TM-0958(I)

PDSS PIMOS Development Support System

平野　喜芳、瀧　和男

September, 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

P D S S

PIMOS Development Support System

(Version 2.52)

September 14 1990

Institute for New Generation Computer Technology

First Laboratory

P D S S

PIMOS Development Support System

(Version 2.52)

P D S S

PIMOS Development Support System

(Version 2.52)

Part-I

September 14 1990

Institute for New Generation Computer Technology

First Laboratory

P D S S

PIMOS Development Support System

(Version 2.52)

Table of Contents

I – 1 PDSS Manual

(in English)

PDSS Manual

(Version 2.52e)

Dec 26 1989

Institute for New Generation Computer Technology

Fourth Laboratory

Acknowledgment

This manual was translated from Japanese version by Dr. Kouichi Wada, a lecturer of Tsukuba university, Dr. Masaki Kohata, a lecturer of Okayama science university, and Dr. Daniel Dure, a visiting researcher of ICOT from France. Translation was done in March, 1989.

Contents

1 What is PDSS

PDSS, which stands for PIMOS Development Support System, is a KL1 system to develop the PIMOS. PDSS is widely compatible with the KL1 system found on Multi-PSI V2, besides implementation details, execution speed, etc. The main differences are enumerated below.

- Some of the functions implemented through software (e.g atom management) are treated by the compiler. Some atom related operations are available as built-in predicates.
- Code management is done by compiler.
- The only resource known by PDSS is the number of performed reductions.
- The I/O device stream has a different form.
- Because PDSS is a single processor system, there is no processor pointing function for process dispersion.

Another function of PDSS is to provide tools for the development of parallel programs. To this end, PDSS has been written in a style which ensures portability and it will be installed onto various UNIX [1] systems. We tried to build PDSS as a handy development tool. We expect it to be improved along the development of PIMOS.

PDSS consists mainly of two parts : one is the language processing system which executes KL1 and the other is the user interface system, called Micro PIMOS. Micro PIMOS is a single user, single task operating system which provides I/O and code management functions to its user. Its description is held in chapter 3. Figure 1 shows PDSS configuration.

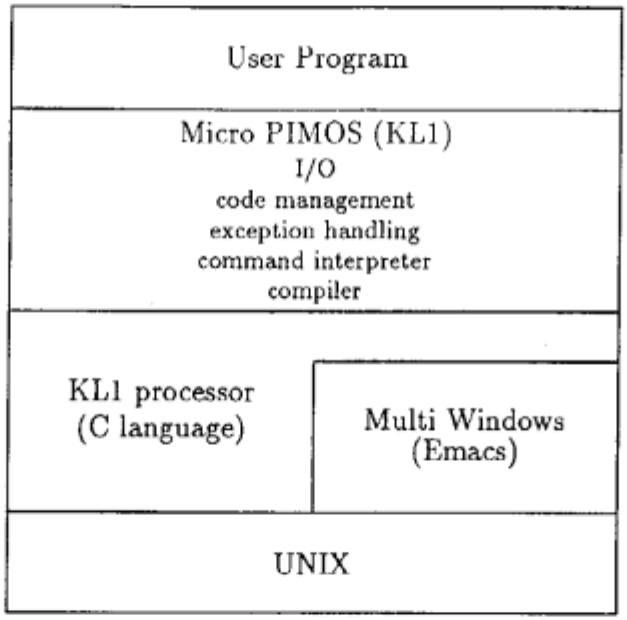

Figure 1: PDSS configuration

In figure 1, we see that a multi-window environment is provided through the GNU-Emacs full-screen editor. Its library has been written in Emacs-LISP.

In PDSS, I/O and code magement functions use a special built-in stream, called device stream. Specifications of this stream can be found in Appendix-1 and Appendix-2. Anyway, the average user doesn't have to use device stream directly, as most necessary facilities are provided in Micro-PIMOS libraries.

[1] UNIX is a trademark of Bell Laboratories.

2 KL1 Language Specification

The language specification of the KL1 dialect executable on PDSS lies in this chapter. Note that there may be differences between this specification and ones found on other systems, such as Multi-PSI V2.

2.1 Outline

KL1 is a language based on GHC (Guarded Horn Clauses) which moreover embeds some extensions related to OS description, modular programming, etc. KL1 also has some restrictions, due to implementation limitations. Its main characteristics are now described :

Sequentiality of guard

Unification of head parameters and execution of guard goals are performed sequentially, from left to right. In the following example, suspension occurs until variable X is instantiated. Note that the following predicate does not fail.

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

Guard restrictions

Only built-in predicates can be used within the guard. These predicates are described in section 2.6.

Equality of variables

Equality of unbound variables is not checked in the guard part. Suspension occurs in the following program, until variables X and Y are instantiated, independently from the execution order of the goals of the topmost clause.

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

Module functionality

Clustering clauses in several modules allows modular compilation and debugging. In the current version, to each file corresponds a unique module.

Sho-en

An original functional unit, called Sho-en, has been introduced. It is possible to control the execution priority and resource allocation of each Sho-en. OS itself is constructed as such a Sho-en.

Exception handling

Handling of exceptions occuring during the execution of a program is described in KL1, using Sho-en and second-order predicates.

Failure handling

All failures are considered as exceptions within KL1 and execution of a program can be resumed using exception handling facilities.

2.2 Sho-en

A Sho-en is the minimum unit of resource management, priority management and exception handling which exists in the language. Two streams, called control and report streams, are connected to each Sho-en. The control stream is used to control the Sho-en, and can carry various commands. The report stream carries information and requests coming from the Sho-en. Users of Sho-en can handle exceptions if they write programs interpreting the information from the report stream.

Resource management functions

The resource managed by PDSS is the number of performed reductions. It can be seen as a rough measure of the computing time and memory usage. For all goals which belong to the same Sho-en, it is possible to

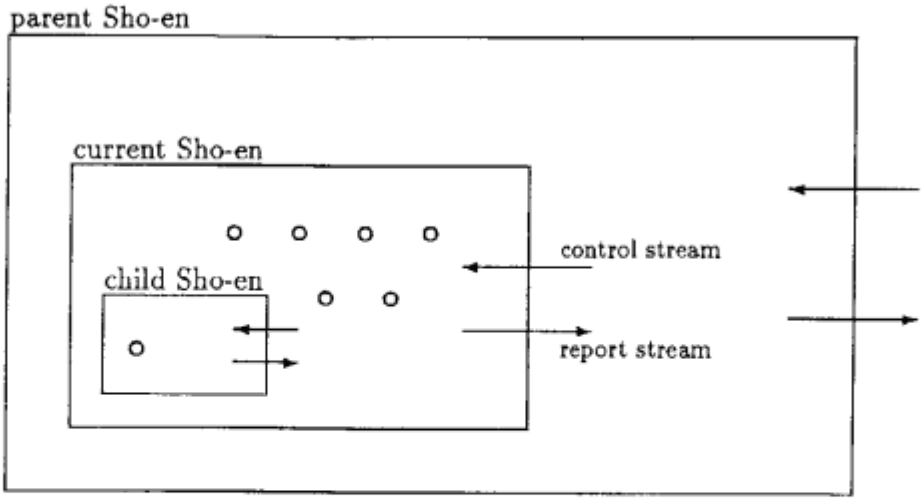

Figure 2: An instant picture of some Sho-ens

specify the maximum number of reductions. By default, the system assigns the maximum possible number of reductions. When the Sho-en is generated, i.e when it starts, this amount (or the default amount) is attributed. When reduction allocation is exhausted during the course of execution, an exception `resource_low` is inserted in the report stream of the Sho-en. It is possible to increase the reduction resource via an `add_resource(R)` command in the control stream, as explained later.

The resource comsumption control is performed in a discrete manner : independently from the maximum number of allowed reductions, there is a system dependent reduction granularity according to which control is exerted. Typically, a few thousands reductions. Resource control can be seen as a recursive allocation process : when a Sho-en starts, it is allocated, say, 2000 reductions. When this number is exhausted, a 2000 reductions resource is subtracted from the parent Sho-en and added to the current Sho-en. This may trigger a recursive process, during which reduction allocation is eventually done at the debts of some grand-father Sho-en. During this process, and only at this time, the maximum allocation limit is checked.

Priority management

Another function of the Sho-en is priority management. Each Sho-en holds a record of upper and lower priority bounds, for inner goals. Goals cannot be executed with a priority beyond upper bound and below lower bound. Priority specification is described in section 2.3.

2.2.1 Sho-en generation

Sho-en generation is performed using the "Sho-en" system module, which contains the predicates execute/7 (and an old format execute/8 also remains). Below, `code` is a three elements vector : {`module-name`, `predicate-name`, `number-of-args`}. `argument` is a vector with arguments of the goal. (In Multi-PSI V2, `code` data type is used for the `code` argument.)

```
execute(code, argument, minimum-priority,
        maximum-priority, tag, control, report)

execute(module-name, predicate-name, argument, minimum-priority,
        maximum-priority, tag, control, report)
```

Above, `minimum-priority` holds the value used to calculate the lower limit of priority bounds within which goals are executed. It is an integer which ranges from 0 to 4096 and specifies what degree the lower limit is made lower. When it is 0, it is the same as the lower limit of the parent Sho-en, and when it is 4096 , it is the same priority of the goal:execute. `Maximum-priority` holds the value used to calculate the upper limit. It is also an integer which ranges from 0 to 4096, and sepecifies what degree the upper limit is made upper. When it is 0, it is the same priority of the goal:execute, and when it is 4096, it is the same as the upper limit of the parent Sho-en. That is shown in Figure3. `Tag` is a bit mask used to filter the exceptions received by the Sho-en.

Tag is described in Appendix-8 fully. Control stream is unified with `control`, and report stream is unified with `report`. The initial state of the generated Sho-en is suspend, and the allowed reduction count is not set.

```
≪ ex ≫  'Sho-en':execute({primes,do,3},{1,300,PRIMES},0,2,-1,CONTROL,REPORT)
```

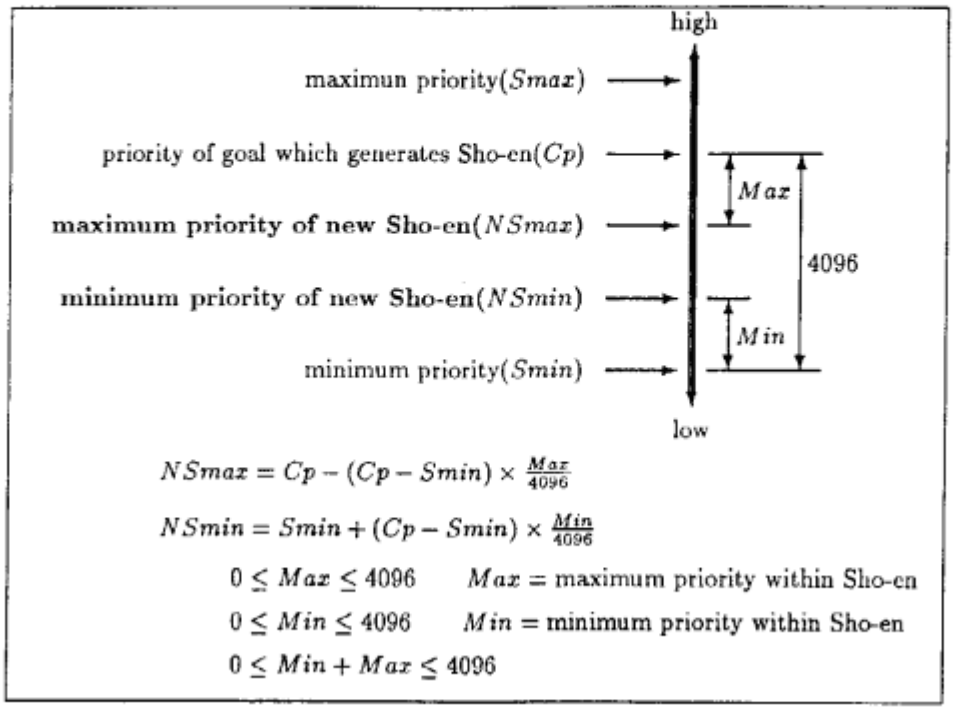

Figure 3: Calculation of Sho-en Priority

2.2.2 Control stream

Below are the commands which can be inserted in the control stream. When the control stream is closed, the Sho-en is abandoned. Conversely, if the user does never close the stream, execution of Sho-en itself never stops.

start
Activates goal execution in the Sho-en.

stop
Suspends goal execution. Previous command causes execution to resume.

abort
Aborts goal execution once and for all. In particular, start command cannot resume execution.

add_resource(Reduction)
Adds `Reduction` to the current number of allowed reductions.

allow_resource_report
This command is an answer to the exception `resource_low`. This exception cannot be reported again until this command is inserted.

statistics
Asks statistics about the Sho-en. Information is inserted in the report stream.

2.2.3 Report stream

The following information can be found in the report stream.

Acknowledgment messages to control stream commands :

Here are the responses to commands put in the control stream.

started
`Start` message has been received.

stopped
`Stop` message has been received.

aborted
`Abort` message has been received.

resource_added
`Add_ressource` message has been received.

resource_report_allowed
`Allow_resource_report` message has been received. Exception **`resource_low`** can be reported again after this message.

statistics_started
`Statistics` message received. The statistic information itself is reported once collected.

Status information

Here is the information reported whenever Sho-en status changes.

terminated
Execution of Sho-en has finished. If **`abort`** had been sent previously, this message indicates that the execution has been aborted. Otherwise, it indicates success of all goals.

resource_low
The number of performed reductions is close to the maximum allowed amount, or this amount is not sufficient. When this exception occurs, Sho-en state becomes **`suspend`**. No other **`resource_low`** report can occur before that **`allow_resource_report`** is inserted in the control stream.

Statistic information

Here, we get statistic information about the Sho-en, whenever collection has been done.

statistics(Info)
Unifies the statistic information with **`Info`**, which is one-element-vector , indicating the number of reductions performed. This number includes reductions performed by children Sho-ens.

Exception information

Here is the description of exceptions which can be reported by a Sho-en. Excluding deadlock, they specify the handling processes for the exceptions. If an exception condition is detected by PDSS, **`apply(NewCode, NewArgv)`** is generated within the Sho-en to handle the exception. Then, system waits for the unification of this goal with **`NewCode`** and **`NewArgv`**. The predicate specified with **`NewCode`** must be declared as public. When **`NewCode`** is unified with $\square$, no goal is executed any more.

exception(ExcpCode, Info, NewCode, NewArgv)
Exception occured in Sho-en. **`ExcpCode`** is a positive integer which indicates the type of exception. **`Info`** is an information about the exception, and it is unique corresponding to the type of exception. Each exception code is described in Appendix-7. The new code and arguments of the goal chosen by the user, in place of the failing goal which caused exception, should be unified with **`NewCode`** and **`NewArgv`**. **`ExcpCode`** and `Info` are described as follows. Below `Caller` is the code of the predicate which calls a built-in predicate. **`OpCode`** is the operation code of the built-in predicate, **`Argv`** is the argument vector, and `Code` is a three elements vector: {module-name, preicate-name, number-of-arguments}.

ExcpCode	meaning	:: Info	definition
0	Illegal Input Type	:: {0, Caller, OpCode, Pos, Argv}	Pos is the position of
		::	the invalid argument($1 \sim 7$)
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 is the invalid input data for merger
		::	FMI is the input stream to merger
9	Reduction Failure	:: {0, Code, Argv}	
10	Unification Failure	:: {0, X, Y}	X and Y are the terms which
		::	causes failure in body part unification
12	Raised	:: {0, RType, RInfo}	RType,RInfo are the terms
		::	given by the built-in predicate:raise/3
16	Incorrect Priority	:: {0, Caller, OpCode, Argv}	
17	Module Not Found	:: {0, Code, Argv}	
18	Predicate Not Found	:: {0, Code, Argv}	

deadlock(ExcpCode, Info)

Deadlock has been detected in Sho-en. `ExcpCode` is an integer indicating that the type of exception is deadlock. `Info` is an information about the exception, and now its format is shown below. `DGoal` is the code of the predicate which causes deadlock, or □ (in the case that deadlock is detected in global garbage collection). `DType` is an integer indicating deadlock type(see chapter 6). `GoalsList` is the list of codes of goals which are deadlock roots.

ExcpCode	meaning	:: Info	definition
11	Deadlock	:: {0, DGoal, DType, GoalsList}	described above

2.3 Priority

In KL1, it is possible to specify the priority at which each goal is executed. There are logical and physical priorities, and each goal can have its own logical priority. There are different levels of physical priority in the system, and the scheduler converts logical priority into physical priority when it connects goals to the goal stack. (As physical priority is less accurate than logical priority, user should not expect scheduling to reflect exactly the logical priority.) Upper/lower limits of priority in the Sho-en are also logical.

Priority of a goal is specified relatively to its parent goal, or relatively to the Sho-en it belongs to. The former method is called "relative self specification in the belonging Sho-en" and the later is called "rate specification in the belonging Sho-en".

Rate specification in the belonging Sho-en

Goal priority is specified by a value relative to the upper/lower limit of the belonging Sho-en. It is written as follows :

Goal @ priority(*, Rate)

In this case, the goal priority is computed shown in Figure 4 :

Relative self specification in the belonging Sho-en

Goal priority is specified by a value relative to the logical priority of the parent goal. This priority cannot exceed the upper/lower limit of the Sho-en.

Goal @ priority($, Rate)

This time, priority is computed accoding to the sign of Rate as shown in Figure 5 (in case of plus) and Figure 6 (in case of minus).

2.4 Syntax

Differences between GHC and KL1 are described here. Main differences are concerned with :

- Module definition
- Clause ordering
- Priority specification

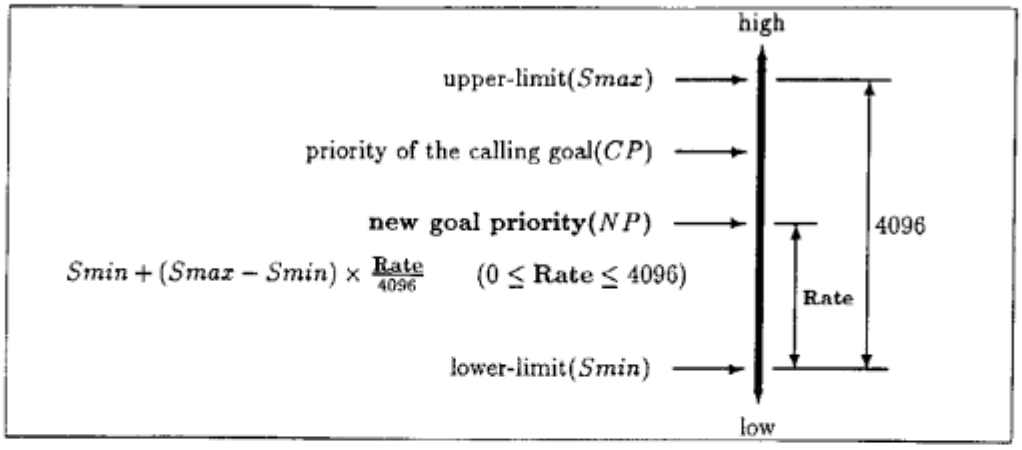

Figure 4: computation of the goal priority by rate specification in the belonging Sho-en

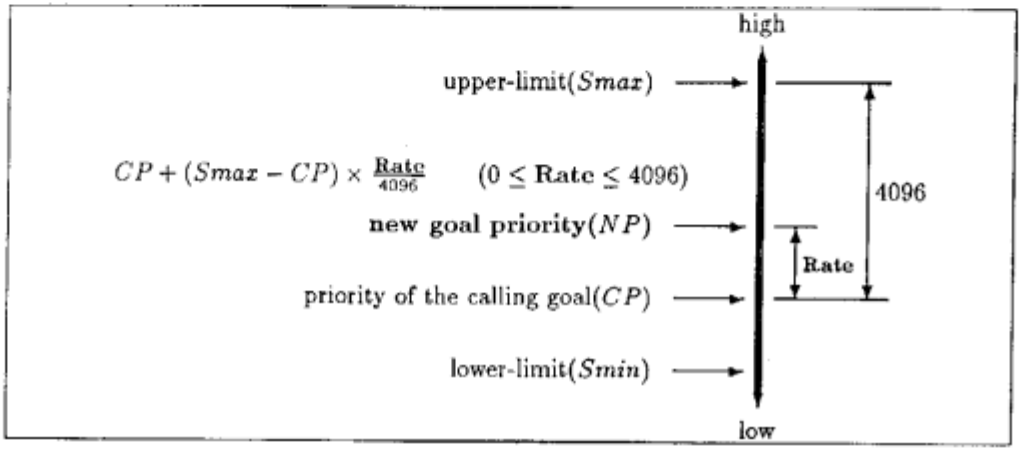

Figure 5: computation of the goal priority by relative self specification in the belonging Sho-en(plus)

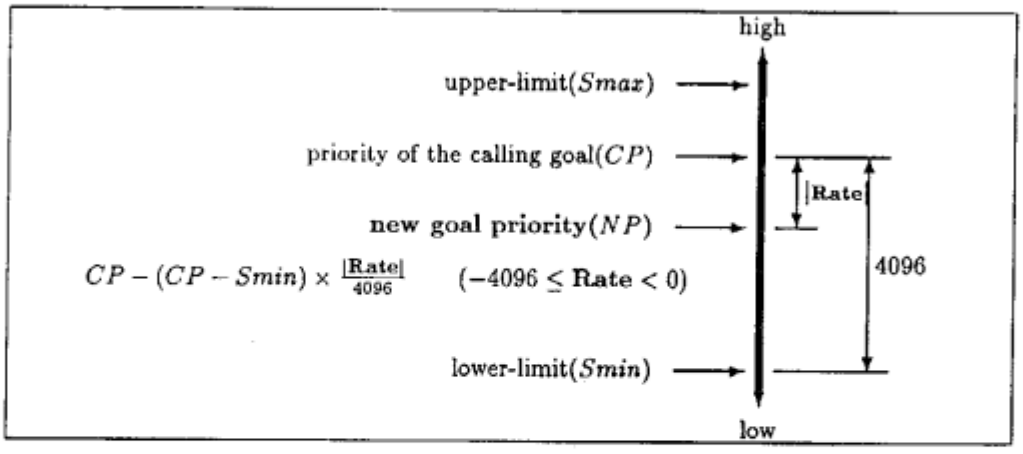

Figure 6: computation of the goal priority by relative self specification in the belonging Sho-en(minus)

- Macros description

The macros are described later in this document.

2.4.1 Module definition

The following is a module definition :

```
:- module module-name.
```

This declaration must appear at the head of any module. Furthermore, any predicate defined in this module but used outside of it should be declared as follows :

```
:- public predicate-name/number-of-arguments, ....
```

Note that predicates executed by a built-in predicate `apply` and those specified at Sho-en generation must be declared public. Multiple definitions of a predicate, spreading over several clauses, cannot be split by the definition of a different predicate. Doing otherwise causes display of the message "Assembler: Doubly defined label.".

A goal whose definition pertains to a different module can be used as indicated below :

```
module-name : goal-name
```

Conversely, any goal without "`module-name :` " is recognized as a goal called inside the module. That is, a goal name is efficient in each module, and the same predicate names can be used as different predicate definition as far as they are belong to different modules.

2.4.2 Clause ordering

Compilation of KL1 program goes through clause indexing, in order to maximize efficiency. This may in return change the order according to which clauses are selected for evaluation. If clause evaluation ordering is a necessity, the following statements should be used.

Scheduling order

The statement `alternatively` can be used to separate two sets of clauses, the first of which should be scheduled with a higher priority. However, if all clauses in the first set are suspended or fail, evaluation of clauses in the second set starts.

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

Evaluation order

The `otherwise` statement is more straightforward : clauses following it are evaluated only if all of the preceeding clauses failed. Suspension does not trigger anything in this case.

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

2.5 Data types

Here are the data types supported by PDSS. The system recognizes only these data types as "data", that is, only these data types can make sense for the system.

- Unbound variables ··· `A, A12, B, _abc, _`
  As Prolog, an unbound variable consists of charactors and numerals, and begins with a capital charactor or a underscore. Unbound variables of a same name in one clause are identified as the same. Note that isolated underscores are recognized as different each other.
- Atoms ··· `abc, 'ABC', :=, 'can''t'`
  As Prolog, an atom consists of charactors which begin with a small letter, or only of signs, or of charactors quoted with single quotation marks. To use a sigle quotation mark: ' as an element of an atom name, use two single quotation marks and quote the term with single quotation marks. Note that '$' is used by the system as a particular usage, so users had better not use it as an atom (or an element of it).

- Integers ··· `123, 16'ACE, 8'37, +3, -5`
  Usually an integer is expressed by the decimal notation, and ranges from −2147483648 to 2147483647. Then the sign of it is taken as a part of an integer, unless spaces do not follow it. When an integer in format x'y is used, the radix base x may vary between 2 and 36, with classical convention for figures. But the sign cannot be included in this expression. Note that x'y format produces a syntax error on the Prolog-based compiler, which makes sense for the reader of PDSS. The form `radix-base#number` should be used instead. Note that the `number` is expressed as a string(e.g. `16#"12AC"`).

- Floating Point ··· `1.23, 1.0e10, 3.0E-30, -2.0`
  Floating point is expressed by following format:
  `[sign] number` $^+$ `decimal point number`$^+$ `[e or E [sign of index number] number`$^+$ `]`
  Above, `[...]` expresses an option and `number`$^+$ expresses numeral(s). It is not allowed to include any space in this expression. As in an integer, the sign is taken as a part of an floating pont. PDSS supports a single-precision (32 bits) floating point number. It expresses a value by about seven places of decimal which ranges from -3.402823×10^{38} to 3.402823×10^{38}.
  To unify two floating point numbers, comparison of bit patterns of internal forms is used. Even if displayed two values seems to be unifiable, there are some possibilities of failing in unification. So, for general, unification of two floating point numbers should not be intended.

- Lists ··· `[1,2,3], [X|Y]`
  A list is expressed by [ ]. Car and Cdr can be expressed by using '|'.

- Vectors ··· `{1,2,{3,4}}, f(X), {}`
  A vector is a structure of one dimension arrays, possibly of 0 dimension. To express a vector, it is allowed to use { } or the functor format. For example, `f(a)` and `{f,a}` have the same structure.

- Strings ··· `"abc", "", """"`
  A string is expressed by charactors qouted with double qoutation marks: ". To use a double quotation mark: " as an element of a string, use two double quotation marks and quote the term with double quotation marks. The size of the string allowed by the system ranges from 1 bit to 32 bits. On PDSS, a string expressed by `"..."` is recognized as an 8 bits string. On Prolog-based compiler, `"..."` is not precisely distinguished from lists, because a string is expressed by a list of codes of charactors on Prolog. So, a macro expression of the form `string#"..."` should be used to generate a string. Here, strings, excluding 8 bits strings, are used only for internal forms, and cannot be used in source files as constants. Indeed, unification of two strings will success if the strings have identical length and contents.

2.6 Built-in predicates

We now give the list of the built-in KL1 primitives supported by PDSS. The following is an example of the format we use :

$$\underline{\text{vector(X, ^Size)}} :: \underline{\text{G}}$$

Call format — Valid location for occurence

In this case, **G** means that the predicate can appear in the guard of the clause. Some predicates can occur in the body, in which case the letter **B** is used. **GB** denotes predicates which can occur in both places. Besides, arguments with a ^ are outputs, whereas other arguments are inputs. One should take this into account, because binding an output argument with an already instantiated variable may cause suspension. Also, unification occuring in the guard is passive, whereas unification in the body can be active.

For some of the predicates described therein, input parameters should verify some domain constraints. Typically, to divide a number by 0 is not a very sound operation. If a domain constraint is not respected, depending on the predicate position, two different things may happen : if the predicate is used within the guard part of a clause, this clause fails. If the predicate is in the body, an exception occurs.

The system presents arithmetic macros for arithmetic operation, so it is not necessary to describe relevant built-in predicates. Macros are described in capter 2.7.

2.6.1 Type checking

wait(X) :: G
If `X` is unbound, suspension occurs. Otherwise, this predicate succeeds.

atom(X) :: G
If `X` is unbound, suspension occurs. If `X` is an atom, this predicate succeeds, otherwise it fails.

integer(X) :: G
If `X` is unbound, suspension occurs. If `X` is an integer, this predicate succeeds, otherwise it fails.

floating_point(X) :: G
If `X` is unbound, suspension occurs. If `X` is a floating point, this predicate succeeds, otherwise it fails.

list(X) :: G
If `X` is unbound, suspension occurs. If `X` is a list, this predicate succeeds, otherwise it fails.

vector(X) :: G
If `X` is unbound, suspension occurs. If `X` is a vector, this predicate succeeds, otherwise it fails.

string(X) :: G
If `X` is unbound, suspension occurs. If `X` is a string, this predicate succeeds, otherwise it fails.

unbound(X, ^Result) :: B
This primitive always succeeds. If `X` is unbound, `Result` is unified with a three-elements-vector `{PE, Addr, X}`. Here, `PE` is the number of PE which holds the variable `X` (on PDSS always unified with 0), and `Addr` is an address of the variable `X`. Conversely, if `X` is bound when this primitive is executed, `Result` is unified with `{X}`. This primitive never causes suspension.
≪ ! ≫The values of `PE` and `Addr` will be changed after garbage collection.

2.6.2 Diff

diff(X, Y) :: G
If `X` and `Y` are identified as not to be unifiable by comparison of the two terms, this predicate succeeds. Conversely if `X` and `Y` can be identified as completely to have the same structure, the predicate fails. Otherwise suspension occurs. Following macro can be used.

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

≪ ! ≫Comparison of the terms is made in depth first from left to right. If any unbound variables are found in comparison, comparison procedure is stopped and suspension occurs. If `X` and `Y` are structured terms, unifiability checking is limited in depth. If terms are unifiable in the depth limit, the predicate will fail, although a difference may exist deeper in the structures.

2.6.3 Arithmetic comparison (Integer)

equal(Integer1, Integer2) :: G
If `Integer1` or `Integer2` is unbound, suspension occurs. If both `Integer1` and `Integer2` are integers and are equal, this predicate succeeds. Otherwise it fails. Following macro can be used.

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

not_equal(Integer1, Integer2) :: G
If `Integer1` or `Integer2` is unbound, suspension occurs. If both `Integer1` and `Integer2` are integers and are not equal, this predicate succeeds. Otherwise it fails. Following macro can be used.

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

less_than(Integer1, Integer2) :: G
If `Integer1` or `Integer2` is unbound, suspension occurs. If both `Integer1` and `Integer2` are integers and the value of the former is less than that of the latter, this predicate succeeds. Otherwise it fails. Following macro can be used.

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

not_less_than(Integer1, Integer2) :: G

If Integer1 or Integer2 is unbound, suspension occurs. If both Integer1 and Integer2 are integers and the value of the former is larger than, or equal that of the latter, this predicate succeeds. Otherwise it fails. Following macro can be used.

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

2.6.4 Arithmetic operations (Integer)

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

If Integer1 or Integer2 is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The result of the addition is unified with NewInteger, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

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

If Integer1 or Integer2 is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The result of the subtraction is unified with NewInteger, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

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

If Integer1 or Integer2 is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The result of the multiplication is unified with NewInteger, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

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

If Integer1 or Integer2 is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The result of the division is unified with NewInteger. Then if overflow is detected, or if Interger2 is bound to 0, failure or exception occurs. Following macro can be used.

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

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

If Integer1 or Integer2 is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The result of the rest of the euclidian division is unified with NewInteger. Then if overflow is detected, or if Interger2 is bound to 0, failure or exception occurs. Following macro can be used.

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

minus(Integer, ^NewInteger) :: GB

If Integer is unbound, suspension occurs. If it is not an integer, failure or exception occurs. NewInteger is unified with Integer with sign exchanged, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

increment(Integer, ^NewInteger) :: GB

If Integer is unbound, suspension occurs. If it is not an integer, failure or exception occurs. The result to add 1 to Integer is unified with NewInteger, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

decrement(Integer, ^NewInteger) :: GB

If `Integer` is unbound, suspension occurs. If it is not an integer, failure or exception occurs. The result to subtract 1 from `Integer` is unified with `NewInteger`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

abs(Integer, ^NewInteger) :: GB

If `Integer` is unbound, suspension occurs. If it is not an integer, failure or exception occurs. The absolute value of `Integer` is unified with `NewInteger`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

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

If `Integer1` or `Integer2` is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The lesser value of two input parameters is unified with `NewInteger`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

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

If `Integer1` or `Integer2` is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The larger value of two input parameters is unified with `NewInteger`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

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

If `Integer1` or `Integer2` is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The result of a bitwise logic and operation is unified with `NewInteger`. Following macro can be used.

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

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

If `Integer1` or `Integer2` is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The result of a bitwise logic or operation is unified with `NewInteger`. Following macro can be used.

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

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

If `Integer1` or `Integer2` is unbound, suspension occurs. If terms are not integers, failure or exception occurs. The result of a bitwise logic exclusive or operation is unified with `NewInteger`. Following macro can be used.

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

complement(Integer, ^NewInteger) :: GB

If `Integer` is unbound, suspension occurs. If it is not an integer, failure or exception occurs. This unifies `NewInteger` with the 1's complement of `Integer`. This is equivalent to exclusive_or(-1, `Integer`, `NewInteger`). Following macro can be used.

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

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

If `Integer` is unbound, suspension occurs. If it is not an integer, failure or exception occurs. If `ShiftWidth` is unbound, suspension occurs. If it should be an integer in the range [0,31], failure or exception occurs. `NewInteger` is unified with the result of logic bitwise shift. Following macro can be used.

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

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

If `Integer` is unbound, suspension occurs. If it is not an integer, failure or exception occurs. If `ShiftWidth` is unbound, suspension occurs. If it should be an integer in the range [0,31], failure or exception occurs. `NewInteger` is unified with the result of right logic bitwise shift. Following macro can be used.

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

2.6.5 Arithmetic comparison (Floating point)

floating_point_equal(Float1,Float2) :: G

If `Float1` or `Float2` is unbound, suspension occurs. If both `Float1` and `Float2` are floating points and equal, this predicate succeeds. Otherwise it fails. Following macro can be used.

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

floating_point_not_equal(Float1, Float2) :: G

If `Float1` or `Float2` is unbound, suspension occurs. If both `Float1` and `Float2` are floating points and are not equal, this predicate succeeds. Otherwise it fails. Following macro can be used.

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

floating_point_less_than(Float1,Float2) :: G

If `Float1` or `Float2` is unbound, suspension occurs. If both `Float1` and `Float2` are floating points and the value of the former is less than that of the latter, this predicate succeeds. Otherwise it fails. Following macro can be used.

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

floating_point_not_less_than(Float1,Float2) :: G

If `Float1` or `Float2` is unbound, suspension occurs. If both `Float1` and `Float2` are floating points and the value of the former is larger than, or equal that of the latter, this predicate succeeds. Otherwise it fails. Following macro can be used.

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

2.6.6 Arithmetic operations (Floating point)

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

If `Float1` or `Float2` is unbound, suspension occurs. If terms are not floating points, failure or exception occurs. The result of the addition is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫On Multi-PSI V2, if overflow is detected, failure or exception will not occur, and `infinity` outputs into `NewFloat`.

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

If `Float1` or `Float2` is unbound, suspension occurs. If terms are not floating points, failure or exception occurs. The result of the subtraction is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫On Multi-PSI V2, if overflow is detected, failure or exception will not occur, and `infinity` outputs into `NewFloat`.

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

If `Float1` or `Float2` is unbound, suspension occurs. If terms are not floating points, failure or exception occurs. The result of the multiplication is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫On Multi-PSI V2, if overflow is detected, failure or exception will not occur, and `infinity` outputs into `NewFloat`.

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

If `Float1` or `Float2` is unbound, suspension occurs. If terms are not floating points, failure or exception occurs. The result of the division is unified with `NewFloat`. Then if overflow is detected, or if `Float2` is bound to 0, failure or exception occurs. Following macro can be used.

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

≪ ! ≫On Multi-PSI V2, if overflow is detected, failure or exception will not occur, and `infinity` outputs into `NewFloat`.

floating_point_minus(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. `NewFloat` is unified with `Float` with sign exchanged, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_abs(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. The absolute value of `Float` is unified with `NewFloat`, Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

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

If `Float1` or `Float2` is unbound, suspension occurs. If terms are not floating points, failure or exception occurs. The lesser value of two input parameters is unified with `NewFloat`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

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

If `Float1` or `Float2` is unbound, suspension occurs. If terms are not floating points, failure or exception occurs. The larger value of two input parameters is unified with `NewFloat`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_floor(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. The largest integer not greater than `Float` is unified with `NewFloat`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_sqrt(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. If `Float` is negative number, failure or exception occurs. The square root of `Float` is unified with `NewFloat`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_ln(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. If `Float` is less than 0.0, failure or exception occurs. The natural logarithm of `Float` is unified with `NewFloat`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_log(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. If `Float` is less than 0.0, failure or exception occurs. The base 10 logarithm of `Float` is unified with `NewFloat`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_exp(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. e is raised to `Float` power and the result is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

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

If `Float1` or `Float2` is unbound, suspension occurs. If terms are not floating points, failure or exception occurs. If `Float1` is not a negative number, and `Float2` is not an integer, failure or exception occurs. `Float1` is raised to `Float2` power and the result is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_sin(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. sin(`Float`) is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_cos(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. cos(`Float`) is unified with `NewFloat`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_tan(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. tan(`Float`) is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_asin(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. If `Float` should be in the range [-1.0,1.0], failure or exception occurs. arcsin(`Float`) is unified with `NewFloat`, Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_acos(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. If `Float` should be in the range [-1.0,1.0], failure or exception occurs. arccos(`Float`) is unified with `NewFloat`. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_atan(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. arctan(`Float`) is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

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

If `Float1` or `Float2` is unbound, suspension occurs. If terms are not floating points, failure or exception occurs. If `Float2` is 0.0, failure or exception occurs. arctan(`Float`/ `Float2`) is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_sinh(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. sinh(`Float`) is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_cosh(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. sinh(`Float`) is unified with `NewFloat`, then if overflow is detected, failure or exception occurs. Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

floating_point_tanh(Float, ^NewFloat) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. tanh(`Float`) is unified with `NewFloat`, Following macro can be used.

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

≪ ! ≫This is not supported on Multi-PSI V2.

2.6.7 Conversion (Integer - Floating Point)

floating_point_to_integer(Float, ^Integer) :: GB

If `Float` is unbound, suspension occurs. If it is not a floating point, failure or exception occurs. Otherwise this predicate converts `Float` into integer (The integer is returned the value of the least integer greater than or equal to) , and unifies it with `Integer`. Following macro can be used.

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

integer_to_floating_point(Integer, ^Float) :: GB

If `Integer` is unbound, suspension occurs. If it is not an integer, failure or exception occurs. Otherwise this predicate converts `Integer` into floating point, and unifies it with `Float`. Following macro can be used.

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

2.6.8 Vector predicates

vector(X, ^Size) :: G

If `X` is unbound, suspension occurs. If `X` is a vector, this predicate succeeds and `Size` is unified with the vector size. Otherwise the predicate fails.

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

If `X` is unbound, suspension occurs. If `X` is a vector, `Size` is unified with the vector size, and `NewVector` is unified with a copy of `X`. Otherwise exception occurs. This is useful to duplicate a vector and avoid inter-process references, which are the pain for the garbage collector.

new_vector(^Vector, Size) :: B

If `Size` is unbound, suspension occurs. If it is not positive or null, exception occurs. If the size of a vector freshly allocated with size `Size` is beyond the heap size, exception occurs. Otherwise this predicate unifies `Vector` with a freshly allocated vector, filled with zeros, which size is given by `Size`.

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

This predicate is used to extract one element from a vector. If `Vector` or `Position` is unbound, suspension occurs. If `Vector` is not a vector, the predicate fails. `Position` indicates the rank of the element, starting from 0. If `Position` is not a positive integer, or if it is beyond the size of the vector, the predicate fails. Otherwise `Element` is unified with the result of extraction.

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

This predicate is used to extract one element from a vector. If `Vector` or `Position` is unbound, suspension occurs. If `Vector` is not a vector, exception occurs. `Position` indicates the rank of the element, starting from 0. If `Position` is not a positive integer, or if it is beyond the size of the vector, exception occurs. Otherwise `Element` is unified with the result of extraction, and `NewVector` is unified with a copy of `Vector`. This is useful to avoid multiple references which could impair garbage collector operations.

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

This predicate is used to extract one element from a vector and replaces it with a new element, and makes a copy of the original vector. If any of `Vector` or `Position` or `NewElem` are unbound, suspension occurs. If `Vector` is not a vector, exception occurs. `Position` indicates the rank of the element, starting from 0. If `Position` is not a positive integer, or if it is beyond the size of the vector, exception occurs. Otherwise `OldElem` is unified with the result of extraction. Then the predicate makes a copy of `Vector` and replaces the element specified by `Position`, and unifies the modified vector with `NewVect`.

2.6.9 String predicates

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

If `X` is unbound, suspension occurs. If `X` is a string, `Size` is unified with the number of characters in the string and `ElementSize` is unified with the length of each character, expressed in bits. Otherwise the predicate fails.

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

If `X` is unbound, suspension occurs. If `X` is a string, `Size` is unified with the number of characters in the string and `ElementSize` is unified with the length of each character, expressed in bits. Then `NewString` is unified with a copy of the string. Otherwise exception occurs.

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

If `Size` or `ElementSize` is unbound, suspension occurs. `Size` specifies the length of the string. If `Size` is not a positive integer, exception occurs. `ElementSize` specifies how many bits contains each character. If `ElementSize` is an integer in the range [1,32], exception occurs. If the size of a string freshly created with size specified by `Size` and `ElementSize` is beyond the heap size, exception occurs. Otherwise the prediate unifies `String` with a freshly created string specified by `Size` and `ElementSize`. The string is filled with zeros.

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

This predicate is used to extract one character from a string. If `String` or `Position` is unbound, suspension occurs. If `String` is not a string, this prediacte fails. If `Position` is not a positive integer or if it is beyond the length of the string, the predicate fails. Otherwise `Element` is unified with the charactor(integer) specified by `Position`. `Position` of the first character is 0.

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

This predicate is used to extract one character from a string, and makes a copy of an original string. If `String` or `Position` is unbound, suspension occurs. If `String` is not a string, exception occcurs. If `Position` is not a positive integer or if it is beyond the length of the string, exception occurs. Otherwise `Element` is unified with the charactor(integer) specified by `Position`, and `NewString` is unified with a copy of `String`. `Position` of the first character is 0.

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

If `String` or `Position` or `NewElement` is unbound, suspension occurs. If `String` is not a string, exception occcurs. If `Position` is not a positive integer or if it is beyond the length of the string, exception occurs. If `NewElement` is not an integer or if it is beyond the width of one element, exception occurs. Otherwise this predicate makes a copy of `String` and replaces the element of `Position` with `NewElement`, and unifies the modified string with `NewString`.

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

If String or Position or Length is unbound, suspension occurs. If String is not a string, exception occcurs. If Position is not a positive integer or if it is beyond the length of the string, exception occurs. If Length is not a positive integer or if the length of Position + Length exceeds the length of String, exception occurs. Otherwise this predicate extracts a substring from the original string, starting from Position with length Length. Then SubString is unified with the result, and NewString is unified with the original one.

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

If String or Position or SubString is unbound, suspension occurs. If String is not a string, exception occcurs. If Position is not a positive integer or if it is beyond the length of the string, exception occurs. If the width of one element of SubString is not the same as that of String, exception occurs. If the length of SubString + Position exceeds the length of String, exception occurs. Otherwise this predicate substitutes element(s) of the String specified by Position for SubString, and NewString is unified with the result of this substitution.

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

If String1 or String2 is unbound, suspension occurs. If two input strings are not the same bitwise type, exception occurs. Otherwise this predicate unifies NewString with the result of concatenating String2 after String1.

2.6.10 Atom predicates

intern_atom(^Atom, String) :: B

If String is unbound, suspension occurs. If String is not a string of 8-bits characters, exception occurs. Otherwise this predicate transforms a string of 8-bits characters String into an atom, whose name matches the string contents. This atom is unified with Atom.

≪ ! ≫On Multi-PSI V2, this is not a built-in predicate but a function of operating system.

new_atom(^Atom) :: B

This predicate creates a new atom and unifies it with Atom. The atom has no printing name.

atom_name(Atom, ^String) :: B

If Atom is unbound, suspension occurs, and if it is not an atom, exception occurs. Otherwise String is unified with a 8-bits string which contains the name of the atom.

≪ ! ≫On Multi-PSI V2, this is not a built-in predicate but a function of operating system.

atom_number(Atom, ^Number) :: B

If Atom is unbound, suspension occurs, and if it is not an atom, exception occurs. Otherwise Number is unified with the atom number corresponding to Atom. The atom number is identically given by the system to each atom according to the order of creation of it.

2.6.11 Code predicates

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

If any of Mod or Pred or Arity are unbound, suspension occurs. If Mod or Pred is not an atom, or if Arity is not a positive integer, exception occurs. Otherwise Code is unified with the code specified by Mod, Pred, and Arity. If the module, specified by Mod, does not exist, or if the predicate is not found(i.e. it is not defined or is not be delared as public), Code is unified with an atom □.

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

If Code is unbound, suspension occurs. If Code is not a three-elements-vector, exception occurs. Any elements of Code are unbound, suspension occurs. The fist element of it is a module name, and if it is not an atom or if it does not exist, exception occurs. The second element is a predicate name, and if it is not an atom or if it is not found, exception occurs. The third element is a number of arguments of the predicate, and if it is not a positive integer, exception occurs. Otherwise this predicate unifies the module name with Mod, and the predicate name with Pred, and the number of arguments with Arity. Info is unified with an integer indicating whether the predicate is spied(1), or not(0).

2.6.12 Stream support

merge(In, ^Out) :: B

This primitive can be used to merge one or more input streams (In) and unify the result with Out. A vector of streams, if given as one of the input, is divided into its stream components. The following is a partial definition of this predicate :

```
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).
                  ...
```

2.6.13 Second order function

apply(Code, Args) :: B

If Code or Args is unbound, suspension occurs. If Code is not a three-elements-vector, exception occurs. Any elements of Code are unbound, suspension occurs. The fist element of it is a module name, and if it is not an atom or if it does not exist, exception occurs. The second element is a predicate name, and if it is not an atom or if it is not found, exception occurs. The third element is a number of arguments of the predicate, and if it is not a positive integer, exception occurs. If Args is not a vector or if the number of arguments of it is not identical with the third element of Code, exception occcurs. Otherwise this primitive calls the predicate specified by Code, with arguments specified by Args.

2.6.14 Special I/O functions

read_console(^Integer) :: G

This predicate unifies Integer with a number read from the console window.
≪ ! ≫The language processor is halted during this operation. This predicate is used mainly for debugging purposes.

display_console(X) :: G

This predicate displays the current value of X on the console window, even if this variable is unbound.

put_console(X) :: G

If X is an integer, this primitive puts the equivalent ASCII character on the console window. If X is an 8-bits character string, the string is put on the console window. If X is undefined, or has any other type, the predicate does nothing. Anyway no line feed or carriage return is added.

2.6.15 Other predicates

raise(Tag, Type, Info) :: B

If Tag is unbound, suspension occurs. If it is not a positive integer, exception occurs. A ground-term does not include any unbound variables. If Type is not a ground-term, suspension occurs. Otherwise this predicate causes Tag to be logically and-ed with the tag of all ancestor Sho-ens, starting from the current Sho-en until the top, recursively. This process stops as soon as the result of the and operation is not zero. Then, a message is inserted in the report stream of the referred Sho-en. This message is as follows (also see exception information in chapter 2.2) :

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

consume_resource(Red) :: B

If `Red` is unbound, suspension occurs. If it is not a positive integer, exception occurs. Otherwise this predicate emulates the consumption of computing resources, as if due to actual reductions. `Red` is the number which is added to the count of performed reductions. If this count exceeds the allowed maximum, the `resource_low` condition occurs.

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

If `X` is unbound, suspension occurs. Otherwise this predicate unifies `Value` with a hash code computed according to `X`, and also unifies `NewX` with `X`.

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

This predicate unifies `ProcessorNumber` with the processor number of the processor executing this predicate. `X` and `Y` are unified with the coordinates of this processor, depending on the topology of the connection network. On PDSS, which emulates execution by a single processor, `ProcessorNumber` is unified arbitrarily with 0, and `X` and `Y` are unified with 1.

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

This predicate unifies `CurrentPriority` with the priority of the goal executing this predicate. And it also unifies `ShoenMin` with the lower limit of priority in the Sho-en, `ShoenMax` with the upper limit of priority.

2.7 Macros

For ease of writing, several categories of macros have been introduced in KL1.

- Macros for the description of constants.
- Macros for arithmetic comparison.
- Macros for conditional branch.
- Macros for the declaration of implicit arguments.

In the current version, users cannot define their own macros.

2.7.1 Constant description macros

The following macros generate constant numbers.

`Base#"character-string"`

This macro generates an integer number, in the integer `Base` specified before the sharp sign. The base must be from 2 to 36. Figures can be taken in [0,9] and [a/A,z/Z], as most commonly.

`string#"character-string"`

This macro can be used to generate a string of default type. Within PDSS, default type is ASCII stored as 8 bits characters. On Multi-PSI V2, characters are taken within the JIS Kanji set, stored as 2 bytes characters.

`ascii#"character-string"`

This is useful to assert that the generated string is coded in ASCII, within one byte characters.

`#"character"`

This macro generates a character, using the default representation of the system. In this aspect, it is similar to the `string` macro introduced above. On Multi-PSI V2, characters are taken within the JIS Kanji set.

`c#"character"`

This macro asserts generation of an ASCII character, stored as a single byte.

`ascii#character-atom`

This has the same effect, but the character is entered as an atom, not between double quotes. (ex: `ascii#'['`).

`key#lf`

This macro generates a line feed in ASCII.

`key#cr`

This macro generates a carriage return in ASCII.

For Integer

Priority	Operator	Expanded pattern
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)

For Floating Point

Priority	Operator	Expanded pattern
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)

Table 1: Arithmetic Comparison Macros

2.7.2 Unification macros

`left-hand = right-hand`
This macro performs unification of left and right hands. It can be used in body and guard.

`left-hand \= right-hand`
This is equivalent to diff(`left-hand`, `right-hand`). This macro can be used only in guard part of a clause.

`left-hand := right-hand`
This macro unifies the left hand with the right hand, but if integer operation macros feature in the right hand, evaluation takes place. This macro, which can be used in guard or body, is similar to the "is" operator of Prolog.

`left-hand $:= right-hand`
This macro unifies the left hand with the right hand, but if floating point operation macros feature in the right hand, evaluation takes place. This macro, which can be used in guard or body, is similar to the "is" operator of Prolog.

2.7.3 Arithmetic comparison macros

Arithmetic comparison operators can be used in guard, in place of built-in predicates. But integer or floating point operation macros in both hands of the comparison are not evaluated when built-in predicates are used. The comparison operators are described in Table 1.

2.7.4 Arithmetic operation macros

Arithmetic macros are using `+,-,*,/`. The conversion of data types needs to describe it explicitly, because it is not performed automatically by the system. Expansion is done according to the following rules :

- The right hand of the `:=`, `$:=` macro
  The result of the evaluation of the right hand of the operator is unified with the left hand (as for the "is" operator of Prolog). `:=` is used for integer operation, and `$:=` for floating point operation.
- Both hands of comparison macros
  The both results of the evaluation are compared to each other. The operator without `$` is used for integer operation, and one with `$` is for floating point operation.
- The right hand of implicit argument macro `<=`, `$<=`
  The result of evaluation of the right hand side is unified with the argument specified by left hand side. `<=` is used for integer operation, and `$<=` is for floating point operation.

Priority	Type	Operator	Expansion pattern	Generated built-in predicate	
				in integer expression	in floating point expression
	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)
as a term		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)

Table 2: Arithmetic operation macros

- In the case that `~(expression)` or `$~(expression)` is used to explicitly require expansion of the expression.
  ex: `p(~(X+Y+1))` becomes `add(X,Y,A),add(A,1,B),p(B)`.

Macros embedding constants are evaluated during compilation.

The arithmetic operators are described in Table 2. The higher is the priority, the lower is the precedence. It is always possible to make a term by using ( ).

≪ ! ≫On Multi-PSI V2, supported floating point operators are only on the arithmetic operation.

When users need to constrain expansion of macros, it is used back quotes.

- "(term)
  All the expansion of macro in "term" are completely constrained.

- '(term)
  If "term" is a structure, the expansion of the top level of it is exclusively constrained, and at the deeper level macros are expanded.

2.7.5 Macros for implicit argument passing

It is very unconvenient to rewrite arguments which appear recurrently in the head of several clauses. To lighten this tedium, implicit argument support(through macros) has been provided.

Two kinds of parameter declarations are possible, depending on the scope which is desired. The first one is global, i.e applies to a module wholly, whereas the second is local to a part of a module :

```
:- implicit arg-name : type { , arg-name : type , ... }.

:- local_implicit arg-name : type { , arg-name : type , ... }.
```

Here, `arg-name` (atom) is the name of the implicit argument. Type can be : `shared`, `stream`, `oldnew` or `string`.

The global implicit declaration can appear only once in a module, right after the public declaration. Local declarations can appear several times in a module. Each time it appears, it invalidates the previous declaration. To suppress the usage of all implicit arguments, use the following :

```
:- local_implicit.
```

The name space of local and global arguments are the same, so that different names have to be used.

Using `-->` in place of `:-`, means that a predicate uses implicit arguments. They are inserted in the predicate arguments list, before arguments explicitly given by the user. Exact order is as follows :

1. Global arguments
2. Local arguments
3. explicit arguments (order is not changed)

≪ ex ≫

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

p(X) --> true | q(X), r.
   %% Here, a and b are added to the argument list.
    ...

:- local_implicit d:oldnew.
   %% At this point a,b and d are added.
    ...

:- local_implicit d:shared, e:stream.
   %% From this point, a,b,d and e are added.
   %% The type of d has changed, from oldnew to shared.
    ...

:- local_implicit.
   %% From this point, only the global args. a and b will be added.
    ...
```

To access a global argument in a clause, `&` must be put before the argument. To update a value (or unify with some value), the operator `<=` or `$<=` or `<<=` is used. If the argument is a string or a vector, use :

```
&arg-name(position)
```

to update or access one of its elements. The first element has position 1. The following is a presentation of each type, with some examples.

shared argument type

If no update of a shared argument occurs within a given clause, all goals of the clause share the same instance of the argument. This is illustrated in example a) below. If the value has to be updated in the clause, use the following syntax :

```
&arg-name  <= new-value
&arg-name $<= new-value
```

The new value is effective after update. That means that the respective position of update statement and goals in a clause determines whether the old argument value or the new argument value is used. This is illustrated in examples b) to d).

```
≪ ex ≫ definition:  :- implicit counter:shared.

a) before expansion:  p --> true | q, r.
   after expansion:   p(Cnt) :- true | q(Cnt), r(Cnt).

b) before:  p --> true | &counter <= &counter + 1, q.
   after:   p(Cnt) :- true | add(Cnt,1,Cnt1), q(Cnt1).

c) before:  p --> true | &counter <= &counter+1, &counter <= &counter*2, q.
   after:   p(Cnt) :- true | add(Cnt,1,Cnt1), multiply(Cnt1,2,Cnt2), q(Cnt2).

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

stream argument type

This type is provided to ease output stream management. If no update occurs within the clause, the streams coming from goals are merged into the argument stream. This is illustrated below, in example a). To update the stream, i.e. insert elements, the following syntax should be used :

```
&arg-name <<= [element 1, element 2, ... ]
```

This is illustrated below, in example b). Note that the relative position of the update within the clause conditions the insertion order, although this may be of little importance for streams.

```
≪ ex ≫ definition:  :- implicit window:stream.

a) before expansion:  p --> true | q, r.
   after expansion:   p(Win) :- true | Win={In1,In2}, q(In1), r(In2).

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

oldnew argument type

This type calls after a pair of arguments, in a similar fashion to Prolog's DCG, except that arguments are not restricted to difference lists. As an example, when using a vector as an updatable table, to improve efficiency, one often restricts the number of references to 1. To this end, the oldnew argument type is useful. Also, if you use this type of argument for a difference list, there is a notation to concatenate elements to the list, like for the stream type :

```
&arg-name <<= [element 1, element 2, ... ]
```

If the argument is an integer or a floating point, use the following for update :

```
&arg-name  <= new-value
&arg-name $<= new-value
```

If argument is a vector, use the following syntax. Then on the right side of <= integer operation macros are expanded, and on the left side of $<= floating point operation macros are expanded.

```
{ element update(1) }
        &arg-name(position)  <= new-value
        &arg-name(position) $<= new-value

{ element reference }
        &arg-name(position)    %% cannot also appear on the left side of <=

{ element update(2) }
        &arg-name(position) <<= [element 1, element 2, ... ]
```

update(1) and update(2) can be used in body part only, as they use the built-in predicate set_vector_element/5. Reference can appear in guard, where it appeals to vector_element/3, or in body, where it uses vector_element/5. The relationship between homonymous elements is similar to that of DCG (from left to right, top to bottom). See example a),b) and f) below.

In (1), replacement is done at the specified position, as illustrated in examples c) and d). In (2), the right side list is inserted in the difference list. New tail is set to the location specified by position. See example e).

There is also a way to refer the current old value of some argument :

```
&arg-name(old)
```

This is useful in particular to access the value of a counter. This is illustrated in example g) below.

```
≪ ex ≫ definition:  :- implicit mutter:oldnew.

a) before expansion:  p --> true | q, r.
   after expansion:   p(Old,New) :- true | q(Old,Mid), r(Mid,New).

b) before:  p --> true | &mutter <<= [naha], r.
   after:   p(Old,New) :- true | Old=[naha|Mid], r(Mid,New).

c) before:  p --> true | &mutter(3) <= naha, r.
   after:   p(Old,New) :- true | set_vector_element(Old,3,_,naha,Mid), r(Mid,New).

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

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

f) before:  p --> true | &mutter <= &mutter+1, r.
   after:   p(Old,New) :- true | add(Old,1,Mid), r(Mid,New).

g) before:  p(X) --> true | X = [&mutter(old)|XX], &mutter <= &mutter+1, p(XX).
   after:   p(Old,New,X) :- true | X=[Old|XX], add(Old,1,Mid), p(Mid,New,XX).
```

string argument type

This works basically as the previous type, except that predicates string_element/3 and set_string_element/4 are used instead of vector-based predicates.

Automatic generation of terminating processes

When no user-defined goal is called in the body, the following unifications are automatically performed, depending on the type of declared arguments :

shared type :: Nothing done.
stream type :: Unification with the atom [].
oldnew type :: Old and New are unified.
string type :: Old and New are unified.

Implicit arguments expansion control

If you call a predicate with no implicit arguments from a predicate with implicit arguments, use double braces : {{ predicate(...) }}, in order to suppress argument addition. See example below :

```
:- 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.
```

In the previous example, three global implicit arguments are declared, with types `stream`, `oldnew` and `shared`. Predicates using `-->` instead of `:-` are regarded as having three implicit arguments and are converted at preprocessing time. As an example, loop predicate is expanded as follows :

```
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).
```

Note that the check/4 predicate, used between braces, has no implicit argument, and is expanded as a predicate of arity 4. In order to use some of implicit arguments when calling this predicate, `&` has to be put before the names of the implicit arguments which are explicitely specified in the call.

≪ ! ≫Can implicit arguments take any value, declared types notwithstanding. As a matter of fact, the macro processor only expands. If the programmer is not careful enough, errors may be difficult to detect.

2.7.6 Conditional branch macros

The following is a notation which allows conditional execution within a single clause, as in DEC10 Prolog :

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

If the goal on the left hand side of -> is a condition and if this condition is satisfied, then the goal on the right hand side is executed. The preprocessor generates the following KL1 clauses, from the above example :

```
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).
```

The predicate `'$foo/2/0'` has been generated by the preprocessor. More generally, predicates starting with a dollar sign are generated by the preprocessor. The user should not use the same convention!

≪ ! ≫Only built-in predicates may be included in a conditional branch.
≪ ! ≫The Prolog-based compiler does not support nested conditions, whereas the KL1 compiler does.

2.7.7 Macro library

Macros in system's library to be used should be declared at the top of the module. Declaration goes as follows :

```
:- with_macro macro-definition-name.
```

where `macro-definition-name` is an atom.

The macro definition files are located in a system dependant directory. In this file, macros are defined as follows :

```
fileio#normal      => 0.
fileio#end_of_file => 1.
fileio#read_error  => 2.
fileio#write_error => 3.
```

In the current version, the left part from the sharp sign must be an atom.

3 Micro PIMOS

Micro PIMOS is a very simple operating system which provides various services for KL1 users on PDSS. It is basically designed for single user, single task operations. Services supported by Micro PIMOS follow :

- Command Interpreter.
- I/O Functions (windows, files, etc.).
- Code Management.
- Display of exception information.

On Micro PIMOS, all commands given to the command interpreter are executed within a unit called 'task'. The task is implemented using Sho-en functions described in section 2.2.

≪ ! ≫Bits 15:31 in exception tag are reserved for Micro PIMOS. When using functions of Micro PIMOS, user must not modify bits 15:31 in tag of his Sho-en. Beside commands, a way to use Micro PIMOS functions is to issue requests to Micro PIMOS through the user's goal which supervises the Sho-en.

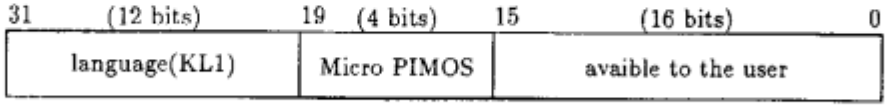

Figure 7: Sho-en exception tag

3.1 Command interpreter

When Micro PIMOS starts, a command interpreter is created to provide user with an interface to PDSS.

When the command interpreter is invoked, it issues a prompt and waits for the next command. Default prompt is `| ?-`, or `[debug]?-` when debugging mode is on.

The command interpreter starts by executing the file "~/.pdssrc" (if it exists) as a command file. User can set up a convenient working environment via this file. About command file, refer to command take/1.

3.1.1 Command input format

It is possible to write one or more commands in a command line or command file. Depending on the delimiter, commands are executed as follows :

- comma (",")
  Commands both before and after the delimiter are executed in parallel.
- semicolon (";")
  Waits for the termination of commands before the delimiter, and then execute remaining commands. (sequential execution.)
- vertical line ("|")
  After executing the commands before the delimiter, displays the value of variables after this sign. The delimiter between variables is a comma, and `all` stands for all variables.

Lists of commands may be embedded in ( ) to form nested commands.

≪ ex ≫
```
| ?- comp("bench");(stat(bench:primes(1,300,P))|P),save(bench).
     % After compiling "bench.kl1", executes the goal 'bench:primes'
     % and saves the code in parallel. During the execution of the
     % goal, reports statistical information and indicates
     % the value of variable 'P' after termination.
```

Constant description macros in a command line(terms) are expanded. Macros are described in 2.7.

≪ ex ≫
```
| ?- X=16#"FF",Y=16"Z",Z="FF"|all.
X=255
Y=16#"FF"
Z="FF"
```

3.1.2 Commands

Here are the commands supported by the command interpreter, in its current version. Some of the commands expect files to have an extension. If no extension is found, operation is done with the default extension, following the specified filename. Strings or atoms can be used to specify filenames.

Built-in predicates

The command interpreter can execute the built-in predicates which can be described in the body part as a commands. Description using := and arithmetic macros are also possible.

Basic commands

ModuleName:Goal
Executes `Goal` in the module `ModuleName`. Maximum number of reductions is set according to the value of environment variable `reduction`. If the number of performed reductions crosses the limit, the user will be asked whether to continue or to abort.

help
Displays the list of available help commands.

help(Type)
Displays the list of available commands specified by `Type` as follows :
1: builtin, 2: basic, 3: code, 4: dir, 5: debug, 6: env, all.

gc
Invokes GC over the heap area.

gc(all)
Invokes GC for both heap and code areas.

take(FileName)
Executes the command file specified by `FileName`. There is no restriction as to the type of command which can be used in such a file.

% or /* */ are available to mark comments, as in KL1 program.

cputime
Display the CPU time since PDSS started. Unit is millisecond.

cputime(^Time)
Unifies the CPU time since PDSS started with the variable `Time`. The result is an integer and unit is millisecond.

apply(CommandName, ArgsList)
Executes `CommandName` upon each element specified in `ArgList`. ModuleName:PredicateName is also possible for `CommandName`.

stat
Displays the current memory status.

stat(Commands)
Displays the execution time (CPU time) and reduction count of `Commands`.

window(IOStream)
Opens a new window. About commands which deal with I/O streams, refer to the section 3.2. Window name is set automatically.

add_op(Precedence, Type, Operator)
Add an operator to the window of command interpreter.

remove_op(Precedence, Type, Operator)
Deletes an operator from the window of command interpreter.

operator(Operator)
Displays the definition of the operator `Operator` in the window of command interpreter.

operator(OperatorName, ^Definition)
Unifies the definition of operator `Operator`, (format is {Precedence, Type}) in the window of command interpreter with `Definition`.

replace_op_pool(^OldOpPool,NewOpPool)
Unifies the old operator pool with `OldOpPool`, and replaces the operator pool with `NewOpPool`. The format of an operator pool is [{OpName, [{Precedence, Type}, ...]}, ...].

change_op_pool(NewOpPool)
Changes the operator pool to `NewOpPool`.

halt
Terminates PDSS. All windows are closed automatically.

Code commands

comp(FileName)
Compiles the KL1 source file (with extension `.kl1`) `FileName` and loads the result into code area. Trace mode is off for the newly loaded module.

comp(FileName, OutFileName)
Compiles the KL1 source file (with extension `.kl1`) `FileName` and outputs the result into the file (with extension `.asm`) `OutFileName`.

compile(FileName)
Compiles the KL1 source file (with extension `.kl1`) `FileName` and outputs the result into file (with extension `.asm`) `FileName`. Then, loads it into code area and saves into file (with extension `.sav`). `FileName` can also hold a list of files.

load(FileName)
Loads the previously saved file `FileName` (with extension `.sav`) into code area. If such a file does not exist, assembler file (extension is `.asm`) is loaded into code area. Trace mode of the newly loaded module is off.

dload(FileName)
As above, but trace mode of the newly loaded module is on. Then debug mode of it is set to on.

save(ModuleName)
Saves the executable code module `ModuleName` to the directory specified by environment variable `savedir`. By default, the directory is `~/.PDSSsave`. It can be changed with the 'ch_savedir' command. `ModuleName` is also used to determine file name. Extension is `.sav`.

save(ModuleName, FileName)
Saves the executable code `ModuleName` to the file `FileName` (with extension `.sav`).

save_all
Saves all loaded modules (except the ones which have already been saved by the `save(ModuleName)` command) into the directory specified by environment variable `savedir`.

ch_savedir(Directory)
Changes the default directory for auto_load and auto_save, to the directory specified by `Directory`. The existence of directory is checked.

listing
Displays information about loaded modules.

listing(^Modules)
Generates a list of all loaded module names and unifies it with `Modules`.

public(ModuleName)
Displays a catalog of public predicates within the module specified by `ModuleName`.

public(ModuleName, ^Public)
Creates a list of information about predicates declared as public, and unifies it with `Public`. Each element of the list is a two-elements-vector of form {predicate-name-atom, arity}.

Directory commands

cd(Directory)
Changes current directory to the directory specified by `Directory`.

pwd
Displays the pathname of current directory.

pwd(^World)
Unifies the pathname of current directory with `World`.

ls(WildCard)
Displays the pathname of file `WildCard`.

ls(WildCard, ^Files)
Creates a list with the pathnames corresponding to `WildCard` and unifies it with `Files`.

rm(WildCard)
Deletes the file corresponding to `WildCard` from the directory.

Debug commands

trace(ModuleName)
Sets the trace mode on for the code of module `ModuleName`. The debug mode is set to on.

notrace(ModuleName)
Sets the trace mode off for the code of module `ModuleName`.

spy(ModuleName, PredicateName, Arity)
Enables spying of the predicate `PredicateName/Arity` in the module `ModuleName`. Then trace mode and debug mode is set to on.

nospy(ModuleName, PredicateName, Arity)
Disables spying of the predicate `PredicateName/Arity` in the module `ModuleName`.

spying(ModuleName)
Displays the list of the predicates currently spied in the module `ModuleName`.

spying(ModuleName,^Spying)
Creates a list of the predicates currently spied in the module `ModuleName`, and then unifies it with `Spying`. Each element of the list is a two-elements-vector formed as {predicate-name-atom, arity}.

debug
Sets debug mode on.

nodebug
Sets debug mode off.

backtrace
Sets display mode on for backtrace information (deadlock information).

nobacktrace
Sets display mode off for backtrace information (deadlock information).

varchk(FileName, Mode, Form)
Checks variables in the KL1 source file (with extension `.kl1`) specified by `FileName` in the mode `Mode`. The result is displayed on the window in the format `Form`. `FileName` can also be a list of files. The definition of `Mode` and `Form` follows :

Mode::	`o` or `one`	··· Displays variables which appear once in a clause.
	`m` or `mrb`	··· Displays variables whose MRB is set.
	`a` or `all`	··· Displays variables of both one and mrb modes.
Form::	`s` or `short`	··· Outputs clauses as a single line.
	`l` or `long`	··· Outputs clauses using line feeds and indentation.

varchk(FileName, Mode)
Checks variables in mode `Mode` and displays in `long` format.

varchk(FileName)
Checks variables in mode `one` and displays in `long` format.

xref(FileName, Mode)
Performs a cross reference check upon the KL1 source file `FileName` (with extension `".kl1"`) and displays result on the window. `FileName` can also be a list of filenames. In this case, references across modules are also checked. `Mode` can be taken amongst the following values :

`c` or `check`	···	Checks only predicate calls.
`l` or `list`	···	Outputs the reference list (table of definition/reference of predicates).
`s` or `system`	···	Outputs the predicates referring to PDSS modules.
`b` or `builtin`	···	Outputs predicates referring to body-built-in predicates.
`g` or `guard`	···	Outputs predicates referring to guard-built-in predicates.
`a` or `all`	···	Outputs all of above predicates.
`List`	···	Outputs a reference list for specified elements. Where `List` can include : * Module name. * Body-built-in predicates. * User-defined predicates.
`short`	···	Checks with no display of predicate information.
`short(Mode)`	···	Checks according to `Mode` with no display of predicate information.
`update`	···	When duplicate module names are found in specified files, recognizes the latter definition as efficient, and checks without a caution.
`update(Mode)`	···	When duplicate module names are found in specified files, recognizes the latter definition as efficient, and checks according to `Mode` without a caution.

xref(FileName)
Checks the cross-references in `check` mode.

xref(FileName, Mode, OutFile)
Checks the cross-references and outputs the list to `OutFile`. Any mode is available except `check`.

profile(ModuleName, Mode)
Displays how many times the predicates which are defined in the module `ModuleName` were called and suspended. `ModuleName` can also be a list of modules. `Mode` can be chosen as follows :

`c` or `call`	···	Sorts according to call count and displays.
`s` or `susp`	···	Sorts according to suspension count and displays.
`n` or `no`	···	Displays following the order of appearance within the code area.

profile(ModuleName)
Executes `profile` command in `call` mode.

reset_profile(ModuleName)
Resets counts of calls and suspensions for predicates defined in `ModuleName`. `ModuleName` can also be a list of modules.

Environment commands

These commands can be used to change values of environment variables for the command interpreter. The following environment variables are used :

name (atom)	meaning
world	Pathname string of current directory.
trace	Mode of tracer (on or off). Initial value is off.
backtrace	Display mode of backtrace (on or off). Initial value in on.
modules	List of module names in which commands are searched.
reduction	Upper limit of the number of reductions assigned when the task was created. The basic allocation unit is 10000 reductions. (0 < number < 100000, Initial value is 10000)
ucounter	Counter used to create the names of work files or work windows.
savedir	String with the pathname of directory in which save/1 and save_all will produce their effects. Initial value is "/.PDSSsave.
loaddir	list of pathnames of directories to examine when auto-loading code. Initial value is ["/.PDSSsave, pathname of library directory, ...] Note: There are more than two library directories, which may differ from one machine to another.
auto_load	Flag for auto_loading (yes or no). Initial value is yes.
plength	Maximum length of structure which can displayed in the window of the command interpreter. Initial value is 10.
pdepth	Maximum depth of structure which can be displayed on the window of the command interpreter. Initial value is 5.
pvar	Displays modes of variables in the window of command interpreter. The value is nu or na. (nu works as _0,_1,_2, ... , and na works as A,B,C, ...) Initial value is nu.

setenv(Name, Value)
Sets the environment variable **Name** after **Value**. Environment variable is set after that **Name** becomes an atom and **Value** becomes a ground term.

getenv(Name, ^Value)
Unifies the value of environment variable **Name** with **Value**.

printenv(Name)
Displays the value of environment variable **Name**.

printenv
Displays the values of all environment variables of the command interpreter.

resetenv
Initializes all environment variables of the command interpreter.

3.2 I/O functions

Micro PIMOS offers two types of I/O services : window and file I/O services. To use them, Micro PIMOS predicates are provided, which give access to command streams. How commands can be inserted in these streams is now described. [] closes a command stream and, by the way, the associated I/O device. Commands are inserted in command stream by a merger.

3.2.1 Command stream attachment

Windows

window:create(Stream, WindowName, ^Status)
Creates a window with name **WindowName** (8 bits string). Command stream is unified with **Stream**. **Status** is unified with the following terms :

success	··· Window successfully created.
error(cannot_create_window)	··· Failure : window cannot be opened.
error(bad_window_name_type)	··· Failure : **WindowName** is not an 8 bits string.

≪ ! ≫When the window is created, it is not in visible state. Use command **show** to make it appear.

window:create(Stream, WindowName)
Creates a window with name **WindowName** (8 bits string). Command stream is unified with **Stream**. If the window cannot be created, the whole task is aborted.

≪ ! ≫When the window is created, it is not in visible state. Use command **show** to make it appear.

Files

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

Opens file with name `FileName` (8 bits string) with mode `Mode`. `Mode` is an atom chosen among : `r` for read, `w` for write and `a` for append. The command stream of this file is unified with `Stream`. `Status` is unified with one of the following terms :

`success`	··· Open successful.
`error(cannot_open_file)`	··· Cannot open the file.
`error(bad_file_name_type)`	··· `FileName` is not an 8 bits string.
`error(bad_open_mode_type)`	··· `Mode` is not atom.
`error(bad_open_mode)`	··· `Mode` is an atom other than `r`, `w` or `a`.

file:create(Stream, FileName, Mode)

Opens a file as the previous command. `Stream`, `FileName` and `Mode` have same meanings, but if open does not succeed, the whole task is aborted.

3.2.2 Command list

Commands allowed in stream are now listed. These commands are common to window and file, unless otherwise specified.

Input commands

getc(^Char)

Reads an ascii character from I/O device. Value is between 0 and 255. `Char` is unified with the result of input. Upon end of file, `Char` is unified with the atom `end_of_file`.

getl(^String)

Reads one line from I/O device. This line is converted into an 8 bits string, unified with `String`. Upon end of file, `String` is unified with the atom `end_of_file`.

getb(^Buffer, Size)

The number of character specified by `Size` is read from I/O and converted into an 8 bits string, which is unified with `Buffer`. If a carriage return or an end of file is encountered, only characters read before are considered as input. Upon end of file, `Buffer` is unified with the atom `end_of_file`.

gett(^Term)

A string containing one term is read. (A term ends with . + CR or . + space) These characters are analyzed and transformed into a term, which is unified with `Term`. If an error occurs during analysis, if input device is a window, error is output on this window and term input is resumed. If input device is a file, error is displayed on the command interpreter window, then next term is read from the file. At end of file, term is unified with `end_of_file`.

gett(^Term, ^Status)

Almost same as `gett/1`, except that `Status` is unified with one of the following terms.

`success`	··· Input a term successful.
`syntax_error(Position)`	··· Syntax error in `Position`.
`ambiguous(Position)`	··· Ambiguous expression in `Position`.
`end_of_file`	··· End of file.
`eof_in_quote`	··· End of file between quotation marks.

When `Status` is `syntax_error`, `ambiguous`, `eof_in_quote`, `Term` is unified with a token list (See Appendix-1 gettkn/4).

getft(^Term, ^NumberOfVariables)

This command is very similar to `gett/1`, but variables in the term are analyzed then output as `$VAR(N,VN)`. N is the variable number ($0 \leq N <$ `NumberOfVariables`) and VN is the variable name (8 bits string). `NumberOfVariables` is unified with the number of variables appearing in the term. Upon end of file, `Term` is unified with `end_of_file` and `NumberOfVariables` is unified with 0.

getft(^Term, ^NumberOfVariables, ^Status)
Almost same as `getft/2`, and see `gett/2` about `Status`.

skip(Char)
Skips until the charactor code `Char` or end of file is found.

≪ ! ≫If end of file has been encountered during the execution of previous commands, successive input commands will return end of file.

Output commands

putc(Char)
Outputs on the I/O device the character with ASCII code `Char`, between 0 and 255.

putl(String)
Outputs the 8 bits string `String` on the I/O device and adds a new line character.

putb(Buffer)
Outputs 8 bits string `Buffer`. No carriage return is added.

putb(Buffer, Count)
Outputs on the I/O device the charactors by certain length specified by `Count` extracted from 8 bits string. If the length of `Bffer` is less than the specified length, this is the same as putb/1.

putt(Term, Length, Depth)
Outputs term `Term`. If structure depth exceeds `Depth` (> 0) or length exceeds `Length` (> 0), remainder is output as '...'. This is similar to Prolog's write.

≪ ! ≫Atoms are not quoted, so that the result of this command may be unsuited to further read using gett or getft.

putt(Term)
Similar to the previous command, but default value is used for depth and length.

puttq(Term, Length, Depth)
This is similar to putt/3 command, but atoms are quoted when necessary.

puttq(Term)
This is similar to putt/1 command, but atoms are quoted when necessary.

nl
Outputs a new line character.

tab(N)
Outputs `N` ($0 \leq N < 1000$) space characters.

≪ ! ≫On Micro PIMOS, I/O is blocked, for efficiency reasons. Buffers are flushed only in the following cases :

- Buffer is full.
- `flush` command has been received.
- I/O device is closed.
- (in the case of windows) some input or show/hide command is received.

Control of output format

Output limitations for structure in putt/1 and puttq/1 commands can be changed as follows :

print_length(Length)
This command changes the default length limit to `Length` (`Length` > 0). Initial value is 10 for windows, 100 for files.

print_depth(Depth)
This command changes the default depth limit to `Depth` (`Depth` > 0). Initial value is 10 for windows, 100 for files.

print_var_mode(VariableMode)
This command is used to change the output format of terms describing variables. `VariableMode` is the new mode which must be nu or na. Initial value is na.

na – Name Mode	::	$VAR(N,VN) → VN (Variable name string).
		$VAR(N) → A,B,C ··· .
nu – Number Mode	::	$VAR(N,VN) → _N (Variable number).
		$VAR(N) → _N (Variable number).

Output buffer commands

The following command control output buffer parameters.

flush(^Status)
This command flushes characters left in buffer. After completion, `Status` is unified with atom `done`.

buffer_length(BufferLength)
This command changes output buffer length to `BufferLength` (> 0). Initial value is 512 bytes for a window and 2048 bytes for files.

Operators

The following commands are related to operators for parsing.

add_op(Precedence, Type, OperatorName)
This command adds an operator with precedence `Precedence` ($1 \leq$ `Precedence` ≤ 1200), type `Type` (an atom chosen among **`fx`**,**`fy`**, **`xf`**, **`yf`**, **`xfy`**, **`xfx`**, **`yfx`**) and name `OperatorName` (atom).
≪ ! ≫When the specified type is competitive with already defined type (fx ↔ fy, xf ↔ yf, xfy ↔ xfx ↔ yfx), the latter is deleted.

remove_op(Precedence, Type, OperatorName)
This command removes an operator. Parameters have the same meaning as in the previous command.

operator(OperatorName, ^Definition)
This command return a list `Definition` of operators matching name `OperatorName` (atom). Each element of the list is in the form {precedence, type}.

replace_op_pool(^OldOpPool, NewOpPool)
Unifies the old operator pool with `OldOpPool`, and replaces the operator pool with `NewOpPool`. The format of an operator pool is [{OpName, [{Precedence, Type}, ...]}, ...].

change_op_pool(NewOpPool)
Changes the operator pool to `NewOpPool`.

Grouped processing of commands

do(CommandList)
This command groups the list of command `CommandList` within a single command. Even though merger is used to insert commands in the stream, sequence of commands in `CommandList` is preserved.

Control command

close(^Status)
Closes I/O operation. It is not possible to send other commands after that one. (Only □ can be sent to close the stream.) `Status` is unified with atom `success`.

Window commands

The following commands are effective only for windows.

show
An hidden window will show up when this command is executed.

hide
A visible window will be hidden when this command is executed.

clear
Clears the window space.

beep
Rings the terminal bell.

prompt(^Old, New)
Changes prompt string displayed in executing `gett` or `getft` command. `Old` is unified with current prompt string(8 bits string) and the new prompt becomes `New` (also an 8 bits string). The initial prompt is `"?-"`.

3.3 Directory management

The directory services of Micro PIMOS are available through the directory command stream. This stream is available via a Micro PIMOS predicate, in a similar fashion to I/O services.

Operations on the directory are done by inserting commands into this stream. The stream can be closed with □.

3.3.1 Acquisition of command stream

directory:create(Stream, DirectoryName, ^Status)
Accesses the directory named `DirectoryName` (8 bits string) and unifies the command stream connected to the directory with `Stream`. `Status` can be unified with the following terms :

`success`	··· Access succeeded.
`error(cannot_access)`	··· The directory cannot be accessed.
`error(bad_directory_name_type)`	··· `DirectoryName` is not an 8 bits string.

3.3.2 Commands

The following commands can be inserted into the command stream.

pathname(^PathName)
Unifies the full pathname of the directory (8 bits string) with `PathName`.

listing(WildCard, ^FileNames, ^Status)
Creates the list of pathnames of files corresponding to `WildCard` and unifies it with `FileNames`. `Status` can be unified with the following terms :

`success`	··· List successfully created.
`error(cannot_listing)`	··· List cannot be created.

delete(WildCard, ^Status)
Deletes files corresponding to `WildCard` (8 bits string) from the directory. `Status` can be unified with the following terms :

`success`	··· Deletion successful.
`error(cannot_delete)`	··· Cannot delete the file.

open(Stream, FileName, Mode, ^Status)
Opens the file `FileName` (8 bits string) with mode `Mode` (atom `r` for read, `w` for write or `a` for append) and unifies the command stream connected to the file with `Stream`. `Status` can be unified with the following terms :

`success`	··· Open successful.
`error(cannot_open_file)`	··· Cannot open the file.
`error(bad_file_name_type)`	··· `FileName` is not an 8 bits string.
`error(bad_open_mode_type)`	··· `Mode` is not atom.
`error(bad_open_mode)`	··· `Mode` is an atom other than `r`,`w` or `a`.

3.4 Device Stream for I/O

To use Input/Output device functions directly from Micro PIMOS, one can use the libraries now described. The purpose of the functions therein is to describe other OS than Micro PIMOS (e.g. PIMOS) in KL1. Average user does not need device streams shown below.

These device streams are supervised by Micro PIMOS. So if a wrong command is inserted, it only results in the failure of user task; the language processor is unaffected.

3.4.1 Securing device stream

User can extract a device stream from Micro PIMOS by using the following predicates. `mpimos_io_device` can also be used as a module name.

mpimos_window_device:windows(Stream)
Unifies the stream which has a function of window device with `Stream`.

mpimos_file_device:files(Stream)
Unifies the stream which has a function of file device with `Stream`.

mpimos_timer_device:timer(Stream)
Unifies the stream which has a function of timer device with `Stream`.

3.4.2 Command

The commands which can be sent to each device stream, stream of opened window, file and directory are just the same as mentionned in Appendix-1. As to the I/O commands for file/window streams, only the commands shown below are allowed.

- Window

Input Only getl(^Line, ^Status, Cdr) is available.
getc/3, getb/4, gettkn/4 are not available.

Output Only putb(Buffer, ^Status, Cdr) is available.
putc/3, putl/3, putt/5 are not available.

- File

Input Only getb(Size, ^Bufferm ^Status, Cdr) is available.
getc/3, getl/3, gettkn/4 are not available.

Output Only putb(Buffer, ^Status, Cdr) is available.
putc/3, putl/3, putt/5 are not available.

3.5 Code management

The principal functions for code management on Micro PIMOS now follow.

- Functions to manage the name and information (like the catalog of public predicates and spied predicates) of loaded modules and display this information upon request.
- Auto_load function of modules which are saved by `save(ModuleName)` or `save_all` commands from the command interpreter.

The directory from which `auto_load` is performed is decided after the environment variable `loaddir` of command interpreter. User had better make a directory `~/.PDSSsave` to use the auto_load function, because the default value of first element of both `loaddir` and `savedir` is `~/.PDSSsave`. The value of environment variables can be changed. User can disable the auto load function by setting the environment variable `auto_load` to `no`.

3.6 Displaying exception information

The KL1 exceptions handled by PDSS are shown in section Appendix-7. On Micro PIMOS, information about an exception which has occurred within the user task is displayed on the window of command interpreter. The task in which exception has occurred is immediately stopped and its resources (windows and files) are released.

Other exceptions, reported by Micro PIMOS, are handled by Micro PIMOS. Those are consequent to an illegal command to the window, trying to open a file that does not exist, etc. In those cases, as in the case of language definition exceptions, information is displayed on the window of command interpreter and the task is immediately stopped. All resources of the task are released.

4 PDSS Optional Parameters

PDSS is usually invoked under GNU-Emacs. This may be seen as the best way to use PDSS, from an execution environment point of view, as all PDSS functions are available. It is possible to execute PDSS on a stand alone basis, but in this case, some functions disappear.

4.1 Usage under GNU-Emacs

To call PDSS under GNU-Emacs, send the following command. Libraries are automatically loaded and PDSS starts.

meta-X pdss *return*

To specify options, type ctrl-U before meta-X. Option contents are described later.

ctrl-U meta-X pdss *return*
PDSS Option?: **[parameter]** *return*

When PDSS starts, a window named "console window" is created. This window is used to trace execution and for input/display at console. Then, several modules are loaded, including runtime support and Micro PIMOS. Then, Micro PIMOS starts. After that, command interpreter window is created and waits for user commands.

When operation is done within GNU-Emacs, PDSS input is asynchronous. Therefore the whole system does not hold when input occurs. There is an exception to this for console inputs while tracing. In this case, system halts until input completion. It is possible to control PDSS by striking control keys in the window. These keys are defined in a GNU-Emacs library. Besides following commands, a complete list of supported keys can be found in Appendix-9.

ctrl-C ctrl-C	::	Turns on trace flag.
ctrl-C ctrl-Z	::	Sends interrupt code 1.
		In Micro PIMOS, this aborts task.
ctrl-C ctrl-T	::	Sends interrupt code 2.
		In Micro PIMOS, this prints number of reductions performed so far.
ctrl-C !	::	Starts GC.
ctrl-C @	::	Aborts PDSS.
ctrl-C ctrl-B	::	Generates a window buffer menu for PDSS.
ctrl-C ESC	::	Reexecutes PDSS system.
ctrl-C k	::	Removes contents of current window.
ctrl-C ctrl-K	::	Deletes contents of the window created by PDSS.
ctrl-C ctrl-Y	::	Reprints the last input string.
ctrl-C ctrl-F	::	Prints manual of built-in predicates.
ctrl-C f	::	Prints manual of command interpreter.

≪ ! ≫When a PDSS window is removed by ctrl-X k, subsequent execution results are not guaranteed to be meaningful.

4.2 PDSS on stand-alone

To use PDSS without GNU-Emacs, type the following command :

pdss [parameter] *return*

Outside of GNU-Emacs, all messages to windows are merged. If any window waits for some input, the whole system stops. Window control keys are not available but, on the other hand, keyboard interrupt is supported. If ctrl-c is typed, one can enter control commands after the prompt.

4.3 Optional parameters

Optional parameters can be specified at start, to modify the execution environment. Possible parameters follow :

-hNNN	:: Size of heap area is **NNN** words. Default is 200 kwords(1 word = 8 bytes).
-cNNN	:: Size of code area is **NNN** bytes. Default is 300 kbytes.
file name	:: Uses this file instead of the standard startup file.
+t/-t	:: Uses start up file or not. Default is to use it.
-v	:: This option changes the way variables appear during trace. By default,variables are printed using their name. If -v option is used, they are identified by their relative position to heap bottom. This may change after each GC, so be careful.
-dNNN	:: Scheduling politics for goals are changed to depth-first. **NNN** gives depth limit for TRO.
-bNNN	:: Scheduling politics for goals are changed to breadth-first. **NNN** gives depth limit for TRO.
-rRR,SS,NNN	:: Scheduling politics for goals are basically depth-first, and some of goals are pushed to the tail of the scheduleing queue according to the calculated random number. This makes possible to simulate the undecided reduction ordering. **RR** specifies the ratio of goals pushed to the tail of the queue by percentage. **SS** is the seed of the calculation of random number. **NNN** gives depth limit for TRO.
-a	:: Inhibits timer interrupt. This is used when debugging PDSS itself, under dbx.

There are two ways to specify these options :

- Options can be given when starting PDSS. They are treated as arguments of the PDSS command.

```
example-1)
       PDSS Option ?: -h300000 -c50000 -v     (Execution under GNU-Emacs)
example-2)
       [UNIX]% pdss -h300000 -c50000 -v       (Execution on stand alone)
```

- Options can also be specified through an environment variable.

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

5 Tracer

The tracer functions supported by PDSS are now described.

5.1 Principle of operation

Basically, in PDSS, trace operations can occur whenever a goal is in one of the following states. These events are called trace points.

- Goal call.
- Suspension due to an uninstanciated argument.
- End of suspension.
- Goal failure.
- Swap out (Caused by interruption or scheduling of a higher priority goal).

There are two ways to operate trace in KL1 : upon predicate execution or upon goal call.

To trace upon predicate execution is to trace when the code, which ones want to trace, is executed. In this case, it is possible to specify trace mode for each module. In the following description, this mode is called "code trace". It is also to trace each predicate separately. This is called "code spying".

To trace upon goals is to trace, or not to trace, descendant goals of each generated goals. In the following description, we call this "goal trace". It also possible to limit trace to the descendant goals of specific goals. This is called "goal spying".

Let's see some example. In the following program, we assume that `foo` is in trace state and `bar` is not. Then, `q` and `r` which are called from `foo` are also traced. Conversely, `q` and `r` which are called by `bar` are not traced.

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

In PDSS, it is possible to specify before or during execution whether or not code trace is done. On the other hand, goal trace status must be on at first; then, some of the goals can be untraced by relevant commands during tracing. Only goals which have both code trace status on and goal trace status on are actually traced.

In case of spying it is possible to specify before or during execution whether or not code spying is done. Goal spying can be done only on goals which is speculated by tracer. The four possible cases of spying are the following :

- Code is spied.
- Goal is spied.
- Code or goal are spied.
- Code and goal are spied.

5.2 How to read the display

Trace display contains 4 different information zones :

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

1. Identity of the Sho-en to which this goal belongs.
2. Type of trace event :

   `CALL` :: Dequeue from goal queue.
   `Call` :: Goal called during TRO.
   `SUSP` :: Suspension due to an uninstanciated argument.
   `Susp` :: Suspension due to an uninstanciated argument which specifies the ratio calculating the priority.
   `RESU` :: End of suspension.
   `FAIL` :: Goal failure.
   `SWAP` :: Swap out.

3. Spy flags :
   - * :: Code of executing goal is spied.
   - $ :: Goal is spied.

4. Goal
   Terms in argument list which are potentially referred several times (MRB is on) are appended with an x mark.
   Variables are shown as follows, according to their type :

   - Ordinary unbound variable :
     First letter is upper case, or underscore, and is followed by a number... `X1`, `_23611`.
   - Some goal waits for instantiation of this variable :
     Format is the same as an ordinary unbound variable, followed by a tilde... `X1~`, `_23611~`.
   - Merger input variable :
     As above, replacing tilde with carret... `X1^`, `_23611^`.

In addition to this description, priority is displayed whenever it changes.

5.3 Commands

The description of tracer commands has the following meta syntax :

Command name :: input format {argument} { [options] }

Help :: ?
Command help.

No Trace :: X
No trace from now on.

No Goal Trace :: x
Turns off trace for the descendants of current goal, i.e. goals called from this goal.

Set Goal Spy :: g
Spies current goal from now on.

Reset Goal Spy :: G
Stops spying current goal, from now on.

Set Module Debug Mode :: d MODULE { MODULE ... }
Sets debug flag on for specified modules. By this mean, code trace is done when predicates from this module are executed.

Reset Module Debug Mode :: D MODULE { MODULE ... }
Effects are opposite to the previous command.

Set Procedure Spy :: p MODULE:PROCEDURE { MODULE:PROCEDURE ... }
Sets trace on for a given predicate in a given module.

Reset Procedure Spy :: P MODULE:PROCEDURE { MODULE:PROCEDURE ... }
Opposite of previous command.

Step :: s [COUNT]
Stops again at next trace point. If `COUNT` is given, stop occurs only after that `COUNT` trace points have passed. Here and in the following, `COUNT` can be considered as a repetition factor.

Step to Next Spied Procedure :: sp [COUNT]
Continues until the next spied predicate is called, then stops.

Step to Next Spied Goal :: sg [COUNT]
As above, but we look for a spied goal.

Step to Next Spied Procedure or Spied Goal :: ss [COUNT]
In this case, any spy case causes stop.

Step to Next Spied Procedure and Spied Goal :: SS [COUNT]
In this case, procedure must be traced and called from a traced goal, to cause stop.

Skip to Next Spied Procedure :: np [COUNT]
This works similarly to sp command, but no trace is done until stop.

Skip to Next Spied Goal :: ng [COUNT]
This works similarly to sg command, but no trace is done until stop.

Skip to Next Spied Procedure or Spied Goal :: ns [COUNT]
This works similarly to ss command, but no trace is done until stop.

Skip to Next Spied Procedure and Spied Goal :: NS [COUNT]
This works similarly to SS command, but no trace is done until stop.

Enqueue This Goal to Head of Ready Goal Queue :: <
Enqueues the goal displayed at the time to head of ready goal queue. This can be specified at RESU and SWAP.

Enqueue This Goal to Tail of Ready Goal Queue :: >
Enqueues the goal displayed at the time to tail of ready goal queue. This can be specified at RESU and SWAP.

Depth First Schedule :: << [DEPTH]
Scheduling politics for goals are changed to depth-first. DEPTH gives depth limit for TRO. Initial value is 2^{31}.

Breadth First Schedule :: >> [DEPTH]
Scheduling politics for goals are changed to breadth-first. DEPTH gives depth limit for TRO. Initial value is 100.

Random Schedule :: >< [RATE[SEED[DEPTH]]]
Scheduling politics for goals are basically depth-first, and some of goals are pushed to the tail of the scheduleing queue according to the calculated random number. This makes possible to simulate the undecided reduction ordering. RATE specifies the ratio of goals pushed to the tail of the queue by percentage. SEED is the seed of the calculation of random number. DEPTH gives depth limit for TRO.

Re-Write Goal :: w LENGTH [DEPTH]
This redisplays current goal and arguments, with modified format limits LENGTH and DEPTH. This is useful when arguments are large.

Where call from :: where
Shows the names of predicate and module which called current traced goal. This is valid only for run-time support routines or built-in predicates (D code).

Monitor Variable :: m VARIABLE_NAME [NAME] [LIMIT]
Monitors the value of variable, whenever it is bound. If the value is a list or a stream, display occurs whenever the top element is bound. Using NAME, it is possible to assign a new identifier to the monitored variable, so that the value is shown under that name. LIMIT is the number of times value can be shown without stopping the system. Without this parameter, whenever a variable is bound, the value is shown and the tracer waits for a user command.
During display of value, whether the value is a list or not is distinguished :

```
mon#var-name => value %%    ... In the case of a list.
mon#var-name == value %%    ... Otherwise.
```

In this situation, the following commands are available :

?	:: Help.
x	:: Stops monitoring this variable or list.
s [COUNT]	:: Goes ahead monitoring value without stop, for COUNT times.
w LENGTH [DEPTH]	:: Redisplays value.
m VAR [NAME] [LIMIT]	:: Sets a different monitoring.

Inspect Ready Queue :: ir [PRIORITY]

Shows goals in ready queue. If `PRIORITY` is given, only goals with that physical priority are shown.

Inspect Variable :: iv VARIABLE_NAME

Shows state of specified variable. When state is HOOK or MHOOK (goals are waiting for the instantiation of this variable), shows the waiting goals. When state is MGHOK (input merger variable), shows merger output variable.

Inspect Sho-en tree :: is

Shows Sho-en tree structure at current time. Horizontal drawing axis is used to represent the parent/children dependency, whereas the vertical axis is used to represent brotherhood. Each Sho-en is described using 5 characters : the first one corresponds to the state of Sho-en and the remaining 4 to its ID. Possible Sho-en status follow :

R :: Ready.
S :: Suspended.
A :: Aborted.

Trace Shoen tree :: ts

Turn on/off the trace flag of Sho-en tree structure. When on, tree structure is shown before and after each modification(e.g. generation, abortion, termination). The format of the description is the same as for above command.

Set Tracer Variable :: set NAME [VALUE]

Tracer variable `NAME` is set to `VALUE`, if present. Otherwise, current value is displayed. Current tracer variables and their default values are now listed.

pv :: Print variable mode. If value is **n**, variables are displayed alphabetically. If **a** is used, relative address are used.

pl :: Print length value.

pd :: Print depth value.

g :: Gate switch which determines whether trace is done or not upon each trace point. Value is made of five characters, each one with value **n**, **t** or **s**. These values correspond to "no trace", "trace (no stop)" and "trace (stop)" respectively. Characters correspond to points CALL/call, SUSP/Susp, RESU, SWAP and FAIL, respectively.

c :: Gate Switch for CALL points. Value is n, t or s.

s :: Gate Switch for SUSP points. Value is **n**, **t** or **s**.

r :: Gate Switch for RESU points. Value is n, t or s.

w :: Gate Switch for SWAP points. Value is n, t or s.

f :: Gate Switch for FAIL points. Value is n, t or s.

6 Dead-lock Detection

In PDSS, there are two dead-lock detection mechanisms. One acts through global GC, whereas the other tries to detect deadlock during execution. During GC, deadlock is always detected, whereas deadlock check done during execution sometimes fails.

In the following, types of deadlock detected in PDSS and tracer messages are shown.

Dead lock detection occurred during GC : Type=0

Example : when executing the following goal :

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

Following information is shown on the console window.

```
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
```

The goal appearing before message "Previous goal is deadlock root!" on console window is the root of the data dependance tree. There are some cases where several such trees exist and root is not unique in general. If there are loops structures, there is no root.

Variable referred only by itself (void variable) waiting for instantiation : Type=10

Example : In the following program, after execution of `p(X)` is executed.

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

Following information is shown on the console window.

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

Waiting for instantiation of a variable which will never be instantiated by other goals : Type = 11

Example : In the following program, after execution of `p(X)`, `q(X)` is executed.

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

Following information is shown on the console window.

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

Input variable of the merger, referred only by itself, waiting for instantiation : Type=12

Example : In the following program, after execution of `merge(In,Out)`, `p(X)` is executed.

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

Following information is shown on the console window.

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

Variable, which has a goal waiting for instantiation, is unified with a void variable : Type=20

Example : In the following program, after execution of `p(X)`, `q(X,Y)` is executed. This case also occurs if `Y` was not void, but would turn void as a result of the execution.

```
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.
```

Following information is shown on the console window.

```
*** 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^).
```

Merger input is unified with void variable : Type=21

Example : In the following program, `p(In,_)` is executed after `merge(In,Out)`.

```
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).
```

Following information is shown on the console window.

```
*** 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
```

Unifying two variables which have goals waiting for instantiation : Type = 22

Example : In the following program, `r(X,Y)` is executed after `p(X)` and `q(X)`.

```
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.
```

Following information is shown on the console window.

```
*** 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^).
```

Unifying input of merger and variable which has a goal waiting for instantiation : Type=23

Example : In the following program, `q(X,In)` is executed after `merge(In,Out)` and `p(X)`.

```
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).
```

Following information is shown on the console window.

```
*** 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
```

Unification of two merger inputs : Type = 24

Example : In the following program, In1=In2 is executed after `merge(In1,Out1)` and `merge(In1,Out2)`.

```
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).
```

Following information is shown on the console window.

```
*** 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
```

A variable which has a goal waiting for instantiation is not referred : Type=30

Example : In the following program, `q(X)` is executed after `p(X)`.

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

Following information is shown on the console window.

```
*** 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^).
```

A merger input variable is not referred : Type=31

Example : In the following program, `p(In)` is executed after `merge(In,Out)`.

```
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).
```

Following information is shown on the console window.

```
*** 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
```

Appendix-1 I/O devices

PDSS provides window, file and timer I/O devices. Commands to these devices are inserted in streams. These devices are defined in modules named **pdss_window_device**, **pdss_file_device** and **pdss_timer_device**. This specification is based upon "FEP-Host I/O Interface (V0.9)". Full features have not be implemented. Some of the messages or commands are therefore dummy or illegal. And macro expression **fep#xxxx** is in the library, so to use these macros, the following definition must be declared in the module.

```
:- with_macro pdss.
```

Acquisition of device stream

Device stream can be obtained by the predicates listed below. These predicates can be called only once after the emulator has been invoked. Twice or more calls will fail.

pdss_window_device:windows(Stream)
Unifies `Stream` with the command stream of window device.

pdss_file_device:files(Stream)
Unifies `Stream` with the command stream of file device.

pdss_timer_device:files(Stream)
Unifies `Stream` with the command stream of timer device.

Device commands

1. Window device
   Window device provides multiple window facility within GNU-Emacs. The following commands can be sent to this device :

   create(BufferName, WindowStream, ^Status, Cdr)
   Opens a window with buffer name `BufferName` (8 bits string), then unifies its command stream with `WindowStream`. When window is opened successfully, `Status` is unified with `fep#normal`. PDSS can't open more than 16 windows at a time. Therefore, it fails when user tries to open too many windows. In this case, `Status` is unified with `fep#abnormal`. I/O commands and control commands described below can be inserted in the command stream of a window which has been successfully opened. Note that reset/4 command must be applies before those commands to set up abort and attention lines. The window is automatically closed when its stream is closed.

   create(WindowStream, ^Status, Cdr)
   Create/3 without buffer name is not available.

   get_max_size(X, Y, PathName, ^Characters, ^Lines, ^Status, Cdr)
   Always returns `Characters` = 80, `Lines` = 40, `Status` = `fep#normal`.

2. File device
   This device provides standard facilities of UNIX files. The following commands can be applied to this device :

   open(PathName, Mode, FileStream, ^Status, Cdr)
   Opens file with path name `PathName` (8 bits string), mode `Mode` (atom:`fep#read` = read mode, `fep#write` = write mode, `fep#append` = append mode), then unifies `FileStream` with the command stream. When the file is opened successfully, `Status` is unified with `fep#normal`. Otherwise, `Status` is unified with `fep#abnormal`. I/O commands and control commands can be applied to a file which has been successfully opened (reset/4 is also requisite for files). The file is automatically closed when its stream is closed.

   directory(PathName, DirectoryStream, ^Status, Cdr)
   Opens directory with path name `PathName` (8 bits string) and unifies `DirectoryStream` with its command stream. When open is successful, `Status` is unified with `fep#normal`. When it fails, `Status` is unified with `fep#abnormal`. Commands can be inserted in a directory stream which has been successfully created (reset/4 is also requisite for directory.). A directory is automatically closed when its stream is closed.

3. Timer device

Unit of time of the timer device is millisecond, but updates are actually performed each second. The following commands can be sent to the timer device :

get_count(^Count, ^Status, Cdr)
`Count` is unified with the total elapsed milliseconds since 00:00:00 AM. `Status` is unified with `fep#normal`.

on_at(Count, ^Now, ^Status, Cdr)
When the time specified by `Count` is reached, `Now` is unified with `fep#wake_up`. `Status` is unified with `fep#normal` when the command is over.

on_after(Count, ^Now, ^Status, Cdr)
When duration specified by `Count` has elapsed, `Now` is unified with `fep#wake_up`. Status is unified with `fep#normal` when the command is over.

Window, file and directory commands

1. Control commands common to windows and files

These commands are available for window and file devices.

reset(AbortLine, ^AttentionLine, ^Status, Cdr)
Sets up abort and attention lines. This command should be issued right after the I/O stream has been generated. To abort an I/O request, `AbortLine` must be unified with `fep#abort` by the host. Once unified, abort line and attention line must be set up again with reset/4 command. Otherwise, the stream can still be closed with □. `AttentionLine` is unified with interrupt code generated by device (integer). Upon interrupt, I/O should be aborted or attention line should be set again with next_attension/3 command.

next_attention(^Attention, ^Status, Cdr)
Only attention line is set by this command. This command is used when user does not want to abort I/O after interrupt.

2. Common input commands

There commands are available for window or file devices working in read mode.

getc(^Char, ^Status, Cdr)
Reads one character and unifies it with `Char`. When input is completed successfully, `Status` is unified with `fep#normal`. If end of file is encountered, `Status` is unified with `fep#end_of_file`.

≪ ! ≫Not available on Multi-PSI V2 FEP.

getl(^Line, ^Status, Cdr)
Reads one line, converts it into an 8 bits string, then unifies it with `Line`. At this time, newline code is removed. When the input is completed successfully, `Status` is unified with `fep#normal`. If end of file is encountered, `Status` is unified with `fep#end_of_file`.

≪ ! ≫Not available for files on Multi-PSI V2 FEP.

getb(Size, ^Buffer, ^Status, Cdr)
Reads the number of bytes specified by `Size` (integer), and converts them into an 8 bits string, unified with `Buffer`. If a newline is encountered while reading from a window, input stops at newline character. When the input is completed successfully, `Status` is unified with `fep#normal`. If end-of-file is encountered, it is unified with `fep#end_of_file`.

≪ ! ≫Not available for windows on Multi-PSI V2 FEP.

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

Reads a string constructed as one term then analyzes this string to extract tokens. The list of generated tokens is unified with `TokenList`.

variable	:: `$VAR(N,String)`
atom	:: `atom(Atom)`
integer	:: `integer(Integer)`
floating point	:: `float(Float)`
string	:: `string(String)`
functor	:: `open(Atom)`
signs	:: `sign(Atom)`
special character	:: atom that has special character as print name.
illegal data	:: `illegal(String)`
end	:: end

When the input is completed successfully, `Status` is unified with `fep#normal`. If end-of-file is found, it is unified with `fep#end_of_file`. If an error occured during token analysis, `Status` is unified with `fep#abnormal`.

≪ ! ≫Not available on Multi-PSI V2 FEP.

3. Common output commands

These commands are available for window and file devices opened in write or append more.

putc(Char, ^Status, Cdr)

Writes the character corresponding to `Char` (integer) according to the ASCII code. `Status` is unified with `fep#normal`.

≪ ! ≫Not available on Multi-PSI V2 FEP.

putl(Line, ^Status, Cdr)

Writes string `Line` (8 bits string) and adds a newline character. `Status` is unified with `fep#normal`.

≪ ! ≫Not available on Multi-PSI V2 FEP.

putb(Buffer, ^Status, Cdr)

Writes string in Buffer (8 bits string). `Status` is unified with `fep#normal`.

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

Writes the term specified by `Term`, whith maximum length `Length` and maximum depth `Depth`. The part of term which exceeds Length or Depth is printed as '...'. `Status` is unified with `fep#normal`. Since this command uses output function for debugging, variables in `Term` are written like A, B, C with a symbol MRB or HOOK.

≪ ! ≫Not available on Multi-PSI V2 FEP.

4. Window control commands

These commands are available only for windows.

close(^Status)

Closes the window. `Status` is unified with `fep#normal`.

flush(^Status, Cdr)

No op. `Status` is unified with `fep#normal`. Data which have been written are automatically flushed, even if flush/2 is not issued.

beep(^Status, Cdr)

Rings the terminal bell. `Status` is unified with `fep#normal`.

clear(^Status, Cdr)

Erases contents of window. `Status` is unified with `fep#normal`.

show(^Status, Cdr)
Makes window visible. `Status` is unified with `fep#normal`. Since the window stays invisible after creation, one has to make it explicitely visible with this command.

hide(^Status, Cdr)
Makes window invisible. `Status` is unified with `fep#normal`.

activate(^Status, Cdr)
Same as show/2.

deactivate(^Status, Cdr)
Same as hide/2.

set_inside_size(Characters, Lines, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

set_size(fep#manipulator, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

set_position(X, Y, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

set_position(fep#manipulator, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

set_title(String, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

reshape(X, Y, Characters, Lines, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

reshape(fep#manipulator, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

set_font(PathName, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

select_buffer(BufferName, ^Status, Cdr)
Not available.

get_inside_size(^Characters, ^Lines, ^Status, Cdr)
Always returns `Characters` = 80, `Lines` = 20, `Status` = `fep#normal`.

get_position(^X, ^Y, ^Status, Cdr)
Always returns `X` = 0, `Y` = 0, `Status` = `fep#normal`.

get_title(^Title, ^Status, Cdr)
Returns name with which the window was created.

get_font(PathName, ^Status, Cdr)
Not available.

5. File control commands

These commands are available for only files.

close(^Status)
Closes file. `Status` is unified with `fep#normal`.

end_of_file(^Status, Cdr)
`Status` is unified with `fep#yes` when the end of file has been encountered. Otherwise, it is unified with `fep#no`.

pathname(^PathName, ^Status, Cdr)
Unifies file pathname with `PathName`. `Status` is unified with `fep#normal`.

6. Directory control command

These commands are available for only directory streams.

pathname(^PathName, ^Status, Cdr)
Unifies pathname of directory with `PathName`. `Status` is unified with `fep#normal`.

listing(WildCard, FileNameStream, ^Status, Cdr)
Unifies the list of pathnames corresponding to `WildCard` (8 bits string) with `FileNameStream`. `Status` is unified with `fep#normal`. `FileNameStream` can include a command next_file_name (^FileName, ^Status, Cdr). Then one file name (8 bits string) is returned through `FileNameStream` and `Status` is unified with `fep#normal`. When no more files are available, `Status` is unified with `fep#end_of_file`.

delete(WildCard, ^Status, Cdr)
Deletes all files corresponding to `WildCard` (8 bits string). PDSS can not recover deleted files. `Status` is unified with `fep#normal`.

undelete(WildCard, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

purge(WildCard, ^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

deleted(WildCard, ^FileNameStream, ^Status, Cdr)
Returns a stream from which deleted files corresponding to `WildCard` (8 bits string) can be extracted. This list is always empty. Status is unified with `fep#normal`.

expunge(^Status, Cdr)
No op. `Status` is unified with `fep#normal`.

Appendix-2 Code device

This device manages code. Code can be manipulated by inserting commands into device stream. (Currently, only Micro PIMOS is allowed to use code device stream, which is not available for the average user.)

assemble_module(^ModuleName, FileName, ^Status)
Assembles the file `FileName` (8 bits string) and loads it into the code area. `ModuleName` is unified with the atom named after the assembled module name. `Status` is unified with `success`, `cannot_open_file`, `memory_limit`, `module_protected` or `load_error`, depending on how the operation has been proceeded.

load_module(^ModuleName, FileName, ^Status)
Loads file specified by `FileName` into code area. File format should be either save or assembler format. `ModuleName` is unified with the atom named after the loaded module name. `Status` is unified with `success`, `cannot_open_file`, `memory_limit`, `module_protected` or `error`, depending on the course of operations.

save_module(ModuleName, FileName, ^Status)
Saves the module `ModuleName` (atom) to file `FileName` (8 bits string). `Status` is unified with `success`, `cannot_open_file` or `module_not_found`.

remove_module(ModuleName, ^Status)
Deletes module `ModuleName` (atom). `Status` is unified with `success`, `module_not_found` or `module_protected`.

debug(Flag, ^Status)
Switches debugging mode on or off. `Flag` is `on` or `off`(atom). `Status` is unified with `success` or `undefined_module`.

backtrace(Flag, ^Status)
Switches backtrace (display of deadlocked goals detected during global GC) on or off. `Flag` is atom `on` or `off`. `Status` is unified with `success` or `undefined_module`.

trace_module(ModuleName, Mode, ^Status)
Changes trace mode of the module `ModuleName` (atom) to `Mode`. Mode is `on` or `off`(atom). `Status` is unified with `success`, `module_not_found`, `undefined_mode` or `native_code_module`.

get_module_status(ModuleName, ^Mode, ^Status)
Checks trace mode of the module `ModuleName` (atom). `Mode` is unified with `on` or `off`, according to the state of trace mode. `Status` is unified with `success`, `module_not_found` or `native_code_module`.

spy_predicate(ModuleName, PredicateName, Arity, Mode, ^Status)
Changes trace mode of the predicate `PredicateName/Arity` in module `ModuleName` (atom), to `Mode`. `Mode` is atom `on` or `off`. `Status` is unified with `success`, `module_not_found`, `predicate_not_found`, `undefined_mode` or `native_code_module`.

get_spied_predicates(ModuleName, ^Predicates, ^Status)
Unifies `Predicates` with a list of information about the predicates spied in the module `ModuleName` (atom). Each element is a two-elements-vector of the form {predicate name atom, arity}. `Status` is unified with either `success` or `module_not_found`.

get_public_predicates(ModuleName, ^Public, ^Status)
Unifies `Public` with a list of information about public predicates in the module `ModuleName`(atom). Each element is a two-elements-vector of the form {predicate name atom, arity}. `Status` is unified with either `success` or `module_not_found`.

Appendix-3 PIMOS common utilities

These utility programs were developped for PIMOS, but can be used on PDSS as well. When provided modules are called, these utilities are loaded automatically by Micro PIMOS auto-load function.

PIMOS provides the following conversion and store functions are common utilities which can be used in both PIMOS and application programs. When one wishes to use these facilities, he can get the conversion result or object connection stream by calling predicates of the modules provided in PIMOS. User manipulates objects by inserting messages in this stream, through a merger.

- Comparison : a function which generates a total order upon KL1 data.
- Hashing : a classical hash function.
- Pool without key : bag, stack, queue, sorted bag.
- Pool with key : keyed bag, keyed set, keyed sorted bag, keyed sorted set.

1. Comparison

Generally, any KL1 data can compare via this mechanism.

comparator:sort(X, Y, ^S, ^L, ^Swapped)

Compares `X` and `Y`, then unifies the left hand element of the relation with `S` and the right one with `L`. If `X` = `Y`, `S` is unified with `X` and `L` with `Y` (like this relation is said to be stable). Besides, if `X` is larger than `Y`, `Swapped` is unified with `yes`, or with `no` otherwise.

Definition of the comparison relation :

If types of both data are different, order is the type order, i.e. integer, atom, string, list, vector, from left to right. Otherwise, the relation is defined as follows :

- integer···Comparison between integers.
- atom ···Comparison between atom numbers.
- string ···Lexicographic order, if strings are of the same type. Otherwise, type order.
- list ···Comparison of Car. If they are the same, comparison of Cdr, and so on.
- vector ···Comparison of the number of elements. If it is the same for both vectors, proceeds as for lists.

2. Hashing

Standard hash function via this mechanism.

hasher:hash(X, ^H, ^Y)

`H` is unified with a non negative integer holding hash result. `Y` is unified with `X`.

Hash function definition

- integer···Absolute value.
- atom ···Atom number.
- string ···Cb × first-element + Cm × middle-element + Ce × last-element + string-length. Cb, Cm and Ce are the same as KL0 built-in predicates.
- list ···Car hash value + 5 × Cdr hash value.
- vector ···number of elements + sum (for the first, middle and last elements) of ((2 to the power of element rank+ 1) × element hash value)

3. Pool without key

Any KL1 data stored via this mechanism.

Bag

A basic pool. There are only basic functions put and get. To refer an element in the pool we have to extract it, and there is no way to leave it inside pool.

pool:bag(Stream)

Generates a bag object. `Stream` is the command stream associated to it.

Message protocol :

empty(^YorN)
Returns yes(atom) if the bag is empty, no otherwise.

put(X)
Puts X into the bag.

get(^X)
Gets X from the bag. It is not possible to select a specific element. After this operation, the element is removed from the bag. If no element is in the bag, failure occurs.

get_all(^O)
O is unified with the list of all elements in the bag. If none, it is unified with □.

get_and_put(^X, Y)
Pulls out one element and unifies it with X, then puts Y in its place. If the bag is empty, failure occurs.

Stack

Basically the same as bag, but element order is LIFO.

pool:stack(Stream)
Generates a stack object. Stream is unified with the control stream.

Message protocol : Same as bag protocol.

Queue

Basically the same as bag, but element order is FIFO.

pool:queue(Stream)
Generates a queue. Stream is unified with the control stream.

Message protocol : Same as bag.

Sorted Bag

Works like a bag, but extraction order is 'least element first', according to comparison function.

pool:sorted_bag(Stream)
Generates a sorted bag object, with a standard comparator:sort/5 comparison routine. Stream is unified with the command stream.

pool:sorted_bag(Comparator, Stream)
Works the same, but comparison routine is specified by Comparator, whose format is {module name atom, predicate name atom, arity}. Sorted bag object, which has a Comparator routine, is generated. Stream is unified with the command stream. This predicate must have been declared as public, with the same arity and function as comparator:sort/5.

Message protocol :
Same as bag. get returns the least one and get_all returns a list sorted in ascending order.

Keyed pool

KL1 data are stored with a key by this mechanism.

Keyed Bag

Basic pool with a key. This is based on a hash table.

pool:keyed_bag(Stream)
Generates a keyed bag object, using the standard hash function (hasher:hash/3). Stream is unified with the command stream. Initial hash table size is 1.

pool:keyed_bag(Stream, Size)
This works the same as the previous predicate, except that hash size is given by Size.

pool:keyed_bag(Hasher, Stream, Size)
With this predicate, it is not only possible to specify hash table size, but also the hash function. `Hasher` is of the form {module name atom, predicate name atom, arity}. The corresponding predicate must have been declared as public, and have same arity and function as hasher:hash/3.

Message protocol :

empty(^YorN)
Returns `yes`(atom) if bag is empty and `no` otherwise.

empty(Key, ^YorN)
As above, but subset of elements with key `Key` is examined.

put(Key, X)
Puts `X` into the bag, using key `Key`.

get(Key, ^X)
Unifies `X` with one element with key `Key`. If there are several possible choices, one is picked up at random. After this operation, the chosen element is removed from the bag. If no element with key `Key` is in the bag, failure occurs.

get_all(^O)
`O` is unified with the list of all elements in the bag. Each item in the list is of the form {key, element}. If the bag is empty, `O` is unified with [].

get_all(Key, ^O)
`O` is unified with the list of all elements with key `Key`.

get_and_put(Key, ^X, Y)
Unifies `X` with an element with key `Key`, then replaces it with `Y`. If there is no such element, failure occurs.

Keyed Set

This works like a keyed pool, except that duplicated keys are not allowed.

pool:keyed_set(Stream)
This creates a keyed set. `Stream` is unified with the command stream. Standard hash function(hasher:hash/3) is used. Initial hash table size is one.

pool:keyed_set(Stream, Size)
This works the same, but hash table size is `Size`.

pool:keyed_set(Hasher, Stream, Size)
This works the same, but it is also possible to specify the hash function which should be used. See keyed_bag/3 predicate above.

Message protocol :

empty(^YorN)
Returns `yes`(atom) if the bag is empty, no otherwise.

empty(Key, ^YorN)
This works the same, but only the subset of elements with key `Key` is analyzed.

put(Key, X, ^OldX)
Adds an element with key `Key` and value `X`. If there is already an element with the same key, its value is updated, and `OldX` is unified with {old value}. Otherwise, `OldX` is unified with {}.

get(Key, ^X)
Unifies `X` with the element with key `Key`. If there is no such element, failure occurs. The element is removed from the set after this operation.

get_all(^O)
All elements are removed from the set, and `O` is unified with a list whose elements are of the form {key, data}. If the set was already empty, `O` is unified with [].

get_all(Key, ^O)

`O` is unified with a list containing element with key `Key`, which is removed from the set. If there is no such element, `O` is unified with $\square$.

get_and_put(Key, ^X, Y)

Replaces element with key `Key` with `Y`. Old value is return in `X`. If there is no element with such a key, failure occurs.

Keyed Sorted Bag

This is similar to sorted bag, but sort is performed only upon key.

pool:keyed_sorted_bag(Stream)

Generates a keyed sorted bag, using standards compare routine (comparator:sort/5). `Stream` is unified with the command stream.

pool:keyed_sorted_bag(Comparator, Stream)

Works the same, but `Comparator` can be used to specify the sort routine. Refer to sorted_bag/2 above, and comparator:sort/5.

Message protocol :

It is similar to the one of keyed bag, but data comes out in increasing order of key.

Keyed Sorted Set

This is similar to keyed sorted bag, but identical keys are not allowed.

pool:keyed_sorted_set(Stream)

Generates keyed sorted set object, with standard compare routine (comparator:compare/5). `Stream` is unified with command stream.

pool:keyed_sorted_set(Comparator, Stream)

Works the same but `Comparator` can be used to specify the comparison predicate. See sorted_bag/2 above, and comparator:sort/5, for more information.

Message protocol :

This is the same as the one of keyed set, but data comes out in increasing order of key.

Appendix-4 Reserved module names

The following module names are reserved by PDSS, and should not be used. Names marked with * are available.

```
  'Sho-en'
* directory
* file
* window
* mpimos_io_device
  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_libdir
  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
  pdss_runtime_active_unify
  pdss_runtime_debug
  pdss_runtime_exception_handling
  pdss_runtime_body_builtin
  kl1cmp_blttbl
  kl1cmp_command
  kl1cmp_compile
  kl1cmp_mrb
  kl1cmp_normalize
  kl1cmp_output
  kl1cmp_reader
  kl1cmp_register
  kl1cmp_macro
  kl1cmp_macro_arg
  kl1cmp_mtbl
  kl1cmp_struct
```

Appendix-5 Reserved operator names

The following operators are defined for PDSS windows and file input.

```
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   xfx   mod
 300   xfy   **
 200    fx   &
 150   xf    ++
 150   xf    --
 100   xfx   #
 100    fx   #
```

Appendix-6 List of built-in predicates

1. Type checking.

```
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

The following operator is available :  \=
```

3. Comparison

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

The following operators are available :
    =:=, =\=, <, =<, >, >=
```

4. Arithmetic Operations

```
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

:=, <= can be used with the following operators :
    +, -, *, /, mod, /\, \/, xor, <<, >>
```

5. Floating Point Comparison

```
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

The following operators are available :
   $=:=, $=\=, $<, $=<, $>, $>=
```

6. Floating Point Operations

```
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
floatin_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

$:=, $<= can be used with the following operators :
    +, -, *, /, **
```

7. Floating Point Conversion

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

8. Vectors

```
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. Strings

```
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. Atoms

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

11. Code

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

12. Stream support

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

13. Second order function

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

14. Special I/O

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

15. Others

```
raise(Tag, Type, Info) :: B
consume_resource(Reduction) :: B
hash(X, ^Value, ^NewX) :: B
current_processor(^ProcessorNumber, ^X, ^Y) :: B
current_priority(^CurrentPriority, ^ShoenMin, ^ShoenMax) :: B
```

Appendix-7 Exception codes

- Illegal Input Type :: 0
  An illegal data type appeared as an input argument of some built-in predicate.
- Range Overflow :: 1
  The range of some input argument of a built-in predicate(excluding arithmetic operation) is incorrect.
- Integer Overflow :: 3
  As a result of integer operation, overflow occurs. Zero division is included here.
- Floating Point Error :: 5
  The range of some input argument of a floating point operation is incorrect. Or as a result of floating point operation, overflow occurs.
- Illegal Merger Input :: 8
  Data different from [] or list or vector has been input through merger.
- Reduction Failure :: 9
  No candidate clause are selected for goal execution.
- Unification Failure :: 10
  Body unification has failed.
- Raised :: 12
  A built-in predicate `raise/3` is executed.
- Incorrect Priority :: 16
  Assigned priority is outside of the Sho-en bounds.
- Module Not Found :: 17
  An unloaded module has been referred to.
- Predicate Not Found :: 18
  A given predicate does not appear in the required module.
- Deadlock :: 11
  Deadlock is detected in the Sho-en.

Appendix-8 Reserved Sho-en tags

In current version, the following bits of Sho-en tag are reserved for the KL1 language and for Micro PIMOS usage.

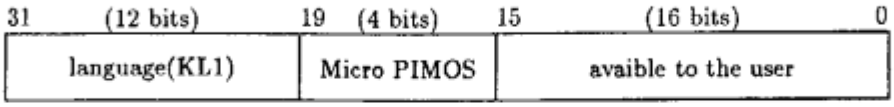

Figure 8: Sho-en exception tag

- bit 16 — I/O stream required from parent Sho-en.
- bit 17 — Error message sent to parent Sho-en.
- bit 18 — Message output on Micro PIMOS shell window.
- bit 19 — Not used.
- bit 20 — Deadlock detected.
- bit 21 — Illegal input type.
- bit 22 — Range overflow.
- bit 23 — Integer overflow.
- bit 24 — Floating point error.
- bit 25 — Not used.
- bit 26 — Illegal merger input.
- bit 28 — Unification failure.
- bit 29 — Incorrect priority.
- bit 30 — Module not found.
- bit 31 — Exception while calling a built-in predicate.

Appendix-9 GNU-Emacs library

There are two library modes in PDSS. The first one is the kl1-mode, used to edit programs, and the second is PDSS-mode, used to run PDSS. Commands defined in each mode are shown below :

1. kl1-mode

ctrl-C ctrl-C
Compiles all the text in the buffer in which command has been executed, as if this text was KL1 source code.

ctrl-C ctrl-R
Copies specified range of text in the buffer `PDSS=COMPILER`.

ctrl-C ctrl-D
Compiles the contents of the buffer `PDSS=COMPILER` as a KL1 program. Then, looks for the assembler file(*.asm) which has the same name as the buffer and updates parts of this file which have changed. Eventually, generates save file.
This command is used with **ctrl-C ctrl-R** to recompile updated parts only. Assembly files should therefore not be deleted.

meta-X pdss-kl1cmp-switch-macro-mode

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

meta-X pdss-kl1cmp-switch-debug-mode

meta-X pdss-kl1cmp-switch-system-mode
Changes options of the Prolog version compiler. Commands with no argument work as toggle switches, while arguments 1/0 corresponds to on/off. Detailed meaning and initial values of these options are described in Appendix-10. Above commands correspond to `e`, `i`, `d` and `s` options, resp.
When using KL1 version compiler, these commands are not available.

≪ ! ≫Information and prompt are output in buffer `PDSS=COMPILE`, but basically, user does not need to type anything in this buffer.

ctrl-C ctrl-F
Displays the manual of built-in predicates.

2. PDSS-mode

a. Window/buffer operations

meta-.
Displays a candidate string, beginning with `?-` and matches the previous string which has been entered. This is used to repeat the last interpreted command.

ctrl-C ctrl-Y
Redisplays previous input.

ctrl-C k
Deletes all text in the buffer.

ctrl-C ctrl-K
Deletes all text in all PDSS-mode buffers.

ctrl-C ctrl-B
Displays buffer menu of PDSS-mode buffer.

ctrl-C m
Looks for the pattern module-name:predicate-name from the beginning of current line, and start insert at its current position. This is convenient to set variable name when setting variable monitor in the tracer.

ctrl-C ctrl-F
Displays built-in predicate manual.

ctrl-C f
Displays command manual for command interpreter.

ctrl-X k
Kills buffer, but gives a warning if PDSS is running.

b. KL1 program control

ctrl-C ctrl-Z
Inserts 1 into the attention stream of the KL1 window process that corresponds to current buffer. This is treated as a task stop request by Micro PIMOS.

ctrl-C ctrl-T
Puts 2 in the same buffer as above. This causes display of statistic information from Micro PIMOS.

c. Emulator control

ctrl-C !
Garbage collection request.

ctrl-C @
Stops PDSS system, but the buffers used as Micro PIMOS windows are left untouched.

ctrl-C ESC
Restarts PDSS.

3. Mode independent command

ctrl-C ctrl-P
Displays next PDSS-mode buffer in current window. The PDSS buffer group is managed as a circular list. So, if user repeats this command, all buffers are displayed one by one.

ctrl-C p
This is almost the same as the previous command, but display occurs in the other window.

Appendix-10 Using command procedures for compiling

This is the description of the command procedure to compile a KL1 program, used as a UNIX command. It may be useful to compile it within a makefile. There are two versions of this command : one for the KL1/KL1 compiler and the other for the KL1/Prolog compiler. Basic usage rules are the same, but some available options are different.

Command :

pdsscmp [options] file names ...

Options :

`+i / -i` :: Indexing code is generated or not. Default is not to generate it. KL1/Prolog only.

`+m / -m` :: Code for MRB-GC is generated or not. Default is to generate it.

`+a / -a` :: Assemble is performed or not. Default is to perform it. When performed, an assembler file (xxx.asm) and a save file (xxx.sav) are generated. Otherwise, only assembler file is generated.

`+s / -s` :: Compiles for Micro PIMOS or for user. Default is to compile for user. System-mode private built-in predicates can be used in the first case. Built-in predicates in this manual can be compiled with the user version. KL1/Prolog only. (All built-in predicates can be compiled in KL1/KL1.)

`-o=PATH` :: Changes output directory to `PATH`. Current working directory is the default.

File name :

xxx.asm :: Assembles an assembler-file (xxx.asm) and creates a save file (xxx.sav).

xxx.kl1 :: Compiles a source file (xxx.kl1), makes assembler file and then assembles it to make save file.

xxx :: same as xxx.kl1.

Examples :

- To compile and assemble the two source files append.kl1 and queen.kl1, and then to make append.asm, append.sav, queen.asm and queen.sav in the current directory :

```
pdsscmp  append.kl1 queen.kl1    or
pdsscmp  append queen
```

- To compile and assemble append.kl1 and assemble queen.asm :

```
pdsscmp  append.kl1 queen.asm
```

- To compile and assemble all .kl1 files in the directory `source` and then to put assemble and save files in directory `object`:

```
pdsscmp  -o=object source/*.kl1
```

Appendix-11 Sample program

```
:- 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(_)].


| ?- 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.
```

Appendix-12 What to do if a bug is found out...

1. When you find system bugs, please inform the PDSS development group. E-mail address is :

   pdss@icot21.icot.junet

   In your mail, include the following information :

   a. PDSS (emulator, Micro PIMOS) version number.

   b. Compiler version number.

   c. The program in which you found the bug.

   d. How to start it, and what happens.

   e. Execution log and queer points.

2. If it is a bug of a program of your own, go at the least through the following list :

   a. Have you done varchk?

   b. In case of deadlock, if there are the following goals in the incriminated part, you may have forgotten to close command stream to file or window, or you may have requested the output of undefined variables. Check your code.

```
mpimos_file:xxxxxx( ... )      or
mpimos_window:xxxxxxx( ... )
merge( ... ) in mpimos_file:xxxxx/x
merge( ... ) in mpimos_window:xxxxx/x
```

Index

I – 2 PDSS Manual

(in Japanese)

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 "klicmp_" 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 kli-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。

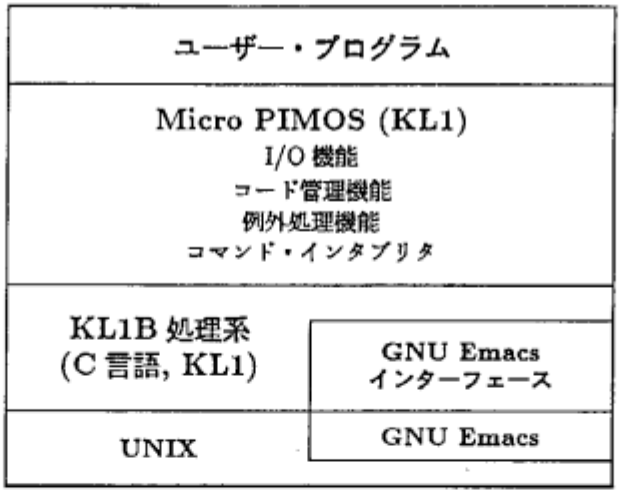

図 1: PDSS の全体構成

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: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]?- 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。

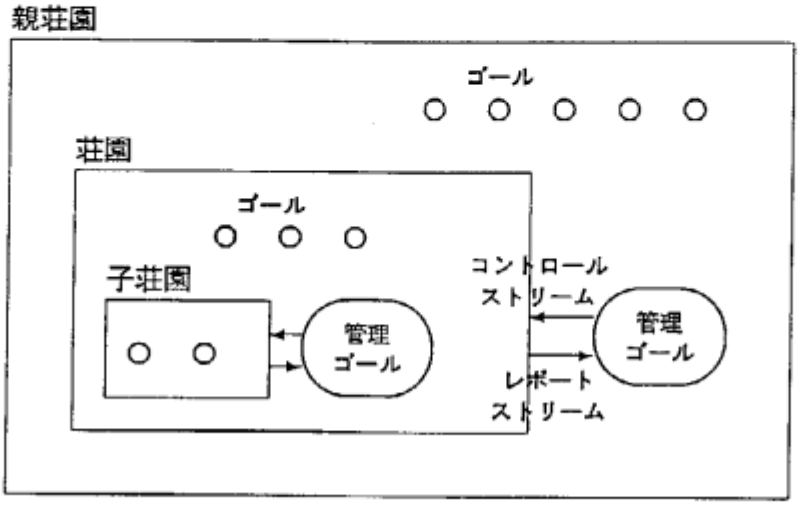

図 2: 荘園のイメージ

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。

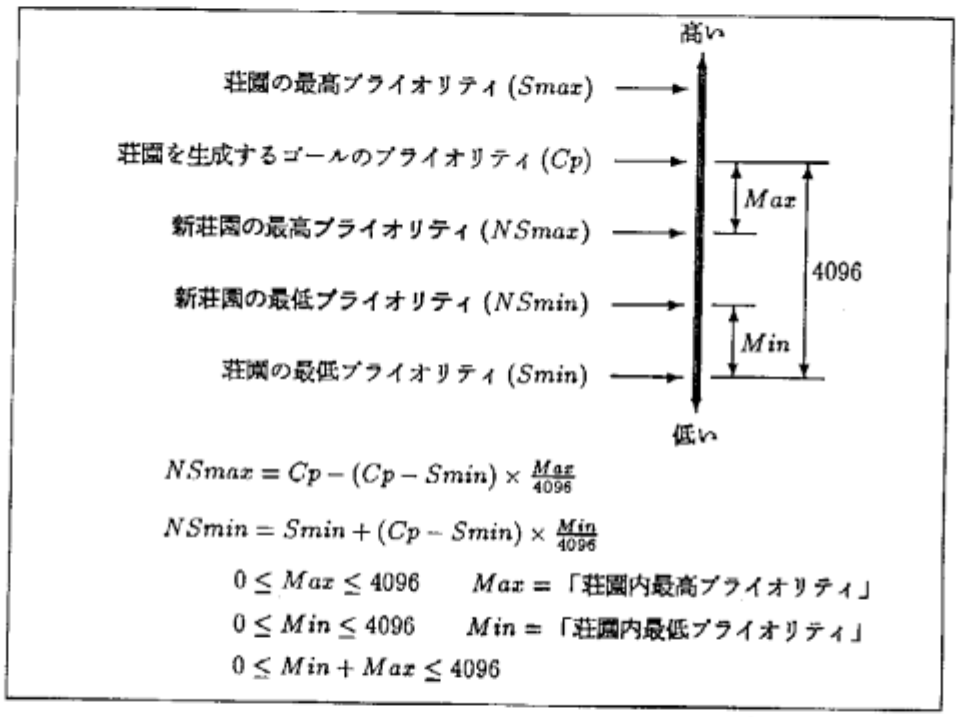

図 3: 荘園のプライオリティの計算

「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。

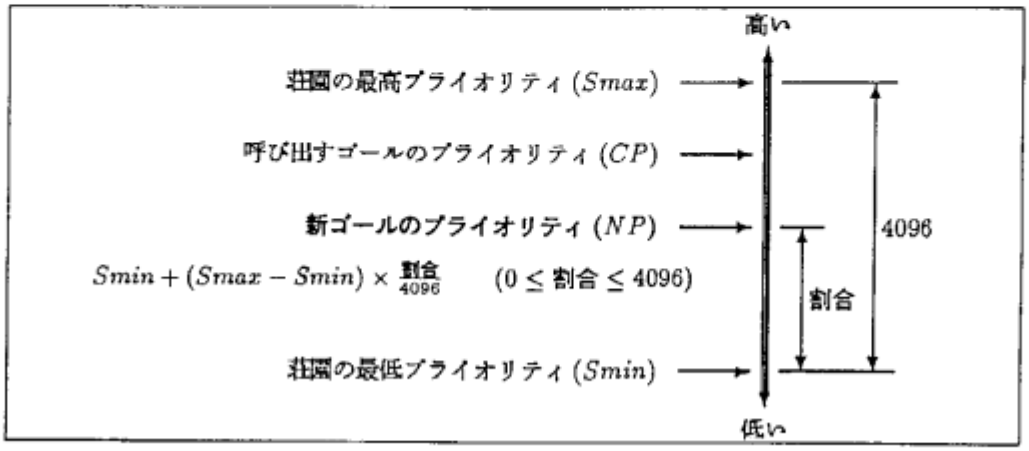

図 4: 所属荘園内割合指定によるプライオリティの計算

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。

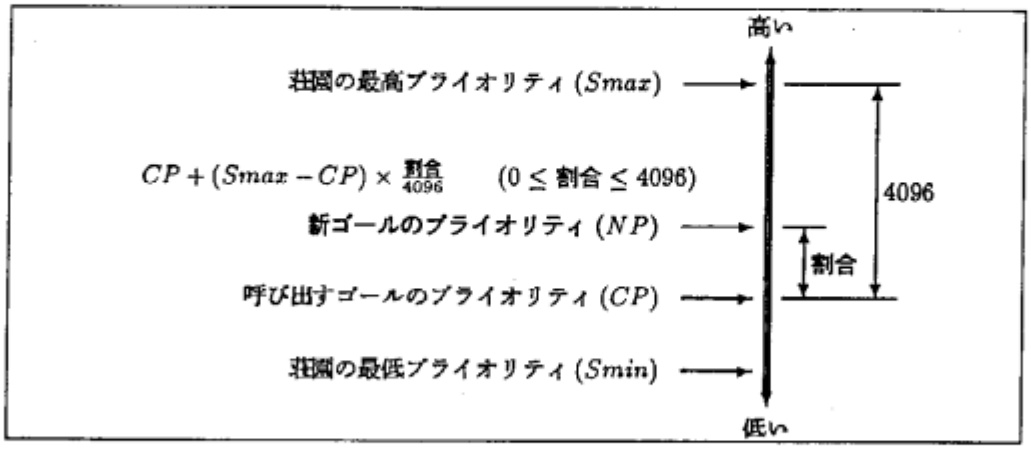

図 5: 所属荘園内自己相対指定によるプライオリティの計算 (正)

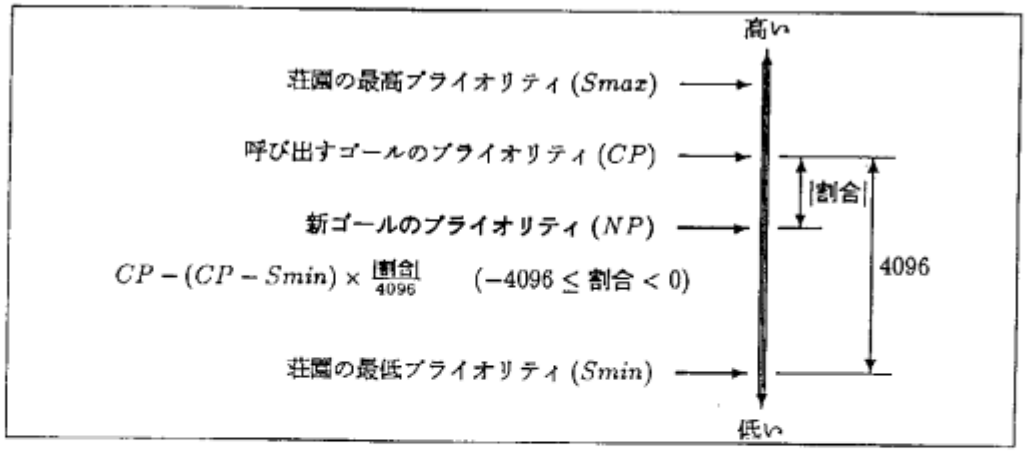

図 6: 所属荘園内自己相対指定によるプライオリティの計算 (負)

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。

```
vector(X, ^Size)   ::   G
      ↑                  ↑
   呼び出し形式        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。

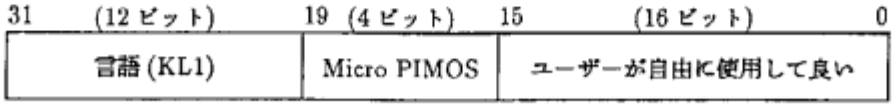

図 7: 荘園の例外タグ

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 コマンドとほぼ同じであるが、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 とほぼ同じだが、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 コマンドとほぼ同じ。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 とほぼ同じだが、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 とほぼ同じだが、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(^FileNam ^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
kl1cmp_blttbl
kl1cmp_command
kl1cmp_compile
kl1cmp_mrb
kl1cmp_normalize
kl1cmp_output
kl1cmp_reader
kl1cmp_register
kl1cmp_macro
kl1cmp_macro_arg
kl1cmp_mtbl
kl1cmp_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 口, 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。

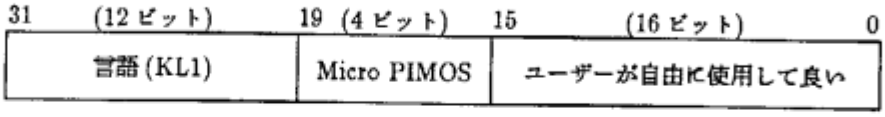

図 8: 荘園の例外タグ

- 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

I – 3 PDSS Installation Manual

(in English)

17/09/1990
ICOT 1st. Lab.

PDSS (V2.52) Installation Manual

PDSS "PIMOS Development Support System" is the KL1 system on an ordinary (not multi-processor) UNIX machine. PDSS was developed by ICOT as an environment for PIMOS development. PDSS includes compiler, byte-code interpreter and Micro-PIMOS (command interpreter, I/O libraries, ...). PDSS also includes two emacs lisp libraries: KL1-mode and PDSS-mode. KL1-mode is the major mode for editing KL1 source code. PDSS-mode is the major mode for interacting with PDSS through multiple emacs buffers. In ICOT, We are using PDSS on SUN-3, SUN-4 and Sequent Symmetry, and you can easily install PDSS on BSD 4.2, 4.3 UNIX system.

1 Contents of PDSS tape

PDSS tape contains following directories and files. You must provide 7MB of file space to read PDSS from tape, and approximately 16MB for full install. If you cannot provide so much file space, see next section.

	name	contents	size
	Makefile	Makefile.	2
	pdsscmp	Shell script to compile KL1 program.	6
	emulator	KL1-B byte code interpreter = abstract KL1 machine.	946
	runtime	Runtime support libraries written in KL1.	139
	mpimos	Micro PIMOS = Command interpreter and I/O libraries.	1554
	compiler	KL1 compiler written in KL1.	1758
	macro	KL1 macro definition files.	12
+	debug_utl	Tool programs to test KL1 source code.	280
+	pimos_utl	Library programs developed by PIMOS group.	184
+	compiler_pl	KL1 cross compiler written in SICStus Prolog.	359
+	emacs	Emacs Lisp libraries. KL1-mode and PDSS-mode.	87
+	test	KL1 test programs.	549
+	sample	KL1 sample programs.	124

† This "+" mark means optional module. KB

2 Restore PDSS from tape

Since PDSS tape was written in tar format, execute following command to restore it. By this command, PDSS will be restored into the new directory named "pdss" under current directory.

```
% tar xf /dev/rts0 pdss
          ^^^^^^^^^
```

Device name of the tape drive. This name depends on machine and media. If you don't know it, refer to the manual of your machine.

If your system doesn't have so much file space, execute following command instead of above one. This command doesn't read source code and assembly language code of KL1 programs, but reads only object code for KL1 machine emulator. This procedure is applicable for the installation onto CISC machines, e.g. 680X0 or 80386. See the comment about `KLB_PACKED_CODE` in "emulator/config.h" file.

```
% tar xf /dev/rts0 FILES.SMALL
% tar xf /dev/rts0 'cat FILES.SMALL'
```

After restore from tape, it is IMPORTANT that you must rename or remove "compiler_pl" directory before making PDSS when you don't have SICStus Prolog. When you don't have GNU Emacs, you must rename or remove "emacs" directory. And you may rename or remove other unnecessary optional module directories which you don't need.

3 Edit the makefile and configuration-file

You must edit some files before make PDSS, since these files contain descriptions depending on machines or site.

- Makefile
  Change `BINDIR`'s value to suit your site. It is used as a target directory of "make install".
- emulator/Makefile
  Change `BINDIR`'s value to suit your site. It is used as a target directory of "make install".
  Change `CFLAGS`'s value if you need.
- emulator/config.h
  You must change some definitions according to comments in the file. When your machine uses CISC type CPU (680X0, 80386), you don't need to change others but path names. But, if your machine use RISC type CPU, you must change `KLB_PACKED_CODE`'s value to `NO`.
- compiler_pl/libdir.pl (When you use cross compiler written in Prolog)
  Change path name to suit your site.
- emacs/Makefile (When you use emacs lisp libraries)
  Change `LIBDIR`'s value to suit your site. It is used as a target directory of "make install".
  Change `EMACS`'s value to suit your site. It is a path name of GNU Emacs used to compile libraries.
- emacs/pdss-init.el (When you use emacs lisp libraries)
  Change `pdss-directory-name`'s value to suit your site. It is a path name of the PDSS toplevel directory.

- pdsscmp Change path names at top 5 lines to suit your site.

4 Make

Now, make PDSS. If you don't change `KLB_PACKED_CODE`'s value nor `KLB_4BYTE_REL_ADDR`'s value, execute following command in PDSS toplevel directory.

```
% make quick
```

But, when you change these value, execute following command. By this command, all KL1-assember codes will be re-assembled.

```
% make reassm
```

When make command has finished successfully, execute following command to copy executable files to shared "bin" directory.

```
% make install
```

And claen PDSS directory if you want.

```
% make clean
```

Refer to a log of PDSS making in appendix-I. It was made on SUN SparcStation-1 (SunOS 4.0.3c) in ICOT.

5 Emacs LISP libraries

PDSS includes two emacs lisp libraries: KL1-mode and PDSS-mode. KL1-mode is the major mode for editing KL1 source code. PDSS-mode is the major mode for interacting with PDSS through multiple emacs buffers. When you want to use these libraries, insert following command into ".emacs" file in your home directory. If you want to set up the environment as all users in your machine can use these libraries, insert same command into "site-init.el" file of your machine.

```
(load "pdss-init")
```

The version of GNU Emacs used in ICOT is Nemacs version 3.2.3 (Nihon-go GNU Emacs, which is Japanese version of Emacs based on 18.55.31), so a few changes might be required if your Emacs version is different.

6 When your machine is not BSD UNIX

Since PDSS was written for BSD UNIX, you must make many changes to install PDSS on System-V UNIX. You must edit at least following files.

- emulator/pdss.c
  `signal()`, `sigblock()`, `sigsetmask()` and `kill()` are used by signal handler.
  `getrusage()` is used to get cpu-time information.
- emulator/io.c
  `signal()`, `sigblock()`, `sigmask()` and `sigsetmask()` are used by signal handler.
  `ioctl()` and `fcntl()` are used to control I/O device.

- emulator/blt_system.c

  `getrusage()` is used to get cpu-time information.

- emulator/blt_iodev.c

  `opendir()`, `clausedir()` and `readdir()` are used to read directory.

  `unlink()` is used to remove file.

  `re_comp()` and `re_exec()` are used for regular expression.

- emulator/timer.c

  `signal()`, `sigblock()`, `sigmask()` and `sigsetmask()` are used by signal handler.

  `alarm()` is used for timer interrupt.

  `time()` is used to get current time.

Appendix-I Log of PDSS makeing in ICOT

This is a log of makeing PDSS on SUN SparcStation-1 (SunOS 4.0.3c) in ICOT.

```
%
% ls                                    ## Now, I am in PDSS directory.
LOG             debug_utl/      emulator/       pimos_utl/
Makefile        doc.e/          macro/          runtime/
compiler/       doc.j/          mpimos/         sample/
compiler_pl/    emacs/          pdsscmp*        test/
%
% mv emacs emacs.notuse                ## Since I don't use emacs,
%                                       ## I rename this directory.
% mv compiler_pl compiler_pl.notuse     ## Since I don't use SICStus Prolog,
%                                       ## I rename this directory.
%
%                                       ## Edit some files here.
%
% make reassm                           ## Since I change KLB_PACKED_CODE,
cd emulator     ; make all              ## I execute "make reassm".
touch pdss.h
cc -O  -sun4 -c  pdss.c
cc -O  -sun4 -c  option.c
cc -O  -sun4 -c  emulate.c
cc -O  -sun4 -c  blt_basic.c
cc -O  -sun4 -c  blt_float.c
cc -O  -sun4 -c  blt_shoen.c
cc -O  -sun4 -c  blt_iodev.c
cc -O  -sun4 -c  blt_code.c
cc -O  -sun4 -c  blt_system.c
cc -O  -sun4 -c  deref.c
cc -O  -sun4 -c  passive.c
cc -O  -sun4 -c  unify.c
cc -O  -sun4 -c  goal.c
cc -O  -sun4 -c  shoen.c
cc -O  -sun4 -c  exception.c
cc -O  -sun4 -c  memory.c
cc -O  -sun4 -c  atom.c
cc -O  -sun4 -c  module.c
cc -O  -sun4 -c  dcode.c
cc -O  -sun4 -c  float.c
```

```
cc -O  -sun4 -c  string.c
cc -O  -sun4 -c  gc.c
cc -O  -sun4 -c  gc_ctrl.c
cc -O  -sun4 -c  gc_cell.c
cc -O  -sun4 -c  gc_code.c
cc -O  -sun4 -c  gc_dead.c
cc -O  -sun4 -c  mrbgc.c
cc -O  -sun4 -c  instr.c
cc -O  -sun4 -c  invassm.c
cc -O  -sun4 -c  assemble.c
cc -O  -sun4 -c  saveload.c
Routine  _write_predicate  too big:              ## Swap area shortage.
        use a lower level of optimization or     ## (ICOT's SUN have
        increase stack limit and / or            ##  small swap area.)
        size of swap space

compiler(iropt) error:   alloca: out of memory
*** Error code 1
make: Fatal error: Command failed for target `saveload.o'
Current working directory /usr2/pdss/pdss2.5/emulator
*** Error code 1
make: Fatal error: Command failed for target `reassm'
% (cd emulator; cc  -sun4 -c  saveload.c)        ## Retry to compile.
/usr2/pdss/pdss2.5/emulator
% make reassm                                    ## Continute "make reassm".
cd emulator     ; make all
cc -O  -sun4 -c  native.c
cc -O  -sun4 -c  tracer.c
cc -O  -sun4 -c  print.c
cc -O  -sun4 -c  io.c
cc -O  -sun4 -c  iosub.c
cc -O  -sun4 -c  timer.c
cc -O  -sun4 -c  path.c
cc -O  -sun4 -c  ctype.c
cc -O -o pdss.x -D"MAKEDATE=\"`date`\"" version.c pdss.o option.o emulate.o blt
_basic.o blt_float.o blt_shoen.o blt_iodev.o blt_code.o blt_system.o deref.o pa
ssive.o unify.o goal.o shoen.o exception.o memory.o atom.o module.o dcode.o flo
at.o string.o gc.o gc_ctrl.o gc_cell.o gc_code.o gc_dead.o mrbgc.o instr.o inva
ssm.o assemble.o saveload.o native.o tracer.o print.o io.o iosub.o timer.o path
.o ctype.o -lm
mv -f pdss.x pdss
rm -f version.o
cc -O  -sun4 -c  pdssasm.c
cc -O -o pdssasm.x -D"MAKEDATE=\"`date`\"" version.c pdssasm.o memory.o atom.o
module.o instr.o assemble.o saveload.o float.o ctype.o -lm
mv -f pdssasm.x pdssasm
rm -f version.o
cc -O  -sun4 -c  pdssmerge.c
cc -o pdssmerge.x pdssmerge.o
mv -f pdssmerge.x pdssmerge
sh -c "test ! -d compiler_pl || (cd compiler_pl ; make all)"
cd runtime      ; make reassm
for f in `echo coddev.sav windev.sav fildev.sav timdev.sav shoen.sav | sed 's/\
.sav//g'`; do ../emulator/pdssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file coddev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file windev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file fildev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file timdev... OK
```

```
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file shoen... OK
cd mpimos       ; make reassm
for f in `echo boot.sav op.sav optbl.sav  parser.sav unparser.sav shell.sav mac
ro_ex.sav  task_monitor.sav utl.sav code_man.sav module_pool.sav  window.sav wi
ndow_man.sav window_drv.sav  file.sav file_man.sav file_drv.sav  dir.sav dir_dr
v.sav timer_man.sav  dummy_window.sav dummy_file.sav dummy_dir.sav  builtin.sav
 cmd_basic.sav cmd_code.sav cmd_debug.sav  cmd_dir.sav cmd_env.sav cmd_utl.sav
pool_mli.sav  mpimos_iodev.sav mpimos_windev.sav  mpimos_fildev.sav mpimos_timd
ev.sav | sed 's/\.sav//g'`; do ../emulator/pdssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file boot... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file op... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file optbl... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file parser... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file unparser... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file shell... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file macro_ex... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file task_monitor... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file utl... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file code_man... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file module_pool... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file window... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file window_man... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file window_drv... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file file... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file file_man... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file file_drv... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dir... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dir_drv... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file timer_man... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dummy_window... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dummy_file... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dummy_dir... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file builtin... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_basic... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_code... OK
```

```
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_debug... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_dir... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_env... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_utl... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file pool_mli... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_iodev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_windev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_fildev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_timdev... OK
cd compiler     ; make reassm
for f in `echo blt.sav com.sav comp.sav mrb.sav norm.sav outp.sav reader.sav  r
eg.sav macarg.sav macro.sav mactbl.sav struct.sav | sed 's/\.sav//g'`; do ../em
ulator/pdssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file blt... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file com... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file comp... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mrb... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file norm... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file outp... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file reader... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file reg... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file macarg... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file macro... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mactbl... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file struct... OK
sh -c "test ! -d debug_utl   || (cd debug_utl   ; make reassm)"
for f in `echo mpimos_xref.sav mpimos_xref_table.sav  mpimos_pretty_printer.sav
 mpimos_varchk.sav | sed 's/\.sav//g'`; do ../emulator/pdssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_xref... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_xref_table... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_pretty_printer... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_varchk... OK
sh -c "test ! -d pimos_utl   || (cd pimos_utl   ; make reassm)"
for f in `echo comparator.sav hasher.sav keyed_bag.sav keyed_sorted_bag.sav  po
ol.sav queue.sav sorted_bag.sav stack.sav | sed 's/\.sav//g'`; do ../emulator/p
dssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
```

```
Make save file comparator... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file hasher... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file keyed_bag... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file keyed_sorted_bag... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file pool... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file queue... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file sorted_bag... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file stack... OK
sh -c "test ! -d emacs       || (cd emacs       ; make all)"
%
%                                        ## Make command has completed.
%                                        ## Now, I test PDSS.
% cd emulator
/usr2/pdss/pdss2.5/emulator
% pdss
***** PDSS-KL1 V2.52.06 (Mon Jul 16 17:53:41 JST 1990) *****
.................................................

*****************************************
**** Micro PIMOS (version 2.5) **********
*****************************************
World is /usr2/pdss/pdss2.5/emulator
Total heap size is 200000 words
Total code size is 274688 bytes
*****************************************

| ?- compile(sample).
"sample" : Compiling ... primes/0,primes/2,gen/3,sift/2,filter/3,display/1,END.

[primes/0]
successdone
Loading ... done
Module name : sample
Saving ... done

yes.
| ?- stat(sample:primes).
[2,3,5,7,11,13,17,19,23,29,31,37,...]
Total of 5880 reductions in 290 msec.

yes.
| ?- halt.

%
```

Appendix-II Sample of "emulator/config.h" file

Following was emulator/config.h file used in ICOT.

```
/*************************************************************************
PDSS の為のディレクトリ / ファイル環境の指定。
>> PDSS_LIBDIR: PDSS が立ち上がる時に、ランタイムライブラリを探しにいくディ
      レクトリのパス名。通常は PDSS のトップレベルのディレクトリと同じ。
>> PDSS_SUPDIR: 立ち上げ用ファイル STARTUP が置いてあるディレクトリのパス名。
      STARTUP ファイルには、立ち上げ時に読み込むべき KL1 モジュールのファ
      イル名と、最初のゴール名が記述してある。
>> PDSS_OBJECT_FILE: エミュレータ本体のパス名。Native Code Module を動的に
      リンクする時に使われる。（しかし、現在 Native Code Generator は
      サポートされていない。）
Configuration of the directory/file environment for PDSS.
>> PDSS_LIBDIR: A path name of the directory which PDSS search for runtime
      libraries at initial startup time. Normally, this is equal to the
      PDSS toplevel directory.
>> PDSS_SUPDIR: A path name of the directory which contains STARTUP file.
      STARTUP file describes path names (relative path from PDSS_LIBDIR)
      of the runtime library which are read at initial startup time, and
      initial goal name.
>> PDSS_OBJECT_FILE: A path name of the PDSS byte-code interpreter executable
      object. It is uesd by dynamic native code module linker.
      (But, native code generator is not supported yet.)
*************************************************************************/
#define PDSS_LIBDIR      "/usr/local3/pdss/release"
#define PDSS_SUPDIR      "/usr/local3/pdss/release/emulator"
#define PDSS_OBJECT_FILE "/usr/local3/pdss/release/emulator/pdss"

/*************************************************************************
PDSS のコードの形式に関する指定。
>> KLB_PACKED_CODE: これが YES の場合には、KL1-B のコードは詰め込まれた
      状態で保持される。この場合、整数等の複数バイトのデータもワードの境界
      に合わせずに詰めて書き込まれる。） マシンの CPU が RISC タイプの場合
      には、これを NO にしておく必要がある。
>> KLB_4BYTE_REL_ADDR: これが YES の場合には、KL1-B のコードにおける相対
      アドレスは 4byte で表現される。そうでなければ 2byte で表現される。
Configuration for instruction trace.
>> KLB_PACKED_CODE: If this switch's value is 'YES', KL1-B instruction
      code (byte-code) is filled in continuously. In this case, multiple
      byte data (e.g. integer) is written not in word alignment. If your
      machine uses RISC type CPU, you must define this switch 'NO'.
>> KLB_4BYTE_REL_ADDR: If this switch's value is 'YES', PDSS use 4 byte
      relative address instead of 2 byte one.
*************************************************************************/
#define KLB_PACKED_CODE    YES
#define KLB_4BYTE_REL_ADDR NO

/*************************************************************************
PDSS が使用するメモリのデフォルトの大きさの指定。
>> HEAP_SIZE: ヒープ領域の大きさ。単位はタグ付きワード (8 バイト)。
      PDSS ではコピー方式の GC を行うので、ここでは全ヒープ領域の半分の
      大きさを指定する。
>> CODE_SIZE: コード領域の大きさ。単位はバイト。ここで指定するのはコード全体
      の大きさである。PDSS は立ち上げ時に、ランタイムライブラリを読み込む
      ために先頭から使用し、その残りを半分に分けてユーザー用として使用する。
Configuration for the default memory size.
>> HEAP_SIZE: Size of head area. This specifies the number of tagged-words
      (8 byte) in the HALF of heap area, since PDSS use copying GC.
>> CODE_AREA: Size of code area. This specifies the number of bytes in the
      whole code area. PDSS loads runtime libraries into top of the code
      area at initial startup time, then PDSS divide code area into 2
```

```
      parts and use divided one for user codes.
*****************************************************************************/
#define HEAP_SIZE  200000
#define CODE_SIZE 1000000

/*****************************************************************************
MRB-GC に関する設定。
>> MRBGC_SWITCH: これが YES の場合に MRB による実時間 GC を行う。
>> MRBGC_COLLECT_VALUE_DEPTH: Collect Value 命令で回収する構造体の深さ。
>> MRBGC_DEREF_TYPE: デレフ時の回収のタイプ。
       0 --> 参照パスで一旦 MRB=ON になった先の回収を行わない。
       1 --> 参照パスで一旦 MRB=ON になった先の回収も行う。
>> MRBGC_FREE_LIST_TYPE: メモリ再利用の為のフリーリストの種類。
       0 --> 1W,2W,3W, ... ,7W,8W,16W,32W の 10 種類を用意する。
       1 --> 1W,2W,4W,8W,16W,32W,64W,128W の 8 種類を用意する。
>> MRBGC_STATISTICS: MRB-GC に関する統計情報の集計。
       0 --> 集計は行わない。
       1 --> フリーリストの使用状況の集計を行う。
       2 --> 1 + 命令ごとの回収状況の集計も行う。
Configuration for MRB-GC.
>> MRBGC_SWITCH: If this switch's value is 'YES', PDSS collect garbage
      incrementally by the MRB scheme in real time.
>> MRBGC_COLLECT_VALUE_DEPTH: The maximum depth of the nested data structure
      which collected by collect_value instruction.
>> MRBGC_DEREF_TYPE: The switch of garbage collection in dereference routine.
      0 -> If MRB-ON pointer is found in reference path, PDSS don't try
           to collect garbage in the forward path of this pointer.
      1 -> In spite of MRB-ON pointer is found in reference path, PDSS
           try to collect garbage in the forward path of this pointer.
>> MRBGC_FREE_LIST_TYPE: The variation of free lists.
      0 -> Use 10 kinds (1W,2W,3W,...,7W,8W,16W and 32W) of free lists.
      1 -> Use 8 kinds (1W,2W,4W,8W,16W,32W,64W and 128W) of free lists.
>> MRBGC_STATISTICS: The switch of statistics of MRB-GC.
      0 -> PDSS don't collect statistics information of MRB-GC.
      1 -> PDSS collect statistics information about free list only.
      2 -> PDSS collect statistics information about free list and
           incremental garbage collecting instructions.
*****************************************************************************/
#define MRBGC_SWITCH               YES
#define MRBGC_COLLECT_VALUE_DEPTH 100
#define MRBGC_DEREF_TYPE            1
#define MRBGC_FREE_LIST_TYPE        0
#define MRBGC_STATISTICS            0

/*****************************************************************************
MRB を利用したデッドロック(無限待ち)の eager detection に関する設定。
>> EAGER_DEADLOCK_DETECTION:
      これが YES の場合には eager detection を行なう。
Configuration for eager detection of deadlock (perpetual suspension).
>> EAGER_DEADLOCK_DETECTION:
      If this switch's value is 'YES', PDSS deetect deadlock eagerly.
*****************************************************************************/
#define EAGER_DEADLOCK_DETECTION YES

/*****************************************************************************
Ready Goal Pool に関する設定。
>> TWO_WAY_READY_GOAL_POOL: これが YES の場合には、先頭と最後の２つの入口を
      持った Ready Goal Pool を使用する。即ち、スタックとキューの両方に
      使えるので、この時には、ゴールのスケジューリングを Breadth First
      に変更する事もできる。NO の場合には、スタック方式のみになる。
Configuration for ready goal pool.
>> TWO_WAY_READY_GOAL_POOL: If this switch's value is 'YES', PDSS use ready
```

```
      goal pool with two entry point: top and last. In this case, ready
      goal pool can be used as stack or queue, and user can select depth
      first or breadth first scheduling. If it's value is 'NO', reday goal
      pool is stack and scheduling is depth first only.
*****************************************************************************/
#define TWO_WAY_READY_GOAL_POOL YES

/*****************************************************************************
コピー GC 時に異常事態になった時の処理に関する設定。
>> GC_SWAP_GOALS_WHEN_PANIC: これが YES の場合には、コピー GC でヒープ領域
      があまり回収できなかった時に、ゴール・スタックの highest プライオリティ
      の中のゴールの順番を入れ替える。これにより、実行順序が変わるので
      ヒープが回収できるようになる可能性がある。
>> GC_ABORT_TASK_WHEN_PANIC: これが YES の場合には、コピー GC でヒープ領域
      がほとんど回収できなかった(GC 直後なのに GC 要求フラグが立っている)時に、
      ユーザーのタスクをアボートする。これにより、PDSS 全体が異常終了する
      のを防ぐ。
Configuration for emergency measures of copying GC.
>> GC_SWAP_GOALS_WHEN_PANIC: If this switch's value is 'YES', PDSS swap
      goals in ready goal pool when copying GC get only small gain.
      By this goal swapping, the sequence of goal execution is changed and
      some data on heap may be consumed and collected.
>> GC_ABORT_TASK_WHEN_PANIC: If this switch's value is 'YES', PDSS abort
      user task when copying GC get little gain. By this abortion, many
      heap uesd by user task are released and whole PDSS may not abort.
*****************************************************************************/
#define GC_SWAP_GOALS_WHEN_PANIC YES
#define GC_ABORT_TASK_WHEN_PANIC YES

/*****************************************************************************
アトムテーブルに関する設定。
>> MAX_ATOMS: PDSS で使用できるアトムの数。
Configuration for atom table.
>> MAX_ATOMS: Maximum number of atoms.
*****************************************************************************/
#define MAX_ATOMS 8000

/*****************************************************************************
整数演算に関する設定。
>> IGNORE_INTEGER_OVERFLOW: この値が YES ならば、組込述語における整数演算
      のオーバーフローを無視する。
>> IGNORE_INTEGER_OVERFLOW: If this value is 'YES', PDSS ignores integer
      overflow in builtin predicates.
*****************************************************************************/
#define IGNORE_INTEGER_OVERFLOW NO

/*****************************************************************************
パッシブユニフィケーションに関する設定。
>> PASSIVE_UNIFY_DEPTH: この値は Passive Unification (wait_value, diff)
      で構造体同士を比較する深さを指定する。この深さまで調べても成功が確定
      しない場合には失敗と見做す。
Configuration for passive unification.
>> PASSIVE_UNIFY_DEPTH: This is the maximum depth of the nested structures
      which compare in passive unification (wait_value, diff). If PDSS
      can't confirm the result of unification while PDSS compare until
      this depth, PDSS regards the unification as failure.
*****************************************************************************/
#define PASSIVE_UNIFY_DEPTH 100

/*****************************************************************************
コンソール出力に関する設定。
>> PRINT_LENGTH, PRINT_DEPTH: コンソールへのターム出力における、長さと深さ
```

```
      の制限の初期値。これを越える部分は "..." で出力する。
Configuration for console outputs.
>> PRINT_LENGTH, PRINT_DEPTH: These are initial maximum length and depth
      of structure which can be displayed in the console. PDSS print the
      part over this limitation link "...".
*****************************************************************************/
#define PRINT_LENGTH 12
#define PRINT_DEPTH  6

/*****************************************************************************
パーザーに関する設定。
>> MAX_PRINTABLE_STRING_ATOM_LENGTH: ウィンドウ / ファイル等に出力可能な文字
      列 / アトムの長さの最大長。文字列 --> アトム の変換もこの制限を受ける。
Configuration for parser.
>> MAX_PRINTABLE_STRING_ATOM_LENGTH: The maximum length of atom/string
      which can be displayed into window/file.  This limitation is also
      used for atom-string conversion.
*****************************************************************************/
#define MAX_PRINTABLE_STRING_ATOM_LENGTH 10000

/*****************************************************************************
組込述語のサスペンドに関する設定。
>> COUNT_DCODE_REDUCTION: この値が 'YES' ならば、組込述語がサスペンドし、
      ゴール (D-Code) がエンキューされた時に、そのゴールの分のリダクション数
      を普通のゴールと同様に数える。
Configuration for builtin predicate suspension.
>> COUNT_DCODE_REDUCTION: If this switch's value is 'YES', PDSS count
      reductions by D-code goals which are generated by builtin predicate
      suspension.
*****************************************************************************/
#define COUNT_DCODE_REDUCTION NO

/*****************************************************************************
インストラクショントレースに関する設定。
>> INSTRUCTION_TRACE: これが YES の場合には、KL1-B のインストラクション
      単位のトレースが可能となる。
>> INSTRUCTION_COUNT: これが YES の場合には、KL1-B の命令種類ごとの実行
      回数の集計を行う。
>> INSTRUCTION_BRANCH_COUNT: これと INSTRUCTION_COUNT が両方とも YES の
      場合には、KL1-B の命令種類ごとに分岐回数の集計を行う。
Configuration for instruction trace.
>> INSTRUCTION_TRACE: If this switch's value is 'YES', PDSS can trace
      KL1-B instructions.
>> INSTRUCTION_COUNT: If this switch's value is 'YES', PDSS counts the
      number of times each KL1-B instuction is executed.
>> INSTRUCTION_BRANCH_COUNT: If both this switch's value and
      INSTRUCTION_COUNT's value are 'YES', PDSS counts the number of
      branches in each KL1-B instuction.
*****************************************************************************/
#define INSTRUCTION_TRACE        NO
#define INSTRUCTION_COUNT        NO
#define INSTRUCTION_BRANCH_COUNT NO
```

I – 4 PDSS Installation Manual

(in Japanese)

1990年9月14日
ICOT 第1研究室

PDSS (V2.52) インストールの手引

PDSS とは "PIMOS Development Support System" の略であり、その名のとおり PIMOS の開発環境として作成された (マルチプロセッサでない) 汎用計算機上で動作する KL1 システムです。 PDSS はコンパイラ、バイトコードインタープリタ、および、MicroPIMOS (コマンドインタープリタと入出力ライブラリ等) から構成されている。また、2つの Emacs Lisp ライブラリ、KL1-mode と PDSS-mode も提供している。 KL1-mode は KL1 のソースコードを編集するためのメジャーモードであり、PDSS-mode は Emacs の複数のバッファを通して PDSS を使用するためのメジャーモードです。ICOT では、PDSS を SUN-3, SUN-4 および Sequent Symmetry 上で使用しており、BSD 4.2, 4.3 UNIX を使用したシステムには簡単にインストールできるようになっている。

1 PDSS のテープの内容

PDSS のテープには以下のようなディレクトリとファイルが含まれている。これらの全てを読み込むためには 7MB、メイクするためには約 16MB のファイルスペースが必要です。もし、これだけの領域が用意できない場合は次章の説明を参考にして下さい。

	名前	内容	サイズ
	Makefile	Makefile	2
	pdsscmp	KL1 ソースをコンパイルするためのシェルスクリプト	6
	emulator	KL1-B バイトコードインタープリタ = KL1 抽象機械	946
	runtime	KL1 で記述したランタイムライブラリ	139
	mpimos	Micro PIMOS = コマンドインタープリタと入出力ライブラリ	1554
	compiler	KL1 で記述した KL1 コンパイラ	1758
	macro	KL1 マクロの定義ファイル	12
+	debug_utl	KL1 ソースをテストするツールプログラム	280
+	pimos_utl	PIMOS グループが作成したライブラリプログラム	184
+	compiler_pl	SICStus Prolog で記述した KL1 コンパイラ	359
+	emacs	Emacs Lisp ライブラリ (KL1-mode と PDSS-mode)	87
+	test	KL1 テストプログラム	549
+	sample	KL1 サンプルプログラム	124
	† この "+" 印はオプションのモジュールを意味する。		KB

2 テープからのリストア方法

PDSS のテープは tar フォーマットで書き込まれているので、リストアするためには以下のようなコマンドを実行すればよい。この結果、現在のカレントディレクトリに "pdss" という名前のディレクトリが作られ、その中に全てが読み込まれる。

```
% tar xf /dev/rts0 pdss
          ----------  これはテープドライブのデバイス名。この名前は機
                      種とメディアに依存するので適当なものを指定する
                      必要がある。もし分からない場合には、インスト－
                      ル先のマシンのマニュアルを調べて下さい。
```

なお、ディスク・スペースが少ない場合には、以下のようにすると、KL1 で書かれたモジュールのソースコードとアセンブラコードを読み込むのを止め、オブジェクトコードだけを読むので使用領域を減らすことができます。このようにしても、対象が 680X0 や 80386 のような CISC タイプの CPU を持つマシンの場合にはインストールが可能です。"emulator/config.h" の KLB_PACKED_CODE に関するコメントを参照して下さい。

```
% tar xf /dev/rts0 FILES.SMALL
% tar xf /dev/rts0 `cat FILES.SMALL`
```

テープからのリストアが完了したら、必ず以下の手続きを行なって下さい。もし SICStus Prolog を持っていない場合には、"compiler_pl" ディレクトリを rename するか remove して下さい。もし GNU Emacs を持っていない場合には、"emacs" ディレクトリを rename するか remove して下さい。その他、オプションモジュールの中で必要のないものがあれば、それを rename するか remove して下さい。

3 Makefile と configuration-file の編集

メイクを行なう前に、以下のようなマシン依存やサイト依存の部分を含んだファイルを編集する必要がある。

- Makefile
  変数 BINDIR の値を自分のサイトに適したものに変更する。これは、"make install" を行なった時のターゲットとして使われる。
- emulator/Makefile
  変数 BINDIR の値を自分のサイトに適したものに変更する。これは、"make install" を行なった時のターゲットとして使われる。
  変数 CFLAGS の値を必要であれば適当なものに変更する。
- emulator/config.h
  ファイル内のコメントに従って各定義を変更する必要がある。もしインストール先のマシンが CISC タイプの CPU(680X0, 80386) を使っている場合には、パス名の定義以外は変えなくても大丈夫です。しかし、RISC タイプの CPU を使っている場合には、KLB_PACKER_CODE の定義を必ず NO に変更する必要があります。
- compiler_pl/libdir.pl (もし Prolog で書かれたクロスコンパイラを使うなら)
  パス名を自分のサイトに適したものに変更する。
- emacs/Makefile (もし Emacs Lisp ライブラリを使うなら)
  変数 LIBDIR の値を自分のサイトに適したものに変更する。これは、"make install" を行なった

時のターゲットとして使われる。
変数 EMACS の値を自分のサイトに適したものに変更する。これは、Emacs Lisp ライブラリをコンパイルするために使われる。

- emacs/pdss-init.el (もし Emacs Lisp ライブラリを使うなら)
  変数 pdss-directory-name の値を自分のサイトに適したものに変更する。これは PDSS のトップレベルのディレクトリのパス名にする。
- pdsscmp
  最初の 5 行で定義しているパス名を変更する。

4 Make

メイクは、もし KLB_PACKED_CODE と KLB_4BYTE_RED_ADDR の値を変更していない場合には、以下のコマンドを PDSS のトップレベルのディレクトリで実行して下さい。

```
% make quick
```

もしこれらの値を変更した場合には、以下のコマンドを実行して下さい。この場合には、全ての KL1 アセンブラコードを再アセンブルします。

```
% make reassm
```

コンパイルが成功した場合には、以下のコマンドにより PDSS を公共ディレクトリにコピーし、一般のユーザーから使えるようにします。

```
% make install
```

最後に (もし必要であれば) 作業用に作ったファイルを削除します。

```
% make clean
```

付録 -I に ICOT でメイクを行った時のログがあるので、それを参考にして下さい。その時の環境は SUN SparcStation-1 (SunOS 4.0.3c) です。

5 Emacs ライブラリ

PDSS は 2 つの Emacs Lisp ライブラリ、KL1-mode と PDSS-mode を提供している。KL1 mode は KL1 のソースコードを編集するためのメジャーモードであり、PDSS-mode は Emacs の複数のバッファを通して PDSS を使用するためのメジャーモードです。これらのライブラリを使いたい場合には、ユーザーのホームディレクトリにある ".emacs" ファイルに以下のコマンドを挿入して下さい。また、全ユーザーが使えるようにする場合には同じものを "site-init.el" ファイルに挿入して下さい。

```
(load "pdss-init")
```

なお、ICOT で使用している GNU-Emacs は "GNU Emacs 18.55.31" をベースにした "Nemacs version 3.2.3" です。他の版を使っている場合には、変更が必要かも知れません。

6 BSD UNIX 以外に移植する場合

PDSS は BSD UNIX 上で動作することを目的に書かれているので、System-V 等の UNIX に移植する場合には多くの変更を必要とします。少なくとも以下の部分に変更が必要になります。

- emulator/pdss.c
  signal(), sigblock(), sigsetmask() および kill() がシグナルハンドラで使われている。
  getrusage() が CPU Time を調べる為に使われている。
- emulator/io.c
  signal(), sigblock(), sigmask() および sigsetmask() がシグナルハンドラで使われている。
  ioctl() および fcntl() が I/O デバイスの制御で使われている。
- emulator/blt_system.c
  getrusage() が CPU Time を調べる為に使われている。
- emulator/blt_iodev.c
  opendir(), clausedir() および readdir() がディレクトリのリスティングのために使われている。
  unlink() がファイルの削除のために使われている。
  re_comp() および re_exec() が正規表現の処理の為に使われている。
- emulator/timer.c
  signal(), sigblock(), sigmask() および sigsetmask() がシグナルハンドラで使われている。
  alarm() がタイマー割り込みのために使われている。
  time() が現在時刻を取り出すために使われている。

付録 -I　ICOT で行なったメイクのログ

以下は ICOT の SUN SparcStation-1 (SunOS 4.0.3c) 上でメイクを行った時のログです。参考にして下さい。

```
%
% ls                                        ## PDSS のディレクトリにいる。
LOG             debug_utl/      emulator/       pimos_utl/
Makefile        doc.e/          macro/          runtime/
compiler/       doc.j/          mpimos/         sample/
compiler_pl/    emacs/          pdsscmp*        test/
%
% mv emacs emacs.notuse                    ## 今回は Emacs を使わないので、
%                                          ## このディレクトリを rename する。
% mv compiler_pl compiler_pl.notuse        ## SICStus Prolog も使わないので、
%                                          ## このディレクトリを rename する。
%
%                                          ## ここで幾つかのファイルを edit する。
%
% make reassm                              ## KLB_PACKED_CODE の定義を変えたので
cd emulator     ; make all                 ## "make reassm" を行なった。
touch pdss.h
cc -O  -sun4 -c  pdss.c
cc -O  -sun4 -c  option.c
cc -O  -sun4 -c  emulate.c
cc -O  -sun4 -c  blt_basic.c
cc -O  -sun4 -c  blt_float.c
cc -O  -sun4 -c  blt_shoen.c
cc -O  -sun4 -c  blt_iodev.c
cc -O  -sun4 -c  blt_code.c
cc -O  -sun4 -c  blt_system.c
```

```
cc -O  -sun4 -c  deref.c
cc -O  -sun4 -c  passive.c
cc -O  -sun4 -c  unify.c
cc -O  -sun4 -c  goal.c
cc -O  -sun4 -c  shoen.c
cc -O  -sun4 -c  exception.c
cc -O  -sun4 -c  memory.c
cc -O  -sun4 -c  atom.c
cc -O  -sun4 -c  module.c
cc -O  -sun4 -c  dcode.c
cc -O  -sun4 -c  float.c
cc -O  -sun4 -c  string.c
cc -O  -sun4 -c  gc.c
cc -O  -sun4 -c  gc_ctrl.c
cc -O  -sun4 -c  gc_cell.c
cc -O  -sun4 -c  gc_code.c
cc -O  -sun4 -c  gc_dead.c
cc -O  -sun4 -c  mrbgc.c
cc -O  -sun4 -c  instr.c
cc -O  -sun4 -c  invassm.c
cc -O  -sun4 -c  assemble.c
cc -O  -sun4 -c  saveload.c
Routine  _write_predicate  too big:                    ## スワップ領域不足が発生。
        use a lower level of optimization or           ## (ICOT の SUN はスワップ領域
        increase stack limit and / or                  ##  を少ししか用意してない)
        size of swap space

compiler(iropt) error:   alloca: out of memory
*** Error code 1
make: Fatal error: Command failed for target 'saveload.o'
Current working directory /usr2/pdss/pdss2.5/emulator
*** Error code 1
make: Fatal error: Command failed for target 'reassm'
% (cd emulator; cc  -sun4 -c  saveload.c)              ## 手動でリトライを行なう。
/usr2/pdss/pdss2.5/emulator
% make reassm                                         ## "make reassm" を継続する。
cd emulator     ; make all
cc -O  -sun4 -c  native.c
cc -O  -sun4 -c  tracer.c
cc -O  -sun4 -c  print.c
cc -O  -sun4 -c  io.c
cc -O  -sun4 -c  iosub.c
cc -O  -sun4 -c  timer.c
cc -O  -sun4 -c  path.c
cc -O  -sun4 -c  ctype.c
cc -O -o pdss.x -D"MAKEDATE=\"`date`\"" version.c pdss.o option.o emulate.o blt
_basic.o blt_float.o blt_shoen.o blt_iodev.o blt_code.o blt_system.o deref.o pa
ssive.o unify.o goal.o shoen.o exception.o memory.o atom.o module.o dcode.o flo
at.o string.o gc.o gc_ctrl.o gc_cell.o gc_code.o gc_dead.o mrbgc.o instr.o inva
ssm.o assemble.o saveload.o native.o tracer.o print.o io.o iosub.o timer.o path
.o ctype.o -lm
mv -f pdss.x pdss
rm -f version.o
cc -O  -sun4 -c  pdssasm.c
cc -O -o pdssasm.x -D"MAKEDATE=\"`date`\"" version.c pdssasm.o memory.o atom.o
module.o instr.o assemble.o saveload.o float.o ctype.o -lm
mv -f pdssasm.x pdssasm
rm -f version.o
cc -O  -sun4 -c  pdssmerge.c
cc -o pdssmerge.x pdssmerge.o
mv -f pdssmerge.x pdssmerge
sh -c "test ! -d compiler_pl || (cd compiler_pl ; make all)"
```

```
cd runtime      ; make reassm
for f in `echo coddev.sav windev.sav fildev.sav timdev.sav shoen.sav | sed 's/\
.sav//g'`; do ../emulator/pdssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file coddev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file windev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file fildev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file timdev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file shoen... OK
cd mpimos       ; make reassm
for f in `echo boot.sav op.sav optbl.sav  parser.sav unparser.sav shell.sav mac
ro_ex.sav  task_monitor.sav utl.sav code_man.sav module_pool.sav  window.sav wi
ndow_man.sav window_drv.sav  file.sav file_man.sav file_drv.sav  dir.sav dir_dr
v.sav timer_man.sav  dummy_window.sav dummy_file.sav dummy_dir.sav  builtin.sav
 cmd_basic.sav cmd_code.sav cmd_debug.sav  cmd_dir.sav cmd_env.sav cmd_utl.sav
pool_mli.sav  mpimos_iodev.sav mpimos_windev.sav  mpimos_fildev.sav mpimos_timd
ev.sav | sed 's/\.sav//g'`; do ../emulator/pdssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file boot... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file op... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file optbl... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file parser... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file unparser... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file shell... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file macro_ex... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file task_monitor... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file utl... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file code_man... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file module_pool... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file window... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file window_man... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file window_drv... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file file... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file file_man... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file file_drv... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dir... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dir_drv... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file timer_man... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
```

```
Make save file dummy_window... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dummy_file... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file dummy_dir... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file builtin... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_basic... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_code... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_debug... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_dir... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_env... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file cmd_utl... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file pool_mli... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_iodev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_windev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_fildev... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_timdev... OK
cd compiler      ; make reassm
for f in `echo blt.sav com.sav comp.sav mrb.sav norm.sav outp.sav reader.sav  r
eg.sav macarg.sav macro.sav mactbl.sav struct.sav | sed 's/\.sav//g'`; do ../em
ulator/pdssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file blt... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file com... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file comp... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mrb... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file norm... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file outp... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file reader... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file reg... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file macarg... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file macro... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mactbl... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file struct... OK
sh -c "test ! -d debug_utl    || (cd debug_utl    ; make reassm)"
for f in `echo mpimos_xref.sav mpimos_xref_table.sav  mpimos_pretty_printer.sav
 mpimos_varchk.sav | sed 's/\.sav//g'`; do ../emulator/pdssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_xref... OK
```

```
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_xref_table... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_pretty_printer... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file mpimos_varchk... OK
sh -c "test ! -d pimos_utl   || (cd pimos_utl   ; make reassm)"
for f in `echo comparator.sav hasher.sav keyed_bag.sav keyed_sorted_bag.sav  po
ol.sav queue.sav sorted_bag.sav stack.sav | sed 's/\.sav//g'`; do ../emulator/p
dssasm $f; done
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file comparator... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file hasher... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file keyed_bag... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file keyed_sorted_bag... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file pool... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file queue... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file sorted_bag... OK
**** PDSS-ASM V2.52.06 (Mon Jul 16 17:53:54 JST 1990) ****
Make save file stack... OK
sh -c "test ! -d emacs       || (cd emacs       ; make all)"
%
%                                        ## ここまででコンパイルは完了。
%                                        ## PDSS が動くかテストしてみる。
% cd emulator
/usr2/pdss/pdss2.5/emulator
% pdss
***** PDSS-KL1 V2.52.06 (Mon Jul 16 17:53:41 JST 1990) *****
.................................................

*****************************************
**** Micro PIMOS (version 2.5) ************
*****************************************
World is /usr2/pdss/pdss2.5/emulator
Total heap size is 200000 words
Total code size is 274688 bytes
*****************************************

| ?- compile(sample).
"sample" : Compiling ... primes/0,primes/2,gen/3,sift/2,filter/3,display/1,END.

[primes/0]
successdone
Loading ... done
Module name : sample
Saving ... done

yes.
| ?- stat(sample:primes).
[2,3,5,7,11,13,17,19,23,29,31,37,...]
Total of 5880 reductions in 290 msec.

yes.
| ?- halt.

%
```

付録 -II　“emulator/config.h” ファイルのサンプル

以下は、ICOT で使っている ”emulator/config.h” ファイルです。

```
/*************************************************************************
PDSS の為のディレクトリ / ファイル環境の指定。
>> PDSS_LIBDIR: PDSS が立ち上がる時に、ランタイムライブラリを探しにいくディ
      レクトリのパス名。通常は PDSS のトップレベルのディレクトリと同じ。
>> PDSS_SUPDIR: 立ち上げ用ファイル STARTUP が置いてあるディレクトリのパス名。
      STARTUP ファイルには、立ち上げ時に読み込むべき KL1 モジュールのファ
      イル名と、最初のゴール名が記述してある。
>> PDSS_OBJECT_FILE: エミュレータ本体のパス名。Native Code Module を動的に
      リンクする時に使われる。（しかし、現在 Native Code Generator は
      サポートされていない。）
Configuration of the directory/file environment for PDSS.
>> PDSS_LIBDIR: A path name of the directory which PDSS search for runtime
      libraries at initial startup time. Normally, this is equal to the
      PDSS toplevel directory.
>> PDSS_SUPDIR: A path name of the directory which contains STARTUP file.
      STARTUP file describes path names (relative path from PDSS_LIBDIR)
      of the runtime library which are read at initial startup time, and
      initial goal name.
>> PDSS_OBJECT_FILE: A path name of the PDSS byte-code interpreter executable
      object. It is uesd by dynamic native code module linker.
      (But, native code generator is not supported yet.)
*************************************************************************/
#define PDSS_LIBDIR      "/usr/local3/pdss/release"
#define PDSS_SUPDIR      "/usr/local3/pdss/release/emulator"
#define PDSS_OBJECT_FILE "/usr/local3/pdss/release/emulator/pdss"

/*************************************************************************
PDSS のコードの形式に関する指定。
>> KLB_PACKED_CODE: これが YES の場合には、KL1-B のコードは詰め込まれた
      状態で保持される。この場合、整数等の複数バイトのデータもワードの境界
      に合わせずに詰めて書き込まれる。） マシンの CPU が RISC タイプの場合
      には、これを NO にしておく必要がある。
>> KLB_4BYTE_REL_ADDR: これが YES の場合には、KL1-B のコードにおける相対
      アドレスは 4byte で表現される。そうでなければ 2byte で表現される。
Configuration for instruction trace.
>> KLB_PACKED_CODE: If this switch's value is 'YES', KL1-B instruction
      code (byte-code) is filled in continuously. In this case, multiple
      byte data (e.g. integer) is written not in word alignment. If your
      machine uses RISC type CPU, you must define this switch 'NO'.
>> KLB_4BYTE_REL_ADDR: If this switch's value is 'YES', PDSS use 4 byte
      relative address instead of 2 byte one.
*************************************************************************/
#define KLB_PACKED_CODE    YES
#define KLB_4BYTE_REL_ADDR NO

/*************************************************************************
PDSS が使用するメモリのデフォルトの大きさの指定。
>> HEAP_SIZE: ヒープ領域の大きさ。単位はタグ付きワード (8 バイト)。
      PDSS ではコピー方式の GC を行うので、ここでは全ヒープ領域の半分の
      大きさを指定する。
>> CODE_SIZE: コード領域の大きさ。単位はバイト。ここで指定するのはコード全体
      の大きさである。PDSS は立ち上げ時に、ランタイムライブラリを読み込む
      ために先頭から使用し、その残りを半分に分けてユーザー用として使用する。
Configuration for the default memory size.
>> HEAP_SIZE: Size of head area. This specifies the number of tagged-words
      (8 byte) in the HALF of heap area, since PDSS use copying GC.
>> CODE_AREA: Size of code area. This specifies the number of bytes in the
      whole code area. PDSS loads runtime libraries into top of the code
      area at initial startup time, then PDSS divide code area into 2
```

```
      parts and use divided one for user codes.
*****************************************************************************/
#define HEAP_SIZE  200000
#define CODE_SIZE 1000000

/*****************************************************************************
MRB-GC に関する設定。
>> MRBGC_SWITCH: これが YES の場合に MRB による実時間 GC を行う。
>> MRBGC_COLLECT_VALUE_DEPTH: Collect Value 命令で回収する構造体の深さ。
>> MRBGC_DEREF_TYPE: デレフ時の回収のタイプ。
       0 --> 参照パスで一旦 MRB=ON になった先の回収を行わない。
       1 --> 参照パスで一旦 MRB=ON になった先の回収も行う。
>> MRBGC_FREE_LIST_TYPE: メモリ再利用の為のフリーリストの種類。
       0 --> 1W,2W,3W, ... ,7W,8W,16W,32W の 10 種類を用意する。
       1 --> 1W,2W,4W,8W,16W,32W,64W,128W の 8 種類を用意する。
>> MRBGC_STATISTICS: MRB-GC に関する統計情報の集計。
       0 --> 集計は行わない。
       1 --> フリーリストの使用状況の集計を行う。
       2 --> 1 + 命令ごとの回収状況の集計も行う。
Configuration for MRB-GC.
>> MRBGC_SWITCH: If this switch's value is 'YES', PDSS collect garbage
      incrementally by the MRB scheme in real time.
>> MRBGC_COLLECT_VALUE_DEPTH: The maximum depth of the nested data structure
      which collected by collect_value instruction.
>> MRBGC_DEREF_TYPE: The switch of garbage collection in dereference routine.
      0 -> If MRB-ON pointer is found in reference path, PDSS don't try
           to collect garbage in the forward path of this pointer.
      1 -> In spite of MRB-ON pointer is found in reference path, PDSS
           try to collect garbage in the forward path of this pointer.
>> MRBGC_FREE_LIST_TYPE: The variation of free lists.
      0 -> Use 10 kinds (1W,2W,3W,...,7W,8W,16W and 32W) of free lists.
      1 -> Use 8 kinds (1W,2W,4W,8W,16W,32W,64W and 128W) of free lists.
>> MRBGC_STATISTICS: The switch of statistics of MRB-GC.
      0 -> PDSS don't collect statistics information of MRB-GC.
      1 -> PDSS collect statistics information about free list only.
      2 -> PDSS collect statistics information about free list and
           incremental garbage collecting instructions.
*****************************************************************************/
#define MRBGC_SWITCH               YES
#define MRBGC_COLLECT_VALUE_DEPTH 100
#define MRBGC_DEREF_TYPE            1
#define MRBGC_FREE_LIST_TYPE        0
#define MRBGC_STATISTICS            0

/*****************************************************************************
MRB を利用したデッドロック (無限待ち) の eager detection に関する設定。
>> EAGER_DEADLOCK_DETECTION:
      これが YES の場合には eager detection を行なう。
Configuration for eager detection of deadlock (perpetual suspension).
>> EAGER_DEADLOCK_DETECTION:
      If this switch's value is 'YES', PDSS deetect deadlock eagerly.
*****************************************************************************/
#define EAGER_DEADLOCK_DETECTION YES

/*****************************************************************************
Ready Goal Pool に関する設定。
>> TWO_WAY_READY_GOAL_POOL: これが YES の場合には、先頭と最後の２つの入口を
      持った Ready Goal Pool を使用する。即ち、スタックとキューの両方に
      使えるので、この時には、ゴールのスケジューリングを Breadth First
      に変更する事もできる。NO の場合には、スタック方式のみになる。
Configuration for ready goal pool.
>> TWO_WAY_READY_GOAL_POOL: If this switch's value is 'YES', PDSS use ready
```

```
      goal pool with two entry point: top and last. In this case, ready
      goal pool can be used as stack or queue, and user can select depth
      first or breadth first scheduling. If it's value is 'NO', reday goal
      pool is stack and scheduling is depth first only.
*****************************************************************************/
#define TWO_WAY_READY_GOAL_POOL YES

/*****************************************************************************
コピーGC 時に異常事態になった時の処理に関する設定。
>> GC_SWAP_GOALS_WHEN_PANIC: これが YES の場合には、コピー GC でヒープ領域
      があまり回収できなかった時に、ゴール・スタックの highest プライオリティ
      の中のゴールの順番を入れ替える。これにより、実行順序が変わるので
      ヒープが回収できるようになる可能性がある。
>> GC_ABORT_TASK_WHEN_PANIC: これが YES の場合には、コピー GC でヒープ領域
      がほとんど回収できなかった(GC直後なのにGC要求フラグが立っている)時に、
      ユーザーのタスクをアボートする。これにより、PDSS全体が異常終了する
      のを防ぐ。
Configuration for emergency measures of copying GC.
>> GC_SWAP_GOALS_WHEN_PANIC: If this switch's value is 'YES', PDSS swap
      goals in ready goal pool when copying GC get only small gain.
      By this goal swapping, the sequence of goal execution is changed and
      some data on heap may be consumed and collected.
>> GC_ABORT_TASK_WHEN_PANIC: If this switch's value is 'YES', PDSS abort
      user task when copying GC get little gain. By this abortion, many
      heap uesd by user task are released and whole PDSS may not abort.
*****************************************************************************/
#define GC_SWAP_GOALS_WHEN_PANIC YES
#define GC_ABORT_TASK_WHEN_PANIC YES

/*****************************************************************************
アトムテーブルに関する設定。
>> MAX_ATOMS: PDSS で使用できるアトムの数。
Configuration for atom table.
>> MAX_ATOMS: Maximum number of atoms.
*****************************************************************************/
#define MAX_ATOMS 8000

/*****************************************************************************
整数演算に関する設定。
>> IGNORE_INTEGER_OVERFLOW: この値が YES ならば、組込述語における整数演算
      のオーバーフローを無視する。
>> IGNORE_INTEGER_OVERFLOW: If this value is 'YES', PDSS ignores integer
      overflow in builtin predicates.
*****************************************************************************/
#define IGNORE_INTEGER_OVERFLOW NO

/*****************************************************************************
パッシブユニフィケーションに関する設定。
>> PASSIVE_UNIFY_DEPTH: この値は Passive Unification (wait_value, diff)
      で構造体同士を比較する深さを指定する。この深さまで調べても成功が確定
      しない場合には失敗と見做す。
Configuration for passive unification.
>> PASSIVE_UNIFY_DEPTH: This is the maximum depth of the nested structures
      which compare in passive unification (wait_value, diff). If PDSS
      can't confirm the result of unification while PDSS compare until
      this depth, PDSS regards the unification as failure.
*****************************************************************************/
#define PASSIVE_UNIFY_DEPTH 100

/*****************************************************************************
コンソール出力に関する設定。
>> PRINT_LENGTH, PRINT_DEPTH: コンソールへのターム出力における、長さと深さ
```

```
      の制限の初期値。これを越える部分は "..." で出力する。
Configuration for console outputs.
>> PRINT_LENGTH, PRINT_DEPTH: These are initial maximum length and depth
      of structure which can be displayed in the console. PDSS print the
      part over this limitation link "...".
*****************************************************************************/
#define PRINT_LENGTH 12
#define PRINT_DEPTH  6

/*****************************************************************************
パーザーに関する設定。
>> MAX_PRINTABLE_STRING_ATOM_LENGTH: ウィンドウ / ファイル等に出力可能な文字
      列 / アトムの長さの最大長。文字列 --> アトム の変換もこの制限を受ける。
Configuration for parser.
>> MAX_PRINTABLE_STRING_ATOM_LENGTH: The maximum length of atom/string
      which can be displayed into window/file.  This limitation is also
      used for atom-string conversion.
*****************************************************************************/
#define MAX_PRINTABLE_STRING_ATOM_LENGTH 10000

/*****************************************************************************
組込述語のサスペンドに関する設定。
>> COUNT_DCODE_REDUCTION: この値が 'YES' ならば、組込述語がサスペンドし、
      ゴール (D-Code) がエンキューされた時に、そのゴールの分のリダクション数
      を普通のゴールと同様に数える。
Configuration for builtin predicate suspension.
>> COUNT_DCODE_REDUCTION: If this switch's value is 'YES', PDSS count
      reductions by D-code goals which are generated by builtin predicate
      suspension.
*****************************************************************************/
#define COUNT_DCODE_REDUCTION NO

/*****************************************************************************
インストラクショントレースに関する設定。
>> INSTRUCTION_TRACE: これが YES の場合には、KL1-B のインストラクション
      単位のトレースが可能となる。
>> INSTRUCTION_COUNT: これが YES の場合には、KL1-B の命令種類ごとの実行
      回数の集計を行う。
>> INSTRUCTION_BRANCH_COUNT: これと INSTRUCTION_COUNT が両方とも YES の
      場合には、KL1-B の命令種類ごとに分岐回数の集計を行う。
Configuration for instruction trace.
>> INSTRUCTION_TRACE: If this switch's value is 'YES', PDSS can trace
      KL1-B instructions.
>> INSTRUCTION_COUNT: If this switch's value is 'YES', PDSS counts the
      number of times each KL1-B instuction is executed.
>> INSTRUCTION_BRANCH_COUNT: If both this switch's value and
      INSTRUCTION_COUNT's value are 'YES', PDSS counts the number of
      branches in each KL1-B instuction.
*****************************************************************************/
#define INSTRUCTION_TRACE        NO
#define INSTRUCTION_COUNT        NO
#define INSTRUCTION_BRANCH_COUNT NO
```