ＰＤＳＳ使用手引き

（第１．１版）

１９８８年３月１４日

新世代コンピュータ技術開発機構
第四研究室

目次

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ているＫＬ１システムである。ＰＤＳＳは，その目的から，なるべく Multi-PSI 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。（Multi-PSI 上のシステムの仕様がまだ確定していないので非常におおまかに述べる）：

(1) 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。
(2) コードの管理も同様に処理系レベルで行っている。
(3) ＰＤＳＳでは、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
(4) 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。
(5) 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。
(6) プライオリティの計算方法が多少異なる。

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は極力マシーンに異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インタフェース部である。 Micro PIMOS に付いては３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に提供する Single User, Single Task の簡易ＯＳである。次頁にＰＤＳＳの全体構成のイメージ図を示す。

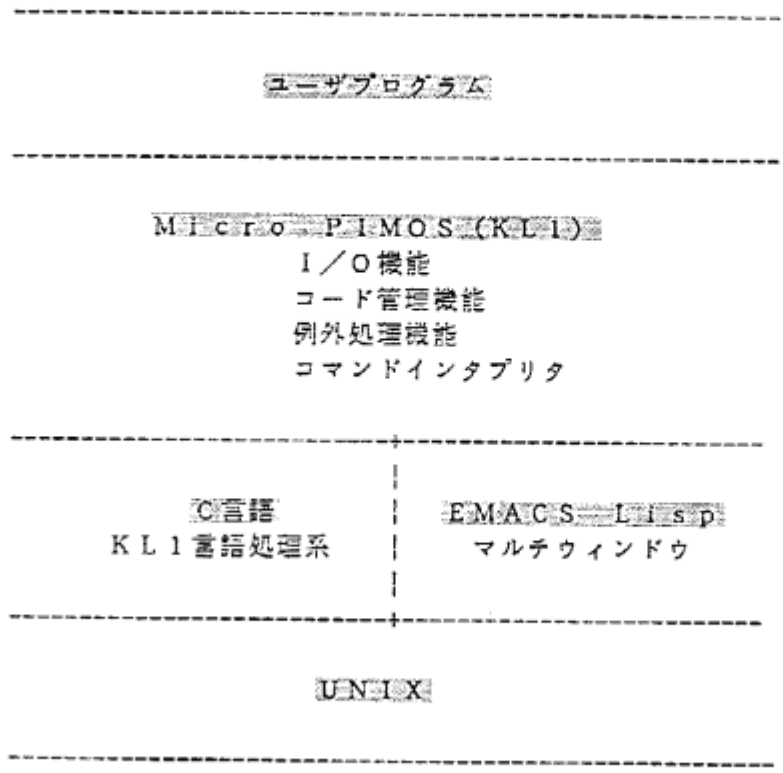


図. ＰＤＳＳの全体構成

図で、ＥＭＡＣＳ　Ｌｉｓｐ部は GNU-Emacs と呼べれるフルスクリーン・エディタのライブラリ定義部でありＰＤＳＳではこの GNU-Emacs の機能を利用してユーザにマルチウィンドウの環境を提供している。

2．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。（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 上のものとは多少異なる場合があります。）

2．1 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。

(1) 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。(失敗しない)

```
goal:     ?- p(a, X, b).
clause:   p(a, c, d) :- true | true.
```

(2) 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み述語に限定している。ガード部で記述可能な組込み述語については２．３章で述べる。

(3) 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.
```

(4) 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。

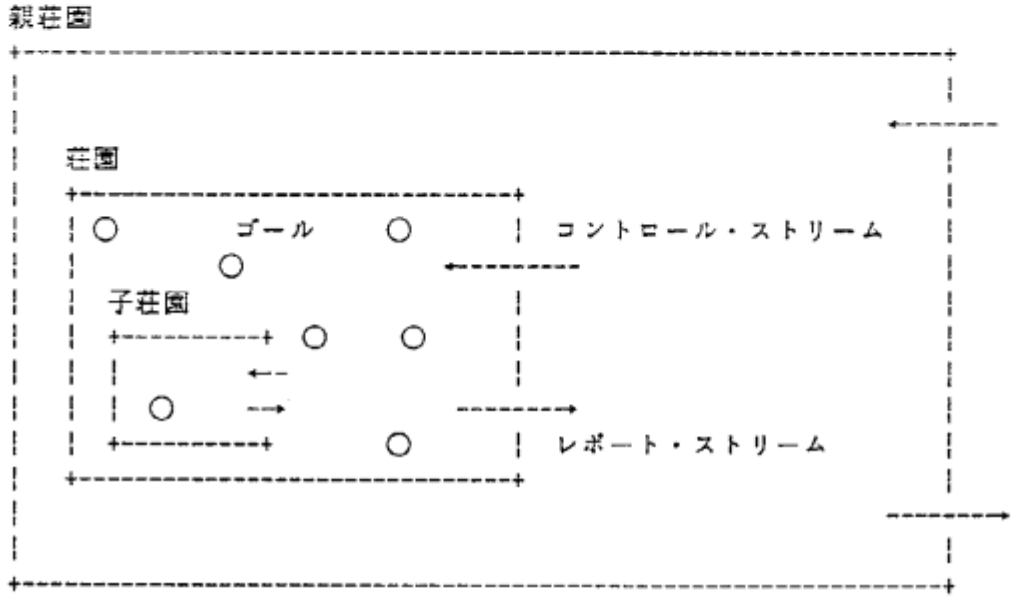
(5) 荘園
荘園と呼ぶ機能単位を導入し，この荘園ごとに実行優先順位の制御や実行量の制御が行える。ＯＳはこの荘園機能を中心に構築される。

(6) 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。

(7) 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。

２．２ 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．荘園のイメージ

(1) 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 more_reduction(R) なる命令を流せばよい。

(2) 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後の章で述べる。

２．２．１ 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 create/8 が定義されている。

ｃｒｅａｔｅ（モジュール名，述語名，引数，荘園内最小プライオリティ，
荘園内最大プライオリティ，タグ，コントロール，レポート）

荘園内最小／大プライオリティは，荘園内で実行されるゴールの，使用できるプライオリティの上限値と下限値を計算するための値。タグは荘園内のどのような例外を受信するかを示すタグ。コントロールにはコントロール・ストリーム，レポートにはレポート・ストリームがユニファイされる。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':create(primes,do,[1,300,Primes], 0,2, 0xFF000000,
                                                Control,Report)
```

２．２．２ 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。

(1) ｓｔａｒｔ
荘園内のゴールの実行を再開可能にする。

(2) 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。

(3) 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。

(4) ｒｅｍｏｖｅ
荘園を放棄する。以降，コントロール・ストリームにどのようなコマンドを流しても無視される。コントロール・ストリームを閉じてもｒｅｍｏｖｅと同じ意味になる。
荘園のコントロール・ストリームを棄てる場合には，ｒｅｍｏｖｅを流すか，ストリームを閉じなければ荘園そのものの実行が終了にならないので注意。

(5) ｓｔａｔｉｓｔｉｃｓ（Ｉｎｆｏ）
荘園内で行われたリダクションの数を Info にユニファイする。この数には子孫の荘園内のリダクション数も含まれている。

(6) ｍｏｒｅ＿ｒｅｄｕｃｔｉｏｎ（Ｒ）
荘園の現在の許容リダクションの上限値にＲで示される値を追加する。「資源の不足例外」に対して用いる。上限値が負の整数であれば無限の値を意味する。

(7) ｐｒｉｏｒｉｔｙ（Ｍｉｎ，Ｍａｘ）
荘園内で実行されるゴールの，使用できるプライオリティの上限値と下限値をＭｉｎとＭａｘから計算しユニファイする。ここで得られるＭｉｎ及びＭａｘは，この荘園の親荘園の下限値とこの荘園を生成したゴールのプライオリティとの差に対する割合であり，正の整数。荘園の下限値は親荘園の下限値にＭｉｎで計算される値を加えた値。上限値は生成したゴールのプライオリティからＭａｘで計算される値を引いた値。

２．２．３ 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，荘園の状態情報と例外情報の２種類のものがある。

(1) 荘園の状態情報

a) ｓｕｃｃｅｅｄｅｄ
荘園内の全てのゴール及び全ての子荘園の実行が終了した。

b) ａｂｏｒｔｅｄ
コントロール・ストリームから流した abort コマンドによって荘園及びその子孫の全ての荘園が放棄された。

(2) 例外情報
例外情報には，その例外に対する処理を指定することができるものがある。以下で NewGoal なる引数を持ったものがそれに当たる。処理系では例外を発見すると、その例外が発生した荘園内に次に示すような例外処理用のゴールを生成し NewGoal にゴールがユニファイされるのを待つようにする。

```
exception_handling(Module:Goal) :- true |
    get_predicate(Goal, PredicateName, Args),   % 述語名と引数の取り
                                                % 出し。
    apply(Module, PredicateName, Args).         % ゴールの実行
otherwise.
exception_handling(Goal) :- true |
    raise(illegal_goal_format(Goal)).       % 指定されたゴールの形式
                                            % の誤り。
```

a) ｅｘｃｅｐｔｉｏｎ（Ｃｏｄｅ，Ｇｏａｌ，ＮｅｗＧｏａｌ）
荘園内において例外事項が発生した。Code は例外の種類を表す正の整数。Goal は例外を起こしたゴールである。NewGoal には例外を起こしたゴールの代りに実行して欲しいゴールをタームの形式でユニファイする。

b) ｅｘｃｅｐｔｉｏｎ（Ｃｏｄｅ，ＯｐＣｏｄｅ，Ａｒｇｓ，ＮｅｗＧｏａｌ）
荘園内で実行しようとした組込み述語で例外事項が発見された。
Code は例外の種類を表す正の整数。OpCode は組込み述語のオペコード。Args は

例外を起こした際の引数。NewGoal には例外を起こしたゴールの代りに実行して欲しいゴールをタームの形式でユニファイする。

c) ｒｅｄｕｃｔｉｏｎ＿ｌｉｍｉｔ
最大許容リダクション数を超えた。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。

d) ｄｅａｄｌｏｃｋ（ＬｉｓｔＯｆＬｏｃｋｅｄＧｏａｌｓ）
荘園内でデッドロック状態が発見された。発見のタイミングはＧＣの時。つまりＧＣを起こさなければデッドロックは発見されない。部分的なデッドロックも，デッドロックしている部分のみが知らされる。ListOfLockedGoals はデッドロックしているゴールのリスト。

e) ｒａｉｓｅ（Ｔｙｐｅ，Ｉｎｆｏ，ＮｅｗＧｏａｌ）
組込み述語 raise が実行された。Type，Info は 述語 raise によって渡されたターム。

２．３ 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。（ここで述べるデータ型は KL1 のプログラムで扱える／扱って意味があるものである。）

(1) 未定義変数
(2) 整数　(-(2**31) から 2**31-1 の範囲)
(3) アトム　・・・ abc, 'ABC'
(4) リスト　・・・ [1,2,3], [1|X]
(5) ベクタ（要素ゼロのものを含む一次元配列）　・・・ {a,Y,b}, f(X), {}
(6) ストリング（null string も含む）　・・・ "abc", ""

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。ダブル・クォート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。

ファンクタ形式で記述／表示されているものも全てベクタである。例えば，f(a) と {f, a} はユニファイ可能であり，ユニフィケーションは成功する。

ストリング同士のユニフィケーションは全ての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。

２．４ 組込述語

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

```
vector(X, ^Size) :: G
---------------    ----
        ↑            ↑
   呼出し形式    記述可能な場所
```

組込み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，その組込み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以下のシンタックスで示されるマクロが用意されている。（演算子の意味は DEC-10 Prolog に準拠しているのでそちらを参照されたし。）

```
<比較式>        ::= <式> <比較演算子> <式>
<算術演算式>    ::= <変数> ":=" <計算式>
<式>            ::= <項> | <式> <算術演算子１> <項>
<項>            ::= <因子> | <項> <算術演算子２> <因子>
<因子>          ::= <整数> | "(" <式> ")"
<比較演算子>    ::= ">"   | "<"   | ">="   | "=<"
<算術演算子１>  ::= "+"   | "-"
<算術演算子２>  ::= "*"   | "/"   | "mod"  |
                    "<<"  | "<<"  |
                    "\/"  | "/\"
```

(1) タイプ・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。

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, ^Length, ^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。

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サイズを Size とユニファイし要素サイズを ElementSize とユニファイする。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サイズを Size とユニファイし要素サイズを ElementSize とユニファイするとともに X を NewString にユニファイする。それ以外は失敗。

atomic(X) :: G
　X が未定義ならば中断。X がアトム，整数，ストリングのいずれかであれば成功。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）で未定義ならば Result に success をユニファイし成功。すでに束縛されていれば Result に failure をユニファイし成功。

(2) 比較

less(Integer1, Integer2) :: G
　Integer1 または Integer2 が未定義ならば中断，双方がが整数でありかつ Integer1 < Integer2 のとき成功。それ以外では失敗。

not_less(Integer1, Integer2) :: G
　Integer1 または Integer2 が未定義ならば中断，双方がが整数でありかつ Integer1 >= Integer2 のとき成功。それ以外では失敗。

greater(Integer1, Integer2) :: G
　Integer1 または Integer2 が未定義ならば中断，双方がが整数でありかつ Integer1 > Integer2 のとき成功。それ以外では失敗。

not_greater(Integer1, Integer2) :: G
　Integer1 または Integer2 が未定義ならば中断，双方がが整数でありかつ Integer1 =< Integer2 のとき成功。それ以外では失敗。

equal(Integer1, Integer2) :: G
　Integer1 または Integer2 が未定義ならば中断，双方がが整数でありかつ Integer1 == Integer2 のとき成功。それ以外では失敗。

not_equal(Integer1, Integer2) :: G
　Integer1 または Integer2 が未定義ならば中断，双方がが整数でありかつ Integer1 =\= Integer2 のとき成功。それ以外では失敗。

not_unified(X, Y) :: G
　X も Y も変数を含む任意の項。
　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。まったく同じ値もしくは構造であることが確定すれば失敗。それ以外は中断。

X ＼= Y とも書いて良い。

(3) 算術演算

add(Integer1, Integer2, ^Integer3) :: 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 Integer3 にユニファイする。

subtract(Integer1, Integer2, ^Integer3) :: 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 Integer3 にユニファイする。

multiply(Integer1, Integer2, ^Integer3) :: 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 Integer3 にユニファイする。

divide(Integer1, Integer2, ^Integer3) :: 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 Integer3 にユニファイする。

modulo(Integer1, Integer2, ^Integer3) :: 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 Integer3 にユニファイする。

negate(Integer1, ^Integer2) :: GB
    Integer1 が未定義ならば中断，整数でなければ失敗/例外。
    Integer1 を符号反転した値を Integer2 にユニファイする。

shift_left(Integer, ShiftWidth, ^NewInteger) :: GB
    Integer または ShiftWidth が未定義変数ならは中断，整数でなければ失敗/例外。
    Integer を ShiftWidth ビットだけ左にシフトし結果を NewInteger とユニファイ
    する。

shift_right(Integer, SiihftWidth, ^NewInteger) :: GB
    Integer または ShiftWidth が未定義変数ならは中断，整数でなければ失敗/例外。
    Integer を ShiftWidth ビットだけ右にシフトし結果を NewInteger とユニファイ
    する。

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 Integer3 にユニファイ
    する。

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 Integer3 にユニファイ
    する。

xor(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 Integer3 にユニファイする。

compliment(Integer1, ^Integer2) :: GB
    Integer1 が未定義ならば中断，整数でなければ失敗/例外。
    Integer1 の各ビットを反転した値を Integer2 にユニファイする。

(4) ベクタ関係

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。
    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新しい未定義変数で初期化される。

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。また，
    Position が未定義ならば中断，非負整数でなければ失敗，Vector の要素数以上の整数の場合も失敗。これ以外の場合は，Vector の Position 番目の要素を Element とユニファイする。

vector_element(Vector, Position, ^Element, ^NewVector) :: G
    Vector が未定義ならば中断，ベクタ以外の値であれば例外。また，
    Position が未定義ならば中断，非負整数でなければ例外，Vector の要素数以上の整数の場合も例外。これ以外の場合は，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, ^OldElement, NewElement, ^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。また，
    Position が未定義ならば中断，非負整数でなければ例外，Vector の要素数以上の整数の場合も例外。これ以外の場合は，Vector の Position 番目の要素を OldElement とユニファイし，Vector の Position 番目を NewElement と置き換えた新しいベクタを生成するとともに，生成したベクタを NewVector とユニファイする。

(5) アトム／ストリング関係

new_string(^String, Size, ElementSize) :: B
    Size 及び ElementSize が未定義なら中断。非負整数でなければ例外。それ以外の場合は，要素数 Size，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ゼロで初期化される。

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。それ以外の場合は，String の Position 番目の要素を 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。それ以外の場合は，String の Position 番目の要素を Element とユニファイするとともに String を NewString とユニファイする。

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。それ以外の場合は，String の Position 番目の要素を NewElement 置き換えた新たなストリングを生成し NewString とユニファイする。

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，同じタイプのストリング以外なら例外。それ以外の場合は 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。

symbol_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。当面ＰＤＳＳでは８ビットストリング。

make_symbol(String, ^Atom) :: 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。

(6) 高階機能

apply(ModuleName, PredicateName, Args) :: B
ModuleName 及び PredicateName が未定義であれば中断，アトム以外なら例外。Args が未定義なら中断，ベクタ以外であれば例外。それ以外の場合はモジュール名が ModuleName，述語名が PredicateName である述語を引数環境 Args で呼び出

す。

(7) ストリーム・サポート

merge(In, Out) :: B
マージャを生成し，そのマージャに対する入力ストリームを In と，出力ストリームを Out とユニファイする。

マージャの論理的な定義は，以下に示すような2から無限の引数まで持つような述語として定義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).
                    . . .
                    . . .
```

merge_in(In1, In2, ^In) :: B
マージャへの入力ストリーム In に {In1, In2} をユニファイする。マージャはベクタを受け取るとベクタ内の全ての要素を新たに入力ストリームとして加える。

(8) 特殊入出力

read_console(^Integer) :: G
コンソール・ウインドウから整数を読み込む。

display_console(X) :: G
コンソール・ウインドウに X の現在の値を（たとえ未定義でも）書き出す。

(9) 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。また Type が基底項でなければ中断。それ以外の場合，組込述語 raise 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
raise(Type, Info, NewGoal) をユニファイする。

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

X をもとにゼロ以上 Width 以下の範囲の整数を生成し Value とユニファイする。ここで X は未定義変数以外の任意の値。Width は整数。X 及び Width が未定義変数であれば中断。

X がベクタの場合，その長さと第一要素をもとに値を生成するため，第一要素が未定義変数であれば中断する。

２．５ 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）をゴールごとに指定できる機能がある。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・キュー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／下限値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／下限値との相対値で指定する。指定方法は，指定したいゴールの直後に，次のように書けば良い。

Ｇｏａｌ@ｐｒｉｏｒｉｔｙ（『指定法』，『割合』）

先に述べたように指定法には，所属荘園内割合指定と所属荘園内自己相対指定の２種類がある。

(1) 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上／下限との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 Goal のプライオリティは，

$$\text{下限値} + (\text{上限値} - \text{下限値}) \times \frac{\text{割合}}{2**12}$$

で 決まる。

(2) 所属荘園内自己相対指定

この指定方法は，呼び出すゴールのプライオリティとの相対値で指定する方法であり次の形式で書く。

Ｇｏａｌ@ｐｒｉｏｒｉｔｙ（Ｓ，『割合』）

このとき Goal のプライオリティは，呼び出すゴールの論理プライオリティをＣｐとすると，

『割合』＞０ならば：

$$Cp + (上限値 - Cp) \times \frac{割合}{2**12}$$

『 割合』＜０ならば：

$$Cp - (Cp - 下限値) \times \frac{割合}{2**12}$$

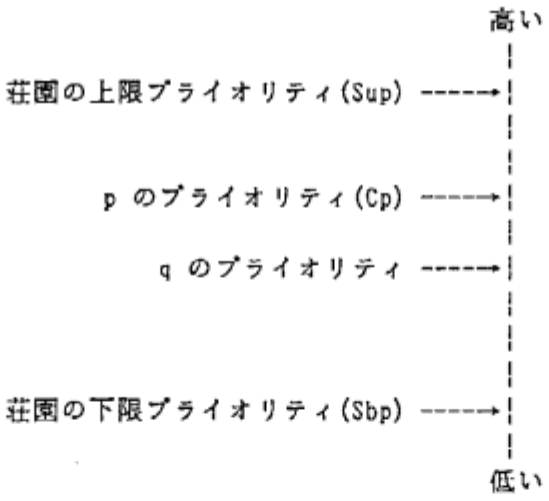
で 決まる。

割合は 2**12 以下で -(2**12) 以上の整数でなければならない。また，割合が 0 の場合は (1) の場合も (2) の場合も呼出しゴールと同じプライオリティになる。

例を示そう、次のプログラムで goal の論理プライオリティが Cp とすると、ゴール q の論理プライオリティは、式－１で計算される。

```
goal:      ?- p.
clause:    p :- true | q@priority(S, -10).
```

プログラム



$$Cp - (Cp - Sbp) \times \frac{10}{2**12} \qquad (式－1)$$

２．６ 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
・条件分岐の記述
・ＤＣＧ風の記法
・プライオリティ指定

が挙げられる。（プライオリティの指定については先の章で述べた通りである。）

(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／引き数.
```

と宣言する必要がある。組込み述語 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。

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。

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記述する。

```
モジュール名：GOAL
```

(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。

・節の優先順位
　述語の定義中（節と節の間）に alternatively. と書く。

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

alternatively. の前後の節（群）の順序関係は保証される。

・逐次実行

述語の定義中（節と節の間）に otherwise. と書く。

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

otherwise. 以前に書かれた節の全てが失敗して始めて otherwise 以降の節（群）が実行される。

(3) 条件分岐の記述

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

' $『Head の述語名』／『引数の数』／『分岐式の番号』'

であり，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。

また，展開系を見ても分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み述語しか書けない。

【注意】現在の版では条件分岐式はネストしてはならない。また，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。

(4) ＤＣＧ風の記法

DEC-10 Prolog におけるＤＣＧの記法と同様，'-->' を使って記述された節は，コンパイラのプリプロセッサによって，d-list を表わす２つの引数を付け加えてコンパイルされる。Prolog におけるＤＣＧとの違いは，リストの生成にしか使えない点である。

例を示そう。次のようなプログラムは，

```
foo(X,Y) --> X>0  | { p(X,Z) }, q(Z,Y).
foo(0,Y) --> true | [Y].
```

以下のように変換される。

```
foo(X,Y,S0,S1) :- X>0 |
    p(X,Z),
    q(Z,Y,S0,S1).
foo(0,Y,S0,S1) :- true | S0=[Y|S1].
```

引数の付加を抑制したいボディ部のゴールは Prolog の場合と同様 '{' と '}' で囲めばよい。ガード部のゴールには拡張引き数は付加されないので特に抑制する必要も無い。また条件分岐の記法とＤＣＧ風の記法は併用して使っても良い。

【注意】現在 KL1 のコンパイラは、Prolog で記述されたコンパイラと KL1 自身で記述されたコンパイラが提供されているが、ＤＣＧ風の記法は Prolog 上のコンパイラでのみ使用可能である。

3. 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。

(1) コマンド・インタプリタ
(2) 入出力機能
(3) コード管理
(4) 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２．２章で説明した荘園の機能を用いて実現されている。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 23:31 ビットを使用しては成らない。(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。)

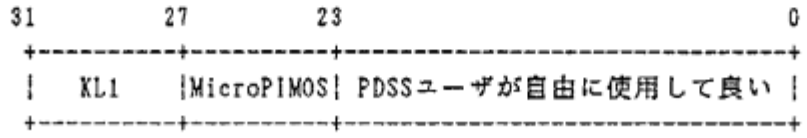


図. 荘園の例外タグ

以下の章では上記のサービスについてそれぞれ簡単に説明を加える。

3．1 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 '| ?- ' でトレース・モードの時は '[debug]?- ' である。

コマンドは、コマンド・インタプリタが提供するコマンドの他に、ボディ部に記述できる組込述語も実行可能である。

3．1．1 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。

- ',' :: 前後の各コマンド群を、並列に実行する。
- ';' :: 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)。
- '|' :: 前のコマンド群の実行終了後、後ろに書かれた変数(区切りの文字は ',')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。
```

3．1．2 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。

(1) 基本コマンド

help ::
    ヘルプコマンドの一覧を表示する。

help(HelpNumber) ::
    HelpNumber で指定されたタイプのコマンドの一覧を表示する。

gc ::
    ヒープ領域の GC を起動する。

gc(h) ::
    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出来るコマンドならば何を書いても良い。
    複数のコマンド列を '.' で区切って書いても良い。この場合、コマンドを ';' で繋げた場合と同じ意味になる。(シンタックスエラーの扱いだけが異なる)
    コメントには KL1 のプログラムと同様 % と /* */ が使える。

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。

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。

halt ::
ＰＤＳＳを終了する。この時開いていたウインドウは全て自動的に閉じられる。

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。

(2) コードコマンド

comp(FileName) ::
FileNameで指定される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・モードはオフ。

comp(FileName, CompFileName) ::
FileName で指定されるKL1ソース・ファイルをコンパイルし、CompFileNameで指定されるファイルに出力する。

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。

load(FileName) ::
FileName で指定されるセーブ・ファイル（あるいはアセンブラ・ファイル）をコード領域にロードする。ロードしたモジュールのトレース・モードはオフ。
ファイルの属性は ".sav" と ".asm" であり両方のファイルが存在する場合，属性が ".sav" の方を優先する。

dload(FileName) ::
FileName で指定されるセーブ・ファイル（あるいはアセンブラ・ファイル）をコード領域にロードする。ロードしたモジュールのトレース・モードはオン。
ファイルの属性は ".sav" と ".asm" であり両方のファイルが存在する場合，属性が ".sav" の方を優先する。

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する。

save(ModuleName) ::
ModuleName で示されるモジュールの実行コードを，指定されたディレクトリ（デフォルトは ～/.PDSSsave）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で，save(ModuleName) 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。

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にし，Modules とユニファイする。

(3) デバックコマンド

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 オンにする。
　このときデバッグ・モードが自動的にオンになる。

notrace(ModuleName) ::
　ModuleName で示されるモジュールのコードのトレース・モードをオフにする。

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・モードをオンにする。

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・モードをオフにする。

spying(ModuleName) ::
　ModuleName で示されるモジュール中のスパイ・モードに成っている述語の一覧を表示する。

debug ::
　デバッグ・モードをオンにする。

nodebug ::
　デバッグ・モードをオフにする。

backtrace ::
　バックトレース情報(デッドロック情報)の表示モードをオンにする。

nobacktrace ::
　バックトレース情報(デッドロック情報)の表示モードをオフにする。

varchk(FileName, Mode) ::
　FileName で指定される KL1 のソース 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ウインドウに表示する。
　FileName はストリング Mode はアトムで指定する。

Mode :: o ... 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 ... ＭＲＢが黒くなる変数を表示する。
a ... o と m の両モードの変数を表示する。

(4) 環境コマンド

コマンド・インタプリタの環境変数には以下のものがある。

world :: カレント・ディレクトリのパス名。
タイプはストリング。
trace :: トレーサのトレース・モード、
値は on または off で初期値は off。
backtrace :: バックトレース情報の表示モード、
値は on または off で初期値は 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。
savedir :: オートロードや save_all の対象となるディレクトリのパス名。
タイプはストリング。 初期値は "~/.PDSSsave"。
auto_load :: オートロードを行うかどうかのフラグ。
値は yes または no で初期値は yes。
reduction :: タスク生成時に与えるリダクション数、
タイプは正の整数。初期値は 100000000。

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、
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参照し, ^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 ::
シェルの持つ環境変数のすべての値を表示する。

resetenv ::
シェル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。

(5) 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。

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。

3．2 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種類のものがある。ユーザは 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は入出力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。

3．2．1 コマンド・ストリームの獲得

以下のゴールを実行することにより，ユーザはコマンド・ストリームを得ることができる。

(1) ウィンドウ

window:create(Stream, WindowName) ..... (a)
window:create(Stream, WindowName, ^Status) ... (b)
    ウィンドウ名 WindowName （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。
    (a) の形式でウィンドウのクリエイトが失敗すると，タスクは強制終了される。
    クリエイト直後のウィンドウは隠れた状態にある。
    (b) の形式の Status には以下のものがユニファイされる。
      success :: 成功。
      error(Info) :: 失敗。
        Info :: cannot_create_window : ウィンドウがオープンできない。
             :: window_already_exist : 同じ名前のウインドウが既に存在している。
             :: bad_window_name_type : WindowName がストリングでない。

(2) ファイル

file:create(Stream, FileName, Mode) ..... (a)
file:create(Stream, FileName, Mode, ^Status) ... (b)
    ファイル名 FileName （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。
    a) の形式でファイルのクリエイトが失敗すると，タスクは強制終了される。
    b) の形式の Status には以下のものがユニファイされる。
      success :: 成功。
      error(Info) :: 失敗。
        Info :: cannot_open_file : ファイルがオープンできない。
             :: bad_file_name_type : FileName がストリングでない。
             :: bad_open_mode_type : Mode がアトムでない。

:: bad_open_mode : Mode が r,w,a 以外のアトムである。

3．2．2 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。これらのコマンドは，流したコマンド・ストリームが繋がるＩ／Ｏにのみで有効で，他のＩ／Ｏには適用されない。

(1) 入力用コマンド

Ｉ／Ｏから入力を行なう。読み込みが終了した（エンド・オブ・ファイルになった）あとの入力コマンドには、end_of_file を返し続ける。

getc(^Char) ::
　Ｉ／Ｏから一文字を読み込み，そのコード（0 ≦ 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，end_of_file をユニファイする。

getl(^String) ::
　Ｉ／Ｏから一行を読み込み，その文字列を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，end_of_file をユニファイする。

getb(^Buffer, Size) ::
　Ｉ／Ｏから Size（0 < Size）で示される数だけ文字を読み込み，その文字列を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、途中でファイルがエンド・オブ・ファイルになった場合は、それまでの文字を入力とする。
　エンド・オブ・ファイルならば，end_of_file をユニファイする。

gett(^Term) ::
　Ｉ／Ｏからターム一個を構成する文字列を読み込み，その文字列の構文解析を行ない，タームに変換し，Term とユニファイする。
　構文解析でエラーがある場合、ウインドウはそのウインドウにエラーを通知し、再び入力待ちになる。ファイルはシェルのウイ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，end_of_file をユニファイする。

getft(^Term, ^NumberOfVariables) ::
　入力方式は gett と同じであるが、ターム中の変数は $VAR(N,VN) で表す。
　N は変数番号（0 ≦ NumberOfVariables），VN は変数名（ストリング）である。
　また、変数の種類数を NumberOfVariables（0 ≦ 種類数）にユニファイする。
　エンド・オブ・ファイルならば，Term には end_of_file を，NumberOfVariables には 0 をそれぞれユニファイする。

(2) 出力用コマンド

Ｉ／Ｏへ出力を行なう。Micro PIMOS ではＩ／Ｏへ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 コマンドが流された時。
・Ｉ／Ｏをクローズした時。
・入力コマンドが流された時(ウインドウのみ)。
・show, hide コマンドが流された時(ウインドウのみ)。

putc(Char) ::
　Ｉ／Ｏへコード Char (0 ≦ Char ≦ 255) で示される文字を書き出す。

putl(String) ::
　Ｉ／Ｏへ String (ストリング) で示される文字列を書き出し，改行を行なう。

putb(Buffer) ::
　Ｉ／Ｏへ Buffer (ストリング) で示される文字列を書き出す。改行は行なわない。

putt(Term, Length, Depth) ::
　Ｉ／Ｏへ Term で示されるタームを，構造体の長さ Length (0 < Length)，
　深さ Depth (0 < Depth) 以内の範囲で書き出す。Length, Depth を超えた部分は
　"..." と出力される。
　putt で書出したタームは必ずしも gett,getft で読めるとは限らないので注意。

putt(Term) ::
　Ｉ／Ｏへ Term で示されるタームを書き出す。構造体の長さ，深さの制限はデフォ
　ルト値を使用する。制限を超えた部分は "..." と出力される。
　putt で書出したタームは必ずしも gett,getft で読めるとは限らないので注意。

puttq(Term, Length, Depth) ::
puttq(Term) ::
　putt と同じだが，必要に応じてアトムにシングルクォーテーション (') を
　付加して出力する。
　puttq の後にピリオドを出力すれば，gett,getft で読み込むことができる。

nl ::
　改行を行なう。

tab(N) ::
　空白を N (0 ≦ N < 1000) で示される数だけ出力する。

(3) 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
を変更する。

print_length(Length) ::
　構造体の長さの制限を，Length (0 < Length) で示される数に設定する。
　初期値は 10 である。

print_depth(Depth) ::
　構造体の深さの制限を，Depth (0 < Depth) で示される数に設定する。

初期値は 10 である。

print_var_mode(VariableMode) ::
　変数を表わすターム SVAR(N,VN), SVAR(N) の出力形式を変更する。
　初期値は na である。
　　na :: Name-Mode.　SVAR(N,VN) -> VN で出力。
　　　　　　SVAR(N)　　-> A,B,C ... で出力。
　　nu :: Number-Mode. SVAR(N,VN) -> _N で出力。
　　　　　　SVAR(N)　　-> _N で出力。

(4) 出力系バッファコマンド

出力用のバッファ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。
　Status には以下のものがユニファイされる。
　　done :: 出力が終了した。

buffer_length(BufferLength) ::
　バッファのサイズを BufferLength (0 < BufferLength) に変更する。
　　初期値　ウィンドウ :: 512 バイト
　　　　ファイル　:: 2048 バイト

(5) 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 ≦ Precedence ≦ 1200), 型 Type (アトム),
　名前 OperatorName (アトム) の演算子を追加する。

remove_op(Precedence, Type, OperatorName) ::
　順位 Precedence (1 ≦ Precedence ≦ 1200), 型 Type (アトム),
　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) の演算子の定義をリストにし,
　Definition とユニファイする。定義は (順位,型) の形式である。

(6) 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
　のマージをしても, CommandList 内のコマンドの連続性は保証される。

(7) 制御コマンド

close(^Status) ::

I／Oをクローズする。close コマンドを流した後は、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）。
Statnnus には以下のものがユニファイされる。
success :: クローズが終了した。

(8) ウィンドウコマンド

ウィンドウI／O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（ストリング）にユニファイし，プロンプトを New で示されるストリングに変更する。

3．3 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。
コマンド・ストリームを [] で閉じると、プロセスは終了する。

3．3．1 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 （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(Info) :: 失敗。
Info :: cannot_access : ディレクトリにアクセスできない。
:: bad_directory_name_type : DirectoryName がストリングでない。

3．3．2 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) ::
現在のディレクトリのフルパス名（ストリング）を PathName にユニファイする。

listing(WildCard, ^FileNames, ^Status) ::
　WildCard（ストリング）で表現されるファイルのパス名（ストリング）のリストを
　作成し，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(Info) :: 失敗。
　　　Info :: cannot_listing : リスティングできなかった。

delete(WildCard, ^Status) ::
　WildCard（ストリング）で表現されるファイルを，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(Info) :: 失敗。
　　　Info :: cannot_delete : 削除できなかった。

open(Stream, FileName, Mode, ^Status) ::
　ファイルをオープンする。
　３．２．１ コマンドストリームの獲得 (2) ファイル の 形式 (b) を参照。

３．４　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ことは無い。

３．４．１　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。

a) mpimos_io_device:windows(Stream)
　ウインドウ・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。

b) mpimos_io_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。

３．４．２　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インドウ、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－２　１　　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。
ただし、共通の入出力コマンドは以下のものしか使用できない。

a) ウインドウ
　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。

b) 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。

3．5　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。

(1) 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・モードになっている述語名の一覧）を要求に応じて表示する機能。

(2) 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・ロード機能を使うためには，ユーザは自身のホーム・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 savedir のデフォルト値が"~/.PDSSsave"である為である。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 savedir の値で決められる。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 にセットすることで可能である。

3．6　例外情報の表示

PDSS で規定している例外には付録－1に示したものがあるが，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。

４．ＰＤＳＳの起動と使用

PDSS の起動法には Micro PIMOS 付で起動する方法と，Micro PIMOS 無しで起動する方法がある。Micro PIMOS 付で起動する場合には必ず GNU-Emacs の下で起動されなければならないが，Micro PIMOS 無しの場合にはそのような制限はない。本手引書では以下主に Micro PIMOS 付で起動する方法と使用法について述べる。

４．１ 起動法

(1) GNU-Emacs の下での実行

PDSS を GNU-Emacs の下で実行する為には，あらかじめ用意されている Emacs-Lisp のライブラリをロードする必要がある。ロード方法は GNU-Emace の中で次のように行なえば良い。

```
M-x load-file
Load File: ~pdss/release/emacs/pdss.elc
Loading /login2/pdss/emacs/pdss.elc ...done
```

ライブラリがロードされた後に以下のようにコマンドを打てば自動的に PDSS が起動される。

```
M-x pdss
```

立ち上げ時のオプションを指定したい時には M-x に先だって C-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。

```
C-u M-x pdss
PDSS Option ?: [ パラメタ ]
```

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 System 全体が停止することはなくなる。
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。これの詳細は後で解説する。

(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/release/emulator/pdss [ パラメタ ]
```

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 System 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。

4．2 パラメタ

起動時に指定できるパラメタと指定方法について述べる。

(1) パラメタの種類

パラメタは以下のものが指定できる。括弧内の値がデフォルト。

```
-hNNN   Heap Area の大きさを NNN cell とする  (200000)
-cNNN   Code Area の大きさを NNN byte とする  (300000)
-wNNN   Heap Area の GC を行うタイミングを指定する  (1000)
        Heap Area の残りが NNN cell 以下になった場合に GC を行う。
-rNNN   Default Reduction Limit を NNN 回とする (1000000)
+v/-v   トレーサー等における変数の表示方法を指定する  (-v)
        +v の場合は Heap-Bottom からの相対アドレス _XXX で表示する。
        -v の場合は A,B,C, ... という名前で表示する
        GC によってずれるので注意。
+t/-t   startup ファイルを使って起動する／しない (+t)
        +/- を付けずにファイル名を書くとそれを startup ファイルとして
        起動する。
```

(2) パラメタの指定方法

パラメタを指定するには次の２とうりの方法がある。

1. 起動時に PDSS Option ?: への入力により指定する。
   例） PDSS Option ?: -h300000 -c50000 +v
2. 環境変数(PDSSOPT)により指定する。
   例） % setenv PDSSOPT "-h300000 -c50000 +v"

このとき，同じ項目に関して両方で指定された場合には起動時の引数が優先される。

4．3 EMACS のコマンド

ロードした emacs ライブラリには以下に示すコマンドが定義されている。

```
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。
```

［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．トレーサ

ＰＤＳＳで実現しているトレーサについて説明する。

5．1 考え方

基本的にゴール単位のトレースである。トレースの方法として，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が可能である。

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開始するものであり，モジュール全体や個々の述語ごとにトレース・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。

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。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) がら呼び出される 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).
```

5．2 見方

トレース・ポイントには，ゴールの呼出し時，中断時，復帰時，失敗時，（割込みか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）スワップ・アウト時があり，それぞれ CALL:, SUSP:, RESU:, FAIL:, SWAP: で表示される。（ゴールの呼出しにはＴＲＯによる呼出しと，ゴール・キューから取り出された呼び出しがありCALL: は後者を意味する。また前者は Call: と表示されるので注意）

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^

また複数箇所から参照されている可能性のあるもの（所謂ＭＲＢオンの状態）にはそのターム表示の直後に 'x' が印字される。

5．3 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（そのゴールから呼び出されるゴール群）のトレースをしない。

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]
　　次のトレース・モードが指定されている述語コードが呼び出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呼び出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]
　　次のトレース・モードのゴールが呼び出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呼び出されるまでトレースを行なった後で止まる。

Step to Next Spy Point :: ss [COUNT]
　　次のトレース・モードが指定されている述語コードが呼び出されるか，次のトレース・モードのゴールが呼び出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呼び出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]
　　次のトレース・モードが指定されている述語コードが呼び出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呼び出された後で止まる。

Skip to Next Spied Goal :: ng [COUNT]
　　次のトレース・モードのゴールが呼び出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呼び出された後で止まる。

Skip to Next Spy Point :: ns [COUNT]
　　次のトレース・モードが指定されている述語コードが呼び出されるか，次のトレース・モードのゴールが呼び出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呼び出された後で止まる。

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。

Set Procedure Spy :: p MODULE:PROCEDURE { MODULE:PROCEDURE }

指定した述語コードをトレース・モードとする。

Set Goal Spy :: g
指定した時点で表示しているゴールをトレース・モードとする。

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。

Reset Procedure Spy :: P MODULE:PROCEDURE { MODULE:PROCEDURE }
指定した述語コードをトレース・モードでなくする。

Reset Goal Spy :: G
指定した時点で表示しているゴールをトレース・モードでなくする。

Re-Write Goal :: w LENGTH [DEPTH]
Print-Length, Print-Depth を変えてゴールを再表示する。

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ーする。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モニターする。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る。LIMIT を指定すると、値をモニターするときに LIMIT 回まで止まらずにモニターできる。LIMIT を指定しないと、具体化される時にその値を表示し、コマンドの入力待ちとなる。

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 の時にはその変数を待っているゴールを表示する。MGHOK の時にはそのマージャの出力側の変数が表示される。

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。変数及びその値を以下に示す。

変数名	意味と値
pv ...	Print Variable Mode. n か a で指定する。 n = Name-Mode (A,B,C ..). a = Address-Mode (_NNNN).
pl ...	Print Length. 整数で指定する。
pd ...	Print Depth. 整数で指定する。
g ...	Gate Switch. n, t, s から成る５文字で指定する。各文字は順に Call, Susp, Resu, Swap, Fail の各 Gate-Switch に対応する。 n = No-Trace. t = Trace (Not Stop). s = Trace (Stop).
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。

6．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

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ゴールが有れば、少なくとも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:request     あるいは,
mpimos_window:request
```

c) deadlock した場合、ロックした中に次のようなゴールが有れば、マージャの入力ストリームへの複数の参照パスが存在し、マージャへのストリームを閉じ忘れているはずです。あなたのプログラムをチェックして下さい。

```
pdss_runtime_body_builtin:active_unify(X^, Z^)
pdss_runtime_body_builtin:active_unify(Y^, Z^)
```

【付録－１】ＰＤＳＳのインストールの手引き

１．ＰＤＳＳのファイル構成

ＰＤＳＳは以下のようなファイルから構成されている。

```
ＰＤＳＳ
  ¦
  +- Makefile
  +- emulator/
  ¦     +- Makefile
  ¦     +- エミュレータのソースプログラム(C)
  +- runtime/
  ¦     +- Makefile
  ¦     +- エミュレータのランタイムサポート用プログラム(KL1)
  ¦     +- エミュレータのセーブ形式ファイル
  +- mpimos/
  ¦     +- Makefile
  ¦     +- MicroPIMOSのソースプログラム(KL1)
  ¦     +- MicroPIMOSのセーブ形式ファイル
  +- compiler/
  ¦     +- Makefile
  ¦     +- KL1コンパイラのソースプログラム(KL1)
  ¦     +- KL1コンパイラのセーブ形式ファイル
  +- compiler_pl/
  ¦     +- Makefile
  ¦     +- KL1コンパイラのソースプログラム(Prolog)
  +- utl/
  ¦     +- Makefile
  ¦     +- ユーティリティ・プログラム(C)
  +- testpro/    (KL1で書いたテスト/ベンチマーク用プログラム)
  +- pdsscmp.org (Prolog版のKL1コンパイラを実行するコマンドプロシジャ)
  +- startup.org (PDSSの立ち上げ時に使用するスタートアップ・ファイル)
```

２．ＰＤＳＳのＭａｋｅ

新しいマシンに移植した場合、ＰＤＳＳのうちＣ言語で記述した部分は再コンパイルする必要があります。また、ＰＤＳＳを置くディレクトリの位置が違っている場合には、プログラム中に書かれた(フルパスの)ファイル名を変更する必要があります。ＰＤＳＳにはこれらのコンパイルやファイル名の書き換えの為のMakefileが定義されており、ＰＤＳＳのトップレベルのディレクトリにcdし、以下のようにして実行することができます。このMakefileは各サブ・ディレクトリについてもmakeするように定義されているので、トップレベルで１回行うだけで全ディレクトリがmakeされます。

```
make ALL
```

なお、Ｃのコンパイル・オプション等を変更したい場合には emulator/Makefile

等を変更してからmakeします。また、PrologやGNU-Emacsのバージョンが違う場合には compiler_pl/Makefile, emacs/Makefile を変更する必要があります。
新しいマシンに移植した場合でもＫＬ１で記述した部分は再コンパイルの必要はありません。どのマシンでもそのまま使えます。

３．各Makefileの説明

(1) Makefile

これはトップレベルのMakefileでサブ・ディレクトリのmakeと、pdsscmp, startupのmakeを行う。

COMMANDS_DIR	`make install' でコマンドファイルをコピーする先のディレクトリ名を指定する。
EMACSLIB_DIR	`make install' でEmacsライブラリをコピーする先のディレクトリ名を指定する。
make ALL	新しいマシンに移植した場合に再コンパイル等を行う。
make all	部分的に変更した時に再コンパイル等を行う。
make install	コマンドファイルやEmacsライブラリをパブリックなディレクトリにコピーする。
make clean	作業用に生成したファイルを削除する。

(2) emulator/Makefile

これはＣ言語で記述したエミュレータのmakeを行う。またソース・プログラムに書かれたフル・パスのファイル名を書き換える。

VERSION	エミュレータの版数を指定する。
CFLAGS	Ｃのコンパイル・オプションを指定する。
make ALL	新しいマシンに移植した場合に再コンパイル等を行う。
make all	部分的に変更した時に再コンパイル等を行う。
make clean	作業用に生成したファイルを削除する。

(3) runtime/Makefile

これはＫＬ１で書かれたエミュレータのランタイム・サポート・ルーチンのmakeを行う。このときＫＬ１プログラムのコンパイルに `pdsscmp' を使用するので、それが使えるようになっていなければならない。

make all	部分的に変更した時に再コンパイル等を行う。

(4) mpimos/Makefile

これはＫＬ１で書かれたＭｉｃｒｏＰＩＭＯＳのmakeを行う。このときＫＬ１プログラムのコンパイルに `pdsscmp' を使用するので、それが使えるようになっていなければならない。

make all	部分的に変更した時に再コンパイル等を行う。

(5) compiler/Makefile

これはＫＬ１で書かれたＫＬ１コンパイラのmakeを行う。このときＫＬ１プロ

グラムのコンパイルに `pdsscmp' を使用するので、それが使えるようになっていなければならない。

make all	部分的に変更した時に再コンパイル等を行う。

(6) compiler_pl/Makefile

これはPrologで書かれたＫＬ１コンパイラのmakeを行う。これはSICStus Prolog用に作られているので、他のPrologでは変更しなければならない。

PROLOG	Prologを起動するコマンド名を指定する。
CMDCMP	ＫＬ１コンパイラをコンパイルし、セーブ形式のファイルを作る為のゴール列を指定する。
CMDTRN	トランスレータをコンパイルし、セーブ形式のファイルを作る為のゴール列を指定する。
make ALL	新しいマシンに移植した場合に再コンパイル等を行う。
make all	部分的に変更した時に再コンパイル等を行う。

(7) utl/Makefile

これはＣ言語で記述したユーティリティ・プログラムのmakeを行う。

make ALL	新しいマシンに移植した場合に再コンパイル等を行う。
make all	部分的に変更した時に再コンパイル等を行う。

(8) emacs/Makefile

これはＰＤＳＳをサポートする為に使われるGNU-Emacsのライブラリのmakeを行う。またライブラリ中に書かれたフル・パスのファイル名を書き換える。

EMACS	ライブラリをコンパイルする為に使うGNU-Emacsのコマンド名を指定する。
make ALL	新しいマシンに移植した場合に再コンパイル等を行う。
make all	部分的に変更した時に再コンパイル等を行う。

【付録－２】デバイス・ストリーム

ＰＤＳＳでは種々のデバイス（後で定義する）はデバイス・ストリームとして提供され，プログラマはこの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。デバイスには現在，ウインドウ（ディスプレイとキーボード），ファイル，メモリ（コード領域）がある。それぞれの意味と機能は以下で説明する。

１．入出力用のデバイス・ストリーム

ＰＤＳＳでは入出力用のデバイス・ストリームとしてウインドウとファイルを提供する。プログラマはこの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_io_device で定義されている。
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 fep#xxxx というマクロ表現は使えないので 'fep#xxxx' というアトムで代用してある。

１．１ 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タが起動した後で１回だけ呼び出すことができる。２回目以降の呼出しはエクセプション "Device called twice" となる。

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

b) pdss_io_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。

１．２ デバイス・コマンド

(1) ウインドウ・デバイス
ウインドウ・デバイスは GNU-Emacs 上でマルチ・ウイ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）であるウインドウをオープンし，そのウイ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インドウの数は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。オープンしたウインドウ・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イ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(8bit-string) 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。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。

1．3 コマンド

(1) 共通の制御用コマンド

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。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 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ドはしたくない時に使われる。

(2) 共通の入出力コマンド

a) 入力用コマンド

ウイ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出来るコマンド。

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。

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。

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。ウイ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。

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	:: int(Integer)
ストリング	:: string(String)
ファンクタ	:: open(Atom)
特殊文字	:: 特殊文字を印字名とするアトム
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。

b) 出力用コマンド
　ウイ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出来るコマンド。

putc(Char, ^Status, Cdr) ::
　Char（整数）で示されるコードの１文字を書き出す。Status には 'fep#normal' がユニファイされる。

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。

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。

(3) ウインドウ用制御コマンド

close(^Status) ::
ウインドウを閉じる。 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) ::
ウイ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) ::
ウイ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イ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) ::
ウイ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) ::
　ウイ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。

(4) ファイル用制御コマンド

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' を返す。
　そうでない時には 'fep#no' を返す。

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。

(5) ディレクトリ用制御コマンド

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(^FileName, ^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。

delete(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。

２．メモリ（コード領域）

コードを管理しているデバイスである。（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。）

a) compile(ListOfInstructions, ^Module, ^Status)
ListOfInstructions は KL1-B の命令列を表わすストリング。
与えられた命令列から内部コードを生成しコードへのポインタを Module とユニファイする。

b) enter_module(ModuleName, Module, ^Status)
Module で示されたコードをモジュール名 ModuleName でコード・テーブルに登録する。Status には success, cannot_open_file, memory_limit のいずれかがユニファイされる。

c) load_one_module(^ModuleName, FileName, ^Status)
FileName で指定されたファイルから ModuleNameで指定されたモジュールをロードする。Status には success, cannot_open_file, memory_limit のいずれかがユニファイされる。

d) save_one_module(ModuleName, FileName, ^Status)
FileName で指定されたファイルに ModuleName 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。

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

f) trace_module(ModuleName, Mode, ^Status)
ModuleName で示されたモジュールのトレース・モードを Mode にセットする。Status には success, module_not_found, undefined_mode のいずれかがユニファイされる。

g) spy_predicate(ModuleName, PredicateName, Arity, Mode, ^Status)
ModuleName 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 にセットする。Status には success, module_not_found, predicate_not_found, undefined_mode のいずれかがユニファイされる。

【付録－３】例外コード

・Illegal Input :: 1
　組込み述語の入力引数に規定されている型の値以外のものが現われた。

・Undefined Module :: 2
　ロードされていないモジュールを参照しようとした。

・Undefined Predicate :: 3
　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。

・Incorrect Priority :: 4
　指定されたプライオリティが若園の上限／下限値の範囲にない。

・Failed :: 5
　ゴールの実行に際してどの候補節も選ばれなかった。

・Unify Failed :: 6
　ボディ部のユニフィケーションが失敗した。

・Device Called Twice :: 7
　デバイス・ストリームを２回以上取ろうとした。

【付録－４】Ｍｉｃｒｏ ＰＩＭＯＳなどで既に定義されているモジュール名

```
'Sho-en'
directory
file
window
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_io_device
mpimos_macro_expander
mpimos_module_pool
mpimos_opcode_table
mpimos_operator_manipulator
mpimos_parser
mpimos_parser
mpimos_task_monitor
mpimos_unparser
mpimos_utility
mpimos_varchk
mpimos_window
mpimos_window_device_driver
mpimos_window_manager
pdss_code_device
pdss_io_device
pdss_runtime_active_unify
pdss_runtime_debug
pdss_runtime_exception_handling
pdss_runtime_body_builtin
```

【付録－５】定義済みオペレーター一覧

優先順位，種別，名前の順に示す。

```
1200, xfx,      (:-)
1200, fx,       (:-)
1150, fx,       (module)
1150, fx,       (public)
1100, xfy,      (;)
1100, xfy,      (|)
1050, xfy,      (->)
1000, xfy,      (,)
 950, xfy,      (:)
 700, xfx,      (=)
 700, xfx,      (\=)
 700, xfx,      (=\=)
 700, xfx,      (=:=)
 700, xfx,      (==)
 700, xfx,      (<)
 700, xfx,      (>)
 700, xfx,      (=<)
 700, xfx,      (>=)
 700, xfx,      (:=)
 500, yfx,      (+)
 500, fx,       (+)
 500, yfx,      (-)
 500, fx,       (-)
 500, yfx,      (/\)
 500, yfx,      (\/)
 400, yfx,      (*)
 400, yfx,      (/)
 400, yfx,      (<<)
 400, yfx,      (>>)
 300, xfx,      (mod)
```

【付録－６】組込み述語

(1) タイプ・チェック

```
wait(X) :: G
atom(X) :: G
integer(X) :: G
list(X) :: G
vector(X, ^Size) :: G
vector(X, ^Size, ^Vector) :: B
string(X, ^Size, ^ElementSize) :: G
string(X, ^Size, ^ElementSize, ^String) :: B
atomic(X) :: G
unbound(X, Result) :: B
```

(2) 比較

```
less(Integer1, Integer2) :: G
not_less(Integer1, Integer2) :: G
greater(Integer1, Integer2) :: G
not_greater(Integer1, Integer2) :: G
equal(Integer1, Integer2) :: G
not_equal(Integer1, Integer2) :: G
not_unified(X, Y) :: G
```

次のオペレータが使える。

<, =<, >, >=, =:=, =\=, \=

(3) 算術演算

```
add(Integer1, Integer2, ^Integer3) :: GB
subtract(Integer1, Integer2, ^Integer3) :: GB
multiply(Integer1, Integer2, ^Integer3) :: GB
divide(Integer1, Integer2, ^Integer3) :: GB
modulo(Integer1, Integer2, ^Integer3) :: GB
shift_left(Integer, ShiftWidth, ^NewInteger) :: GB
shift_right(Integer, ShiftWidth, ^NewInteger) :: GB
or(Integer1, Integer2, ^NewInteger) :: GB
and(Integer1, Integer2, ^NewInteger) :: GB
xor(Integer1, Integer2, ^NewInteger) :: GB
```

:= と次のオペレータを用いて記述してもよい。

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

(4) ベクタ関係

```
new_vector(^Vector, Size) :: B
vector_element(Vector,Position,^Element) :: G
set_vector_element(Vector,Position,
                   ^OldElement,NewElement,^NewVector) :: B
```

(5) アトム／ストリング関係

```
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, ^String) :: B
symbol_name(Atom, ^String) :: B
make_symbol(String, ^Atom) :: B
```

(6) 高階機能

```
apply(ModuleName, PredicateName, Args) :: B
```

(7) ストリーム・サポート

```
merge(^In, ^Out) :: B
merge_in(In1, In2, ^In) :: B
```

(8) 特殊入出力

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

(9) その他

```
raise(Tag, Type, Info) :: B
hash(X, Width, ^Value) :: B
```

【付録－７】ＥＭＡＣＳライブラリ

ＰＤＳＳで提供しているＥＭＡＣＳライブラリには、(1) ＫＬ１のプログラムを編集する場合の kl1-mode と、(2) 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

C-c C-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。

C-c C-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。

C-c C-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。
C-c C-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コンパイルする場合に使用する。このため、*.asm ファイルは削除しない方が良いと思われる。

＜注１＞
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。

＜注２＞
kl1-mode で用いているコンパイラは SICStus Prolog 上のコンパイラである。

(2) PDSS-mode

・ウィンドウ／バッファ操作

M-/ :: 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 shell のコマンドの再実行に使う。

C-c C-y :: 直前に入力した文字列を再表示する。

C-c k :: そのバッファ内のテキストを全て削除する。

C-c C-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。

C-c C-b :: ＰＤＳＳモードのバッファのバッファ・メミューを表示する。

C-x k　:: 通常は Kill-Buffer であるが、ＰＤＳＳのプロセスが実行中は Kill-Buffer を行う前に警告を出す。

・ＫＬ１プログラム制御

C-c C-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。

C-c C-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。

・エミュレータ制御

C-c !　:: ガベージコレクションの要求。

C-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。

C-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

C-c C-p :: そのウィンドウにＰＤＳＳモードのバッファを表示する。ＰＤＳＳのバッファ郡はサーキュラリストになっており、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。

C-c p　:: C-c C-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－８】Ｐｒｏｌｏｇ上のコンパイラの使用方法

Prolog版のコンパイラは以下のコマンド・プロシジャにより使用できます。これを使うことによりＰＤＳＳでロードできるファイルが作られます。

コマンド：

```
~pdss/release/pdsscmp   [ オプション ]  ファイル名 ...
```

オプション：

```
+i / -i ..... インデキシングのコードを出す/出さない。デフォルトでは出さない。(将来はインデキシング有りに変更する予定)
+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。
-d=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:

(1) pdsscmp append.kl1 queen.kl1 または
pdsscmp append queen
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。

(2) pdsscmp append.kl1 queen.asm
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。

(3) pdsscmp -d=object source/*.kl1
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 -d で指定したディレクトリ object の下に作られる。

【付録－９】サンプル・プログラム

```
:- module sample.
:- public primes/2, primes/1.

primes(N, PL) :- true | gen(2, N, NL), sift(NL, PL).

primes(N) :- true |
    gen(2, N, NL), sift(NL, PL),
    window:create([show|Window], "sample"),
    outconv(PL, Window).

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.
```