A,B) と r(B) はゴール・トレースの状態になるが p(Y) から呼び出される q(A,B) と r(B) はゴール・トレースの状態にならない。

```
Goal   :  p(X), p(Y).
Clause:  p(A) :- true | q(A,B), r(B).
```

【注意】Emacs 上で操作している場合、トレース実行中の表示および操作は PDSS-CONSOLE 上で行なわれる。

2.3.2　コマンド

ここでは以下のコマンドが使用されている。

- trace(ModuleName)
  ModuleName で示されるモジュールのコードのトレースモードを ON にする。この時デバッグ・モードが自動的に ON になる。
- notrace(ModuleName)
  ModuleName で示されるモジュールのコードのトレースモードを OFF にする。
- spy(ModuleName,PredicateName,Arity)
  ModuleName で示されるモジュール中の PredicateName/Arity なる述語のスパイを行なう。この時トレース・モード、デバッグ・モードは自動的に ON になる。
- spying(ModuleName)
  ModuleName で示されるモジュール中のスパイされている述語の一覧を表示する。
- nodebug
  デバッグ・モードを OFF にする。

2.3.3　トレーサコマンド

トレーサコマンドの一部を以下に示す。詳しくは、6.3「コマンド」を参照のこと。

- s[COUNT]
  次のトレース・ポイントで止まる。COUNT が指定された場合にはそのステップ数だけトレースを行なったあとで止まる。
- sp[COUNT]
  次のスパイされている述語コードが実行されるまでトレースを行ない、止まる。COUNT が指定された場合にはその回数実行されるまでトレースを行ない、止まる。

2.3.4　操作例

この例では、前述のプログラムをコンパイル、ロードした状態からのコードのトレース例を示す。また、トレーサの出力の詳しい見方は、6.2「見方」を参照。

```
| ?- load("sample").   《sample.sav をコード領域にロード》
"sample" : Loading ... done
Module name : sample

yes.
| ?- trace(sample).   《モジュール sample のコードのトレースモードを ON に》
Setting ... done

yes.
[debug]?- sample:fib(10,L)|L.   《トレースの実行》
>>>>> Priority: 3991 <<<<<
Call  : [0004]sample:fib(10,A).  [step]%
Call  : [0004]sample:fibi(10,1,1,A).  [step]%
Call  : [0004]sample:fibi(10,1,2,A).  [step]%
```

```
 Call  : [0004]sample:fib1(10,2,3,A).  [step]%
 Call  : [0004]sample:fib1(10,3,5,A).  [step]%
 Call  : [0004]sample:fib1(10,5,8,A).  [step]%
 Call  : [0004]sample:fib1(10,8,13,A).  [step]%
 L = [1,1,2,3,5,8]

 yes.
 [debug]?- sample:fib(10,L)|L.   《トレースの実行》
 >>>>> Priority: 3991 <<<<<
 Call  : [0004]sample:fib(10,A).  [step]% s 3   《3 ステップのトレース》
 Call  : [0004]sample:fib1(10,1,1,A).  [step]%
 Call  : [0004]sample:fib1(10,1,2,A).  [step]%
 Call  : [0004]sample:fib1(10,2,3,A).  [step]%
 Call  : [0004]sample:fib1(10,3,5,A).  [step]%
 Call  : [0004]sample:fib1(10,5,8,A).  [step]%
 Call  : [0004]sample:fib1(10,8,13,A).  [step]%
 L = [1,1,2,3,5,8]

 yes.
 [debug]?- spy(sample,fib1,4).   《述語 fig1/4 に対するスパイの設定》
 Setting ... done

 yes.
 [debug]?- sample:fib(10,L)|L.   《スパイの実行》
 >>>>> Priority: 3991 <<<<<
 Call  : [0015]sample:fib(10,A).  [step]% sp  《次のスパイされている述語コードまでトレース》
 Call* : [0015]sample:fib1(10,1,1,A).  [sp]%
 Call* : [0015]sample:fib1(10,1,2,A).  [sp]%
 Call* : [0015]sample:fib1(10,2,3,A).  [sp]%
 Call* : [0015]sample:fib1(10,3,5,A).  [sp]%
 Call* : [0015]sample:fib1(10,5,8,A).  [sp]%
 Call* : [0015]sample:fib1(10,8,13,A).  [sp]%
 L = [1,1,2,3,5,8]

 yes.
 [debug]?- spying(sample).   《モジュール sample 内でスパイされている述語一覧》
 fib1/4, END.

 yes.
 [debug]?- notrace(sample).   《トレースモード OFF》
 Resetting ... done

 yes.
 [debug]?- sample:fib(10,L)|L.   《トレースなしでの実行》
 L = [1,1,2,3,5,8]
```

```
yes.
[debug]?- nodebug.   《デバッグモード OFF》

yes.
| ?- halt.
```

3　KL1 の言語仕様

本章では PDSS で実行可能な KL1 の言語仕様について述べる。尚、本章で述べる KL1 の言語仕様はあくまでも PDSS 上におけるものであり、他のシステム, 例えば Multi-PSI V2 上のものとは多少異なる場合がある。

3.1　概要

KL1 は GHC (Guarded Horn Clauses) を基に設計された言語であり、OS 記述用の機能やモジュール化機能など幾つかの言語機能の拡張と、実現のしやすさという観点からの制限を加えた言語である。KL1 の主な特徴を以下に示す。

ガードの逐次性

ヘッド・ユニフィケション及びガード部に書かれたゴールの実行は基本的にテキストに書かれた順序に従って左から右に逐次実行される。すなわち、次のようなプログラムは変数 X が具体化されない限り中断状態のままである。

```
Goal:     ?- p(X,b).
Clause:   p(a,c) :- true | true.
```

また、構造体の内部のチェックも左から右に逐次実行される。すなわち、次のプログラムは中断する。

```
Goal:     ?- p({X,b},{a,c}).
Clause:   p(X,X) :- true | true.
```

ガード部の制限

ガード部に記述できる述語を一部の組込述語に限定している。ガード部で記述可能な組込述語については 3.6 章で述べる。

変数の同一性

ガード部における変数の同一性のチェックは行っていない。すなわち、次の様なプログラムは変数 X と Y が具体化されないかぎり、ゴールの実行順に因らず中断状態のままである。

```
Goal:     ?- X=Y, p(X,Y).
Clause:   p(A,A) :- true | true.
```

モジュール化機能

幾つかのクローズの集りをモジュールとして宣言することにより、モジュール単位のコンパイル, デバッグが可能。現在は 1 つのファイルが 1 つのモジュールを定義するようになっている。

荘園

荘園と呼ぶ機能単位を導入し、この荘園ごとに実行優先順位の制御や実行量の制御が行える。OS はこの荘園機能を中心に構築される。

例外処理機能

言語が規定する種々の例外に対する処理を荘園機能と高階呼出し機能を用いて KL1 自身で記述することができる。

失敗の扱い

KL1 では失敗は全て例外として扱われ、例外処理機能を用いて実行を続行することも可能である。

3.2　荘園

荘園とは言語が規定している資源管理, プライオリティ管理及び例外処理の最小単位である。荘園にはコントロール・ストリームとレポート・ストリームと呼ぶ2本のストリームが接続されている。コントロール・ストリームは荘園を制御するためのストリームであり、後述する種々のコマンドを流すことができる。レポート・ストリームは例外情報など荘園内からの情報 / 要請が流れ出てくるストリームである。荘園のユーザはこのレポート・ストリームの各種情報を解釈するプログラムを記述することによって例外事項等を適当に操作できる。

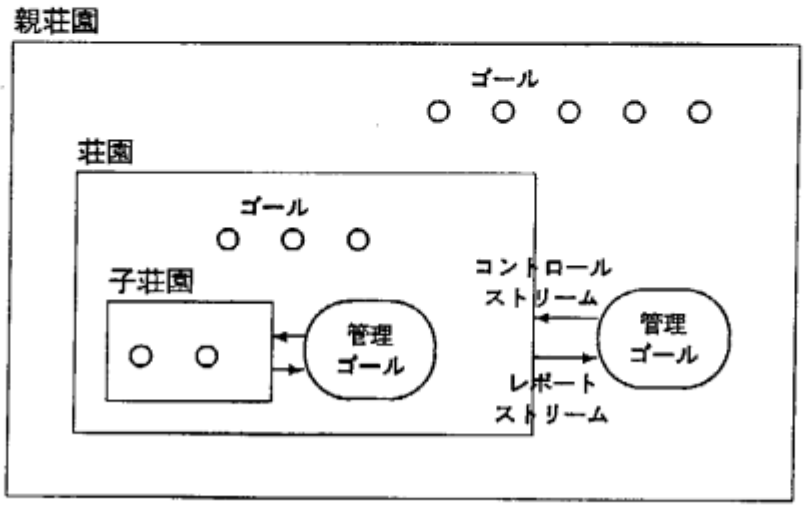

図 2: 荘園のイメージ

資源管理機能

PDSS で管理している「資源」とはゴールのリダクションの数である。これは実行時間やメモリ消費量の非常におおまかな近似と考えれば良い。全てのゴールは必ず何れかの荘園に属しており、各ゴールのリダクションはそのゴールが属している荘園の下で管理される。荘園にはその荘園内で実行できるリダクションの上限値を与えることができる。また、上限値とは別に (上限値の範囲内で) 荘園に対する割り当て量がシステムによって決められており、荘園の生成時 (正確には荘園を start させた時) にはこの割り当て量が上限値を超えない範囲で自動的に与えられる。また、実行中に不足した場合には、その荘園の親荘園が持つ割り当て量 (の残り) からシステムで規定している分割量だけ分割し与えられる (この場合ももちろん荘園の上限値を超えない範囲で行われる)。もしどうしても上限値を超えてしまうような場合は「資源の不足例外」としてその荘園のレポート・ストリームに例外情報が流される。上限値を増やす為には後で述べるようにコントロール・ストリームに add_resource(R) なるコマンドを流せばよい。

プライオリティ管理機能

荘園のもう一つの機能としてゴールのプライオリティ管理がある。各荘園レコードにはその荘園内で実行可能なゴールのプライオリティの上限と下限がセットされており、これを超えるプライオリティでゴールを実行することはできない。ゴールに対するプライオリティの指定方法等については 3.3 章で述べる。

3.2.1　荘園の生成

荘園の生成にはシステムで用意している 'Sho-en' モジュールを使う。荘園モジュールには荘園生成の述語 execute/7 が定義されている (旧仕様の execute/8 も残されている)。

```
execute(コード, 引数, 荘園内最低プライオリティ,
        荘園内最高プライオリティ, タグ, コントロール, レポート)
execute(モジュール名, 述語名, 引数, 荘園内最低プライオリティ,
```

荘園内最高プライオリティ，タグ，コントロール，レポート)

ここで、「コード」とは、3要素のベクタ {モジュール名アトム,述語名アトム,引数の数} である。(以降、「コード」は全てこの形式で表現される。Multi-PSI V2 では「コード」はコード型データを使用する。)「引数」とはゴールの引数を要素とするベクタである。

「荘園内最低プライオリティ」は荘園内で実行されるゴールが使用できるプライオリティの下限値を計算するための値で、下限をどのくらい上げるかを指定する 0 以上 4096 以下の整数であり、0 だと親荘園の下限と同じ値に、4096 だと execute を呼び出すゴールと同じ値になる。「荘園内最高プライオリティ」は荘園内で実行されるゴールが使用できるプライオリティの上限値を計算するための値で、上限をどのくらい下げるかを指定する 0 以上 4096 以下の整数であり、0 だと execute を呼び出すゴールと同じ値に、4096 だと親荘園の下限と同じ値になる。以上を纏めると 図 3 のようになる。

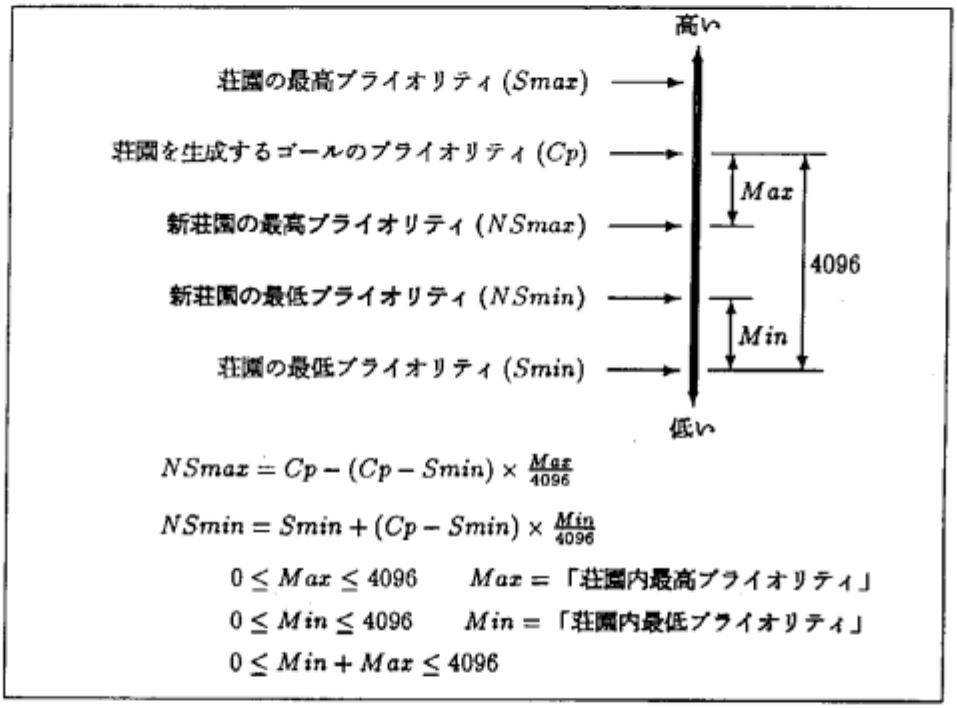

図 3: 荘園のプライオリティの計算

「タグ」は荘園内のどのような例外を受信するかを示すビット・マスクを与える。各ビットの意味は 付録-8 を参照のこと。「コントロール」にはコントロール・ストリームが、「レポート」にはレポート・ストリームがユニファイされる。なお、生成された荘園の初期状態は中断状態にあり、許容リダクション数はセットされていない。

【例】 'Sho-en':execute({primes,do,3},{1,300,PN},0,2,16#"FFFFFFFF",CTR,REP)

3.2.2 コントロール・ストリーム

コントロール・ストリームに流すことができるコマンドには以下のものがある。コントロール・ストリームを閉じると、荘園は放棄される。ストリームを閉じなければ荘園そのものの実行は終了にならないので注意。

start

荘園内のゴールの実行を再開可能にする。

stop

荘園内のゴールの実行を一時中断する。start コマンドによって再開可能状態にすることができる。

abort

荘園内の実行を放棄する。このコマンドを流した後は start コマンドを使っても実行は再開出来ない。

add_resource(Reduction)

荘園の現在の許容リダクションの上限値に Reduction で示される値を追加する。

allow_resource_report

「資源の不足例外」に対しての回答で、以後の不足例外の報告を許す。「資源の不足例外」を一回報告すると allow_resource_report コマンドで許可されるまで「資源の不足例外」の報告は止められる。

statistics

荘園の統計情報を問い合わせる。情報はレポート・ストリームに流される。

3.2.3　レポート・ストリーム

レポート・ストリームから流れ出る情報には以下のものがある。

コントロール・ストリームへのメッセージに対する回答

これはコントロール・ストリームへ流されたメッセージに対する回答で、一対一に対応している。

started

start メッセージを受信した。

stopped

stop メッセージを受信した。

aborted

aborted メッセージを受信した。

resource_added

add_resource メッセージを受信した。

resource_report_allowed

allow_resource_report メッセージを受信した。これ以降、「資源の不足例外」が発生すれば、それが報告される。

statistics_started

statistics メッセージを受信した。統計情報は収集が完了した時点で別のメッセージにより報告される。

状態情報

これは荘園の状態が変わったことを報告する為のものである。

terminated

荘園の実行が終了した。終了の理由は、先に aborted が送られていれば実行放棄、送られてなければ全てのゴールの成功を意味する。

resource_low

最大許容リダクション数を超えそうになった。もしくは割り付け量が足りなかった。荘園はこの例外が発生すると中断状態になる。この報告をした後はコントロール・ストリームに allow_resource_report を流すまで、次の resource_low は報告されない。

統計情報

これは統計情報の収集が完了した時に、その結果を報告する為のものである。

statistics(Info)

荘園内の統計情報を Info にユニファイする。Info は 1 要素のベクタで、要素にリダクション数が格納されている。リダクション数には子孫の荘園内のリダクション数も含まれている。

例外情報

これは荘園の中で例外事象が発生したことを報告するものである。例外情報は、デッドロックを除いて例外に対する処理を指定することができる。処理系では例外を発見すると、それが発生した荘園内に例外処理用のゴール apply(NewCode, NewArgv) を生成し、NewCode, NewArgv に代わりに実行すべきゴールがユニファイされるのを待つようにする。なお、NewCode に指定する述語は Public 宣言されたものでなければならない。また、NewCode = [] になった場合は代わりのゴールは実行されない。

exception(ExcpCode, Info, NewCode, NewArgv)

荘園内で例外が発生した。ExcpCode は例外の種類を表わす正の整数、Info は例外情報で、例外の種類により異なる。例外コードについては付録-7を参照。NewCode, NewArgv には例外を起こしたゴールの代わりに実行して欲しいゴールのコードと引数をユニファイする。ExcpCode, Info について以下に記す。なお、以下で Caller は組込述語を呼んだ述語のコード、OpCode は組込述語のオペコード、Argv は引数ベクタ、Code は { モジュール名, 述語名, 引数の数 } を表わす。

ExcpCode	意味	:: Info	説明
0	Illegal Input Type	:: {0, Caller, OpCode, Pos, Argv}	Pos は不正引数位置 (1 ～ 7)
1	Range Overflow	:: {0, Caller, OpCode, Argv}	
3	Integer Overflow	:: {0, Caller, OpCode, Argv}	
5	Floating Point Error	:: {0, Caller, OpCode, Argv}	
8	Illegal Merger Input	:: {0, Caller, OpCode, MI, FMI}	MI はマージャへの不正入力データ
		::	FMI はマージャへの入力ストリーム
9	Reduction Failure	:: {0, Code, Argv}	
10	Unification Failure	:: {0, X, Y}	X,Y はボディユニフィケーションに
		::	失敗した項
12	Raised	:: {0, RType, RInfo}	RType,RInfo は組込述語
		::	raise/3 によって渡されたターム
16	Incorrect Priority	:: {0, Caller, OpCode, Argv}	
17	Module Not Found	:: {0, Code, Argv}	
18	Predicate Not Found	:: {0, Code, Argv}	

deadlock(ExcpCode, Info)

荘園内でデッドロック状態が発見された。ExcpCode は例外がデッドロックであることをを表わす正の整数、Info は例外情報で、この場合 {0, DGoal, DType, GoalsList} の形式となる。DGoal はデッドロックが発生するきっかけとなった述語のコード又は [](一括型 GC で発見された時)、DType はデッドロックの種類を表わす整数 (7章参照)、GoalsList はデッドロックしているゴールのルートとなっているコードからなるリスト。

ExcpCode	意味	:: Info	説明
11	Deadlock	:: {0, DGoal, DType, GoalsList}	上記参照

3.3 プライオリティ

KL1 にはゴールの実行優先順位 (以下、プライオリティと言う) をゴールごとに指定する機能がある。プライオリティには論理プライオリティと物理プライオリティがあり、ゴールはそれぞれ自身の論理プライオリティを持つ。処理系内にはあらかじめ指定されたレベルの物理プライオリティがあり、スケジューラはゴールをゴール・スタックに繋げる際論理プライオリティを物理プライオリティに変換する (物理プライオリティは論理プライオリティよりも精度が悪いので、ユーザはプライオリティを変えたからといって必ず実際のスケジュールがそのとおりになることを期待してはいけない)。荘園が持つプライオリティの上限 / 下限も論理プライオリティである。

ユーザはゴールのプライオリティを指定する場合その親ゴールの論理プライオリティ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。

所属荘園内割合指定

この指定方法は指定されたゴールが所属している荘園の論理プライオリティの上限 / 下限に対する相対値で指定する方法であり、次のような形式で書く。

Goal @ priority(*, 割合)

このときゴール Goal の論理プライオリティは、図 4のように決められる。

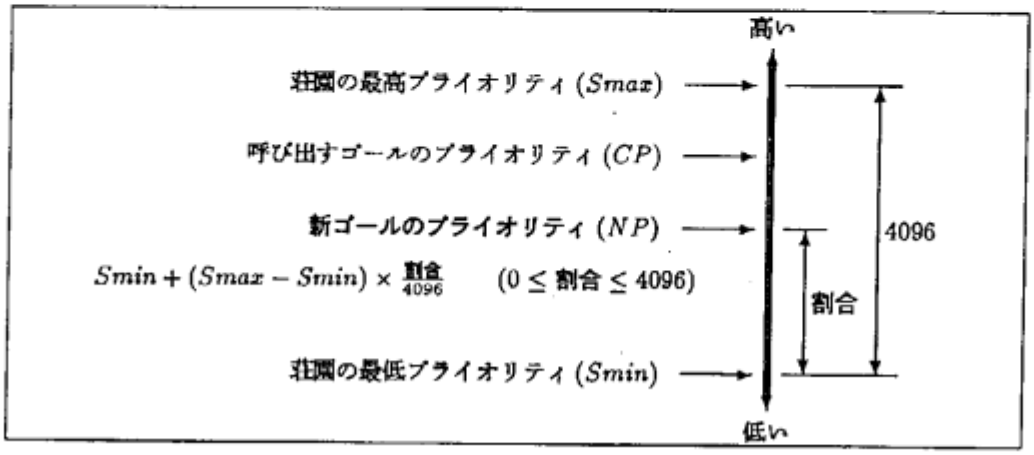

図 4: 所属荘園内割合指定によるプライオリティの計算

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プライオリティ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プライオリティの上限 / 下限を越えることはできない。次のような形式で書く。

Goal @ priority($, 割合)

このときゴール Goal の論理プライオリティは、割合 の正負により、正ならば、図 5 のように、負ならば、図 6のように決められる。

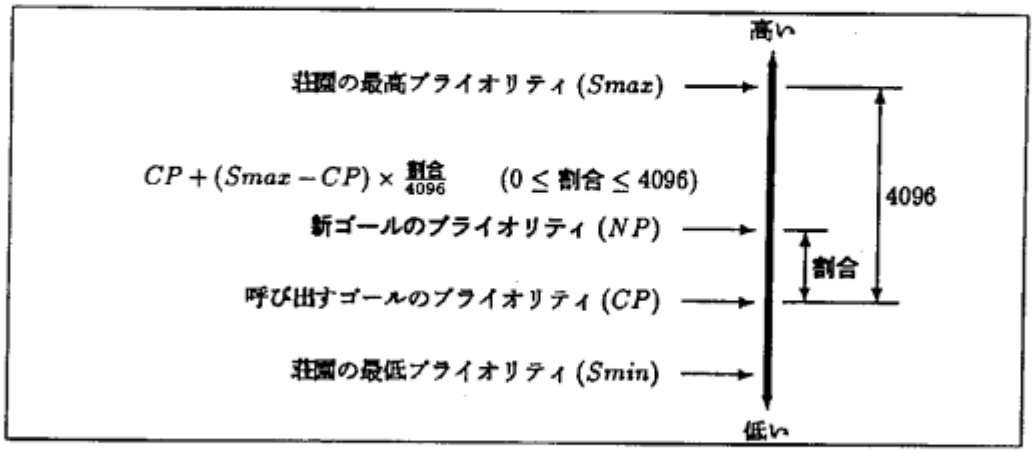

図 5: 所属荘園内自己相対指定によるプライオリティの計算 (正)

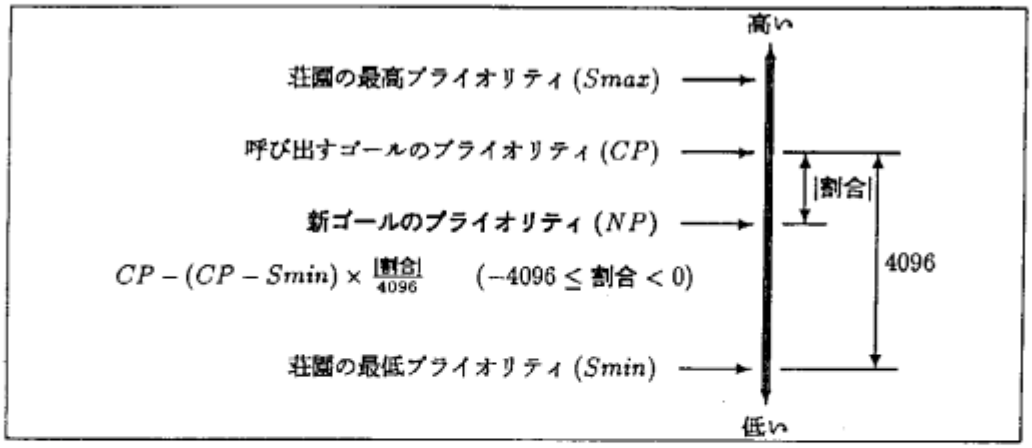

図 6: 所属荘園内自己相対指定によるプライオリティの計算 (負)

3.4　シンタックス

基本的な節のシンタックスは別の稿に譲ることにして、ここでは主に GHC との相違点を中心に述べる。GHC との主な相違点は、

- モジュールの定義
- 節の順序付け
- プライオリティ指定
- マクロ記法

が挙げられる。このうちプライオリティ指定とマクロ記法は別の章で述べてある。

3.4.1　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.
```

で始まらなければならない。またそのモジュール内の述語で他のモジュールに公開する述語は

```
:- public 述語名/引数, 述語名/引数, ... .
```

と宣言する必要がある。組込述語 apply で実行する述語や、荘園の生成の際に指定する述語は全て public 宣言をする必要があるので注意。また、public 宣言はファイルの先頭部 (述語定義を記述する前) に、1 個にまとめて行う必要がある。これは Prolog の場合と違うので注意が必要である。

述語の定義は分割してはならない。すなわち、2 つ以上の述語定義が交互に現われるような場合、同じ述語名で同じ引数であっても別の述語定義として扱われてしまう。これはアセンブル時に "Assembler: Doubly defined label." のエラーになる。

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を付けて次のように明示的に記述する。

```
モジュール名 : ゴール
```

逆に、"モジュール名 : " の付かないゴールは全て、モジュール内呼び出しになる。そして、述語名の空間はモジュールごとに区別されることになる。これにより、モジュールが違えば、同じ名前を別の述語定義に使うことができる。

3.4.2　節の順序付け

KL1 ではコンパイラによるインデキシング等の為、節の実行順序は必ずしもテキスト上の順序と一致しない場合がある。このため、節の間で実行の優先順位を付けたい場合の記述や逐次実行を記述するための記法が用意されてい

る。

節の優先実行

述語の定義中 (節と節の間) に alternatively. と書く。これにより alternatively. の前後の節 (群) の順序関係は保証される。

```
foo([X|XX],Z) :- true | p(X,XX,Z).
        ...
alternatively.
foo(X,[Z|ZZ]) :- true | q(X,Z,ZZ).
        ...
```

逐次実行

述語の定義中 (節と節の間) に otherwise. と書く。これにより otherwise. 以前に書かれた節の全てが失敗して初めて otherwise. 以降の節 (群) が実行される。

```
foo([X|XX]) :- X=a | pa(X,XX).
foo([X|XX]) :- X=b | pb(X,XX).
        ...
otherwise.
foo(X) :- true | q(X).
        ...
```

3.5　データ型

PDSS でサポートしているデータ型には以下のものがある。なお、ここで述べるデータ型は一般ユーザーが KL1 のプログラムで扱える / 扱って意味があるものだけである。

- 変数 ··· `A, A12, B, _abc, _`
  Prolog と同様に、英大文字またはアンダースコアで始まる英数字列で表現する。同一の節の中では、同一の名前のものが同一の変数となる。ただし、アンダースコア 1 文字のものは、全て別々の変数として扱われる。
- アトム ··· `abc, 'ABC', :=, 'can''t'`
  Prolog と同様に、英小文字で始まる英数字列、記号文字だけから成る文字列、引用符 ' で括られた文字列で表現する。なお、引用符 ' 自身をアトムの名前の一部として使う場合には、引用符 ' で括ったなかに、引用符 ' を 2 個書く事で表現する。また、'$' はシステムにより特別な用途に使われるので、一般ユーザーはアトム名 (の一部) として使用しないほうが良い。
- 整数 ··· `123, 16'ACE, 8'37, +3, -5`
  通常は 10 進数で表現し、−2147483648 以上、2147483647 以下の値を持つ。このとき、符号の +/− は整数の一部として扱われる。(符号の後に空白があると整数の一部とは成らない。) また、2 進数から 36 進数で記述する事もできる。これは引用符 ' を使って基数 'N 進数 の形式で記述するもので、この場合には符号を付ける事はできない。この形式は PDSS のリーダーが解釈しているので、Prolog で書かれたコンパイラでは Syntax Error となる場合がある。ソースプログラム上ではマクロ表記 基数#"N 進数 " で記述したほうが良い。
- 浮動小数点数 ··· `1.23, 1.0e10, 3.0E-30, -2.0`
  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。PDSS では単精度 (32bits) を採用して

おり、これは、$-3.402823 \times 10^{38} \sim 3.402823 \times 10^{38}$ の値を 10 進数約 7 桁の精度で表現する。Multi-PSI V2 と違い、Infinity を使う事はできない。

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、従って、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。

- リスト … [1,2,3], [X|Y]

  リストは、[ ] により記述される。Car と Cdr を記述する場合は '|' を用い、[car|cdr] と記述する。
- ベクタ … {1,2,{3,4}}, f(X), {}

  これは、要素ゼロのものを含む一次元配列の構造体で、{ } による記述とファンクタによる記述ができる。そして、例えば f(a) と {f,a} は同じ構造を意味する。
- ストリング … "abc", "", """"

  引用符 " で括られた文字列で表現する。なお、引用符 " 自身をストリングの一部として使う場合には、引用符 " で括ったなかに、引用符 " を 2 個書く事で表現する。ストリングの要素サイズには 1 から 32 ビットまでのものが許されている。PDSS では引用符 "..." で示されたストリングは要素サイズが 8 ビットのストリングを意味する。尚、Prolog で書かれたコンパイラではストリングとリストを正確に区別することができない (Prolog ではストリングは文字コードのリストになる。) ので、マクロ表記 string#"..." で記述したほうが良い。また、8 ビット以外のストリングは内部表現としてだけ使う事ができ、プログラム上に定数として記述する事はできない。

  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文字コードである場合に成功する。

3.6　組込述語

ここでは PDSS で使用可能な組込述語について述べる。各組込述語は次の形式で示されている。

$$\underbrace{\text{vector(X, \^{}Size)}}_{\text{呼び出し形式}} \quad :: \quad \underbrace{\text{G}}_{\text{記述可能な場所}}$$

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ものがある。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。^ 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) であることを意味し、その組込述語の記述されている場所により、ガード部の場合は Passive Unification、ボディ部であれば Active Unification が行われる。なお、「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。

また算術演算については計算式を記述するマクロが用意されているので、組込述語を直接書かなくても良いようになっている。マクロについては 3.7 章を参照のこと。

3.6.1　タイプのチェック

wait(X) :: G

X が未定義ならば中断。それ以外は成功。

atom(X) :: G

X が未定義ならば中断。アトムならば成功。それ以外は失敗。

integer(X) :: G

X が未定義ならば中断。整数ならば成功。それ以外は失敗。

floating_point(X) :: G

X が未定義ならば中断。浮動小数点数ならば成功。それ以外は失敗。

list(X) :: G

X が未定義ならば中断。リストであれば成功。それ以外は失敗。

vector(X) :: G

X が未定義ならば中断。ベクタであれば成功。それ以外は失敗。

string(X) :: G

X が未定義ならば中断。ストリングであれば成功。それ以外は失敗。

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 {PE, Addr, X} なるベクタをユニファイし成功する。ここで、PE は変数のある PE の番号 (整数、PDSS では常に 0)、Addr は変数のアドレス (整数) である。すでに具体化されていれば、Result に {X} をユニファイし成功。

【注意】この組込述語により得られる PE, Addr の値は GC 等により変化する。

3.6.2　diff

diff(X, Y) :: G

X と Y がユニファイアブルでないことが確定すれば成功。まったく同じ値 / 構造であることが確定すれば失敗。それ以外は中断。以下のマクロ表記を用いることができる。

```
X \= Y  <=>  diff(X,Y).
```

【注意】比較は左から深さ優先に行われる。この途中で未定義変数が見つかった場合にはその先を調べずに、すぐサスペンドする。また、X, Y が構造体同士の場合、比較は一定の深さで打ち切られ、その深さまででユニファイアブルならば失敗として扱われる。

3.6.3　整数の比較

equal(Integer1, Integer2) :: G

Integer1 または Integer2 が未定義ならば中断。双方が整数でありかつ Integer1 と Integer2 の値が等しいとき成功。それ以外では失敗。以下のマクロ表記を用いることができる。

```
X =:= Y  <=>  equal(X,Y).
```

not_equal(Integer1, Integer2) :: G

Integer1 または Integer2 が未定義ならば中断。双方が整数でありかつ Integer1 と Integer2 の値が等しくないとき成功。それ以外では失敗。以下のマクロ表記を用いることができる。

```
X =\= Y  <=>  not_equal(X,Y).
```

less_than(Integer1, Integer2) :: G

Integer1 または Integer2 が未定義ならば中断。双方が整数でありかつ Integer1 の値が Integer2 の値より小さいとき成功。それ以外では失敗。以下のマクロ表記を用いることができる。

```
X < Y  <=>  less_than(X,Y).
X > Y  <=>  less_than(Y,X).
```

not_less_than(Integer1, Integer2) :: G

Integer1 または Integer2 が未定義ならば中断。双方が整数でありかつ Integer1 の値が Integer2 の値より小さくないとき成功。それ以外では失敗。以下のマクロ表記を用いることができる。

```
X >= Y  <=>  not_less_than(X,Y).
X =< Y  <=>  not_less_than(Y,X).
```

3.6.4　整数の演算

add(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。Integer1 に Integer2 を加えた値を NewInteger にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z := X + Y  <=>  add(X,Y,Z).
```

subtract(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。Integer1 から Integer2 を引いた値を NewInteger にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z := X - Y  <=>  subtract(X,Y,Z).
```

multiply(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。Integer1 に Integer2 を掛けた値を NewInteger にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z := X * Y  <=>  multiply(X,Y,Z).
```

divide(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。Integer2 が 0 ならば失敗 / 例外。Integer1 を Integer2 で割った商を NewInteger にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z := X / Y  <=>  divide(X,Y,Z).
```

modulo(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。Integer2 が 0 ならば失敗 / 例外。Integer1 を Integer2 で割った余りを NewInteger にユニファイする。以下のマクロ表記を用いることができる。

```
Z := X mod Y  <=>  modulo(X,Y,Z).
```

minus(Integer, ^NewInteger) :: GB

Integer が未定義ならば中断。整数でなければ失敗 / 例外。Integer を符号反転した値を NewInteger にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y := -X  <=>  minus(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

increment(Integer, ^NewInteger) :: GB

Integer が未定義ならば中断。整数でなければ失敗 / 例外。Integer に 1 加えた値を NewInteger にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y := X+1  <=>  increment(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

decrement(Integer, ^NewInteger) :: GB

Integer が未定義ならば中断。整数でなければ失敗 / 例外。Integer から 1 引いた値を NewInteger にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y := X-1  <=>  decrement(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

abs(Integer, ^NewInteger) :: GB

Integer が未定義ならば中断。整数でなければ失敗 / 例外。Integer の絶対値を求め NewInteger にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y := abs(X)  <=>  abs(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

min(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。Integer1 と Integer2 の小さい方の値を NewInteger にユニファイする。以下のマクロ表記を用いることができる。

```
Z := min(X, Y)  <=>  min(X,Y,Z).
```

【注意】Multi-PSI V2 ではサポートされていない。

max(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。Integer1 と Integer2 の大きい方の値を NewInteger にユニファイする。以下のマクロ表記を用いることができる。

```
Z := max(X, Y)  <=>  max(X,Y,Z).
```

【注意】Multi-PSI V2 ではサポートされていない。

and(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。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。以下のマクロ表記を用いることができる。

```
Z := X /\ Y  <=>  and(X,Y,Z).
```

or(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。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。以下のマクロ表記を用いることができる。

```
Z := X \/ Y  <=>  or(X,Y,Z).
```

exclusive_or(Integer1, Integer2, ^NewInteger) :: GB

Integer1 または Integer2 が未定義ならば中断。整数でなければ失敗 / 例外。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。以下のマクロ表記を用いることができる。

```
Z := X xor Y  <=>  exclusive_or(X,Y,Z).
```

complement(Integer, ^NewInteger) :: GB

Integer が未定義ならば中断。整数でなければ失敗 / 例外。Integer の各ビットを反転した値を NewInteger にユニファイする。以下のマクロ表記を用いることができる。

```
Y := \(X)  <=>  complement(X,Y).
```

shift_left(Integer, ShiftWidth, ^NewInteger) :: GB

Integer が未定義変数ならば中断。整数でなければ失敗 / 例外。ShiftWidth が未定義変数ならば中断。0 以上 31 以下の整数でなければ失敗 / 例外。Integer を ShiftWidth ビットだけ左に論理シフトし結果を NewInteger とユニファイする。以下のマクロ表記を用いることができる。

```
Z := X << Y  <=>  shift_left(X,Y,Z).
```

shift_right(Integer, SiihftWidth, ^NewInteger) :: GB

Integer が未定義変数ならば中断。整数でなければ失敗 / 例外。ShiftWidth が未定義変数ならば中断。0 以上 31 以下の整数でなければ失敗 / 例外。Integer を ShiftWidth ビットだけ右に論理シフトし結果を NewInteger とユニファイする。以下のマクロ表記を用いることができる。

```
Z := X >> Y  <=>  shift_right(X,Y,Z).
```

3.6.5　浮動小数点数の比較

floating_point_equal(Float1, Float2) :: G

Float1 または Float2 が未定義ならば中断。双方が浮動小数点数でありかつ Float1 と Float2 の値が等しいとき成功。それ以外では失敗。以下のマクロ表記を用いることができる。

```
X $=:= Y  <=>  floating_point_equal(X,Y).
```

floating_point_not_equal(Float1, Float2) :: G

Float1 または Float2 が未定義ならば中断。双方が浮動小数点数でありかつ Float1 と Float2 の値が等しくないとき成功。それ以外では失敗。以下のマクロ表記を用いることができる。

```
X $=\= Y  <=>  floating_point_not_equal(X,Y).
```

floating_point_less_than(Float1, Float2) :: G

Float1 または Float2 が未定義ならば中断。双方が浮動小数点数でありかつ Float1 の値が Float2 の値より小さいとき成功。それ以外では失敗。以下のマクロ表記を用いることができる。

```
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 または Float2 が未定義ならば中断。双方が浮動小数点数でありかつ Float1 の値が Float2 の値より小さくないとき成功。それ以外では失敗。以下のマクロ表記を用いることができる。

```
X $>= Y  <=>  floating_point_not_less_than(X,Y).
X $=< Y  <=>  floating_point_not_less_than(Y,X).
```

3.6.6　浮動小数点数の演算

floating_point_add(Float1, Float2, ^NewFloat) :: GB

Float1 または Float2 が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float1 に Float2 を加えた値を NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z $:= X + Y  <=>  floating_point_add(X,Y,Z).
```

【注意】Multi-PSI V2 ではオーバーフローは失敗 / 例外にはならず、NewFloat に infinity を出力する。

floating_point_subtract(Float1, Float2, ^NewFloat) :: GB

Float1 または Float2 が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float1 から Float2 を引いた値を NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z $:= X - Y  <=>  floating_point_subtract(X,Y,Z).
```

【注意】Multi-PSI V2 ではオーバーフローは失敗 / 例外にはならず、NewFloat に infinity を出力する。

floating_point_multiply(Float1, Float2, ^NewFloat) :: GB

Float1 または Float2 が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float1 に Float2 を掛けた値を NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z $:= X * Y  <=>  floating_point_multiply(X,Y,Z).
```

【注意】Multi-PSI V2 ではオーバーフローは失敗 / 例外にはならず、NewFloat に infinity を出力する。

floating_point_divide(Float1, Float2, ^NewFloat) :: GB

Float1 または Float2 が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float2 が 0.0 ならば失敗 / 例外。Float1 を Float2 で割った商を NewFloat にユニファイする。このとき、オーバーフローになると失

敗 / 例外。以下のマクロ表記を用いることができる。

```
Z $:= X / Y  <=>  floating_point_divide(X,Y,Z).
```

【注意】Multi-PSI V2 ではオーバーフローは失敗 / 例外にはならず、NewFloat に infinity を出力する。

floating_point_minus(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float を符号反転した値を NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= -X  <=>  floating_point_minus(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_abs(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float の絶対値を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= abs(X)  <=>  floating_point_abs(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_min(Float1, Float2, ^NewFloat) :: GB

Float1 または Float2 が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float1 と Float2 の小さい方の値を NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Z $:= min(X, Y)  <=>  floating_point_min(X,Y,Z).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_max(Float1, Float2, ^NewFloat) :: GB

Float1 または Float2 が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float1 と Float2 の大きい方の値を NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Z $:= max(X, Y)  <=>  floating_point_max(X,Y,Z).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_floor(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float より大きくない、最大の整数値を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= floor(X)  <=>  floating_point_floor(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_sqrt(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float が負ならば失敗 / 例外。Float の平方根を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= sqrt(X)  <=>  floating_point_sqrt(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_ln(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float が 0.0 以下ならば失敗 / 例外。Float の自然対数を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= ln(X)  <=>  floating_point_ln(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_log(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float が 0.0 以下ならば失敗 / 例外。Float の常用対数を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= log(X)  <=>  floating_point_log(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_exp(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。e の Float 乗を求め NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y $:= exp(X)  <=>  floating_point_exp(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_pow(Float1, Float2, ^NewFloat) :: GB

Float1 または Float2 が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float1 が負で Float2 が整数値でない場合は失敗 / 例外。Float1 の Float2 乗を求め NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z $:= X ** Y  <=>  floating_point_pow(X,Y,Z).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_sin(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float の正弦の値を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= sin(X)  <=>  floating_point_sin(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_cos(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float の余弦の値を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= cos(X)  <=>  floating_point_cos(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_tan(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float の正接の値を求め NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y $:= tan(X)  <=>  floating_point_tan(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_asin(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float が -1.0 以上、1.0 以下でなければ失敗 / 例外。Float の逆正弦の値を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= asin(X)  <=>  floating_point_asin(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_acos(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float が -1.0 以上、1.0 以下でなければ失敗 / 例外。Float の逆余弦の値を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= acos(X)  <=>  floating_point_acos(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_atan(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float の逆正接の値を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= atan(X)  <=>  floating_point_atan(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_atan(Float1, Float2, ^NewFloat) :: GB

Float1 または Float2 が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float2 が 0.0 ならば失敗 / 例外。Float1/Float2 の逆正接の値を求め NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Z $:= atan(X/Y)  <=>  floating_point_atan(X,Y,Z).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_sinh(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float の双曲線正弦の値を求め NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y $:= sinh(X)  <=>  floating_point_sinh(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_cosh(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float の双曲線余弦の値を求め NewFloat にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y $:= cosh(X)  <=>  floating_point_cosh(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

floating_point_tanh(Float, ^NewFloat) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float の双曲線正接の値を求め NewFloat にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= tanh(X)  <=>  floating_point_tanh(X,Y).
```

【注意】Multi-PSI V2 ではサポートされていない。

3.6.7　整数 - 浮動小数点数の変換

floating_point_to_integer(Float, ^Integer) :: GB

Float が未定義ならば中断。浮動小数点数でなければ失敗 / 例外。Float を整数に変換 (小数点以下切り捨て) し Integer にユニファイする。このとき、オーバーフローになると失敗 / 例外。以下のマクロ表記を用いることができる。

```
Y := int(X)  <=>  floating_point_to_integer(X,Y).
```

integer_to_floating_point(Integer, ^Float) :: GB

Integer が未定義ならば中断。整数でなければ失敗 / 例外。Integer を浮動小数点数に変換し Float にユニファイする。以下のマクロ表記を用いることができる。

```
Y $:= float(X)  <=>  integer_to_floating_point(X,Y).
```

3.6.8　ベクタ関係

vector(X, ^Size) :: G

X が未定義ならば中断。ベクタであればそのサイズを Size とユニファイする。それ以外は失敗。

vector(X, ^Size, ^NewVector) :: B

X が未定義ならば中断。ベクタであればそのサイズを Size とユニファイし X を NewVector にユニファイする。それ以外は例外。

new_vector(^Vector, Size) :: B

Size が未定義ならば中断。Size が非負整数でなければ例外。Size の値により生成されるベクタの大きさがヒープ領域の大きさより大きくなる場合も例外。これ以外の場合には要素数 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。

vector_element(Vector, Position, ^Element) :: G

Vector が未定義ならば中断。ベクタ以外の値であれば失敗。Position が未定義ならば中断。非負整数でないか Vector の要素数以上の場合は失敗。これ以外の場合には Vector の Position 番目の要素を Element とユニファイする。

vector_element(Vector, Position, ^Element, ^NewVector) :: B

Vector が未定義ならば中断。ベクタ以外の値であれば例外。Position が未定義ならば中断。非負整数でないか Vector の要素数以上の場合は例外。これ以外の場合には 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。

set_vector_element(Vector, Position, ^OldElem, NewElem, ^NewVect) :: B

Vector が未定義ならば中断。ベクタ以外の値であれば例外。Position が未定義ならば中断。非負整数でないか Vector の要素数以上の場合は例外。これ以外の場合には 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。

3.6.9　ストリング関係

string(X, ^Size, ^ElementSize) :: G

X が未定義ならば中断。ストリングであればその要素数を Size とユニファイし要素サイズ (要素のビット長) を ElementSize とユニファイする。それ以外は失敗。

string(X, ^Size, ^ElementSize, ^NewString) :: B

X が未定義ならば中断。ストリングであればその要素数を Size とユニファイし要素サイズ (要素のビット長) 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。それ以外は例外。

new_string(^String, Size, ElementSize) :: B

Size が未定義なら中断。非負整数でなければ例外。ElementSize が未定義なら中断。1 以上 32 以下の整数でなければ例外。Size 及び ElementSize の値により生成されるストリングがヒープ領域の大きさより大きくなる場合も例外。それ以外の場合には 要素数 Size, 要素サイズ (要素のビット長) 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。

string_element(String, Position, ^Element) :: G

String が未定義なら中断。ストリング以外なら失敗。Position が未定義なら中断。非負整数以外または String の要素数以上なら失敗。それ以外の場合には String の Position 番目の要素を整数として Element とユニファイする。

string_element(String, Position, ^Element, ^NewString) :: B

String が未定義なら中断。ストリング以外なら例外。Position が未定義なら中断。非負整数以外または String の要素数以上なら例外。それ以外の場合には String の Position 番目の要素を整数として Element とユニファイするとともに String を NewString とユニファイする。

set_string_element(String, Position, NewElement, ^NewString) :: B

String が未定義なら中断。ストリング以外なら例外。Position が未定義なら中断。非負整数以外または String の要素数以上なら例外。NewElement が未定義なら中断。整数以外ならば例外。String の要素サイズ (要素のビット長) を越えてしまう場合には例外。それ以外の場合には String の Position 番目の要素を NewElement に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。

substring(String, Position, Length, ^SubString, ^NewString) :: B

String が未定義なら中断。ストリング以外なら例外。Position が未定義なら中断。非負整数以外または String の要素数以上なら例外。Length が未定義なら中断。正整数以外または Position+Length が要素数を越えていれば例外。それ以外の場合には 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。

set_substring(String, Position, SubString, ^NewString) :: B

String が未定義なら中断。ストリング以外なら例外。Position が未定義なら中断。非負整数以外または String の要素数以上なら例外。SubString が未定義なら中断。String と同じタイプ (要素サイズ) のストリング以外なら例外。また、Position+(SubString の長さ) が String の要素数を越えていれば例外。それ以外の場合には 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 とユファイする。

append_string(String1, String2, ^NewString) :: B

String1 及び String2 が未定義なら中断。同じタイプ (要素サイズ) のストリング以外なら例外。それ以外の場合には 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。

3.6.10　アトム関係

intern_atom(^Atom, String) :: B

String が未定義ならば中断。8 ビットストリング以外であれば例外。それ以外の場合には String を印字表現とするアトムを生成し Atom とユニファイする。

【注意】Multi-PSI V2 では組込述語でなく OS の機能として提供される。

new_atom(^Atom) :: B

新しいアトムを生成し Atom とユニファイする。このアトムは印字名を持たない。

atom_name(Atom, ^String) :: B

Atom が未定義ならば中断。アトム以外なら例外。それ以外の場合には Atom の印字表現を構成する文字コードからなるストリングを String とユニファイする。PDSS では 8 ビットストリング。

【注意】Multi-PSI V2 では組込述語でなく OS の機能として提供される。

atom_number(Atom, ^Number) :: B

Atom が未定義ならば中断。アトム以外なら例外。それ以外の場合には Atom のアトム番号を Number とユニファイする。アトム番号はアトムが作られた順に付けられる番号。

3.6.11　コード関係

predicate_to_code(Mod, Pred, Arity, ^Code) :: B

Mod が未定義であれば中断。アトム以外であれば例外。Pred が未定義であれば中断。アトム以外であれば例外。Arity が未定義であれば中断。非負整数でなければ例外。それ以外の場合には、モジュール名 Mod, 述語名 Pred, 引数個数 Arity であるコードを Code とユニファイする。モジュールが存在しない場合、及び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) 場合は、アトムの □ を Code にユニファイする。

code_to_predicate(Code, ^Mod, ^Pred, ^Arity, ^Info) :: B

Code が未定義であれば中断。3 要素ベクタ以外なら例外。Code の第 1 要素は ModuleName で、未定義であれば中断。アトム以外なら例外。指定された名前のモジュールがなければ例外。Code の第 2 要素は PredicateName で、未定義であれば中断。アトム以外なら例外。指定された名前の述語がなければ例外。Code の第 3 要素は PredArity で、未定義であれば中断。非負整数以外なら例外。それ以外であれば ModuleName, PredicateName, PredArity を、それぞれ Mod, Pred, Arity とユニファイする。Info には述語定義の情報として、スパイされていない (0) か、スパイされている (1) かを示す整数をユニファイする。

3.6.12　ストリーム・サポート

merge(In, ^Out) :: B

マージャーを生成し、そのマージャー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。マージャー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するこ

とが出来よう。

```
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).
                ...
```

3.6.13　高階機能

apply(Code, Args) :: B

Code が未定義であれば中断。3 要素ベクタ以外なら例外。Code の第 1 要素は ModuleName で、未定義であれば中断。アトム以外なら例外。指定された名前のモジュールがなければ例外。Code の第 2 要素は PredicateName で、未定義であれば中断。アトム以外なら例外。指定された名前の述語がなければ例外。Code の第 3 要素は Arity で、未定義であれば中断。非負整数以外なら例外。Args が未定義なら中断。Arity で指定された要素数のベクタ以外であれば例外。それ以外の場合にはモジュール名が ModuleName, 述語名が PredicateName である述語を引数環境 Args で呼び出す。

3.6.14　特殊入出力

read_console(^Integer) :: G

コンソール・ウィンドウから整数を読み込む。

【注意】入力待ちの時は処理系全体が停止する。

display_console(X) :: G

コンソール・ウィンドウに X の現在の値を (たとえ未定義でも) 書き出す。

put_console(X) :: G

X が整数の場合にはそれを ASCII コードと見なしてコンソール・ウィンドウに 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コンソール・ウィ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 がそれ以外の型か未定義の場合には無視される。

3.6.15　その他

raise(Tag, Type, Info) :: B

Tag が未定義変数であれば中断。正整数以外であれば例外。Type が基底項 (未定義変数を含まないデータ) でなければ中断。それ以外の場合には、この組込述語が実行された荘園から外側の荘園に向かって順にそれらの荘園が保持するタグと Tag の論理積を取り、その結果が最初にゼロ以外になった荘園のレポート・ストリームに以下のメッセージをユニファイする。(3.2 章 例外情報の項を参照)

```
exception(12, {0, Type, Info}, NewCode, NewArgv)
```

consume_resource(Red) :: B

Red が未定義変数であれば中断。正整数以外であれば例外。この組込述語が実行された荘園の許容リダクション数を Red で指定された値だけ強制的に減らす。これにより割り付けられたリダクション数が不足する場合には、通常のリダクションの消費と同様に resource_low となる。

hash(X, ^Value, ^NewX) :: B

X が未定義変数であれば中断。それ以外の場合には X をもとに整数を生成し Value とユニファイする。また、X を NewX にユニファイする。

current_processor(^ProcessorNumber, ^X, ^Y) :: B

この組込述語を実行したゴールを実行している PE の番号を ProcessorNumber に、PE の構成 (縦横の PE 台数) を X, Y にユニファイする。PDSS では常に ProcessorNumber=0, X=1, Y=1 をユニファイする。

current_priority(^CurrentPriority, ^ShoenMin, ^ShoenMax) :: B

この組込述語を実行したゴールのプライオリティを CurrentPriority に、所属荘園の最低 / 最高プライオリティを ShoenMin/ShoenMax にユニファイする。

3.7　マクロ記法

KL1 ではプログラムの記述性を向上する為に各種のマクロ展開機能を用意している。

- 定数記述のマクロ
- ユニフィケーションのマクロ
- 数値比較のマクロ
- 数値演算のマクロ
- 暗黙の引数マクロ
- 条件分岐のマクロ
- マクロ・ライブラリ

なお、現在はユーザーがマクロを定義することはできない。

3.7.1　定数記述の為のマクロ

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 "

デフォルト・タイプのストリングになる。PDSS では ASCII コードからなる 8 ビットストリングになる。(Multi-PSI V2 では JIS 漢字コードからなる 16 ビットストリング。)

ascii#"文字列 "

ASCII コードからなる 8 ビットストリングになる。

#"文字 "

デフォルト・タイプの文字コードになる。PDSS では文字を表す ASCII コードになる。(Multi-PSI V2 では JIS 漢字コード。)

c#"文字 "

文字を表す ASCII コードになる。

整数用

優先度	比較演算子	展開後のパターン
700 (xfx)	`X =:= Y`	equal(X,Y)
	`X =\= Y`	not_equal(X,Y)
	`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)

浮動小数点数用

優先度	比較演算子	展開後のパターン
700 (xfx)	`X $=:= Y`	floating_point_equal(X,Y)
	`X $=\= Y`	floating_point_not_equal(X,Y)
	`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)

表 1: 数値比較のマクロ

`ascii#文字`

文字を表す ASCII コードになる。ここで文字はアトムとする。(例: `ascii#'['`)

`key#lf`

改行を表す ASCII コード (10) になる。

`key#cr`

復改を表す ASCII コード (13) になる。

3.7.2 ユニフィケーションのマクロ

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。

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。

3.7.3 数値比較の為のマクロ

数値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整数演算、浮動小数点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 1 に示すものが用意されている。

優先度	型	演算子	展開後のパターン	生成される組込述語	
				整数式内では	浮動小数点式内では
	yfx	X + 1	Z	increment(X,Z)	
500	yfx	X + Y	Z	add(X,Y,Z)	floating_point_add(X,Y,Z)
	yfx	X - 1	Z	decrement(X,Z)	
	yfx	X - Y	Z	subtract(X,Y,Z)	floating_point_subtract(X,Y,Z)
	fx	- X	Z	minus(X,Z)	floating_point_minus(X,Z)
	yfx	X \/ Y	Z	or(X,Y,Z)	
	yfx	X /\ Y	Z	and(X,Y,Z)	
	yfx	X xor Y	Z	exclusive_or(X,Y,Z)	
	yfx	X * Y	Z	multiply(X,Y,Z)	floating_point_multiply(X,Y,Z)
400	yfx	X / Y	Z	divide(X,Y,Z)	floating_point_divide(X,Y,Z)
	yfx	X << Y	Z	shift_left(X,Y,Z)	
	yfx	X >> Y	Z	shift_right(X,Y,Z)	
300	xfx	X mod Y	Z	modulo(X,Y,Z)	
	xfy	X ** Y	Z		floating_point_pow(X,Y,Z)
		abs(X)	Z	abs(X,Z)	floating_point_abs(X,Z)
項の形で		min(X,Y)	Z	min(X,Y,Z)	floating_point_min(X,Y,Z)
		max(X,Y)	Z	max(X,Y,Z)	floating_point_max(X,Y,Z)
		\(X)	Z	complement(X,Z)	
		floor(X)	Z		floating_point_floor(X,Z)
		sqrt(X)	Z		floating_point_sqrt(X,Z)
		ln(X)	Z		floating_point_ln(X,Z)
		log(X)	Z		floating_point_log(X,Z)
		exp(X)	Z		floating_point_exp(X,Z)
		sin(X)	Z		floating_point_sin(X,Z)
		cos(X)	Z		floating_point_cos(X,Z)
		tan(X)	Z		floating_point_tan(X,Z)
		asin(X)	Z		floating_point_asin(X,Z)
		acos(X)	Z		floating_point_acos(X,Z)
		atan(X)	Z		floating_point_atan(X,Z)
		atan(X/Y)	Z		floating_point_atan(X,Y,Z)
		sinh(X)	Z		floating_point_sinh(X,Z)
		cosh(X)	Z		floating_point_cosh(X,Z)
		tanh(X)	Z		floating_point_tanh(X,Z)
		int(F)	I	floating_point_to_integer(F,I)	
		float(I)	F		integer_to_floating_point(I,F)

表 2: 数値演算の為のマクロ

3.7.4　数値演算の為のマクロ

数値演算の為のマクロは +, -, *, / 等の演算子を使ったもので、整数演算と浮動小数点演算がある。データタイプの自動変換は行われないので明示的に行わなければならない。このマクロ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。

なお、整数演算マクロ展開時に計算可能な場合には、その時点で計算が行われる。

演算子としては表 2 に示すものが用意されている。なお、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。

【注意】Multi-PSI V2 では浮動小数点数の演算に関しては加減乗除だけしか提供されていない。

本来、数値演算のマクロ展開が行われる所であるにもかかわらず、マクロ展開を抑制したい場合がある。この場合には逆クォートを用いることによっ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。

3.7.5　暗黙の引数マクロ

多くの述語が共通して持っているような引数 (例えばエラーレポート用のストリーム等) を述語定義の際に毎回明に記述するのは非常に面倒である。このような場合に便利なマクロ機能がここで説明する「暗黙の引数マクロ」である。 この暗黙の引数マクロ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 , ・・・ }.
```

ここで「引数名」は暗黙の引数を参照する際に用いる名前であり、アトム名で指定する。また、「タイプ」はその引数の種類を指定するもので、ここで宣言された種別に従って述語定義中の展開形式が決められる。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。

なお、グローバルな暗黙の引数の宣言はモジュール定義の頭部 (public 宣言の直後) に一回だけ書くことができる。これは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。

暗黙の引数マクロの展開を行うか、行わないかの指定は、述語定義単位で行うことができる。ある述語が暗黙の引数を持つ場合 (展開が必要) には、その節定義のネックを :- の代わりに --> を用いて定義すれば良い。 --> で定義された節の述語には、宣言されている引数が、宣言された順で、明に記述されている引数の前に付加される。すなわち、引数の順番は

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】

```
:- module test.
:- public XXX.
:- implicit a:oldnew, b:shared.

p(X) --> true | q(X), r.
   %% この時点では暗黙の引数は a, b のみ。
    ・・・

:- local_implicit d:oldnew.
   %% これ以降は暗黙の引数として a, b, d が有ることになる。
    ・・・

:- local_implicit d:shared, e:stream.
   %% これ以降は暗黙の引数として a, b, d, e が有ることになる。
   %% 引数名 d のタイプは shared に変わる点に注意。
    ・・・

:- local_implicit.
   %% これ以降は暗黙の引数として 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)

なる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。

以下、個々のタイプごとにその意味及び使用例を示そう。

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

とすればよい。こうすると、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演算のマクロ展開が行われる。

【例】 宣言:　:- implicit counter:shared.

a) 展開前:　p --> true | q, r.
　展開後:　p(Cnt) :- true | q(Cnt), r(Cnt).

b) 展開前:　p --> true | &counter <= &counter + 2, q.
　展開後:　p(Cnt) :- true | add(Cnt,2,Cnt1), q(Cnt1).

c) 展開前:
```
p --> true |
        &counter <= &counter+2, &counter <= &counter*2, q.
```
展開後:
```
p(Cnt) :- true |
        add(Cnt,2,Cnt1), multiply(Cnt1,2,Cnt2), q(Cnt2).
```

d) 展開前:
```
p --> true | &counter <= &counter(2), q.
```
展開後:
```
p(Cnt) :- true |
        set_vector_element(Cnt,2,Elem,Elem,_), q(Elem).
```

stream 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】 宣言: `:- implicit window:stream.`

a) 展開前: `p --> true | q, r.`
展開後: `p(Win) :- true | Win={In1,In2}, q(In1), r(In2).`

b) 展開前: `p --> true | &window <<= [putb("gazonk")], r.`
展開後: `p(Win) :- true | Win=[putb("gazonk")|Win1], r(Win1).`

oldnew 型 引数

この型は2個の引数が組になったものであり、Prolog における DCG と非常に似ているが、2個の引数が difference list に限定されていない点が異なっている。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、このような目的の為にもこの型の引数は非常に有用である。difference list として用いた場合には stream 型と同様以下のような、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, …]

引数の値が数値等である場合には、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。また、引数の値がベクタである場合の記法としてはなお、<= の右辺は整数演算のマクロ展開が、$<= の右辺は浮動小数点演算のマクロ展開が行われる。

【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】

&引数名 (ポジション)　%% <= の左辺以外

【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) 及び更新 (2) はボディ部のみで利用可能であり、組込述語 set_vector_element/5 が使われる。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 が使われ、ボディ部では set_vector_element/5 が使われる。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 difference list の要素として挿入し、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時々その値を参照するといった場合に便利)。

```
&引数名(old)
```

【例】 宣言:　`:- implicit mutter:oldnew.`

a) 展開前:
```
p --> true | q, r.
```
展開後:
```
p(Old,New) :- true | q(Old,Mid), r(Mid,New).
```

b) 展開前:
```
p --> true | &mutter <<= [naha], r.
```
展開後:
```
p(Old,New) :- true | Old=[naha|Mid], r(Mid,New).
```

c) 展開前:
```
p --> true | &mutter(3) <= naha, r.
```
展開後:
```
p(Old,New) :- true |
    set_vector_element(Old,3,_,naha,Mid), r(Mid,New).
```

d) 展開前:
```
p --> true | &mutter(1) <= &mutter(3), r.
```
展開後:
```
p(Old,New) :- true |
    set_vector_element(Old,3,Elem,Elem,Mid1),
    set_vector_element(Mid1,1,_,Elem,Mid2), r(Mid2,New).
```

e) 展開前:
```
p --> true | &mutter(2) <<= [naha,uhi,ehe], r.
```
展開後:
```
p(Old,New) :- true |
    set_vector_element(Old,2,[naha,uhi,ehe|Cdr],Cdr,Mid),
    r(Mid,New).
```

f) 展開前:
```
p --> true | &mutter <= &mutter+2, r.
```
展開後:
```
p(Old,New) :- true | add(Old,2,Mid), r(Mid,New).
```

g) 展開前:
```
p(X) --> true |
    X = [&mutter(old)|XX], &mutter <= &mutter+2, p(XX).
```
展開後:
```
p(Old,New,X) :- true |
    X=[Old|XX], add(Old,2,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。

終了処理の自動生成

暗黙の引数を持つ節がボディ部にユーザ定義ゴールの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それぞれ次のユニフィケーションが行われる。

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 {{ と }} で囲えば良い。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.

go :- true |
    merge(FILEout, FILEin),
    file:create(FILEin, "del.del", r),
    file:create(Answer, "/tmp/miyadel", w),
    loop(FILEout, Answer,[], 100 ,_).

loop(_) --> &counter =< 0 | true.
otherwise.
loop(A) --> true |
    &counter <= &counter - 1,
    &input <<= [getc(X)],
  {{ check(X, &output, &counter) }},
    loop(A).

check(ascii#a, Oh,Ot, Counter) :- true | Oh=[putt(Counter),nl|Ot].
otherwise.
check(_, Oh,Ot, _) :- true | Oh=Ot.
```

上の例では、input, output, counter なる 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 3 個の暗黙引数が有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ずそのまま 4 引数の述語として展開されている点に注意。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 を使って書かなければならない。

【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。

3.7.6　条件分岐のマクロ

これは DEC-10 Prolog のようにひとつの節の定義の中で複数の条件分岐の記述を許す記法である。例を示そう。

```
foo(X,Y) :- true |
    ( X=:=0 -> p(Y,Z);
      X > 0 -> q(Y,Z);
      otherwise;
      true  -> r(Y,Z) ),
    s(X,Z).
```

上記プログラムで -> の左辺に記述されたゴール (群) が条件式であり、右辺に記述されたゴールがその条件を満たした場合に実行してほしいゴール (群) である。コンパイラのプリプロセッサは上記のようなプログラムを読み込むと次のような複数の節に展開する。

```
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).
```

述語 '$foo/2/0' はプリプロセッ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 (従って、ユーザーは $ で始まる述語を使用しない方が良い)。なお、渡される引数は (条件分岐式内で現われた変数) と (条件分岐式外で現われた変数) の和となる。

【注意】展開系を見てわかるように条件式中には節のガード部で許されている組込述語しか書けない。

【注意】現在の Prolog 版コンパイラ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 2 つ以上の条件分岐式を書くこともできない。KL1 版は可能。

3.7.7　マクロ・ライブラリ

これはシステムのライブラリに登録されているマクロで、モジュール定義の先頭で使うことを宣言した場合にだけ使える。宣言は次のように行う。

```
:- with_macro マクロ定義名.
```

ここで、マクロ定義名はアトムである。

マクロ定義のファイルはシステムで決められたディレクトリに置かれる。この定義ファイルでは

```
fileio#normal      => 0.
fileio#end_of_file => 1.
fileio#read_error  => 2.
```

```
fileio#write_error => 3.
```

のようにマクロの宣言を行なう。 なお、現在の版では必ず xxx#yyy のように # を使ったもので、xxx はアトムでなければならない。

4　Micro PIMOS

Micro PIMOS とは PDSS 上での KL1 ユーザに種々のサービスを提供する非常に簡単な OS の名前であり、基本的に Single User, Single Task を前提として設計されている。Micro PIMOS が提供するサービスには以下のものがある。

- コマンド・インタプリタ
- 入出力機能 (ウィンドウ, ファイル等)
- コード管理
- 例外情報の表示

Micro PIMOS ではコマンド・インタプリタに対して与えられたコマンドは全てタスクと呼ぶ単位で実行される。タスクは 3.2 章で説明した荘園の機能を用いて実現されている。

【注意】例外発生時のタグは現在以下のビットが言語と Micro PIMOS で既に使用されている。この為ユーザはユーザが作った荘園の中で Micro PIMOS の機能を利用したい場合その荘園のタグに 15:31 ビットを使用してはならない。あるいは、その荘園を監視しているユーザのゴールがユーザの責任で、もう一度 Micro PIMOS に対して要求を出さなくてはならない。

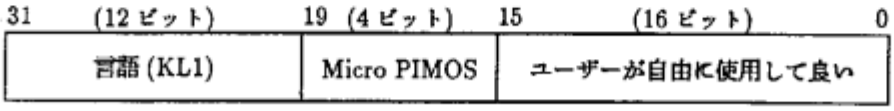

図 7: 荘園の例外タグ

4.1　コマンド・インタプリタ

Micro PIMOS のユーザは Micro PIMOS の起動時に生成されるコマンド・インタプリタを介して PDSS を使用することになる。コマンド・インタプリタが起動すると、プロンプトを出力してコマンドの入力待ちになる。プロンプトは通常は | ?- でデバッグ・モードの時は [debug]?- である。

なお、コマンド・インタプリタの起動時に "~/.pdssrc" なるファイルがあれば、それをコマンド・ファイルとして自動的に実行するので、適当な作業環境を設定することができる。コマンド・ファイルについては take/1 コマンドを参照のこと。

4.1.1　コマンド入力形式

コマンドラインまたはコマンドファイルには複数のコマンドを書くことができる。コマンドは区切りの文字により、以下の様に実行される。

- カンマ (" , ")
  前後の各コマンド群を、並列に実行する。
- セミコロン (" ; ")
  前のコマンド群の実行終了後、後ろのコマンド群を実行する (逐次実行)。
- 縦線 (" | ")
  前のコマンド群の実行終了後、後ろに書かれた変数 (複数の時はカンマで区切る) の具体化された値を表示する。後ろに all と書くと全ての変数の値を表示する。

また、コマンド群を ( ) で囲んで、ネストした記述をすることができる。

【例】
```
| ?- comp("bench");(stat(bench:primes(1,300,P))|P),save(bench).
     %  "bench.kl1" をコンパイルした後、ゴール bench:primes とコードの
     %  セーブを並列に実行する。ゴールの実行に関しては統計情報の表示を
```

% 指定。また実行後に変数 P の値を表示。

またコマンドライン(ターム)中の定数記述のマクロは展開される。マクロについては 3.7章を参照。

【例】
```
| ?- X=16#"FF",Y=16#Z,Z="FF"|all.
X=255
Y=16#"FF"
Z="FF"
```

4.1.2 コマンド

現在コマンド・インタプリタが提供しているコマンドには以下のものがある。ここで、ファイル名に属性とある場合、指定したファイル名に属性が指定されていなければ、そのファイル名に属性を付けたファイルに対してコマンドを実行することを示す。なお、ファイル名はストリング・アトムのどちらで指定しても良い。

組込述語

コマンド・インタプリタではボディ部に記述できる組込述語もコマンドとして実行可能である。また、 := と算術演算マクロを使った記述も使用可能である。

基本コマンド

ModuleName:Goal
ModuleName で示されるモジュールで Goal を実行する。最大リダクション数の上限値には環境変数で決められた値がセットされ、リダクション数がこれを越える場合には実行を継続するか中止するかをユーザに聞く。

help
ヘルプコマンドの一覧を表示する。

help(Type)
Type(整数またはアトム)で指定されたタイプのコマンドの一覧を表示する。タイプには次のものが指定できる。

1: builtin, 2: basic, 3: code, 4: dir, 5: debug, 6: env, all

gc
ヒープ領域の GC を起動する。

gc(all)
ヒープ領域とコード領域の GC を起動する。

take(FileName)
FileName で示されるコマンドファイルを実行する。コマンドファイルにはコマンド・インタプリタが実行できるコマンドならば何を書いても良い。コメントには KL1 のプログラムと同様 % と /* */ が使える。

cputime
PDSS を立ち上げてから現在までに消費した CPU タイムを表示する。CPU タイムの単位はミリ秒。

cputime(^Time)
PDSS を立ち上げてから現在までに消費した CPU タイムを Time とユニファイする。Time は整数で単位はミリ秒。

apply(CommandName, ArgsList)
CommandName で示される同じコマンドを ArgsList で指定された引数の各要素に対して実行する。CommandName には ModuleName:PredicateName も使用できる。

stat
現在のメモリーの状態を表示する。

stat(Commands)

任意のコマンド群 Commands を実行したときの実行時間 (CPU タイム) とリダクション数を表示する。

window(IOStream)

新しいウィンドウをオープンする。IOStream に流せるコマンドは、入出力機能の項を参照のこと。ウィンドウ名は自動的に付けられる。

add_op(Precedence, Type, Operator)

コマンド・インタプリタのウィンドウにオペレータを追加する。

【注意】定義済みのタイプと共存できない場合 (fx ↔ fy, xf ↔ yf, xfy ↔ xfx ↔ yfx)、古い定義を削除する。

remove_op(Operator)

コマンド・インタプリタのウィンドウからオペレータ Operator の定義を全て削除する。

remove_op(Precedence, Type, Operator)

コマンド・インタプリタのウィンドウからオペレータを削除する。

operator(Operator)

コマンド・インタプリタのウィンドウのオペレータ Operator の定義を表示する。

operator(OperatorName, ^Definition)

コマンド・インタプリタのウィンドウのオペレータ Operator の定義 (形式は {Precedence, Type}) を Definition にユニファイする。

replace_op_pool(^OldOpPool, NewOpPool)

旧オペレータプールを OldOpPool とユニファイし、オペレータプールを NewOpPool に変更する。オペレータプールの形式は [{OpName, [{Precedence, Type}, ...]}, ...] である。

change_op_pool(NewOpPool)

オペレータプールを NewOpPool に変更する。

halt

PDSS を終了する。この時開いていたウィンドウは全て自動的に閉じられる。

コードコマンド

comp(FileName)

FileName で指定される属性 ".kl1" の KL1 ソース・ファイルをコンパイルし、コード領域にロードする。新たにロードしたモジュールのトレース・モードは OFF。

comp(FileName, OutFileName)

FileName で指定される属性 ".kl1" の KL1 ソース・ファイルをコンパイルし、OutFileName で指定される属性 ".asm" のファイルに出力する。

compile(FileName)

FileName で指定される属性 ".kl1" の KL1 ソース・ファイルをコンパイルし、属性 ".asm" のファイルに出力する。その後、コード領域へのロード、属性 ".sav" のファイルへのセーブを行なう。FileName にはファイル名のリストも指定できる。

load(FileName)

FileName で指定される属性 ".sav" のセーブ・ファイル (これが無い場合には属性 ".asm" のアセンブラ・ファイル) をコード領域にロードする。新たにロードしたモジュールのトレース・モードは OFF。

dload(FileName)

FileName で指定される属性 ".sav" のセーブ・ファイル (これが無い場合には属性 ".asm" のアセンブラ・ファイル) をコード領域にロードする。新たにロードしたモジュールのトレース・モードは ON。この時デバッグ・モードは自動的に ON になる。

save(ModuleName)

ModuleName で示されるモジュールの実行コードを、環境変数で指定されたディレクトリ (デフォルトは "~/.PDSSsave", ch_savedir コマンドで変更可) にモジュール名をファイル名としてセーブする。

save(ModuleName, FileName)

ModuleName で示されるモジュールの実行コードを FileName で示される属性 ".sav" のファイルにセーブする。

save_all

既にロードされているモジュールの内、save(ModuleName) コマンドでセーブしていないモジュール全てを環境変数で指定されたディレクトリにセーブする。

ch_savedir(Directory)

オートロードや save_all の対象となるディレクトリを Directory で指定されるディレクトリに変更する。この時、Directory が存在するかチェックされる。

listing

既にロードされているモジュールの情報を表示する。

listing(^Modules)

既にロードされているモジュール名のアトムを要素とするリストを生成し、Modules とユニファイする。

public(ModuleName)

ModuleName で示されるモジュールの他のモジュールに公開している述語 (public 宣言されている述語) の一覧を表示する。

public(ModuleName, ^Public)

ModuleName で示されるモジュールの他のモジュールに公開している述語 (public 宣言されている述語) の情報を要素とするリストを生成し、Public とユニファイする。各要素は { 述語名アトム, アリティ } という 2 要素ベクタ。

ディレクトリコマンド

cd(Directory)

カレント・ディレクトリを Directory で指定するディレクトリに変更する。

pwd

カレント・ディレクトリのパス名を表示する。

pwd(^World)

カレント・ディレクトリのパス名を、World とユニファイする。

ls(WildCard)

WildCard で示されるファイルのパス名を表示する。

ls(WildCard, ^Files)

WildCard で示されるファイルのパス名をリストにし、Files とユニファイする。

rm(WildCard)

WildCard で示されるファイルをディレクトリから削除する。

デバッグコマンド

trace(ModuleName)

ModuleName で示されるモジュールのコードのトレース・モードを ON にする。このときデバッグ・モードが自動的に ON になる。

notrace(ModuleName)

ModuleName で示されるモジュールのコードのトレース・モードを OFF にする。

spy(ModuleName, PredicateName, Arity)

ModuleName で示されるモジュール中の PredicateName/Arity なる述語のスパイを行う。この時トレース・モード、デバッグ・モードは自動的に ON になる。

nospy(ModuleName, PredicateName, Arity)

ModuleName で示されるモジュール中の PredicateName/Arity なる述語のスパイを止める。

spying(ModuleName)

ModuleName で示されるモジュール中のスパイされている述語の一覧を表示する。

spying(ModuleName,~Spying)

ModuleName で示されるモジュール中のスパイされている述語のリストを生成し、Spying にユニファイする。要素は { 述語名アトム, アリティ } という 2 要素ベクタ。

debug

デバッグ・モードを ON にする。

nodebug

デバッグ・モードを OFF にする。

backtrace

バックトレース情報 (デッドロック情報) の表示モードを ON にする。

nobacktrace

バックトレース情報 (デッドロック情報) の表示モードを OFF にする。

varchk(FileName, Mode, Form)

FileName で指定される属性 ".kl1" の KL1 ソース・ファイルに対して Mode で指定したモードで変数チェックを行ない、Form で指定された形式でウィンドウに表示する。FileName にはファイル名のリストも指定できる。Mode, Form には以下のものが指定できる。

Mode:: o または one …… クローズ中に一つだけしか出てこない変数を表示する。
m または mrb …… MRB が黒くなる変数を表示する。
a または all …… one と mrb の両モードの変数を表示する。
Form:: s または short …… 該当するクローズを一行に出力する。
l または long …… 該当するクローズを改行 / インデントして出力する。

varchk(FileName, Mode)

モード Mode で変数チェックを行ない、long 形式で表示する。

varchk(FileName)

モード one で変数チェックを行ない、long 形式で表示する。

xref(FileName, Mode)

FileName で指定される属性 ".kl1" の KL1 ソース・ファイルに対して Mode で指定したモードでクロスリファレンス・チェックを行ない、ウィンドウに表示する。FileName にはファイル名のリストも指定できる。この場合、モジュール間呼び出しもチェックされる。Mode には以下のものが指定できる。

c または check	……	述語呼び出しのチェックのみを行なう。
l または list	……	参照リスト (述語の定義 / 参照の表) を出力する。
s または system	……	PDSS のモジュールを参照している述語を出力する。
b または builtin	……	ボディの組込述語を使っている述語を出力する。
g または guard	……	ガードの組込述語を使っている述語を出力する。
a または all	……	上記の全てを出力する。
リスト	……	各要素に指定されたもののみの参照リストを出力する。 指定できるのは、 • Module (そのモジュール全体) • Predicate/Arity (ボディの組込述語) • Module:Predicate/Arity (定義された述語) • guard(Predicate/Arity) (ガードの組込述語)
short	……	述語の情報をウィンドウに出力しないで check を実行。
short(Mode)	……	述語の情報をウィンドウに出力しないで Mode を実行。
update	……	複数ファイルを指定した場合に、同一のモジュール名があった場合に警告を出さずに後ろ側を有効として check を実行。
update(Mode)	……	複数ファイルを指定した場合に、同一のモジュール名があった場合に警告を出さずに後ろ側を有効として Mode を実行。

xref(FileName)

モード check でクロスリファレンス・チェックを行なう。

xref(FileName, Mode, OutFile)

クロスリファレンス・チェックを行い、リストを OutFile に出力する。Mode には check および c 以外が指定できる。

profile(ModuleName, Mode)

ModuleName で指定されるモジュール内で定義されている述語が呼び出された回数とサスペンドした回数をウィンドウに表示する。ModuleName にはモジュール名のリストも指定できる。Mode には以下のものが指定できる。

c または call …… 呼び出された回数が多い順にソートして表示する。
s または susp …… サスペンドした回数が多い順にソートして表示する。
n または no …… コード領域に定義されている順に表示する。

profile(ModuleName)

モード call で profile コマンドを実行する。

reset_profile(ModuleName)

ModuleName で指定されるモジュール内に定義されている述語が呼び出された回数とサスペンドした回数をリセットする。ModuleName にはモジュール名のリストも指定できる。

環境コマンド

これらのコマンドはコマンド・インタプリタが持っている環境変数の値を変更する為のものである。コマンド・インタプリタの環境変数には表 3に示すものがある。

setenv(Name, Value)

Name で示される環境変数を Value で指定する値に設定する。Name はアトムに、Value は基底項になってから環境変数に登録される。

getenv(Name, ^Value)

Name で示される環境変数の値を Value とユニファイする。

printenv(Name)

Name で示される環境変数の値を表示する。

printenv

名前(アトム)	意味
world	カレント・ディレクトリのパス名のストリング。
trace	トレーサのトレース・モードで値は on または off。(初期値は off)
backtrace	バックトレース情報の表示モードで値は on または off。(初期値は on)
modules	コマンドをサーチするモジュール名(アトム)を要素とするリスト。
reduction	タスク生成時に与えるリダクション数制限で単位は 10000 リダクション。 (0 < 数値 < 100000, 初期値は 10000)
ucounter	作業用のウィンドウやファイルの名前生成用カウンタ。
savedir	save/1 や save_all の対象となるディレクトリのパス名のストリング。 (初期値は "~/.PDSSsave")
loaddir	オートロードの時にファイルをサーチするディレクトリのパス名のリスト。 (初期値は ["~/.PDSSsave", ライブラリ・ディレクトリのパス名, ...] 注: ライブラリ・ディレクトリのパス名は複数でマシンごとに違っている。)
auto_load	オートロードを行うかどうかのフラグで値は yes または no。(初期値は yes)
plength	コマンド・インタプリタのウィンドウに表示される構造体の最大長さ。(初期値は 20)
pdepth	コマンド・インタプリタのウィンドウに表示される構造体の最大深さ。(初期値は 5)
pvar	コマンド・インタプリタのウィンドウでの変数の表示モードで値は nu または na。 (nu では _0,_1,_2, ... 表示, na では A,B,C, ... 表示, 初期値は nu)

表 3: コマンド・インタプリタの環境変数

コマンド・インタプリタの持つ環境変数のすべての値を表示する。

resetenv

コマンド・インタプリタの持つ環境変数のすべての値を初期化(立ち上げた時と同じ値に)する。

4.2　入出力機能

Micro PIMOS の入出力サービスにはウィンドウとファイルの2種類がある。ユーザーは入出力サービスを利用したい場合、Micro PIMOS が提供する述語を呼び出すことにより I/O に対するコマンド・ストリームと呼ぶストリームを得ることができる。コマンド・ストリームには以下で示す種々の I/O コマンドを流すことができ、それによってユーザーは入出力処理を行うことができる。コマンド・ストリームを □ で閉じると、I/O は自動的にクローズされる。また、コマンド・ストリームにはマージャーが挿入されている。

4.2.1　コマンド・ストリームの獲得

ウィンドウ

window:create(Stream, WindowName, ^Status)

ウィンドウ名 WindowName (8 ビットストリング)のウィンドウを生成し、そのウィンドウに繋がるコマンド・ストリームを Stream とユニファイする。Status には以下のものがユニファイされる。

success ····· 成功
error(cannot_create_window) ····· 失敗: ウィンドウがオープンできない
error(bad_window_name_type) ····· 失敗: WindowName が 8 ビットストリングでない

【注意】生成直後のウィンドウは隠れた状態にある。show コマンドで表示される。

window:create(Stream, WindowName)

ウィンドウ名 WindowName (8 ビットストリング)のウィンドウを生成し、そのウィンドウに繋がるコマンド・ストリームを Stream とユニファイする。生成が失敗するとタスク全体が強制終了される。

【注意】生成直後のウィンドウは隠れた状態にある。show コマンドで表示される。

ファイル

file:create(Stream, FileName, Mode, ^Status)

ファイル名 FileName (8 ビットストリング) のファイルを、モード Mode (アトム, r: リード , w: ライト, a: アペンド) でオープンし、そのファイルに繋がるコマンド・ストリームを Stream とユニファイする。Status には以下のものがユニファイされる。

success	…… 成功
error(cannot_open_file)	…… 失敗: ファイルがオープンできない
error(bad_file_name_type)	…… 失敗: FileName が 8 ビットストリングでない
error(bad_open_mode_type)	…… 失敗: Mode がアトムでない
error(bad_open_mode)	…… 失敗: Mode が r,w,a 以外のアトムである

file:create(Stream, FileName, Mode)

ファイル名 FileName (8 ビットストリング) のファイルを、モード Mode (アトム, r: リード , w: ライト, a: アペンド) でオープンし、そのファイルに繋がるコマンド・ファイルのオープンが失敗するとタスク全体が強制終了される。

4.2.2　コマンド

コマンド・ストリームに流せるコマンドを以下に示す。これらは特に指定の無い限りウィンドウ / ファイルに共通に使うことができる。

入力用コマンド

I/O からの入力を要求する。

getc(^Char)

I/O から 1 文字を読み込み、そのコード (ASCII コード, $0 \leq$ コード ≤ 255) を Char とユニファイする。エンド・オブ・ファイルならばアトム end_of_file をユニファイする。

getl(^String)

I/O から 1 行を読み込み、それを 8 ビットストリングに変換し String とユニファイする。エンド・オブ・ファイルならばアトム end_of_file をユニファイする。

getb(^Buffer, Size)

I/O から Size(Size $>$ 0) で示される数だけ文字を読み込み、それを 8 ビットストリングに変換し Buffer とユニファイする。ウィンドウからの入力で途中で改行になった場合や、入力途中でエンド・オブ・ファイルになった場合は、そこまでの文字を入力とする。エンド・オブ・ファイル状態ならばアトム end_of_file をユニファイする。

gett(^Term)

I/O からターム 1 個を構成する文字列 (ピリオド＋改行 または ピリオド＋スペース まで) を読み込み、その文字列の構文解析を行ないタームに変換し、結果を Term とユニファイする。構文解析でエラーがある場合、ウィンドウではそのウィンドウにエラーを通知し、再び入力待ちになる。ファイルではコマンド・インタプリタのウィンドウにエラーを通知し、その次のタームを読み込む。エンド・オブ・ファイルならばアトム end_of_file をユニファイする。

gett(^Term, ^Status)

gett/1 コマンドとほぼ同じだが、Status に以下のものがユニファイされる。

success　…… ターム読み込み成功
syntax_error(Position)　…… 文法エラー
ambiguous(Position)　…… 曖昧な表現
end_of_file　…… エンド・オブ・ファイル
eof_in_quote　…… クォート中にエンド・オブ・ファイル

Status が syntax_error, ambiguous, eof_in_qoute の場合は、Term にはトークン・リスト (付録-1 gettkn/4 を参照) をユニファイする。

getft(^Term, ^NumberOfVariables)

gett/1 コマンドと殆ど同じであるが、ターム中の変数は $VAR(N,VN) で表す。N は変数番号 ($0 \leq N < $ NumberOfVariables), VN は変数名 (8 ビットストリング) である。また、変数の種類数を NumberOfVariables にユニファイする (種類数 ≥ 0)。エンド・オブ・ファイルならば Term にアトム end_of_file を NumberOfVariables には 0 をそれぞれユニファイする。

getft(^Term, ^NumberOfVariables, ^Status)

getft/2 と殆ど同じだが、Status については gett/2 を参照。

skip(Char)

文字コード Char が出現するか、エンド・オブ・ファイルとなるまで読み飛ばす。

【注意】読み込みが終了した (エンド・オブ・ファイルになった) あとの入力要求にはアトム end_of_file を返し続ける。

出力用コマンド

I/O へ出力を要求する。

putc(Char)

I/O へ文字コード Char (ASCII コード, $0 \leq$ Char ≤ 255) で示される文字を書き出す。

putl(String)

I/O へ String (8 ビットストリング) で示される文字列を書き出し、改行を行う。

putb(Buffer)

I/O へ Buffer (8 ビットストリング) で示される文字列を書き出す。改行は行わない。

putb(Buffer, Count)

I/O へ Buffer の先頭から Count 文字を出力する。Buffer の長さが Count より小さければ putb/1 と同じ。

putt(Term, Length, Depth)

I/O へ Term で示されるタームを書き出す。この時、構造体の長さが Length (Length > 0), 深さが Depth (Depth > 0) を越える部分は ... と省略して出力される。Prolog の write に相当。

【注意】アトムにクォート付けを行わないので、このコマンドで書き出したものが必ずしも gett,getft で読めるとは限らないので注意。

putt(Term)

putt/3 コマンドと殆ど同じ。構造体の長さ, 深さの制限はデフォルト値を使用する。

puttq(Term, Length, Depth)

putt/3 と殆ど同じだが、必要に応じてアトムにクォート付けを行う。Prolog の writeq に相当。

puttq(Term)

putt/1 と殆ど同じだが、必要に応じてアトムにクォート付けを行う。

nl

改行を行う。

tab(N)

空白を N ($0 \leq N < 1000$) で示される数だけ出力する。

【注意】Micro PIMOS では I/O デバイスとの通信回数を減らすため、出力データのブロッキングを行ない出力用の

バッファに溜めているので、コマンドを送っただけでは出力されない。バッファがI/Oへ送られるのは、以下の場合である。

- バッファが一杯になった時
- flush コマンドを受け付けた時
- I/O を閉じた時
- 入力要求コマンド, show/hide コマンドを受け付けた時 (ウィンドウのみ)

出力形式の制御

putt/1, puttq/1 コマンド実行時における構造体の出力制限のデフォルト値等を変更する。

print_length(Length)

構造体の長さの制限のデフォルトを Length (Length > 0) で示される値に設定する。初期値はウィンドウに対しては 10、ファイルに対しては 100 である。

print_depth(Depth)

構造体の深さの制限のデフォルトを Depth (Depth > 0) で示される値に設定する。初期値はウィンドウに対しては 10、ファイルに対しては 100 である。

print_var_mode(VariableMode)

変数を表わすターム $VAR(N,VN), $VAR(N) の出力形式を変更する。VariableMode はアトムで 'na' または 'nu' のいずれかを与える。初期値は 'na' である。

na – Name Mode	::	$VAR(N,VN) → VN (変数名ストリング) で出力
		$VAR(N) → A,B,C ... で出力
nu – Number Mode	::	$VAR(N,VN) → _N (変数番号) で出力
		$VAR(N) → _N (変数番号) で出力

出力用バッファコマンド

出力用のバッファの制御に関するコマンドである。

flush(^Status)

バッファに溜ったデータを出力する。出力が完了すると Status にアトム 'done' がユニファイされる。

buffer_length(BufferLength)

バッファのサイズを BufferLength (BufferLength > 0) に変更する。バッファサイズの初期値はウィンドウが 512 バイト, ファイルが 2048 バイト。

演算子

構文解析に用いる演算子に関するコマンドである。

add_op(Precedence, Type, OperatorName)

順位 Precedence ($1 \leq$ Precedence ≤ 1200), 型 Type (アトム: fx, fy, xf, yf, xfy, xfx, yfx のいずれか), 名前 OperatorName (アトム) の演算子を追加する。

【注意】定義済みのタイプと共存できない場合 (fx ↔ fy, xf ↔ yf, xfy ↔ xfx ↔ yfx)、古い定義を削除する。

remove_op(OperatorName)

名前 OperatorName の定義を全て削除する。

remove_op(Precedence, Type, OperatorName)

順位 Precedence ($1 \leq$ Precedence ≤ 1200), 型 Type (アトム: fx, fy, xf, yf, xfy, xfx, yfx のいずれか), 名前 OperatorName (アトム) の演算子を削除する。

operator(OperatorName, ^Definition)

名前が OperatorName (アトム) の演算子の定義を要素とするリストを生成し Definition とユニファイする。

要素は {順位, 型} の形式である。

replace_op_pool(^OldOpPool, NewOpPool)

旧オペレータプールを OldOpPool とユニファイし、オペレータプールを NewOpPool に変更する。オペレータプールの形式は [{OpName, [{Precedence, Type}, ...]}, ...] である。

change_op_pool(NewOpPool)

オペレータプールを NewOpPool に変更する。

一括処理

do(CommandList)

コマンドのリスト CommandList を一括して I/O に送る。コマンド・ストリームのマージを行っても CommandList 内のコマンドの連続性は保証される。

制御コマンド

close(^Status)

I/O をクローズする。close コマンドを送った後はコマンド・ストリームにコマンドを流すことはできない。(□ で閉じることができるのみ) Status にはアトム 'success' がユニファイされる。

ウィンドウコマンド

ウィンドウにのみ有効なコマンドである。

show

隠れているウィンドウを表示する。

hide

表示されているウィンドウを隠す。

clear

ウィンドウをクリアする。

beep

ベルを鳴らす。

prompt(^Old, New)

gett, getft コマンド実行時に出力される現在のプロンプトを Old (8 ビットストリング) にユニファイし、プロンプトを New (8 ビットストリング) に変更する。プロンプトの初期値は "?- "。

4.3　ディレクトリの管理

Micro PIMOS のディレクトリ・サービスを利用したい場合、入出力サービスと同様に、Micro PIMOS が提供する述語を呼び出すことによりディレクトリ・コマンド・ストリームと呼ぶストリームを得ることができる。ユーザーはこのストリームにコマンドを流すことによりディレクトリに関する処理を行うことができる。コマンド・ストリームが不要になった時は □ で閉じればよい。

4.3.1　コマンド・ストリームの獲得

directory:create(Stream, DirectoryName, ^Status)

ディレクトリ名 DirectoryName (8 ビットストリング) のディレクトリをアクセスし、ディレクトリに繋がるコマンド・ストリームを Stream とユニファイする。Status には以下のものがユニファイされる。

success	…… 成功
error(cannot_access)	…… 失敗: ディレクトリにアクセスできない
error(bad_directory_name_type)	…… 失敗: DirectoryName が 8 ビットストリングでない

4.3.2 コマンド

ディレクトリのコマンド・ストリームに流せるコマンドを以下に示す。

pathname(^PathName)

ディレクトリのフルパス名 (8 ビットストリング) を PathName にユニファイする。

listing(WildCard, ^FileNames, ^Status)

WildCard (8 ビットストリング) で表現されるファイル群のパス名のリストを生成し、FileNames にユニファイする。Status には以下のものがユニファイされる。

success	…… 成功
error(cannot_listing)	…… 失敗: リスティングできなかった

delete(WildCard, ^Status)

WildCard (8 ビットストリング) で表現されるファイル群をディレクトリから削除する。Status には以下のものがユニファイされる。

success	…… 成功
error(cannot_delete)	…… 失敗: 削除できなかった

open(Stream, FileName, Mode, ^Status)

ファイル名 FileName (8 ビットストリング) のファイルを、モード Mode (アトム, r: リード , w: ライト, a: アペンド) でオープンし、そのファイルに繋がるコマンド・ストリームを Stream とユニファイする。Status には以下のものがユニファイされる。

success	…… 成功
error(cannot_open_file)	…… 失敗: ファイルがオープンできない
error(bad_file_name_type)	…… 失敗: FileName が 8 ビットストリングでない
error(bad_open_mode_type)	…… 失敗: Mode がアトムでない
error(bad_open_mode)	…… 失敗: Mode が r,w,a 以外のアトムである

4.4 入出力用のデバイス・ストリーム

Micro PIMOS 内から入出力デバイスの機能を直接利用できるように、以下のライブラリを用意している。これらの機能は KL1 で Micro PIMOS 以外の OS (例えば PIMOS) などを記述する為の機能であり、通常のユーザは以下で説明するデバイス・ストリームを使用する必要は無い。これにより得られるデバイス・ストリームは Micro PIMOS により監視されているので、間違ったコマンド等を送ってもユーザー・タスクの失敗になるだけで、処理系自身がおかしくなることはない。

4.4.1 デバイス・ストリームの確保

Micro PIMOS 内からデバイス・ストリームは次の述語により取り出すことができる。それぞれのモジュール名は mpimos_io_device でもよい。

mpimos_window_device:windows(Stream)

ウィンドウ・デバイスの機能を持つストリームを Stream とユニファイする。

mpimos_file_device:files(Stream)

ファイル・デバイスの機能を持つストリームを Stream とユニファイする。

mpimos_timer_device:timer(Stream)

タイマ・デバイスの機能を持つストリームを Stream とユニファイする。

4.4.2 コマンド

それぞれのデバイス・ストリームに送ることができるコマンド、及びオープンされたウィンドウ、ファイル、ディレクトリの各ストリームに送ることができるコマンドは、付録-1 に説明のある入出力用のデバイス・ストリームの

ものと同じであるので、そちらを参照のこと。ただし、ファイル / ウィンドウの入出力コマンドとしては以下のものしか使用できない。

- ウィンドウ

  入力 getl(^Line, ^Status, Cdr) のみ使用可能。
  getc/3, getb/4, gettkn/4 は使用できない。

  出力 putb(Buffer, ^Status, Cdr) のみ使用可能。
  putc/3, putl/3, putt/5 は使用できない。

- ファイル

  入力 getb(Size, ^Buffer, ^Status, Cdr) のみ使用可能。
  getc/3, getl/3, gettkn/4 は使用できない。

  出力 putb(Buffer, ^Status, Cdr) のみ使用可能。
  putc/3, putl/3, putt/5 は使用できない。

4.5　コードの管理

Micro PIMOS におけるコード管理機能の主なものには次のものがある。

- ロードされたモジュールの名前とそのモジュール中の各種情報 (例えば、他のモジュールに公開している述語名の一覧, スパイされている述語名の一覧等) を管理し要求に応じて表示する機能。
- コマンド・インタプリタから save(ModuleName) や save_all コマンドを使ってセーブしたモジュールのオート・ロード機能。

オートロードの対象となるディレクトリは、コマンド・インタプリタの環境変数 loaddir の値で決められる。オート・ロード機能を使うためにはユーザーは自身のホーム・ディレクトリの直下に "~/.PDSSsave" なるディレクトリを作ることが望ましい。これは環境変数 loaddir の第一要素および savedir のデフォルト値が "~/.PDSSsave" である為で、これらの環境変数の値は変更可能である。オート・ロード機能を抑制したい時は環境変数 auto_load の値を 'no' にすることで可能である。

4.6　例外情報の表示

PDSS における KL1 で規定している例外には 付録-7 に示したものがある。Micro PIMOS ではユーザ・タスク内で発生した例外の情報はコマンド・インタプリタが持つウィンドウに表示される。また、例外を起こしたタスクは自動的に強制終了され、そのタスクが使用していた資源 (ウィンドウやファイル) は解放される。

Micro PIMOS により報告される例外には、言語で規定している例外以外に Micro PIMOS によって規定されている例外もある。ウィンドウに対するコマンドが誤りだったり、存在しないファイルをオープンしようとした場合などである。これらの例外発生時にも言語定義例外と同様、例外の情報がコマンド・インタプリタが持つウィンドウに表示され、例外を起こしたタスクは自動的に強制終了され、そのタスクが使用していた資源は解放される。

5　起動とオプション・パラメタ

PDSS を実行する場合、通常は GNU-Emacs の下で実行する。実行環境を考えると、これが最も良い方法であり、PDSS の全ての機能が使える。PDSS 単体で動かすこともできるが、この場合には機能が制限される。

5.1　GNU-Emacs 下での実行

PDSS を GNU-Emacs の下で実行する為には、GNU-Emacs に対して以下のようなコマンドを与えれば自動的にライブラリがロードされ PDSS が起動される。

meta-X pdss *return*

立ち上げ時のオプションを指定したい時には meta-X に先だって ctrl-U を入力する。オプション・パラメタの内容については後で述べる。

ctrl-U meta-X pdss *return*
PDSS Option?: [パラメタ] *return*

PDSS が起動されると、まずコンソール・ウィンドウと呼ばれるウィンドウが作られる。このウィンドウは実行のトレースを行ったり read_console や display_console の入出力先として使用されるウィンドウである。PDSS はコンソール・ウィンドウを生成した後ランタイム・サポート・ルーチンや Micro PIMOS の各種モジュールをロードし、Micro PIMOS の起動を行う。Micro PIMOS が起動されるとコマンド・インタプリタの入出力用ウィンドウが自動的に生成されユーザからのコマンド入力待ちとなる。

GNU-Emacs の下で起動された場合、PDSS からの入力要求は非同期入力となるので、入力要求のために処理系全体が停止することはなくなる (トレーサー等のコンソールに対する入力要求では全体が停止する)。また、ウィンドウへのコントロール・キー入力により PDSS を制御することができる。これは GNU-Emacs のライブラリで定義されているもので、次のようなコマンドが提供される。これ以外にも定義されているものがあるので詳しくは 付録-9 を参照のこと。

ctrl-C ctrl-C	::	トレース・フラグをオンにする。
ctrl-C ctrl-Z	::	割込みコード 1 を入力。 Micro PIMOS ではタスクの強制終了を意味する。
ctrl-C ctrl-T	::	割込みコード 2 を入力。 Micro PIMOS ではタスクのその時点までのリダクション数を表示。
ctrl-C !	::	GC を起動する。
ctrl-C @	::	PDSS の実行を強制終了させる。
ctrl-C ctrl-B	::	PDSS に関する Window Buffer Menu の生成。
ctrl-C ESC	::	PDSS システムを再起動する。
ctrl-C k	::	現在カーソルが表示されている Window の中身を削除する。
ctrl-C ctrl-K	::	PDSS で生成した Window の中身を削除する。
ctrl-C ctrl-Y	::	最後に入力した文字列を再表示する。
ctrl-C ctrl-F	::	組込述語のマニュアルを表示する。
ctrl-C f	::	コマンド・インタプリタへのコマンドのマニュアルを表示する。

【注意】ctrl-X k で PDSS に関するウィンドウを削除した場合、その後の実行結果は保証されていない。

5.2　PDSS 単体での実行

PDSS を GNU-Emacs を使わずに実行する為には以下のコマンドを実行すれば良い。

pdss [パラメタ] *return*

GNU-Emacs を使わない場合には各ウィンドウへの出力は全て混ざって出力される。また、どこか1つのウィンドウで入力待ちとなると処理系全体が停止する。ウィンドウへのコントロール・キーによる制御も使えないので、代わりにキーボード割り込みがサポートされる。これはキーボードから ctrl-C を入力することにより行われ、プロンプトに従って制御用コマンドを入力することができる。

5.3　オプション・パラメタ

起動時に指定できるオプション・パラメタと指定方法について述べる。パラメタの種類には以下のものがあり、これらを指定することにより標準と違った環境で実行することができる。

`-hNNN`	:: Heap Area の大きさを `NNN` word とする。(デフォルトは 200000 word, 1 word = 8 byte)
`-cNNN`	:: Code Area の大きさを `NNN` byte とする。(デフォルトは 500000 byte)
ファイル名	:: 標準のスタートアップ・ファイルの代わりに指定したファイルをスタートアップ・ファイルとして立ち上げる。
`+t/-t`	:: スタートアップ・ファイルを使った起動をする / しない。(デフォルトは `+t`)
`-v`	:: トレーサー等における変数の表示方法を指定する。通常は `A,B,C, ...` という名前で表示するが、`-v` を指定すると Heap Bottom からの相対アドレス `_XXX` で表示する。このアドレスは GC によって変わるので注意が必要。
`-dNNN`	:: ゴールのスケジューリングを Depth First に変更する。`NNN` は (execute 命令による)TRO による実行の深さ制限を指定する。
`-bNNN`	:: ゴールのスケジューリングを Breadth First に変更する。`NNN` は (execute 命令による)TRO による実行の深さ制限を指定する。
`-rRR,SS,NNN`	:: ゴールのスケジューリングを Depth First を基本とし、乱数により一部ゴールをスケジューリング・キューの最後に回すことにより、マルチプロセッサの場合の非決定的な実行順序をシミュレートするモードに変更する。`RR` はゴールをスケジューリング・キューの最後に回す割合でパーセントで指定する。`SS` は擬似乱数を生成する場合の種を指定する。`NNN` は (execute 命令による)TRO による実行の深さ制限を指定する。
`-a`	:: タイマー割り込みを禁止する。dbx で処理系自身をデバッグする時に使う。

これらのオプション・パラメタを指定する方法は2通り用意されている。

- 起動時に PDSS Option ?: への入力, pdss コマンドの引数により指定する。

例 -1)

```
PDSS Option ?: -h300000 -c50000 -v     (GNU-Emacs の下で実行)
```

例 -2)

```
[UNIX]% pdss -h300000 -c50000 -v       (PDSS 単体で実行)
```

- 環境変数 (PDSSOPT) により指定する。

例)

```
[UNIX]% setenv PDSSOPT "-h300000 -c50000 -v"
[UNIX]% pdss
```

6　トレーサ

PDSS で実現しているトレーサについて説明する。

6.1　考え方

PDSS のトレースは基本的にゴール単位のトレースであり、ゴールが次のような状態になった時にトレースが行われる。このタイミングをトレース・ポイントと呼ぶ。

- ゴール呼び出し
- 具体化を待つ為の中断
- 中断からの復帰
- ゴールの失敗
- スワップ・アウト (割込み、もしくはより高いプライオリティがスケジュールされたことによる)

KL1 のトレースの方法としては「コードに注目したトレース」と実行中の「ゴールに注目したトレース」の 2 通りが考えられる。

コードに注目したトレースとは、トレースしたいコードが呼び出された時にトレースを行うものであり、モジュールごとにトレース・モードを指定できる。以下ではこれをコード・トレースと呼ぶ。また、更に細かく、特定の述語にだけ注目してトレースを行うこともでき、これをコードのスパイと呼ぶ。

ゴールに注目したトレースとは、生成された各ゴールごとにその子孫のゴール (すなわちそのゴールの実行によって生成されるゴール) をトレースするか、あるいはトレースしないかを指定するものである。以下ではこれをゴール・トレースと呼ぶ。また、特に指定したゴールの子孫だけをトレースするという指定もでき、これをゴールのスパイと呼ぶ。

例を考えてみよう。以下のプログラムで foo がゴール・トレースの状態にあり、bar がその状態になかったとすると、foo から呼び出される p も、更にそこから呼ばれる q, r もゴール・トレースの状態になるが、同じコード p, q, r でも、bar から呼び出されたゴールはゴール・トレースの状態にならない。

```
foo :- p.      bar :- p.      p :- q, r.
```

PDSS ではコード・トレースの ON/OFF を実行前 / 実行中にモジュール名で指定することができ、ゴール・トレースの ON/OFF は最初は必ず ON で、実行中にトレーサーで見ているゴールを OFF にすることができるようになっている。そして、このコード・トレースとゴール・トレースの両方が ON になっているゴールだけをトレースするようにしている。

スパイについては、コード・スパイの ON/OFF を実行前 / 実行中にモジュール名と述語名で指定することができ、ゴール・スパイは実行中にトレーサーで見ているゴールにだけ設定することができる。そして、次のような 4 通りの組み合わせを指定できるようになっている。

- コードがスパイされているものである。
- ゴールがスパイされているものである。
- コードがスパイされているものであるか、または、ゴールがスパイされているものである。
- コードがスパイされているものであり、且つ、ゴールもスパイされているものである。

6.2　見方

トレーサの表示は次のような 4 つの情報を持っている。

```
[0012] CALL *$ module:goal(a1,a2)
1      2    3  4
```

1. このゴールが所属している荘園 ID

2. トレース・ポイントの種類::

CALL :: ゴール・キューから取り出されたことによるゴール呼び出し
Call :: TRO によるゴール呼出し
SUSP :: 引数の具体化を待つ為の中断
Susp :: プライオリティの具体化を待つ為の中断
RESU :: 中断からの復帰
FAIL :: ゴールの失敗
SWAP :: スワップ・アウト

3. スパイ・フラグ::

* :: このゴールが実行しているコードがスパイされている。
$ :: このゴールがスパイされている。

4. ゴール

引数のタームのうち、複数箇所から参照されている可能性のあるもの (所謂 MRB-ON の状態) にはそのターム表示の直後に x が表示される。
また、変数はその性質により以下のように表示される。

- 普通の未定義変数:
  先頭にアルファベットの大文字か '_' が付いた数字... X1, _2361
- ゴールによって具体化が待たれている変数:
  普通の未定義変数の表記の直後に '~' が付いたもの... X1~, _2361~
- マージの入力である変数:
  普通の未定義変数の表記の直後に '^' が付いたもの... X1^, _2361^

なお、これらの表示の他、トレースしているプライオリティが変化するごとに、プライオリティが表示される。

6.3　コマンド

トレーサに対するコマンドを次のシンタックスで示す。

コマンド名 :: 入力形式 引数 [オプション]

Help :: ?
コマンドのヘルプ。

No Trace :: X
以下トレースしない。

No Goal Trace :: x
そのゴールの子孫のゴール・トレースを OFF にする。これにより、そのゴールから呼び出されるゴール群のトレースは行われない。

Set Goal Spy :: g
指定した時点で表示しているゴールのスパイを行う。

Reset Goal Spy :: G
指定した時点で表示しているゴールのスパイを止める。

Set Module Debug Mode :: d MODULE [MODULE ...]
指定したモジュールのデバッグ・フラグをセットする。これにより、このモジュールを実行した場合にコード・トレースが行われる。

Reset Module Debug Mode :: D MODULE [MODULE ...]
指定したモジュールのデバッグ・フラグをリセットする。これにより、このモジュールを実行してもコード・トレースは行われない。

Set Procedure Spy :: p MODULE:PROCEDURE [MODULE:PROCEDURE ...]

指定した述語コードのスパイを行う。

Reset Procedure Spy :: P MODULE:PROCEDURE [MODULE:PROCEDURE ...]

指定した述語コードのスパイを止める。

Step :: s [COUNT]

次のトレース・ポイントで止まる。COUNT が指定された場合にはそのステップ数だけトレースを行なった後で止まる。

Step to Next Spied Procedure :: sp [COUNT]

次のスパイされている述語コードが実行されるまでトレースを行い止まる。COUNT が指定された場合にはその回数だけ実行されるまでトレースを行なった後で止まる。

Step to Next Spied Goal :: sg [COUNT]

次のスパイされているゴールが実行されるまでトレースを行い止まる。COUNT が指定された場合にはその回数だけ実行されるまでトレースを行なった後で止まる。

Step to Next Spied Procedure or Spied Goal :: ss [COUNT]

次のスパイされている述語コードが実行されるか、次のスパイされているゴールが実行されるまでトレースを行い止まる。COUNT が指定された場合にはその回数だけ実行されるまでトレースを行なった後で止まる。

Step to Next Spied Procedure and Spied Goal :: SS [COUNT]

次の述語コードとゴールが共にスパイされているものが実行されるまでトレースを行い止まる。COUNT が指定された場合にはその回数だけ実行されるまでトレースを行なった後で止まる。

Skip to Next Spied Procedure :: np [COUNT]

次のスパイされている述語コードが実行されるまでトレースしない。COUNT が指定された場合にはその回数だけ実行された後で止まる。

Skip to Next Spied Goal :: ng [COUNT]

次のスパイされているゴールが実行されるまでトレースしない。COUNT が指定された場合にはその回数だけ実行された後で止まる。

Skip to Next Spied Procedure or Spied Goal :: ns [COUNT]

次のスパイされている述語コードが実行されるか、次のスパイされているゴールが実行されるまでトレースしない。COUNT が指定された場合にはその回数だけ実行された後で止まる。

Skip to Next Spied Procedure and Spied Goal :: NS [COUNT]

次の述語コードとゴールが共にスパイされているものが実行されるまでトレースしない。COUNT が指定された場合にはその回数だけ実行された後で止まる。

Enqueue This Goal to Head of Ready Goal Queue :: <

指定した時点で表示しているゴールを強制的に、スケジューリング・キューの先頭にエンキューする。RESU と SWAP のトレース・ポイントで指定できる。

Enqueue This Goal to Tail of Ready Goal Queue :: >

指定した時点で表示しているゴールを強制的に、スケジューリング・キューの最後にエンキューする。CALL, Call, RESU と SWAP のトレース・ポイントで指定できる。

Depth First Schedule :: << [DEPTH]

ゴールのスケジューリングを Depth First に変更する。DEPTH は (execute 命令による)TRO による実行の深さ制限を指定する。デフォルトは 2^{31}。

Breadth First Schedule :: >> [DEPTH]

ゴールのスケジューリングを Breadth First に変更する。DEPTH は (execute 命令による)TRO による実行の深さ制限を指定する。デフォルトは 100。

Random Schedule :: >< [RATE [SEED [DEPTH]]]

ゴールのスケジューリングを Depth First を基本とし、乱数により一部ゴールをスケジューリング・キュー

の最後に回すことにより、マルチプロセッサの場合の非決定的な実行順序をシミュレートするモードに変更する。RATE はゴールをスケジューリング・キューの最後に回す割合でパーセントで指定する。SEED は擬似乱数を生成する場合の種を指定する。DEPTH は (execute 命令による)TRO による実行の深さ制限を指定する。

Re-Write Goal :: w LENGTH [DEPTH]

Print-Length, Print-Depth を一時的に変えてゴールを再表示する。

Where call from :: where

トレース中のゴールの呼びだし元ゴールのモジュール名及び述語名を表示。ランタイムサポートルーチン及び組込述語 (所謂 D コード) の場合のみ有効。

Monitor Variable :: m VARIABLE_NAME [NAME] [LIMIT]

変数が具体化された時にその値をモニタする。具体化された値がリスト (ストリーム) の場合は先頭の要素が決まる毎にその値を表示する。NAME を指定すると、モニターする変数に適当な名前を付けることができ、値の表示はその名前で表示する。LIMIT を指定すると、具体化された値を表示するときに LIMIT 回まで止まらずに表示する。LIMIT を指定しないと、具体化される度にその値を表示し、コマンドの入力待ちとなる。
値を表示する時、具体化された値がリストの場合とその他の場合では区別される。リストの場合には値として HEAD 部だけが表示される。

```
mon# 変数名 => 値 %%     …… リストの場合
mon# 変数名 == 値 %%     …… リスト以外の場合
```

この際に入力できるコマンドには以下のものがある。

```
?                          :: ヘルプ
x                          :: この変数 / リストのモニタリングを中止
s [COUNT]                  :: COUNT 回コマンド待ちにならないで進む
w LENGTH [DEPTH]           :: 表示した値の再表示
m VAR [NAME] [LIMIT]       :: 新たにモニタをセットする
```

Inspect Ready Queue :: ir [PRIORITY]

レディ・キュー内のゴールを表示する。PRIORITY が指定された場合にはその物理プライオリティ・キュー内のゴールだけを表示する。

Inspect Variable :: iv VARIABLE_NAME

指定した変数の状態を表示する。もし HOOK や MHOOK の時 (すなわちその変数の具体化を待っているゴールが既に存在している場合) にはその変数を待っているゴールを表示する。MGHOK の時 (すなわちマージャーへの入力となっている変数の場合) にはそのマージャーの出力側の変数が表示される。

Inspect Shoen tree :: is

指定した時点での荘園のツリー構造を表示する。横方向は親子関係、縦方向は兄弟関係。各荘園は 5 文字で表わされ、1 文字目は荘園の状態、2 ～ 5 文字は荘園の ID。荘園の状態を以下に示す。

R :: レディ状態
S :: 停止状態
A :: アボート状態

Trace Shoen tree :: ts

荘園のツリー構造のトレース・フラグを ON/OFF に切り替える。フラグが ON の場合、荘園の生成, アボート, ターミネートがあった時点で、その前後のツリーを表示する。ツリーの表示形式は上記と同じ。

Set Tracer Variable :: set NAME [VALUE]

NAME で指定されたトレーサーの変数に値をセットする。値が指定されなかった場合にはその変数の値を表示する。変数及びそのデフォルト値を以下に示す。

pv :: Print Variable Mode。 n か a で指定する。 n は Name-Mode (`A,B,C, ...`)、 a は Address Mode (`_NNN`) を意味する。

pl :: Print Length。整数で指定する。

pd :: Print Depth。整数で指定する。

g :: Gate Switch (トレース・ポイントの各状態でトレースを行うかどうかを決めるスイッチ)。 n,t,s から成る 5 文字で指定する。各文字は順に `CALL/Call, SUSP/Susp, RESU, SWAP, FAIL` の各 Gate Switch に対応する。 "n" は No-Trace, "t" は Trace (Not Stop), "s" は Trace (Stop) を意味する。

c :: Gate Switch – `CALL`。 n, t, s の何れかを指定する。

s :: Gate Switch – `SUSP`。 n, t, s の何れかを指定する。

r :: Gate Switch – `RESU`。 n, t, s の何れかを指定する。

w :: Gate Switch – `SWAP`。 n, t, s の何れかを指定する。

f :: Gate Switch – `FAIL`。 n, t, s の何れかを指定する。

7　デッドロック検出

PDSS におけるデッドロックの検出機能には 2 つのものがある。一方は、一括 GC の際に発見されるデッドロックであり、他方は、実行時に発見されるデッドロックである。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必ずそれが発見されるが、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。

以下では PDSS で発見されるデッドロックの種類とその場合のトレーサの出力例を示す。

一括 GC でデッドロックが発見された: Type=0

例 – 次のゴールを実行した場合。

```
Goal :: ?- add(X,_,Y), divide(Y,_,Z), modulo(Z,_,_).
```

コンソールウィンドウに表示される情報は次のとおり。

```
GC-1      KL1-Data    Srec    Grec   Prec   HeapTotal      Code
Used :        2594      20      75     10       30112         0
Deadlock::[0001]$$$SYSTEM:modulo(A~,B,C)
Deadlock::[0001]$$$SYSTEM:divide(D~,E,A~)
Deadlock::[0001]$$$SYSTEM:add(F~,G,D~)
*** Previous goal is deadlock root!
Shoen is terminated by deadlock!
After:        1083      14      57      3       15344         0
GCed :        1511       6      18      7       14768         0
              word     rec     rec    rec        byte      byte
GC Time: 40 msec
```

コンソールウィンドウに表示される “Previous goal is deadlock root!” は、直前に表示されているゴールがデッドロックしているゴール群のデータ待ちに関する依存関係を木として解析した場合に、その依存関係木の根になっていることを意味する。依存関係木が複数有る場合も存在し、必ずしも根は一つではない。また、ループ構造になっていれば根は存在しない。

自分だけしか参照していない変数 (所謂ボイド変数) の具体化を待とうとした: Type=10

例 – 以下のプログラムで p(X) が実行された場合。

```
Goal :: ?- module:go.
Clause-1 :: go   :- p(X).
Clause-2 :: p(a) :- true | true.
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock occurred. [suspend(UNDEFoo)]
*** Waiting for instantiation of a void variable.
[0001]module:p(A).
```

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: Type=11

例 – 以下のプログラムで p(X) の実行終了後 q(X) が実行された場合。

```
Goal :: ?- module:go.
Clause-1 :: go   :- true | p(X), q(X).
Clause-2 :: p(a) :- true | true.
Clause-3 :: q(a) :- true | true.
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock occurred. [suspend(HOOKoo)]
*** Waiting for instantiation of a variable which never be instantiated.
[0001]module:q(A^).
[0001]module:p(A^).
```

自分だけしか参照していないマージャーの入力変数の具体化を待とうとした: Type=12

例 – 以下のプログラムで merge(In,Out) の実行終了後 p(X) が実行された場合。

```
Goal :: ?- module:go.
Clause-1  :: go   :- true | merge(In, Out), p(In), ….
Clause-2  :: p(a) :- true | true.
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock occurred. [suspend(MGHOKo)]
*** Waiting for instantiation of a merger input variable.
[0001]module:p(A^).
[0001]merge(A^,B) in module:go/0
```

具体化を待っているゴールを有するような変数をボイド変数とユニファイしてしまった: Type=20

例 – 以下のプログラムで p(X) の実行終了後 q(X,Y) が実行された場合。Y が最初からボイド変数でなく、実行の結果としてボイド変数になった場合でも同じである。

```
Goal :: ?- module:go.
Clause-1 :: go :- p(X), q(X,Y).
Clause-2 :: p(a) :- true | true.
Clause-3 :: q(A,B) :- true | A=B.
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock occurred. [unify(HOOKoo,VOID)]
*** A variable which has a goal waiting for instantiation is unified with
*** a void variable.
*** Unification occurred in module:q/2
[0001]module:p(A^).
```

マージャーの入力変数をボイド変数とユニファイしてしまった: Type=21

例 – 以下のプログラムで merge(In,Out) の実行終了後 p(In,_) が実行された場合。

```
Goal :: ?- moduel:go.
Clause-1  :: go :- true | merge(In, Out), p(In,_), q(Out).
Clause-2  :: p(A,B) :- true | A=B.
Clause-3  :: q([_|Cdr]) :- true | q(Cdr).
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock will occur. [unify(MGHOKo,VOID)]
*** A merger input variable is unified with a void variable.
*** Unification occurred in module:p/2
[0001]merge(A^,B) in module:go/0
```

具体化を待っているゴールを有するような変数同士をユニファイしてしまった: Type=22

例 - 以下のプログラムで p(X) 及び q(X) が実行終了後 r(X,Y) が実行された場合。

```
Goal :: ?- module:go.
Clause-1 :: go :- true | p(X), q(Y), r(X,Y).
Clause-2 :: p(a) :- true | true.
Clause-3 :: q(a) :- true | true.
Clause-4 :: r(A,B) :- true | A=B.
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock occurred. [unify(HOOKoo,HOOKoo)]
*** Unifying two variables which have goals waiting for instantiation.
*** Unification occurred in module:r/2
[0001]module:p(A^).
[0001]module:q(B^).
```

具体化を待っているゴールを有するような変数とマージャーの入力変数をユニファイしてしまった: Type=23

例 - 以下のプログラムで merge(In,Out) 及び p(X) が実行終了後 q(X,In) が実行された場合。

```
Goal :: ?- module:go.
Clause-1 :: go   :- true | merge(In, Out), p(X), q(X, In), r(Out).
Clause-2 :: p(a) :- true | true.
Clause-3 :: q(A,B) :- true | A=B.
Clause-4 :: r([_|Cdr]) :- true | r(Cdr).
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock occurred. [unify(HOOKoo,MGHOKo)]
*** Unifying variable which has a goal waiting for instantiation is unified
*** and a merger input variable.
*** Unification occurred in module:q/2
[0001]module:p(A^).
[0001]merge(B^,C) in module:go/0
```

マージャーの入力変数同士をユニファイしてしまった: Type=24

例 - 以下のプログラムで merge(In1,Out1) 及び merge(In2,Out2) が実行終了後 In1=In2 が実行された場合。

```
Goal :: ?- module:go.
Clause-1 :: go :- true | merge(In1, Out1), merge(In2, Out2),
                         p(In1,In2), q(Out1), r(Out2).
Clause-2 :: p(A,B) :- true | A=B.
Clause-3 :: q([_|Cdr]) :- true | q(Cdr).
Clause-4 :: r([_|Cdr]) :- true | r(Cdr).
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock will occur. [unify(MGHOKo,MGHOKo)]
*** Unifying two merger input variables.
*** Unification occurred in module:p/2
[0001]merge(A^,B) in module:go/0
[0001]merge(C^,D) in module:go/0
```

具体化を待っているゴールを有するような変数を具体化しないまま他のどのゴールからも参照しなくなった: Type=30

例 - 以下のプログラムで p(X) が実行終了後 q(X) が実行された場合。

```
Goal :: ?- module:go.
Clause-1 :: go   :- true | p(X), q(X).
Clause-2 :: p(a) :- true | true.
Clause-3 :: q(_) :- true | true.
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock occurred. [collect(HOOKoo)]
*** A variable which has a goal waiting for instantiation was abandoned.
*** Collect_value occurred in module:q/1
[0001]module:p(A~).
```

マージャーの入力変数を具体化しないままどのゴールからも参照しなくなった: Type=31

例 - 以下のプログラムで merge(In,Out) が実行終了後 p(In) が実行された場合。

```
Goal :: ?- module:go.
Clause-1 :: go   :- true | merge(In, Out), p(In), q(Out).
Clause-2 :: p(_) :- true | true.
Clause-3 :: q([_|Cdr]) :- true | q(Cdr).
```

コンソールウィンドウに表示される情報は次のとおり。

```
*** Deadlock will occur. [collect(MGHOKo)]
*** A merger input variable was abandoned.
*** Collect_value occurred in module:p/1
[0001]merge(A^,B) in module:go/0
```

付録-1　入出力用のデバイス

PDSS では入出力用のデバイスとしてウィンドウ，ファイルとタイマを提供している。ユーザーはこのデバイスに繋がるストリームに規定のコマンドを流すことによりデバイスを利用することができる。これらのデバイスはモジュール pdss_window_device, pdss_file_device, pdss_timer_device で定義されている。

なお、この入出力用デバイスの仕様は "FEP・本体間 I/O インタフェース仕様書 (V0.9)" に準拠している。しかし、全ての機能を実現することは出来ないので、幾つかのメッセージはダミーであったり、受付けないものもある。

また、ここで使用している fep#xxxx というマクロ表現は、ライブラリ・マクロであり、これを使いたいモジュールでは

```
:- with_macro pdss.
```

という宣言を行う必要がある。

デバイス・ストリームの確保

デバイス・ストリームは次の述語により取り出すことができる。これらの述語はエミュレータが起動した後で1回だけ呼び出すことができ、2回目以降の呼出しは失敗する。

pdss_window_device:windows(Stream)
　ウィンドウ・デバイスに繋がるストリームを Stream とユニファイする。

pdss_file_device:files(Stream)
　ファイル・デバイスに繋がるストリームを Stream とユニファイする。

pdss_timer_device:timer(Stream)
　タイマ・デバイスに繋がるストリームを Stream とユニファイする。

デバイス・コマンド

1. ウィンドウ・デバイス
   ウィンドウ・デバイスは GNU-Emacs 上でマルチ・ウィンドウの機能を提供するデバイスである。このデバイスには次のコマンドを送ることができる。

   create(BufferName, WindowStream, ^Status, Cdr)
   　バッファ名が BufferName (8 ビットストリング) であるウィンドウをオープンし、そのウィンドウに繋がるストリームを WindowStream にユニファイする。オープンが成功した場合には Status に fep#normal がユニファイされる。PDSS では同時にオープンできるウィンドウの数は 16 個までであるので、それを超えた場合にはオープンは失敗し、Status に fep#abnormal がユニファイされる。
   　オープンしたウィンドウ・ストリームには後述する入出力コマンドや制御用コマンドを送ることができる (実際には、アボート・ラインとアテンション・ラインを張るための reset/4 コマンドを送った後でなければならない)。また、ストリームを閉じるとウィンドウは自動的にクローズされる。

   create(WindowStream, ^Status, Cdr)
   　バッファ名のない create/3 は使用できない。

   get_max_size(X, Y, PathName, ^Characters, ^Lines, ^Status, Cdr)
   　常に Characters=80, Lines=40, Status=fep#normal を返す。

2. ファイル・デバイス
   ファイル・デバイスは UNIX のファイル機能を提供するデバイスである。このデバイスには次のコマンドを送ることができる。

   open(PathName, Mode, FileStream, ^Status, Cdr)
   　パス名が PathName (8 ビットストリング) であるファイルをモード Mode (アトム: fep#read = リード・モード, fep#write = ライト・モード, fep#append = アペンド・モード) でオープンし、そのファ

イルに繋がるストリームを FileStream にユニファイする。オープンが成功した場合には Status に fep#normal がユニファイされる。なんらかの原因でファイルがオープンできない場合には fep#abnormal がユニファイされる。オープンしたファイル・ストリームには後述する入出力コマンドや制御用コマンドを送ることができる。(ファイルの場合も reset/4 を送っておく必要がある。) また、ストリームを閉じるとファイルは自動的にクローズされる。

directory(PathName, DirectoryStream, ^Status, Cdr)

パス名が PathName (8 ビットストリング) であるディレクトリをオープンし、そのディレクトリに繋がるストリームを DirectoryStream にユニファイする。オープンが成功した場合には Status に fep#normal がユニファイされる。なんらかの原因でオープンできない場合には fep#abnormal がユニファイされる。オープンしたディレクトリ・ストリームには後述するコマンドを送ることができる (ディレクトリの場合も reset/4 を送っておく必要がある)。また、ストリームを閉じるとディレクトリは自動的にクローズされる。

3. タイマ・デバイス

タイマ・デバイスはタイマ機能を提供するデバイスである。このデバイスには次のコマンドを送ることができる。時間の単位はミリ秒であるが、実際にカウントされるのは秒単位である (切り上げされる)。

get_count(^Count, ^Status, Cdr)

午前 0 時 0 分 0 秒から現在までのミリ秒数を Count にユニファイする。Status には fep#normal がユニファイされる。

on_at(Count, ^Now, ^Status, Cdr)

Count で指定された時刻になると、Now に fep#wake_up をユニファイする。Status にはコマンドを受け取った時点で fep#normal をユニファイする。

on_after(Count, ^Now, ^Status, Cdr)

Count で指定された時間が経過すると、Now に fep#wake_up にユニファイする。Status にはコマンドを受け取った時点で fep#normal をユニファイする。

ウィンドウ, ファイル, ディレクトリのコマンド

1. ウィンドウ, ファイル共通の制御用コマンド

これはウィンドウとファイルに共通するコマンドである。

reset(AbortLine, ^AttentionLine, ^Status, Cdr)

アボート・ラインとアテンション・ラインを張る。このコマンドは I/O ストリームが生成された直後に出されなければならない。 AbortLine には I/O 要求をアボートしたい時に、本体側から fep#abort をユニファイする。これが一旦ユニファイされると、再度の reset/4 コマンドでアボート・ラインとアテンション・ラインを張りなおすか、ストリームを □ で閉じるかのどちらかでなければならない。AttentionLine にはデバイス側から割込みコード (整数) がユニファイされる。この場合には I/O をアボートするか、next_attention/3 コマンドによりアテンション・ラインを張りなおさなければならない。

next_attention(^Attention, ^Status, Cdr)

アテンション・ラインのみを張りなおす。アテンション入力はあったがアボートはしたくない時に使われる。

2. 共通の入力コマンド

これはウィンドウおよびリード・モードでオープンしたファイルに対して送ることができるコマンドである。

getc(^Char, ^Status, Cdr)

1 文字を読み込みその文字コードを Char にユニファイする。読み込みが成功した場合には Status には fep#normal がユニファイされる。 もしエンド・オブ・ファイルだった場合には fep#end_of_file がユニファイされる。

【注意】Multi-PSI V2 FEP ではサポートされない。

getl(^Line, ^Status, Cdr)

1行を読み込みそれを 8 ビットストリングにして Line にユニファイする。この時改行コードは取り除かれる。読み込みが成功した場合には Status には `fep#normal` がユニファイされる。もしエンド・オブ・ファイルだった場合には `fep#end_of_file` がユニファイされる。

【注意】Multi-PSI V2 FEP のファイルではサポートされない。

getb(Size, ^Buffer, ^Status, Cdr)

Size (整数) で指定されたバイト数だけ読み込み 8 ビットストリングにして Buffer にユニファイする。ウィンドウからの入力で途中で改行になった場合には改行までを入力とする。読み込みが成功した場合には Status には `fep#normal` がユニファイされる。もしエンド・オブ・ファイルだった場合には `fep#end_of_file` がユニファイされる。

【注意】Multi-PSI V2 FEP のウィンドウではサポートされない。

gettkn(^TokenList, ^Status, ^NumberOfVariables, Cdr)

ターム 1 個を構成しうる文字列を読み込み、その文字列のトークン解析を行い、生成したトークンのリストを TokenList にユニファイする。また、トークン・リストの中の変数の種類数を NumberOfVariables にユニファイする。トークンの形式を以下に示す。

変数	:: $VAR(N,String)
アトム	:: atom(Atom)
整数	:: integer(Integer)
浮動小数点数	:: float(Float)
ストリング	:: string(String)
ファンクタ	:: open(Atom)
符号	:: sign(Atom)
特殊文字	:: 特殊文字を印字名とするアトム
異常データ	:: illegal(String)
終端	:: end

読み込みが成功した場合には Status には `fep#normal` がユニファイされる。もしエンド・オブ・ファイルだった場合には `fep#end_of_file` がユニファイされる。またトークン解析でエラーがある場合には `fep#abnormal` がユニファイされる。

【注意】Multi-PSI V2 FEP ではサポートされない。.

3. 共通の出力コマンド

これはウィンドウおよびライト・モードまたはアペンド・モードでオープンしたファイルに対して送ることができるコマンドである。

putc(Char, ^Status, Cdr)

Char (整数) で示される (ASCII) コードの 1 文字を書き出す。Status には `fep#normal` がユニファイされる。

【注意】Multi-PSI V2 FEP ではサポートされない。

putl(Line, ^Status, Cdr)

Line (8 ビットストリング) で示されるストリングを書出し、改行する。Status には `fep#normal` がユニファイされる。

【注意】Multi-PSI V2 FEP ではサポートされない。

putb(Buffer, ^Status, Cdr)

Buffer (8 ビットストリング) で示されるストリングを書き出す。Status には `fep#normal` がユニファイされる。

putt(Term, Length, Depth, ^Status, Cdr)

Term で示されるタームを構造体の長さが Length 以内、深さが Depth 以内の範囲で書き出す。Length

および Depth を超えた部分は ... が出力される。Status には fep#normal がユニファイされる。このコマンドはデバッグ用出力関数を流用しているので、Term に含まれる変数は A,B,C のように出力される。また、MRB や HOOK の記号が付加される。

【注意】Multi-PSI V2 FEP ではサポートされない。

4. ウィンドウ用制御コマンド

これはウィンドウに対してだけ送ることができるコマンドである。

close(^Status)

ウィンドウを閉じる。Status には fep#normal がユニファイされる。

flush(^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。出力したデータは flush/2 を実行しなくても自動的に flush される。

beep(^Status, Cdr)

ベルを鳴らす。Status には fep#normal がユニファイされる。

clear(^Status, Cdr)

ウィンドウに表示されている内容を消す。Status には fep#normal がユニファイされる。

show(^Status, Cdr)

ウィンドウを見える状態にする。Status には fep#normal がユニファイされる。作られたばかりのウィンドウは見えない状態になっているのでこのコマンドにより見えるようにする必要がある。

hide(^Status, Cdr)

ウィンドウを見えない状態にする。Status には fep#normal がユニファイされる。

activate(^Status, Cdr)

show/2 と同じ。

deactivate(^Status, Cdr)

hide/2 と同じ。

set_inside_size(Characters, Lines, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

set_size(fep#manipulator, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

set_position(X, Y, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

set_position(fep#manipulator, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

set_title(String, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

reshape(X, Y, Characters, Lines, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

reshape(fep#manipulator, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

set_font(PathName, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

select_buffer(BufferName, ^Status, Cdr)

使用できない。

get_inside_size(^Characters, ^Lines, ^Status, Cdr)

常に Characters=80, Lines=20, Status=fep#normal を返す。

get_position(^X, ^Y, ^Status, Cdr)

常に X=0, Y=0, Status=fep#normal を返す。

get_title(^Title, ^Status, Cdr)

ウィンドウを生成した時に指定したバッファ名を返す。

get_font(PathName, ^Status, Cdr)

使用できない。

5. ファイル用制御コマンド

これはファイルに対してだけ送ることができるコマンドである。

close(^Status)

ファイルをクローズする。Status には fep#normal がユニファイされる。

end_of_file(^Status, Cdr)

ファイルがエンド・オブ・ファイルの状態の時は Status に fep#yes をユニファイする。そうでない時には fep#no をユニファイする。

pathname(^PathName, ^Status, Cdr)

ファイルのパス名を PathName に返す。Status には fep#normal がユニファイされる。

6. ディレクトリ用制御コマンド

これはディレクトリに対してだけ送ることができるコマンドである。

pathname(^PathName, ^Status, Cdr)

ディレクトリのパス名を PathName に返す。Status には fep#normal がユニファイされる。

listing(WildCard, FileNameStream, ^Status, Cdr)

WildCard (8 ビットストリング) で指定されるファイルのパス名のリストを取り出すストリームを FileNameStream に返す。Status には fep#normal がユニファイされる。FileNameStream には next_file_name(^FileNa ^Status, Cdr) というコマンドを流すことができ、FileName に 1 つのファイル名 (8 ビットストリング) が返され、Status に fep#normal がユニファイされる。もうファイルが無い場合には Status に fep#end_of_file がユニファイされる。

delete(WildCard, ^Status, Cdr)

WildCard (8 ビットストリング) で指定されるファイルを全て削除する。PDSS では一旦消したファイルを回復することはできない。Status には fep#normal がユニファイされる。

undelete(WildCard, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

purge(WildCard, ^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

deleted(WildCard, ^FileNameStream, ^Status, Cdr)

WildCard (8 ビットストリング) で指定されるファイルのうち、削除中のファイルのパス名のリストを取り出すストリームを FileNameStream に返す。ただし、このリストは必ず空である。Status には fep#normal がユニファイされる。

expunge(^Status, Cdr)

なにもしない。Status には fep#normal がユニファイされる。

付録-2　コード・デバイス

これはコードを管理しているデバイスであり、このデバイス・ストリームに規定のコマンドを流すことによりコードに関する操作を行うことができる。(現時点ではコード・デバイス・ストリームは Micro PIMOS でのみ使用しておりユーザに提供していない。)

assemble_module(^ModuleName, FileName, ^Status)

FileName (8 ビットストリング) で指定されたファイルをアセンブルし、コード領域に置く。ModuleName にはアセンブルされたモジュール名のアトムをユニファイする。Status には 'success', 'cannot_open_file', 'memory_limit', 'module_protected', 'load_error' のいずれかがユニファイされる。

load_module(^ModuleName, FileName, ^Status)

FileName (8 ビットストリング) で指定されたファイルをコード領域にロードする。ファイルの形式はセーブ形式かアセンブラ形式。ModuleName にはロードされたモジュール名のアトムをユニファイする。Status には 'success', 'cannot_open_file', 'memory_limit', 'module_protected', 'load_error' のいずれかがユニファイされる。

save_module(ModuleName, FileName, ^Status)

FileName (8 ビットストリング) で指定されたファイルに ModuleName (アトム) で指定されたモジュールをセーブする。Status には 'success', 'cannot_open_file', 'module_not_found' のいずれかがユニファイされる。

remove_module(ModuleName, ^Status)

ModuleName (アトム) で示されたモジュールを削除する。Status には 'success', 'module_not_found' , 'module_protected' のいずれかがユニファイされる。

debug(Flag, ^Status)

デバッグ・モードの ON/OFF を行う。Flag はアトムで 'on' または 'off' を与える。Status には 'success' または 'undefined_mode' がユニファイされる。

backtrace(Flag, ^Status)

バックトレース (一括型 GC で検出されたデッドロック・ゴールの表示) の ON/OFF を行う。Flag はアトムで 'on' または 'off' を与える。Status には 'success' または 'undefined_mode' がユニファイされる。

trace_module(ModuleName, Mode, ^Status)

ModuleName (アトム) で示されたモジュールのトレース・モードを Mode に変更する。Mode はアトムで 'on' または 'off' を与える。Status には 'success', 'module_not_found', 'undefined_mode', 'native_code_module' のいずれかがユニファイされる。

get_module_status(ModuleName, ^Mode, ^Status)

ModuleName (アトム) で示されたモジュールのトレース・モードを調べる。Mode にはトレース・モードの ON/OFF により 'on' または 'off' がユニファイされる。Status には 'success', 'module_not_found', 'native_code_module' のいずれかがユニファイされる。

spy_predicate(ModuleName, PredicateName, Arity, Mode, ^Status)

ModuleName (アトム) で示されたモジュール内の PredicateName/Arity で示された述語のトレース・モードを Mode に変更する。Mode はアトムで 'on' または 'off' を与える。Status には 'success', 'module_not_found', 'predicate_not_found', 'undefined_mode', 'native_code_module' のいずれかがユニファイされる。

get_spied_predicates(ModuleName, ^Predicates, ^Status)

ModuleName (アトム) で示されたモジュールでスパイされている述語の情報をリストの形にして Predicates にユニファイする。各要素は 2 要素ベクタで { 述語名アトム, アリティ } の形式。Status には 'success', 'module_not_found' のいずれかがユニファイされる。

get_public_predicates(ModuleName, ^Public, ^Status)

ModuleName (アトム) で示されたモジュールで public 宣言されている述語の情報をリストの形にして Pub-

lic にユニファイする。各要素は 2 要素ベクタで { 述語名アトム, アリティ } の形式。Status には 'success', 'module_not_found' のいずれかがユニファイされる。

付録 -3　PIMOS 共通ユーティリティ

ここで説明するユーティリティ・プログラムは PIMOS の為に開発されたものであるが、PDSS 上でも使用することができる。これらのユーティリティは提供されるモジュールを呼び出すことにより、Micro PIMOS のオート・ロード機能により自動的にロードされ、使用することができる。

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ユーティリティとして次のような変換, 格納機能を提供する。ユーザーはこれらのユーティリティを利用したい場合、PIMOS が提供するモジュールの各述語を呼び出すことにより、変換結果, またはオブジェクトにつながるストリームを得ることができる。ユーザーはこのストリームにメッセージを送ることにより各オブジェクトを操作する。なお、このストリームにはマージャーが挿入されている。

- 比較: どんな KL1 データでも一意に比較できる機能
- ハッシング: 標準のハッシュ関数
- キーなしの P ブール: Bag, Stack, Queue, Sorted Bag
- キー付きのブール: Keyed Bag, Keyed Set, Keyed Sorted Bag, Keyed Sorted Set

1. 比較

一般にどんな KL1 データでも一意に比較できる機構を提供する。

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。両者が等しい場合には 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：

両者のタイプが異なる場合、大小関係はタイプ間の大小関係として整数, アトム, 文字列, リスト, ベクタの順に定義される。両者が同じタイプなら大小関係は以下のように定義する。

- 整数 …… 通常の符号つき整数の大小関係。
- アトム …… アトム番号の大小関係。
- 文字列 …… 通常の辞書順の大小関係。ただし、文字列のタイプが異なる場合にはタイプの大きさの大小関係。
- 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 …… 要素数の大小関係。要素数が等しければ第 1 要素、それも等しければ第 2 要素 … 等々の大小関係。

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 をタイプによって以下のようなハッシュ関数でハッシュして、その値を H に返す。また、Y に X をそのまま返す。H には必ず非負の整数が値として返される。

ハッシュ関数の定義：

- 整数 …… 絶対値。
- アトム …… アトム番号。
- 文字列 …… Cb × 最初の要素 + Cm × 中央要素 + Ce × 最後の要素 + 要素数。ただし、係数 Cb, Cm, Ce は KL0 の組込述語と同様。
- リスト …… Car のハッシュ値 + 5 × Cdr のハッシュ値。
- ベクタ …… 最初, 中央, 最後の各要素に関する ((2 の要素番号乗 + 1) × 要素のハッシュ値) の和 + 要素数。

3. キーなしのプール

一般にどんな KL1 データでも格納しておける仕組みを提供する。

Bag

基本的なプール。入れる, 出すという基本機能しかない。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。

pool:bag(Stream)

Bag のオブジェクトを生成し、そこへのストリーム Stream を得る。

メッセージプロトコル：

empty(^YorN)

空ならば yes 、そうでなければ no を返す。

put(X)

要素の追加。

get(^X)

要素の取り出し。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、その要素は Bag からなくなる。要素がないのに取り出そうとするとフェイルする。

get_all(^O)

中身を全部取り出す。取り出される形式は要素のリストである。空の場合には □ を返す。

get_and_put(^X, Y)

要素を取りだし、同じ位置に新しい要素を格納する。要素がないとフェイルする。

Stack

基本的には Bag と同じだが、取り出し順が LIFO になっている。

pool:stack(Stream)

Stack のオブジェクトを生成し、そこへのストリーム Stream を得る。

メッセージプロトコル： Bag と同様。

Queue

基本的には Bag と同じだが、取り出し順が FIFO になっている。

pool:queue(Stream)

Queue のオブジェクトを生成し、そこへのストリーム Stream を得る。

メッセージプロトコル： Bag と同様。

Sorted Bag

基本的には Bag と同じだが、取り出し順が“小さい順”になっている。比較ルーチンが stable なら、(弱い意味で) 小さい順に入れればその順を保存して取り出される。

pool:sorted_bag(Stream)

標準の比較ルーチン (comparator:sort/5) 付きで Sorted Bag のオブジェクトを生成し、そこへのストリーム Stream を取り出す。

pool:sorted_bag(Comparator, Stream)

第1引数に比較ルーチンを { モジュール名アトム, 述語名アトム, アリティ } の形式で指定することにより、その比較ルーチン付きで Sorted Bag を生成し、そこへのストリーム Stream を取り出す。ここで、指定された述語は comparator:sort/5 と同じ引数個数, 内容で, public 宣言されている必要がある。

メッセージプロトコル：

Bag と同様。ただし get はその時点での最小のものを返し、get_all は小さい順に返す。

4. キー付きのプール

任意の KL1 データをキー付きで格納しておける仕組みを提供する。

Keyed Bag

基本的なキー付きプール。ハッシュテーブルにより実現している。

pool:keyed_bag(Stream)

標準のハッシュ関数 (hasher:hash/3) 付きで Keyed Bag のオブジェクトを生成し、そこへのストリーム Stream を取り出す。ハッシュテーブルのサイズの初期値は 1 である。

pool:keyed_bag(Stream, Size)

第2引数にハッシュテーブルのサイズの初期値を指定することにより、標準のハッシュ関数 (hasher:hash/3), 指定されたテーブルサイズで Keyed Bag のオブジェクトを生成し、そこへのストリーム Stream を取り出す。

pool:keyed_bag(Hasher, Stream, Size)

第1引数にハッシュ関数を { モジュール名アトム, 述語名アトム, アリティ } の形式で指定し、第2引数にハッシュテーブルのサイズの初期値を指定することにより、指定されたハッシュ関数, テーブルサイズで Keyed Bag を生成し、そこへのストリーム Stream を取り出す。ここで、指定された述語は hasher:hash/3 と同じ引数個数, 内容で、public 宣言されている必要がある。

メッセージプロトコル：

empty(^YorN)

空ならば yes 、そうでなければ no を返す。

empty(Key, ^YorN)

キーが Key と等しい要素があれば no、そうでなければ yes を返す。

put(Key, X)

キーが Key, 値が X の要素の追加。

get(Key, ^X)

キーが Key の要素の取り出し。同じキーのものが複数ある場合にはどれが取れてくるかは決めない。取り出すとプールからなくなる。要素がないのに取り出そうとするとフェイルする。

get_all(^O)

中身を全部取り出す。取り出されるものの形式は {Key, Data} のリストである。空の場合には □ を返す。

get_all(Key, ^O)

キーが Key の要素を全部取り出す。取り出されるものの形式は Data のリストである。対応するものがなければ □ を返す。

get_and_put(Key, ^X, Y)

キーが Key の要素を取りだし、同じ位置に新しい要素を格納する。要素がないとフェイルする。

Keyed Set

重複を許さないキー付きのプール。

pool:keyed_set(Stream)

標準のハッシュ関数 (hasher:hash/3) 付きで Keyed Set のオブジェクトを生成し、そこへのストリーム Stream を取り出す。ハッシュテーブルのサイズの初期値は 1 である。

pool:keyed_set(Stream, Size)

第 2 引数にハッシュテーブルのサイズの初期値を指定することにより、標準のハッシュ関数 (hasher:hash/3), 指定されたテーブルサイズで Keyed Set のオブジェクトを生成し、そこへのストリーム Stream を取り出す。

pool:keyed_set(Hasher, Stream, Size)

第 1 引数にハッシュ関数を { モジュール名アトム, 述語名アトム, アリティ } の形式で指定し、第 2 引数にハッシュテーブルのサイズの初期値を指定することにより、指定されたハッシュ関数, テーブルサイズで Keyed Set を生成し、そこへのストリーム Stream を取り出す。ここで、指定された述語は hasher:hash/3 と同じ引数個数, 内容で、public 宣言されている必要がある。

メッセージプロトコル：

empty(^YorN)

空ならば yes、そうでなければ no を返す。

empty(Key, ^YorN)

キーが Key と等しい要素があれば no、そうでなければ yes を返す。

put(Key, X, ^OldX)

キーが Key, 値が X の要素の追加。キーが Key のものが既にあれば更新され, OldX に前の値が { 値 } の形で返される。キーが Key のものがなければ OldX には {} が返される。

get(Key, ^X)

キーが Key の要素の取り出し。取り出すとプールからなくなる。要素がないのに取り出そうとするとフェイルする。

get_all(^O)

中身を全部取り出す。取り出されるものの形式は {Key, Data} のリストである。空の場合は □ を返す。

get_all(Key, ^O)

キーが Key の要素を全部取り出す。対応するものがなければ □ 、あれば 1 要素のリストが返る。

get_and_put(Key, ^X, Y)

キーが Key の要素を取りだし、同じ位置に新しい要素を格納する。要素がないとフェイルする。

Keyed Sorted Bag

Sorted bag と同様だが、ソートをキーだけに対して行う点が異なる。

pool:keyed_sorted_bag(Stream)

標準の比較ルーチン (comparator:sort/5) 付きで Keyed Sorted Bag のオブジェクトを生成し、そこへのストリーム Stream をユニファイする。

pool:keyed_sorted_bag(Comparator, Stream)

第 1 引数に比較ルーチンを { モジュール名アトム, 述語名アトム, アリティ } の形式で指定することにより、その比較ルーチン付きで Keyed Sorted Bag を生成する。ここで、指定された述語は comparator:sort/5 と同じ引数個数, 内容で、public 宣言されている必要がある。

メッセージプロトコル：

Keyed Bag と同様だが、取り出し時にキーの小さい順に取り出すことを決めている点が異なる。比較が stable で同じキーのものが複数ある場合は入れた順に取り出す。

Keyed Sorted Set

Keyed Sorted Bag と同様だが、キーの重複を許さない点が異なる。

pool:keyed_sorted_set(Stream)

標準の比較ルーチン (comparator:sort/5) 付きで Keyed Sorted Set のオブジェクトを生成し、そこへのストリーム Stream を取り出す。

pool:keyed_sorted_set(Comparator, Stream)

第1引数に比較ルーチンを { モジュール名アトム, 述語名アトム, アリティ } の形式で指定することにより、その比較ルーチン付きで Keyed Sorted Set を生成する。ここで、指定された述語は comparator:sort/5 と同じ引数個数, 内容で、 public 宣言されている必要がある。

メッセージプロトコル：

Keyed Set と同様。ただし、 get_all/1 はキーの小さい順に返す。

付録-4　PDSS で使用しているモジュール名

以下のモジュール名は PDSS 側で使われているので、同じ名前をユーザーが使うことはできません。(* 印の付いているものは可能。)

```
  'Sho-en'
* directory
* file
* window
* mpimos_io_device
  mpimos_monogyny_list_index
  mpimos_booter
  mpimos_builtin_predicate
  mpimos_cmd_basic
  mpimos_cmd_code
  mpimos_cmd_debug
  mpimos_cmd_directory
  mpimos_cmd_environment
  mpimos_cmd_utl
  mpimos_code_manager
  mpimos_command_interpreter
  mpimos_directory
  mpimos_directory_device_driver
  mpimos_file
  mpimos_file_device_driver
  mpimos_file_manager
  mpimos_window_device
  mpimos_file_device
  mpimos_timer_device
  mpimos_macro_expander
  mpimos_module_pool
  mpimos_opcode_table
  mpimos_operator_manipulator
  mpimos_parser
  mpimos_task_monitor
  mpimos_unparser
  mpimos_utility
* mpimos_varchk
  mpimos_window
  mpimos_window_device_driver
  mpimos_window_manager
* mpimos_xref
* mpimos_xref_table
* mpimos_pretty_printer
  pdss_code_device
  pdss_window_device
  pdss_file_device
  pdss_timer_device
  klicmp_blttbl
  klicmp_command
  klicmp_compile
  klicmp_mrb
  klicmp_normalize
  klicmp_output
  klicmp_reader
  klicmp_register
  klicmp_macro
  klicmp_macro_arg
  klicmp_mtbl
  klicmp_struct
```

付録 -5　定義済みオペレーター覧

Micro PIMOS のウィンドウ / ファイルでは予め次のオペレータが定義されている。

```
1200   xfx   :-
1200    fx   :-
1200   xfx   -->
1150    fx   module
1150    fx   public
1150    fx   implicit
1150    fx   local_implicit
1150    fx   with_macro
1100   xfy   ;
1100   xfy   |
1090   xfx   =>
1050   xfy   ->
1000   xfy   ,
 800   xfx   :
 700   xfx   =
 700   xfx   \=
 700   xfx   ==
 700   xfx   =:=
 700   xfx   $=:=
 700   xfx   =\=
 700   xfx   $=\=
 700   xfx   <
 700   xfx   $<
 700   xfx   >
 700   xfx   $>
 700   xfx   =<
 700   xfx   $=<
 700   xfx   >=
 700   xfx   $>=
 700   xfx   :=
 700   xfx   $:=
 700   xfx   <=
 700   xfx   $<=
 700   xfx   <<=
 700   xfy   @
 500   yfx   +
 500    fx   +
 500   yfx   -
 500    fx   -
 500   yfx   /\
 500   yfx   \/
 500   yfx   xor
 400   yfx   *
 400   yfx   /
 400   yfx   <<
 400   yfx   >>
 300   xfy   **
 300   xfx   mod
 200    fx   &
 150   xf    ++
 150   xf    --
 100   xfx   #
 100    fx   #
```

付録-6　組込述語一覧

1. タイプのチェック

```
wait(X) :: G
atom(X) :: G
integer(X) :: G
floating_point(X) :: G
list(X) :: G
vector(X) :: G
string(X) :: G
unbound(X, ^PE, ^Addr, ^NewX) :: B
```

2. diff

```
diff(X, Y) :: G
```

次のオペレータで記述してもよい:　\=

3. 整数の比較

```
equal(Integer1, Integer2) :: G
not_equal(Integer1, Integer2) :: G
less_than(Integer1, Integer2) :: G
not_less_than(Integer1, Integer2) :: G
```

次のオペレータで記述してもよい:　=:=, =\=, <, =<, >, >=

4. 整数の演算

```
add(Integer1, Integer2, ^NewInteger) :: GB
subtract(Integer1, Integer2, ^NewInteger) :: GB
multiply(Integer1, Integer2, ^NewInteger) :: GB
divide(Integer1, Integer2, ^NewInteger) :: GB
modulo(Integer1, Integer2, ^NewInteger) :: GB
minus(Integer, ^NewInteger) :: GB
increment(Integer, ^NewInteger) :: GB
decrement(Integer, ^NewInteger) :: GB
abs(Integer, ^NewInteger) :: GB
min(Integer1, Integer2, ^NewInteger) :: GB
max(Integer1, Integer2, ^NewInteger) :: GB
and(Integer1, Integer2, ^NewInteger) :: GB
or(Integer1, Integer2, ^NewInteger) :: GB
exclusive_or(Integer1, Integer2, ^NewInteger) :: GB
complement(Integer, ^NewInteger) :: GB
shift_left(Integer, ShiftWidth, ^NewInteger) :: GB
shift_right(Integer, ShiftWidth, ^NewInteger) :: GB
```

:=, <= と次のオペレータを用いて記述してもよい:

+, -, *, /, mod, /\, \/, xor, <<, >>

5. 浮動小数点数の比較

```
floating_point_equal(Float1, Float2) :: G
floating_point_not_equal(Float1, Float2) :: G
floating_point_less_than(Float1, Float2) :: G
floating_point_not_less_than(Float1, Float2) :: G
```

次のオペレータで記述してもよい: $=:=, $=\=, $<, $=<, $>, $>=

6. 浮動小数点数の演算

```
floating_point_add(Float1, Float2, ^NewFloat) :: GB
floating_point_subtract(Float1, Float2, ^NewFloat) :: GB
floating_point_multiply(Float1, Float2, ^NewFloat) :: GB
floating_point_divide(Float1, Float2, ^NewFloat) :: GB
floating_point_minus(Float, ^NewFloat) :: GB
floating_point_abs(Float, ^NewFloat) :: GB
floating_point_min(Float1, Float2, ^NewFloat) :: GB
floating_point_max(Float1, Float2, ^NewFloat) :: GB
floating_point_floor(Float, ^NewFloat) :: GB
floating_point_sqrt(Float, ^NewFloat) :: GB
floating_point_ln(Float, ^NewFloat) :: GB
floating_point_log(Float, ^NewFloat) :: GB
floating_point_exp(Float, ^NewFloat) :: GB
floating_point_pow(Float1, Float2, ^NewFloat) :: GB
floating_point_sin(Float, ^NewFloat) :: GB
floating_point_cos(Float, ^NewFloat) :: GB
floating_point_tan(Float, ^NewFloat) :: GB
floating_point_asin(Float, ^NewFloat) :: GB
floating_point_acos(Float, ^NewFloat) :: GB
floating_point_atan(Float, ^NewFloat) :: GB
floating_point_atan(Float1, Float2, ^NewFloat) :: GB
floating_point_sinh(Float, ^NewFloat) :: GB
floating_point_cosh(Float, ^NewFloat) :: GB
floating_point_tanh(Float, ^NewFloat) :: GB
```

$:=, $<= と次のオペレータを用いて記述してもよい。

+, -, *, /, **

7. 整数-浮動小数点数の変換

```
floating_point_to_integer(Float, ^Integer) :: GB
integer_to_floating_point(Integer, ^Float) :: GB
```

8. ベクタ関係

```
vector(X, ^Size) :: G
vector(X, ^Size, ^NewVector) :: B
new_vector(^Vector, Size) :: B
vector_element(Vector, Position, ^Element) :: G
vector_element(Vector, Position, ^Element, ^NewVector) :: B
set_vector_element(Vector, Position, ^OldElem, NewElem, ^NewVect) :: B
```

9. ストリング関係

```
string(X, ^Size, ^ElementSize) :: G
string(X, ^Size, ^ElementSize, ^NewString) :: B
new_string(^String, Size, ElementSize) :: B
string_element(String, Position, ^Element) :: G
string_element(String, Position, ^Element, ^NewString) :: B
set_string_element(String, Position, NewElement, ^NewString) :: B
substring(String, Position, Length, ^SubString, ^NewString) :: B
set_substring(String, Position, SubString, ^NewString) :: B
append_string(String1, String2, ^NewString) :: B
```

10. アトム関係

```
intern_atom(^Atom, String) :: B
new_atom(^Atom) :: B
atom_name(Atom, ^String) :: B
atom_number(Atom, ^Number) :: B
```

11. コード関係

```
predicate_to_code(Mod, Pred, Arity, ^Code) :: B
code_to_predicate(Code, ^Mod, ^Pred, ^Arity, ^Info) :: B
```

12. ストリーム・サポート

```
merge(In, ^Out) :: B
```

13. 高階機能

```
apply(Code, Args) :: B
```

14. 特殊入出力

```
read_console(^Integer) :: G
display_console(X) :: G
put_console(X) :: G
```

15. その他

```
raise(Tag, Type, Info) :: B
consume_resource(Red) :: B
```

```
hash(X, ^Value, ^NewX) :: B
current_processor(^ProcessorNumber, ^X, ^Y) :: B
current_priority(^CurrentPriority, ^ShoenMin, ^ShoenMax) :: B
```

付録 -7　例外コード

- Illegal Input Type :: 0
  組込述語の入力引数に規定されている型以外のものが現われた。
- Range Overflow :: 1
  組込述語 (数値演算以外) の入力引数に規定されている範囲以外のものが現われた。
- Integer Overflow :: 3
  整数演算を行った結果、オーバーフローが発生した。0 による整数除算もここに含まれる。
- Floating Point Error :: 5
  浮動小数点演算の入力引数に規定されている範囲以外のものが現われた。また、演算を行った結果、オーバーフローが発生した。
- Illegal Merger Input :: 8
  マージャーの入力に対して □, リスト, ベクタ以外のデータをユニファイしようとした。
- Reduction Failure :: 9
  ゴールの実行に際してどの候補節も選ばれなかった。
- Unification Failure :: 10
  ボディ部のユニフィケーションが失敗した。
- Raised :: 12
  組込述語 raise/3 が実行された。
- Incorrect Priority :: 16
  指定されたプライオリティが規定されている範囲にない。
- Module Not Found :: 17
  ロードされていないモジュールを参照しようとした。
- Predicate Not Found :: 18
  指定されたモジュール内に指定された述語が定義されていない。
- Deadlock :: 11
  荘園内でデッドロック状態が発見された。

付録 -8　PDSS で使用している荘園のタグ

荘園のタグは現在以下のビットが言語と Micro PIMOS で既に使用されている。

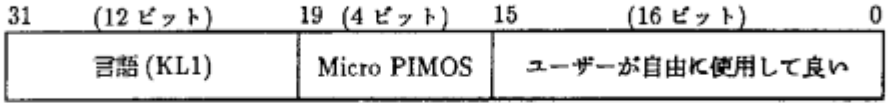

図 8: 荘園の例外タグ

- ビット 16 — 親荘園への I/O ストリームの要求。
- ビット 17 — 親荘園へのエラー情報の伝達。
- ビット 18 — Micro PIMOS の Shell ウィンドウへのメッセージ出力。
- ビット 19 — 未使用。
- ビット 20 — Deadlock。
- ビット 21 — Illegal Input Type。
- ビット 22 — Range Overflow。
- ビット 23 — Integer Overflow。
- ビット 24 — Floating Point Error。
- ビット 25 — 未使用。
- ビット 26 — Illegal Merger Input。
- ビット 27 — Reduction Failure。
- ビット 28 — Unification Failure。
- ビット 29 — Incrrect Priority。
- ビット 30 — Module Not Found。
- ビット 31 — Predicate Not Found。

付録 -9　GNU-Emacs ライブラリ

PDSS で提供している GNU-Emacs ライブラリには、KL1 のプログラムを編集する場合の "kl1-mode" と PDSS の実行時に使われる "PDSS-mode" がある。以下ではそれぞれのモードで定義されているコマンドの一覧を示す。

1. kl1-mode

ctrl-C ctrl-C

コマンドを実行したバッファ内の全てのテキストを KL1 のプログラムとしてコンパイルする。

ctrl-C ctrl-R

PDSS=COMPILER なるバッファに指定された領域内のテキストをコピーする。

ctrl-C ctrl-D

PDSS=COMPILER なるバッファの内容を KL1 のプログラムとしてコンパイルする。この後、本コマンドを実行したバッファのファイル名から同一のファイル名を持つアセンブラ・ソースファイル (*.asm) を探し、更新された部分のみを変更し、*.sav ファイルを生成する。
これは ctrl-C ctrl-R コマンドと組み合わせて、プログラムの変更した部分のみを再コンパイルする場合に使用する。このためには *.asm ファイルは削除しない方が良い。

meta-X pdss-kl1cmp-switch-indexing-mode

meta-X pdss-kl1cmp-switch-mrbgc-mode

meta-X pdss-kl1cmp-switch-system-mode

Prolog 版コンパイラのオプションを変更する。引数なしで ON/OFF のトグル、引数 1/0 で ON/OFF となる。オプションの意味、初期値についてはコンパイル用コマンドプロシジャ (pdsscmp) の使用法の項を参照のこと。それぞれ i, m, s オプションに相当する。
KL1 版コンパイラを使用している場合には、この指定は不可能。
【注意】PDSS=COMPILE なるバッファに各種の情報やプロンプトが表示されるが、ユーザは基本的には何もこのバッファに入力する必要が無い。(出力専用)

ctrl-C ctrl-F

組込述語のマニュアルを表示する。

2. PDSS-mode

a. ウィンドウ / バッファ操作

meta-.

直前の文字列 (単語) を略語として、バッファ中で ?-　から始まり略語にマッチする文字列を候補として表示する。主にコマンド・インタプリタのコマンドの再実行に使う。

ctrl-C ctrl-Y

直前に入力した文字列を再表示する。

ctrl-C k

そのバッファ内のテキストを全て削除する。

ctrl-C ctrl-K

PDSS モードである全てのバッファのテキストを削除する。

ctrl-C ctrl-B

PDSS モードのバッファのバッファ・メニューを表示する。

ctrl-C m

このコマンドを実行した行の先頭から "モジュール名:述語名" のパターンを探しコマンドを入力した位置に挿入する。トレーサで変数のモニタをセットする場合に変数の名前を付ける為に用いると便利。

ctrl-C ctrl-F

組込述語のマニュアルを表示する。

ctrl-C f

コマンド・インタプリタへのコマンドのマニュアルを表示する。

ctrl-X k

通常は Kill-Buffer であるが、PDSS のプロセスが実行中の場合は Kill-Buffer を行う前に警告を出す。

b. KL1 プログラム制御

ctrl-C ctrl-Z

このバッファに対応する KL1 のウィンドウ・プロセスのアッテンション・ストリームに 1 を流す。Micro PIMOS ではタスクの停止要求として扱う。

ctrl-C ctrl-T

このバッファに対応する KL1 のウィンドウ・プロセスのアッテンション・ストリームに 2 を流す。Micro PIMOS ではタスクの統計情報の表示要求として扱う。

c. エミュレータ制御

ctrl-C !

ガベージコレクションの要求。

ctrl-C @

PDSS 全体の停止。Micro PIMOS のウィンドウとして使っていたバッファはそのまま残される。

ctrl-C ESC

PDSS を再起動する。

3. モードに関係無く定義されるコマンド

ctrl-C ctrl-P

そのウィンドウに PDSS モードのバッファを表示する。PDSS のバッファ群はサーキュラリストで管理されており、続けてこのコマンドを入力すると、次々に他のまだ表示されていないバッファを表示する。

ctrl-C p

ctrl-C ctrl-P とほぼ同様の機能であるが、他のウィンドウにバッファを表示する点が異なる。

付録-10　コンパイル用コマンドプロシジャの使用法

これは KL1 のコンパイルを行う為のコマンドプロシジャであり、UNIX のコマンドとして使うことができる。make コマンドのなかでコンパイルする為に有用である。これには KL1/KL1 コンパイラを使う版と KL1/Prolog コンパイラを使う版の 2 種類あるが、基本的な使い方は同じである。(使えるオプションが一部違う。)

コマンド:

pdsscmp [オプション] ファイル名 ...

オプション:

+i / -i :: クローズ・インデキシングのコードを出す / 出さない。デフォルトでは出す。(Prolog 版のみ有効, KL1 版ではインデキシングのコードを出せない。)

+m / -m :: MRB-GC の為のコードを生成する / しない。デフォルトでは生成する。生成した場合実行速度が若干低下する。(Prolog 版のみ有効, KL1 版では常に生成する。)

+a / -a :: アセンブルを行う / 行わない。デフォルトは行う。アセンブルを行う場合は出力としてアセンブラ・ファイル (xxx.asm) とセーブ形式ファイル (xxx.sav) が出力され、行わない場合はアセンブラ・ファイルだけ出力される。

+s / -s :: Micro PIMOS 用のコンパイル / ユーザー用のコンパイル。デフォルトはユーザー用。Micro PIMOS 用のコンパイルではシステムモード専用の組込述語が使えるようになる。このマニュアルに書かれている組込述語は全てユーザー用でコンパイル可能。(Prolog 版のみ, KL1 版では常に全組込述語をコンパイル可能。)

-o=PATH :: 出力ディレクトリを PATH に変更する。デフォルトはカレント・ワーキング・ディレクトリ。

ファイル名:

xxx.asm :: アセンブラ・ファイル (xxx.asm) をアセンブルしセーブ・ファイル (xxx.sav) を作る。

xxx.kl1 :: ソース・ファイル (xxx.kl1) をコンパイルしアセンブラ・ファイル (xxx.asm) を作る。さらにそれをアセンブルしセーブ・ファイル (xxx.sav) を作る。

xxx :: xxx.kl1 と書いたのと同じ。

使用例:

- 2 つのソース・ファイル append.kl1 と queen.kl1 のコンパイルとアセンブルを行い append.asm, appens.sav および queen.asm, queen.sav をカレント・ワーキング・ディレクトリに作る。

```
pdsscmp  append.kl1 queen.kl1    または
pdsscmp  append queen
```

- append.kl1 のコンパイルとアセンブル, queen.asm のアセンブルを行う。

```
pdsscmp  append.kl1 queen.asm
```

- ディレクトリ source の下の .kl1 ファイル総てについてコンパイルとアセンブルを行う。この時 xxx.asm と xxx.sav のファイルは -o で指定したディレクトリ object の下に作られる。

```
pdsscmp  -o=object source/*.kl1
```

付録-11 サンプル・プログラム

```
:- module sample.
:- public primes/1, primes/2.

primes(N) :- true | primes(N, PL),
    window:create([show|Window], "sample"), outconv(PL, Window).
primes(N, PL) :- true | gen(2, N, NL), sift(NL, PL).

gen(Max, S):- true | gen(1, Max, S).
gen(N, Max, S) :- N =< Max, M := N+1 | S=[N|S1], gen(M, Max, S1).
gen(N, Max, S) :- N > Max | S=[].

sift([P|L], S) :- true | S=[P|S1], filter(P, L, K), sift(K, S1).
sift([],    S) :- true | S=[].

filter(P, [Q|L], K) :- Q mod P=:=0 |             filter(P, L, K).
filter(P, [Q|L], K) :- Q mod P=\=0 | K=[Q|K1], filter(P, L, K1).
filter(P, [],    K) :- true        | K=[].

outconv([P|PL], W) :- true | W=[putt(P),nl|W1], outconv(PL, W1).
outconv([], W) :- true | W=[putb("END"),getc(_)].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 実行結果
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

| ?- sample:primes(10,PL).

yes.
| ?- sample:primes(10,PL)|PL.
PL = [2,3,5,7]

yes.
| ?- sample:primes(10).
2
3
5
7
END

yes.
| ?- halt.
```

付録-12　バグを発見したら

1. システムのバグを発見した人はすみやかに PDSS 開発グループに御知らせ下さい。連絡先は

   pdss@icot21.icot.junet

   です。また、デバッグを効率的に行う為に以下の点を明確にお願いします。

   a. PDSS(エミュレータ, Micro PIMOS) のバージョン番号
   b. コンパイラのバージョン番号
   c. バグの発見されたプログラム
   d. プログラムの起動方法と症状
   e. 実行時のログと異常と思われる個所

2. あなたのプログラムのバグであれば、次のことに気を付けて下さい。

   a. varchk は通したか。
   b. deadlock した場合、ロックした中に次のような形式のゴールが有ればファイルやウィンドウへのコマンド・ストリームが閉じられていないか、出力データに未定義変数が含まれている状態です。あなたのプログラムをチェックして下さい。

```
mpimos_file:xxxxxx( ... )      あるいは
mpimos_window:xxxxx( ... )
merge( ... ) in mpimos_file:xxxxx/x
merge( ... ) in mpimos_window:xxxxx/x
```

索引