

TM-0946

PDSS Manual (Version 2.52.e)

by
K. Hirano

August, 1990

© 1990, ICOT

ICOT

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

(03)3456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

PDSS Manual

(Version 2.52e)

Dec 26 1989

Institute for New Generation Computer Technology

Fourth Laboratory

Copyright (C) 1988,89 by ICOT

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	1
2	KL1 Language Specification	2
2.1	Outline	2
2.2	Sho-en	2
2.2.1	Sho-en generation	3
2.2.2	Control stream	4
2.2.3	Report stream	4
2.3	Priority	6
2.4	Syntax	6
2.4.1	Module definition	8
2.4.2	Clause ordering	8
2.5	Data types	8
2.6	Built-in predicates	9
2.6.1	Type checking	9
2.6.2	Diff	10
2.6.3	Arithmetic comparison (Integer)	10
2.6.4	Arithmetic operations (Integer)	11
2.6.5	Arithmetic comparison (Floating point)	13
2.6.6	Arithmetic operations (Floating point)	14
2.6.7	Conversion (Integer - Floating Point)	17
2.6.8	Vector predicates	17
2.6.9	String predicates	18
2.6.10	Atom predicates	19
2.6.11	Code predicates	19
2.6.12	Stream support	20
2.6.13	Second order function	20
2.6.14	Special I/O functions	20
2.6.15	Other predicates	20
2.7	Macros	21
2.7.1	Constant description macros	21
2.7.2	Unification macros	22
2.7.3	Arithmetic comparison macros	22
2.7.4	Arithmetic operation macros	22
2.7.5	Macros for implicit argument passing	24
2.7.6	Conditional branch macros	27
2.7.7	Macro library	28
3	Micro PIMOS	29
3.1	Command interpreter	29
3.1.1	Command input format	29
3.1.2	Commands	30
3.2	I/O functions	34
3.2.1	Command stream attachment	34
3.2.2	Command list	35
3.3	Directory management	38
3.3.1	Acquisition of command stream	38
3.3.2	Commands	38
3.4	Device Stream for I/O	39
3.4.1	Securing device stream	39
3.4.2	Command	39
3.5	Code management	39
3.6	Displaying exception information	40
4	PDSS Optional Parameters	41

4.1	Usage under GNU-Emacs	41
4.2	PDSS on stand-alone	41
4.3	Optional parameters	41
5	Tracer	43
5.1	Principle of operation	43
5.2	How to read the display	43
5.3	Commands	44
6	Dead-lock Detection	47
	Appendix	50
	Appendix-1 I/O devices	50
	Appendix-2 Code device	55
	Appendix-3 PIMOS common utilities	56
	Appendix-4 Reserved module names	60
	Appendix-5 Reserved operator names	61
	Appendix-6 List of built-in predicates	62
	Appendix-7 Exception codes	65
	Appendix-8 Reserved Sho-en tags	66
	Appendix-9 GNU-Emacs library	67
	Appendix-10 Using command procedures for compiling	69
	Appendix-11 Sample program	70
	Appendix-12 What to do if a bug is found out...	71
	Index	73

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.

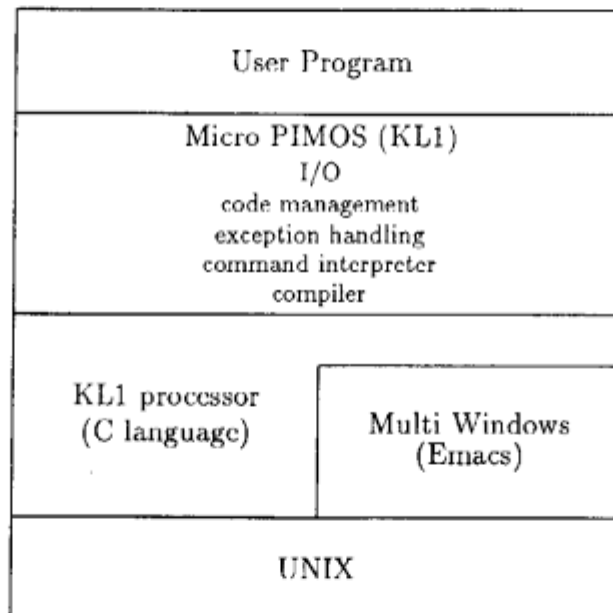


Figure 1: PDSS configuration

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 management 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.

¹UNIX 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 occurring 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

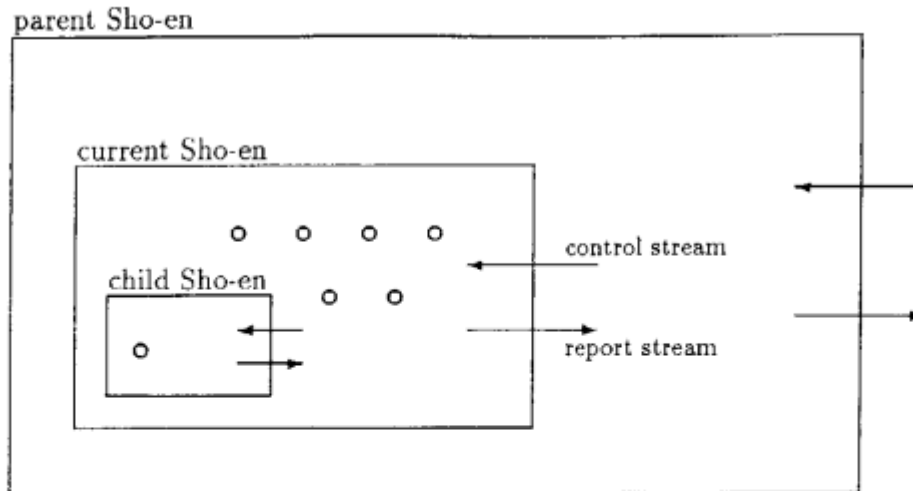


Figure 2: An instant picture of some Sho-ens

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 consumption 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 specifies 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)

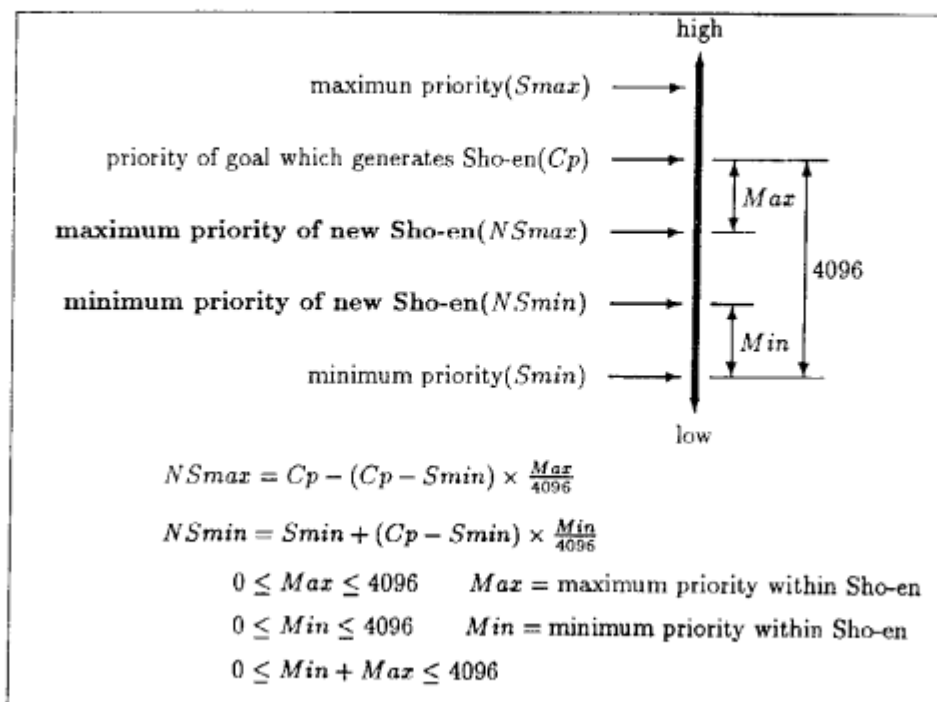


Figure 3: Calculation of Sho-en Priority

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_resource 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 [], no goal is executed any more.

exception(ExcpCode, Info, NewCode, NewArgv)

Exception occurred 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 ~ 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 \square (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 according 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

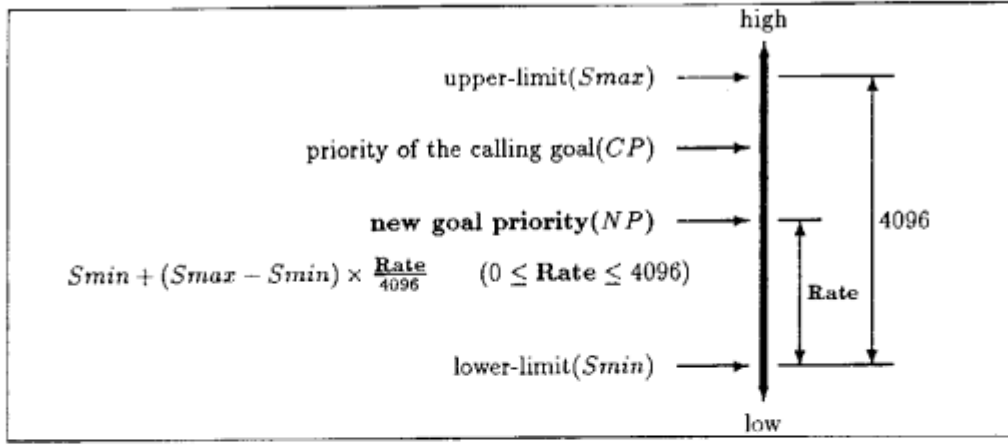


Figure 4: computation of the goal priority by rate specification in the belonging Sho-en

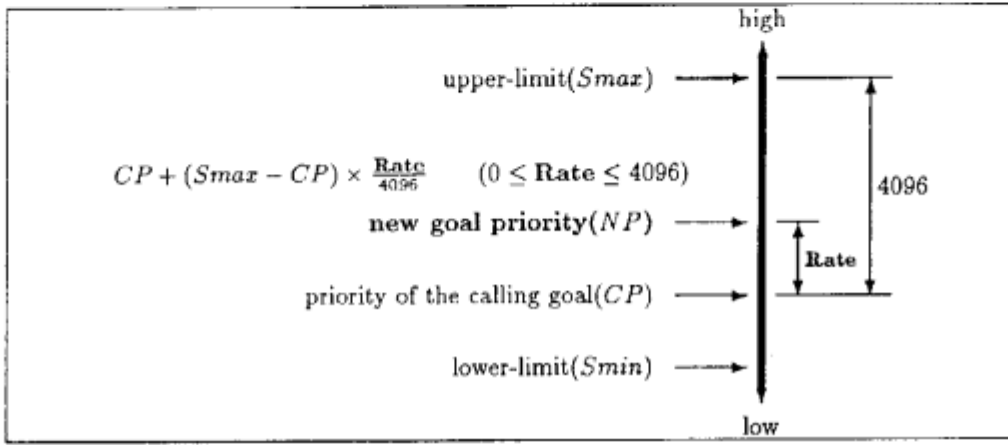


Figure 5: computation of the goal priority by relative self specification in the belonging Sho-en(plus)

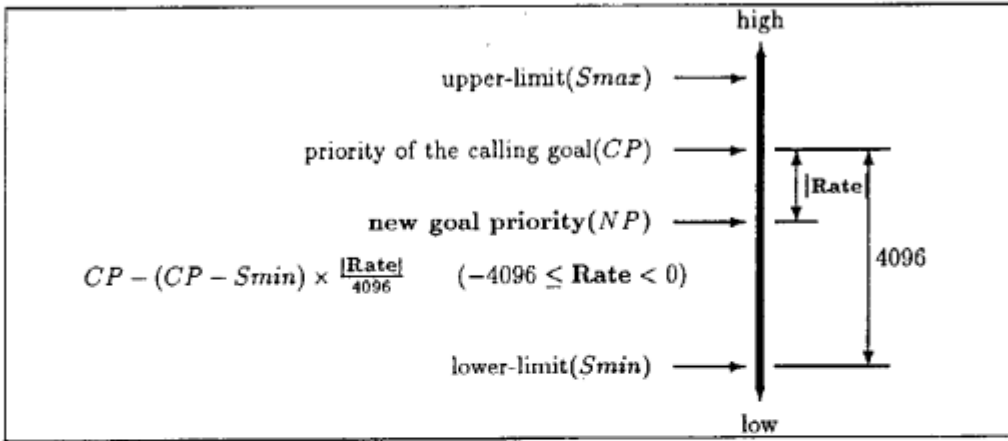


Figure 6: computation of the goal priority by relative self specification in the belonging Sho-en(minus)

- 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 characters and numerals, and begins with a capital character 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 characters which begin with a small letter, or only of signs, or of characters 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 point. 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 'l'.

- 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 characters quoted with double quotation 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 characters 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 :

$$\begin{array}{ccc} \text{vector}(X, \text{~Size}) :: \underline{G} & & \\ \uparrow & \uparrow & \\ \text{Call format} & \text{Valid location for occurrence} & \end{array}$$

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 occurring 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 chapter 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 **Integer2** 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 **Integer2** 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.

$Y := X + 1 \quad \Leftrightarrow \quad \text{increment}(X, Y).$

« ! » 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.

$Y := X - 1 \quad \Leftrightarrow \quad \text{decrement}(X, Y).$

« ! » 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.

$Y := \text{abs}(X) \quad \Leftrightarrow \quad \text{abs}(X, Y).$

« ! » 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.

$Z := \min(X, Y) \quad \Leftrightarrow \quad \min(X, Y, Z).$

« ! » 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.

$Z := \max(X, Y) \quad \Leftrightarrow \quad \max(X, Y, Z).$

« ! » 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.

$Z := X \wedge Y \quad \Leftrightarrow \quad \text{and}(X, Y, Z).$

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.

$Z := X \vee Y \quad \Leftrightarrow \quad \text{or}(X, Y, Z).$

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.

$Z := X \text{ xor } Y \quad \Leftrightarrow \quad \text{exclusive_or}(X, Y, Z).$

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 $\$:= \min(X,Y) \quad \Leftrightarrow \quad \text{floating_point_min}(X,Y,Z).$

« ! » 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 $\$:= \max(X,Y) \quad \Leftrightarrow \quad \text{floating_point_max}(X,Y,Z).$

« ! » 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 $\$:= \text{floor}(X) \quad \Leftrightarrow \quad \text{floating_point_floor}(X,Y).$

« ! » 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 $\$:= \text{sqrt}(X) \quad \Leftrightarrow \quad \text{floating_point_sqrt}(X,Y).$

« ! » 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 $\$:= \ln(X) \quad \Leftrightarrow \quad \text{floating_point_ln}(X,Y).$

« ! » 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 $\$:= \log(X) \quad \Leftrightarrow \quad \text{floating_point_log}(X,Y).$

« ! » 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 $\$:= \exp(X) \quad \Leftrightarrow \quad \text{floating_point_exp}(X,Y).$

« ! » 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.

$Y \$:= X * Y \iff \text{floating_point_pow}(X, Y, Z).$

« ! » 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.

$Y \$:= \sin(X) \iff \text{floating_point_sin}(X, Y).$

« ! » 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.

$Y \$:= \cos(X) \iff \text{floating_point_cos}(X, Y).$

« ! » 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.

$Y \$:= \tan(X) \iff \text{floating_point_tan}(X, Y).$

« ! » 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.

$Y \$:= \text{asin}(X) \iff \text{floating_point_asin}(X, Y).$

« ! » 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.

$Y \$:= \text{acos}(X) \iff \text{floating_point_acos}(X, Y).$

« ! » 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.

$Y \$:= \text{atan}(X) \iff \text{floating_point_atan}(X, Y).$

« ! » 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 predicate 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 predicate 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 character(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 occurs. 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 character(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 occurs. 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 occurs. 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 occurs. 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 declared as public), **Code** is unified with an atom \square .

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 first 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 first 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 occurs. 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)	<code>X == Y</code>	<code>equal(X,Y)</code>
	<code>X \= Y</code>	<code>not_equal(X,Y)</code>
	<code>X < Y</code>	<code>less_than(X,Y)</code>
	<code>X > Y</code>	<code>less_than(Y,X)</code>
	<code>X <= Y</code>	<code>not_less_than(Y,X)</code>
	<code>X >= Y</code>	<code>not_less_than(X,Y)</code>

For Floating Point

Priority	Operator	Expanded pattern
700 (xfx)	<code>X \$== Y</code>	<code>floating_point_equal(X,Y)</code>
	<code>X \$\= Y</code>	<code>floating_point_not_equal(X,Y)</code>
	<code>X \$< Y</code>	<code>floating_point_less_than(X,Y)</code>
	<code>X \$> Y</code>	<code>floating_point_less_than(Y,X)</code>
	<code>X \$<= Y</code>	<code>floating_point_not_less_than(Y,X)</code>
	<code>X \$>= Y</code>	<code>floating_point_not_less_than(X,Y)</code>

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
500	yfx	$X + 1$	Z	increment(X,Z)	
	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	$X \setminus / Y$	Z	or(X,Y,Z)	
	yfx	$X /\setminus Y$	Z	and(X,Y,Z)	
400	yfx	$X \text{ xor } Y$	Z	exclusive_or(X,Y,Z)	
	yfx	$X * Y$	Z	multiply(X,Y,Z)	floating_point_multiply(X,Y,Z)
	yfx	X / Y	Z	divide(X,Y,Z)	floating_point_divide(X,Y,Z)
	yfx	$X \ll Y$	Z	shift_left(X,Y,Z)	
300	yfx	$X \gg Y$	Z	shift_right(X,Y,Z)	
	xfx	$X \text{ mod } Y$	Z	modulo(X,Y,Z)	
as a term	xfy	$X ** Y$	Z		floating_point_pow(X,Y,Z)
		abs(X)	Z	abs(X,Z)	floating_point_abs(X,Z)
		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)
		$\setminus(X)$	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 $\sim(\text{expression})$ or $\$(\text{expression})$ is used to explicitly require expansion of the expression.

ex: $p(\sim(X+Y+1))$ becomes $\text{add}(X,Y,A), \text{add}(A,1,B), p(B)$.

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 inconvenient 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,Ot, Cnt1),
    loop(In1, Oh,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 explicitly 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.

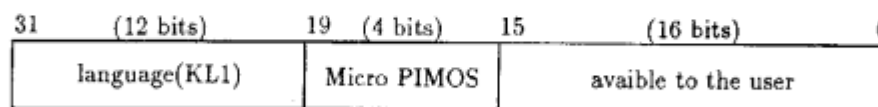


Figure 7: Sho-en exception tag

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 **ArgsList**. **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(IOSStream)

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.

load(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 mr	... Displays variables whose MRB is set.
	a or all	... Displays variables of both one and mr 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 is 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.
loadaddr	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 ($0 \leq N < \text{NumberOfVariables}$) 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(ˆTerm, ˆNumberOfVariables, ˆStatus)

Almost same as **getft/2**, and see **gett/2** about **Status**.

skip(Char)

Skips until the character 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 characters by certain length specified by **Count** extracted from 8 bits string. If the length of **Buffer** 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	::	$\$VAR(N, VN) \rightarrow VN$ (Variable name string).
		$\$VAR(N) \rightarrow A, B, C \dots$
nu – Number Mode	::	$\$VAR(N, VN) \rightarrow _N$ (Variable number).
		$\$VAR(N) \rightarrow _N$ (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** ($1 \leq \text{Precedence} \leq 1200$), 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 ($fx \leftrightarrow fy$, $xf \leftrightarrow yf$, $xfy \leftrightarrow xfx \leftrightarrow yfx$), 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 \square 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 KLI. 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 mentioned 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, ~Buffer, ~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 `load_dir` 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 `load_dir` and `save_dir` 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 scheduling 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 uninstantiated 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 uninstantiated argument.
Susp  :: Suspension due to an uninstantiated argument which specifies the ratio calcu-
        lating 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 2³¹.

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 scheduling 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 redisplay 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 :: ?- module: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,HOKoo)]
*** 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 been 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 applied 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_attention/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

These 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 occurred 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 mode.

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**, with 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 explicitly 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_module** 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 developed 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... $C_b \times \text{first-element} + C_m \times \text{middle-element} + C_e \times \text{last-element} + \text{string-length}$. C_b , C_m and C_e are the same as KL0 built-in predicates.
- list...Car hash value + 5 \times Cdr hash value.
- vector...number of elements + sum (for the first, middle and last elements) of ((2 to the power of element rank+ 1) \times 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 `[]`.

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 standard 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'	pdss_window_device
* directory	pdss_file_device
* file	pdss_timer_device
* window	pdss_runtime_active_unify
* mpimos_io_device	pdss_runtime_debug
monogyny_list_index	pdss_runtime_exception_handling
mpimos_booster	pdss_runtime_body_builtin
mpimos_builtin_predicate	klicmp_blttbl
mpimos_cmd_basic	klicmp_command
mpimos_cmd_code	klicmp_compile
mpimos_cmd_debug	klicmp_mrb
mpimos_cmd_directory	klicmp_normalize
mpimos_cmd_environment	klicmp_output
mpimos_cmd_util	klicmp_reader
mpimos_code_manager	klicmp_register
mpimos_command_interpreter	klicmp_macro
mpimos_libdir	klicmp_macro_arg
mpimos_directory	klicmp_mtbl
mpimos_directory_device_driver	klicmp_struct
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	

Appendix-5 Reserved operator names

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

1200	xfx	:-	150	xf	++
1200	fx	:-	150	xf	--
1200	xfx	-->	100	xfx	#
1150	fx	module	100	fx	#
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	z			

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
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
```

\$:=, \$<= 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.

31	(12 bits)	19	(4 bits)	15	(16 bits)	0
language(KL1)		Micro PIMOS		avaible to the user		

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:xxxxxx( ... )
merge( ... ) in mpimos_file:xxxxx/x
merge( ... ) in mpimos_window:xxxxx/x
```


Index

abs	12,14,23	Code Commands	31
acos	16,23	Debug Commands	32
add	11,23	Directory Commands	32
add_op	30,37	Environment Commands	33
add_resource	4	Command Input Format	29
add_resource	3	comp	31
alternatively	8	Compile	31,67,69
and	12,23	Command Procedures	69
append_string	19	compile	31
apply	5,20,30	complement	13,23
Arithmetic Comparison Macros	22	Conditional Branch Macros	27
Arithmetic Operation Macros	22	Constant Description Macros	21
asin	16,23	Constraint of Macro Expansion	23
atan	16,17,23	consume_resource	21
atom	10	Control Stream	2,4
Atoms	8	cos	16,23
atom_name	19	cosh	17,23
atom_number	19	cputime	30
auto_load function	39	create	34,35,38
backtrace	32	Cross Reference Check	33
Bag	56	current_priority	21
beep	38	current_processor	21
buffer_length	37	Data Types	8
Built-in Predicates	9,30	Deadlock	6,47
Arithmetic Comparison(Floating Point) ...	13	deadlock	47
Arithmetic Comparison(Integer)	10	debug	32
Arithmetic Operations(Floating Point) ...	14	decrement	12,23
Arithmetic Operations(Integer)	11	delete	38
Atom Predicates	19	Device Stream	39
Code Predicates	19	Command	39
Conversion	17	Securing Device Stream	39
Second Order Function	20	diff	10
Special I/O Functions	20	directory	38
Stream Support	20	Directory Management	38
String Predicates	18	Acquisition of Command Stream	38
Type Checking	9	Commands	38
Vector Predicates	17	Directory Command Stream	38
cd	32	display_console	20
change_op_pool	31,37	divide	11,23
ch_savedir	31	dload	31
Clause Ordering	8	do	37
clear	38	Environment Variables	33
close	37	equal	10,22
Code	3	Equality	2
Code Device	55	Evaluation	8
Code Device Stream	55	Exception	2,40
Code Management	39	Exception Code	65
Code Trace	43	Exception Information	5,40
code_to_predicate	19	Exception Tag	29,66
Commafn Interpreter		exclusive_or	12,23
Commands	30	execute	3
Command Interpreter	29	exp	15,23
Commands		Failure	2
Basic Commands	30		

tanh	17,23	✓	12,23
Token Format	52	\=	10,22
trace	32	\	13,23
Tracer	43	“	23
Commands	44	‘	23
Trace Points	43	-	11,12,14,23
		-	23
unbound	10		
Unbound Variables	8		
Unification Macros	22		
varchk	32,33		
Variables Check	32		
vector	10,17		
Vectors	9		
vector_element	18		
wait	9		
Window	34,50,51-54		
window	30,34		
with_macro	28,50		
xor	12,23		
xref	33		
.....	29		
#	21,28		
\$:=	22		
\$<=	22,24,25		
\$<	13,22		
\$:=	13,22		
\$=<	13,22		
\$=\=	13,22		
\$>=	13,22		
\$>	13,22		
\$-	23		
&	24		
**	16,23		
*	11,14,23		
+	11,12,14,23		
,	29		
/\	12,23		
/	11,14,23		
:=	22		
!	29		
<<=	24,25		
<<	13,23		
<=	22,24,25		
<	10,22		
=:=	10,22		
=<	11,22		
=>	28		
=\=	10,22		
=	22		
>=	11,22		
>>	13,23		
-->	24		
->	27		
>	10,22		
@	6		

Module definition	8	Queue	57
modulo	11,23	raise	20
Monitor Variables	45	read_console	20
mpimos_file_device	39	remove_op	30,37
mpimos_timer_device	39	replace_op_pool	31,37
mpimos_window_device	39	Report Stream	2,4
multiply	11,23	Exception Information	5
new_atom	19	Statistic Information	5
new_string	18	Status Information	5
new_vector	17	resetenv	34
nl	36	reset_profile	33
nobacktrace	32	Resource	2
nodebug	32	Maximum Number of Reductions	3
nospy	32	Resource_low Exception	3
notrace	32	Resource Management Functions	2
not_equal	10,22	rm	32
not_less_than	11,22	save	31
Oldnew Argument	25	save_all	31
open	38	Scheduling	8
operator	30,31,37	Sequentiality	2
or	12,23	setenv	34
otherwise	8	set_string_element	18
pathname	38	set_substring	19
pdsscmp	69	set_vector_element	18
PDSS-mode	67	Shared Argument	24
PDSS Configuration	1	shift_left	13,23
PDSS Invocation	41	shift_right	13,23
Optional Parameters	41	Sho-en	2
Stand-alone	41	Generation	3
Under GNU-Emacs	41	Tag	3,66
PIMOS Common Utilities	56	Sho-en Priority	3
Comparison	56	Sho-en System Module	3
Hashing	56	Sho-en Tag	3,66
Pool without Key	56	show	38
Stack	57	sin	16,23
predicate_to_code	19	sinh	17,23
printenv	34	skip	36
print_depth	37	Sorted Bag	57
print_length	36	spy	32
print_var_mode	37	Spying	43
Priority	6	Code Spying	43
Logical Priority	6	Goal Spying	43
Physical Priority	6	spying	32
Rate Specification	6	Spy Flags	44
Relative Self Specification	6	sqrt	15,23
Priority Management	3	stat	30
profile	33	Stream argument	25
prompt	38	string	10,18
public	8,31	Strings	9,21
Public Declaration	8	String Argument	26
putb	36	string_element	18
putc	36	substring	19
putl	36	subtract	11,23
putt	36	Syntax	6
puttq	36	tab	36
put_console	20	take	30
pwd	32	tan	16,23

File	34,35,50,51-54
file	35
float	17,23
Floating Point	9
floating_point	10
floating_point_abs	14,23
floating_point_acos	16,23
floating_point_add	14,23
floating_point_asin	16,23
floating_point_atan	16,23
floating_point_cos	16,23
floating_point_cosh	17,23
floating_point_divide	14,23
floating_point_equal	13,22
floating_point_exp	15,23
floating_point_floor	15,23
floating_point_less_than	13,22
floating_point_ln	15,23
floating_point_log	15,23
floating_point_max	15,23
floating_point_min	14,23
floating_point_minus	14,23
floating_point_multiply	14,23
floating_point_not_equal	13,22
floating_point_not_less_than	13,22
floating_point_pow	15,23
floating_point_sin	16,23
floating_point_sinh	17,23
floating_point_sqrt	15,23
floating_point_subtract	14,23
floating_point_tan	16,23
floating_point_tanh	17,23
floating_point_to_integer	17,23
floor	15,23
flush	37
gc	30
getb	35
getc	35
getenv	34
getft	35,36
getl	35
gett	35
Goal Definition Pertaining to a Different Module	8
Goal Trace	43
Guard	2
halt	31
hash	21
help	30
hide	38
implicit	24
Implicit Argument Macros	24
Access to Arguments	24
Expansion	24
Expansion Control	26
Global Declaration	24
Local Declaration	24
Terminating Process	26

Update Arguments	24
increment	11,23
Input/Output Device	39
Device Commands	50
Device Stream	50
File Device	50
Securing Device Stream	50
Timer Device	51
Window Device	50
int	17,23
integer	10
Integers	9
integer_to_floating_point	17,23
intern_atom	19
I/O Devices	50
I/O Functions	34
Commands	35
Command Stream	34
Command Stream Attachment	34
Control Commands	37
Control of Output Format	36
File	35
Grouped Processing	37
Input Commands	35
Operators	37
Output Buffer Commands	37
Output Commands	36
Window	34
Window Commands	37
Keyed Bag	57
Keyed Pool	57
Keyed Set	58
Keyed Sorted Bag	59
Keyed Sorted Set	59
kl1-mode	67
KL1 Language Specification	2
less_than	10,22
list	10
listing	31,38
Lists	9
ln	15,23
load	31
local_implicit	24
log	15,23
ls	32
Macros	21
Macro Library	28
max	12,15,23
Maximum Priority	3
merge	20
Micro PIMOS	29
min	12,15,23
Minimum Priority	3
minus	11,23
mod	11,23
Module	2
module	8