ProLog

by BIM

VOLUME ONE

Credits and Acknowledgements

Prolog by BIM (originally ***BIM_Prolog***) is a Prolog implementation resulting from a joint project between BIM and the Department of Computer Science of the K.U.L. (Katholieke Universiteit Leuven) in Belgium. This project was made possible thanks to ongoing support from DPWB/SPPS (Diensten voor de Programmatie van het Wetenschapsbeleid - Services pour la Programmation de la Politique Scientifique, Belgium) and the European Commission in the context of the ESPRIT research programme.

Trademarks

INDEX

Symbols

A

B

C

D

E

F

G

H

I

M

N

N

O

P

Q

R

S

T

U

V

W

Z

PRINCIPAL COMPONENTS

Principal Components

Contents

ProLog by BIM - Reference Manual
Principal Components
Chapter 1

The Engine

1.1 Invoking the engine

Starting

The ProLog engine is invoked by:

% BIMprolog <command line options>

The full syntax of the command line arguments is (in BNF-like format):

<command line options>:
(<ProLog options>)* |
(<ProLog options>)* - (<user-defined options>)*

<ProLog options>:
-<category><name> |
-<category><name><value> |
<file>

<category>:

- the table options (T),
- the please options (P),
- the debug options (D),
- the compiler options (C),
- the special options

<name> :
A full description of the option names category per category is given further on.

<value> :
+ (on) | - (off) | <number> | <number><scale>
When no <value> is specified, the reverse of the previous value is taken when appropriate.

<scale>:
k (1024) | m (1,048,576) | p (%)

<file> :
The .pro extension for source files is optional. If a file name is given, for which no .pro-extended file exists, the original name will be taken as the complete file name. When several files are specified, they are consulted in the given order.

<user-defined options>:
All options after the delimiter (the '-' sign between blanks) are considered to be user defined options. They are available to the application through the **argc/1**, **argv/1**, **argv/2** predicates.

Ending

A Prolog session is terminated by typing the end-of-file character (^D), or using the builtin predicates **stop/0** or **halt/0 .**

Help

An overview of the command line options can be interactively obtained by :

```
% BIMprolog -help
```

Consulting files

The different kinds of files are :

file.**pro**	ProLog source file
file.**wic**	intermediate code file ; this file is portable between all Sun architectures.
file.**lis**	listing file (created whenever errors or warnings occur and the -Cl compiling option is set).

Each of the consulted files may contain any number of queries at any place in the file. Such queries are executed before the clauses physically following it in the file are consulted, and the queries may contain requests to consult other files.

A file does not need to be compiled before consulting. When a source code file is consulted which is more recent than its corresponding intermediate code file, or if this intermediate code file does not exist, it is recompiled automatically.

When consulting the file *file.pro*, this file is not the current input stream. So queries which invoke read builtins, never read from the file being consulted.

Options which preceed the filename, and which influence parsing or compilation, are automatically passed to the compiler. So the UNIX-command

```
% BIMprolog -Pc file
```

compiles, if necessary, the file 'file.pro' with compatibility syntax (DEC-10) and continues with the top-level in that mode, while the UNIX-command

```
% BIMprolog file -Pc
```

compiles the file in ProLog syntax (default) and continues with the top-level in compatibility syntax.

1.2 Table options

Table description

ProLog uses several tables for storing user programs, user data and system data:

Heap (H) :

used to store values of variables, to construct and store structured terms, to store certain information that is needed during backtracking. Entries that become obsolete are removed during garbage collection.

Stack (S) :

contains backtracking information and the predicate environments.

Code (C), (I) :

contains the intermediate code. The static (Compiled) predicates take more space than dynamic (Interpreted) ones but the execution will be much faster.

Data (D) :

one entry for each atom, real and pointer; entries that become obsolete, are removed during garbage collection.

Functors (F) :

one entry for each functor that has been used.

Record Keys (R) :

one entry for each tuple (key1, key2) used in a record predicate; erased keys are removed on garbage collection.

Backup Heap (B) :

contains structured terms that are recorded; terms associated with erased keys are removed during garbage collection.

Temporary and Permanent Strings (Sp), (St) :

contains temporary and permanent strings essentially used in predicate names and external language manipulation.

System Tables :

these tables are totally hidden to the user.

Table size

The tables used by the engine can be divided into three classes :

- Tables that are only slightly affected by the application.
- Tables whose size is influenced by the complexity of the executed goal.
- Tables whose size depends on the size of the program.

These differences are reflected in the way the size of the tables are determined.

Tables whose size is independent of the user's applications are kept invisible to the user, and have a size that is determined by the system. Some of these tables expand or shrink automatically, following the resource requirements of the executing system. This all happens transparently for the user, unless expansion becomes impossible due to a lack of available (virtual) memory, in which case, an overflow occurs.

Tables that are largely influenced by the execution, have an automatic expansion capability. They will expand to follow the needs of the executing goal. Since they are dependent on the user's program, the user has the possibility to determine the size and the expansion behavior of these tables.

Tables whose size depends on the program, but less on the execution, are currently not expandable. Here too, the user can determine the size of the table. In the future, these tables may also be made auto-expandable.

Table control

All tables that are controllable by the user, are characterised by a minimum size (**m**), which is fixed by the system, and four parameters that can be set by the user or by its application.

b (base)..... Base size of the table, before any expansion

t (threshold)..... Threshold for deciding about table expansion

e (expansion)..... Amount used for a single expansion

l (limit) ..... Hard upper limit, no expansion beyond this size

The following table reflects the default settings for the different visible tables :

Table Name		Customizable	Garbage Collection	Expandable	Minimal size m	Base size b	Treshold t	Expansion e	Limit size l
Heap	(H)	Yes	Yes	Yes	4k	32k	25%	100%	1m
Stack	(S)	Yes	No	Yes	4k	32k	25%	100%	1m
Data	(D)	Yes	Yes	No	2k	4k	-	-	1m
Functor	(F)	Yes	No	No	1k	2k	-	-	1m
Interpreted Code	(I)	Yes	Yes	No	4k	16k	-	-	1m
Compiled Code	(C)	Yes	No	No	4k	16k	-	-	1m
Record Keys	(R)	Yes	Yes	No	1k	2k	-	-	1m
Backup Heap	(B)	Yes	Yes	No	1k	8k	-	-	1m
Permanent String	(Sp)	No	No	Yes	-	-	-	-	-
Temporary String	(St)	No	Yes	Yes	-	-	-	-	-

These parameters can be set either when invoking the engine by using the command line table option **-T**, or when the system is running, using the **table/2** builtin. The following constraints apply:

- The base **b** cannot be set to a value smaller than the minimum size (and will be set to **m** when trying so).
- The limit **l** must be bigger than the minimum size **m**.
- When changing parameters when the system is up, the **b** parameter cannot be made bigger than the actual table size, while the **l** cannot be made smaller than that size.

When starting up the system, each table is created with **b** as its size. Whenever the table becomes full, the garbage collector is activated, if there is one for that table. After garbage collection, or immediately in case there is no garbage collector, the size of the free part of the table is compared with the threshold **t**. If there is less free space than **t**, the table is expanded with an amount, determined by the expansion parameter **e**, unless this would make the table bigger than the limit size **l**. In that case, a fatal overflow occurs. A non-expandable table will immediately cause a fatal overflow if the free part becomes smaller than the threshold **t**, after garbage collection.

Parameter setting

Parameter values can either be absolute, i.e. expressed in number of table entries, or relative, i.e. expressed as a percentage.

The meaning of a relative value depends on the parameter, and is given in the table below :

Parameter	Meaning of relative value
b (base)	None (not allowed)
t (threshold)	Percentage of the actual table size
e (expansion)	Percentage of the actual table size
l (limit)	Percentage of the base table size (**b**)

A parameter setting is given as an atom, composed of the identification letter for the parameter and a value (with no space in between). The value is either a simple integer number, or an integer number followed immediately by a scale factor, which is a single character.

The following rule describes this :

<ParamSetting> ==> <ParamId><ParamValue>
<ParamValue> ==> <Integer> | <Integer><Scale>
<ParamId> ==> b | t | e | l
<Scale> ==> k | m | p | %

The meaning of the scale factors is :

k ..... kilo..... 1024

m .....mega ..... 1,048,576 (= 1024*1024)

p or %.....percentage

Global table option

Name	Value	Description
warn or **w**	+/-	Enable or disable messages when doing garbage collection or table expansion

This global table option can be set in the BIMprolog command line or by using the builtin **table/2**.

Table command line option

The table command line option is **-T**, and is used as follows :

% BIMprolog <TableOption>

<TableOption>	==>	-T<OptionId><OptionValue> | -T<TableId><TableCommand> | -T<TableId><ParamSettingList>
<OptionId>	==>	w | warn
<OptionValue>	==>	+ | -
<ParamSettingList>	==>	<ParamSetting> | <ParamSetting>,<ParamSettingList>

These options should be placed before any file names on the command line, although they may occur anywhere. Placing them at the beginning ensures they will be honoured completely. As soon as a file has to be consulted during startup of the system, it has to allocate the tables and thus changes to an 'up' state. As a result, the run-time constraints on setting table parameters become active. The major consequence of this is that the base table size (**b**) cannot be changed anymore.

Help

An overview of the table options can be interactively obtained by :

% BIMprolog -Th

1.3 Please options

The please options can be used to toggle certain switches in the ***ProLog*** engine and they may be mixed with the files to be consulted, in this case and if it is meaningful, the option is passed to the compiler :

-Pae indicates if the \ escape character is active or not.
default : - (not active).

-Pc work in DEC-system 10 compatibility mode.
default : - (native ***ProLog*** syntax).

-Pd starts the interactive session producing debug code.
default : - (non debug code).

-Pe Translates the in_line evaluation.
default: + (on).

-Ped starts ***ProLog*** with its window-oriented debugger environment.
default : - (no debugger environment active).

-Pem starts ***ProLog*** with its window-oriented programming environment monitor.
default : - (no programming environment active).

-Pfr*atom* The atom given with option fr specifies the format with which a real is printed out. It can be any format that may be used in a formatted print. See also **printf/2-3**.
default : '%.15e'.

-Pq starts the interactive session with querymode on (this means that the ***ProLog*** engine will prompt with a '?-', such that a query can be entered without having to type the '?-'); the next occurrence of this option cancels the previous.
default : - (querymode off).

-Pra*atom* The end-of-file atom is set to the atom *atom*.
default : 'end_of_file'.

-Prc*val* The end-of-file character is set to the integer *val*.
default : -1 .

-Prf switches the behavior when an end-of-file character is encountered.
default : + (fail upon reading EOF).

-Ps starts the interactive session with showsolution turned on. In this mode whenever a query is solved, the value of the variables in the query are printed out. If other solutions exist typing a ';' will indicate that they are required, a <CR> means termination of the goal evaluation.
default : - (show_solution off).

-Pw switches the general warning flag on and off (suppress all warnings).
default : + (give warnings).

-Pwd*depth* the term is only considered up to a depth of *depth*, when printing out a structured term; All subterms of that level are printed as '...'; giving 0 as *depth* results in '...' for the complete term; a negative number is read as infinity.
default : -1 (infinity).

-Pwf Determines whether output operations have to be followed by a flush.
default : + (on).

-Pwm provides explicit module qualification when printing terms.
default : - (no module qualification).

-Pwp usage of prefix functor form in output.
default : - (no prefix functor form).

-Pwq usage of quotes in output.
default : - (no quotes).

For more details on the please switches, see the builtin predicate **please/2**.

Help

An overview of the please options can be interactively obtained by :

```
% BIMprolog -Ph
```

1.4 Debug options

-Dp indicates whether a command must be requested after activation of the debugger or a (re)consult.
default: + (prompt is requested).

-Dc control of choice point destruction under debugger execution (see also **debug/2** builtin).
default : - (only marking of choice points).

-Dtr switches recording of trace on or off.
default : + (on).

-Dwm prints explicit module qualification in debugger output.
default : - (no module qualification).

-Dwq prints necessarry quotes in debugger output.
default : - (no quotes).

-Dwp provides prefix operators in debugger output.
default : - (no prefix).

-Dtd defines the allowed depth of the recorded trace.
default : -1 (infinity).

-Dwd nesting depth for structured terms in debugger output.
default : -1 (infinity).

Help

An overview of the debug options can be interactively obtained by :

% BIMprolog -Dh

More details about debug options can be found in the Part on the Debugger.

1.5 Compiler options

Compiler options, can be specified in the BIMprolog command line. They have to be preceeded by '-C' and must be specified one at a time.

For example:

% BIMprolog -Cw -Cd filename

An overview of the compiler options can be interactively obtained by:

% BIMprolog -Ch

A full description of the compiler options is given in chapter 1.2 .

1.6 Special options

Initial consult

The ProLog system, upon initialization, silently consults the file **.pro** if it exists in the current directory or in the user's home directory. This .pro file can be used to set preferred default options.

For example:

% BIMprolog -Pem -Ps+ -Pq+

can be obtained by putting the following queries in the .pro file :

% cat.pro
?- please (em,on).
?- please (s,on).
?- please (q,on).

The following option must follow the table size options immediately and must come before any file to be consulted :

-**f**[*startfile*] do not consult the *.pro* file (in the home directory) as user initialization, but instead, take *startfile* as startup file.
If no argument is given with this option, no initial consult will be performed.

Consulting libraries

Consulting ProLog library files :

-L*libfile* consult the file *libfile* which is first found in the possible library directories. Library directories are searched in the following order:
1. All directories given in the library path environment variable ($BIM_PROLOG_LIB), in the same order as given in that variable.
2. The system library directory ($BIM_PROLOG_DIR/lib/).
3. The working directory.

-H*file* consult the specified *file* from the ***ProLog*** home directory ($BIM_PROLOG_DIR).

Saved state

-r*file* Restores the ***ProLog*** session by using the saved state file, *file*. See also the builtin **save**/1 .

1.7 System size

The following table gives for each table the number of bytes used per entry:

Table	Number of bytes per entry
H	**5**
S	**5**
D	**11**
F	**14**
I	**8**
C	**36 (sun3) / 44 (sun4) / 48 (sun386i)**
R	**16**
B	**5**

The size of the kernel is:

sun4	**700k**
sun3	**640k**
sun386	**700k**

With these figures, the total core needed by a ProLog session can be calculated.

For example :

The core needed to start up *the engine with default options* on a sun4 is:

160 kilo bytes	(32k H)
160 kilo bytes	(32k S)
44 kilo bytes	(4k D)
28 kilo bytes	(2k F)
128 kilo bytes	(16k I)
704 kilo bytes	(16k C)
32 kilo bytes	(2k R)
40 kilo bytes	(8k B)
700 kilo bytes	(kernel sun4)

1,996 kilo bytes	

ProLog by BIM - Reference Manual
Principal Components
Chapter 2

The Compiler

2.1 Invoking the compiler

The ***ProLog*** compiler translates ***ProLog*** source code into a form more suitable for execution. One can easily manage without calling it explicitly, since the ***ProLog*** system will invoke it when consulting a source file which is more recent than its translated form. However, sometimes it is useful to use it to check the syntax of source files, or to speed up the consult when actually running the program.

Compiling all sources before starting the engine makes more swap space available for the engine. Otherwise, enough swap space must be available for both the compiler and the engine to run simultaneously.

The compiler is called by :

% BIMpcomp [-acdehlpwxA] file1[.pro] [file2[.pro] ...]

The ***ProLog*** compiler converts a Prolog source file into a ***ProLog*** object file containing intermediate code. The object file produced has the same name as the source file but with an extension *.wic*, **file.wic**. If the compilation was not successful, the errors (syntactic or semantic) and warnings are printed on standard error. A listing file (whose name is the filename extended with *.lis*) can be generated by setting the -l option in the compilation command.

Also, *BIMpcomp* displays (on stderr) a message notifying the unsuccessful completion of the compilation. (More details about the messages of BIMpcomp can be found in Appendix).

2.2 Compiler options

The options that are interpreted by BIMpcomp are the following :

-a compiles as if the source file contained a '**:- alldynamic**' directive.
default : inactive.

-c terms are parsed according to the DEC-10 Prolog syntax. Definite clause grammar rules are only translated with this option.
default : inactive.

-d compiles the complete file to debugger code. This has the same effect as putting a '**:- setdebug**' directive in the beginning of the file (without any following '**:- setnodebug**' directives).
default : inactive.

-e translates the in-line evaluation.
default : translation is done.

-h help option.

-l error messages and warnings are redirected into a listing file whose name is the source filename extended with *.lis* .
default : error messages are written to stderr.

-p includes operator declarations in the object file.
The effect is the same as if one calls the corresponding operator predicates interactively.
(see also the builtin predicate **consult/1**).
default : operator predicates are not active interactively.

-w controls all warnings.
default : warnings are shown.

-x activates the \ escape character.
default : inactive.

-A*n* if *n* is a positive integer, the sizes of the internal tables of the compiler are multiplied by *n* to be able to compile larger programs. Otherwise, when *n* is a negative integer, the sizes of the internal tables are divided by *-n*, such that the compiler consumes less core.

2.3 Compiler tables

If during compilation, the default size of the compiler tables is not big enough, the system increases them automatically. Nevertheless, the user has the possibility to manually increase the compiler tables with the **-A***n* compiler option. This helps speed up compilation time for huge files.

2.4 Static & dynamic procedures

The compiler distinguishes between two kinds of procedures : dynamic and static. By default all procedures are static, and to make a procedure dynamic, an explicit declaration is needed.

For example

:- dynamic foo/3 .

To make all procedures in a file dynamic :

:- alldynamic.

The difference between static and dynamic procedures is that once loaded - by consulting a file - the static procedure cannot be modified by consult, assert or retract. Attempts to do so, will result in error messages and/or warnings. *Note* that reconsult can replace a static procedure with an alternative definition.
Dynamic procedures can be modified at run time, while *static* procedures are executed more efficiently since they are compiled into native code.

Mode declarations and indexing can be specified in order to improve static and dynamic predicates.
Additional hashing is available for dynamic predicates (see Builtin Predicates Chapter 2 - In-core DB manipulation - Optimization, and Directives - Chapter 1- Optimization).

Note that the choice between static and dynamic predicates does not affect the ***ProLog*** debugger. The use of the debugger option allows to (re)consult, assert or retract both kinds of procedures.

2.5 Filename expansion

Wherever a file name is expected, the file can be given either with its full path or with a relative path. In addition, a number of meta symbols can be used to include some environment dependent parts in the file name. These are expanded automatically by ***ProLog***.

Places where this expansion takes place are :

BIMpcomp and BIMprolog arguments
include/1
consult/1
fopen/3, tell/1, see/1
extern_load/2, extern_load/3

There is also a builtin **expand_path/2** for explicitly doing this expansion.

The following meta symbols are recognized. They can only be used at the beginning of the filename.

Symbol	expansion
~	Home directory of the user ($HOME).
~user	Home directory of 'user'.
$VAR	Value of VAR in environment.
-L	The first library directory in which the given filename is found. Library directories are searched in the following order: 1. All directories given in the library path environment variable ($BIM_PROLOG_LIB). In the same order as given in that variable. 2. The system library directory ($BIM_PROLOG_DIR/lib/). 3. The working directory.
-H*file*	Consult *file* from the ***ProLog*** home directory ($BIM_PROLOG_DIR).

ProLog by BIM - Reference Manual
Principal Components
Chapter 3

The Linker

3.1 Introduction

The ***ProLog*** linker (**BIMlinker**) can be used to create customized versions of ***ProLog***. There are two possible levels that can be created : a customized development system, or a run-time system.

With a development version, the user of the created system still has access to the full ***ProLog*** development environment. This environment can be adapted with other defaults and extended with some specific programs and data as decided by the generator of the customized system.

A run-time version is meant to be an end-user product in which it is not visible what the underlying engine is. Therefore, it does not include the ***ProLog*** development environment. It includes the whole application of the run-time generator, and can only be used to run that specific application.

3.2 Using BIMlinker

The ***ProLog*** system can be linked with a number of external compiled object modules and libraries into a single executable. A set of compiled ***ProLog*** files can also be linked into the new system. It can be tuned for alternative default options.

A generic form of a **BIMlinker** call is :

BIMlinker [*link_options*] * - [*targets*] *

The linker options *link_options* must be given as first arguments. They are separated by a single '-' sign from the other arguments that are meant to be ***ProLog*** options and target Prolog file names. The linker options include both options for **BIMlinker** and for the UNIX linker **ld**. All options that are not recognized by **BIMlinker** are passed to **ld**.

The *targets* can be any Prolog files and ***ProLog*** options. For files that are mentioned as targets, only the .wic files are opened. It is not verified whether they are more recent than the corresponding Prolog source file or even if this exists. Options can be given to set the new default values in the generated system. All indicated Prolog files will be part of the initial environment of the new system. If they contain external load declarations, these are executed by **BIMlinker**. All necessary external code will be linked in the created system. Prolog code will be loaded automatically each time the system is started up. For a run-time system, this code is restored from a data file that is generated by **BIMlinker**. As a result, the given files are no longer needed after generation of the run-time system. In a development system, the code is (silently) loaded from the compiled Prolog files. Which means that these files cannot be removed. (See further for important notes on file naming.)

To use **BIMlinker**, the environment variable BIM_PROLOG_DIR must be defined and indicate the home directory for ***ProLog***.

The generated executable file, and for run-time systems also the data file, are left in the working directory. It is the user's responsibility to move these files to the right location.

Options

BIMlinker options are used to specify the environment of the generated system and to control how it is to be generated.

-o *exec_file*	Names the executable output file as *exec_file*. Default : -Hbin/BIMprolog
-r *data_file*	A run-time system is generated with data file *data_file*. Default : no run-time generation.
-e *error_file*	Error descriptions are taken from *error_file*. Default : $BIM_PROLOG_DIR/install/errors.o
-m *main_file*	The file containing the main() routine, is *main_file*. Default : $BIM_PROLOG_DIR/install/main.o
-h *home_var*	The environment variable *home_var* will be used to find the new system's home directory. Default : BIM_PROLOG_DIR
-l *libs_var*	The environment variable *libs_var* will be used to find the library directory paths. Default : BIM_PROLOG_LIB
-x	Enable incremental linking in the resulting system. Default : incremental linking is disabled by stripping the symbol table, saving space in the executable file.
-g	Retain the complete symbol table, to enable symbolic debugging of linked procedures. Default : non-external symbols are stripped from the symbol table, even with -x.
-q	Link quietly. Default : diagnostics are reported during linking.

Environment variables

ProLog uses two environment variables, one to determine its home directory, and one as a library directory path. The home directory is used to find system files, and in expanding -H in file names (see below for file name expansion). The other variable is supposed to contain a list of directory paths, separated with ':'. These directories are searched for library files when expanding -L in file names. The names of these variables can be determined when creating a new system with the linker. This is particularly useful when a run-time system is generated : the application has no relation anymore with ***ProLog***, and therefore may not reference its home directory anymore.

Symbol tables

Using **BIMlinker**, one can control the contents of the symbol table of the generated executable. By default, it is completely stripped. This saves space in the executable file. However, as a result, the generated system cannot be used to link incrementally additional external programs. To enable this incremental linking, a part of the symbol table must be retained in the created executable. The -x option indicates this. For debugging, this does not suffice; the whole symbol table, including debugger information, must be retained. This can be indicated with the -g option. A system that is generated (and compiled) with -g, can be debugged with symbolic debuggers like **dbx**.

A system that does not have to be able to link incrementally, will also be linked to use shared libraries instead of static libraries. This potentially saves a lot of space in the executable file. To force usage of static libraries, the **ld** linker option -Bstatic must be added to the linker option list.

Error description file

An error description file must be a compiled file in UNIX object file format. It is generated from a Prolog source file in two steps. First a C source file is generated with the program **BIMerrgen**. If the error description file is called *errors.pro*, then

BIMerrgen *errors*

generates a C file *errors.c* that has to be compiled with

cc -c *errors.c*

creating the object file *errors.o*.

A specification of the contents of an error description file can be found in *General Builtins - Error Handling*.

Main file

A main file is a file that contains the routine *main()* that is called as the toplevel routine in the generated system. The default main file makes ***ProLog*** become the master of the process. By creating your own main file, it is possible to make the external program master of the process. The main file must be compiled prior to calling **BIMlinker**. (See section *Embedded systems* on how to make the external program master).

Filenames and expansion

The file names in the linker options and the target file names, can be given in several ways, resulting in the files being placed in different directories. It is important to choose the right way, as the files will be searched for automatically in these places, on each invocation of the new system.

A file location can be absolute, relative or environment dependent.

absolute location
Path starting with '/': this absolute path name.
Path not starting with '.': this path, extended to an absolute path .

relative location
Path starting with '.' : the given relative path, from the working directory in which the system is started up.

environment dependent location
Path starting with '-H': relative from the application's home directory.
Path starting with '-L': relative from one of the library directories.

An absolute location should be used if the files are always on the same location.

A relative location is useful when the files are always on a fixed relative distance from the directory in which the system is started up.

The most flexible way is the environment dependent location, especially for system related files. The whole system can be moved to any directory : by setting the system's home directory environment variable, the system files will be found at the new place.

Environment dependent file names are expanded as follows.

-H*path*
Expanded to $*APPL_HOME*/*path*, with *APPL_HOME* the variable that indicates the application's home directory, as set with the linker option -h.

-L*path*
Expanded to *DIR*/*path*, with *DIR* the first directory from the following list, in which the given file can be found.

- Directories in $*APPL_LIBS*, with *APPL_LIBS* the variable that indicates the application's library directory path, as set with the linker option -l.
- Application home library $*APPL_HOME*/lib.
- ***ProLog*** home directory $BIM_PROLOG_DIR/lib.
- Working directory.

3.3 Creating customized development systems

A customized development system is created with **BIMlinker**, using the default mode (contrary to generating a run-time system by using the linker option -r). Typically, the call of **BIMlinker** has the form :

BIMlinker -x -o *exec_file - options targets*

The -x linker option is necessary to enable incremental linking in the created system. The new system is named *exec_file*, as indicated with linker option -o. It will have default op-

tions as given in *options* and when it is started up, the Prolog files *targets* are silently consulted.

A special case is where the error descriptions must be replaced :

BIMlinker -x -o *exec_file* -e *error_file* - *options targets*

Here the -e linker option indicates that the error descriptions are in the (compiled) file *error_file*.

3.4 Creating run-time systems

With **BIMlinker**, it is possible to generate a stand-alone run-time version of any ***ProLog*** application. Such a system can be delivered to an end-user as a package of two files : an executable and a data file. Depending on the application, additional data or program files can accompany the package.

To turn an application into a run-time system, a few simple conventions must be respected (See section *Conventions for run-time systems*).

Before generating a run-time system with **BIMlinker**, all external files and Prolog files must be compiled. It is a good idea to maintain a *makefile* to do this. This can also contain the command to generate the run-time system.

The command below is a typical call of **BIMlinker** to generate a run-time system.

BIMlinker -o *ApplRun* -r *ApplData* -h *ApplHome* -l *ApplLibs* - *options targets*

The executable file is named *ApplRun* and the data file *ApplData*. The environment variable that will indicate the application's home directory is called *ApplHome*, and the one for the library directories *ApplLibs*. The *options* are used as default ***ProLog*** options in the run-time system and the *targets* are linked into the system. External code in these targets is linked into the executable file, and Prolog code is stored in the data file.

Here also, the -e linker option can be used to replace the standard error descriptions.

Both files, *ApplRun* and *ApplData* are generated in the working directory.

3.5 Conventions for run-time systems

Some conditions must be met in order to create a run-time system. The first one concerns the toplevel of the engine. The others refer to the program file organization.

Toplevel condition

A run-time system has no Prolog engine toplevel. The execution of such a system only consists of the execution of a special predicate : **main/0**. As soon as **main/0** terminates, the execution of the run-time system stops.

As a result, the application that is turned into a run-time system, must have a definition for the (global) predicate **main/0**, which starts up the application.

Program file organization

To clarify the discussion about program organization, it is necessary to identify the two different phases in consulting Prolog files and linking external objects files. The following terminology is used :

creation time
: Indicates the moment when the run-time system is created. This is at the application developer's site.

execution time
: This is during execution of the application, which is at the customer's site.

The rules that determine when Prolog code is consulted or when external code is linked are:

- All Prolog files that are given as targets to the ***ProLog*** linker, are consulted at creation time.
- The Prolog files that are consulted at creation time, are saved into the run-time system. They are not needed at execution time.
- The object files and libraries that are linked at creation time (i.e. if they appear in external load directives in files that are given as targets to **BIMlinker**), are saved into the run-time system. They are not needed at execution time.
- The Prolog files that are consulted at execution time, must be delivered to the user in a compiled form (.wic).
- The object files and libraries that are incrementally linked at execution time (i.e. if they appear in external load directives of files that are not given as targets to **BIMlinker**), must be delivered to the user in a compiled form (.o).
- The object files that are incrementally linked at creation time, are not saved in the run-time system and they are not linked at execution time. This should be avoided. This includes the external load directives in files that are consulted from one of the target Prolog files, during creation time.
- Queries in files that are consulted at creation time, are executed at creation time. If they contain consults, these are also done at creation time.
- Queries in files that are consulted at execution time, are ignored.

No query is needed to start the run-time application. At execution time, the global predicate **main/0** is automatically called.

Run-time system organization

The run-time system consists of at least two files, that usually reside in the same directory. It is possible to parameterize the location of that directory in such a way that the customer can decide where he will put the whole system. This is accomplished through the use of an environment variable that contains the path of the application's home directory. The creator of the run-time system decides on the name of that variable.

The application home directory is referred to from the Prolog code, with the -H filename expansion (in consults). This has exactly the same behavior as in the ***ProLog*** development system, where the home directory variable is named BIM_PROLOG_DIR.

As a run-time system does not include a compiler, all additional files must be compiled prior to delivering it to the customer. Only the .wic files must be included.

3.6 Embedded systems

An embedded system is a run-time system in which not ***ProLog*** , but the external program is the master. The *main()* routine must be defined in the external program, and the object file name that contains this routine, must be given to **BIMlinker** with the -m linker option.

In order to enable ***ProLog*** to initialize, the *main()* routine must call the external ***ProLog*** routine

```
BIM_Prolog_initialize ( pargc , argv )
int * pargc;
char * argv [];
```

This routine expects the command line arguments that are passed to the generated system. The number of command line arguments (argc) must be passed by reference. The initialization routine will strip off the command line arguments that it can use.

Calling **BIM_Prolog_initialize**() should happen before doing anything else in the external program (even before printing any messages).

3.7 Examples

Four examples are given : the first one is somewhat the minimal application that could be made. The second one uses an external predicate. Example three uses several external packages (using the XView and Xlib interface). The last one demonstrates an embedded system.

Minimal example

The source program consists of the file Appl1.pro. This file contains a definition for **main/0**:

```
main :-
    write( 'Hello from the Run-Time System.\n' ) .
```

The following commands are issued to generate the run-time system :

```
% BIMpcomp Appl1
% BIMlinker -h APPL1_HOME -r -HAppl1Data -o -HAppl1Run - -TCb0 Appl1
```

The output of this **BIMlinker** call is something like :

```
Warning : Environment variable APPL1_HOME is undefined.
Using system's home directory instead.
Linking target file Appl1.wic
Final linking into Appl1Run
Initializing data for Appl1Run
Warning : Cannot expand -HAppl1Data : unknown system home directory.
Warning : Cannot expand -HAppl1Run : unknown system home directory.
compiled /usr/local/Bprolog/demos/runtime/Appl1/src/Appl1.pro
consulted Appl1.pro
Creating initialized data file Appl1Data.
```

As we did not define the application home directory variable APPL1_HOME, the linker gives a warning. For this application, this variable is not used and therefore we can ignore the warnings.

Then the application can be moved to its destination home directory. Suppose we will place it in ~/Appl1:

```
% mv Appl1Run Appl1Data ~/Appl1
```

If there is a BIMCODE file with a legal code in that directory, we can start the application:

```
% setenv APPL1_HOME ~/Appl1
% $APPL1_HOME/Appl1Run
```

This will produce the message:

```
Hello from the Run-Time System.
```

Small example with external predicate

For this application, the program consists of two source files : Appl2.pro and Appl2.c. The Prolog file contains the following declarations and definitions:

```
:- extern_load ( [ date_time ] , [ 'Appl2.o' , '-lc' ] ) .
:- extern_predicate ( date_time ( string : r ) ) .

go :-
    date_time (_date_time),
    write ( 'Hello from the Run-Time System.\n' ),
    printf ( 'On this system, it is now %s\n' , _date_time ),
    write ( 'Bye.\n' ) .

main :-
    go .
```

The C file has a definition for date_time() :

```
#include <time.h>

char *date_time()
{
    long t;

    t = time(0);
    return( ctime(&t) );
}
```

The run-time system is generated with the following commands:

```
% cc -w -c Appl2.c
% BIMpcomp Appl2
% BIMlinker -h APPL2_HOME -r -HAppl2Data -o -HAppl2Run - -TCb0 Appl2
```

The output of this **BIMlinker** call is similar to:

```
Warning : Environment variable APPL2_HOME is undefined.
Using system's home directory instead.
Linking target file Appl2.wic
Final linking into Appl2Run
Initializing data for Appl2Run
Warning : Cannot expand -HAppl2Data : unknown system home directory.
Warning : Cannot expand -HAppl2Run : unknown system home directory.
compiled /usr/local/Bprolog/demos/runtime/Appl2/src/Appl2.pro
consulted Appl2.pro
Creating initialized data file Appl2Data.
```

As we did not define the application home directory variable APPL2_HOME, the linker gives a warning. For this application, this variable is not used and therefore we can ignore the warnings.

Then the application can be moved to its destination home directory. We will place it in ~/Appl2 :

```
% mv Appl2Run Appl2Data ~/Appl2
```

If there is a BIMCODE file with a legal code in that directory, we can start the application:

```
% $APPL2_HOME/Appl2Run
```

This will say something like:

```
Hello from the Run-Time System.
On this system, it is now Thu Sep 20 10:27:31 1990
Bye.
```

Example using XView

This application is a combination of Prolog code, external C code and libraries. It also illustrates the usage of Prolog libraries at execution time (when the run-time system is executed). It is made of the Prolog files *hanoi.pro*, *fixintro.pro*, *varintro.pro*, *main.pro* and the C file *banner.c*, which includes a bitmap defined in the file *banner.pr*. The *hanoi.pro* file is mainly the hanoi demonstration program. The Prolog file *main.pro* contains the declarations for the external predicate **banner/3** and also for the top-level predicate **main/0**. Note that this one is outside the hanoi module, to keep it global. Also note that there is a query to load the file *hanoi.pro* during creation of the system. The **main/0** predicate calls an introduction predicate that first executes the fixed introduction and the consults the *varintro.pro* file from the application library (using -L). Each of these intro files write a message. As **main/0** is only executed when the run-time system is executed (not when it is created), the *varintro* is just as well only consulted and executed during execution of the generated system. The *fixintro* on the other hand, is given as target to the linker, and therefore it is linked into the generated run-time system. So it does not have to be consulted anymore at execution time. But it is also executed from the **main/0** predicate and therefore not executed at creation time. The query to consult *hanoi.pro*, in the file *main.pro* is executed at creation time because *main.pro* is a target of **BIMlinker** and so, *hanoi.pro*, is linked into the generated system.

The contents of *main.pro* :

```
:- import go/0 from hanoi .

main :-
    intro,
    go$hanoi .

intro :-
    fixintro,
    consult ( '-Lvarintro' ) .

:- module ( hanoi ) .
:- extern_load ( [ banner ] , [ 'banner.o' ] ) .
:- extern_predicate ( banner ( pointer : r , integer : o , integer : o ) ) .

?- consult ( hanoi ) .
```

The C file *banner.c* defines the *banner()* routine:

```
#include <xview/xview.h>
#include <xview/svrimage.h>

#define banner_width 672
#define banner_height 352
#define banner_depth 1
static short banner_bits[] = {
#include "banner.pr"
};

Server_image banner ( width , height )
int * width, * height;
{
    return( xv_create ( XV_NULL , SERVER_IMAGE ,
        XV_WIDTH , banner_width ,
        XV_HEIGHT , banner_height ,
        SERVER_IMAGE_DEPTH , banner_depth ,
        SERVER_IMAGE_BITS , banner_bits ,
        0 ) );
} /* banner */
```

The following commands create the run-time system:

```
% cc -c banner.c
% BIMpcomp hanoi main fixintro varintro
% BIMlinker -Bstatic -h APPL3_HOME -r -HAppl3Data -o -HAppl3Run - \
    -TDb8k -TFb4k main -Lxview -Lxlib fixintro
```

The output of this **BIMlinker** call is similar to:

```
Warning : Environment variable APPL3_HOME is undefined.
Using system's home directory instead.
Linking target file main.wic
Linking target file /usr/local/Bprolog/lib/xview.wic
Linking target file /usr/local/Bprolog/lib/xlib.wic
Linking target file fixintro.wic
Final linking into Appl3Run
Initializing data for Appl3Run
Warning : Cannot expand -HAppl3Data : unknown system home directory.
Warning : Cannot expand -HAppl3Run : unknown system home directory.
compiled /usr/local/Bprolog/demos/runtime/Appl3/src/main.pro
consulted main.pro
executing query
compiled hanoi.pro
consulted hanoi.pro
compiled -w /usr/local/Bprolog/lib/xview.pro
consulted xview.pro
executing query
compiled -w /usr/local/Bprolog/lib/xlib.pro
consulted xlib.pro
```

```
compiled /usr/local/Bprolog/demos/runtime/Appl3/src/fixintro.pro
consulted fixintro.pro
Creating initialized data file Appl3Data.
```

The application can now be moved to its destination home directory (e.g. ~/Appl3):

```
% mv Appl3Run Appl3Data ~/Appl3
% mv varintro.wic ~/Appl3/lib
```

If there is a BIMCODE file with a legal code in that directory, the application is started:

```
% setenv APPL3_HOME ~/Appl3
% $APPL3_HOME/Appl3Run
```

This will give the following output and start the hanoi demo with a banner on its canvas window.

```
Fixed intro to hanoi.
compiled /home/demo/Appl3/lib/varintro.pro
consulted varintro.pro
executing query
Variable intro to hanoi.
```

Embedded system

As a an example of an embedded system, the following program starts in C by printing a message, then calls a Prolog predicate that prints a message from Prolog, and then prints a final message from C. It consists of a Prolog file and a C file. The Prolog file contains the following definition :

```
hello :- write ( 'Hello from ProLog !\n' ) .
```

The C file contains the *main()* routine :

```
#include <BPextern.h>

main ( argc , argv )
int argc;
char * argv[];
{
    BP_Atom name;
    BP_Functor pred;

    BIM_Prolog_initialize ( &argc , &argv );
    printf ( "Calling Prolog\n" );

    name = BIM_Prolog_string_to_atom ( FALSE , "hello" );
    pred = BIM_Prolog_get_predicate ( name , 0 );
    BIM_Prolog_call_predicate ( pred );
```

```
        printf ( "Returned from Prolog\n" );
        exit ( 0 );
    }
```

The run-time system is generated with the following commands:

```
% cc  -c -I$BIM_PROLOG_DIR/include Appl4.c
% BIMpcomp Appl4
% BIMlinker -h APPL4_HOME -r -HAppl4Data -o -HAppl4Run -m Appl4.o - \
    -TCb0 Appl4
```

The output of this **BIMlinker** call is something like :

```
Warning : Environment variable APPL4_HOME is undefined.
Using system's home directory instead.
Linking target file Appl4.wic
Final linking into Appl4Run
Initializing data for Appl4Run
Warning : Cannot expand -HAppl4Data : unknown system home directory.
Warning : Cannot expand -HAppl4Run : unknown system home directory.
compiled /usr/local/Bprolog/demos/runtime/Appl4/src/Appl4.pro
consulted Appl4.pro
Creating initialized data file Appl4Data.
```

As we did not define the application home directory variable APPL4_HOME, the linker gives a warning. For this application, this variable is not used and therefore we can ignore the warnings.

Then the application can be moved to its destination home directory. We will place it in ~/Appl4 :

```
% mv Appl4Run Appl4Data ~/Appl4
```

If there is a BIMCODE file with a legal code in that directory, we can start the application:

```
% $APPL4_HOME/Appl4Run
```

This will say something like:

```
Calling Prolog
Hello from ProLog !
Returned from Prolog
```

SYNTAX

Syntax

Contents

ProLog by BIM - Reference Manual
Syntax
Chapter 1

Syntax

1.1 Types range

Integers:

The range of integers is from -268435456 up to 268435455 (i.e. 29 bits);
Arithmetic on integers is performed modulo 2^{29} .

Default base is 10 but numbers can be represented in any base from 1 to 36.
Notation is: <base>'<number>
If the base is greather than 10, digits greater than 9 are represented by characters a-z or A-Z.
For Example: 16'3a7 stands for the hexadecimal number 3a7.

Reals:

Reals are manipulated in double precision: Approximatively 16 significant decimal digits.
For Example: 5.0 -546.3E23 -0.012e31 78.0e-2

Pointers:

Pointers are coded in 32 bits and are represented by a hexadecimal number beginning with '0x'.
For Example: 0x170056

Atoms:

The length of atoms is restricted to 16384 characters .

Lists:

The length of lists is only restricted by the available memory of the heap.

Functors:

The maximum arity of a functor is 255.

Predicates:

The maximum arity of a predicate is 32.

1.2 Native syntax

Introduction

This section contains the complete ***ProLog*** syntax, in the usual (BNF) format.

Note the following:

- Alternatives within a syntax rule are placed on different lines, or separated by "|".
- Spaces in the rules have no syntactic meaning. They are used merely to enhance readability.
- Lexical entities are separated from each other by spaces or other separators, such as <LF> tabs, etc.
  ab45 is ONE lexical entity,
  whereas the following two strings stand for TWO entities each:
  45ab
  **qw.
- Nonterminals are enclosed between <>.

ProLog syntax rules

<program> ⇒ *((<directive> . <eoln>) |*
(<clause> . eoln>) |
*(<query> . <eoln>))**

<directive> ⇒ *:- <subterm 1199>*

<clause> ⇒ *<head>*
<head> :- <goals>

<head> ⇒ *<term 1199>*
not equal to <integer> or <variable> or <real>

<query> ⇒ *?- <goals>*

<goals> ⇒ *<subterm 1199>*

<subterm N> ⇒ *<term M> where M =< N*

<term N> ⇒ *<op(N,fx)> <subterm N-1>*
<op(N,fy)> <subterm N>
<subterm N-1> <op(N,xfx)> <subterm N-1>

		<subterm N-1> <op(N,xfy)> <subterm N> *<subterm N> <op(N,yfx)> <subterm N-1>* *<subterm N-1> <op(N,xf)>* *<subterm N> <op(N,yf)>*
<term 0>	⇒	*<functor> (<arglist>)* *(<subterm 1200>)* *<constant>* *<variable>* *<list>*
op (N,T)>	⇒	*<name>* A name that has been declared as an operator of type T and precedence N, but not placed within single quotes.
<functor>	⇒	*<name>* Not declared as an operator.
<arglist>	⇒	*<subterm 999>* *<subterm 999> , <arglist>*
<constant>	⇒	*<atom>* *<integer>* *<real>* *<pointer>*
<list>	⇒	*[]* Denotes the empty list. *[<term 1200>]*
<variable>	⇒	*<underscore> <restname>*
<atom>	⇒	*<letter> <alfanum>** *<special sequence>* *' (<char>/''/\')* '* *! / , / ; / []*
<integer>	⇒	*<number>* *- <number>* *<based number>* *- <based number>*
<number>	⇒	*<digit>*

<digit> <number>
Range: the range on your machine in 29 bits

<based number> ⇒ *<base>'<alfanum><alfanum>* /*
where <base> is an integer between 1 and 36.
For example: 16'abc
35'j1
0'(<char>|')
For example: 0'\t

<real> ⇒ *<integer>.*
.<number>
<integer>.<number>
<integer>.<number><exponent>
<integer><exponent>

<pointer> ⇒ *0x (<digit>|a|b|c|d|e|f|A|B|C|D|E|F)**

<exponent> ⇒ *(E | e) (+ | - | <empty>) <number>*

<name> ⇒ *'<string of char>'*
<letter> <restname>
;
!
,
<special sequence>

<special sequence>
⇒ *<special char>*
<special char> <special sequence>

<special char> ⇒ *+ | - | * | / | ^ | < | > | = | ` | ~ | : | . | ? | % | $ | & | @ | \ | # |*

<string of char> ⇒ *<char>*
<char> <string of char>
inside a string of characters, the \ has the same meaning as in a C string

<alfanum> ⇒ *<letter>*
<digit>
<underscore>

<restname> ⇒ *<empty>*
<alfanum><restname>

<char> ⇒ *Any printable character different from ' (single quote).*

If the escape character \ is active, the following escape sequences are recognized and translated to:

\<cr>	*nothing (for a string on several lines)*
\'	*' (single quote)*
**	*\ (backslash)*
\n	*<newline>*
\t	*<tab>*
\b	*<backspace>*
\r	*<return>*
\f	*<formfeed>*
\0xyz	*character with ascii code 0xyz (octal)*

<letter> ⇒ *A | B | ... | Z | a | b | ... | z*

<digit> ⇒ *0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*

<underscore> ⇒ _

<eoln> ⇒ *The end of line character is installation dependent.*

<empty> ⇒

<comment> ⇒ *{ < Any text not containing a } > }*
A comment can be placed anywhere, and can serve as a delimiter

1.3 Dec-10 syntax

Compatibility

The ***ProLog*** default syntax differs from the DEC-10 Prolog syntax in the following ways:

- All variables must start with an underscore.
  For example: _var
  (See the builtin predicate **warn_uppercase/0** and the -c option of the compiler).
- A quoted string is never considered as an operator. Thus, if + is an infix operator, 1 + 2 is a valid term, but, 1 '+' 2 is not.
- All operators should have the correct number of operands:
  write (+) is erroneous !
- One restriction is placed on the types which an operator can have simultaneously: a postfix operator cannot be of any other type.
- [] is treated internally as the atom *nil*. Write ([]) will therefore display *nil* and the goal []=nil succeeds.
- [a] is equivalent to .(a,nil), [a,b] is equivalent to .(a,[b]), [a | b] is equivalent to .(a,b). The equivalent forms are always interchangeable.
- A point . followed by a space, or an end of line character is not necessarily an endpoint.
  For example: succ(_x,_y) :- _y is _x + 1.
  is a clause without an endpoint: 1. is interpreted as the real number 1.0 and not as the integer 1 followed by an endpoint.
- Literals of type *pointer* are notated in hexadecimal form with a leading **0x**.
  For example: **0x**0 stands for the null pointer
  0xabc for a pointer with integer value = 2748

Compatibility with the DEC-10 Prolog syntax is offered when using the option -c of the compiler and/or the run time system, except that the empty list [] and the atom nil still unify.

As an extension explicit module qualification is possible in this syntax (see Modules). Although the DEC-10 syntax does not support the pointer format 0x... , it is possible to create pointers with the builtin predicate **pointertoint/2**.

In compatibility syntax mode, the compiler assumes that all unrecognized directives are queries. As a result, the **:-**/1 operator may be used instead of **?-**/1 to set queries in a file.

1.4 Operators

There are three groups of operators:

1) Infix: xfx, xfy, yfx
2) Prefix: fx, fy
3) Postfix: xf, yf

'x' represents an argument whose precedence must be strictly lower than that of the operator.
'y' represents an argument whose precedence is lower or equal to that of the operator.

One is advised not to change the predefined operators which are list below, especially ',' and '|'.

Name	Type	Precedence	Usage
:-	xfx	1200	clauses ('if')
-->	xfx	1200	definite clause grammmars
:-	fx	1200	directive
?-	fx	1200	queries
;	xfy	1100	clauses ('or')
->	xfy	1050	clauses ('if then else')
dynamic	fx	1050	directive
mode	fx	1050	directive
,	xfy	1000	clauses ('and')
|	xfy	1000	list
\+	fy	900	builtin ('not')
not	fy	900	builtin
module	fx	800	directive
import	fx	800	directive
local	fx	800	directive
global	fx	800	directive
from	xfx	750	directive
=	xfx	700	builtin
is	xfx	700	builtin
=..	xfx	700	builtin
==	xfx	700	builtin
\=	xfx	700	builtin
=:=	xfx	700	builtin
=\=	xfx	700	builtin
?=	xfx	700	builtin
<	xfx	700	builtin
>	xfx	700	builtin

=<	xfx	700	builtin
>=	xfx	700	builtin
<>	xfx	700	builtin
@<	xfx	700	builtin
@>	xfx	700	builtin
@=<	xfx	700	builtin
@>=	xfx	700	builtin
index	xfx	500	directive
+	yfx	500	expressions
-	yfx	500	expressions
/\	yfx	500	expressions
\/	yfx	500	expressions
+	fx	500	expressions
-	fx	500	expressions
*	yfx	400	expressions
/	yfx	400	expressions
//	yfx	400	expressions
<<	yfx	400	expressions
>>	yfx	400	expressions
**	xfx	300	expressions
mod	xfx	300	expressions
^	xfy	200	expressions
÷	xfy	100	directive

For operators declaration, see Directives-Operators and Builtins-Operators.

1.5 DCG's

DCG (Definite Clause Grammars) is a notational extension of Prolog that makes it easy to implement formal grammars in Prolog. Grammar rules are translated into Prolog clauses when they are compiled or consulted under the DEC-10 prolog syntax (i.e. -c option for BIMprolog and BIMpcomp) .

The transformation of Definite Clause Grammar-rules is described by the following prolog program to be read in compatibility syntax.

```
dcg_rule((_nterm, _term --> _dcg_body), (_pro_head :- _pro_body)) :-
        !,
        dcg_nonterminal( _nterm, _pro_head, _S0, _St ),
        append( _term, _S, _St ),
        dcg_body( _dcg_body, _pro_body, _S0, _S ) .

dcg_rule( ( _dcg_head --> _dcg_body ), ( _pro_head :- _pro_body ) ) :-
        dcg_nonterminal( _dcg_head, _pro_head, _S0, _S ),
        dcg_body( _dcg_body, _pro_body, _S0, _S ) .

dcg_body( ( _dcg_b1 ; _dcg_b2 ), ( _pro_b1 ; _pro_b2 ), _S0, _S ) :- !,
        dcg_body( _dcg_b1, _pro_b1, _S0, _S ),
        dcg_body( _dcg_b2, _pro_b2, _S0, _S ) .

dcg_body( ( _dcg_b1, _dcg_b2 ), ( _pro_b1, _pro_b2 ), _S0, _S ) :- !,
        dcg_body( _dcg_b1, _pro_b1, _S0, _St ),
        dcg_body( _dcg_b2, _pro_b2, _St, _S ) .

dcg_body( !, !, _S, _S ) :- ! .

dcg_body( ( { _pro } ), _pro, _S, _S ) :- ! .

dcg_body( [], true, _S, _S ) :- ! .

dcg_body( [ _h | _t ], ( _S0 = _St ), _S0, _S ) :- !,
        append( [ _h | _t ], _S, _St ) .

dcg_body( _nterm, _pro_term, _S0, _S ) :-
        dcg_nonterminal( _nterm, _pro_term, _S0, _S ) .

dcg_nonterminal( _dcg_term, _pro_term, _S0, _S ) :-
        _dcg_term =.. [ _functor | _dcg_arg ],
        append( _dcg_arg, [ _S0, _S ], _pro_arg ),
        _pro_term =.. [ _functor | _pro_arg ] .
```

This example on Definite Clause Grammar rules is extracted from:
W.F.Clocksin and C.S.Mellish: "Programming in Prolog"
Springer-Verlag 1981

Example file:

```
:- compatibility.

?- op(100,xfx,&).
?- op(150,xfy,'->').

:- op(100,xfx,&).
:- op(150,xfy,'->').

sentence(P) --> noun_phrase(X,P1,P),verb_phrase(X,P1).

noun_phrase(X,P1,P) -->
        determiner(X,P2,P1,P),
        noun(X,P3),
        rel_clause(X,P3,P2).
noun_phrase(X,P,P) -->
        proper_noun(X).

verb_phrase(X,P) -->
        trans_verb(X,Y,P1),
        noun_phrase(Y,P1,P).
verb_phrase(X,P) -->
        intrans_verb(X,P).

rel_clause(X,P1,(P1&P2)) -->
        [that], verb_phrase(X,P2).
rel_clause(_,P,P) -->
        [].

determiner(X,P1,P2, all(X,(P1->P2))) -->
        [every].
determiner(X,P1,P2, exists(X,(P1&P2))) -->
        [a].

noun(X,man(X)) -->
        [man].
noun(X,woman(X)) -->
        [woman].
```

```
proper_noun(john) -->
        [john].

trans_verb(X,Y,loves(X,Y)) -->
        [loves].

intrans_verb(X,lives(X)) -->
        [lives].
```

Execution:

```
?- sentence(X,[every,man,loves,a,woman],[]).
X = all(_11,man(_11) -> exists(_26,woman(_26) &
 loves(_11,_26)))
Yes ;
No
```

BUILTIN PREDICATES

Builtin Predicates

Contents

ProLog by BIM - Reference Manual
Builtin Predicates
Chapter 1

Input - Output

1.1 Introduction

Notation

The ***ProLog*** builtin predicates are described as follows :

functor/arity arg1 : ...
.
argn : ...

where "functor" is the functor name, "arity" is the number of arguments, and "argi : ..." is a description of the "i"th argument.

Arguments

For each argument an indication is given about the required or allowed degree of instantiation for the predicate to make sense.

A distinction is between

- *free* the argument must be completely free
- *ground* the argument must be completely instantiated
- *partial* the argument must not be free
- *any* no restrictions on this argument

Possible types can be :

- integer
- real
- pointer
- atom
- atomic (= integer, real, pointer or atom)
- list
- term

1.2 General remarks

Most I/O builtin predicates exist in two variants. The predicates that read from or write to files, have an arity which is 1 higher than the corresponding predicates which act on current streams.

I/O predicates used to read from or write to a logical file:
The logical file must be connected to a 'physical' file by **fopen/3** prior to reading or writing. Closing such a file is done with **close/1**. A file can also be referred to by its file pointer instead of a logical filename. This pointer can be used in external routines. In this way, one can open files in an external routine and write/read this file from ***ProLog*** or external routines can write/read files opened in a Prolog predicate.
Pathnames in physical file names can be given in an abbreviated format (e.g. ~, ~user, $ATOM,...). For more information see also the section 'file name expansion' in Principal Components.

The logical files **stdin**, **stdout** and **stderr** are always open. They must not be closed. They refer to the physical file */dev/tty* (which is the user terminal).

Predicates used to read from or write to the current stream:
The streams are initialised to */dev/tty* (the physical filename). Redirection is performed with **see/1** (for input), and **tell/1** (for output). To know the names of the current streams, use **seeing/1** and **telling/1**. To close the current streams, use **seen/0** and **told/0**. If the current stream is */dev/tty*, these last two predicates succeed, but do not close the streams. ***ProLog*** writes its error messages to the current error output stream which is associated with stderr (physical name /dev/tty). The current error output stream can be redirected with **tell_err/1** and **told_err/0.** The name of the current error output stream can be consulted with **telling_err/1.**

Two different behaviors are available for reading end-of-file. Either, the read fails, or it returns a special value. This behavior can be specified with a please/2 option. When the **readeoffail** option is set, every read builtin that reads past the end of file fails. When this option is unset, the read builtins return a special value, that is specified with another option of please/2. Builtins that read characters return the *end-of-file character*, set with **readeofchar**, which defaults to the integer -1. Builtins that read terms, return the *end-of-file atom*, set with **readeofatom**, which defaults to the atom end_of_file.
See also **please/2**.

UNIX allows the same file to be opened any number of times. This is also possible in ***ProLog***. No warning is given, although it can cause unusual effects.
The maximum number of files which can be opened simultaneously is operating system dependent (30 in SunOS 3.5, 64 in SunOS 4).

1.3 Opening and closing Files

fopen/3

fopen (_LogFileName, _PhysFileName, _Mode)
fopen (_FilePointer, _PhysFileName, _Mode)

arg1 : atom (ground) or pointer (free)
arg2 : ground : atom (has to be a valid physical file name)
arg3 : ground : atom (must be 'w', 'r' or 'a')

fopen/3 opens the file with physical filename *arg2*, for *reading* (r), *writing* (w) or *appending* (a), (depending on *arg3*) and associates to it the logical file name *arg1* or, when free, *arg1* will be instantiated to a file pointer. This logical file name is used in all I/O operations referring to the same physical file. If the file cannot be opened, **fopen/3** fails.

fclose/1

fclose (_LogFileName)
fclose (_FilePointer)

arg1 : ground : atom or pointer

Closes the file specified by *arg1*. Any subsequent I/O operation referring to the same logical file name will fail and will issue a message, except for **fopen/3** - because logical file names can be reused.

close/1

close (_PhysFileName)

arg1 : ground : atom

Closes the file specified by *arg1*. If *arg1* is a current stream, an error message appears.

1.4 Redirection of standard I/O

Standard output

tell/1

tell (_PhysFileName)
tell (_FilePointer)

arg1 : ground : atom or pointer

The current output stream becomes *arg1*.
If *arg1* is an atom, it must be a physical filename. If the file has already been opened as the current output stream, output will be appended to the file. If the indicated physical file has not yet been opened as current output stream, or has been closed again, it is opened for writing.
If *arg1* is a pointer it is assumed to be the file pointer of an open file. Output will be appended to that file.
The previous current output stream is not closed. Failure can occur if the maximum number of open files is reached or if the file is protected.

told/0

Closes the current output stream and sets the new current output stream to **stdout**. If the current output stream was **stdout**, **told/0** succeeds, but does not close **stdout**.

telling/1

telling (_PhysFileName)
telling (_FilePointer)

arg1 : any : atom or pointer

The current output stream is unified with *arg1*. If it was opened with a physical filename, this name is returned. If it was opened with a file pointer, *arg1* is unified with that file pointer. The name of the current output stream for **stdout** is */dev/tty*.

tellingptr/1

tellingptr (_FilePointer)

arg1 : any : pointer

Arg1 is unified with the file pointer of the current output stream.

Standard input

see/1

see (_PhysFileName)
see (_FilePointer)

arg1 : ground : atom or pointer

Analogous to **tell/1** but for the current input stream.

seeing/1

seeing (_PhysFileName)
seeing (_FilePointer)

arg1 : any : atom or pointer

Analogous to **telling/1** but for the current input stream. The name of the current input stream for **stdin** is */dev/tty*.

seeingptr/1

seeingptr (_FilePointer)

arg1 : any : pointer

Arg1 is unified with the file pointer of the current input stream.

seen/0

Closes the current input stream, and sets the new current input stream to **stdin**. If the current input stream was **stdin**, **seen/0** succeeds, but it does not close **stdin**.

Standard error

tell_err/1

tell_err (_PhysFileName)
tell_err (_FilePointer)

arg1 : ground : atom or pointer

All error messages from ***ProLog***, normally written on **stderr**, will be written on the file specified by *arg1*. ***ProLog*** may still write some (very urgent) error messages on **stderr**.
It has an analogous behavior as **tell/1** but for the current error output stream.

told_err/0

Closes the current error output stream and sets the new current error output stream to **stderr**. If the current error output stream was **stderr**, **told_err/0** succeeds, but it does not close **stderr**.

telling_err/1

telling_err (_PhysFileName)
telling_err (_FilePointer)

arg1 : any : atom

Analogous to **telling/1** but for the current error output stream. The name of the current error output stream for **stderr** is */dev/tty*.

telling_errptr/1

telling_errptr (_FilePointer)

arg1 : any : pointer

Arg1 is unified with the file pointer of the current error output stream.

1.5 Reading from files

Reading terms

read/2

read (_LogFileName, _Term)
read (_FilePointer, _Term)

arg1 : ground : atom or pointer
arg2 : any : term

Arg2 is unified with the next term read from the file specified by *arg1*. **read/2** reads characters from the input file until an endpoint is found (i.e. a period ".", followed by an end of line character)or until it encounters an error.

If any error occurs, **read/2** fails and the reason of failure can be tested (see predicate **error_status/3**). On syntax errors, an error recovery is performed : the next endpoint is searched for and the following **read/2** will attempt to read the term that follows. The line producing the error is written to the current error output stream, together with an indication of the position of the error and an error message.

When **read/2** fails because the unification with *arg2* fails a complete term from the input has been read !

read/1

read (_Term)

arg1 : any : term

As read/2, but it reads from the current input stream.

The prompt '@ ' is displayed if the current input stream is a terminal (see also **prompt/1**, **pread/2**, and **pvread/3**). No error recovery is done when the current input stream is a terminal.

When the eof character (^D) is entered in response to **read/1**, any subsequent **read/1** from the terminal will fail (the top level of ***ProLog*** will undo this effect). So, in writing interactive programs, one should be careful to test for an eof or, if possible, ignore it.

It is possible to reset eof with **fseek/1-2** predicates.

vread/3

vread (_LogFileName, _Term, _NameVarList)
vread (_FilePointer, _Term, _NameVarList)

arg1 : ground : atom or pointer
arg2 : any : term
arg3 : free : list

The next term from file *arg1* is read and is unified with *arg2*. List *arg3* contains the names of the variables that appear in the term. This is represented by terms of the form (name = _var). An example is given in **vread/2**.

vread/2

vread (_Term, _NameVarList)

arg1 : any : term
arg2 : free : list

Similar to **vread/3** but from the current input stream.

For example :

```
?- vread (_term, _VarList) .
@ struct (go (_x), 'comments', _d, a (_y)) .
    _term = struct (go (_9), comments, _6, a (_11))
    _VarList = [x = _9, d = _6, y = _11]
Yes
```

pread/3

pread (_LogFileName,_Prompt, _Term)
pread (_FilePointer,_Prompt, _Term)

arg1 : ground : atom or pointer
arg2 : ground : atom
arg3 : any : term

Reads a term from the file specified by *arg1* using as prompt *_arg2*. The term that is read is unified with *arg3*.

pread/2

pread (_Prompt, _Term)

arg1 : ground : atom
arg2 : any

Similar to **pread/3** but from the current input stream.

pvread/3

pvread (_Prompt, _Term, _NameVarList)

arg1 : ground : atom
arg2 : any : term
arg3 : free : list

Similar to **vread/3**, but from the current input stream and using as prompt *_arg1*.

pvread/4

pvread (_LogFileName,_Prompt, _Term, _NameVarList)
pvread (_FilePointer,_Prompt, _Term, _NameVarList)

arg1 : ground : atom or pointer
arg2 : ground : atom
arg3 : any : term
arg4 : free : list

Similar to **pvread/3** but reading is done from the file specified by *arg1*.

sread/2

sread (_Atom, _Term)

arg1 : ground : atom
arg2 : any : term

sread/2 reads a term from the atom specified by *arg1*.

For example :

```
?- sread ( 'f( _x)', _y),functor(_y,_name,_arity) .
   _y = f ( _3)
   _name = f
   _arity = 1
Yes
```

svread/3

svread (_Atom, _Term, _NameVarList)

arg1 : ground : atom
arg2 : any : term
arg3 : free : list

Combination of **sread/2** and **vread/3** : the term is read from atom *arg1*.

readln/2

readln (_File,_Line)

arg1 : ground : atom or pointer
arg2 : any : atom

The next text line is read from the file specified by *arg1* and unified with *arg*2. A text line ends at a newline character or at the end of file. The newline character is discarded.

readln/1

readln (_Line)

arg1 : any : atom

Same as **readln/2** but from the current input stream.

Reading characters

readc/2

readc (_LogFileName, _Char)
readc (_FilePointer, _Char)

arg1 : ground : atom or pointer
arg2 : any : atom (of length one)

Arg2 is unified with the next character on the input file specified by *arg1*.

readc/1

readc (_Char)

arg1 : any : atom (of length one)

As **readc/2** but from the current input stream.

bctr/2

bctr (_LogFileName, _Char)
bctr (_FilePointer, _Char)

arg1 : ground : atom or pointer
arg2 : any : atom (of length one)

Arg2 is instantiated to the next character read from the input file specified by *arg1*. This predicate backtracks until the end of file, and then fails. It is a "backtracking-read".

bctr/1

bctr (_Char)

arg1 : any : atom (of length one)

As **bctr/2**, but from the current input stream.

get0/2

get0 (_LogFileName, _AsciiCode)
get0 (_FilePointer, _AsciiCode)

arg1 : ground : atom or pointer
arg2 : any : integer

Arg2 is unified with the ASCII code of the next character from the file specified by *arg1*.

get0/1

get0 (_AsciiCode)

arg1 : any : integer

As **get0/2**, but from the current input stream.

get/2

get (_LogFileName, _AsciiCode)
get (_FilePointer, _AsciiCode)

arg1 : ground : atom or pointer
arg2 : any : integer

Arg2 is unified with the ASCII code of the next printable character from the file specified by *arg1*. Non-printable characters are simply skipped.

get/1

get (_AsciiCode)

arg1 : any : integer

As **get/2**, but from the current input stream.

Skipping predicates

skip/2

skip (_LogFileName, _AsciiCode)
skip (_FilePointer, _AsciiCode)

arg1 : ground : atom or pointer
arg2 : ground : integer

Skips characters in the file specified by *arg1*, and stops after the first character with ASCII code *arg2*.

skip/1

skip (_AsciiCode)

arg1 : ground : integer

As **skip/2**, but on the current input stream.

1.6 Writing to files

Writing terms

write/2

write (_LogFileName, _Term)
write (_FilePointer, _Term)

arg1 : ground : atom or pointer
arg2 : any : term

Arg2 is written to the file specified by *arg1* - which is a logical filename or a file pointer - using the current operator declarations. Any variables in *arg2* will be printed as a number, prefixed with an underscore. Each variable has a unique number.

For example :

The ***ProLog*** commands

```
?- fopen( outputfile, 'data.pro', w),
   write( outputfile, '9th'), write( outputfile, ' '),
   write( outputfile, Symphony), write( outputfile, '\n'),
   write( outputfile, 'Ludwig von Beethoven'),
   fclose( outputfile).
Yes
```

will create a file 'data.pro' containing the following lines :

```
9th Symphony
Ludwig von Beethoven
```

write/1

write (_Term)

arg1 : any : term

As **write/2**, but on the current output stream.

writeq/2

writeq (_LogFileName, _Term)
writeq (_FilePointer, _Term)

arg1 : ground : atom or pointer
arg2 : any : term

As **write/2**, but atoms that need to be quoted are quoted, and spaces are inserted where appropriate, so that terms written with **writeq/2** can be read with **read/1** or **read/2**.

But :

- The term written is not terminated by a period.
- The same operator declarations should be active when reading, as at the time of writing.
- If illegal atoms (e.g. ones containing control characters) have been constructed by using **name/2** or **atomtolist/2**, an error message will be issued when re-reading again.

For example :

The ***ProLog*** commands (see also the example in **write/2**)

```
?- fopen( outputfile, 'data.pro', w),
   writeq(outputfile,composer('Ludwig von Beethoven')),
   write(outputfile,'.\n'),
   fclose(outputfile).
Yes
```

will create a file 'data.pro' containing the following line :

```
composer('Ludwig von Beethoven').
```

writeq/1

writeq (_Term)

arg1 : any : term

As writeq/2, but to the current output stream.

writem/2

writem (_LogFileName, _Term)
writem (_FilePointer, _Term)

arg1 : ground : atom or pointer
arg2 : any : term

As **write/2**, but all non-global names are written with their explicit module qualification.

writem/1

writem (_Term)

arg1 : any : term

As **writem/2**, but on the current output stream.

display/2

display (_LogFileName, _Term)
display (_FilePointer, _Term)

arg1 : ground : atom or pointer
arg2 : any : term

As **write/2**. The term is written in normal functor form, except for lists, which are written with the bracket notation. Operators are quoted and precede their arguments.

display/1

display (_Term)

arg1 : any : term

As **display/2**, but writing is done to the current output stream.

swrite/2

swrite (_Atom, _Term)

arg1 : free : atom
arg2 : any : term

Arg1 is instantiated with the atom made of term *arg2*.

vwrite/3

vwrite (_LogFileName, _Term, _NameVarList)
vwrite (_FilePointer, _Term, _NameVarList)

arg1 : ground : atom or pointer
arg2 : any : term
arg3 : partial : list of (atom = free)

Term *arg2* is written to the file specified by *arg1*. Variables are written with the names mentioned in list *arg3*. This must be a list of (name = _var) tuples.
It is undefined which of both names will be printed when two variables of the term are unified.

vwrite/2

vwrite (_Term, _NameVarList)

arg1 : any : term
arg2 : partial : list of (atom = _var)

Similar to **vwrite/3** but to the current output stream.

svwrite/3

svwrite (_Atom, _Term, _NameVarList)

arg1 : free : atom
arg2 : any : term
arg3 : partial : list of (atom = _var)

Combination of **swrite/2** and **vwrite/3** : *arg1* is instantiated with the atom made of term *arg2*.

Writing characters

put/2

put (_LogFileName, _AsciiCode)
put (_FilePointer, _AsciiCode)

arg1 : ground : atom or pointer
arg2 : ground : integer

The character with ASCII code *arg2* is written to the file specified by *arg1*.

put/1

put (_AsciiCode)

arg1 : ground : integer

As **put/2**, but to the current output stream.

Formatted write predicates

printf/3

printf (_LogFileName, _Format, _Value)
printf (_FilePointer, _Format, _Value)

arg1 : ground : atom or pointer
arg2 : ground : atom
arg3 : ground : atomic or list of atomic elements

printf/3 writes to the file specified by *arg1*. The format used is specified by *arg2* in much the same way as for the function fprintf of the C language.

The printing specifications of the conversion are :

%d	integer printed in **decimal** notation
%o	integer printed in **octal** notation without sign and leading zero
%x	integer printed in **hexadecimal** notation without sign and leading '0x'
%u	integer printed in **unsigned** decimal notation
%f	real printed in **decimal** notation
%e	real printed in **exponential** notation
%g	real printed in its **shortest** form (**decimal** or **exponential** notation)
%c	character
%s	string

Between the %-sign and the conversion character the user can give more specifications of the conversion.

n	in case of an **integer** and a **string**, *n* is the minimum length of the field
n.m	in case of a **real**, *n* is the minimum length of the field and *m* indicates the number of characters after the decimal point
-	left adjustment
0	zero padding to the left

For example :

```
?- printf( myfile,'5 printed with length 3 : %3d \n', 5).
Yes
```

The elements of *arg3* are taken sequentially from the list as arguments to the format.

For example :

```
?- printf ( 'The total amount is %d %s and %d %s    \n',
                                        [12,'US$',35,cents]) .
The total amount is 12 US$ and 35 cents
Yes
```

Invalid combinations might provoke crashes (just as in C).

printf/2

printf (_Format, _Value)

arg1 : ground : atom
arg2 : ground : atomic or list of atomic elements

As **printf/3**, but the format is *arg1* and writing is done to the current output stream.

sprintf/3

sprintf (_Atom, _Format, _Value)

arg1 : free : atom
arg1 : ground : atom
arg2 : ground : atomic or list of atomic elements

Arg1 is instantiated with the atom made of arg2 and arg3 as specified in printf/3.

nl/1

nl (_LogFileName)

arg1 : ground : atom or pointer

An end-of-line character is written to the file specified by *arg1*.

nl/0

As **nl/1**, but to the current output stream.

spaces/2

spaces (_LogFileName, _Count)
spaces (_FilePointer, _Count)

arg1 : ground : atom or pointer
arg2 : ground : integer

Writes *arg2* spaces (*arg2* >= 0) to the file specified by *arg1*.

spaces/1

spaces (_Count)

arg1 : ground : integer

As **spaces/2**, but to the current output stream.

tab/2

tab (_LogFileName, _Count)
tab (_FilePointer, _Count)

arg1 : ground : atom or pointer
arg2 : ground : integer

Writes *arg2* tab characters (*arg2* >= 0) to the file specified by *arg1*.

tab/1

tab (_Count)

arg1 : ground : integer

As **tab/2** but to the current output stream.

User-defined write predicates

print/2

print (_LogFileName, _Term)
print (_FilePointer, _Term)

arg1 : ground : atom or pointer
arg2 : any : term

Arg2 is written to the file *arg1* in a user defined format. If there exists a definition of **portray/2** and *arg2* is instantiated, this definition is used to write *arg2*. If *arg2* is free or there is no definition for **portray/2**, *arg2* is written with **write/2**.

print/1

print (_Term)

arg1 : any

Same as **print/2**, but to the current output stream, and using **write/1** or **portray/1**.

For example :

```
> ?- print( 'a/1').
a/1
> portray( _x) :- writeq( _x).

> ?- print( 'a/1').
'a/1'
> ?- retractall( portray( _)).

> portray(_x) :- printf( 'printed integer is %3d \n', _x).

> ?- print( 3).
printed integer is  3
```

1.7 Flushing and file pointer positioning

flush/0

The current output stream is flushed.

flush_err/0

The current error output stream is flushed.

flush/1

flush (_LogFileName)
flush (_FilePointer)

arg1 : ground : atom or pointer

The file, specified by *arg1*, is flushed.

ftell/2

ftell (_LogFileName, _FilePosition)
ftell (_FilePointer, _FilePosition)

arg1 : ground : atom or pointer
arg2 : free : integer

Arg2 is unified with the current file pointer position of file *arg1*.

fseek/2

fseek (_LogFileName, _FilePosition)
fseek (_FilePointer, _FilePosition)

arg1 : ground : atom or pointer
arg2 : ground : integer

The file pointer position of file *arg1*, is moved to position *arg2*.

Note that, although the type of the argument *_fileposition* is stated in the manual, programs should not rely on it, since this type may change. In particular, programs should not perform arithmetic on *_fileposition*.
Note this special use : **fseek(0)** resets the filepointer to the beginning of the file.

1.8 Status of file operations

End of file

eof/1

eof (_LogFileName)
eof (_FilePointer)

arg1 : ground : atom or pointer

Succeeds if the end of the file specified by *arg1* is reached. The end of file (EOF) is only reached after an attempt has been made to read past the last character of the file.

eof/0

As **eof/1**, but for the current input stream.

1.9 Miscellaneous predicates

all_open_files/1

all_open_files (_ListOpenFiles)

arg1 : free : list

The list of open files is unified with *arg1*.

For each open file the list contains the physical filename, the logical filename and the mode the file was opened for (r,w,a).

If the file was opened with **see/1** or **tell/1**, a default logical filename is given by the system.
No information is given concerning **stdin**, **stdout** and **stderr**.

For example :

all_open_files/1 could return the list:

```
?- fopen(SRC,'source.pro',w).
Yes
?- fopen(DATA,'data.pro',w).
Yes
?- see(input).
Yes
?- all_open_files(_list).
     _list = [source.pro,SRC,w,data.pro,DATA,w,input,.6,r]
Yes
```

prompt/1

prompt (_ReadPrompt)

arg1 : any : atom

If *arg1* is ground, the prompt to be used by **read/1** and **read/2** becomes *arg1*. If *arg1* is free, it is instantiated to the current prompt (see also **pread/2**, **pvread/3**).
The default prompt is '@ '.

iprompt/1

iprompt (_BIM_Prolog_Prompt)

arg1 : any : atom

If *arg1* is instantiated, the ***ProLog*** prompt is changed to *arg1*. The default prompt in no_query mode is '> '.
If *arg1* is free, it is instantiated to the current ***ProLog*** prompt.

exists/1

exists (_PhysFileName)

arg1 : ground : atom

This succeeds if *arg1* is the physical name of an existing file.

ProLog by BIM - Reference Manual
Builtin Predicates
Chapter 2

General Builtins

2.1 Switches

please/2

please (_OptionName, _Value)
please (_ShortHand, _Value)

arg1 : ground : atom
arg2 : free or ground

Arg1 is the name of an option and *arg2* is its value. If *arg2* is free it is instantiated to the current value of the option. Otherwise the option's value is changed to *arg2*. The following table gives an overview of available options. The value column indicates what kind of value the option can take. For most of them this is either the atom 'on' or 'off'. The default value of an option is underlined.

ae : atomescape	*on/off*	Sets the '\'-escape character active or not.
ed : envdebug	*on/off*	Sets the environment debugger on or off.
em : envmonitor	*on/off*	Sets the environment monitor on or off.
c : compatibility	*on/off*	Controls the parser to accept the DEC-10 Prolog syntax.
d : debugcode	*on/off*	Indicates if the interactive compiler has to generate debug code or non-debug code.
e : eval	*on/off*	Translates in-line evaluation
fr : formatreal	*atom('%.15e')*	The atom specifies the format with which a real is printed out. It can be any format that may be used in a formatted print. For possible formats, see **printf/3**.
q : querymode	*on/off*	In querymode, queries can be entered without having to type the '?-'.
ra : readeofatom	*atom (end_of_file)*	End-of-file atom.
rc : readeofchar	*integer (-1)*	End-of-file ascii-code.
rf : readeoffail	*on/off*	Fail upon reading end-of-file.
s : showsolution	*on/off*	Specifies whether the solution of a query has to be printed out or not. If printed out, the system prompts the user after each solution for a <CR> to abort the query or a ';' to search the next solution.
w : warn	*on/off*	controls output of warnings and error messages

wd : writedepth	*integer (-1)*	Printing of a structured term is limited to *arg1* levels of nesting. All sub-terms of that level are printed as '...'. If *arg1* is negative, the limitation is removed.
wf : writeflush	*on/off*	Determines whether output operations have to be followed by a flush.
wm : writemodule	*on/off*	Determines the indication of module qualifiers in printing.
wp : writeprefix	*on/off*	Determines if operators may be used or have to be written in normal prefix functor form.
wq : writequotes	*on/off*	Determines the usage of quotes in printing. These may be necessary if the printed terms must be readable by ***ProLog***.

See also the command line please options.

2.2 Conversions

Conversion of types

ascii/2

ascii (_Char, _AsciiCode)

arg1 : any : atom (of length one)
arg2 : any : integer (range 0..255)

Arg2 is the ASCII code of *arg1*.

At least one of the arguments must be ground.

For example :

```
?- ascii( a, _x), write( _x).
97
?- ascii( _x, 48), write( _x).
0
?- ascii( '\n', _x), write( _x).
10
```

inttoatom/2

inttoatom (_Integer, _Atom)

arg1 : any : integer
arg2 : any : atom

Arg2 is the atom made with the digits of *arg1*.

At least one of the arguments must be ground.

realtoatom/2

realtoatom (_Real, _Atom)

arg1 : any : real
arg2 : any : atom

Arg2 is the atom made with the digits of *arg1*.

At least one of the arguments must be ground.

pointertoint/2

pointertoint (_Pointer, _Integer)

arg1 : any : pointer
arg2 : any : integer

Arg2 is the integer value for pointer *arg1*. If the value of *arg1* is outside the range of integers, it is truncated.

At least one of the arguments must be ground.

For example :

```
?- pointertoint( _x, 32) .
   _x = 0x20
Yes
```

pointertoatom/2

pointertoatom (_Pointer, _Atom)

arg1 : any : pointer
arg2 : any : atom

Arg2 is the atom representation of the pointer *arg1*. A pointer is represented in hexadecimal, preceded by '0x'. Leading zeros in the hexadecimal representation are omitted in the atom
At least one of the arguments must be ground.

For example :

```
?- pointertoatom( 0x0020, _x) .
   _x = 0x20
Yes
```

atomtolist/2

atomtolist (_Atomic, _ListOfChar)

arg1 : any : atomic
arg2 : any : list

Arg2 is the list built from the characters of *arg1*. So, *arg2* is a list of atoms of length one.
At least one of the arguments must be ground.

For example :

```
?- atomtolist( prolog, _x), atomtolist(_y, _x) .
   _x = [p, r, o, l, o, g]
   _y = prolog
Yes
```

When *arg1* is free, its type (atom or number) will be (partially) determined by the first element of *arg2*. So, if the first element of the list of *arg2* is a numeric character (which ranges between '0' and '9'), it could be a pointer (if the list starts with the elements '0' and 'x'), a real (if the first part of the list is a combination of numeric characters and a dot) or an integer (in the other cases, also when an element appears in the list not belonging to this range).

Due to internal conversions (on integers and reals),**atomtolist/2** is not a one-to-one mapping :

For example :

```
?- atomtolist( _x, [ '0','3']) , atomtolist( _x, ['3']) .
```

Succeeds.

asciilist/2

asciilist (_Atom, _AsciiList)

arg1 : any : atom
arg2 : any : list of integer

Succeeds if *arg1* is the atom composed of the symbols in list *arg2*. These symbols are the ASCII representation of the characters of the atom.

At least one of the arguments must be ground.

name/2

name (_Atomic, _ListOfAsciiCodes)

arg1 : any : atomic
arg2 : any : list

As **atomtolist/2**, except that the second argument is a list of ASCII codes instead of characters.

For example :

```
?- name(Prolog,_List).
  _List = [80,114,111,108,111,103]
Yes
```

Due to internal conversions (on integers and reals),**name/2** is not a one-to-one mapping :

For example :

?- name(_x, [48,51]) , name(_x, [51]) .

Succeeds.

Conversion of terms

=../2

_Term =.. [_Functor \|_ArgList]

arg1 : partial or free
arg2 : any : list

Pronounced "univ".

Arg1 is the term built with the elements of the list *arg2*. The name of the functor of *arg1* is the first element of the list *arg2* and the arguments of the term *arg1* are the remaining elements (if any) of the list *arg2*.

If *arg1* is partially instantiated, there are no restrictions on *arg2*.
If *arg1* is free, *arg2* has to be a nil terminated list from which the first element is an atomic.

If *arg1* is atomic, the corresponding second argument is a list, containing only *arg1*.

For example :

```
?- _term =.. [doc,arg1,35].
   _term = doc(arg1,35)
Yes
?- doc(param,_,23) =.. _list.
   _list = [doc,param,_2,23]
Yes
```

functor/3

functor (_Term, _Functor, _Arity)

arg1 : free or partial
arg2 : any : atom or integer
arg3 : any : integer

Arg1 is the term with functor *arg2*, and arity *arg3*. If *arg1* is free then *arg2* and *arg3* must be instantiated.

For example :

```
?- functor(_term,a,3).
   _term = a(_2,_3,_4)
Yes
?- functor(a(_,_,_),_functorname,_arity).
   _functorname = a
   _arity = 3
Yes
```

Note this special case where *arg2* is an integer :

```
?- functor(1,_functorname,_arity).
   _functorname = 1
   _arity = 0
Yes
```

arg/3

arg (_ArgNumber, _Term, _Arg)

arg1 : ground : integer
arg2 : partial : term
arg3 : any : term

Arg3 is the *arg1*'th argument of the term *arg2*.
If *arg1* <= 0 or *arg1* > arity of *arg2* then the predicate fails.

For example :

```
?- arg(2,a(43,12,76),_arg).
   _arg = 12
Yes
```

numbervars/4

numbervars (_Term, _LowNum, _HighNum, _Atom)

arg1 : any : term
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : atom

Instantiates all variables of *arg1* to a unique atom, constructed according to *arg2*, *arg3* and *arg4*. *Arg4* must be an atom ending on a character.

For example :

```
?- numbervars(a(_w,_x,_y,_z),3,5,ABC).
   _w = ABC3
   _x = ABC4
   _y = ABD3
   _z = ABD4
Yes
```

Let X denote the atom *arg4* without its last character, then the atoms used to do the numbering are constructed as follows :

X + last letter of **arg4 + arg2**
X + last letter of **arg4 + (arg2 + 1)**
.....
X + last letter of **arg4 + (arg3 - 1)**
(X + last letter of **arg4 + 1**) + **arg2**
.....

numbervars/3

numbervars (_Term, _LowNum, _HighNum)

arg1 : any : term
arg2 : ground : integer
arg3 : ground : integer

numbervars/3 is defined as :
numbervars(_x, _y, _z) :- **numbervars**(_x, _y, _z, A).

For example :

```
?- numbervars(a(_w,_x,_y,_z),3,5).
   _w = A3
   _x = A4
   _y = B3
   _z = B4
Yes
```

2.3 Atom manipulation

atomlength/2

atomlength (_Atom, _Length)

arg1 : ground : atom
arg2 : free : integer

The length of *arg1* (number of characters) is unified with *arg2*.

atomconcat/3

atomconcat (_Atom1, _Atom2, _ConcatAtom)

arg1 : any : atomic
arg2 : any : atomic
arg3 : any : atomic

The instantiated arguments can be of type atom, integer, real or pointer. Non-atom arguments are automatically converted to atoms before concatenation.
Arg3 is the concatenation of items *arg1* and *arg2*. At most one of the arguments may be free.

For example :

```
?- atomconcat(atom,_part2,atomconcat).
   _part2 = concat
Yes
```

atomconcat/2

atomconcat (_ListOfAtomics, _ConcatAtom)

arg1 : ground : list of atomics
arg2 : free : atom

The items of *arg1* can be of type atom, integer, real or pointer. Non-atom arguments are automatically converted to atoms before concatenation.
Arg2 is the atom that is constructed by concatenating all items of the list *arg1* in the same order.

For example :

```
?- atomconcat([atom,concat,'/',2],_pred).
   _pred = atomconcat/2
Yes
```

atomconstruct/3

atomconstruct (_Atom, _Repeat, _RepeatAtom)

arg1 : ground : atom
arg2 : ground : integer
arg3 : free : atom

Arg3 is an atom constructed as a sequence of *arg2* times *arg1*.

For example :

```
?- atomconstruct(atom,5,_arg3).
   _arg3 = atomatomatomatomatom
Yes
```

atompart/4

atompart (_Atom, _AtomPart, _StartPos, _Length)

arg1 : ground : atom
arg2 : any : atom
arg3 : any : integer
arg4 : any : integer

Atom *arg2* is a part of atom *arg1*, starting at position *arg3* and with length *arg4*. If *arg2* is free, it is instantiated to the part of *arg1* as specified by *arg3* and *arg4*, which have default values of 1 and the length of *arg1* respectively.
If *arg2* is instantiated and *arg3* is free, *arg3* will be instantiated to the starting position of the first occurrence of *arg2* in *arg1*.

For example :

```
?- atompart ('pattern matching', chi, _start, _length) .
   _start = 12
   _length = 3
Yes
```

atompartsall/3

atompartsall (_Atom, _AtomPart, _StartPos)

arg1 : ground : atom
arg2 : ground : atom
arg3 : free : integer

This is the non-deterministic version of **atompart/4**. It succeeds for each part *arg2* of *arg1*. *Arg2* has to be instantiated and *arg3* will be instantiated to the starting positions of the atom parts (by backtracking).

atomverify/3

atomverify (_Atom, _VerifyAtom, _Position)

arg1 : ground : atom
arg2 : ground : atom
arg3 : free : integer

Atom *arg1* is verified against occurrences of characters in the atom *arg2*. *Arg3* is instantiated to the first position in *arg1* of a character of *arg2*. If no such character of *arg2* appears in *arg1*, *arg3* is instantiated to 0.

For example :

```
?- atomverify ('character', at, _position) .
   _position = 3
Yes
```

atomverify/5

atomverify (_Atom, _VerifyAtom,_Start,_Length,_Position)

arg1 : ground : atom
arg2 : ground : atom
arg3 : any : integer
arg4 : any : integer
arg5 : any : integer

Atom *arg1* is verified against occurrences of characters of atom *arg2*. This verification starts at position *arg3* and goes over a length of *arg4*. The position of the first occurrence found, is unified with *arg5*. If no character from *arg2* can be found in the indicated range of *arg1*, *arg5* is unified with 0.
A negative length *arg4*, indicates backward searching from the starting position *arg3*.
If the start and length arguments are free, they are instantiated to the default values of 1 for the start and the remaining length of atom *arg1* for the length.
If the start is free and the length is negative, the start is instantiated to the rightmost position of *arg1*.
If the length is free and the start is at or beyond the end of *arg1*, the length is instantiated to the negative length of *arg1*.

For example :

```
?- atomverify ('character', a,4,3, _position) .
   _position = 5
Yes
?- atomverify ('character',c,_,-5, _position) .
   _position = 6
Yes
```

lowertoupper/2

lowertoupper (_Lowercase, _Uppercase)

arg1 : any : atom
arg2 : any : atom

If *arg1* is free, it is instantiated to the lower case conversion of *arg2*. If *arg2* is free, it is instantiated to the upper case conversion of *arg1*. One of the arguments must be instantiated and the other one free.

2.4 In-core database manipulation

One is advised to be careful when using the prolog database predicates : it is modifying the code of the program. The result of changing a predicate that is in execution, is unpredictable (=implementational update).

In order to simulate global data, the **record** predicates are more suitable.

Asserting

assert/1

assert (_Clause)

arg1 : partial

Arg1, which must be a valid clause, is added to the ***ProLog*** database.The clause is added as the last clause of the predicate concerned. The predicate must not be a loaded static predicate or a builtin predicate.

For example :

```
?- assert( a( 1)), assert (( go :- a( _x), write( _x), nl)).
   _x = _9
Yes
?- listing .

a(1) .

go :-
   a(_9),
   write(_9),
   nl .
Yes
```

Asserts the fact **a/1** and the predicate **go/0** that calls **a/1**.

assert/2

assert (_Clause, _SeqNr)

arg1 : partial
arg2 : ground : integer

As **assert/1**, except that *arg1* is asserted as the *arg2'nd* definition. If *arg2* is zero or negative or higher than the current number of clauses in the predicate, *arg1* is asserted as the last definition.

vassert/2

vassert (_Clause, _NameVarList)

arg1 : partial
arg2 : list of (atom = _var)

Same as **assert/1** with *arg2* a list of elements of the form (name = _var) that gives the names of the variables in the clause.

For example :

```
?-vassert((a(_x,_y) :- write(a(_y,_x))),[(x=_x),(y=_y)]).
 _x = _14
 _y = _15
Yes
```

vassert/3

vassert (_Clause, _SeqNr, _NameVarList)

arg1 : partial
arg2 : ground : integer
arg3 : list of (atom = _var)

Same as **assert/2** with *arg3* a list of elements of the form (name = _var) that gives the names of the variables in the clause.

asserta/1

asserta (_Clause)

arg1 : partial

Asserts clause *arg1* in the database as the first definition of the corresponding predicate.

assertz/1

assertz (_Clause)

arg1 : partial

Asserts clause *arg1* in the database as the last definition of the corresponding predicate.

Updating

update/1

update (_Clause)

arg1 : partial

If *arg1* is not of the form (_x :- _y) with _x partially instantiated, it is interpreted as (_x :- true).

If no definition exists for the principal functor of _x, then *arg1* is asserted. If one or more definitions exist, they are all retracted and replaced by *arg1*.

For example :

```
?- assert(a(123)),
       assert(a(456)),
       listing(a/1),
       write('---'),
       update(a(321)),
       listing
(a/1).
a(123) .
a(456) .
---
a(321) .

Yes
```

Retracting

retract/1

retract (_Clause)

arg1 : partial

The first clause which is unifiable with *arg1*, is retracted from the ***ProLog*** database. On backtracking, the next clause unifiable with *arg1*, is retracted. If no clause matches, the predicate fails.

If *arg1* is not of the form (_x :- _y) with _x partially instantiated, the first argument is interpreted as (_x :- true).

Only dynamic facts can be retracted with **retract/1**.
Retracting may cause unexpected results, if a clause of a predicate is retracted while this predicate is executing.

For example :

Suppose the internal ***ProLog*** database contains the following facts :

```
fact(one).
fact(two).
fact(three).
```

then :

```
?- fact(_x),retract(fact(_y)).
    _x = one
    _y = one
Yes
```

retract/2

retract (_Clause, _SeqNr)

arg1 : partial
arg2 : any : integer

As **retract/1**, except that *arg2* is the number of the clause. Backtracks if *arg2* was free before the call.

retractall/1

retractall (_ClauseHead)

arg1 : partial

All clauses with heads unifiable with *arg1*, are retracted from the ***ProLog*** database. This predicate always succeeds.

If *arg1* is a static predicate, all definitions of this predicate are retracted. New definitions for the predicate can be loaded. But any pending calls of the predicate may yield unexpected behavior if the new definitions are dynamic instead of static.

It should be noted that a **retractall/1** of a dynamic predicate does not remove any index or mode information. If this is required, **abolish/1** or **abolish/2** should be used.

abolish/2

abolish(_Name,_Arity)

arg1 : ground : atom
arg2 : ground : integer

Retracts all definitions of the predicate with name *arg1* and arity *arg2*. Any index or mode information about the predicate is also removed.

abolish/1

abolish(_Term)

arg1 : partial : term

Retracts all definitions of the predicate with same principal functor as *arg1* . Any index or mode information about the predicate is also removed.

Clause retrieval

All clause retrieval predicates, including retract predicates, behave differently for hidden predicates. They succeed and fail as for visible predicates, but the head and body arguments are not instantiated (just as the **?=/2** predicate checks for unifiability without actually instantiating anything). Any other arguments (i.e. reference and index) are instantiated normally.

clause/2

clause (_ClauseHead, _ClauseBody)

arg1 : partial
arg2 : any

Searches a clause in the ***ProLog*** database, with a head unifiable with *arg1*. Its body is unified with *arg2*. If the retrived clause is a fact, th body is unified with the atom true. By backtracking, the predicate finds all solutions.

For example :

Suppose a fact **a(1)** and a predicate **go/0** which prints out this fact :

```
?- clause(a(_),_body).
   _body = true
Yes
?- clause(go,_body).
   _body = a(_4) , write(a(_4))
Yes
```

The compiler 'flattens' conjunctions and disjunctions in the body of a clause, so **clause/2** and **retract/1** can only retrieve 'flattened' bodies, so the following query fails :

```
?- _body=((b,c),d),
      assert((a:-_body)),
      listing(a/0),
      clause(a,_assertedbody),
      _body = _assertedbody.

a :-
    b,
    c,
    d .
No
```

Some static predicates are transformed (**\+ | not | DCG's | ->**), so **clause/2** may not give what you expect.

clause/3

clause (_ClauseHead, _ClauseBody, _SeqNr)

arg1 : partial
arg2 : any
arg3 : any : integer

As **clause/2**, except that the third argument is the number of the clause. If *arg3* is instantiated at the call, no backtracking is done.

vclause/3

vclause (_ClauseHead, _ClauseBody, _NameVarList)

arg1 : partial
arg2 : any
arg3 : list

Same as **clause/2** with *arg3* a list of elements of the form (name= value) that gives the names of the variables in the clause and their values. The values may get instantiated during the unification of the head in the call of vclause

For example :

```
> a(_x,_y) :- write(a(_y,_x)).

> ?-vclause(a(_u,_v),_b,_vlist).
   _u = _12
   _v = _13
   _b = write(a(_13,_12))
   _vlist = [y = _13,x = _12]
Yes
> ?-vclause(a(1,_v),_b,_vlist).
   _v = _13
   _b = write(a(_13,1))
   _vlist = [y = _13,x = 1]
Yes
> -vclause(a([_v|_w],_v),_b,_vlist).
   _v = _14
   _w = _15
   _b = write(a(_14,[_14 | _15]))
   _vlist = [y = _14,x = [_14 | _15]]
Yes
>
```

vclause/4

vclause (_ClauseHead, _ClauseBody, _SeqNr, _NameVarList)

arg1 : partial
arg2 : any
arg3 : any : integer
arg4 : list of (atom=free)

Same as **clause/3** with *arg4* a list of elements of the form (name= value) like in **vclause/3**.

Referenced clause manipulation

A **clause reference** is an identifying key, associated with the clause. It is unique and invariant for each clause (even after dynamic code garbage collection). Previously asserted clauses (with non-referenced assert or consult) also have an associated reference (which can be determined with rclause).

rassert/2

rassert (_Clause, _ClauseRef)

arg1 : partial : clause
arg2 : free : integer

The clause *arg1* is stored in the database and *arg2* is instantiated to its reference.

rassert/3

rassert (_Clause, _SeqNr, _ClauseRef)

arg1 : partial : clause
arg2 : ground : integer
arg3 : free : integer

The clause *arg1* is asserted as *arg2*'nd clause of the predicate concerned. If *arg2* is zero or negative or higher than the current number of clauses in the predicate, *arg1* will be added as last clause. *Arg3* is instantiated to the clause's reference.

rvassert/4

rvassert (_Clause, _SeqNr, _ClauseRef, _NameVarList)

arg1 : partial : clause
arg2 : ground : integer
arg3 : free : integer
arg4 : free : list of (atom = free)

Same as **rassert/3** with *arg4* a list of the variable names which occur in the clause.

rasserta/2

rasserta (_Clause, _ClauseRef)

arg1 : partial : clause
arg2 : free : integer

The clause *arg1* is asserted as the first clause of the predicate concerned. *Arg2* is instantiated to its reference.

rassertz/2

rassertz (_Clause, _ClauseRef)

arg1 : partial : clause
arg2 : free : integer

The clause *arg1* is asserted as the last clause of the predicate concerned. *Arg2* is instantiated to its reference.

rclause/3

rclause (_ClauseHead, _ClauseBody, _ClauseRef)

arg1 : any : term
arg2 : any : term
arg3 : any : integer

Arg2 is unified with the body of a clause whose head unifies with *arg1*. *Arg3* is instantiated to the reference of this clause. Different solutions can be found by backtracking. If *arg3* is ground, *arg1* and *arg2* are unified with the head an body of the clause that has *arg3* as reference. If *arg3* is free, *arg1* must be partially instantiated.

rclause/4

rclause (_ClauseHead, _ClauseBody, _SeqNr, _ClauseRef)

arg1 : any : term
arg2 : any : term
arg3 : any : integer
arg4 : any : integer

Arg2 is unified with the body of a clause whose head unifies with *arg1*. *Arg3* is unified with the position number of this clause and *arg4* with its reference. Different solutions can be found by backtracking. If *arg3* is ground, the clause with that position number is taken. If *arg4* is ground, the clause with that reference is taken. If *arg4* is free, *arg1* has to be partially instantiated.

rvclause/4

rvclause (_ClauseHead, _ClauseBody, _ClauseRef, _NameVarList)

arg1 : any : term
arg2 : any : term
arg3 : any : integer
arg4 : list of (atom = value)

Same as **rclause/3** with *arg*4 a list of elements of the form (name= value) as in **vclause/3** and *arg*3 the clause reference.

rvclause/5

rvclause (_ClauseHead, _ClauseBody, _SeqNr, _ClauseRef, _NameVarList)

arg1 : any : term
arg2 : any : term
arg3 : any : integer
arg4 : any : integer
arg5 : list of (atom = free)

Same as **rclause/4** with *arg*5 a list of elements of the form (name= value) as in **vclause/4** and *arg*4 the clause reference.

rretract/1

rretract (_ClauseRef)

arg1 : ground : integer

The clause with reference *arg1* is removed from the database.

rretract/2

rretract (_Clause, _ClauseRef)

arg1 : any : clause
arg2 : ground : integer

The clause with reference *arg2* and matching *arg1* is removed from the database.

rretract/3

rretract (_Clause, _SeqNr, _ClauseRef)

arg1 : any : clause
arg2 : any : integer
arg3 : ground : integer

The clause with reference *arg*3 and matching *arg1* is removed from the database. *Arg2* is unified with its position number.

rdefined/1

rdefined(_ClauseRef)

arg1 : ground : integer

Succeeds if *arg1* is a clause reference for an existing clause.

Optimisation

Dynamic predicates can be indexed on one of their arguments. If desired the indexing can be hashed. They can also have mode declarations which will be checked in debug mode.
Index and mode declarations for dynamic predicates can be given with directives (in the same way as for static code) or with the following builtin predicates.

mode/1

mode (_Term)

arg1 : ground : term

The predicate with as functor the principal functor of *arg1*, sets the modes as indicated in the arguments of term *arg1*.

index/2

index (_Name/_Arity, _ArgNb)

arg1 : ground : atom/integer
arg2 : ground : integer or integer/integer

The predicate described by *arg1* (in the form *_Name/_Arity*), is indexed on the argument specified in *arg2*. If *arg2* has the form of argnr/size, the size is taken as the length of the hash table based on the indexed argument. Otherwise there is no hashing.
It is impossible to change the indexing of a predicate.
The default for dynamic predicates is to be indexed on the first argument, regardless of how the dynamic predicate is created (either by consulting a file or by asserting interactively).

Any declaration for a dynamic predicate (**dynamic/1**, **mode/1**, **index/2**) defines the predicate (even if there are no clauses for it). As a result, reconsulting a file with such a declaration in it, will retract all existing clauses defining the predicate.

An indexed dynamic predicate can be rehashed using:

rehash/2

rehash (_Name/_Arity, _TableSize)

arg1 : ground : atom/integer
arg2 : ground : integer

The predicate *arg1* is rehashed with a hash table of size *arg2*.
Any existing hash table is first removed.
The predicate must have an argument indexed (but not necessarily hashed).

When definitions are asserted for an indexed predicate that is being executed, it is not assured that these will also be used on backtracking. Furthermore, there may be a different behavior depending on the argument that is indexed. As the behavior is undefined, it may also change in future releases. New calls will see the modifications.

Test predicates

has_a_definition/1

has_a_definition(_Term)

arg1 : partial : term

Succeeds if the principal functor of arg1 is a predicate with a definition, be it in Prolog, in an external language or in a database. The only difference with current_predicate/2 or predicate_type/2, is that dynamic predicates not necessarily have a definition. If only a mode/1 or index/2 declaration was issued for a dynamic predicate, it will exist as predicate, but without a definition.

Global values

The predicates for managing the internal database exist in a versions with two keys an a version with one key. For predicates with two keys, the second key can be viewed as a domain name. Predicates with one key act on the default domain (default=0).
In the following record predicates if a key is a structured term, the principal functor of this term (functor name/arity) will serve as key.

The predicates **record_push/2** and **record_pop/2** can be used for simulating **global stacks.**
The predicates **recorded_arg/3** and **rerecord_arg/3** are meant for simulating global arrays.

record/3

record (_Key, _DomainKey, _Term)

arg1 : partial : term
arg2 : partial : term
arg3 : any : term

If there is a term in the internal database associated with *arg1* and *arg2* then this call fails. Otherwise a copy of *arg3* is stored in the internal database and associated to those keys.

record/2

record (_Key, _Term)

arg1 : partial : term
arg2 : any : term

This predicate is defined as :
record(_Key, _Term) :- record(_Key, 0, _Term)

rerecord/3

rerecord (_Key, _Domainkey, _Term)

arg1 : partial : term
arg2: partial : term
arg3 : any : term

If there is a term in the internal database associated with *arg1* and *arg2* then it will be erased first. Then a copy of *arg3* is stored in the internal database and associated to those keys.

rerecord/2

rerecord (_Key, _Term)

arg1 : partial : term
arg2 : any : term

This predicate is defined as :
rerecord(_Key, _Term) :- rerecord(_Key, 0, _Term)

recorded/3

recorded (_Key, _DomainKey, _Term)

arg1 : partial : term
arg2 : partial : term
any : term

If there is a term in the internal database associated with *arg1* and *arg2* then this term is unified with *arg3*. Otherwise this call fails.

For example :

```
?- record(key1,dom,(p(_x):-a(_x),b(_x))),recorded(key1,dom,_term).
   _x = _4
   _term = p(_17) :- a(_17) , b(_17)
Yes
?- rerecord(key1,dom,(g(_x):-c(_x),d(_x))),recorded(key1,dom,_term).
   _x = _4
   _term = g(_17) :- c(_17) , d(_17)
Yes
```

recorded/2

recorded (_Key, _Term)

arg1 : partial : term
arg2 : any : term

This predicate is defined as :
recorded(_Key, _Term) :- recorded(_Key, 0, _Term)

erase/2

erase (_Key, _DomainKey)

arg1 : partial : term
arg2: partial : term

Any association with arg1 and arg2 is erased from the internal database.
This predicate always succeeds.

erase/1

erase (_Key)

arg1 : partial : term

This predicate is defined as :
erase(_Key) :- erase(_Key, 0)

erase_all/1

erase_all(_DomainKey)

arg1 : partial : term

All entries in the internal database with second key *arg1* are erased.

erase_all/0

This predicate is defined as :
erase_all :- erase_all(0)

is_a_key/2

is_a_key (_Term1, _Term2)

arg1 : partial : term
arg2: ground : term

Succeeds if the combination of *arg1* and *arg2* is used as key. This means that there is a value associated with them in the internal database.

is_a_key/1

is_a_key (_Term)

arg1 : ground : term

This predicate is defined as:
is_a_key(_Term) :- is_a_key(_Term, 0)

current_key/2

current_key(_Key, _DomainKey)

arg1 : partial : term
arg2 : partial : term

Succeeds for any currently existing key formed by *arg1* and *arg2* . If one or both of the arguments are free, they are instantiated to all existing combinations, one at a time, by backtracking.

current_key/1

current_key(_Key)

arg1 : partial : term

This predicate is defined as:
current_key(_Key) :- current_key(_Key, 0)

The predicates **record_push/2** and **record_pop/2** can be used for simulating **global stacks**.

record_pop/3

record_pop(_Key, _DomainKey, _ListHead)

arg1 : partial : term
arg2 : partial : term
arg3 : free : term

Arg3 is the head of the term associated with *arg1* and *arg2*.
The term associated with those keys is replaced by its tail.

For example :

```
?- record(key1,dom1,[]),
       record_push(key1,dom1,a),
       record_push(key1,dom1,b),
       record_push(key1,dom1,c),
       recorded(key1,dom1,_stack),
       record_pop(key1,dom1,_x),
       record_pop(key1,dom1,_y),
       recorded(key1,dom1,_newstack).
   _stack = [c,b,a]
   _x = c
   _y = b
   _newstack = [a]
  Yes
```

record_pop/2

record_pop(_Key, _ListHead)

arg1 : partial : term
arg2 : free : term

This predicate is defined as:
record_pop(_Key, _Term) :- record_pop(_Key, 0, _term)

record_push/3

record_push (_Key, _DomainKey, _Term)

arg1 : partial : term
arg2 : partial : term
arg3: any : list

The term associated with *arg1* and *arg2* is replaced by a list whose head is *arg3* and whose tail is the previous term.

record_push/2

record_push (_Key, _Term)

arg1 : partial : term
arg2 : any : list

This predicate is defined as:

record_push(_Key, _Term) :- record_push(_Key, 0, _term)

The predicates **recorded_arg/3** and **rerecord_arg/3** are meant for simulating global arrays.

recorded_arg/4

recorded_arg(_SelectList, _Key, _DomainKey, _SelectArg)

arg1 : ground : list of integer
arg2 : partial : term
arg3 : partial : term
arg4: free : term

The index list *arg1* locates *arg4* in the term associated with *arg2* and *arg3*. The first element of the index list is the index in the highest level of the structure determined by *the key*.

For example :

```
?- record(key1,dom1,a(1,b(c(2,_,x),3,4),5)),
       recorded_arg([2,1,3],key1,dom1,_arg).
   _arg = x
Yes
```

Meaning that the third argument of the first argument of the second argument of the term associated with the key, is the atom x.

recorded_arg/3

recorded_arg(_SelectList, _Key, _SelectArg)

arg1 : ground : list of integer
arg2 : partial : term
arg3 : free : term

This predicate is defined as:

recorded_arg(_SelectList, _Key, _Term) :-
recorded_arg(_SelectList, _Key, 0, _Term)

rerecord_arg/4

rerecord_arg(_SelectList, _Key, _DomainKey, _SelectArg)

arg1 : ground : list of integer
arg2 : partial : term
arg3 : partial : term
arg4 : ground : term

Arg4 replaces the term, occuring in the term associated with *arg2* and *arg3*, on the location described by the index list *arg1*. The first element of the list is the index in the highest level of the structure.

rerecord_arg/3

rerecord_arg(_SelectList, _Key, _SelectArg)

arg1 : ground : list of integer
arg2 : partial : term
arg3 : ground : term

This predicate is defined as:

rerecord_arg(_SelectList, _Key, _Term) :-
rerecord_arg(_SelectList, _Key, 0, _Term)

2.5 Program manipulation

Listing of predicates

listing/0

All clauses of the ***ProLog*** database are written to the current output stream. The output is a ***ProLog*** source program without directives. The builtin predicates and the hidden predicates are not listed.

For example :

Interactively :

```
> ?- op( 100, xfx, a) .
> _x a _y :- write(ok) .
> ?- tell( 'file.pro') , listing , told.
> ?- stop .
```

Now the file 'file.pro' contains the clause :

```
_x a _y :- write(ok) .
```

but not the operator declaration. If a list of all operator declarations is wanted, use **all_directives/0**, defined in the next paragraph.

Variables are generally written with their symbolic names : optimisation may slightly alter the code syntax.

Variables in clauses, asserted during a query, using **(assert/1** or **assert/2)** appear as an underscore, followed by a number. To retain the variable names, the **vassert** predicates can be used.

listing/1

listing (_PredName)
listing (_PredName/_Arity)
listing (_PredList)

arg1 : ground

Arg1 must be one of the following forms :

- Atom : the clauses of the predicates with functor name *arg1* are listed on the current output stream.
- Atom/integer : the clauses of the predicate with functor name equal to the atom, and arity equal to the integer, are listed on the current output stream.
- A list of the above two forms : for any member of the list the corresponding **listing/1** instruction is executed.

See also the note on directives in **listing/0**.

flisting/1

flisting (_LogFileName)
flisting (_FilePointer)

arg1 : ground : atom or pointer

flisting/1 is equivalent to **listing/0** but output is sent to the file associated with *arg1*. This file has to be opened before writing to it.

flisting/2

flisting (_LogFileName, _PredName)
flisting (_FilePointer, _PredName)
flisting (_LogFileName,_PredName/_Arity)
flisting (_FilePointer, _PredName/_Arity)
flisting (_LogFileName, _PredList)
flisting (_FilePointer, _PredList)

arg1 : ground : atom or pointer
arg2 : ground (see arg1 of **listing/1***)*

flisting/2 is equivalent to **listing/1**, but the output is sent to the file associated with *arg1*.

mlisting/1

mlisting(_ModuleName)

arg1 : ground : atom

Lists all predicates defined in the module *arg1* on the current output stream.

mlisting/2

mlisting(_LogFileName,_ModuleName)
mlisting(_FilePointer,_ModuleName)

arg1 : ground : atom or pointer
arg2 : ground : atom

equivalent to **mlisting/1** but the output is written to the file specifed by arg1.

Listing directives

all_directives/0

The current operator declarations and dynamic declarations are listed on the current output stream.

all_directives/1

all_directives (_LogFileName)
all_directives (_FilePointer)

arg1 : ground : atom or pointer

The current operator declarations and dynamic declarations are listed on the file *arg1*.

Inquiring atoms, functors and predicates

current_atom/1

current_atom (_Atom)

arg1 : any : atom

Gives all atoms currently in the ***ProLog*** system one at a time by backtracking, or succeeds once, if *arg1* is instantiated to an atom. Lots of the atoms you will get are defined by the system (e.g. functor names of builtin predicates.).

current_predicate/2

current_predicate(_PredName,_PredTerm)

arg1 : any : atom
arg2 : any : term

Succeeds for all currently defined predicates with name *arg1* and most general unifying term *arg2*. If *arg1* and *arg2* are free, this predicate generates the functors off all existing predicates, one at a time by backtracking.

current_op/3

current_op (_Precedence, _Assoc, _Operator)

arg1 : any : integer between 0 and 1200
arg2 : any : atom (one of xfx, xfy, yfx, xf, yf, fx, fy)
arg3 : any : atom

Gives the operators currently in the ***ProLog*** system (*arg3*), and their precedence (*arg*1) and type (*arg*2), one at a time by backtracking.

current_functor/2

current_functor (_FuncName, _FuncTerm)

arg1 : any : atom
arg2 : any : term

Gives the name of the functors currently in the ***ProLog*** system (*arg1*), and the most general term corresponding to it (*arg2*), one at a time by backtracking.

all_functors/1

all_functors (_FuncTerm)

arg1 : any : term

Gives all general terms currently in the ***ProLog*** system one at a time by backtracking, or succeeds once if *arg1* is a correct term.

Consulting files

consult/1

consult (_PhysFileName)

arg1 : ground : atom

Arg1 is a filename possibly preceeded by some compiler options. The file specified in *arg1* is consulted, i.e. it is compiled if necessary and loaded into the system. The filename is expanded following the rules explained with **expand_path/2**. The predicate always succeeds.

Options can be any compiler option. More explanation on these options can be found in - *The Compiler - Options.*

Static procedures in the file should have a name/arity which differs from all the procedures already loaded. Dynamic procedures in the file should have a name/arity which differs from all the static procedures already loaded. If not, a warning is given, and no loading of the procedure causing the problems takes place.

If the file has been compiled with the **-p** option, the operators active at compilation time replace the current active operators.

For example :

?- consult ('-d file').

compiles the file 'file.pro' to debug code and loads it.

?- consult ('-LUnixFileSys')

consults the UnixFileSys predicates library.

The conventions on passing options to the compiler are as follows. We distinguish between different sets of options :

previous: those options that were used for the previous compilation of the file

current: the current options of the system that also apply to the compiler (see below)

specified: the options that are explicitly specified in the consult

The desired options are defined as :

Consult argument	Desired options
'file'	previous options
'- file'	current options of the ***ProLog*** engine
'-x -y ... file'	previous, overriden by specified options
'- -x -y ... file'	current, overriden by specified options

Source files are (re)compiled under the following conditions:

- The intermediate code file (.wic) does not exist.
- The Prolog source file (.pro) is more recent than the corresponding .wic file.
- The specified options are different from the previous options.

For clarity, the system will tell which options are used for the recompilation or which ones were used in the previous compilation.

The following table lists the current ***ProLog*** options that have an equivalent compile-time option (see *'BIMpcomp Options'*).

BIMprolog	BIMpcomp
-Pc	-c
-Pd	-d
-Pe	-e
-Pw	-w
-Pae	-x

A short hand notation exists for consulting files, using the list notation. An example can be found in **reconsult/1**, explained hereafter.

reconsult/1

reconsult (_PhysFileName)

arg1 : ground : atom

Most of the comments in **consult/1** also apply to this predicate. But the definitions in the file erase the existing procedures with the same name/arity in the ***ProLog*** database. This is true for static an dynamic predicates, but not for external predicates and database relations. A static predicate however should be replaced by a new static predicate, otherwise any pending calls of the predicate may yield unexpected behavior if the new definitions are dynamic instead of static.
The predicate always succeeds.

For example:

Suppose 'demo.pro' was not compiled before, and the interpreter is activated with default options :

```
?- [demo].
compiling demo.pro          { Using defaults }
consulted demo.pro
```

```
?- ['-d demo'].
compiling -d demo.pro          { Compile to debug code }
reconsulted demo.pro

?- [-demo].                    { Using previous options }
compiled -d demo.pro           { No recompilation : -d is remembered }
reconsulted demo.pro

?- ['-c demo'].                { Override previous options with -c }
compiling -c -d demo.pro       { Recompile : -d is still remembered }
reconsulted demo.pro

?- ['- demo'].                 { Use current options : defaults }
compiling demo.pro             { Recompilation : no -d, no -c }
reconsulted demo.pro
```

Hiding

hide/1

hide (_Term)

arg1 : partial : term

The predicate defined by the principal functor of *arg1* is hidden. This means that **listing/0** (and all its variants) will no longer display the definitions (if any) of this predicate. **Clause/2** and its variants will succeed but the variables will not be instantiated. All builtin predicates are hidden.
The predicate always succeeds.

2.6 Operators

op/3

op (_Precedence, _Assoc, _Operator)

arg1 : any : integer between 0 and 1200.
arg2 : ground : atom (one of xfx, xfy, yfx, xf, yf, fx, fy)
arg3 : ground : atom or list of atoms

If *arg1* is instantiated, the atom *arg3*, or all atoms of the list *arg3* are made an operator with precedence *arg1* and type *arg2*. The scope of this operator declaration is the current interactive session.
If *arg1* is instantiated to 0, the operator definition with name *arg3* and type *arg2* is removed (if it existed).
If *arg1* is free, it is instantiated to the precedence of the operator with name *arg3* and type *arg2* if such an operator exists, otherwise it fails.
To declare an operator in a file, it is not sufficient to establish the operator interactively, use the directive **op/3** .

A conflicting operator declaration overrides the previous declaration.

It is possible to override builtin operators, or to have two operators with the same name, provided that one is binary (xfy, yfx, xfx) and the other unary (fx, fy, yf, xf).

The list of predefined operators can be found in the *Syntax* part of this manual.

2.7 Modules

This paragraph contains the builtin predicates related to modules. More information can be found in the *Module* part of this manual.

module/1

module (_ModName)

arg1 : any : atom

If instantiated, the current module becomes *arg1*. If free, *arg1* will be instantiated to the name of the current module.

module/2

module (_Predicate, _ModName)

arg1 : partial : not integer, real or pointer
arg2 : any : atom

Unifies *arg2* with the module qualification of the principal functor of *arg1*.

module/3

module (_QualTerm, _ModName, _Term)

arg1 : any
arg2 : any : atom
arg3 : any

Arg3 is the term constructed from *arg1* by stripping the module qualification from the principal functor of *arg1*, and unifying this qualification with *arg2*. If *arg1* is free, *arg2* must be an atom and *arg3* must be partially instantiated.

mod_unif/2

mod_unif (_Term1, _Term2)

arg1 : any
arg2 : any

Unifies the 2 arguments, as if they had no module qualification at any level (i.e as if they belonged to the global module).

2.8 Table manipulation

A full description of the different tables and the possible manipulations can be found in *The Engine - Table Options*. The next section explains the builtin predicate **table/2** which has three possible usages.

table/2

table (_OptionId , _OptionValue)

arg1 : ground : atom
arg2 : any : atom

Arg1 is the name of an option and *arg2* is its value. If *arg2* is free, it is instantiated to the current value of the option. Otherwise the value of the option is set to *arg2*.

Possible options and values are:

w : warn	on/off
t : time	integer value

If *arg2* is free, it will instantiated to the time spent in garbage collection and table expansion since the last reset.

If *arg2* is instantiated, it resets the time counter for garbage collection and table expansion to the value *arg2*.

table (_TableId , _TableCommand)

arg1 : ground : atom
arg2 : ground : atom

This usage is for executing table commands. *Arg1* is the table identifier and *arg2* is the command. The indicated command is performed on table *arg1* if possible.

Currently, there is only one possible command: collect. This invokes garbage collection and table expansion if necessary and possible

table (_TableId , _ParamSettings)

arg1 : ground : atom
arg2 : ground : atom
ground : list of atom
any : list of integer or real (of length 4)

With this usage, the table parameters can be set at run-time or their values can be retrieved
Arg1 specifies the table.
If arg2 is an atom, it defines the value of one parameter of table *arg1*. If *arg2* is a list of atoms, it contains parameter settings for table arg1.
Arg2 can be a partially instantiated list of integers or reals. In this case it is a positional list of length 4. The instantiated elements define a new value for the corresponding parameter of table *arg1*. The free elements will be instantiated to the current value of the corresponding parameter of table *arg1*.

The correspondence between the position in the list and the defined parameter is as follows:

Position	*Parameter*
1	base
2	expansion
3	treshold
4	limit

If *arg2* is free, it will be instantiated to a list of 4 elements, containing the current parameter values of table *arg1*.

2.9 Test predicates

Mode

var/1

var (_Term)

arg1 : any : term

Succeeds if *arg1* is a free variable.

nonvar/1

nonvar (_Term)

arg1 : any : term

Succeeds if *arg1* is (partially) instantiated.

ground/1

ground (_Term)

arg1 : any : term

Succeeds if *arg1* is completely instantiated.

Term type

atom/1

atom (_Term)

arg1 : any : term

Succeeds if *arg1* is an atom.

integer/1

integer (_Term)

arg1 : any : term

Succeeds if *arg1* is an integer.

real/1

real (_Term)

arg1 : any : term

Succeeds if *arg1* is a real.

pointer/1

pointer (_Term)

arg1 : any : term

Succeeds if *arg1* is a pointer.

number/1

number (_Term)

arg1 : any : term

Succeeds if *arg1* is either a real or an integer.

atomic/1

atomic (_Term)

arg1 : any : term

Succeeds if *arg1* is a real, an integer, an atom or a pointer.

term_type/2

term_type(_Term,_Type)

arg1 : any : term
arg2 : any : atom

The type of term *arg1* is unified with *arg2*. The type is described with one of the following names :

Term	*Type*
Free variable	var
Integer	integer
Real	real
Pointer	pointer
Atom	atom
Functor	functor

Functor type

static_functor/1

static_functor (_Term)

arg1 : partial : term

Succeeds if *arg1* is a static predicate.

dynamic_functor/1

dynamic_functor (_Term)

arg1 : partial : term

Succeeds if *arg1* is a dynamic predicate.

database_functor/1

database_functor (_Term)

arg1 : partial : term

Succeeds if *arg1* is a relation in the currently open external database.

external_functor/1

external_functor (_Term)

arg1 : partial : term

Succeeds if *arg1* is an external predicate.

hidden_functor/1

hidden_functor (_Term)

arg1 : partial : term

Succeeds if *arg1* is a hidden predicate.

builtin/1

builtin (_Term)

arg1 : partial : term

Succeeds if *arg1* is a builtin predicate.

Predicate type

predicate_type/2

predicate_type(_Term,_Type)

arg1 : partial : term
arg2 : any : atom

The type of term *arg1*, whose principal functor is a predicate, is unified with *arg2*. This predicate fails if *arg1*'s functor is not a predicate. The type is described with one of the following names :

Term	Type
Builtin predicate	builtin
Static predicate	static
Dynamic predicate	dynamic
Database predicate	database
External predicate	external

2.10 Evaluation of expressions

An expression can be evaluated by calling the builtin **is/2**, or it can be evaluated in-line with the evaluation builtin **?/1**. Any appearance of a **?/1** term is replaced by the value that is the result of evaluating the argument. This is accomplished by a compile time source translation. To avoid this translation, the compiler option -e can be used to disable it.

For example:

The program below writes the numbers from 10 to 1.

```
> count_down( 0 ) :- ! .
> count_down( _Count ) :-
      write( _Count ), count_down( ? ( _Count - 1 ) ) .
> ?- count_down( 10 ) .
```

The second clause is equivalent to the transformed clause :

```
> count_down( _Count ) :-
      write( _Count ),
      _Count1 is _Count - 1, count_down( _Count1 ) .
```

An in-line evaluation may also appear in the head of a clause.

Assignment

is/2

_Result is _AritExpr

arg1 : any : real or integer
arg2 : ground : term

The term *arg2* is evaluated and the result (integer or real) is unified with *arg1*.

Arg2 must be a term built with one of the following functors:

functor	arity	function
+	**2**	addition
-	**2**	subtraction
*	**2**	multiplication
/	**2**	division (if one of the operands is real, the result is real, else the result is an integer)
//	**2**	integer division (if the operands are real, they are converted to integer before the division is attempted; the result is an integer)
mod	**2**	modulo operator
**	**2**	exponentiation
^	**2**	exponentiation
trunc	**1**	truncation
round	**1**	rounding
real	**1**	convert integer to real
abs	**1**	absolute value
sign	**1**	sign
+	**1**	unary +
-	**1**	unary -
\	**1**	bitwise not (complement)
/\	**2**	bitwise and (conjunction)
\/	**2**	bitwise or (disjunction)
>>	**2**	bitwise right shift
<<	**2**	bitwise left shift

Also the following mathematical functions can be used inside an arithmetic expression:

cos/1	**exp/1**
acos/1	**pow/2**
sin/1	**sqrt/1**
asin/1	**log10/1**
tan/1	**log/1**
atan/1	
atan2/2	

Note that

- real arithmetic is performed using in double precision.
- modulo and bitwise operations only succeed on integer arguments. If the operations are called with non-integer arguments a warning is displayed.
- **trunc/1** truncates a positive real to the largest integer smaller than or equal to it. Moreover, **trunc**(-f) = - **trunc**(f). If i is an integer, **trunc**(i) = i.
- **round**(r) = **trunc**(r + 0.5) and **round**(-r) = - **round**(r) for positive r.
- overflow is not detected.
- division by 0 is detected and a warning is displayed.

For example:

```
?- _x is 1.5 + sqrt( 1.0 + exp( 2)),
      _r is sqrt( _x**2 + 3**2) .
   _x = 4.396386731590008e+00
   _r = 5.322425790342282e+00
Yes
```

The builtin **is/2** can be defined in Prolog as:

? _X is _X .
To clarify this, let's rename this predicate to IS :
IS(? (_X) , _X) .
Which is equivalent to :
IS(_Y , _X) :- _Y is _X .

Moreover, by using the external language interface one can perform the arithmetic operations in another language (C, Pascal, Fortran,...) and use the available libraries.

Pointer arithmetic

pointeroffset/3

pointeroffset(_Pointer1,_Offset,_Pointer2)

arg1 : any : pointer
arg2 : any : integer
arg3 : any : pointer

Pointer *arg1*, adjusted with offset *arg2*, is pointer *arg3*. At most one of the arguments may be free. The offset can be positive or negative.

Random generator

A random generator is available: it generates integers in the range 0 to 2^{28}-1.

srandom/1

srandom (_SeedInt)

arg1 : ground : integer

Arg1 is the seed for the generator.

random/1

random (_RandInt)

arg1 : free : integer

Arg1 is the output of the random generator.

For example :

```
?- srandom(175677098),random(_x).
     _x = 226629092
Yes
```

2.11 Comparison

Equality

==/2

_Term1 == _Term2

arg1 : any : term
arg2 : any : term

This succeeds if *arg1* is identical to *arg2*.

For this comparison to succeed even any variables within the terms must be identical. It is therefore a 'stronger' test than unification

For example :

```
?- _x=5, _y=5, _y==_x.
   _x = 5
   _y = 5
Yes
?- _x=a, _y==_x.
No
?- _x=5, _y=5.0, _x==_y.
No
```

\==/2

_Term1 \== _Term2

arg1 : any : term
arg2 : any : term

Equivalent to \+(==/2), where \+/1 is explained in the section 3.8 - Metalevel.

For example :

```
?- _x\==5 .
   _x = _0
Yes
?- 5\==5 .
No
?- 5\==5.0 .
Yes
```

Unification

?=/2

_Term1 ?= _Term2

arg1 : any : term
arg2 : any : term

Succeeds if *arg1* and *arg2* are unifiable, but it does not unify them.

For example :

```
?- 5 ?= 5 .
Yes
?- 5 ?= 76 .
No
?- _x ?= 5 .
   _x = _0
Yes
```

=/2

_Term1 = _Term2

arg1 : any : term
arg2 : any : term

Unifies *arg1* with *arg2* if possible, otherwise fails.

For example :

```
?- _x = p(4,b) .
   _x = p(4,b)
Yes
?- p(_x,_y) = p(4,b) .
   _x = 4
   _y = b
Yes
?- p(4,a ) = p(4,b) .
No
```

\=/2

_Term1 \= _Term2

arg1 : any : term
arg2 : any : term

Succeeds if *arg1* and *arg2* are not unifiable.

occur/2

occur (_Term1, _Term2)

arg1 : any : term
arg2 : any : term

Unifies *arg1* with *arg2* if the unification does not create infinite terms, otherwise fails.

This predicate implements 'sound' unification. The predicate is very useful in avoiding problems with infinite terms.

For example :

Trying to unify ...
?- [_x|_t] = [a,b|_t].

... loops infinitely, while occur/2 simply fails ...

?- occur([_x|_t],[a,b|_t]).
No

occurs/2

occurs (_Part, _Term)

arg1 : any : atomic
arg2 : any : term

Succeeds if *arg1* occurs in *arg2*, else fails.

For example :

?- occurs(1,[6,2,4,1,7]).
Yes
?- occurs(c,g(b(d,a))).
No

Arithmetic comparison

The arguments of the builtin predicates below must be ground, they must be arithmetic expressions built according to the rules mentioned in **is/2**.
Atom comparison is not supported by these predicates; use standard order comparison instead.

_AritExp1 *operator* ***_AritExp2***

arg1 : ground : term
arg2 : ground : term

Operator	*Checks if*
>/2	The evaluation of *arg1* is *greater than* the evaluation of *arg2*
</2	The evaluation of *arg1* is *less than* the evaluation of *arg2*
>=/2	The evaluation of *arg1* is *greater than or equal to* the evaluation of *arg2*.
=</2	The evaluation of *arg1* is *less than or equal to* the evaluation of *arg2*.
<>/2	The evaluation of *arg1* is *different from* the evaluation of *arg2*.
=\=/2	The evaluation of *arg1* is *different from* the evaluation of *arg2*.
=:=/2	The evaluation of *arg1* is *equal to* the evaluation of *arg2*.

Standard order comparison

There is no restriction on the arguments of the builtin predicates below. They refer to the so called 'standard order of terms'.

Terms are ordered according to the following criteria:

- Variables @< atoms @< numbers @< pointers @< terms
- Variables are ordered according to their age (roughly).
- Atoms are ordered alphabetically.
- Numbers are ordered numerically.
- Pointers are ordered numerically.
- Terms are ordered according to their arity.
- If the arities are equal, they are ordered according to their name
- If name and arity are equal, they are ordered recursively according to their arguments, from left to right.

Term1** @Operator**Term2

arg1 : any : term
arg2 : any : term

@Operator	*Action*
@</2	Checks if *arg1* is *less than arg2*.
@>/2	Checks if *arg1* is *greater than arg2*.
@=</2	Checks if *arg1* is *less than or equal to arg2*.
@>=/2	Checks if *arg1* is *greater than or equal to arg2*.

For example :

```
?-  _a @< _b,
    _b @< atom,
    atom @< aton,
    aton @< 12,
    12 @< 23.2,
    23.4 @< func(_a),
    func(_a) @< func(f(a)),
    func(f(a)) @< func(f(12)),
    func(f(12)) @< func(_,_).
    _a = _9
    _b = _10
Yes
```

Sorting

sort/2

sort(_List,_SortedList)

arg1 : ground : list of term
arg2 : free : list of term

The list *arg1* is sorted in ascending standard order, and any doubles are removed. The resulting list is unified with *arg2*.

For example :

```
?-  sort( [ 2 , abc , f(3) , 2 ] , _X ) .
_X = [abc,2,f(3)]
Yes
```

keysort/2

keysort(_List,_SortedList)

arg1 : ground : list of (term-term)
arg2 : free : list of (term-term)

List *arg1* should contain elements of the form (_Key-_Value), where both _Key and _Value may be terms. It is sorted in ascending standard order on the _Key parts. The resulting list is unified with *arg2*. Duplicates are not removed. Elements with the same key remain in the same order as in the original list.

For example :

```
?- keysort( [ k3-val3 , k1-val1 , k3-val3b , k2-val2 ] , _X ) .
_X = [k1 - val1,k2 -val2,k3 - val3,k3 - val3b]
Yes
```

keysort/3

keysort(_List,_KeyPred,_SortedList)

arg1 : ground : list of (term-term)
arg2 : ground : atom
arg3 : free : list of (term-term)

The predicate with name *arg2* and arity 2 is used to retrieve the key part of each element in list *arg1*. This list *arg1* is sorted in ascending standard order on the keys that are returned by the key predicate. The resulting list is unified with *arg3*. Duplicates are not removed.

The key predicate *arg2* is specified as

_KeyPred/2

_KeyPred(_KeyValueTerm,_Key)

arg1 : ground : term
arg2 : free : term

Arg2 is unified with the key part of key/value term *arg1*.

For example :

```
> data( [ table(Smith,John,London),
          table(VanHalen,John,NewYork),
          table(Smith,Emmy,Brussel) ] ) .
> getkey( table(_Key1,_Key2,_Value) , _Key1-_Key2 ) .
> ?- data( _Data ), keysort( _Data , getkey , _X ).
   _X = [table(Smith,Emmy,Brussel),table(Smith,John,London),
    table(VanHalen,John,NewYork)]
Yes
```

The data is sorted, first on the first argument of **table/2**, and for equal first arguments, next on the second argument. This is determined by the predicate **getkey/2**.

keysort/4

keysort(_List,_Template,_Key,_SortedList)

arg1 : ground : list of (term-term)
arg2 : partial : term
arg3 : partial : term
arg4 : free : list of (term-term)

List *arg1* should consist of elements of the form *arg2* (i.e. each element of the list must be unifiable with *arg2*). The key part of such an element is given by *arg3*. The list *arg1* is sorted in ascending standard order on the keys that are determined by the template and key. The resulting list is unified with *arg2*. Duplicates are not removed.

For example :

```
> data( [tale(Smith,John,London), table(VanHalen,John,NewYork),
      table(Smith,Emmy,Brussel)] ) .
> ?- data( _Data ), keysort( _Data , table(_Key1,_Key2,_) ,
      _Key1-_Key2 , _X ).
_X = [table(Smith,Emmy,Brussel), table(Smith,John,London),
 table(VanHalen,John,NewYork)]
Yes
>
```

The data is sorted, first on the first argument of table/2, and for equal first arguments, next on the second argument.

keysort/2 is described in terms of **keysort/4** as:

```
keysort( _List , _SortedList ) :-
      keysort( _List , _Key - _Value , _Key , _SortedList ) .
```

2.12 Metalevel

All solutions predicates

findall/3

findall (_Collect, _Goal, _SolutionList)

arg1 : any
arg2 : partial : term
arg3 : any : list

Arg3 is a list of all instances of *arg1* for which *arg2* succeeds. Instances are ordered as they were entered in the ***ProLog*** database, including duplicates. An empty list is returned when no solution is found.

For example :

Suppose the ***ProLog*** database contains the following facts :

```
a(3,2,h) .
a(7,3,b) .
a(2,5,i) .
a(3,3,h) .
a(9,4,m) .
```

Then,

```
?- findall(point(_x,_y),a(_x,_y,h),_list).
   _x = _12
   _y = _13
   _list = [point(3,2),point(3,3)]
Yes
```

bagof/3

bagof (_Collect, _Goal, _SolutionList)
bagof (_Collect, _Exist ^_Goal, _SolutionList)

arg1 : any
arg2 : partial : term
arg3 : any : list

Arg3 is a list of all instances of *arg1* for which *arg2* holds. Instances are ordered as they were entered in the ***ProLog*** database, including duplicates. There is backtracking on the values of all the variables occuring in *arg2* and not in *arg1*, (even on free variables). This predicate fails if *arg2* is not properly instantiated, and

also when no solutions exist.
In order to avoid backtracking, the following notation for *arg2* may be used: _a ^ _b
In this case, no backtracking will occur on the values of the variables occuring in _a.
This is called **existential quantification**

For example :

Suppose the ***ProLog*** database contains the following facts :

```
a(5,2,h) .
a(7,3,b) .
a(2,5,i) .
a(3,3,h) .
a(9,4,m) .
```

Then,

```
?- bagof([_x,_y],a(_x,_y,_z),_list).

{Give me the list _list [_x,_y] for each _z for which a(_x,_y,_z) succeeds}

    _x = _11
    _y = _13
    _z = h
    _list = [[5,2],[3,3]]
Yes ;
    _x = _11
    _y = _13
    _z = b
    _list = [[7,3]]
Yes ;
    _x = _11
    _y = _13
  _z = i
  _list = [[2,5]]
Yes ;
    _x = _11
    _y = _13
    _z = m
    _list = [[9,4]]
Yes ;
No
```

```
?- bagof([_x,_y],_z^a(_x,_y,_z),_list).
{Give me the list _list of [_x,_y] such that there exists  a _z for which
a(_x,_y,_z) succeeds}
    _x = _9
    _y = _11
    _z = _14
    _list = [[5,2],[7,3],[2,5],[3,3],[9,4]]
Yes
```

Since in the notation _a ^ _b, _a may be any term, the above goal is equivalent to:

```
?- bagof([_x,_y], f(_z)^a(_x, _y,_z), _list) .
```

setof/3

setof (_Collect, _Goal, _SortedSolutionList)
setof (_Collect, _Exist ^_Goal, _SortedSolutionList)

arg1 : any
arg2 : partial : term
arg3 : any : list

As **bagof/3**, except that duplicates are discarded and the output list is ordered according to the standard order comparison @< (See section 3.7).

Metacall

call/1

call (_Goal)

arg1 : partial

call(_x) is equivalent to the metacall _x. A **!/0** inside a metacall cuts away backtrackpoints up to the parent of the clause in which the metacall appears. Fails if _x
is free.

Negation

\+/1

\+ *_Goal*

arg1 : partial

This succeeds if the principal functor of *arg1* is not provable, otherwise it fails. No check is made to see whether *arg1* is ground or not.

not/1

not _Goal

arg1 : ground

As **\+/1** but checks whether *arg1* is completely instantiated. If the argument is ground, the execution of the query terminates immediately and control returns to the toplevel of the ***ProLog*** engine.

2.13 Execution control

Logical

fail/0

Always fails.

true/0

Always succeeds.

Mark and cut

!/0

Discards all choice points made since the parent goal started execution.

mark/1

mark (_CutMark)

arg1 : free : integer

Sets a marker and instantiates *arg1* to it.

cut/1

cut (_CutMark)

arg1 : ground : integer

The call to **cut/1** cuts away all choicepoints created since **mark/1** with the same argument. The argument of **cut/1** should be the same as the argument of **mark/1** at the same level, i.e. within the same clause definition.The absence of a specification of what happens in other cases should discourage the undisciplined use of **mark/1** and **cut/1**. More than one cut can refer to the same mark. Although the type of _CutMark is stated in the manual, programs should not rely on it, since it may change. In particular, programs should not perform arithmetic on _CutMark

For example :

```
a :- mark( _x), b, c, cut( _x).
```

The call to **mark/1** instantiates _x to an integer whose value is irrelevant to the prolog programmer. **cut/1** uses this integer to cut away the choicepoints created by b and c.

Control builtins for catch & throw

block/3

block(_Goal,_Catcher,_Recovery)

arg1 : partial : term
arg2 : partial : term
arg3 : partial : term

The goal *arg1* is executed. If this goal succeeds, then the call of **block/3** succeeds. If it fails, this call also fails.

If during the execution of the goal *arg1*, **exit_block/1** is called, the execution control is changed. An implicit cut and fail is performed up to the call of **block/3**. Then the ball argument of the **exit_block/1** call is unified with the catcher *arg2*. If this unification succeeds, the recovery handler *arg3* is called, and its success or failure determines the success or failure the **block/3** call.

If the unification of ball and catcher does not succeed, another cut and fail is performed up to the next, older invocation of **block/3**. An error occurs if there is no active call of **block/3** whose catcher matches against the ball of the **exit_block/1**.

exit_block/1

exit_block(_Ball)

arg1 : partial : term

Terminates execution of the most enclosing **block/3** call whose catcher argument matches the ball *arg1*.

See also **block/3**

For example :

The catch & throw predicates are useful for error handling. The following call installs an application specific error handler for executing _Goal :

```
block( _Goal , app_error(_ErrNr,_Data) ,
    app_err_handler(_ErrNr,_Data) )
```

To raise an error during execution of this goal :

```
exit_block( app_error(123,_ErrData) )
```

The error number (123) and some specific error data is passed to the error handler via the ball-catcher link. The application error handler will be called with this data.

Condition

->/2

_IfGoal -> _ThenGoal
_IfGoal -> _ThenGoal; _ElseGoal

The specification of **if then else** in *ProLog* is :

```
_IfGoal -> _ThenGoal :- _IfGoal, !, _ThenGoal.

( _IfGoal -> _ThenGoal); _ElseGoal :- _IfGoal, !, _ThenGoal.
( _IfGoal -> _ThenGoal); _ElseGoal :- _ElseGoal.
```

A clause containing an **if then else** construct is transformed in the following way :

```
g1 -> g2; g3
```

becomes

```
mark( _mark0), ( g1, cut( _mark0), g2; g3)
```

and

```
g1 -> g2
```

becomes

```
mark( _mark1), ( g1, cut( _mark1), g2)
```

For example :

```
?- assert(test(a)).
Yes
?- ( test( a) -> write( ok); write( nok) ).
okYes
?- ( test( b) -> write( ok); write( nok) ).
nokYes
```

Loop

repeat/0

This backtrackable predicate always succeeds. It is defined as :

```
repeat.
repeat:-repeat.
```

Exit from query

exit/0

Stops the current query, and returns to the toplevel of the ***ProLog*** engine.

abort/0

The same as **exit/0.**

2.14 System control

Exit from ProLog

halt/0

Ends a ***ProLog*** session.

stop/0

The same as **halt/0**.

Saved state

save/1

save(_PhysFileName)

arg1 : ground : atom

Saves the current state of the session. All predicates, except the external ones, are saved. Saving should only be performed when no external database is open.
A saved state can be restored by invoking the engine with the special option -r followed by the filename *arg1*.

Information predicates

information/2

information(_Key,_Identification)

arg1 : ground : atom
arg2 : free : list of atoms

The information data, associated with key *arg1* is returned in the list *arg2*. Possible keys for *arg1* are :

Arg1	*Arg2*
version	[machine,os,release number,release date]
copyright	[copyright notice]
owner	[coordinates BIM]
distributor	[coordinates of your local distributor]

information/1

information(_Key)

arg1 : ground : atom

The information data, associated with key *arg1* is printed on stderr.

information/0

All information data is printed on stderr.

Statistics

statistics/3

statistics (all, _, _)
statistics (_Option, _TotalSize, _Used)

arg1 : ground : atom
arg2 : free : integer
arg3 : free : integer

Arg2 and *arg3* are instantiated to the total size and used size of the table indicated by *arg1*.
Possible tables are:

arg1	*table*
H	heap
S	local stack
I	interpreted code
C	compiled code
D	constant data
F	functors
Sp	permanent strings
St	temporary strings
B	backup heap
R	record keys

If *arg1* is instantiated to the atom **all**, the statistics of all the tables is printed on the current output stream. *Arg2* and *arg3* remain uninstantiated.

statistics/0

This predicate is defined as:

statistics :- statistics(all,_,_).

UNIX system calls

shell/1

shell (_ShellCommand)

arg1 : ground : atom

Arg1 is a shell command and it is executed in a new shell. The new shell is of the same type as indicated in the environment in which ***ProLog*** is running.

For example :

```
?- shell(date).
Fri Dec 31 23:59:59 MET 1999
Yes
```

system/1

system (_ShellCommand)

arg1 : ground : atom

The same as **shell/1**.

sh/0

Equivalent to **shell**('sh').

csh/0

Equivalent to **shell**('csh').

For example :

```
?- csh.
csh % date
Fri Dec 31 23:59:59 MET 1999
csh % ^D
Yes
```

getenv/2

getenv(_Var,_Value)

arg1 : ground : atom
arg2 : free : atom

arg2 is instantiated to the value of the environment variable with name *arg1*.

For example :

```
?- getenv(HOME,_val)
_val = /prolog/tests
Yes
```

expand_path/2

expand_path(_Path,_ExpPath)

arg1 : ground : atom
arg2 : any : atom

The path *arg1* is expanded and unified with *arg2*.
The following meta symbols are recognized. They may only be used at the beginning of the path.

Symbol	Expansion
~	Home directory of the user ($HOME)
~user	Home directory of 'user'
$VAR	Value of VAR in environment
-Llibraryfile	The first library directory in which the given filename is found. Library directories are searched in the following order: 1. All directories given in the library path environment variable ($BIM_PROLOG_LIB). In the same order as given in that variable. 2. The system library directory ($BIM_PROLOG_DIR/lib/). 3. The working directory
-Hfile	The file will be looked for in the ***ProLog*** home directory ($BIM_PROLOG_DIR)

Time predicates

cputime/1

cputime (_CpuTimeUntilNow)

arg1 : free : real

Arg1 is instantiated to the number of seconds of user CPU time of the BIMprolog process used since ***ProLog*** was started.
The accuracy of **cputime/1** is operating system dependent.

time/1

time (_Goal)

arg1 : partial

Arg1 is considered as a goal and is executed. The time in seconds taken to execute the input goal is written on the current output stream.

time/2

time (_Goal, _CpuTimeTillSuccess)

arg1 : partial
arg2 : any : real

As **time/1**, but *arg2* is instantiated to the time needed to execute *arg1*.

Command level arguments

argc/1

argc(_Number)

arg1 : any : integer

Succeeds with *arg1* the number of command line arguments. Only user-defined arguments, i.e. arguments after the delimitor (the '-' sign between blanks), are counted.

argv/1

argv(_ArgList)

arg1 : free : list of atoms

Succeeds with *arg1* the list of command line arguments. Only arguments after the delimitor (the '-' sign between blanks), are counted.

The arguments appear in the list in the same order as on the command line. They are all passed literally as atoms.

argv/2

argv(_Index,_ArgValue)

arg1 : ground : integer
arg2 : any : atom

Succeeds with *arg2* the *arg1*'th command line argument. Only arguments after the delimitor (the '-' sign between blanks), are counted. The argument is passed literally as an atom.

For example :

```
csh % BIMprolog - Useroption1 Useroptionf2
ProLog by BIM - release 3.0 Sun4 01-Nov-1990
?- argv(1,_ArgValue).
_ArgValue = Useroption1
Yes
```

2.15 Signal handling

The signal handling mechanism of ***ProLog*** provides the user with easy-to-use tools to catch and handle UNIX signals. This is achieved by installing signal handler predicates for the signals of interest.
Whenever a signal occurs, the query being executed is interrupted at the next call port. The corresponding handler predicate is then executed and after a solution is found, the interrupted query is resumed. If there are other solutions left for the handler predicate, they will be found on backtracking. If the handler has no solutions at all, the interrupted query is resumed with a failure (causing backtracking).
When the handler predicate is called, the signals of the same type are suspended until the control is explicitely reset to accept status. The supported signals are the Unix ones:: SIGINT, SIGSEGV, SIGPIPE, ...

A number of signals have a default handler installed. This handler prints out a message of the form *** SIGNAL *** SIGname and terminates the current query.

The following predicates can be used to install handlers.

install_prolog_handler/2

install_prolog_handler (_Signal, _PredicateName)

arg1 : ground : atom
arg2 : ground : atom

The predicate with name *arg2* and arity 2 is installed as handler for signal *arg1*.

Such a signal handler predicate has the following specification:

signal_handler_predicate/2

signal_handler_predicate (_Signal, _Goal)

arg1 : ground : atom
arg2 : partial : term

When the signal for which this signal handler is installed, is caught, this predicate will be called with *arg1* instantiated to the signal name, and with *arg2* the goal which was in execution.

install_external_handler/2

install_external_handler (_Signal, _Pointer)

arg1 : ground : atom
arg2 : ground : pointer

The external routine with address *arg2* is installed as handler for signal *arg1*.

which_prolog_handler/2

which_prolog_handler (_Signal, _PredicateName)

arg1 : ground : atom
arg2 : any : atom

Arg2 is unified with the name of the current prolog handler predicate for signal *arg1*. The predicate fails if no prolog handler exists.

which_external_handler/2

which_external_handler (_Signal, _Pointer)

arg1 : ground : atom
arg2 : any : pointer

Arg2 is unified with the address of the current external handler routine for signal *arg1*. The predicate fails if no external handler exists.

Some predicates exist to control the signal delivery.

signal/2

signal (_Signal, _Atom)
signal (_Signal, _Term)

arg1 : ground : atom
arg2 : partial : atom or compound term

The handling of signals of type *arg1* is controlled, depending on the value of *arg2*:

accept
pending or new signals will be handled as soon as possible

ignore
subsequent signals are ignored

suspend
subsequent signals are suspended until they are accepted again

raise
a signal is generated

clear
pending signals are cleared

status/2
status (_State, _NbOfSignals)
arg1 : free : atom
arg2 : free : integer

Arg1 is instantiated to the state of the signal handling, being **accept, ignore suspend, raise** or **clear.**
Arg2 is instantiated to the number of pending signals.

wait/0

This predicate waits until a signal occurs and then succeeds.

wait/1

wait (_Signal)

arg1 : ground : atom or list of atoms

This predicate waits until a signal from *arg1* occurs and then succeeds.

If *arg1* is the empty list, this predicate behaves as **wait/0**.

toplevel/1

toplevel (_PredicateName)

arg1 : ground : atom

The current query is terminated immediately and the toplevel query
?- _PredicateName is started.
One application of this predicate is to call it at the end of a signal handler, when the suspended query must not be resumed.

2.16 Error handling

error_message/2

error_message(_Messages,_Switch)

arg1 : ground : integer or list of integers
arg2 : any : atom : on/off

The error messages specified in *arg1* are switched on or off as indicated by *arg2*. In case an error occurs for which the message is switched off, it is treated as in warning off mode : no message is printed out.

Messages can be specified with an integer number for a single message, or with a list of integers indicating several single messages, or integer pairs of the form *_x-_y*, specifying the range of messages from *_x* to *_y* (inclusive).

The switch argument *arg2* may be free for single integer *arg1*. In this case, it will be instantiated to the current switch status of the specified message.

error_ status/3

error_status (_ErrClass, _ErrNumber, _InfoList)

arg1 : free : atom
arg2 : free : integer
arg3 : free : list

If there is an error pending, it is reported. *Arg1* is the error class (such as OVERFLOW, WARNING, ...). *Arg2* is the number of the error. Any additional parameters for the error are given in *arg3*. This depends on the type of error.
After this inquiry, the error status is cleared. If no error occured since the previous status inquiry, the predicate fails.

error_print/0

The latest error is printed out, regardless of the warn switch. If no error messages have been issued previously on will be generated.

error_raise/3

error_raise(_ErrClass,_ErrNumber,_InfoList)

arg1 : ground : atom
arg2 : ground : integer
arg3 : ground : list

An error with number *arg2* and of class *arg1* is issued, with *arg3* as argument information list.
The error class name (*arg1*) must be one from the table below.

The argument information list (*arg3*) must contain all the arguments that appear in the message. These can be integers, atoms or functors (in the form of a term). For the BUILTIN class errors, the first argument of that list must be the name of the builtin, in the form name/arity.

The error messages can be customized by means of an error description file. This file gives a description of which error messages must be handled and how they must be rendered. At the installation of ***ProLog***, such a description file must be provided, for linking into the BIMprolog executable. At run-time, any program can modify the error handling by incrementally loading a description file. The default error description file is:

$BIM_PROLOG_DIR/install/errors.pro

By instructing the ***ProLog*** linker (BIMlinker) to use another error description file, the error handling can be customized at installation time. To modify it at run-time, the builtin **error_load/1** must be used.

error_load/1

error_load(_ErrorFile)

arg1 : ground : atom

The error description file *arg1* is consulted. It replaces the previous error descriptions. Such an error description file contains facts for the following predefined predicates:

err_class/4

err_class(_ClassNr,_ClassId,_ClassText,_ClassMessage)

arg1 : ground : integer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : atom

The error class with number *arg1* gets as internal identifier *arg2*, and as text for rendering *arg3*. A global class message is given as *arg4*.

The class number *arg1* must be in the range from 1 to 16. It is only used by the system to uniquely identify the error classes. The internal identifier is used in the builtins **error_status/3** and **error_raise/3** to identify the error class. The error class text is printed in error messages between the *** markers. The global class message follows the second *** marker, preceeding the specific error message.

The global message may refer to the error arguments with %i, for the i'th argument.

The system predefined error classes are:

Index	*Name*
1	SYNTAX
2	SEMANTIC
3	COMPOVFL
4	WARNING
5	RUNTIME
6	BUILTIN
7	OVERFLOW
8	MODE

These classes should not be redefined, except for the corresponding text. Redefining the class identifier may break programs that use the error handling builtins.

err_ordinal/2

err_ordinal(_Ordinal,_OrdinalText)

arg1 : ground : integer
arg2 : ground : atom

This predicate gives the textual representation *arg2* of ordinal number *arg1*. The text *arg2* is used in an error message when referring to the *arg1*'th argument.

The ordinal must be in the range from 1 to 5. And there should be a definition for each of these five ordinals.

err_msg_range/2

err_msg_range(_ErrorNumberFrom,_ErrorNumberTo)

arg1 : ground : integer
arg2 : ground : integer

A range of error messages is specified. The first number of the range is *arg1* and the last is *arg2*. Any existing error message ranges that overlap with this range, are discarded.

The range minimum must be greater than 0, and its maximum must be greater than the minimum.

All defined error messages must be in an existing range, except for the special error message with number 0.

The system predefined error message range is from 100 to 999.

At most 255 different ranges may be defined.

err_msg/2

err_msg(_ErrorNumber,_ErrorMessage)

arg1 : ground : integer : 100..2000
arg2 : ground : atom

The message *arg2* corresponds to the error with number *arg1*.
The error number must be in one of the defined error message ranges (see **err_msg_range/2**), except if it is 0. This special message is a catchall : it is used for each error that has no associated message defined.

The text may refer to the error arguments with %i, for the i'th argument.

DIRECTIVES

Directives

Contents

ProLog by BIM - Reference Manual
Directives
Chapter 1

Directives

Introduction

Directives are parser or compiler commands which influence the way clauses are parsed and translated. Definitions of predicates can be mixed with directives, but this complicates later understanding of how the program executes, especially when module directives are used. (See *Modules*). Most of the directives mentioned in this chapter have an equivalent builtin predicate.

A directive has the following form

:- <body> .

It looks like a clause without a head.

General remark : the arguments of directives have to be ground. If one of the arguments of a directive is incorrect, either a warning is given, or an error message appears, and the directive is ignored.

1.1 General directive

option/1

arg1 : ground : atom or list of atoms

The option or option list *arg1* is set. Each option setting is formed by a letter, indicating the option and possibly a + or - to set the option value. If this is omitted, the previous value is toggled.

The corresponding directives are still supported for compatibility.

Option	Default	Description	Directives
a	-	Generate dynamic code	alldynamic
c	-	Parse in DEC-10 syntax	[no]compatibility
d	-	Generate debug code	set[no]debug
e	+	Translate in-line evaluation	-
h	-	hide predicates	[no]hide
l	-	Generate listing file	-
p	-	Include operator definitions	-
w	+	Give warning messages	-
x	+	Allow \ as atom escape sign	-

For example:

:- option('d+').

1.2 Dynamic and static

dynamic/1

arg1 : ground : atom/integer

The predicate with name and arity specified by *arg1* is compiled as a dynamic predicate. By default, predicates are compiled to static code. The dynamic/1 directive is defined as a prefix operator.

Remark

If the dynamic/1 directive appears after a module declaration, then a **local/1** directive for *arg1* is implied. Such dynamic/1 declaration is useful only for predicates defined in the same module and can be applied to both global and local predicates.

For example:

```
:- module(one) .
:- dynamic a/4 .
:- global b/2 .
:- dynamic b/2 .
```

This defines a local dynamic predicate a/4 and a global dynamic predicate b/2 .

alldynamic/0

All predicates in the file following the alldynamic/0 directive are dynamic.

1.3 Debugging

For the following two directives we refer to *Debugger*.

setdebug/0

All predicates in the file following the setdebug/0 directive and before any setnodebug/0 directive are translated into debug code.

setnodebug/0

All predicates in the file following the setnodebug/0 directive are translated into no_debug code.

1.4 Compatibility

compatibility/0

The rest of the file following the compatibility/0 directive will be parsed in the DEC-10 Prolog syntax.

warn_uppercase/0

A warning is issued for each atom which starts with an uppercase letter. This can be used to translate sources coming from other Prolog environments where an uppercase stands for a variable.

1.5 Hiding

hide/0

All predicates in the file following the hide/0 directive and before any nohide/0 directive are hidden. This influences the builtin predicates listing and clause.

nohide/0

All predicates in the file following the nohide/0 directive are visible. This influences the builtin predicates listing and clause.

1.6 File inclusion

include/1

arg1 : ground : atom

The file is "included" textually in the source file by the compiler. (i.e one file is composed by inserting the contents of file arg1 at the place of the directive. The whole file is submitted as a unit to the compiler). As a result, any directives in the included file are also active in the second part of the original source file.

Pathname expansion on *arg1* is done automatically (~, ~<user>, $<variable>,-L,-H).

1.7 Optimisation

mode/1

arg1 : ground : term

With this directive, one can indicate the intended input-output use of a particular predicate. The directive is defined as a prefix operator.

There are 3 possible modes for an argument :

i	(+)	**the argument is ground when the predicate is called**
o	(-)	**the argument is free when the predicate is called**
?	(?)	**anything else**

The symbols between parentheses can be used in compatibility mode ('-c' option).

For example :

:- mode append(i, i, o) .

declares that append will only be called with its first two arguments completely instantiated and with the third argument free. The compiler uses this information to generate more efficient code for static and dynamic predicates.

Remarks

1. Errors in mode declarations can lead to failing goals or (exceptionally) to machine exception, i.e. segmentation faults.

2. If the mode/1 directive appears after a module declaration, a **local/1** directive for *arg1* is implied. Such mode/1 declaration is useful only for predicates defined in the same module and can be applied to both global and local predicates.

For example:

```
:-module(one) .
:-mode a/4 .
:-global b/2 .
:-mode b/2 .
```

This defines a local predicate a/4 and a global predicate b/2 .

index/2

arg1 : ground : atom/integer
arg2 : ground

The compiler uses this directive to generate special indexing instructions on the argument(s) *arg2* of the predicate with name and arity specified by *arg1*. The directive is defined as a infix operator.

Static predicates
Arg2 must have the form : integer or (integer, integer, ...)
At most 3 arguments can be specified.

The order in which arguments are used for indexing is the same as given by the user in the index declaration, except that 'i' mode arguments are indexed on before '?' mode arguments and 'o' mode arguments are never indexed on.

The default indexing is on the first 3 arguments.

Dynamic predicates
Arg2 must have the form : integer or integer1/integer2.
When specified, integer2 is taken as the length of the hash table.
Otherwise there is no hashing.
At most 1 argument can be indexed.
By default, the compiler will index dynamic predicates on their first argument.

Remarks
1. If an index declaration is given for a predicate, the compiler will never index on arguments that are not specified. But the compiler will not always index on the arguments that are specified. For instance, an index put on an output argument - as specified by a mode declaration for the same predicate - is meaningless.

For example:

```
:- append/3 index ( 3, 1) .
:- mode ( append ( i, ?, o)) .
```

with the usual definition of **append/3**.
This results in indexing on argument 1 only : argument 3 is an output argument and so, it is not a reasonable candidate for indexing.

2. To avoid the indexing completely, specify 0 as *arg2*.

3. If the index/1 directive appears after a module declaration then a **local/1** directive for *arg1* is implied. Such index/1 declaration is useful only for predicates defined in the same module and can be applied to both global and local predicates.

For example:

```
:-module(one) .
:-a/4 index (2,3) .
:-global b/2 .
:-b/2 index (1,2) .
```

This defines a local predicate a/4 and a global predicate b/2 .

1.8 Operators

op/3

arg1 : ground : integer between 0 and 1200.
arg2 : ground : atom (one of xfx, xfy, yfx, xf, yf, fx, fy)
arg3 : ground : atom or list of atoms

The atom *arg3* or the list of atoms *arg3* are made operators with precedence *arg1* and type *arg2*. The directive is only active in the rest of the file.
If *arg1* is 0, the operator definitions for *arg3* with type *arg2* are omitted.

If the directive appears after a module declaration then a **local/1** directive for *arg3* is implied.

The full list of the predefined operators is mentioned in *Syntax*.

For example

without explicit module qualification:

```
:-module(mod) .
:-op(100,yfx,plus) .
_x plus _y :- _res is _x + _y, write(_res) .
a(_x,_y) :- _x plus _y .
```

with explicit module qualification:

```
:-module(mod) .
:-op(100,yfx,plus) .
_x plus$mod _y :- _res is _x + _y, write(_res) .
a$mod(_x,_y) :- _x plus$mod _y .
```

1.9 Modules

The directives on modules are mentioned below. More explanation and some examples can be found in *Modules*.

module/1

arg1 : ground : atom

The module name is set to *arg1*.

local/1

arg1 : ground : atom : name/arity

The local/1 directive is defined as a prefix operator and is used to declare arg1 as a local functor.

global/1

arg1 : ground : atom : name/arity

The global/1 directive is defined as a prefix operator and is used to define *arg1* as a global functor.

import/1

arg1 : ground : term

The import/1 directive is defined as a prefix operator and is used to enable the use, inside one module, of predicate *arg1* of another module. *Arg1* is declared with the predefined from/2 infix operator.

(This page intentionally left blank.)

MODULES

Modules

Contents

ProLog by BIM - Reference Manual
Modules
Chapter 1

Modules

1.1 Introduction

This chapter concerns the use of ***ProLog*** modules.

ProLog supports a module concept that makes it easy and natural to split a program in separate components - called modules - which interact only through a set of predicates the programmer has specified. In doing so, the programmer avoids inadvertent name clashes and misuse of code, hides implementation details and enhances the readability and maintainability of the programs.

The ***ProLog*** module concept is flat, static and name/arity based, i.e. modules cannot be nested, the meaning of names is determined at compile-time and functors with the same name, but different arity, can belong to different modules.

As the meaning of names is determined at compile-time, the handling of terms with module qualification incurs no run-time overhead. The use of modules does not influence efficiency at all.

It is possible to program without modules, and programs already written without the use of modules will remain executable without any adjustment.

1.2 Directives

module/1

:- module *(_Modulename)* **.**

arg1 : ground : atom

The module name is set to *arg1*. The predicates declared after the module/1 directive belong to the module *arg1* (i.e. they are local to the module). The predicates textually before the directive are global (they belong to the global module whose name is '', the atom of length zero). Predicates defined in a file without a module/1 directive, are global. There can be, at most, one module/1 directive in a file.

For example:

```
...
global predicates
...
:-module(mod) .
...
predicates local to mod
...
```

The same module/1 directive can appear in two different files. Since compilation is on a per file basis, the locality of functors in one file cannot be influenced by the other file. This principle is never violated.

local/1

*:- **local** _Name / _Arity .*

arg1 : ground : term

The local directive only makes sense when defined after a module directive. The directive is used:

- to declare that *arg*1 belongs to the module : e.g. useful if there is no predicate definition in the file and definitions of the predicate could be asserted at runtime.
- to implement **data hiding**. The local/1 directive for *arg1* prevents *arg1* from being unified with functors with the same name/arity but belonging to a different module.
- to redefine *arg1* locally as a builtin predicate. For this purpose, the local/1 directive is even necessary! A warning is given by the compiler stating that you are redefining a builtin predicate (unless you use the '-w' flag).

Remark: the implied local directive
The directives **op/3, index/2, mode/1** and **dynamic/1** influence the module qualification of the functors they declare. If they occur after the **module**/1 directive a local/1 directive for that functor is implied. In that case op(_prec,xfx,atom) must be considered as using the functor atom(_,_) and op(_prec,fx,atom) as using the functor atom(_) .

For example:

version without emq	version with emq
:-module(mod) .	**:-module(mod) .**
:-local g/1 .	**:-local g/1 .**
a(_l) :- bagof(_x,g(_x),_l) .	**a$mod(_l) :- bagof(_x,g$mod(_x),_l) .**

Compare with:

:-module(mod) .	**:-module(mod) .**
a(_l) :- bagof(_x,g(_x),_l) .	**a$mod(_l) :- bagof(_x,g$(_x),_l) .**

Both columns are identical: the right column uses explicit module qualification (emq).

global/1

*:- **global** _Name / _Arity .*

arg1 : ground : term

The global/1 directive is used to define *arg1* as a global functor. It is only useful when specified textually after a module/1 directive.

For example:

```
:-module(mod) .          :-module(mod) .
:-global a/1 .           :-global a/1 .
a(_x) :- b(_x) .         a$(_x) :- b$mod(_x) .
b(17) .                  b$mod(17) .
```

import/1

*:- **import** _Name / _Arity **from** _ModuleName .*

arg1 : ground : term

The import/1 directive is used to enable the use inside one module of predicates from other modules. *Arg1* is the functor that will be imported from *arg2*.

For example:

version without emq

```
:-module(mod) .
:-import g/1 from bar .
a(_x) :- g(_x) .
```

version with emq

```
:-module(mod) .
:-import g/1 from bar .
a$mod(_x) :- g$bar(_x) .
```

Compare with:

```
:-module(mod) .          :-module(mod) .
a(_x) :- g(_x) .         a$mod(_x) :- g$(_x) .
```

Without the import directive the occurrence of g/1 in the body of a$mod/1 would be taken as a call to a global predicate g/1 .

1.3 Module builtin predicates

module/1

module (_ModName)

arg1 : any : atom

If instantiated, the current module becomes *arg1*. If free, *arg1* will be instantiated to the name of the current module.

module/2

module (_Predicate, _ModName)

arg1 : partial : not integer, real or pointer
arg2 : any : atom

Unifies *arg2* with the module qualification of the principal functor of *arg1*.

For example :

```
?- module( a$one(_x),_y) , write(_y) .
one
```

module/3

module (_QualTerm, _ModName, _Term)

arg1 : any
arg2 : any : atom
arg3 : any

Arg3 is the term constructed from *arg1* by stripping the module qualification from the principal functor of *arg1*, and unifying this qualification with *arg2*. If *arg1* is free, *arg2* must be an atom and *arg3* must be partially instantiated.

For example :

```
?- module( a$one( b$two), one, a( b$two)) .
Yes
```

Succeeds.

mod_unif/2

mod_unif (_Term1, _Term2)

arg1 : any
arg2 : any

Unifies the 2 arguments, as if they had no module qualification at any level (i.e as if they were globals).

For example:

```
?- please(wq,on),mod_unif( a$one( _x), a$two( b$three)),
   module( _x,_y) .
   _x = b
   _y = ''
Yes
```

writem/2

writem (_LogFileName, _Term)
writem (_FilePointer, _Term)

arg1 : ground : atom or pointer
arg2 : any

As **write/2**, but all non-global names are written with their explicit module qualification. (See chapter on I/O predicates)

writem/1

writem (_Term)

arg1 : any

As **writem/2**, but on the current output stream. (See chapter on I/O predicates)

mlisting/1

mlisting (_ModName)

arg1 : ground : atom

Writes on the current output stream all the predicate definitions that are local to the module named *arg1*.

1.4 Explicit module qualification

Atoms and functors are fully identified by their name, arity and the module in which they were defined. The **explicit module qualification (emq)** is the syntactic correct way to express this. The global module has the name '', the atom of length zero.
The other ***ProLog*** items - integers, reals, pointers and variables - have no module qualification.

For example :

atom$modA
functor$modA(_arg1,atom$modA,globalatom$)
globalatom$
globalfunctor$(_arg1,atom$modA,globalatom$)
3.567
_var

denote atom and functor/3 as defined in the module modA, and globalatom and globalfunctor/3 as globals.

Explicit module qualification is only allowed with module names that appeared previously in a directive: the compiler will issue error messages if this rule is violated. An import directive can be used.

The use of emq is not obligatory and is actually discouraged. If no module qualification is used, the rules of the next section will be applied to determine to which module the functor belongs. In case of ambiguity, explicit qualification must be used.

1.5 Resolving module qualification

The rules for resolving the module qualification are (these apply at compile time):

- if foo/n is explicitely qualified with a module, then foo/n belongs to the module it is qualified with.
- if foo/n is not explicitely qualified but has a local declaration, an implied local declaration or a definition after the module declaration, foo/n is local.
- otherwise, if foo/n is imported from only one module, foo/n belongs to that module.
- otherwise, if foo/n is not imported, foo/n is global.
- otherwise, an error against the module rules was made.

In case of ambiguities, explicit module qualifications must be used. Ambiguities arise in the following cases:

- when the same functor is imported from different modules.
- when a functor is imported and has also a local declaration.
- when a functor is global and also local.
- when a functor is global and imported.

1.6 General builtins and modules

This paragraph explains the behavior of some general builtins when used inside a module. The most important rules are summarised below:

- The following predicates always create global atoms:

  numbervars/3
  numbervars/4

- The following predicates create global atoms in their first argument when output:

  ascii/2
  name/2
  atomtolist/2

- The predicate **atomtolist/2** also creates global atoms in its second argument when output.
- The predicate **read/1** interpretes functors locally if there already exists a local indication, otherwise functors are global.
- If a functor is derived from another functor, it inherits its module qualification.
  Examples are **=../2**, **functor/3**, used with any i/o pattern.
- The predicates **all_directives/0-1** do not output module declarations.
- The **write** predicates write terms without module qualification except when using **please(wm,on)** and for the predicates **writem/1-2**.

1.7 General directives and modules

The directives **op/3, index/2, mode/1** and **dynamic/1** influence the module qualification of the functors they declare as operator or as dynamic. If they occur after the module directive, the effect is that a local directive for that functor is implied.

The directive op(_prec,xfx,atom) must be considered as using the functor atom(_,_) and op(_prec,fx,atom) as using the functor atom(_) .

The directive dynamic(foo/n) (and foo/n index i and mode foo/n) must be considered as using the functor foo/n, this is foo(_,_,_,...,_) .The mode and index declarations have a similar effect as the dynamic declaration.

1.8 Interactive mode and modules

When a ***ProLog*** session starts (in the global module), none of the import, local or global declarations appearing in the consulted files have effect.
If a predicate was loaded local to a module, then it cannot be called, unless an explicit module qualification is used or unless you position yourself in the module.
In interactive mode explicit module qualification is always allowed.

Positioning within a module is done with the **module/1** builtin. Typing in new definitions in no_querymode, will cause them to be added to the new module. The assert predicates will assert the term exactly as it is specified (i.e. terms without emq will always be added to the global module)

For example:

```
?- module( modA) .
Yes
```

In this case all predicates from modA are accessible without explicit qualification.

EXTERNAL LANGUAGE INTERFACE

External Language Interface

Contents

Introduction

To be used effectively in an industrial environment products based on Prolog require a flexible interface to different types of hardware and software components. ***ProLog*** provides access to the external world via a general external language interface. The interface allows for easy communication with virtually every software package running on the hardware platform on which the final application needs to be delivered.

Through this interface the ***ProLog*** programmer can invoke their application procedures and functions written in a procedural language (C, Pascal, Fortran, Assembler). All the usual data types as well as complex data structures can be communicated through this interface. Moreover, backtracking external functions can be defined, and ***ProLog*** predicates can be invoked from the external functions. This constitutes the most straightforward and versatile connection mechanism ever made available by a Prolog system.

To illustrate its versatility, the external language interface has been used to couple ***ProLog*** with existing graphics, windowing and database packages. Among these are : SunView, XView, Xlib, SunUnify, Ingres and Sybase, giving the application builder unrestricted access to graphics and windowing facilities as well as to the operational data in relational databases.

ProLog by BIM - Reference Manual
External Language Interface
Chapter 1

Linking External Routines

1.1 Incremental linking

Before a routine written in another language than Prolog can be used from a Prolog program, it must be linked into the ***ProLog*** system and loaded. Using **BIMlinker**, one can create a new, customized ***ProLog*** system that has the required external routines linked in. Another method is to link these external routines incrementally in an already running system. This is a more flexible method, but it introduces some run-time overhead each time a set of routines must be linked and loaded. Incremental linking can be achieved by consulting a file or interactively from the top level of the running system. In both cases, a number of declarations must be provided. First, a linker directive to indicate which external routines are requested for and in which object files or libraries they can be found. Second, for each external routine a declaration of its mapping to a Prolog predicate must be given, including the declaration of the arguments.

The incremental linker opens the indicated object files and libraries to look up the requested routines. They are linked into the running system and their code is loaded in memory. Finally, the external routines are associated with Prolog predicates, as described in the declarations.

When linking routines from libraries, the code for the routines will only be loaded if it is not yet in memory. If it has already been loaded, the external routine is immediatley associated with the Prolog predicate.

1.2 Linker directive

The linker directives **extern_load/2,3** tell the incremental linker of ***ProLog*** which external routines must be linked and which object files and libraries must be opened to find them. In a file, this must be used as compiler directive, and interactively as a builtin predicate.

extern_load/3

extern_load (_Externals , _Objects , _Size)

arg1 : ground : list of atom
arg2 : ground : list of atom
arg3 : ground : integer

The linker is informed that it must link the external routines whose names are listed in *arg1*. And that these can be found by opening the object files and libraries that are listed in *arg2*. This also gives *arg3* as hint for the size of the code that must be loaded.

The order of object file names and libraries in *arg2* is important : all files are opened in the same order as they appear in the list. The libraries must be at the right place in the list, as the linker only retrieves those routines from a library that are unresolved at the moment the library is opened.

The size *arg3* is only a hint : the linker will allocate enough space to hold all the code that must be loaded. When the size is given as 0, the linker will print out the real size of the loaded code.

extern_load/2

extern_load (_Externals , _Objects)

arg1 : ground : list of atom
arg2 : ground : list of atom

Same as **extern_load/3** but without size hint.

1.3 Predicate mapping declarations

Each external routine can be mapped to one or more Prolog predicates or functions. This is performed with the **extern_predicate/1,2,3** and **extern_function/1,2,3** declarations. In a file, these must be used as compiler directive, and interactively as a builtin predicate.

extern_language/1

extern_language (_Language)

arg1 : ground : atom

The default language for following declarations is set to *arg1*. It must be one of the supported language identifiers (see *<language>* below).The default language is C.

extern_predicate/3

extern_predicate (_Language , _ExternalName , _PredDecl)

arg1 : ground : atom
arg2 : ground : atom
arg3 : ground : term

The external routine with external name *arg2*, and written in language *arg1*, is mapped to a Prolog predicate as described in *arg3*.

The external name *arg2* must be the same as in the external source code. If necessary, the incremental linker will append pre- and postfixes to find the name in the symbol tables.

The mapping description *arg3* consists of a term with the same name and arity as the Prolog predicate with which the external routine must be associated. Its arguments are the declarations for parameter passing. (See *<pred_decl>* below for a precise syntax).

Any existing mapping for the indicated Prolog predicate, is overridden.

extern_predicate/2

extern_predicate (_Language , _PredDecl)

extern_predicate (_ExternalName , _PredDecl)

arg1 : ground : atom
arg2 : ground : term

Same as **extern_predicate/3**, but with either the language or external name argument omitted.

If the language is omitted, it is assumed to be the default language, as set with **extern_language/1**.

If the external name is omitted, it is assumed to be the same as the name of the Prolog predicate, as defined in *arg2*.

extern_predicate/1

extern_predicate (_PredDecl)

arg1 : ground : term

Same as **extern_predicate/3**, but with the default language and with the external name the same as the Prolog predicate name.

extern_function/3

extern_function (_Language , _ExternalName , _FuncDecl)

arg1 : ground : atom
arg2 : ground : atom
arg3 : ground : term

The external routine with external name *arg2* and written in language *arg1*, is mapped to a Prolog function as described in *arg3*.

The external name *arg2* must be the same as in the external source code. If necessary, the incremental linker will append pre- and postfixes to find the name in the symbol tables.

The mapping description *arg3* has the form *term : type*. The term has the same name and arity as the Prolog function with which the external routine must be associated. Its arguments are declarations for the parameter passing. The type indicates the type of the function result. (See <*func_decl*> below for a precise syntax).

Any existing mapping for the indicated Prolog function, is overridden.

extern_function/2

extern_function (_Language , _FuncDecl)

extern_function (_ExternalName , _FuncDecl)

arg1 : ground : atom
arg2 : ground : term

Same as **extern_function/3**, but with either the language or external name argument omitted.

If the language is omitted, it is assumed to be the default language, as set with **extern_language/1**.

If the external name is omitted, it is assumed to be the same as the name of the Prolog function, as defined in *arg2*.

extern_function/1

extern_function (_FuncDecl)

arg1 : ground : term

Same as **extern_function/3**, but with the default language and with the external name the same as the Prolog function name.

The syntax for the Prolog predicate and function declarations used in the mapping decalarations can be described as follows.

<pred_decl>	⇒	*<pro_name>* [(*<arguments>*)]
<func_decl>	⇒	*<pro_name>* [(*<arguments>*)] : *<type>*
<arguments>	⇒	*<argument>* [, *<arguments>*]
<argument>	⇒	*<type>* [: *<struct>*] [: *<mode>*]
<type>	⇒	integer | short | long | real | float | double | pointer | atom | string | string : *<string_size>* | bpterm | untyped
<struct>	⇒	array | list
<mode>	⇒	i | o | m | r | e
<string_size>	⇒	*<integer>*
<language>	⇒	C | Fortran | Pascal
<ext_name>	⇒	*<atom>*

The *<pro_name>* is the name of the Prolog predicate or function the external routine must be mapped to.

A declaration for an external predicate consists of a declaration for each of its arguments. For an external function the result type must also be declared.

If no *<struct>* is given, the parameter is a simple argument.

If no *<mode>* is given, it defaults to i (input).

Argument declarations (type, structure and mode) are described in detail in the next chapter.

1.4 Interactive linking

For interactive incremental linking, some additional builtins are provided. Two for erasing existing declarations and one to activate the incremental linker.

extern_clear/0

extern_clear

All existing declarations and linker directives are erased.

Before giving declarations for a new linking phase, any existing declarations should first be removed as these would otherwise be merged with new declarations.

extern_clear/1

extern_clear (_Predicate)

extern_clear (_Function : _ReturnType)

arg1 : partial : term

All existing declarations matching *arg1* are erased. The first form, a term with principal functor the predicate name, only matches external predicate declarations. The second form, a term with principal functor a function name, and with a return type (that can be free), only matches functions.

extern_go/0

extern_go

The incremental linker is activated to link and load the external routines as described in previously given declarations.

After completion, the existing declarations remain effective.

1.5 Mapping inquiry

Once an external routine is mapped to a Prolog predicate, the external name and the address of the external routine for this predicate can be retrieved.

extern_name_address/3

extern_name_address (_Predicate , _ExternalName , _Address)

arg1 : partial : term
arg2 : free : atom
arg3 : free : pointer

Succeeds if the principal functor of *arg1* is an external predicate.

The external name of the routine with which the Prolog predicate is associated, is unified with *arg2* and *arg3* is instantiated to the address of the external routine.

1.6 Objects and libraries

Object files must be compiled to a standard Unix object file (.o file) before they can be linked. To compile a file with base name *externals* to such an object file, named *externals.o*, one of the following commands can be used (depending on the source language).

Table : minimal compile commands

Language	Compile command
C	cc -c externals.c
Fortran	f77 -c externals.f
Pascal	pc -c externals.p

For standard applications, the following libraries must be added at the end of the object list in the **extern_load/2,3** directive.

Table : standard libraries to open

Language	Libraries
C	-lc
Fortran	-lF77 -lI77 -lc
Pascal	-lpc -lm -lc

ProLog by BIM - Reference Manual
External Language Interface
Chapter 2

Calling External Routines from Prolog

2.1 Parameter mapping declarations

When an external routine is mapped to a Prolog predicate or function, the mapping must describe how parameters must be passed. This is accomplished by giving a declaration for each argument in the external predicate or function declaration. Predicate and function declarations are described in detail in the previous chapter. Here the argument declarations are specified.

An argument declaration has the following form:

<argument> ⇒ *<type>* [: *<struct>*] [: *<mode>*]

<type> ⇒ integer | short | long | real | float | double | pointer | atom | string | string : *<string_size>* | bpterm | untyped

<struct> ⇒ array | list

<mode> ⇒ i | o | m | r | e

<string_size> ⇒ *<integer>*

Types

Types *integer* and *real* are language dependent default types that are mapped to either a short or long corresponding type, depending on the language used.

Table : mapping of *integer* and *real*

Language	Default type	Mapped type
C	integer real	long double
Fortran	integer real	long float
Pascal	integer real	long double

The special type *untyped* is mapped to a normal type depending on the type of the corresponding argument at the moment the external routine is called.

Table : mapping of *untyped*

Actual type	Mapped type
integer	integer
real	real
pointer	pointer
atom	string or string : *s*

Correspondences between pre-defined types and data types in the external language are listed in the following tables.

C types

Table : argument types and corresponding C data types

Type	C Data type	Description
integer	int	Long integer (4 byte)
long	long	Long integer (4 byte)
short	short	Short integer (2 byte)
real	double	Long real (8 byte)
double	double	Long real (8 byte)
float	float	Short real (4 byte)
pointer	BP_Pointer	Pointer (4 byte)
atom	BP_Atom	Atom identifier (internal form)
string	BP_String	String (null-terminated)
string : *s*	BP_String	String (fixed size)
bpterm	BP_Term	Term pointer (internal form)

Type *string* is a null-terminated character array. Type *string : s* is a fixed size array of *s* characters. If *s* is 0, the size of the array is determined at the moment of calling the external routine, and it is passed, followed by the array.

Fortran types

Table : argument types and corresponding Fortran data types

Type	Fortran type	Description
integer	integer	Long integer (4 byte)
long	integer	Long integer (4 byte)
short	integer * 2	Short integer (2 byte)
real	real	Short real (4 byte)
double	real * 8	Long real (8 byte)
float	real	Short real (4 byte)
pointer	integer	Pointer (4 byte)
atom	integer	Atom identifier (internal form)
string : *s*	character * *s*	String (fixed size)
bpterm	integer	Term pointer (internal form)

There are no corresponding Fortran data types for *pointer*, *atom* and *bpterm*. These can be represented in Fortran as *integer*.

Type *string*, which is a null-terminated character array, is not allowed for Fortran routines. If *s* is 0 in type *string : s*, it is set to the length of the atom at the moment the external routine is called.

Pascal types

Table : argument types and corresponding Pascal data types

Type	Pascal type	Description
integer	integer	Long integer (4 byte)
long	integer	Long integer (4 byte)
short	-32768..32768	Short integer (2 byte)
real	real	Long real (8 byte)
double	real	Long real (8 byte)
float	shortreal	Short real (4 byte)
pointer	^*any*	Pointer (4 byte)
atom	integer	Atom identifier (internal form)
string : *s*	array of char	String (fixed size)
bpterm	integer	Term pointer (internal form)

There are no corresponding Pascal data types for *atom* and *bpterm*. They can be represented in Pascal as *integer*. Type *pointer* can be a Pascal pointer to any type.

Type *string*, which is a null-terminated character array, is not allowed for Pascal routines. In type *string : s*, *s* must not be 0.

Structures

Parameters can be passed as simple arguments or in a structured form. The ***ProLog*** built-in interface provides two ways for structuring arguments : the *list* and the *array* structures.

A *list* is used to map one single, composed Prolog parameter to a sequence (list) of consecutive external parameters. The Prolog parameter must be a Prolog list of simple parameters. The external corresponding parameters are simple parameters, one for each element of the Prolog list. For input as well as for output parameters, the Prolog argument must be instantiated to a flat linear list. The number of elements in that list determines the number of parameters that are passed between the predicate and the external routine. It is the responsibility of the external routine to use or provide the right number of arguments. The *list* structure corresponds to the *varargs* feature of C (especially when combined with *untyped* as type). The number of arguments of the external routine is variable and can have different types.

An *array* maps one single, composed Prolog parameter to one single, composed external parameter, structured as an array. The Prolog parameter can be either a Prolog list or a Prolog term of simple parameters. The external corresponding parameter is an array. For input as well as for output parameters, the Prolog argument must be instantiated to either a flat linear list or a flat term. The number of elements in the external array is the same as the number of elements in the list, or as the arity of the term. The elements of the external array are mapped to the elements of the list or to the arguments of the term. If a term is used in Prolog, its functor is ignored and may be chosen arbitrarily. An *array* structured parameter must have at least one argument.

Modes

A mode determines in which direction the parameter is passed. The mode is abbreviated to a single mnemonic letter. The following table gives the full names of the modes and the corresponding mnemonics.

Table : mode mnemonics

Mnemonic	Full name
i	input
o	output
m	mutable
r	return
e	evaluate

Mode *r* is used to retrieve the return value of an external routine. For every external predicate there can be at most one *return* parameter, and it must always be the first parameter. Alternatively, a return value of an external routine does not have to be retrieved. If there is no *return* parameter declared, it will simply be ignored. A Prolog function, mapped to

an external function cannot have a *return* parameter since the return value of the external function is mapped to the Prolog function result.

Mode *m* is a kind of in-out mode. This does not mean that it introduces destructive assignment in Prolog. If the parameter was instantiated when calling the external routine, and it is changed when exiting it, a failure will occur. The *mutable* mode is only significant for *string* type and *array* structured parameters. These are passed by reference. With an *output* mode, a double reference is passed : the interface expects the external routine to pass back a reference. With mode *m*, only one reference is passed, and the external routine may change the object that is behind that reference. This corresponds to the typical C passing of arrays. An array is passed to a routine by giving the address of the array. The called routine can then change the contents of that array. The same applies to *string* parameters, as they are arrays of characters.

Mode *e* is a variant of *i*. The actual argument is first evaluated (as the second argument of **is/2**), and then passed with *input* mode. This is similar to the in-line expression evaluation that is denoted with **?/1**.

Restrictions

There are some restrictions concerning the type correspondence between Prolog and external data types :

- Prolog integers are limited to 29 bits of precision.
- External short reals (*float*) are mapped to Prolog reals in an unspecified way : the additional decimal digits will not necessarily be 0.

Not all combinations of language, type, structure and mode are allowed. The following restrictions hold :

- For external functions the argument modes must be *i* (*input*) or *e* (*evaluate*).
- For external functions the result type is restricted to numerical types (*integer*, *real* and variants).
- Type *string* is not allowed in Fortran and Pascal.
- Type *string : 0* is not allowed in Pascal.
- Type *string : 0* in mode *r* (*return*) is not allowed.
- Mode *e* (*evaluate*) is only allowed with simple numerical types (*integer*, *real* and variants).
- Mode *o* (*output*) is the same as *m* (*mutable*) in Fortran and Pascal.
- Structure *array* in mode *r* (*return*) is not allowed in Fortran.
- Structure *list* in mode *r* (*return*) is not allowed.
- Type *bpterm* is only allowed in mode *i* (*input*).
- Type *untyped* is only allowed in mode *i* (*input*).

- Type *untyped* in structure *array* is not allowed.

A schematic overview of non-allowed combinations is given below. The * stands for anything.

Combinations that are not allowed for any language :

untyped : * : *m*	*bpterm* : * : *m*	* : *list* : *r*	*pointer* : *e*
untyped : * : *o*	*bpterm* : * : *o*	*string* : 0 : * : *r*	*atom* : *e*
untyped : * : *r*	*bpterm* : * : *r*		*string* : *e*
untyped : * : *e*	*bpterm* : * : *e*	* : *list* : *e*	*string* : * : *e*
untyped : *array* : *	*bpterm* : *array* : *	* : *array* : *e*	

Combinations that are not allowed for Fortran and Pascal :

Fortran	Pascal
string : * : *	*string* : * : *
* : *array* : *r*	*string* : 0 : * : *

2.2 General parameter passing rules

The global parameter passing rules are described by specifying the actions that are performed at *call* (when calling the external routine from Prolog), and at *exit* (when returning from the external routine). These are given for each of the modes.

input - call
The actual parameter type is converted from the Prolog type to the external type as described in the declaration. This converted value is passed to the external routine following the parameter passing conventions for the language concerned.

input - exit
No actions are performed.

output - call
Enough memory is allocated to contain a value of the declared type, in the external language. This memory zone is referenced by one level and this reference is passed to the external routine.

output - exit
The reference that was passed to the external routine is dereferenced to retrieve the value of the parameter. This value is type converted from the external type to the corresponding Prolog type as declared. The resulting Prolog value is then unified with the actual parameter of the Prolog predicate. This may result in instantiation or simply success or failure.

mutable - call
The actual parameter is type converted from the Prolog type to the external type as described in the declaration. If the actual parameter is not instantiated, a default value is taken instead. If for the declared language, structure and type, the value already has a reference, this reference is passed to the external routine. Otherwise, an extra level of reference to the value is created and this reference is passed.

mutable - exit
The reference that was passed to the external routine is dereferenced to retrieve the value of the parameter. This value is type converted from the external type to the corresponding Prolog type as declared. The resulting Prolog value is then unified with the actual parameter of the Prolog predicate. This may result in instantiation or simply success or failure. A failure provokes a warning message.

return - call
No actions are performed.

return - exit
The function result of the external routine is type converted from the external type to the corresponding Prolog type as declared. The resulting Prolog value is then unified with the actual parameter of the Prolog predicate. This may result in instantiation or simply success or failure.

evaluate - call

The actual parameter is evaluated as an expression (as if it were the second argument of a call of **is/2**). The result of this evaluation is type converted from the Prolog type to the external type as described in the declaration. This converted value is passed to the external routine following the parameter passing conventions for the language concerned.

evaluate - exit

No actions are performed.

2.3 Parameter passing specification

Detailed specifications of parameter passing for each language, type and mode are given by means of pictures, representing the parameter structures at *call* and *exit* time. A basic memory cell of 4 bytes is represented by a 1/2 inch wide box.

Boxes with a thick border, represent the data that is actually passed as parameter (i.e. that is pushed on the stack or in the input registers). A shaded box with thick border, represents data that is actually passed as function result (returned in a register).

A ? in a box, means its value is undefined.

Shaded boxes at the *exit*, indicate zones that may or should have been modified by the external routine.

Boxes that exist only at the *exit*, are zones that must be managed by the external language. These must remain living at least *during exit*. If the external data is made *static*, there are no problems involved. However, if it is allocated dynamically, special care has to be taken. It may not be deallocated when the external routine is left, but only when the Prolog predicate is reentered. As this may introduce extra memory management efforts, these parameter passing modes, combined with dynamic data structures should be avoided when unnecessary. These cases are labeled with the remark 'may be deallocated after *exit*'.

All memory zones that are managed by the external interface, remain in effect as long as the call of the external routine is active.

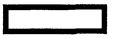
The maximum length of an atom is indicated as MAX_ATOM, which is 16384.

In most cases, a string length is extended to get an even number of characters. This is indicated with s' = even(s), which means that s' = s if s is even, and s' = s + 1 if s is odd.

C - Simple input

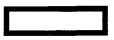
integer / long : i

call



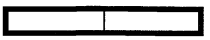
short : i

call



float : i

call



real / double : i

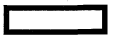
call

pointer : i

call

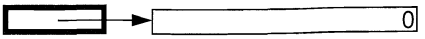
atom : i

call



string : i

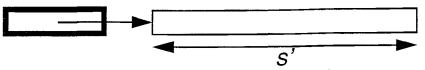
call



The text zone is long enough to hold the actual string and a terminating 0 byte.

string : s : i (s > 0)

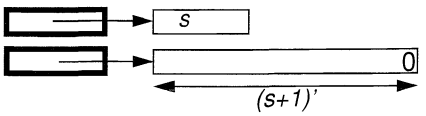
call



s' = even(s). The text zone is s' bytes.

string : 0 : i

call



(s+1)' = even(s+1). The text zone is (s+1)' bytes. And s is set to the length of the actual string.
Remark : There are two consecutive external arguments for this parameter.

C - Simple output

integer / long : o

call

exit

short : o

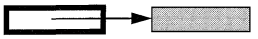
call

exit

float : o

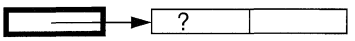
call

exit



real / double : o

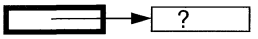
call



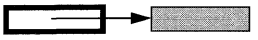
exit

pointer : o

call

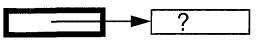


exit

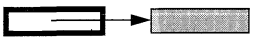


atom : o

call

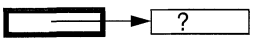


exit



string : o

call

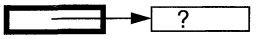


exit

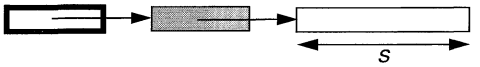
The text zone must be terminated with a 0 byte. It may be deallocated after *exit*.

string : s : o (s > 0)

call



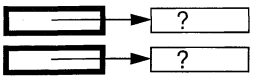
exit



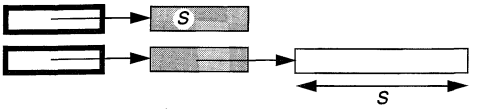
The whole length s of the text zone is considered as the resulting string. The text zone may be deallocated after *exit*.

string : 0 : o

call



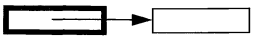
exit

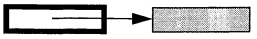


The external routine has to indicate the length of the text zone at *exit*. The text zone may be deallocated after *exit*.

C - Simple mutable

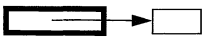
integer / long : m

call 

exit 

Default value : 0.

short : m

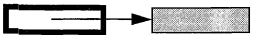
call 

exit 

Default value : 0.

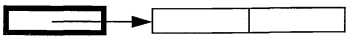
float : m

call 

exit 

Default value : 0.

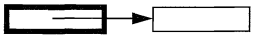
real / double : m

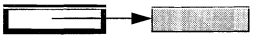
call 

exit 

Default value : 0.

pointer : m

call 

exit 

Default value : 0x0.

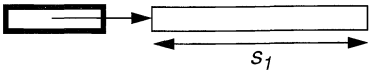
atom : m

call 

exit 

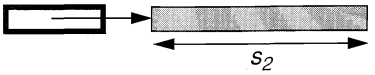
Default value : empty atom.

string : m

call 

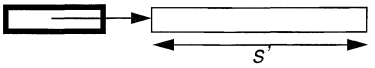
Default for s : MAX_ATOM.

The text zone contains the actual string (default empty), terminated with a 0 byte.

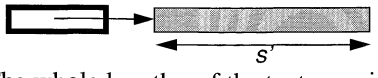
exit 

The text zone must contain the string, terminated with a 0 byte. And $s_2 \leq s_1$.

string : s : m (s > 0)

call 

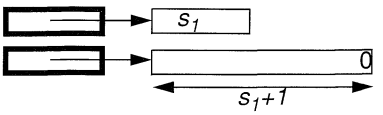
s' = even(s). The text zone contains the actual string (default empty), terminated with a 0 byte, truncated to s' bytes.

exit 

The whole length s of the text zone is considered as the resulting string.

string : 0 : m

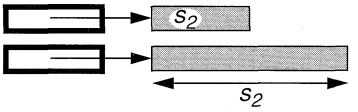
call



Default for s : MAX_ATOM.

The text zone contains the actual string (default empty), terminated with a 0 byte.

exit



The external routine has to indicate the length of the text zone at *exit*. And $s_2 \leq s_1$. The whole length s_2 of the text zone is considered as the resulting string.

C - Simple return

integer / long : r

exit

short : r

exit

float : r

exit

real / double : r

exit

pointer : r

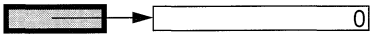
exit

atom : r

exit

string : r

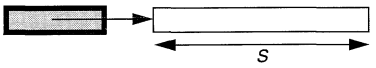
exit



The text zone must contain the string, terminated with a 0 byte. It may be deallocated after *exit*.

string : s : r (s > 0)

exit



The whole length s of the text zone is considered as the resulting string.

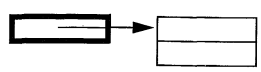
string : 0 : r

Not allowed.

C - Array input

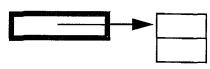
integer / long : array : i

call



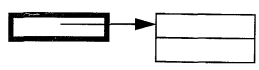
short : array : i

call



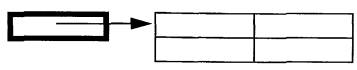
float : array : i

call



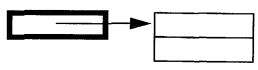
real / double : array : i

call



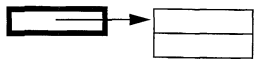
pointer : array : i

call



atom : array : i

call



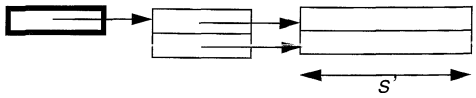
string : array : i

call

The text zones are long enough to hold the actual strings with a terminating 0 byte.

string : s : array : i (s > 0)

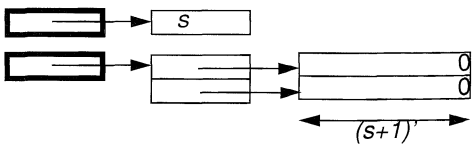
call



s' = even(s). The text zones are s' bytes long.

string : 0 : array : i

call

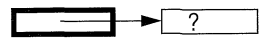


(s+1)' = even(s+1). The text zones are (s+1)' bytes long. Size s is set to the length of the longest actual string. The strings in the text zones are terminated with a 0 byte.

C - Array output

integer / long : array : o

call



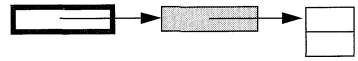
exit

The array may be deallocated after *exit*.

short : array : o

call

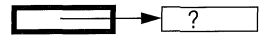
exit



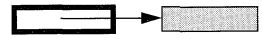
The array may be deallocated after *exit*.

float : array : o

call



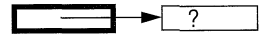
exit



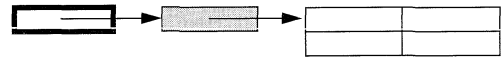
The array may be deallocated after *exit*.

real / double : array : o

call



exit



The array may be deallocated after *exit*.

pointer : array : o

call

exit

The array may be deallocated after *exit*.

atom : array : o

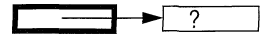
call

exit

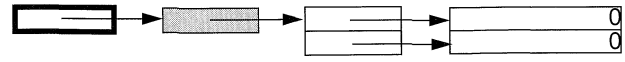
The array may be deallocated after *exit*.

string : array : o

call

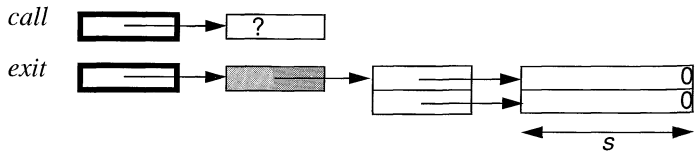


exit



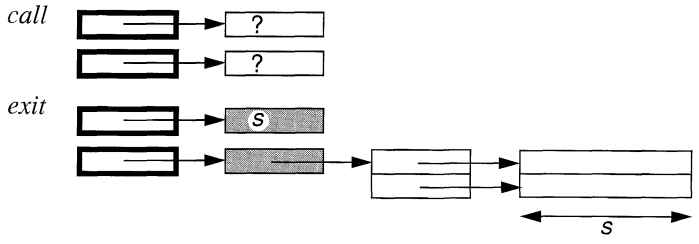
The text zones must end on a 0 byte. The pointer array and text zones may be deallocated after *exit*.

string : s : array : o (s > 0)



The whole length s of the text zones are considered as the resulting strings. The pointer array and text zones may be deallocated after *exit*.

string : 0 : array : o

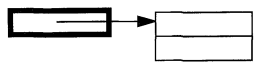


The external routine has to indicate the length of the text zones at *exit*. The pointer array and text zones may be deallocated after *exit*.

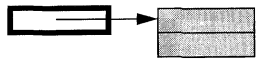
C - Array mutable

integer / long : array : m

call



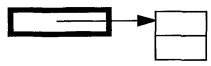
exit



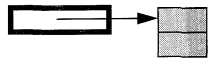
Default value : 0.

short : array : m

call



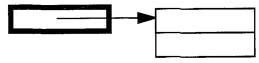
exit



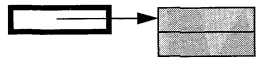
Default value : 0.

float : array : m

call



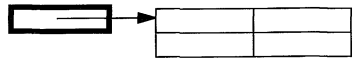
exit



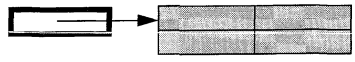
Default value : 0.

real / double : array : m

call



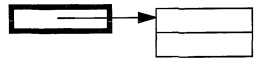
exit



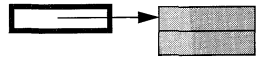
Default value : 0.

pointer : array : m

call



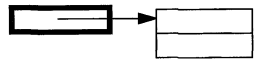
exit



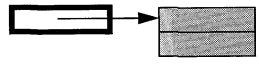
Default value : 0x0.

atom : array : m

call



exit



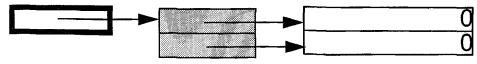
Default value : empty atom.

string : array : m

call

The text zones are long enough to hold the actual strings (default emtpy) with a terminating 0 byte.

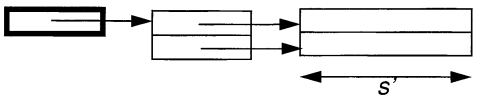
exit



The text zones must end on a 0 byte. The text zones may be deallocated after *exit*.
Remark : The text zones at *exit* are not necessarily the same as those at *call*.

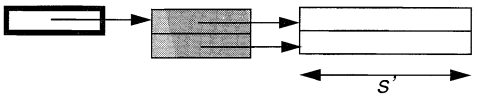
string : s : array : m (s > 0)

call



s' = even(s). The text zones are s' bytes long.

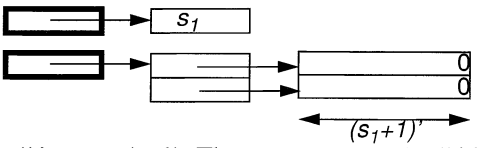
exit



The whole length s of the text zones are considered as the resulting strings. The text zones may be deallocated after *exit*.
Remark : The text zones at *exit* are not necessarily the same as those at *call*.

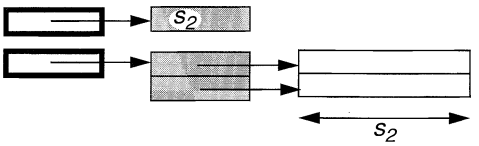
string : 0 : array : m

call



(s+1)' = even(s+1). The text zones are (s+1)' bytes long. And s is set to the length of the longest actual string. The strings in the text zones are terminated with a 0 byte.

exit

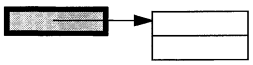


The external routine has to indicate the length of the text zones at *exit*. The text zones may be deallocated after *exit*.
Remark : The text zones at *exit* are not necessarily the same as those at *call*.

C - Array return

integer / long : array : r

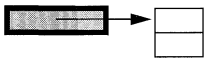
exit



The array may be deallocated after *exit*.

short : array : r

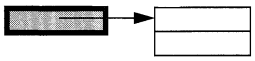
exit



The array may be deallocated after *exit*.

float : array : r

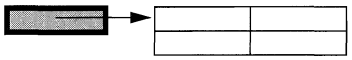
exit



The array may be deallocated after *exit*.

real / double : array : r

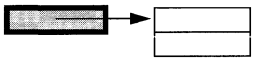
exit



The array may be deallocated after *exit*.

pointer : array : r

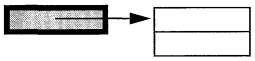
exit



The array may be deallocated after *exit*.

atom : array : r

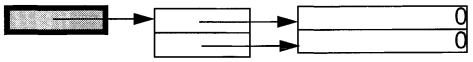
exit



The array may be deallocated after *exit*.

string : array : r

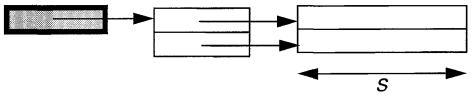
exit



The text zones must have a terminating 0 byte. The pointer array and text zones may be deallocated after *exit*.

string : s : array : r (s > 0)

exit



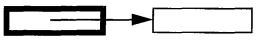
The whole length s of the text zones are considered as the resulting strings. The pointer array and text zones may be deallocated after *exit*.

string : 0 : array : r

Not allowed.

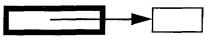
Fortran - Simple input

integer / long : i

call 

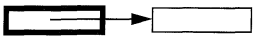
Default value : 0.

short : i

call 

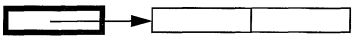
Default value : 0.

real / float : i

call 

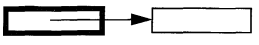
Default value : 0.

double : i

call 

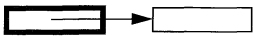
Default value : 0.

pointer : i

call 

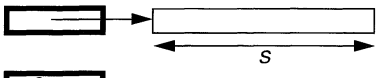
Default value : 0x0.

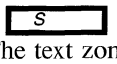
atom : i

call 

Default value : empty atom.

string : s : i (s > 0)

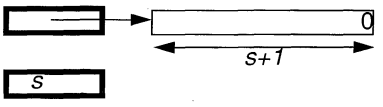
call 



Default : empty string.

The text zone contains the actual string, terminated with a 0 byte, truncated to s bytes.
Remark : There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

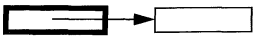
string : 0 : i

call 

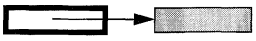
The text zone contains the actual string (default empty), terminated with a 0 byte.
Remark : There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

Fortran - Simple mutable

integer / long : m

call 

Default value : 0.

exit 

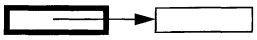
short : m

call 

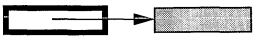
Default value : 0.

exit 

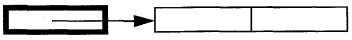
real / float : m

call 

Default value : 0.

exit 

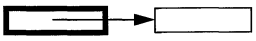
double : m

call 

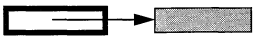
Default value : 0.

exit 

pointer : m

call 

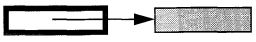
Default value : 0x0.

exit 

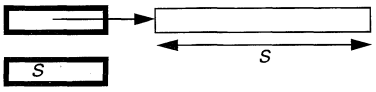
atom : m

call 

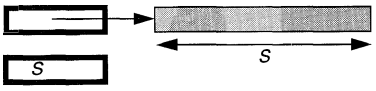
Default value : empty atom.

exit 

string : s : m (s > 0)

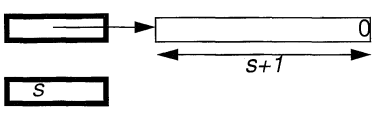
call 

The text zone contains the actual string (default empty), terminated with a 0 byte, truncated to s bytes.

exit 

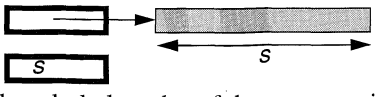
The whole length s of the text zone is considered as the resulting string.
Remark : There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

string : 0 : m

call 

Default for s : MAX_ATOM.

The text zone contains the actual string (default empty), terminated with a 0 byte.

exit 

The whole length s of the text zone is considered as the resulting string.
Remark : There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

Fortran - Simple return

integer / long : r

exit

short : r

exit

real / float : r

exit

double : r

exit

pointer : r

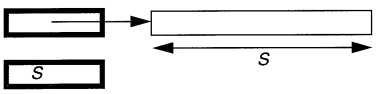
exit

atom : r

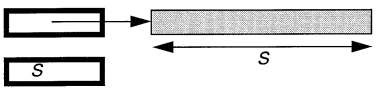
exit

string : s : r (s > 0)

call



exit



The whole length s of the text zone is considered as the resulting string.
Remark : A Fortran *return string* is treated as a *mutable string* parameter.

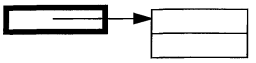
string : 0 : r

Not allowed.

Fortran - Array input

integer / long : array : i

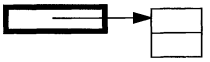
call



Default value : 0.

short : array : i

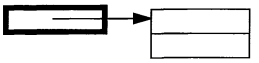
call



Default value : 0.

real / float : array : i

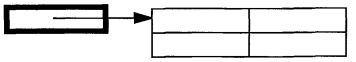
call



Default value : 0.

double : array : i

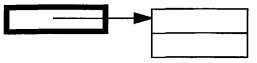
call



Default value : 0.

pointer : array : i

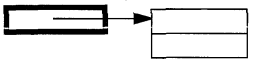
call



Default value : 0x0.

atom : array : i

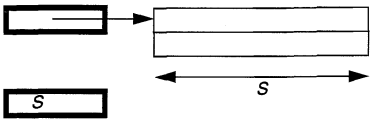
call



Default value : empty atom.

string : s : array : i (s > 0)

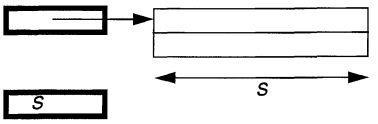
call



The text zones contain the actual strings (default empty), terminated with a 0 byte, truncated to s bytes.

string : 0 : array : i

call

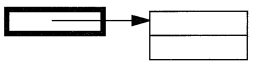


The text zones contain the actual strings (default empty), terminated with a 0 byte, truncated to s bytes. Where s is set to the length of the longest actual string.

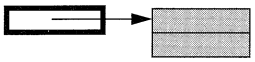
Fortran - Array mutable

integer / long : array : m

call



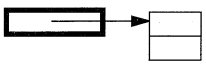
exit



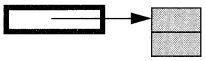
Default value : 0.

short : array : m

call



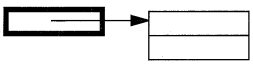
exit



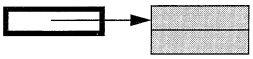
Default value : 0.

real / float : array : m

call



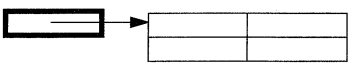
exit



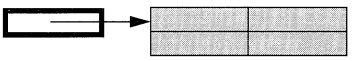
Default value : 0.

double : array : m

call



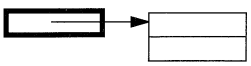
exit



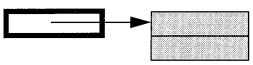
Default value : 0.

pointer : array : m

call



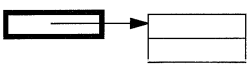
exit



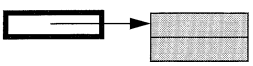
Default value : 0x0.

atom : array : m

call



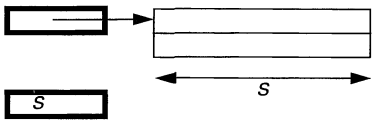
exit



Default value : empty atom.

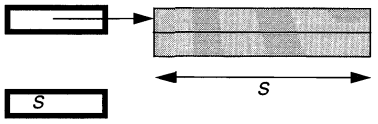
string : s : array : m (s > 0)

call



The text zones contain the actual strings (default empty), terminated with a 0 byte, truncated to s bytes.

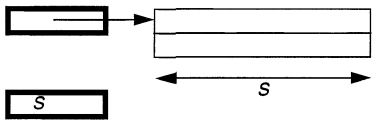
exit



The whole length s of the text zones are considered as the resulting strings.
Remark : There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

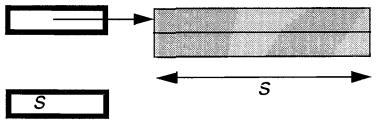
string : 0 : array : m

call



The text zones contain the actual strings (default empty), terminated with a 0 byte, truncated to s bytes. Where s is set to the length of the longest actual string.

exit

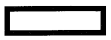


The whole length s of the text zones are considered as the resulting strings.
Remark : There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

Pascal - Simple input

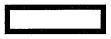
integer / long : i

call



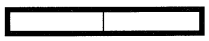
short : i

call



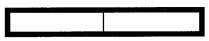
float : i

call



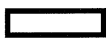
real / double : i

call



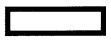
pointer : i

call



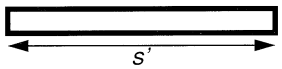
atom : i

call



string : s : i (s > 0)

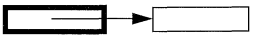
call



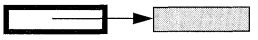
s' = even(s). The text zone is s' bytes.

Pascal - Simple mutable

integer / long : m

call 

Default value : 0.

exit 

short : m

call 

Default value : 0.

exit 

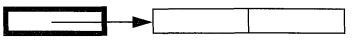
float : m

call 

Default value : 0.

exit 

real / double : m

call 

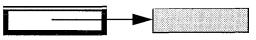
Default value : 0.

exit 

pointer : m

call 

Default value : 0x0.

exit 

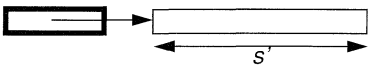
atom : m

call 

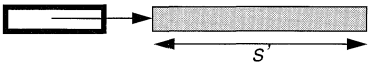
Default value : empty atom.

exit 

string : s : m (s > 0)

call 

s' = even(s). The text zone contains the actual string (default empty), terminated with a 0 byte, truncated to s' bytes.

exit 

The whole length s of the text zone is considered as the resulting string.

Pascal - Simple return

integer / long : r

exit

short : r

exit

float : r

exit

real / double : r

exit

pointer : r

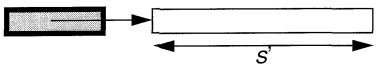
exit

atom : r

exit

string : s : r (s > 0)

exit

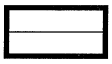


s' = even(s). The whole length s of the text zone is considered as the resulting string. The text zone may be deallocated after *exit*.

Pascal - Array input

integer / long : array : i

call

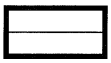


short : array : i

call

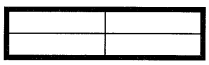
float : array : i

call



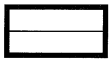
real / double : array : i

call



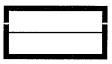
pointer : array : i

call



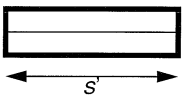
atom : array : i

call



string : s : array : i (s > 0)

call

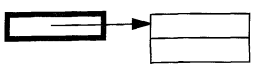


s' = even(s). The text zones are exactly *s'* bytes long.

Pascal - Array mutable

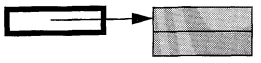
integer / long : array : m

call



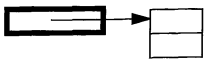
Default value : 0.

exit



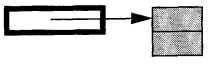
short : array : m

call



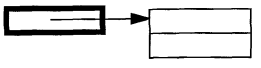
Default value : 0.

exit



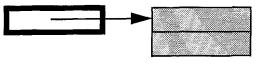
float : array : m

call



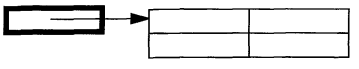
Default value : 0.

exit



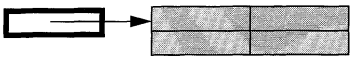
real / double : array : m

call



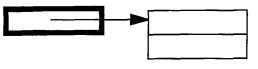
Default value : 0.

exit



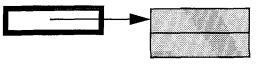
pointer : array : m

call



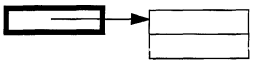
Default value : 0x0.

exit



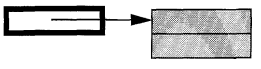
atom : array : m

call



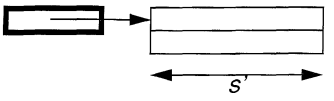
Default value : empty atom.

exit



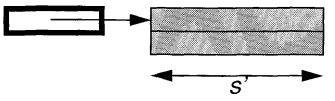
string : s : array : m (s > 0)

call



s' = even(s). The text zones are s' bytes long.

exit

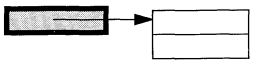


The whole length s of the text zones are considered as the resulting strings.

Pascal - Array return

integer / long : array : r

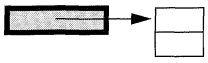
exit



The array may be deallocated after *exit*.

short : array : r

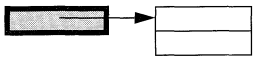
exit



The array may be deallocated after *exit*.

float : array : r

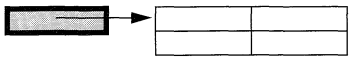
exit



The array may be deallocated after *exit*.

real / double : array : r

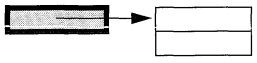
exit



The array may be deallocated after *exit*.

pointer : array : r

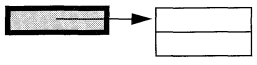
exit



The array may be deallocated after *exit*.

atom : array : r

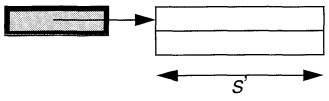
exit



The array may be deallocated after *exit*.

string : s : array : r (s > 0)

exit



s' = even(s). The whole length s of the text zones are considered as the resulting strings. The array may be deallocated after *exit*.

2.4 Backtracking external predicates

The ***ProLog*** external language interface provides facilities for simulating backtracking external predicates. Such predicates behave in a similar way as normal non-deterministic Prolog predicates. For this purpose, a number of builtin predicates and external builtin routines are provided.

mark_repeat/2

mark_repeat (_Info , _ChoicePointId)

arg1 : partial : term
arg2 : ground : atom

This predicate succeeds repeatedly on backtracking. The choicepoint identifier *arg2* is a global identification name for the choicepoint that controls this repeat. The information *arg1* is associated with this choicepoint, and is user definable.

recent_mrepeat/2

recent_mrepeat (_Info , _ChoicePointId)

arg1 : any : term
arg2 : ground : atom

The information associated with the most recent, still active choicepoint with identifier *arg2*, is unified with *arg1*. If no such choicepoint exists, this predicate fails.

The external routines for simulating backtracking are :

BIM_Prolog_rm_mrepeat (*choice_point_id*)
BP_String *choice_point_id*;

The most recent choicepoint with identifier *choice_point_id* is removed, by cutting it away.

BIM_Prolog_rm_all_mrepeat (*choice_point_id*)
BP_String *choice_point_id*;

All active choicepoints with identifier *choice_point_id* are removed, by cutting them away.

A common method for simulating backtracking external predicates with these predicates and routines is the following. An external routine is called to set up an iterator externally. In Prolog, a choicepoint is created with **mark_repeat/2**. This will be used as the Prolog iteration controller. The main external routine is called to generate the next solution in the iterator. When the last solution is found, the external routine must cut away the choicepoint. Another fail in Prolog will then no longer retry the iteration.

The **recent_mrepeat/2** builtin is useful for doing resource management, like memory management, in case a Prolog cut would have removed external choicepoints.

2.5 Examples

External function

An external function *add()* is defined and mapped to a Prolog function *add/2* which takes two real arguments, adds them and returns the real result. The external definition is given in the C file *add.c*.

The C definition, in *add.c*, is quite simple :

```
double add ( x , y )
double x, y;
{
    return ( x + y );
}
```

This file must be compiled with :

```
cc -c add.c
```

Prolog declarations :

```
:- extern_load ( [ add ] , [ 'add.o' ] ) .
:- extern_function ( add ( real : e , real : e ) : real ) .
```

The arguments are declared to be of type *evaluate*, so that we can use expressions as arguments of the function.

To evaluate the expression

```
2 * add ( sin ( _A ) , add ( 1 , _B ) ) + _B
```

and assign the result to _X, the following Prolog goal can be executed :

```
_X is 2 * add ( sin ( _A ) , add ( 1.0 , real ( _B ) ) ) + _B
```

The explicit type conversion of _B to real is necessary as the interface does not perform implicit type conversions of the arguments.

Predicate with external symbol table

Suppose we have an existing library that contains a routine that returns an enumeration type value (e.g. an error status). Enumeration types in C are implemented as integers. To make this transparent to the Prolog user, an external routine is added between this library routine and the Prolog predicate. This routine looks up the integer code in a table and returns an atom representing the symbolic enumeration value.

A definition of the enumeration type and the header of the library routine might be something like :

```
typedef enum {
    NoError ,
    SyntaxError,
    . . .
} ErrorStatus;

. . .

ErrorStatus get_error_status ( );
```

The external table with the translation from enumeration value to symbolic name, must be initialized :

```
error_table [ NoError ] = BIM_Prolog_string_to_atom ( "NoError" );
error_table [SyntaxError ] = BIM_Prolog_string_to_atom ( "SyntaxError" );
. . .
```

The extra external routine, placed between the Prolog predicate and the library routine and defined in file *interface.c*, becomes then :

```
BP_Atom Iget_error_status ( )
{
    return ( error_table [ get_error_status ( ) ] );
}
```

To use this in Prolog, the following declarations are required :

```
:- extern_load ( [ Iget_error_status ] , [ 'interface.o' ] ) .
:- extern_predicate ( Iget_error_status , get_error_status ( atom : r ) ) .
```

Retrieving the error status is done as follows :

```
?- get_error_status ( _Status ) .
_Status = NoError
```

The extra external routine is named after the library routine, but with a suffix I, so that the Prolog predicate can have exactly the same name as the library routine, making the interface transparent to the Prolog user.

Mutable parameter

As an illustration of a mutable parameter, consider the C function *strcpy()* which copies a string.

The C code for this function, in file *strcpy.c* is :

```
strcpy ( o , i )
char * o, * i;
{
    while ( *o++ = *i++ ) ;
}
```

The characters from the input array *i* are copied in the output array *o*. This must be large enough to hold the string, and the terminating null byte.

The Prolog declarations for this external predicate are :

```
:- extern_load ( [ strcpy ] , [ 'strcpy.o' ] ) .
:- extern_predicate ( strcpy ( string : m , string : i ) ) .
```

As first argument, a *mutable string* is declared. As a result, the argument will be passed with no extra reference, as would be the case for an *output* mode. Calling the predicate with a free first argument, will pass the external routine a pointer to a character array that is big enough to hold the largest possible Prolog atom.

```
?- strcpy ( _String , 'Input String' ) .
_String = Input String
```

Array parameters

An external predicate is defined that calculates the vector product of two vectors with real coordinates.

In C, the code for the routine *vector_product()*, in file *vector.c* is :

```
double vector_product ( x , y , n )
double * x, * y;
int n;
{
    int count;
    double product;

    product = 0.0;
    count = n;
    while ( count-- ) product += ( *x++ ) * ( *y++ );
    return ( product );
}
```

An extra argument indicates the sizes of the vectors, as this cannot be determined from the vector itself.

The Prolog declarations to use this, are :

```
:- extern_load ( [vector_product ] , [ 'vector.o' ] ) .
:- extern_predicate ( vector_product (real:r , real:array:i , real:array:i , integer:i) ) .
```

A possible call :

```
?- vector_product ( _P , v ( 1.0 , 2.0 ) , v ( 3.0 , 4.0 ) , 2 ) .
_P = 11.0
```

The external predicate is called with two terms of arity 2. The number of elements, 2 is also passed.

Backtracking external predicate

The next example illustrates the general framework for backtracking external predicates with a simple iterator over an external table. In the external program, a table with names is managed. The interface to this program provides predicates for iterating from Prolog over the table, to retrieve all entries that match a given pattern.

C code for the iterator routines, stored in a file *iterator.c* :

```
#define TABLE_ITERATOR "TABLE_ITERATOR"

typedef struct {
    char pattern [ MAX_PATTERN ];
    int index;
} IteratorRecord, * Iterator;

int table_iterate_start ( pattern , iterator , identifier )
char * pattern;
Iterator * iterator;
char ** identifier;
{
    Iterator iter;

    iter = ( Iterator ) malloc ( sizeof ( IteratorRecord ) );
    if ( ! iter ) return ( 1 );

    iter -> index = 0;
    strcpy ( iter -> pattern , pattern );
    * iterator = iter;
    * identifier = TABLE_ITERATOR;
    return ( 0 );
}

int table_iterate_next ( iterator , name )
Iterator iterator;
char ** name;
{
    int index;

    index = match_pattern_from ( iterator -> pattern , iterator -> index );
    if ( index )
    {
        *name = get_table_name( index );
        iterator -> index = index;
        return ( 0 );
    }
    else
    {
        BIM_Prolog_rm_mrepeat ( TABLE_ITERATOR );
        free ( iterator );
```

```
            return ( 1 );
        }
    }
```

An iterator externally consists of a record that holds the current index in the table and the pattern that must be matched for this iterator. Each new invocation of an iterator allocates a new iterator record. The address of this record is returned as choicepoint related information. It must be used to retrieve the next solution. When an iterator is invoked, with *table_iterate_start()*, a choicepoint identifier is also passed to the calling Prolog predicate. In general, the Prolog part as well as the external part can decide on the identifier. If the allocation of a new iterator is impossible, the routine returns the error code 1, otherwise 0.

The next solution routine *table_iterate_next()* searches for the next table entry that matches the pattern in the specified iterator. If one is found, the iterator is adapted, and the name is returned. The function result is 0 in this case. If no matching entry can be found anymore, the most recent table iterator choicepoint is removed, the iterator is deallocated and the function returns the error code 1.

In Prolog, the following declarations must be provided together with some wrapping code that uses the external routines to make a real Prolog backtracking predicate *table_pattern_match/2*.

```
:- extern_load ( [ table_iterate_start , table_iterate_next ] , [ 'iterator.o' ] ) .
:- extern_predicate ( table_iterate_start ( integer:r , string:i , pointer:o , string:o ) ) .
:- extern_predicate ( table_iterate_next ( integer:r , pointer:i , string:o ) ) .

table_pattern_match ( _Pattern , _Match ) :-
    table_iterate_start ( 0 , _Pattern , _Iterator , _Identifier ),
    mark_repeat ( _Iterator , _Identifier ),
    table_iterate_next ( 0 , _Iterator , _Match ) .
```

Calling this predicate with first argument instantiated to a pattern, will return all matching names in the second argument, one at a time by backtracking.

ProLog by BIM - Reference Manual
External Language Interface
Chapter 3

Calling Prolog Predicates from C

3.1 Access to Prolog predicates

It is possible to call predicates that are defined in ***ProLog*** from an external routine. To do this, the predicate's name and arity must be known. With this information, the predicate's handle can be obtained. The predicate can be called using this handle and passing the desired parameters.

The following functions can be used to retrieve the handle of a predicate.

BP_Atom **BIM_Prolog_string_to_atom** (*protect* , *string*)
int *protect*;
BP_String *string*;

The null-terminated character array *string* is converted to a ***ProLog*** atom,which is returned in its internal representation. That atom is necessary to retrieve the predicate's handle. The *protect* flag may be FALSE for this purpose, as the atom is only needed to retrieve the handle and may be destroyed afterwards.

BP_Functor **BIM_Prolog_get_predicate** (*name* , *arity*)
BP_Atom *name*;
int *arity*;

The handle for the predicate with name *name* and arity *arity*, is returned. The *name* argument must be an atom (in its internal representation). If the predicate does not exist at the moment of the call, its future handle is returned. So, it is perfectly possible to search the handle of a predicate that is not yet defined. It must only exist at the moment it is called.

To inquire the name and arity from a predicate handle, the following function can be used.

int **BIM_Prolog_get_name_arity** (*functor* , *atom* , *arity*)
BP_Functor *functor*;
BP_Atom * *atom*;
int * *arity*;

The name of predicate handle *functor* is stored in *atom*, and its arity in *arity*.

If *functor* is not a legal functor, an error message is issued and the function returns FALSE. Otherwise it returns TRUE.

3.2 Calling Prolog predicates

There are two ways of calling Prolog predicates: one way returns only one solution; the other returns multiple solutions. They are used, respectively, for deterministic and non-deterministic predicates. Nevertheless, searching for only one solution of a non-deterministic predicate and, conversely, searching for different solutions of a deterministic predicate is allowed.

Single solution call (deterministic call)

The following function is used to find one single solution of a predicate with a known handle:

int **BIM_Prolog_call_predicate** (*functor* { , *spec* [, *size*] [, *value*] })
BP_Functor *functor*;
int *spec*;
int *size*;
union *value*;

The Prolog predicate with handle *functor* is called, and its first solution is retrieved. The function returns TRUE if the call succeeded and a solution is retrieved. It returns FALSE if an error occurred or if there is no solution.

For each argument of the predicate a parameter descriptor must be passed to this routine. This consists of a sequence of up to three arguments, including a specifier *spec*, in certain cases a size *size* and in most cases a value *value*. The specifier is a combination of mode, structure and type specification for the argument. The optional size is a further specification for certain types of parameter passing mode. (See section *Parameter mapping declarations* for detailed information on the parameter descriptor).

After calling the predicate and copying the output argument values, an implicit cut and fail is performed. As a result, all unifications that were done during this call, are undone. This is particularly important for *BP_T_BPTERM* arguments : these will never be further instantiated with this routine. To avoid this behavior, an iterative call should be made instead of a deterministic one.

Multiple solution call (iterative call)

If more than one solution of a predicate is required, an iterator must be used. There are three predicates for this purpose : one to set up the iterator, one to perform a next iteration step, and one to terminate the iteration.

int **BIM_Prolog_setup_call** (*functor* { , *spec* , [*size*] [, *value*] })
BP_Functor *functor*;
int *spec*;
int *size*;
union *value*;

An iterator is set up for calling the predicate with handle *functor*, and for iterating over its solutions.

If *functor* is not a legal functor, or if it was impossible to set up another iterator, or if the argument descriptions were erroneous, an error message is issued and the function returns FALSE. Otherwise it returns TRUE.

This function is analoguous to **BIM_Prolog_call_predicate**() except that it does not call the predicate and thus, no solution is retrieved.

A new iterator may be set up while other iterators are still active.

One has to be careful that the locations for output arguments, whose addresses are passed as value for the argument, remain in effect throughout the whole iteration cycle over the predicate. In particular, they should not be addresses of local variables in a routine that is left after the iterator has been set up.

int **BIM_Prolog_next_call** ( )

The next solution of the most recently created iterator is retrieved. The function returns TRUE if a new solution was found, and FALSE if no more solutions could be found.

Before searching the next solution, an implicit fail is performed, thereby undoing all unifications since the previous iteration. Unlike the routine **BIM_Prolog_call_predicate**(), no cut and fail is performed after the solution is found. This means a *BP_T_BPTERM* argument can be further instantiated during an iteration.

If no iterator is active, an error message is issued and the function returns FALSE.

int **BIM_Prolog_terminate_call** ( )

The most recently created iterator is terminated and destroyed.

An implicit cut and fail is performed, undoing all unifications since the iterator was set up.

Printing error messages

When an external routine must give an error message, it can be printed directly on the error stream (*stderr*), but it can also go via the ***ProLog*** error printing routine. This way, it will be compatible with the rest of the application where it concerns the warning switch and the redirection of the error stream.

BIM_Prolog_error_message (*message*)
char * *message*;

The error message with text *message* is rendered on the current error stream, if the warning switch is on.

3.3 Parameter mapping declarations

For each argument of a Prolog predicate called from an external routine, a parameter descriptor must be passed from the external program to the interface. This descriptor is a sequence, consisting of a specifier, in certain cases a size and in most cases a value. A specifier is an integer that has information about type, structure and mode of the parameter encoded. The size argument is only required (and allowed) for certain combinations of type and structure. The value argument is either the value of the parameter for a input, or the address of an external variable for an output parameter.

The specifier is composed of three values, one for the mode, one for the structure and one for the type, by or'ing them together or taking their sum. The values must be chosen from the tables below. To use these values, the external interface definition file *BPextern.h* must be included.

Types

A table of available type specifiers is given below together with the corresponding C data type.

Table : argument types and corresponding C data types

Type	C Data type	Description
BP_T_INTEGER	int	Long integer (4 byte)
BP_T_LONG	long	Long integer (4 byte)
BP_T_SHORT	short	Short integer (2 byte)
BP_T_REAL	double	Long real (8 byte)
BP_T_DOUBLE	double	Long real (8 byte)
BP_T_FLOAT	float	Short real (4 byte)
BP_T_POINTER	BP_Pointer	Pointer (4 byte)
BP_T_ATOM	BP_Atom	Atom identifier (internal form)
BP_T_STRING	BP_String	String (null-terminated)
BP_T_STRINGS	BP_String	String (fixed size)
BP_T_BPTERM	BP_Term	Term pointer (internal form)
BP_T_UNTYPED		Run-time actual type
BP_T_VOID		Ignore predicate argument

Type *BP_T_STRING* is a null-terminated character array. Type *BP_T_STRINGS* is a fixed size array of characters. The size specifier determines the number of characters in the string.

For *BP_T_UNTYPED* the type of the parameter is determined by the actual Prolog parameter. This type can only be used for output parameters because it is impossible to determine the type of an external variable at run-time.

With *BP_T_VOID* it is possible to ignore a parameter of a Prolog predicate. The predicate is called with a void variable at the corresponding argument place. And nothing is retrieved from it after the call returns.

Structures

There are two structure specifiers : *BP_S_SIMPLE* and *BP_S_ARRAY*.

A *BP_S_SIMPLE* parameter is an unstructured simple argument.

A *BP_S_ARRAY* parameter is at the external side, an array, and at the Prolog side a Prolog list or a flat term. The size specifier indicates the number of elements in the array. In general, if the array is passed from the external routine to Prolog, the size must also be passed in the same direction, and if the array is passed from Prolog back to the external routine, the size will also be passed back from Prolog.

Modes

Parameters can be passed in three different modes between an external routine and a Prolog predicate : *BP_M_IN*, *BP_M_OUT* and *BP_M_MUTE*.

Input mode *BP_M_IN* is used to pass a value from the external routine to the Prolog predicate.

An *output* parameter (*BP_M_OUT*) is used to retrieve a value from a Prolog predicate. The value part of the parameter descriptor must be the address of the external variable that will receive the value of the corresponding Prolog argument.

The *mutable* mode *BP_M_MUTE* is a variant of the *output* mode. It is also used to return a value from a Prolog predicate. The value part of the parameter descriptor must be the address of the external variable that will receive the value of the corresponding Prolog argument.

The difference between *output* and *mutable* is that for *output*, the external interface will provide the necessary memory for the resulting data, while for *mutable* parameters, it is the responsibilty of the external routine to provide all necessary space. As a consequence, an *output* parameter has one level more of indirection for complex parameters, like strings and arrays.

Restrictions

Not all combinations of type, structure and mode are allowed. The following restrictions hold :

- Type *BP_T_STRINGS* is not allowed in structure *BP_S_ARRAY*.
- Type *BP_T_UNTYPED* is not allowed in mode *BP_M_IN*.
- Type *BP_T_BPTERM* is the same in all modes, and has the effect of mode *BP_M_IN*.
- Type *BP_T_VOID* is the same in all modes, and has the effect of mode *BP_M_IN*.

Overview of combinations that are not allowed :

BP_T_STRINGS | *BP_S_ARRAY* | *
BP_T_UNTYPED | * | *BP_M_IN*

3.4 General parameter passing rules

The global parameter passing rules are described by specifying the actions that are performed at *call* (when going from the external routine to Prolog), and at *exit* (when coming back from Prolog to the external routine). These are given for each of the modes.

input - call
: The actual parameter is type converted from the external type to the Prolog type as described in the specifier. This converted value is passed to the Prolog predicate.

input - exit
: No actions are performed.

output - call
: No actions are performed.

output - exit
: The actual Prolog parameter value is type converted from the Prolog type to the external type as described in the specifier. That value is stored at the location whose address was given in the parameter descriptor.

mutable - call
: No actions are performed.

mutable - exit
: The actual Prolog parameter value is type converted from the Prolog type to the external type as described in the specifier. That value is stored at the location whose address was given in the parameter descriptor.

3.5 Parameter passing specification

Detailed specifications of parameter passing for each language, type and mode are given by means of pictures, representing the parameter structures at *call* and *exit* time. A basic memory cell of 4 bytes is represented by a 1/2 inch wide box.

Boxes with a thick border, represent the data that is actually passed as parameter (i.e. that is pushed on the stack or in the output registers).

A ? in a box, means its value is undefined.

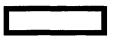
Shaded boxes at the *exit*, indicate zones that may have been modified by the Prolog predicate.

Boxes that exist only at the *exit*, are zones that are managed by the external language interface. There are two types of such zones : short-term and long-term memory. The short-term zones contain valid data only as long as control stays in the external routines. From the moment control goes back to ***ProLog*** , in whichever way, these zones may be destroyed or overwritten. As a result, if the external routine needs the data for a longer period, it has to copy it. These short-term zones should not be overwritten by the external routines. They are indicated as 'short-term zone'. The long-term memory zones can be used freely by the external routines, and should be deallocated when no longer needed. ***ProLog*** will not deallocate these zones automatically.

Simple input

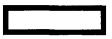
BP_T_INTEGER / BP_T_LONG | BP_S_SIMPLE | BP_M_IN

call



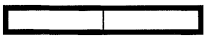
BP_T_SHORT | BP_S_SIMPLE | BP_M_IN

call



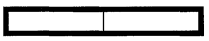
BP_T_FLOAT | BP_S_SIMPLE | BP_M_IN

call



BP_T_REAL / BP_T_DOUBLE | BP_S_SIMPLE | BP_M_IN

call

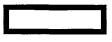


BP_T_POINTER | BP_S_SIMPLE | BP_M_IN

call

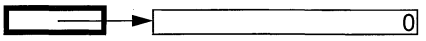
BP_T_ATOM | BP_S_SIMPLE | BP_M_IN

call



BP_T_STRING | BP_S_SIMPLE | BP_M_IN

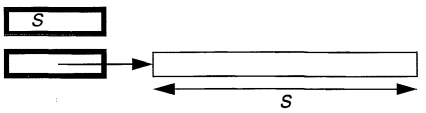
call



The text zone must hold the actual string and a terminating 0 byte.

BP_T_STRINGS / BP_S_SIMPLE | BP_M_IN

call

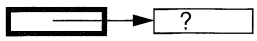


The first s bytes of the text zone are considered to be the string.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

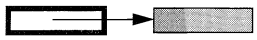
Simple output

BP_T_INTEGER / BP_T_LONG | BP_S_SIMPLE | BP_M_OUT

call



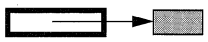
exit



BP_T_SHORT | BP_S_SIMPLE | BP_M_OUT

call

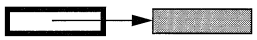
exit



BP_T_FLOAT | BP_S_SIMPLE | BP_M_OUT

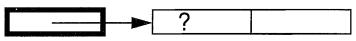
call

exit



BP_T_REAL / BP_T_DOUBLE | BP_S_SIMPLE | BP_M_OUT

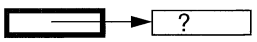
call



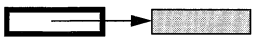
exit

BP_T_POINTER | BP_S_SIMPLE | BP_M_OUT

call

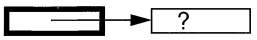


exit



BP_T_ATOM | BP_S_SIMPLE | BP_M_OUT

call



exit

BP_T_STRING | BP_S_SIMPLE | BP_M_OUT

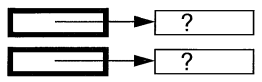
call

exit

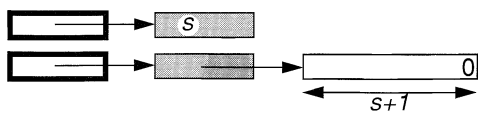
The text zone contains the string terminated with a 0 byte. It is short-term memory and should not be overwritten.

BP_T_STRINGS / BP_S_SIMPLE | BP_M_OUT

call



exit

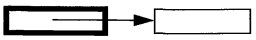


The text zone contains the string terminated with a 0 byte. It is short-term memory and should not be overwritten.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

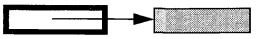
Simple mutable

BP_T_INTEGER / BP_T_LONG | BP_S_SIMPLE | BP_M_MUTE

call



exit



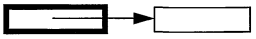
BP_T_SHORT | BP_S_SIMPLE | BP_M_MUTE

call

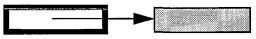
exit

BP_T_FLOAT | BP_S_SIMPLE | BP_M_MUTE

call

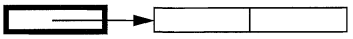


exit



BP_T_REAL / BP_T_DOUBLE | BP_S_SIMPLE | BP_M_MUTE

call

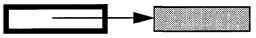


exit

BP_T_POINTER | BP_S_SIMPLE | BP_M_MUTE

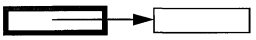
call

exit

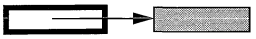


BP_T_ATOM | BP_S_SIMPLE | BP_M_MUTE

call

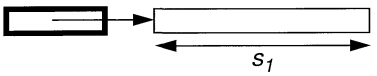


exit



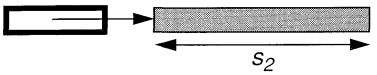
BP_T_STRING | BP_S_SIMPLE | BP_M_MUTE

call



Default for s : MAX_ATOM. The text zone must be long enough to hold the string.

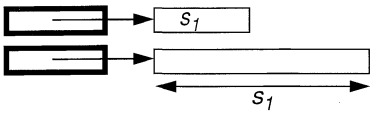
exit



The text zone contains the string, terminated with a 0 byte. And $s_2 \leq s_1$.

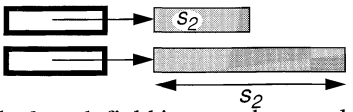
BP_T_STRINGS / BP_S_SIMPLE | BP_M_MUTE

call



The text zone must be at least s bytes long.

exit



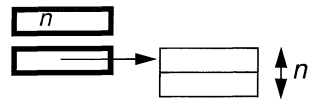
The length field is set to the actual length of the text zone that has been filled. And $s_2 \leq s_1$. If there is enough place left, the string is terminated with a 0 byte.

Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

Array input

BP_T_INTEGER / BP_T_LONG | BP_S_ARRAY | BP_M_IN

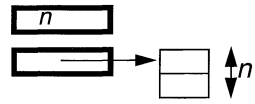
call



Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_SHORT | BP_S_ARRAY | BP_M_IN

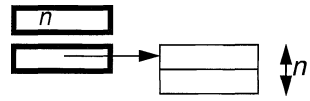
call



Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_FLOAT | BP_S_ARRAY | BP_M_IN

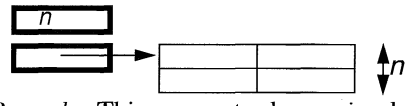
call



Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_REAL / BP_T_DOUBLE | BP_S_ARRAY | BP_M_IN

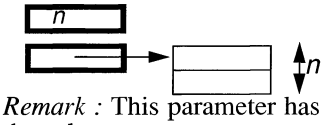
call



Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_POINTER | BP_S_ARRAY | BP_M_IN

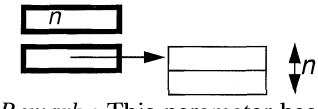
call



Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_ATOM | BP_S_ARRAY | BP_M_IN

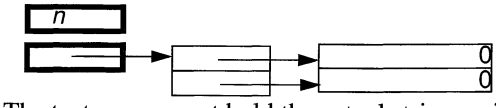
call



Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRING | BP_S_ARRAY | BP_M_IN

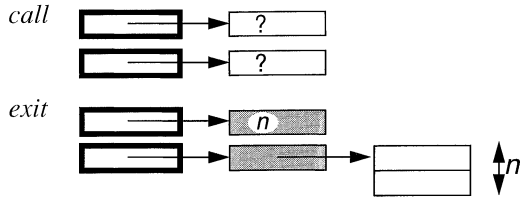
call



The text zones must hold the actual strings with a terminating 0 byte.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

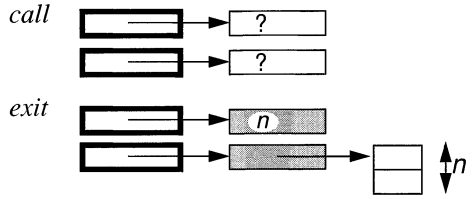
Array output

BP_T_INTEGER / BP_T_LONG | BP_S_ARRAY | BP_M_OUT



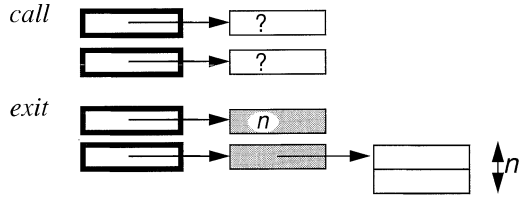
The array is long-term memory and must be freed when no longer needed.
Remark : This parameter has a *size* descriptor argument, immediately prceding the value argument.

BP_T_SHORT | BP_S_ARRAY | BP_M_OUT



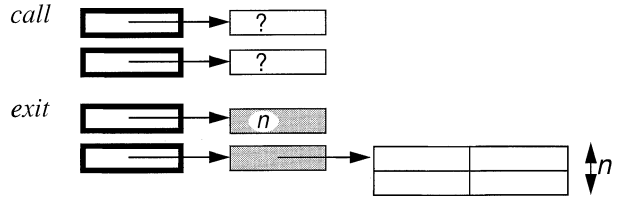
The array is long-term memory and must be freed when no longer needed.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_FLOAT | BP_S_ARRAY | BP_M_OUT



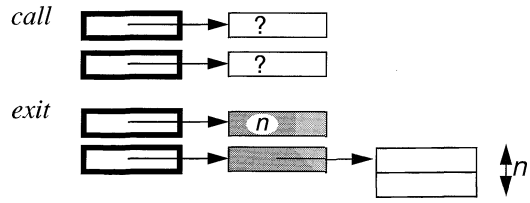
The array is long-term memory and must be freed when no longer needed.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_REAL / BP_T_DOUBLE | BP_S_ARRAY | BP_M_OUT



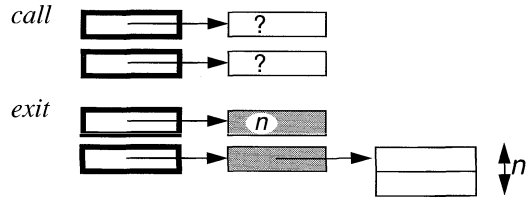
The array is long-term memory and must be freed when no longer needed.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_POINTER | BP_S_ARRAY | BP_M_OUT



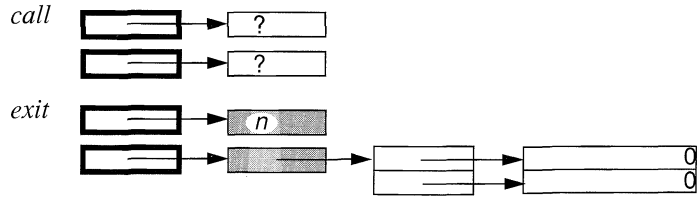
The array is long-term memory and must be freed when no longer needed.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_ATOM | BP_S_ARRAY | BP_M_OUT



The array is long-term memory and must be freed when no longer needed.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRING | BP_S_ARRAY | BP_M_OUT

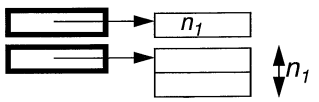


The text zones contain the strings with a terminating 0 byte. They are short-term memory and should not be overwritten. The array is long-term memory and must be freed when no longer needed.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

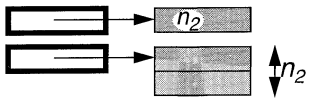
Array mutable

BP_T_INTEGER / BP_T_LONG | BP_S_ARRAY | BP_M_MUTE

call



exit

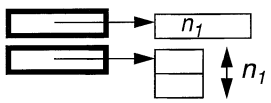


The length is set to the actual returned array length, with $n_2 \leq n_1$.

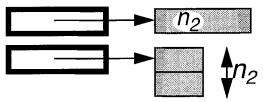
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_SHORT | BP_S_ARRAY | BP_M_MUTE

call



exit

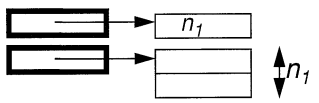


The length is set to the actual returned array length, with $n_2 \leq n_1$.

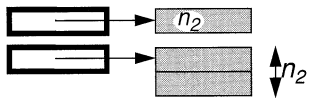
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_FLOAT | BP_S_ARRAY | BP_M_MUTE

call



exit

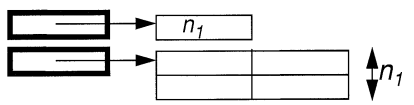


The length is set to the actual returned array length, with $n_2 \leq n_1$.

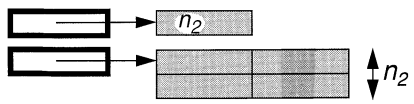
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_REAL / BP_T_DOUBLE | BP_S_ARRAY | BP_M_MUTE

call



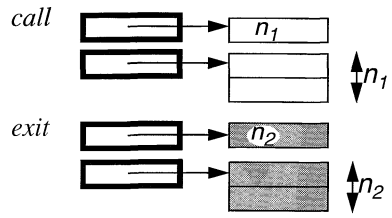
exit



The length is set to the actual returned array length, with $n_2 \leq n_1$.

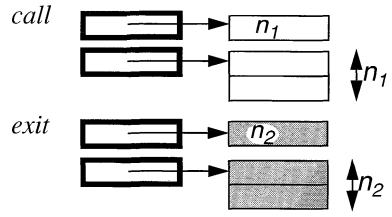
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_POINTER | BP_S_ARRAY | BP_M_MUTE



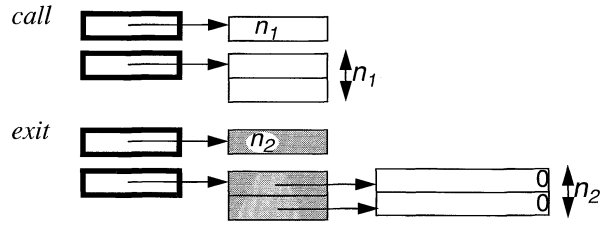
The length is set to the actual returned array length, with $n_2 \leq n_1$.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_ATOM | BP_S_ARRAY | BP_M_MUTE



The length is set to the actual returned array length, with $n_2 \leq n_1$.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRING | BP_S_ARRAY | BP_M_MUTE



The length is set to the actual returned array length, with $n_2 \leq n_1$. The text zones contain the strings with a terminating 0 byte. They are temporary and should not be overwritten.
Remark : This parameter has a *size* descriptor argument, immediately preceding the value argument.

ProLog by BIM - Reference Manual
External Language Interface
Chapter 4

External Manipulation of Prolog Terms

4.1 Representation of terms

A Prolog term can be passed between a Prolog predicate and an external routine, in both directions. This is accomplished by indicating *bpterm* as argument type in a declaration of the external routine, or by using the type *BP_T_BPTERM* when calling a Prolog predicate from an external routine.

The corresponding value of a term in the external routine, is an opaque handle that represents the term on the system's heap. The data type for such a handle is *Term*, which is defined in the external language interface include file *BPextern.h*. The term should only be manipulated, using the external ***ProLog*** routines. Relying on other features is implementation dependent and may break your program in future releases.

The general form of a term can be defined recursively :

<*term*> ⇒ <*variable*> | <*simple term*> | <*structured term*>
<*simple term*> ⇒ <*integer*> | <*real*> | <*atom*> | <*pointer*>
<*structured term*> ⇒ <*structure*> | <*list*>
<*structure*> ⇒ <*functor*> (<*term*>) *
<*list*> ⇒ <*term*> <*term*>

A term is either a variable, corresponding to an uninstantiated Prolog variable, or a simple or a structured term. A simple term is a Prolog integer, real, atom or pointer. All other Prolog terms are structured terms. They consist of a functor followed by one or more arguments which are again terms. The number of arguments is the same as the arity of the functor. A special case of structured term is the list, which has two arguments, and whose functor is always **./2**.

In order to manipulate terms and sub-terms, a number of types are defined. These allow external routines to distinguish between different kinds of terms, and to map the values to external variables of the right type. For certain types of terms, the value can be represented externally in different ways. Therefore, a number of type variants are defined. ***ProLog*** always returns a base type when the type of a term is requested. External routines, however, may use the variant types, when retrieving or setting term values.

Table : base types with their variant types

Base Type	Variant Types
BP_T_INTEGER	BP_T_LONG BP_T_SHORT
BP_T_REAL	BP_T_DOUBLE BP_T_FLOAT
BP_T_ATOM	BP_T_STRING

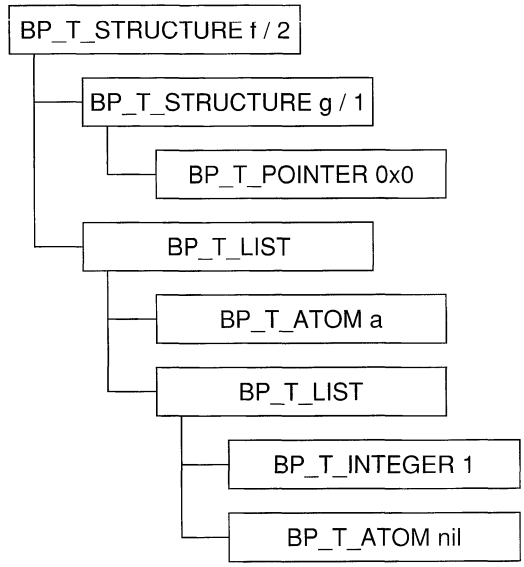
The table below gives a list of all defined term types (base and variants), with the corresponding C data type and a description of the value of a term of that type.

Table : term types and corresponding C types

Term type	C Data type	Value
BP_T_INTEGER	int	Integer
BP_T_LONG	long	Long integer
BP_T_SHORT	short	Short integer
BP_T_REAL	double	Real
BP_T_DOUBLE	double	Long real
BP_T_FLOAT	float	Short real
BP_T_POINTER	BP_Pointer	Pointer
BP_T_ATOM	BP_Atom	Atom (internal form)
BP_T_STRING	BP_String	String (character array)
BP_T_LIST	BP_Functor	List functor ./2 (internal form)
BP_T_STRUCTURE	BP_Functor	Term functor (internal form)
BP_T_VARIABLE	int	Variable number

Remark : The number of a variable is the same number that is used when it is written out. This means that it may change during the program's life (particularly after garbage collection of the heap).

Example : representation of the term f (g (0x0) , [a , 1])



4.2 Term decomposition

The standard method to decompose a term, is first ask its type, and then retrieve its value, of the returned type or one of this type's variants.

Retrieving the type of a term

int **BIM_Prolog_get_term_type** (*term*)
BP_Term *term*;

The type of the term *term* is returned. This is one of the defined base types.

If it is not a legal term, an error message is issued and the special value BP_T_ILLEGAL is returned.

Retrieving the value of a term

int **BIM_Prolog_get_term_value** (*term* , *type* , *addr_value*)
BP_Term *term*;
int *type*;
union * *addr_value*;

The value of a term *term* is retrieved and stored at the address given by *addr_value*.
The *type* determines how it must be returned. This may differ from the type returned by **BIM_Prolog_get_term_type**(). It can also be one of its variant types.
The address passed as *addr_value*, must be the address of a variable with a type that corresponds to the indicated type *type*.

If *term* is not a legal term, or if *type* is an illegal type, or a type that is not conform with *term*, an error message is issued and FALSE is returned. Otherwise, TRUE is returned.

Remark : when requesting a BP_T_STRING, the *addr_value* parameter must be the address of a pointer variable. This variable will be set to a pointer to the textual representation of the atom. This is a null-terminated array of characters that reside in the system's memory, and therefore should not be overwritten. It is not ensured that this string will remain at the same position during the whole program life time. As a result, if the text is needed by the external routines for a longer period, it should be copied to external private memory.

Retrieving an argument of a term

int **BIM_Prolog_get_term_arg** (*term* , *argnr* , *arg*)
BP_Term *term*;
int *argnr*;
BP_Term * *arg*;

The *argnr*'th argument of term *term* is retrieved and stored at the address given by *arg*. If *term* is not a legal term or if *argnr* is greater than the arity of the term *term*, an error message is issued and the function returns FALSE. Otherwise it returns TRUE.

4.3 Term construction

A term can be constructed in the external routines in mainly two ways : by creating a new term, or by instantiating an existing term.

It is important to understand that during construction of a term, the heap may become exhausted causing the garbage collector to be invoked. If this happens, previously constructed terms may be moved to another place or may even be destroyed. In this case, the term handle becomes a dangling reference. To prevent this, one can either ensure that there is enough space left on the heap, before starting the construction of the term, or already created terms can be protected against destruction or movement during garbage collection. (See section *Life time of terms* for protecting terms).

Ensuring heap space for constructing a term

int **BIM_Prolog_term_space** (*size*)
int *size*;

If the heap has not enough space left for a term of *size* heap cells, the heap garbage collector is activated on the spot. If there is still not enough space left (after possible heap expansions), the function returns FALSE. Otherwise it returns TRUE.

Calculating the size of a term is straightforward. A basic element takes one cell. A list takes two cells per element : one for the head (which contains the element), and one for the tail (which is either again a list, or the atom nil). A structure takes as much cells as its arity plus one.

Creating a new term

BP_Term **BIM_Prolog_new_term** ( )

A new term is created on the heap and returned as the function result. This new term is a free variable.

This function may invoke the heap garbage collector.

If there was not enough space for creating the new term, 0 is returned.

Unifying a term to a value

int **BIM_Prolog_unify_term_value** (*term* , *type* [, *value*])
BP_Term *term*;
int *type*;
union *value*;

The term *term* is unified with a value *value* of type *type*. If unification succeeded, the function returns TRUE. Otherwise it returns FALSE and *term* is left unchanged.
The type of the argument *value*, must correspond to the one mentioned as *type*. (See section *Representation of terms* for a correspondance table.)
For type BP_T_LIST, no *value* is required.

If *term* is not a legal term, an error message is issued and the function returns FALSE.

Unification is done along the following rules :

- If *type* is BP_T_VARIABLE, and *value* is not the number of a free variable, unification fails.
- If *type* and *value* represent a free variable, it is instantiated to *term*.
- If the base type of *term* is different from the base type of *type*, unification fails.
- If *term* is (partially) instantiated, it is unified with *value* in the standard Prolog way of unification.
- A free *term* that is instantiated to a list or a structure will have its arguments free variables.

Unifying two terms

int **BIM_Prolog_unify_terms** (*term1* , *term2*)
BP_Term *term1*;
BP_Term *term2*;

The two terms *term1* and *term2* are unified. If unification succeeded, the function returns TRUE. Otherwise it returns FALSE.

If *term1* or *term2* is not a legal term, an error message is issued and the function returns FALSE.

Unification is done in the standard Prolog way.

4.4 Life time of terms

Terms that are passed in a call of an external routine will exist as long as that call is active (or longer), and will not be destroyed by garbage collection. Further instantiations of the term will also remain as long as the term itself exists, and no backtracking occurs.

Externally created terms can be moved on the heap or completely destroyed during garbage collection of the heap, which may occur during any construction of a term.

Any term or sub-term can be moved when garbage collection of the heap occurs. As a result, external variables that are term handles (data type *Term*), are potential dangling references after garbage collection.

To prevent terms from being moved or destroyed, they can be protected with the external routines described below. A better way to avoid this kind of problems, is to check that there is enough space left on the heap before constructing a (large) term (see section *Term construction*).

Protecting a term

BP_Term **BIM_Prolog_protect_term** (*term*)
BP_Term *term*;

The term *term* is protected against destruction or movement during garbage collection. Its protected version is returned.

If *term* is not a legal term, an error message is issued and the function returns 0.

Protecting an already protected term, has no effect : the term itself is returned.

Unprotecting a term

BP_Term **BIM_Prolog_unprotect_term** (*term*)
BP_Term *term*;

The term *term* is unprotected against destruction or movement during garbage collection. Its unprotected version is returned.

If *term* is not a legal term, an error message is issued and the function returns 0.

Unprotecting a non-protected term, has no effect : the term itself is returned.

4.5 Type conversion of simple terms

A number of external functions is provided for converting between different representations of ***ProLog*** internal data.

Conversion from string to atom

BP_Atom **BIM_Prolog_string_to_atom** ([*protect* ,] *string*)
int *protect*;
BP_String *string*;

The null-terminated character array *string* is converted to a ***ProLog*** atom and its internal representation is returned.
If *protect* is TRUE, the atom will be protected against destruction in garbage collection and will remain permanently in the data tables of ***ProLog***. If *protect* is FALSE, the atom will be a temporary atom. As a result, it may be destroyed in garbage collection, if there is no reference to it from a term known by ***ProLog***. If *protect* is omitted, it defaults to TRUE.

It is recommended to avoid making protected atoms, as this irreversibly fills up the data tables. Only when the atom must exist for the rest of the program's life, it should be made protected.

After having made an atom from the string, the character array pointed to by *string* is no longer needed by ***ProLog***.

Conversion from sized string to atom

BP_Atom **BIM_Prolog_strings_to_atom** ([*protect* ,] *string* , *length*)
int *protect*;
BP_String *string*;
int *length*;

The character array *string* of *length* bytes, is converted to a ***ProLog*** atom and its internal representation is returned.
The meaning of *protect* is the same as in **BIM_Prolog_string_to_atom**().

Conversion from atom to string

BP_String **BIM_Prolog_atom_to_string** (*atom*)
BP_Atom *atom*;

The textual representation of the atom *atom* is returned. This is a pointer to a character array. That array should not be overwritten, and it is not guaranteed to contain the same string during the whole life of the program (especially not after garbage collection of the data tables). If the string is needed for a longer time in the external routine, it should be copied.

If *atom* is not a legal atom, an error message is issued and the function returns 0.

Saving a string

BP_String **BIM_Prolog_save_string** (*string*)
BP_String *string*;

The null-terminated character array *string* is saved in the string tables of ***ProLog***, and the saved version is returned. The array that is returned, should not be overwritten. It is guaranteed that it will always contain the same string.

If there is not enough space left in the string tables to save the string, an error message is issued and 0 is returned.

This function is an easy and efficient replacement for malloc(). However, it should not be used for temporary saves, as the saved string remains permanently in ***ProLog***'s string tables.

4.6 Examples

Term decomposition

The C routine *print_term()*, defined below, prints out the term it receives from ***ProLog***, in the same style as the builtin predicate **write/1**.

To use it from ***ProLog***, the following declaration is needed :

```
:- extern_predicate ( print_term ( bpterm ) ) .
```

The C code for *print_term()* :

```
#include <BPextern.h>

print_term ( term )
BP_Term term;
{
    int type, nr;
    BP_Atom atom_nil;
    BP_Term arg;
    int int_val;
    int * ptr_val;
    double real_val;
    BP_Atom atom_val;
    BP_Functor struct_val;

    atom_nil = BIM_Prolog_string_to_atom( "nil" );
    type = BIM_Prolog_get_term_type( term );

    switch ( type )
    {
        case BP_T_INTEGER :
            BIM_Prolog_get_term_value ( term , type , &int_val );
            printf ( "%d" , int_val );
            break;

        case BP_T_REAL :
            BIM_Prolog_get_term_value ( term , type , &real_val );
            printf ( "%f" , real_val );
            break;

        case BP_T_POINTER :
            BIM_Prolog_get_term_value ( term , type , &ptr_val );
            printf ( "0x%x" , ptr_val );
            break;

        case BP_T_ATOM :
            BIM_Prolog_get_term_value ( term , type , &atom_val );
            printf ( "%s" , BIM_Prolog_atom_to_string ( atom_val ) );
            break;
```

```
case BP_T_VARIABLE :
    BIM_Prolog_get_term_value ( term , type , &int_val );
    printf ( "_%d" , int_val );
    break;

case BP_T_STRUCTURE :
    BIM_Prolog_get_term_value ( term , type , &struct_val );
    BIM_Prolog_get_name_arity (struct_val , &atom_val , &int_val );
    printf ( "%s(" , BIM_Prolog_atom_to_string ( atom_val ) );
    BIM_Prolog_get_term_arg ( term , 1 , &arg );

    print_term ( arg );
    for ( nr = 2 ; nr <= int_val ; nr++ )
    {
        printf ( "," );
        BIM_Prolog_get_term_arg ( term , nr , &arg );
        print_term ( arg );
    }
    printf ( ")" );
    break;

case BP_T_LIST :
    printf ( "[" );
    BIM_Prolog_get_term_arg ( term , 1 , &arg );
    print_term ( arg );
    BIM_Prolog_get_term_arg ( term , 2 , &arg );
    term = arg;

    while ( BIM_Prolog_get_term_type ( term ) == BPT_LIST )
    {
        printf ( "," );
        BIM_Prolog_get_term_arg ( term , 1 , &arg );
        print_term ( arg );
        BIM_Prolog_get_term_arg ( term , 2 , &arg );
        term = arg;
    }

    if ( BIM_Prolog_get_term_value ( term,BPT_ATOM,&atom_val )
        && atom_val == atom_nil )
        ;
    else
    {
        printf ( " | " );
        print_term ( term );
    }
    printf ( "]" );
    break;
```

```
            default :
                printf( "???" );
                break;
        }

    } /* print_term */
```

The routine distinguishes between the different base types and prints the simple terms in their specific format. Structured terms are further decomposed and their arguments are printed by recursively calling the print_term() routine.

Term construction

In this example, external routines are used to maintain a list of attribute settings. The routine *get_attribute()* takes a structure representing an attribute. It looks up the attribute name in its tables and instantiates the value if it is found. If no associated value is found, the value is unified with the default value.

An attribute is specified as :

attribute/3

attribute (_AttrName , _AttrValue , _AttrDefault)

arg1 : atom : attribute name
arg2 : any : attribute value
arg3 : any : attribute default value

*Arg*2 and *arg*3 are free if unknown.

The declaration to use the *get_attribute()* routine from ***ProLog*** is :

```
:- extern_predicate( get_attribute( bpterm ) ) .
```

The C code for the routine :

```
#include <BPextern.h>

static BP_Functor attribute_3;
/* initialized with
    attribute_3 = BIM_Prolog_get_predicate (
        BIM_Prolog_string_to_atom ( "attribute" ) , 3 ); */

get_attribute ( term )
BP_Term term;
{
    BP_Functor functor;
    BP_Atom attr_name;
    BP_Term arg1, arg2, arg3;
    int int_val;
    double real_val;
```

```
    if ( ! BIM_Prolog_get_term_value ( term,BPT_STRUCTURE,&functor )
        || functor != attribute_3 )
        goto get_attribute_exception;

    BIM_Prolog_get_term_arg ( term , 1 , &arg1 );
    if ( ! BIM_Prolog_get_term_value ( arg1,BPT_ATOM,&attr_name ) )
        goto get_attribute_exception;

    BIM_Prolog_get_term_arg ( term , 2 , &arg2 );

    if ( look_up_attribute_int ( attr_name , &int_val ) )
        BIM_Prolog_unify_term_value ( arg2 , BPT_INTEGER , int_val );
    else if ( look_up_attribute_real ( attr_name , &real_val ) )
        BIM_Prolog_unify_term_value ( arg2 , BPT_REAL , real_val );
    else /* return default value */
    {
        BIM_Prolog_get_term_arg ( term , 3 , &arg3 );
        BIM_Prolog_unify_terms ( arg2 , arg3 );
    }

    return;

get_attribute_exception :
    BIM_Prolog_error_message ( "get_attribute/1 : arg1 is not an attribute\n" );

} /* get_attribute */
```

No special arrangements must be made for term protection. The only term construction that is performed, is instantiation of (a part of) a term that is passed from Prolog.

DEBUGGER

Debugger

Contents

Introduction

For easy development of Prolog programs, the ***ProLog*** system includes an advanced programming environment. This programming environment is based on state-of-the-art Prolog debugging techniques and exploits all features of currently available multiwindowing and high resolution screen supporting workstations for an increased comfort and productivity of the ***ProLog*** programmer.

Apart from the conventional tracing package based on the box model, ***ProLog*** incorporates two novel debugging techniques. First, the tracing package has been extended with features for source-oriented debugging, improving the link between the execution monitoring and the original source produced by the user. Source line numbers and variable names are meaningful to the debugger and enhance its usability and effectiveness for debugging large and complex applications.

A second novel debugging technique is the post-mortem analysis. After the execution in debugging mode has produced an incorrect result or fails, the trace information can be analysed in an a-posteriori manner. The tracing information is displayed using the structure of the original program, providing a top-down, breadth-first view of the program execution instead of the sequential depth-first view of traditional Prolog debugging packages. An algorithmic debugging strategy has been implemented on top of this post-mortem debugging facility.

The multiwindowing programming environment integrates the ***ProLog*** system and its debugger with the text editor, and structures the various interaction modes in an ergonomic user interface. The input to and the output from the system are distributed logically among different subwindows, all monitored by a main interaction window. Each subwindow is an editing window in its own right and thus offers the standard editing functions, e.g. its contents can be saved for later inspection. Interaction with the system is not restricted to typing commands, but whenever possible appropriate buttons are provided, such that actions require as little manipulation as possible. The current ***ProLog*** session, the original source, error messages, data files, and all control panels are readily available and tighlty interlinked.

Experience with similar multiwindow programming environments for conventional procedural languages has shown a considerable positive impact on achieving a higher productivity in and quality of developing end-user applications.

ProLog by BIM - Reference Manual
Debugger
Chapter 1

Getting Started

1.1 Preparing a program for debugging

Before a predicate can be debugged, it must be compiled to debug code. The difference between dynamic or static code (which is generated by default) and debug code, is that the latter contains some additional information. This allows the system to trace the predicate while executing. The execution of debug code is therefore slower than the execution other codes.

There are several ways to compile a predicate to debug code. One is to compile the complete file containing the predicate with the '-Cd' option. The effect is that all predicates defined in the file, are compiled to debug code.

With the second method one has finer control over which predicates are to be debugged. It consists of placing directives around the predicates that must be compiled to debug code. The following directives are provided for this purpose :

Debugger directives

setdebug/0

Turns the compiler into debug code generation for the following predicates.

setnodebug/0

Turns the compiler into normal non-debug code generation for the following predicates.

option/1

If *arg1* equals 'd+' then this directive is equivalent to the setdebug/0 directive. If *arg1* equals 'd-' then this directive is equivalent to the setnodebug/0 directive.

Note that this predicate can also be called with other values for *arg1*. The complete explanation of **option/1** is given in the chapter on directives

To compile a complete file to debug code, using this method, it is sufficient to place a single **setdebug/0** directive at the beginning of the file. The corresponding **setnodebug/0** at the end of the file may be omitted.

All definitions of a predicate must be compiled in the same way. It is not allowed to compile some clauses of a predicate to debug code and the others to normal code.

A predicate that is compiled to debug code is treated as if made dynamic. This implies that it may be retracted or asserted.

Predicates that are bug-free do not have to be compiled to debug code, even when they are called from a debug coded predicate. The result is that the bug-free predicates will not be fully traced. This can however be confusing when following the execution or analysing the trace afterwards.

Finally a predicate can also be compiled to debug code by entering the predicate interactively. The interactive compiler however, produces dynamic code by default. Debug code can be obtained by using the builtin **please/2** : please (debugcode, on).
It is not allowed to enter clauses to be compiled to another type of code than the type of the previously entered (or consulted) clauses for the same predicate.

1.2 Overall debugger control

debug/1

debug(_Command)

arg1 : ground : atom

The atom *arg1* is treated as a debugger command and executed.

This enables debugger commands to be executed without entering the debugger. It only applies to the commands that make sense when no query is being solved.

This predicate is especially useful in your .pro file to set up your preferred debugging environment (using the alias and command commands).

debug/2

debug (_DebugOption, _Value)

arg1 : ground : atom
arg2 : ground or free

Arg1 is the name of a debugger option and *arg2* is its value. If *arg2* is free, it will be instantiated to the current value of the option. Otherwise the option's value is changed to *arg2*.

The table below gives the possible options and the allowed values. Default values are underlined.

c : cut	*on/<u>off</u>*	Controls the choice point destruction under debugger execution. When off, choice points are only marked when cut, upon backtracking this is mentioned as a 'fail due to cut'. When on, choice points are removed immediately, saving stack space.
p : prompt	*<u>on</u>/off*	Whether a command should be requested after activation of the debugger or consult.
td : tracedepth	*integer (default : -1)*	Allowed depth of the recorded trace.
tr : tracerecord	*<u>on</u>/off*	Recording of trace.
wd : writedepth	*integer (<u>-1</u>)*	Nesting depth for output of structured terms.
wm : writemodule	*on/<u>off</u>*	Provides module qualification in debugger output.
wp : writeprefix	*on/<u>off</u>*	Usage of prefix functor form in debugger output.
wq : writequotes	*on/<u>off</u>*	Usage of quotes in debugger output.

Environment commands

alias

alias <key> <string>

Defines <key> as an alias for <string>. Without argument, a list of currently defined aliases is printed.

command

command <cmd> <pred>[:<type>[:<modif>]]
command <cmd>
command

Defines a new command, named <cmd>. When issued, the predicate <pred> is called. If an argument type <type> was specified, it is passed to <pred>/1, otherwise <pred>/0 is called. The <type> specifier determines how the selection is to be interpreted to form the command argument.

Available argument types are :

literal : take literally
number : expand to an integer number
atom : expand to an atom
linenr : replace by the source file name and line number
filename : expand to a Unix path and file name
varname : expand to a variable name (leading _ is stripped)
predname : expand to a predicate name/arity

A *linenr* is returned as a list of three elements: the absolute path, the base name of the file and the line number.
A *filename* is returned as a list of two elements: the absolute path and the base name of the file.
The type can be modified with:

- *list* : the command can have several (at least one) arguments of the same type, separated by blanks or comma's
- *optional* : the argument is optional

A list of arguments is passed to the predicate as a Prolog list. If no actual argument is present for an optional argument, the empty list is passed as the argument.

Without arguments, command prints a list of user defined commands.

With only one argument, any existing definition for that command is destroyed.

For example:

If the following definitions are given :

```
command delete do_delete:number:list
command file do_file:filename
command print do_print:varname:list
```

The following commands invoke the predicate calls :

delete 1,2,3	**?-do_delete([1,2,3]).**
file abc	**?-do_file(['/path','abc']).**
print _x,_y	**?-do_print([x,y]).**

Multiple commands

The input to the debugger consists of a sequence of commands. Each command consists of a keyword possibly followed by a number of arguments. Multiple commands can be given on the same input line, separating them with ';'. The effect will be as if the commands were entered one by one each time the debugger prompts for a command. A sequence of commands can be redefined as one single command in combination with the alias command

For example:

To combine the advantages of seeing full trace information and source line indication you can use the command :

```
'creep ; show'.
```

To use this several times an alias can be defined as follows :

```
alias slow 'creep ; show'.
```

1.3 Output from the debugger

The main output of the debugger is a *trace* of the executed query. This trace consists of a single line for each port of the boxes that correspond to predicates that are compiled to debug code. Parts of it may be omitted, in the stepping debugger by using a command that skips certain ports and during trace recording by simply setting off the recording.

Each line in trace mode starts with a number indicating the box nesting level. The top-level box is at level 1 and all boxes that are nested deeper are at a higher level.

A line in trace mode which is recorded for post execution debugging, has an extra leading number giving the line number of the recorded trace; the starting line number is 1.

The next portion of the line is an indentation that reflects the nesting depth. It is cyclic (i.e. once the nesting is more than 16, the indentation restarts from the left.)

A symbol then follows, reflecting the kind of port that is displayed and the subgoal that is in execution at that port. The arguments of the displayed predicate are printed with the value that corresponds to their instantiation level at that port. At a failed unification port, the arguments are unified as far as possible. If for instance, the unification failed on the second argument of a predicate, then the first argument will be shown in its unified form and the others in their non-unified form.

If an error occurred between this port and the following one, the error message is attached to this line of information.

Writing out the arguments of a predicate is done to the same nesting depth as set by

?- debug (writedepth , _x) .

The symbols that identify the kind of port consist of one or two characters using the following basic rules :

- The first character indicates which of the five ports it is:
  - ? call port
  - > unify port
  - + exit port
  - - fail port
  - < redo port
- For builtin predicates, the second character is the same as the first.
- For user defined predicates or builtins defined in Prolog, a possible second character gives some extra information.

Complete summary of the different port indicator symbols :

?	call port
>	unify port
+	exit port
-	fail port
<	redo port
?*	call of non-debug code predicate
+*	exit of a fact
-~	fail during unification
-!	fail because alternatives are cut away
-0	fail because predicate is undefined
??	call of a builtin
++	exit of a builtin
--	fail of a builtin
<<	redo of a builtin
!!	fail of a builtin because alternatives are cut away

Finally, the alternatives of an OR-list are indicated with ';'.

ProLog by BIM - Reference Manual
Debugger
Chapter 2

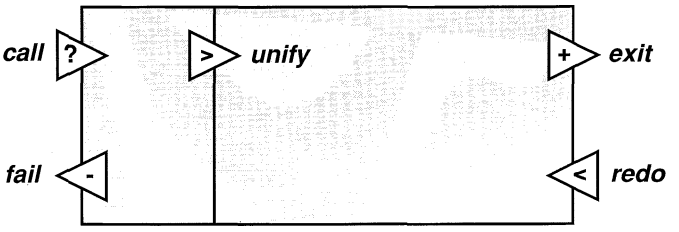
Stepping Debugger

2.1 Box model

The debugger is based on a five-port box model. It is possible to step from one port to the other in the execution of a query. One can also take larger steps to advance more rapidly.

During this stepping, intervention by the user is possible.

With each predicate a box with five ports is associated. For each activation of the predicate a new instance of that box is created. During execution the box is entered or exited via its ports. At those ports, the debugger gives its information. It is possible to indicate at which ports information is required, and even to halt execution at certain ports.



A first port is the *call* port which is passed whenever the predicate is called as a subgoal. Immediately following this port is the *unify* port. This port is reached after unification of the calling subgoal with the head of the predicate. Another input port of the box is the *redo* port. Whenever an alternative definition of the predicate is tried (after failure has occurred), the box is entered via its redo port. The next port that will be passed, is again the unify port. The box also has two output ports. The *exit* port is used when execution of the predicate has succeeded i.e. when all subgoals are solved. The other output is the *fail* port which is used when the predicate fails. This can occur due to a failing unification or because a subgoal could not be solved.

Whenever a port name must be given, one can choose between the full name or an abbreviation which is the first letter of the name.

2.2 Controlling the stepping debugger

There are several levels to control the stepping debugger. First of all, it can be used in two modes :

- *full tracing* → trace mode
- *spypoint checking* → debug mode

Deciding which mode to use can be decided when starting the debugger. It can however also be determined at each step, because it is possible to switch between both alternatives during execution.

Invoking and leaving the debugger

The following builtin predicates are used to invoke and leave the debugger. They only effect the queries following the command, and not the current one, if they are called from a predicate. From the moment the debugger is started, all subsequent queries are executed in debugging mode.

debug/0

Invokes the debugger and starts it in spypoint checking (debug) mode. Spypoint checking assumes spies have been placed on the ports that must be traced. Ports without a spy are not traced.

nodebug/0

Turns the debugger off.

trace/0

Invokes the debugger and starts it in full tracing (trace) mode. In full tracing mode, the debugger traces all ports it encounters.

notrace/0

Turns the debugger off.

The **debug/0** and **trace/0** predicates initially switch to debug mode (as on a leashed port). In this mode the system waits for breakpoint setting, a request for information (see the section on Source Line Debugging) or the run command which will allow one to invoke the predicate to be executed.

run

Leaves debug mode.

Spypoints

A spypoint on a port of a predicate is an indication to the system that it must catch the control flow at that port. Interaction with the debugger is only possible at those points. The user can select if a port must be shown or not. A port can be leashed or not. If the port is leashed the system halts execution and waits for a command of the user. An unleashed port is shown but execution continues.

Getting and removing spypoints

Several builtin predicates are provided for setting and removing spypoints. Normally, when setting a spy, one must indicate on what predicate it must be set and on which ports of its box. As it occurs frequently that one wishes to set spies on the same set of ports for several predicates, the possibility exists to define default ports for setting spies.

When a predicate is initially loaded, it has no spies. Spies that are set remain active until the end of the session, or until they are explicitly removed. *Note* that when one leaves the debugger, the same spypoints remain set and become active again when the debugger is reinvoked in the same session.

showspy/0

Prints the current spypoints on the current output stream.

spydefault/1

spydefault (_Port)
spydefault ([_Port | _PortList])

arg1 : free or ground : atom or list of atom

If *arg1* is ground, the default ports on which spies will be set are defined by *arg1*. This can be one atom (to select only a single port), or a list of atoms (to indicate a list of ports) chosen from call, redo, exit, fail, unify. The port names can be abbreviated to their first letter. If *arg1* is free, it is instantiated to the current spydefault setting (and will always be a list of full port names). Originally the spydefault includes the five ports.

showspydefault/0

Prints the current spydefault setting on the current output stream.

spy/0

Sets spies on all predicates currently in the database. The spies are set on all ports that are currently selected (with **spydefault/1).**
Any existing spypoints are left unchanged.

spy/1

spy (_Predname / _Arity)

spy (_Predname)
spy ([_Predname / _Arity \| _PredList])
spy ([_Predname \| _PredList])

arg1 : ground : atom/integer, atom or list

Sets spies on the predicates given in *arg1* on the currently selected ports. A single predicate is given in the form name/arity. When the arity is omitted spies are set on all predicates with *arg1* as functor name. The predicates must be given, as one predicate or collected in a list in *arg1*. Any existing spypoints are left unchanged.

spy/2

spy (_SpyList, _Port)
spy (_SpyList, [_Port \| _PortList])

arg1 : ground : atom/integer, atom or list
arg2 : ground : atom or list of atom

Sets a spy on the predicates given in *arg1* on the ports indicated by *arg2*. The predicates must be given as single predicate or collected in a list in *arg1*. The ports are given in a list in *arg2* or as one single atom and may be abbreviated to their first letter. Any existing spypoints are left unchanged.

nospy/0

Removes the spypoints from all ports of all predicates in the database.

nospy/1

nospy (_PredName / _Arity)
nospy (_PredName)
nospy ([_PredName / _Arity \| _PredList])
nospy ([_PredName \| _PredList])

arg1 : ground : atom/integer, atom or list

Removes the spypoints from all ports of the predicates given in *arg1*. The predicates must be specified as in **spy/1**.

nospy/2

nospy (_PredList, _Port)
nospy (_PredList, [_Port \| _PortList])

arg1 : ground : atom/integer or list of atom/integer
arg2 : ground : atom or list of atom

Removes the spypoints from all ports indicated in *arg2* of the predicates given in *arg1*. The predicates must be given as single predicate or collected in a list in *arg1*. The ports are given in a list in *arg2* or as one single atom.

Controlling port selection

The user can select the ports he wants to see during debugging. The non-selected ports will never be shown and execution will never be suspended at such ports.

showports/1

showports (_Port)
showports ([_Port | _Portlist])

arg1 : ground or free : atom or list of atom

If *arg1* is instantiated, these ports will become the selected ports to be shown during stepping debugging. If *arg1* is free, it is instantiated to the currently selected ports.

Controlling leashing

Another level of debugger control is known as *leashing*. A port can be leashed or unleashed. Whenever the debugger traces a leashed port, it halts execution and waits for a command from the user. An unleashed port on the other hand, is traced but execution continues immediately.

Leashing control is done on the port level. It is the same for all predicates. It is thus not possible to have different ports leashed for different predicates.

leash/1

leash (_Port)
leash ([_Port | _PortList])

arg1 : ground or free : atom or list of atom

Sets leashing on the ports given in *arg1*. This can be a single atom giving the name of a port or a list of such atoms. The previous leash setting is undone. If *arg1* is free, it is instantiated to the current leash setting which is a list of full port names.

showleash/0

Prints the currently leashed ports on the current output stream. By default the call, unify and redo ports are leashed.

The leash setting is not affected by turning the debugger on or off. It remains unchanged until the next call to **leash/1** with a non-free argument.

Finally, it is possible to *interrupt* a running program if the debugger is active. When interrupted, a choice is given to either abort the execution (i.e. return to the top-level), continue execution, or switch to full tracing. In the latter two cases, the interrupt handler is called and executed. See *Builtin Predicates - Signal handling* on how to change the default handler (the default handler returns to the top-level with an error-message).

Actions on leashed ports

On a leashed port, execution is suspended and the user has to give a command to determine how execution should continue.

The available commands are listed below, together with the key that has to be entered to invoke them. Some commands can also be invoked by typing their first letter as specified between brackets after each command. A sequence of commands can be given on the same command line separating them with '**;**'.

New commands can be defined by the **command** command as mentioned in the previous chapter getting started.

The default command, which is assumed when the empty string is entered, depends on whether the predicate is user-defined or builtin. For user-defined predicates the **creep** command is executed, while for builtins the **go on** command is used.

help (h or ?)

Prints an overview of available commands.

help <*command*> (h or ?)

Prints more information for <command>.

alias <*key*> <*string*>

Defines <key> as an alias for <string>. Without argument, a list of currently defined aliases is printed.

backp (b)

Goes back zero levels. This means restart execution of the first definition of the current predicate (go back to the call port of the same box). Side effects are not undone. On a port of a builtin predicate, this command has the same effect as **creep**.

backp <*n*> (b)

Goes back <n> levels. This means, restart execution of the first definition of the predicate activated <n> levels back. No side effects are undone. If <n> equals zero, this command is exactly the same as back without argument. On a port of a builtin predicate, it has the same effect as creep. Use the command **where** to get an overview of the activated predicates at each level.
In counting the levels, static predicates not compiled to debug code are discarded. The same is true for predicates that have been cut in debug cut mode.

button <*command*>

Appends a button for <command> in the debugger window .

creep (c)

Resumes execution with full tracing until the next leash port.

Depth <*n*> (D)

Sets the writedepth for the debugger to <n> levels. Its default value is 10. This is analogous to **debug(writedepth, n)**.

fail (f)

The system acts as if the current predicate failed for some reason and starts backtracking. On the fail port this command has the same effect as creep.

go on (g)

Resumes execution without tracing until the redo port of the same box or until the first port after the box is left, whichever comes first. This is extremely easy for predicates for which one does not want to see any other ports except the call port (e.g. builtins). When arriving at that call port and giving the **go on** command, the unify and exit ports are skipped and the next call in the goal is displayed.

leap (l)

Resumes execution without tracing until the next spypoint or the next port of the same box, whichever comes first.

menu <*command*>

Appends a menu item for <command> in the debugger window .

Module (M)

Switches the toggle for the printing of module qualifications in trace lines. This is analogous to **debug(writemodule,_)**.

nextp (n)

Resumes execution without tracing until the next spypoint.

Prefix (P)

Switches the toggle for the usage of operators in printing trace lines. This is analogous to **debug(writeprefix,_)**.

prolog (p)

A query prompt is displayed and a Prolog call can be entered. This call will be executed in non-debug mode. The results are given according to the value of the

'showsolution'-option. Afterwards, control returns to the debugger at the same point and a new command is expected. This can be used e.g. to request a listing, or to set or remove spypoints during program execution.

Quotes (Q)

Switches the toggle for the usage of quotes in trace lines. This is analogous to **debug(writequotes,_)**.

quit (q)

Aborts and returns to the toplevel of the system.

redo (r)

Restarts execution of the same definition of the predicate that is currently being executed. This brings the execution back to the redo port of the same box (or the call port if the current definition is the first one). Note that no side effects will be undone. However, if a trace is recorded, it will also turn back so that nothing will appear twice. On a port of a builtin predicate, this command has the same effect as creep.

skip (s)

Resumes execution without tracing until the next port of the same box or until the first port after the box is left, whichever comes first. Leaving a box means the exit or fail port is passed. Both conditions may seem equivalent. The difference is when the output ports are not leashed, because then the debugger continues in skip mode. This command entered on an exit or fail port has the same effect as creep.

Trace (T)

Sets recording of the trace on or off .

unbutton *<command>*

Deletes the button for <command> in the debugger window.

unmenu *<command>*

Deletes the menu item for <command> in the debugger window.

where (w)

Gives a trace-back list of the active predicates. First the immediate ancestor is printed, followed by its ancestor and so on until the top goal is reached. The debugger remains at the same port and a new command is prompted.

2.3 Source line debugging

Instead of using a predicate oriented approach, programs can also be debugged in a source oriented fashion. This means that the debugging process follows the structure of the program as it was written.

In ***ProLog***, source-oriented debugging is based on program lines rather than on predicates.
Break points can be set on program lines. It is possible to set multiple break points: one on each line. Resuming the program execution can then be done line per line.

Break points

A break point is a variant of a leashed spypoint. The execution always stops at a break point. The corresponding line of the program source is printed out.

If a breakpoint is set on a clause heading (i.e. a predicate definition), the relation between break points and spypoints is as follows :

- A break point that is set at a line containing a clause heading is only put on the unify port of the corresponding clause of the predicate.
- A spypoint at the unify port of a predicate is put on the unify ports of all the clauses of that predicate.
- If the predicate has only one definition, a break point will have the same effect as a spypoint on its unify port.

If a break point is set on a line containing a call (as first term), the break point will be put on both the call port and the redo port of the box that is created for that predicate activation. In contrast to a spypoint on the call and redo ports of that predicate, the break point will not be encountered if the predicate is called from other places.

Integration in stepping debugger

The source line debugger is incorporated in the stepping debugger. There are some additional commands to control it. Both methods can be mixed.

The source line commands can be executed not only from a break point, but also from each leashed spypoint.

Commands

The following set of commands are available as an addition to the stepping commands. New commands can be defined by the command **command** as mentioned in the previous chapter. A sequence of commands can be given on the same command line separating them with ' ; ' (spaces are significant).

Some notes on the arguments of these commands:

<line>	A line is indicated by its number. By default, this refers to the current source file. To name a line in a different file, without changing the current source file, the line number may be preceded by <filename>, the source file name between back quotes.
<filename>	A file is always named by its base name (without the .pro extension). The usual UNIX rules for absolute or relative names apply.
<pred>	A predicate must be given in the form atom/arity.
<varname>	The name of a variable is its symbolic name with the leading '_'.

clear *<line>*

Any break point at <line> is removed.

cont

Execution continues up to the next break point.

delete *<number>*

The break point with identifier <number> is removed.

up

This command does not affect the execution. It only brings the focus environment one level higher in the active predicate chain, and allows the **print** command to be used on variables in the ancestor environments.

down

This is to be used after one or more **up** commands. It brings the focus environment one level lower. This command does not affect the execution.

back

Goes back to the previous level.

file *<filename>*

The source file with name <filename>.pro is loaded as current source file.

list *<line1>, <line2>*

A piece of the program source, from <line1> up to <line2> is displayed.

next

The current line is executed. As soon as the following line becomes active, execution is halted again. The called subgoal is not entered. It also stops if there is no following line for the predicate or if the current line leads to a failure.

pred *<pred>*

The first line of the first definition of predicate <pred> is displayed. This command has no effect if <pred> has no definition or was not compiled for debugging.

print *<varname>*

The value of the variable with name <varname> and defined in the environment of the predicate that is being displayed, is printed out. If that variable is not defined at that moment and in the current environment, this is indicated accordingly.

The **up** and **down** commands may be used to reach variables in other environments.

show

The line where execution is currently stopped, is displayed.

status

A list of all active break points is printed. The number between brackets is the break point identifier.

step

Execution is resumed for a single step. This means that it stops from the moment that another line becomes active. This can be the following line or the heading line of a called subgoal or a line reached after failure.

stop at *<line>*

A break point is set at the indicated <line>. If the given line does not contain either a heading or a call, the source is scanned backward for a line that does contain one.

stop in *<pred>*

Break points are set at all lines that contain a heading of predicate <pred>. This is a variant of setting a spypoint on the unify port of the predicate. It is not completely the same because the unify port does not have to be leashed in order for the spypoint to become a break point.

ProLog by BIM - Reference Manual
Debugger
Chapter 3

Post Execution Debugging

3.1 Trace recording control

Post execution debugging requires the query to be executed and its trace to be recorded. This can be done in conjunction with the stepping debugger or quietly, without tracing anything during execution. Afterwards the recorded trace can be analysed.

Normally, the debugger will not record the trace of a program. If it has to, one must indicate this before entering the query. In that case, the following query (and only the first one) will have its trace recorded. A recorded trace can be analysed immediately after termination of the query or after execution of some other queries (but then without recording the trace).

When a trace is recorded, it is possible to temporarily turn the recording off and on. This may lead to problems when trying to analyse the trace with the builtin analysis algorithm.

Another method to ignore some parts of the trace is to set the depth of the trace that must be recorded. This is done with

?- debug (tracedepth, _x).

which means that from that moment on, only _x levels of the trace will be recorded. This method may also lead to problems when trying to analyse the trace with the builtin analysis alghorithm. When _x is negative, the limit on the depth is unset.

The following predicates provide control over the trace recording process :

keeptrace/0

A trace of the execution of the following query will be recorded. After completing the query execution, analyze/0 is automatically activated.

The previous section of actions on leashed ports explains a facility to turn recording of the trace on and off interactively. This has only effect when the trace is currently recorded.

3.2 Trace analysis

Once a trace is recorded, it can be analyzed either manually by zooming through it, or in a more efficient way by using the algorithmic debugging builtin predicate.

The predicates that are provided for zooming on the trace are :

zoomln/2

zoomln (_FromLine, _ToLine)

arg1 : ground : integer
arg2 : ground : integer

Displays lines *arg1* to *arg2* of the trace. If *arg1* is less than 1 it is replaced by 1 and if *arg2* is greater than the number of the last recorded line, it is assumed to be equal to it. If the indicated range is empty, an error message is printed, giving the total number of recorded lines.

zoomld/2

zoomld (_FromLine, _Depth)

arg1 : ground : integer
arg2 : ground : integer

Gives the trace from line *arg1* on for a maximal nesting depth of *arg2*. All lines that are nested more than *arg2* levels deeper than the line *arg1*, are not printed. Output is terminated at the first trace line that has a smaller level than the first line or that is a call port of the same level as the first line. If *arg2* equals zero, only lines of the same level as the first line are printed.

The analysis algorithm is invoked with :

analyze/0

Searches interactively for bugs in the query for which a trace has been recorded.

The output of this analyzer resembles the source form of the predicates. Each goal predicate that is considered is displayed by its head and all its subgoals, one per line. The line on which the head is displayed has as number 0, the first subgoal is on line 1 and so on for the following subgoals.

A predicate that failed, will only have its succeeded subgoals shown. The subgoal that caused the failure is also printed, followed by the indication *failed*. If the predicate failed because the unification of the head failed, then this indication is printed on the head line and no subgoals are shown.

A predicate that was compiled to non-debug code, is displayed without its subgoals, as if it were a fact.

The analyzer first displays the query as a goal predicate. It then waits for a command before continuing. At this point, one can choose to investigate a subgoal or to return to the calling predicate using one of the following commands.

If the debugger window is active, the corresponding program source for each predicate definition is displayed in the source window.

The analyzer always starts by displaying the goal predicate at its call port. It considers the first solution of the query, even if a previous analysis has investigated following solutions. And it starts such that for succeeded subgoals only the solution is investigated and not the failures.

Commands

New commands can be defined by the **command** command as mentioned in the previous chapter . A sequence of commands can be given on the same command line separating them with ';'

advance (a)

If the current goal predicate has failed, or if its failures are requested for, it is possible that there are several failures to be analyzed. Initially the first is displayed. With this command the analyzer advances to the next failure or solution (if there is one). On the top level of the query, this command can be used to advance from solution to solution or failure. Each failure or solution of the query will be investigated. This corresponds somewhat to the backtracking mechanism of Prolog. Once the last failure or solution has been displayed, this command has no further effect.

alias <key><string>

Defines <key> as an alias for <string>. Without argument, a list of currently defined aliases is printed.

back (b)

Leaves this level and go back to the caller. On top level (the query) this command is ignored.

call (c)

The goal predicate will from now on be displayed at its call port. This means that the head and all subgoals will have their arguments unified as far as they were just before being called. If for instance, the first subgoal has a free variable as parameter, it will be displayed as variable. If this subgoal instantiates this variable and the next subgoal also has it as parameter, then for the second subgoal, it will be displayed in its instantiated form.

detail (d)

Gives a more detailed output for the goal predicate. Both the call port and the exit port of the subgoals (and the head) are printed out. Each line is printed in the same form as for the zoom predicates.

exit (e)

From now on the goal predicate is to be displayed at its exit port. All parameters are shown as far instantiated as they were after completion of each subgoal.

failure (f)

Investigates the failing subgoal. This is exactly the same as entering the number of the failed subgoal as command. If all subgoals succeeded, nothing happens.

Failures (F)

If a subgoal has succeeded, the analyzer will only display its solution when you ask to investigate it. It could also be the case that you want to see the failures that occured before the solution was found. With this command a switch can be turned that indicates the analyzer if it has to investigate previous failures or not.

help (h or ?)

Prints this overview in a short form.

invest <*n*>

Continues investigation of subgoal number <n>. If <n> is outside the range of displayed subgoals, the command will be ignored. Commands '1', '2', up to '9' are predefined as aliases for 'invest 1' and so on.

quit (q)

Aborts analysis and return to the top level of the system.

WINDOWING ENVIRONMENT

Windowing Environment

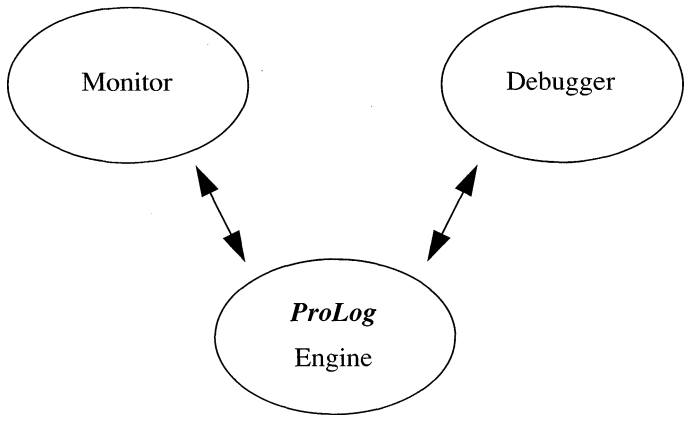
Contents

ProLog by BIM - Reference Manual
Windowing Environment
Chapter 1

XView

1.1 Window configuration

The ***ProLog*** window environment consists of several window frames, each running as a separate process. The following figure gives a view of this window and process configuration. It also shows the windows interaction.



The master window is the one running the ***ProLog*** engine. This is the frame in which the system is started. This is usually a *shelltool* or a *cmdtool*.

The other windows are not active by default. They can be activated separately at any time. It is also possible to desactivate them when desired. The activity of these windows is controlled by the **please/2** predicate with switches **envmonitor (em)** and **envdebug (ed)** for the monitor and debugger window respectively.

To pop-up the monitor window, for example, enter the query :

> ?- please(em , on) .

As all other switches, the environment control switches can also be specified in the command line. The following command will start up the system with both windows active :

% BIMprolog -Pem -Ped

Whenever a window is activated, the master process will start up a new process that creates the window frame. Upon creation, the defaults database is consulted to get the user's preferred window layout. When running, the monitor and debugger windows interact with their master.

These windows can be closed, resized and moved around the screen like any other XView frame.

In the following chapters, the three processes with their associated frames will be described. First comes the master window, then the monitor window and finally the debugger window.

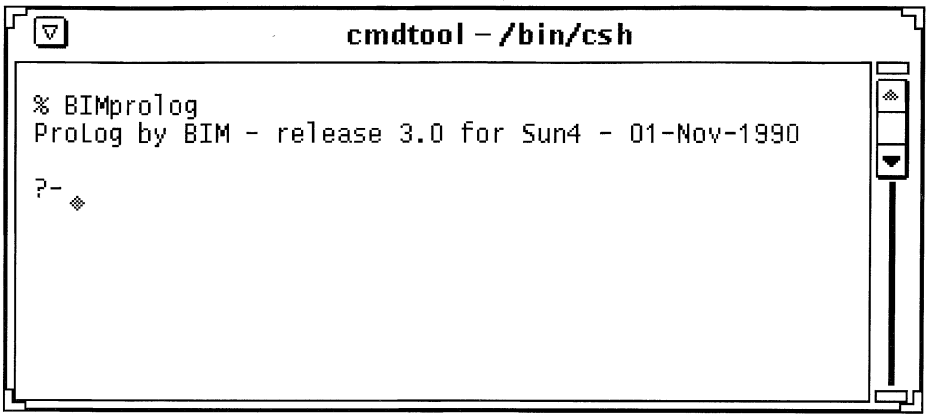
Remarks:

- Note that a special chapter in the ***ProLog*** User's Guide is devoted to the use of the Windowing Debugger.

- In order to conform to the X standard, the windowing environment of ***ProLog*** is described in its XView version. A SunView version is also available with the same functionality as explained in the following pages.

1.2 Master window

As previously stated, the master window is the window in which the ***ProLog*** engine was started up, being in most cases either a *shelltool* or a *cmdtool.*



This window will continue its role as standard input and output channel. This means that every program interaction goes via this window. Also, the top-level of the engine uses this window as communication port. Queries have to be entered in it, and the answers are given in the same window.

When the debugger window is not active, the master window also plays the role of debugger interaction window. All commands to the debugger and all trace information given by the debugger, use the same window.

If the debugger window is active, all debugger interaction will be channeled through that window instead of going via the master window.

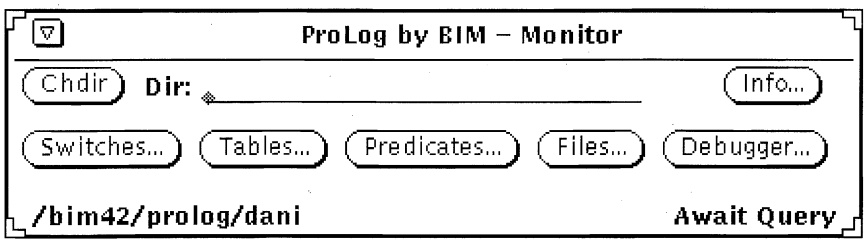
1.3 Monitor window

The purpose of the monitor window is to provide some means of monitoring the system through a user-friendly interface instead of having to type in whole sequences of goals.

It offers a number of predicates implemented with windows and buttons. This eases the monitoring of the operation mode of the system.

Another advantage over normal typed-in queries, is the capability of asynchronous interactions. One can easily change the system's operation mode during debugging, without having to abort the execution or enter the prolog-call mode.

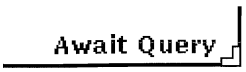
The base frame of the monitor window contains a set of pop-up buttons, that open a sub-



window with available commands on the indicated topic. Each topic will be described separately in the following sections.

Status report

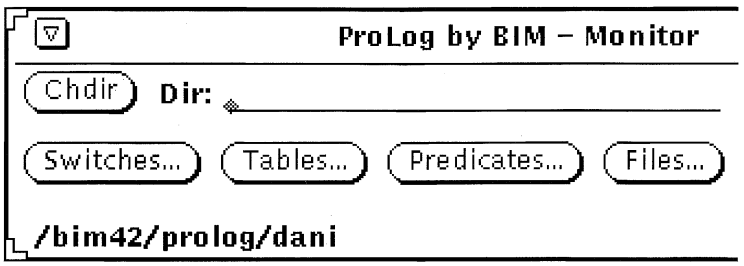
In the right corner of the monitor reports the current system status. This can be one of the following :



The system is waiting at the top-level for a query (or any clause) to be entered. This suggests that you can use the master window for interaction.

The system is executing a query. In this state, it is impossible to start another query or to enter a clause.
Interactions with the system are not possible, except for interrupts. If the execution is in debug mode, interactions can be enabled when the execution of the query is suspended.

Working directory



The current working directory of the engine is indicated in the left bottom corner of the monitor.

The working directory can be changed by entering a relative path in the **Dir :** field , hitting <CR> and clicking the **Chdir** button. As a result, the current directory will be changed with this relative value (if this is possible and allowed) and the value will be removed.

Taking the frame from the figure above, when the name 'project' is typed in the 'Dir:' field and the **Chdir** button is pressed, the command

% cd project

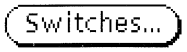
is executed, changing the current directory to

/bim42/prolog/dani/project

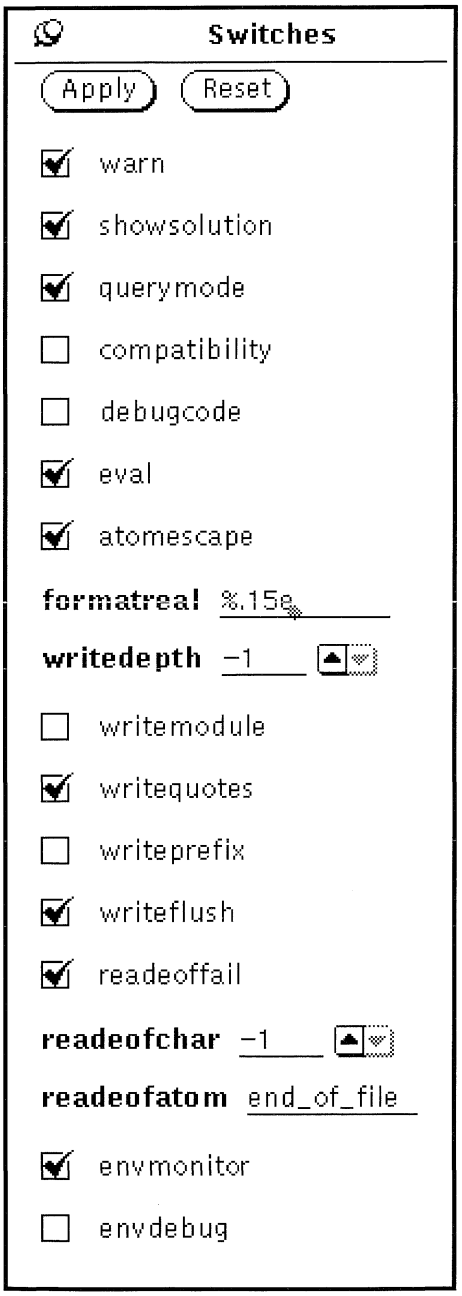
The **Dir :** field will be erased.

This setting of the working directory has effect on the whole ***ProLog*** system. All active windows will have their current directory changed accordingly.

Switches



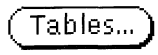
By pressing the **Switches** button, a graphical implementation of the builtin predicate **please/2** is popped-up.



The toggle options are represented by toggle buttons.
The value options have a value field that can be edited.

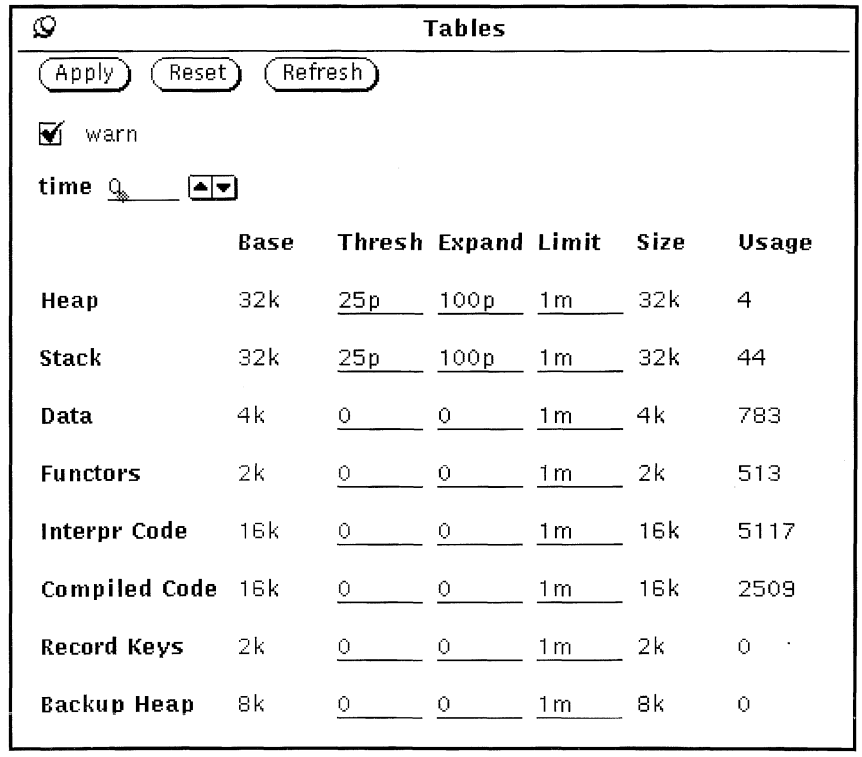
Clicking a toggle value switches the values on and off. A numeric value button (e.g. writedepth) can be increased or decreased by pushing the up or down arrow. It can also be changed by editing the value field. The other options can be changed by editing the value field attached to that option.
The changes which were made by clicking or editing value fields are only applied and sent to the engine when the Apply button is pushed. The Reset button resets the state of the window and unsets the changes which where not yet applied.

Tables



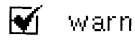
By pressing the **Tables** button , the tables window is popped up.

This **Tables** window represents graphically the builtin predicate **table/2**. It has one toggle button for the option warn and one numeric button for the option time. The rest of the window reflects the current table size options of the different visible tables used by the ***ProLog*** system.

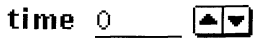


	Base	Thresh	Expand	Limit	Size	Usage
Heap	32k	25p	100p	1m	32k	4
Stack	32k	25p	100p	1m	32k	44
Data	4k	0	0	1m	4k	783
Functors	2k	0	0	1m	2k	513
Interpr Code	16k	0	0	1m	16k	5117
Compiled Code	16k	0	0	1m	16k	2509
Record Keys	2k	0	0	1m	2k	0
Backup Heap	8k	0	0	1m	8k	0

The value of warn can be switched by pushing its toggle button.



The value of time can be changed by the numeric button attached to it.



The parameters of the different tables that are underlined can be changed by editing their value.

Note that in this release only the Heap and the Stack are expandable.

All changes made are only active when the Apply button is pushed.

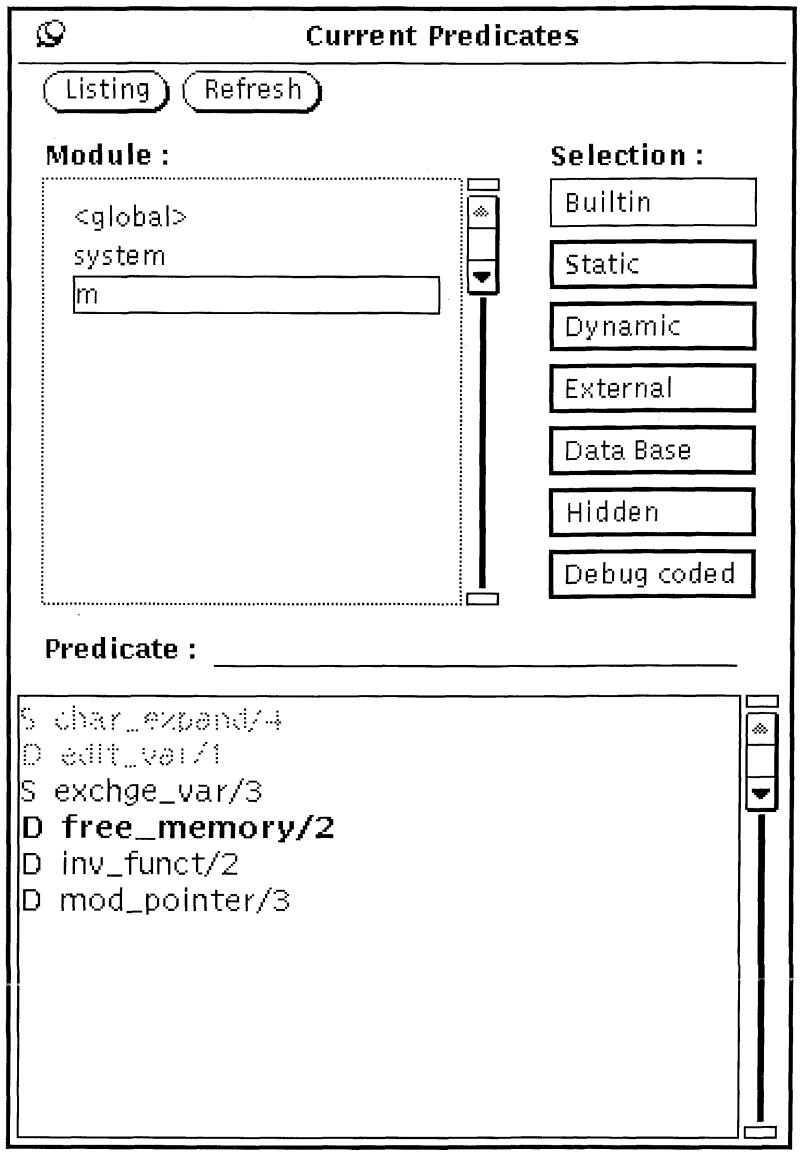
With the Reset button, the original state of the window can be set.

Pushing the Refresh button shows the values of the different parameters at that time. The Refresh button can be pushed when a query is active.

Predicates

Predicates...

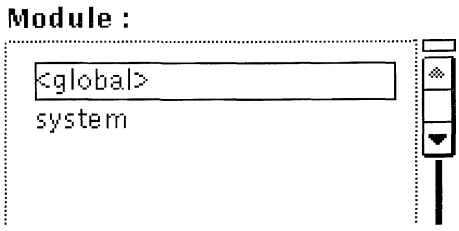
A possible configuration of the **Predicates** pop-up window is given in the figures below.



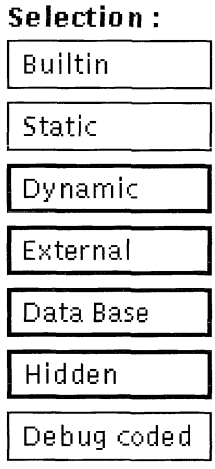
The window contains two selection fields: one for the module and one for the predicate classes. The lower part of the window displays the list of selected predicates.

All predicates that are displayed belong to a single module. It is not possible to select predicates from several modules simultaneously.

Selection of the module is done by clicking the desired module in the modules subwindow. All currently known modules are given in this window.



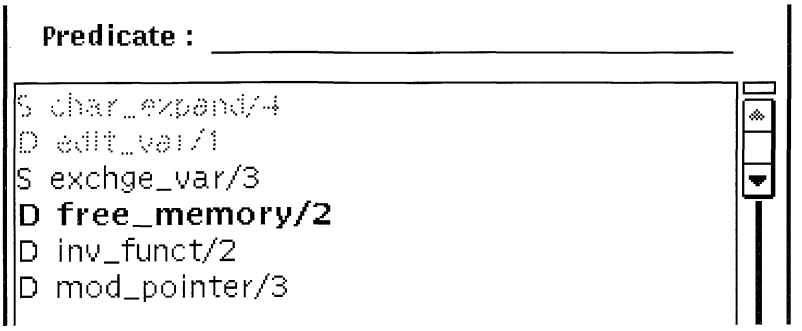
A second selection criterion is the predicate class. The Selection window gives an overview of the different classes.



The first five classes are disjunctive collections of predicates. A built-in predicate is assumed to be only built-in and not static or dynamic or anything else, and so on.

The two last items overlap with the other classes. A hidden predicate can be static or dynamic. Debug coded predicates form a subclass of the dynamic predicates.

The selections can be mixed to include several classes of predicates at once.

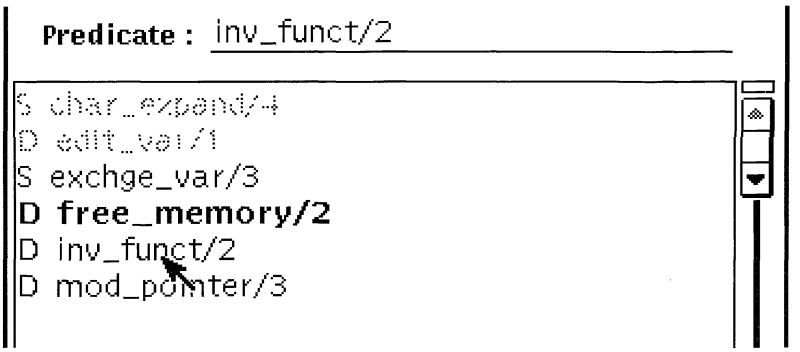


In the list of selected predicates, the class a predicate belongs to is indicated by the corresponding abbreviation in the left column. Hidden predicates are displayed in grayed-out font. Debug coded predicates are indicated by using a bold font.

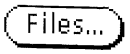
The example in the figure has static and dynamic predicates selected. Hidden predicates are also requested for. Because dynamic predicates are selected, the debug coded predicates are also shown. For example, *free_memory*/2 is a debug coded predicate. The predicate *exchge_var*/3 is a static predicate and *edit_var/1* is a dynamic and hidden predicate.

The **refresh** button should be used whenever important changes to the ***ProLog*** data base are made. Adding new predicates or deleting existing ones, doesn't change the list of selected predicates automatically. Another use of **refresh** is when the selection of predicates has been changed.

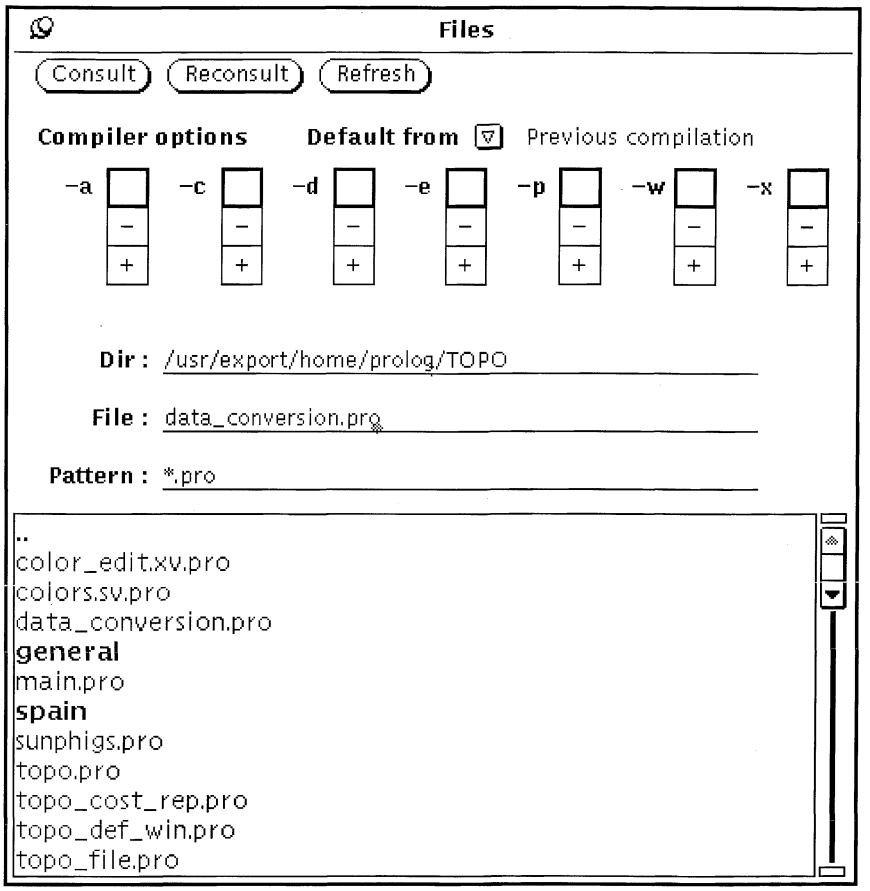
The button labeled **listing** invokes the builtin **listing/1** with as argument the selected predicate. This is the predicate in the field **Predicate** which can be entered either manually by typing it in or copying it from another window. It can also be copied automatically by clciking it in the predicate list with the left mouse button.



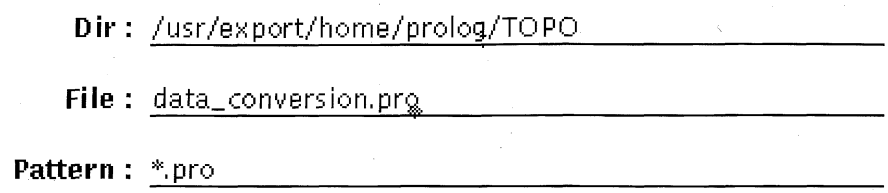
Files



This button pops up the **Files** window that provides an easy way to browse, compile, consult and reconsult files.

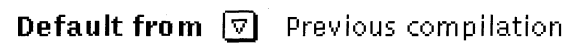


The upper part of the window provides some control facilities. The lower part contains the list of source files and subdirectories in the specified directory.



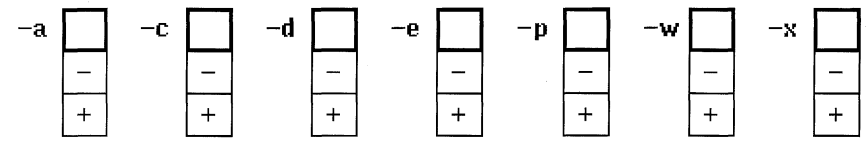
The current directory can be modified by editing the **Dir:** field (and clicking the refresh button) or by clicking on a directory name (displayed in boldface) in the list of source files. The **File :** field contains the selected file. It is normally selected from the file list by clicking it with the left mouse button. The field can also be edited. The **Pattern:** field allows to reduce the number of files in the browser. Only the files matching the pattern will be displayed.

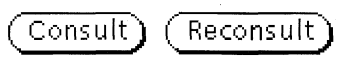
At the top of the control panel, the compiler options can be set.
The default options can be taken either from the previous compilation of the file



or from the current engine options.

Whatever the selected default is, these default options are always overwritten by the options (-a[lldynamic], -c[ompatibility], -d[ebug], -e[valuation], -p [operators], -w[arning], -x [atomescape]) set with the choices "no-change / off / on".



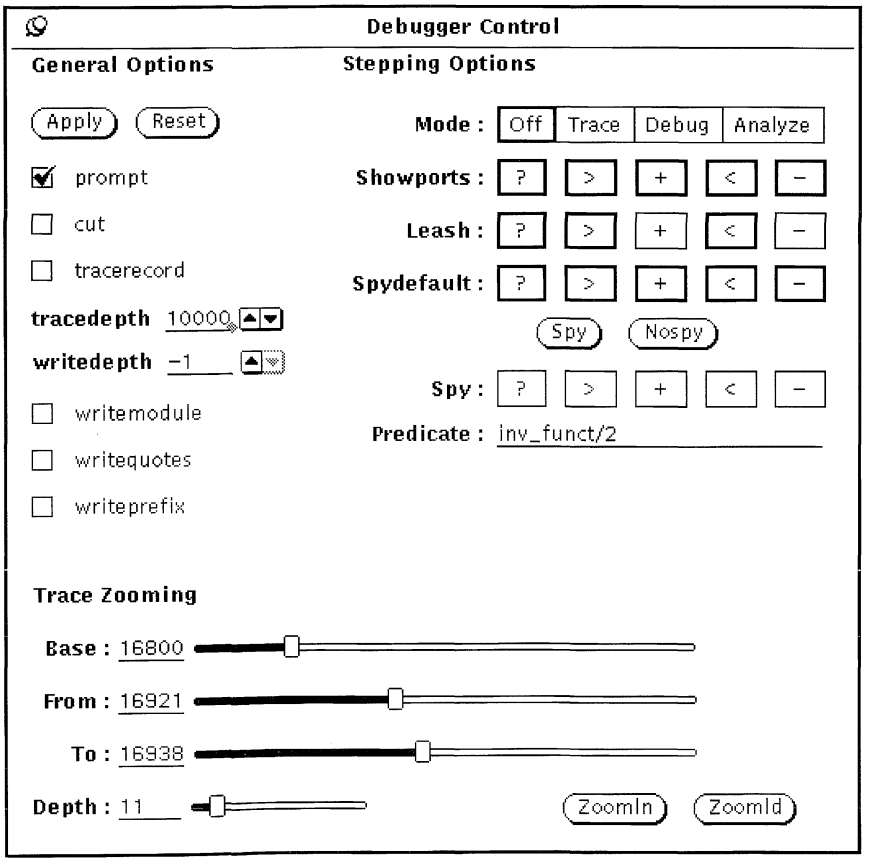


The buttons **Consult** and **Reconsult** invoke the builtins **consult/1** and **reconsult/1** respectively. Their argument is constructed from the compiler option list and from the directory and file name fields. Beware that the file is not consulted immediately as the button is released. It might take some seconds to load big files.

Debugger

Debugger...

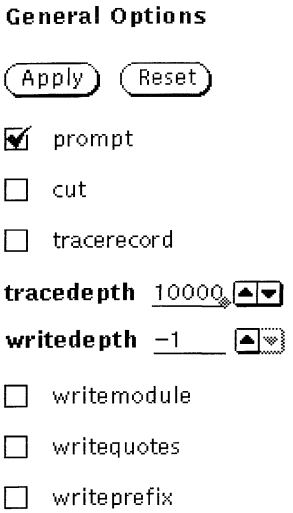
The **Debugger** pop-up window is the most extended one. Therefore it is divided into three logical regions : one for general options, one for stepping options and one for trace zooming facilities.



Each of these regions will be discussed separately in the following paragraphs.

General options

The general options are the graphical equivalent of the **debug/2** built-in. Each toggle option is represented by a button and each value option has a value field.

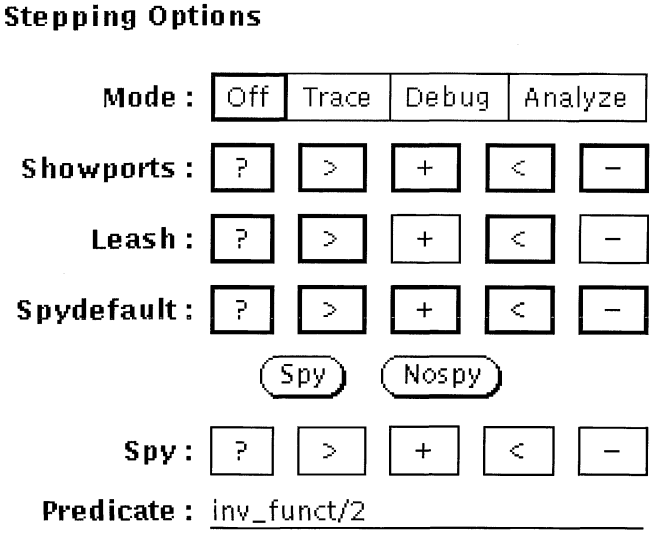


Options can be changed by clicking the corresponding toggle button or by editing the value and hitting return. Numeric values can be increased or decreased by using their associated arrows.

All changes made are only active when the Apply button is pushed.

With the Reset button, the original state of the window can be set.

Stepping options



The first item sets the debugger mode. The debugger can be switched off, in which case the engine executes queries in normal mode. It can be in either trace or debug mode, which is the same as when the builtin predicates **trace/0** or **debug/0** are invoked. The last mode is analyze, which is the mode that results from calling the builtin **keeptrace/0**.

The three following items provide the builtin predicates **showports/1**, **leash/1** and **spydefault/1**. The five ports are represented by their symbolic abbreviation :

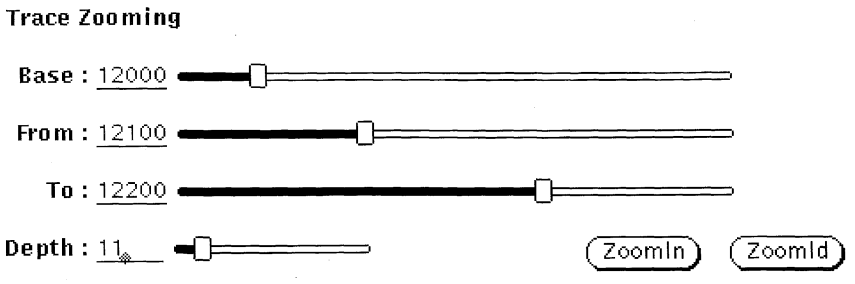
- **? call port**
- **> unify port**
- **+ exit port**
- **< redo port**
- **- fail port**

Changing the settings can simply be done by clicking on the port that must be switched.

The other items supply the **spy/2** and **nospy/2** builtin predicates. The field with label **Predicate :** holds the predicate whose spies are displayed and can be adapted. It can be filled in by editing the field or by clicking a predicate in the predicate list. Ports of the predicate that have a spy point on them, are selected. Spies on separate ports can be switched by clicking the ports. The default spy setting can be applied by using the buttons **Spy** and **Nospy** to set or unset spies.

Trace zooming

The lower part of the debugger control panel contains provisions for zooming through the trace. This corresponds to the built-ins **zoomln/2** and **zoomld/2**. These can be activated by pressing the appropriate buttons.



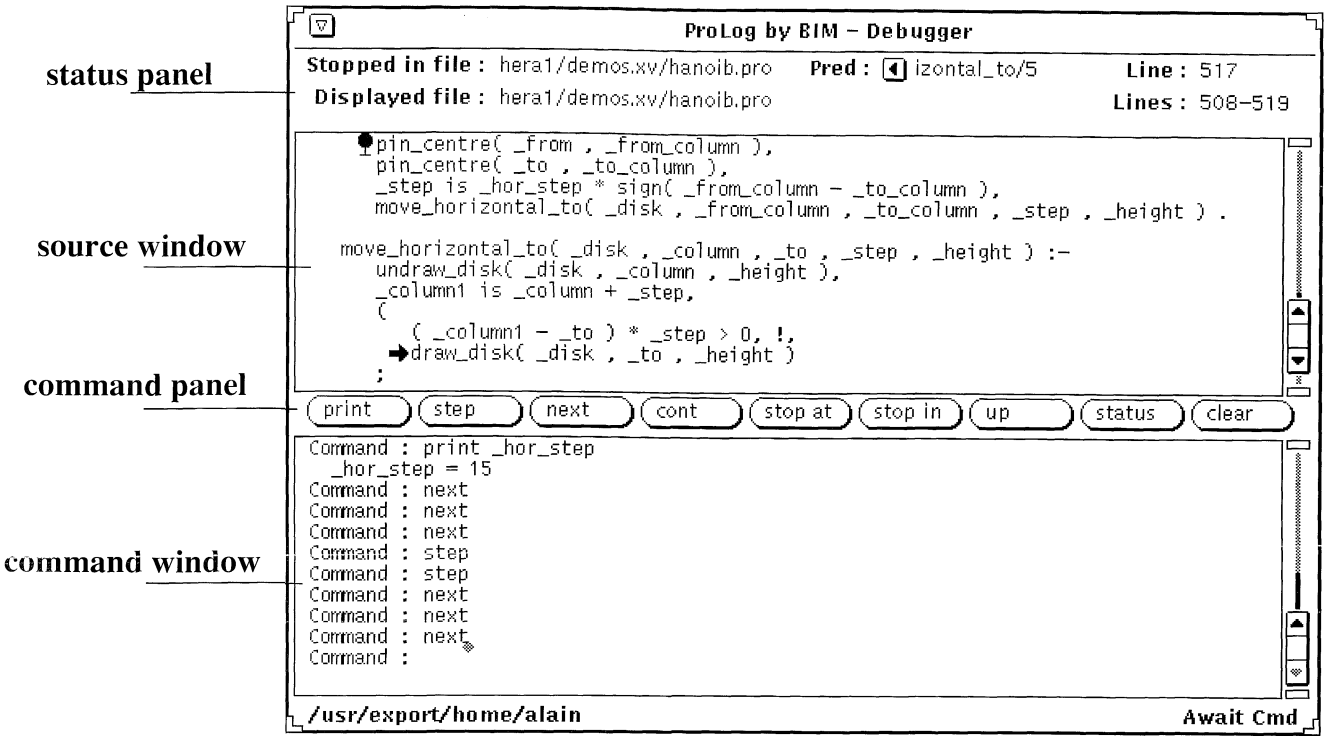
The arguments that will be used, are retrieved from the sliders. The **From** and **To** sliders give the first and second argument for **zoomln/2**. The **From** and **Depth** sliders are used for the arguments of **zoomld/2**.

The **Base** slider is used to set a base value for the **From** and **To** sliders. As it would be impossible to represent the whole range of trace lines in a single slider (there can be many thousands of lines), the sliders only have a range of 300 lines. This is the maximum range where each line number can still be selected. To be able to zoom through higher numbered trace lines, the minimum value of these ranges can be set with the **Base** slider.

In the figure, the base value is set to 12000. This implies that the range of the other sliders goes from 12000 through 12300. They are currently set to 12100 and 12200.

1.4 Debugger window

Window layout



The debugger window consists of :

status panel

This panel holds status information.

source window

A text window (not editable) containing the program source.

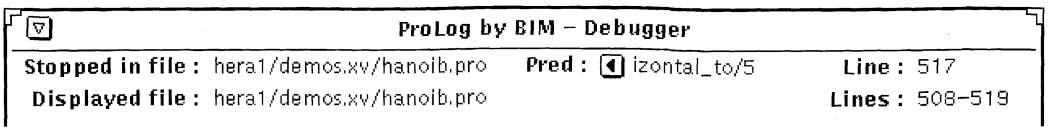
command panel

A selection of debugger commands can be found in this panel.

command window

Output from and input to the debugger goes through this text window. (It can also be edited).

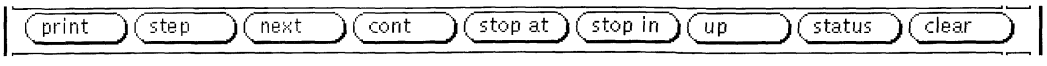
Status panel



The first line gives information on the current break point. It states in which file and on which line execution has stopped. Moreover, the currently active predicate is mentioned.

The second line indicates what file is displayed in the source window and also the range of the displayed lines.

Command panel



The command panel is empty in *nodebug* mode. In the other modes it holds a set of buttons that represent the commands that are available in that mode. There is also a pop-up menu with less frequently used commands.

Which commands are displayed as buttons and which on the menu, and in what order, is customizable through the defaults facilities of the chosen windowing system (see also the section on defaults). The buttons an menu items can be changed temporarily during a session with the command button, unbutton, menu, unmenu.

For commands that take an argument, the argument must be selected before pressing the button.

Any string type arguments can be selected by indicating a single letter of the string. This is automatically expanded to include the whole string (as long as it doesn't contain special characters).

A line number is formed by selecting the line in the source window.

When a predicate is needed as an argument, it should be selected with its name/arity. If only the name is selected, a '/' will be appended and the arity must be entered manually.

There is one button, labeled *invest* in *analyze* mode that acts differently from the others. It is used to go down one level in the execution tree. Therefore the number of the subgoal must be entered. This can be done by typing it in the command window or by indicating the line that contains that subgoal in the same window.

Source window

The source window is used to display the program source during debugging. Whenever the execution is suspended it is updated to display the line where the execution is stopped. This can be either at a break point, or after a stepwise continuation (with the commands *step* or *next*). If the predicate in which the execution is stopped, resides in another file than the one displayed, that file is automatically loaded.

The position of the execution suspension is indicated by a black arrow, as in the following picture.

```
➡horizontal_step( { Left/Right Movement Interlace } 15 ) .
```

This can be a line either containing a clause head (when execution reached a *unify port*), or a line containing a subgoal call (for other ports).

In the same window, all break points that are set, are indicated by a stop sign.

```
move_horizontal_disk( _disk , _from , _to , _height ) :-
   horizontal_step( _hor_step ),
   pin_centre( _from , _from_column ),
 ●pin_centre( _to , _to_column ),
   _step is _hor_step * sign( _to_column - _from_column ),
   move_horizontal_to( _disk , _from_column , _to_column , _step , _height ) .
```

At every moment during a suspension of the execution, the line where that suspension occurred, can be displayed in the source window. Therefore, the *show* command can be used. This displays the line in the window, indicated by a hollow arrow.

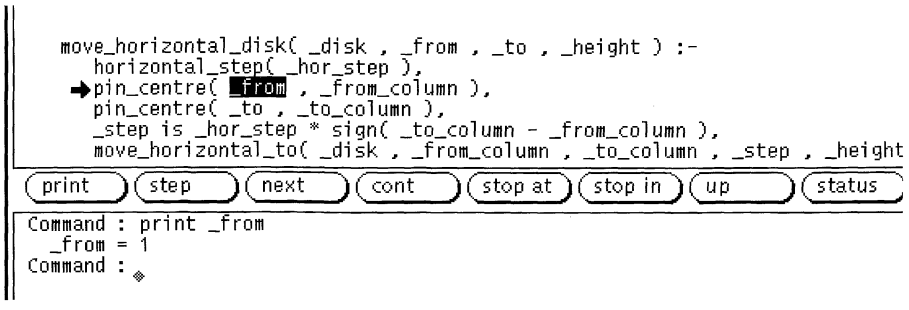
```
move_horizontal_disk( _disk , _from , _to , _height ) :-
 ⇨horizontal_step( _hor_step ),
```

This can be used also in combination with trace line oriented debugging. When the execution is stopped at a spy point, or after a stepping command, the source window is not automatically updated. If the user nevertheless wishes to see the corresponding source line, the *show* command can be used to this purpose.

Another use of this command is to reorientate after having scrolled through the source text.

It is also possible to ask for the source of any predicate by using the *pred* command. This brings the first clause of the desired predicate in the source window, pointed to by a hollow arrow.

At a break point, one can print out the current value of a variable. The easiest way to do so is by pointing to the variable in the source window and then pressing the *print* button.



The only variables that can be interrogated are those in the currently focussed environment. To reach variables in environments of ancestors or descendant predicates, the *up* and *down* command can be used to move the focus to the desired environment. When moving the focus, the source window is automatically updated to display the new focussed environment. If this is not the environment where the execution is suspended, the line is indicated with a hollow arrow instead of a black one.

1.5 Defaults

Default settings of the different frames of the monitor and debugger windows can be changed. One can add the changed defaults to his .Xdefaults file following the X11 conventions.

Following is a list of the parameters of the monitor window that can be changed.

Name	Type
BIM_Prolog.monitor.font	string
Font for text.	
BIM_Prolog.monitor.font.scale	enumeration
Scaling of the choosen font (value: small, medium, large, extralarge)	
BIM_Prolog.monitor.x	integer
Horizontal offset of frame on screen	
BIM_Prolog.monitor.y	integer
Vertical offset of frame on screen.	
BIM_Prolog.monitor.info.x	integer
Horizontal offset of info pop-up from monitor frame	
BIM_Prolog.monitor.info.y	integer
Vertical offset of info pop-up from monitor frame	
BIM_Prolog.monitor.switches.x	integer
Horizontal offset of switches pop-up from monitor frame	
BIM_Prolog.monitor.switches.y	integer
Vertical offset of switches pop-up from monitor frame	
BIM_Prolog.monitor.tables.x	integer
Horizontal offset of tables pop-up from monitor frame	
BIM_Prolog.monitor.tables.y	integer
Vertical offset of tables pop-up from monitor frame	
BIM_Prolog.monitor.predicates.x	integer
Horizontal offset of predicates pop-up from monitor frame	
BIM_Prolog.monitor.predicates.y	integer
Vertical offset of predicates pop-up from monitor frame	
BIM_Prolog.monitor.files.x	integer
Horizontal offset of files pop-up from monitor frame	
BIM_Prolog.monitor.files.y	integer
Vertical offset of files pop-up from monitor frame	
BIM_Prolog.monitor.debugger.x	integer
Horizontal offset of debugger pop-up from monitor frame	
BIM_Prolog.monitor.debugger.y	integer
Vertical offset of debugger pop-up from monitor frame	

The following table gives an overview of the parameters for the debugger window.

Name	Type
BIM_Prolog.debugger.font Font for text windows	string
BIM_Prolog.debugger.font.scale Scaling of text font (Value: small, medium, large, extralarge)	enumation
BIM_Prolog.debugger.width Width of the frame in columns	integer
BIM_Prolog.debugger.srclines Number of lines in the source window	integer
BIM_Prolog.debugger.cmdlines Number of lines in the command window	integer
BIM_Prolog.debugger.x Horizontal offset of frame on screen	integer
BIM_Prolog.debugger.y Vertical offset of frame on screen	integer
BIM_Prolog.debugger.steppingbutton Stepping : Choose active buttons	string
BIM_Prolog.debugger.analyzebutton Analyze : Choose active buttons	string
BIM_Prolog.debugger.steppingmenu Stepping : Choose active menu items	string
BIM_Prolog.debugger.analyzemenu Analyze : Choose active menu items	string

The value of the last four parameters of the above table must be a string. This string contains the codes (a letter followed by a digit) for the buttons or items that must be set. The following table shows the correspondence between the debugger commands and its code.

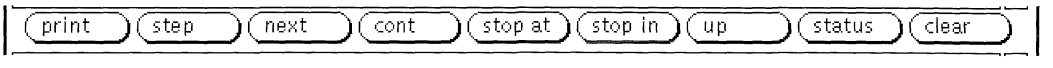
	A	B	C	D	E	F
1	alias	Depth	creep	stop at	cont	print
2	command	Module	go on	stop in	next	pred
3	button	Quote	skip	status	step	file
4	unbutton	Trace	nextp	clear	back	list
5	menu	Prefix	leap	delete	up	run
6	unmenu	-	fail	-	down	-
7	help	-	backp	-	show	-
8	prolog	-	redo	-	-	-
9	quit	-	where	-	-	-

For example:

The default setting of the stepping buttons is :

F1 E3 E2 E1 D1 D2 E5 D3 D4

means that following buttons appear in the command panel of the debugger window:



APPENDIX

Appendix

Contents

A. *Messages from the engine*

There are 7 classes of error messages in the ***ProLog*** system:

SYNTAX	**syntax error message**
SEMANTIC	**semantic error message**
WARNING	**warning message**
OVERFLOW	**overflow message**
BUILTIN	**incorrect use of builtin predicate**
RUNTIME	**run-time error message**
MODE	**incorrect use of mode declaration**

Error messages appear by default on the screen. An option exists for the ***ProLog*** translator (the -l option) to print error messages in a listing file.

The error message format is:

***** <class> <error_number> *** <error_message>**

For Example:

***** BUILTIN 483 *** attempt to divide by zero**

Note that warnings are the only error messages which do not stop the process of parsing, compilation and execution of a program. However, a warning issued from a directive means that this directive has been ignored.

The error numbers are divided in ranges. The following table lists the different ranges and the corresponding types of messages

100-249	messages from the parser
250-299	compiler errors (only from BIMpcomp)
300-399	general builtin errors
400-499	specific builtin errors
500-599	I/O errors
600-699	loader errors
700-799	general errors
800-899	external language interface errors
900-999	messages from the debugger

Internal error messages and *panic error* messages should be reported to your local distributor or BIM, if possible with the context in which the error occurred.

B. Bibliography

1. Bratko [I.].
Prolog Programming for Artificial Intelligence. Addison-Wesley Publishing Company, 1986.

2. Campbell [J.A.].
Implementations of Prolog. Ellis Horwood Ltd, 1984.

3. Clocksin [W.F.] and Mellish [C.S.].
Programming in Prolog. Springer Verlag, 1981.

4. Coelho [H.], Cotta [J.C.] and Pereira [L.M.].
How To Solve It With Prolog. Ministerio da Habitacao e Obras Publicas Laboratorio Nacional de Engenharia Civil, Lisbon, Portugal, 1982 (3rd Edition).

5. Gray [P.M.D.]. and Lucas [R.J.].
Prolog and databases. Ellis Horwood Ltd, 1988.

6. Hogger [C.J.].
Introduction to Logic Programming. Academic Press, 1984.

7. Kowalski [R.].
Logic for Problem Solving. Artificial Intelligence Series, North Holland, 1979.

8. Shapiro [E.].
Algorithmic Program Debugging. MIT Press, 1982.

9. Sterling [L.] and Shapiro [E.].
The Art of Prolog. Advanced Programming Techniques. MIT Press, 1986.

C. Software Performance Report

BIM will use the comments submitted on this form for the improvement of the ***ProLog*** system and its documentation.

Is this manual understandable, usable and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Did you find inconsistencies between the manual and the behavior of the system? Please specify.

Did you discover a bug in the system? Please add listings of programs and results.

Name : _____

Organization : _____

Address : _____

_____

_____

Telephone : _____

Date : _____

315 The specified argument must be an integer or a nil-terminated list of integers.

316 The specified argument must be an atom or a nil-terminated list of atoms.

317 The specified argument must be an atom, integer, real or pointer or a nil-terminated list of such elements.

318 The specified argument must be a nil-terninated list.

320 The specified argument must be a predicate descriptor of the form name/arity.

325 None of the arguments is instantiated.

330 Integer not permitted as head of a clause.

331 Real not permitted as head of a clause.

332 Variable not permitted as head of a clause.

333 Integer not permitted as goal.

334 Real not permitted as goal.

350 Attempt to assert a static predicate.

351 A directive cannot be asserted.

352 A query cannot be asserted.

360 Attempt to redefine a builtin predicate.

361 There are too many variables in the specified term.

362 There are too many tokens in the clause to be asserted.

Specific builtin errors

401 Illegal metacall : integer not permitted as goal.

402 Illegal metacall : real not permitted as goal.

403 Illegal metacall : pointer not permitted as goal.

404 The specified functor is not a callable predicate.

405 Illegal metacall : the specified predicate is unknown.

406 Illegal metacall : the specified predicate is unknown.

407 The specified builtin is unknown.

408 The specified argument must be ground.

410 The second argument must be a global atom.

411 The third argument must be a compound term with a principal functor belonging to the global module.

412 The argument of cut/1 must be instantiated.

413 Retract : This predicate is not a dynamic predicate.

414 The specified functor is not a dynamic prdicate.

415 The maximum number of arguments for a list is exceeded

416 The specified term cannot be asserted as a data base predicate.

417 The specified functor already has a functor.

420 Second argument of **/2 must be a non-negative integer.

421 Arguments of mod/2 must be integers.

422 The second argument of op/3 must be an atom of (xfx,xfy,yfx,fx,fy,xf,yf).

423 The arguments have to be an integer in the range 1..1200, an operator type and a functor name.

424 The first argument must be free or a single character.

425 The first argument must be a single character and the second an integer or one of both may be free.

426 The second argument must be a nil-terminated list and first element an atom.

427 Arguments of bit operators must be integers.

428 The specified argument must be a list of characters, not longer than the longest atom allowed.

429 The specified argument must be a list of integer character codes, not longer than the longest atom allowed.

430 At most one argument may be free, the others must be atoms.

431 The first argument must be a nil-terminated list of atoms.

432 The atom to be constructed is too long.

433 One of the arguments must be an atom and the other free.

434 The given or calculated start position and length arguments must be positive.

435 The arguments must be a pointer, an integer and a pointer. Only one may be free.

439 The keys must be instantiated.

440 The specified key is already in use.

441 The specified key is already in use.

442 Add to non existing key is not allowed.

443 Add to non existing key is not allowed.

444 Record heap overflow.

445 Too many clause references.

446 The specified reference is not a valid clause reference.

450 The specified Key is not an acceptable key.

455 Except for this one, no error occurred previously.

456 Error in loading of error set.

457 The first argument must be an integer or a list of integers and integer pairs (int-int). The second argument must be on or off.

458 The specified error number is out of defined error ranges.

459 The argument list for an error must consist of integers, atoms and functors (name/arity).

460 This predicate is not suitable for dynamic indexing.

461 Illegal size for dynamic index hash table.

465 Dynamic indexing only allowed on dynamic predicates without definitions.

466 Dynamic modes are only allowed on dynamic predicates.

470 The first argument must be a symbolic signal name.

471 The second argument must be one of status/2, ignore, accept, suspend, status, raise or clear.

472 The arguments of status/2 must be free.

473 The signal argument must be a signal name or a list of signal names.

474 A Prolog handler was installed outside of the system's control.

475 An external handler was installed outside the system's control. The specified handler was the previous Prolog handler.

480 Allowed comparison : (int or real) with (int or real).

481 Atoms are not allowed in evaluable expressions.

482 Type of operands are incompatible.

483 Attempt to divide by zero.

484 Non numerical operands.

485 The result of an expression evaluation is an integer, real or atom.

486 Module zero is undefined.

490 The specified command line arguments require too much memory.

491 A command line argument number are outside the specified range.

Input/output errors

500 The mode for opening must be w, r or a.

501 Use an atom as logical filename or a pointer.

502 This logical file not in use.

503 This logical file is already in use.

504 This file is open for output, not for input.

505 This file is open for input, not for output.

508 Unable to open this file : too many files already open.

509 Cannot open this file.

510 Standard file cannot be closed.

511 Use an atom as current stream name or a pointer.

Loader errors

600 Cannot find this file (neither corresponding object file).

601 Error during compilation of this file.

602 The specified predicate is not loaded.

603 During loading: no space enough to allocate dynamically the temporary table of functors.

604 During loading: no space enough to allocate dynamically the temporary table of constants.

605 During loading: no space enough to allocate dynamically the temporary table of modules.

607 During loading: conflict between a static predicate and an existing predicate with the same name and arity.

608 During loading: conflict between a dynamic predicate and an existing predicate with the same name and arity.

609 During loading: all definitions of a predicate must be compiled either with or without debug option.

610 During loading of dynamic predicate: all definitions must be indexed on the same argument.

611 During loading of dynamic predicate: all definitions must have the same

modes.

612 The specified predicate changed from static to dynamic. This may cause problems for existing calls of the predicate from static code.

620 During loading: BIM_Prolog and the object file have a different version number.

621 Unable to open the specified file.

622 Unable to refind the previous position in the specified file.

623 Skipping bad compiler option.

624 The specified file cannot be consulted with this system.

630 Restore option needs a file name to restore from.

640 Unable to open this file for state saving.

641 Unable to open saved state file.

642 Unable to find the initialised data for thge specified file.

651 Too many source files active.

652 Not enough memory for keeping source line information.

653 This file has already been consulted; source line information may be incorrect.

681 Not enough resources for starting the window.

682 Could not fork off the window.

683 Monitor window has died.

684 Debugger window has died.

685 Cannot execute monitor.

686 Cannot execute debugger.

General errors

700 Illegal call : unknown predicate.

701 This predicate cannot be called.

702 A too complex structure was encountered during garbage collection.

703 Unable to open the BIM_Prolog system file.

710 The hashtable used when looking up constants is full.
711 The table of constants is full.
712 The hashtable used when looking up functors is full.
713 The table of functors is full.
714 The table of characters is full.
715 Not enough code space recollected.
716 The table of modules is full.
717 Execution stopped : overflow of the local stack.
718 Execution stopped : overflow of the heap.
719 The hash table of the internal data base is full.
720 The internal data base heap is full.
721 The internal data base variable table is full.

730 The structure is too deeply nested. A maximum depth of 500 is allowed.

740 There are too many clauses for this predicate .
741 Not enough memory for dynamic hash table.

750 Undefined input argument.
751 Non-undefined output argument.

760 Input mode error in argument.
761 Output mode error in argument.

A.3 Messages from External Language Interface

800	External interface type conflict in one parameter of this predicate.
801	Uninstantiated input parameter for external predicate or function.
802	External output parameter of this predicate must be instantiated to a list.
803	Untyped parameter of external predicate has illegal type.
804	External parameter of structure array for this predicate must be a non-empty list or a structured term.
805	External parameter of structure list for this predicate must be flat and ending on nil.
806	Unification upon return from external function has failed.
807	Too many parameters for external function call.
808	Not enough memory for external interface buffer.
809	The specified external routine is not a predicate.
810	The specified external routine is not a function.
811	Parameters for this predicate cannot have the list structure.
812	Untyped input parameters for this predicate are not allowed.
813	An output parameter of this predicate was instantiated to the wrong type.
814	An output parameter of this predicate has not been instantiated.
815	Not enough memory for output array parameter.
816	Try to call something that is not a predicate.
817	Not enough memory for external call of a predicate.
818	Try to find a next solution or end a non-active iteration.
819	The specified string could not be saved.
820	Not enough memory during external module load.
821	Dynamic linking has failed.
822	Unable to open temporary external load module.
823	Too many external predicates.
824	Too many external parameter declarations.
825	Trying to redefine this predicate as external predicate.
826	External routine not loaded.
827	Too long list of objects to link.
828	Too many external protected terms.

829 External protected terms will be destroyed.

836 The specified call required too much memory.
837 External output arguments cannot be protected during garbage collection.

840 Arithmetic on pointers cannot succeed.
841 Atoms not allowed as expression arguments.
842 Functors not allowed as expression arguments.
843 Illegal term as expression arguments.
844 Illegal type for the specified term.
845 The specified argument cannot be retrieved.
846 Too many external protected terms.
847 External protected terms will be destroyed.

850 Not enough memory for external predicates.
851 Dynamic linking has failed.
852 Unable to open temporary external load module.
853 Not enough memory for dynamic linking.
854 Not enough memory for restoring external predicates.
855 The specified predicate cannot be redefined.
856 The specified routine cannot be loaded.
857 The specified symbol table is incompatible with the running BIM_Prolog.

A.4 Messages from Debugger

900	Bad (list of) port name(s) in this argument.
901	Give predicates in the form name/arity .
902	Unknown predicate.
903	Unknown predicate.
904	Bad predicate specification.
905	The first argument must be an integer between 0 and 2.
906	Unable to open temporary files for the debugger.
907	Not that many lines in the trace.
908	No trace has been recorded for analysis.
909	Too many subgoals in predicate to analyze.
910	Not enough memory to keep more lines in the trace (aborting keeptrace).
911	This file does not contain the required source line information.
912	This source file has not yet been consulted.
913	Not enough memory to set more break points.
914	Too many defined aliases.
915	Too many user defined commands.
916	Bad argument type specifier.
917	Bad argument type modifier.
918	Cannot read the specified source file.
919	No spies set.

B.1 Bibliography

1. Bratko [I.].
Prolog Programming for Artificial Intelligence. Addison-Wesley Publishing Company, 1986.

2. Campbell [J.A.].
Implementations of Prolog. Ellis Horwood Ltd, 1984.

3. Clocksin [W.F.] and Mellish [C.S.].
Programming in Prolog. Springer Verlag, 1981.

4. Coelho [H.], Cotta [J.C.] and Pereira [L.M.].
How To Solve It With Prolog. Ministerio da Habitacao e Obras Publicas Laboratorio Nacional de Engenharia Civil, Lisbon, Portugal, 1982 (3rd Edition).

5. Gray [P.M.D.]. and Lucas [R.J.].
Prolog and databases. Ellis Horwood Ltd, 1988.

6. Hogger [C.J.].
Introduction to Logic Programming. Academic Press, 1984.

7. Kowalski [R.].
Logic for Problem Solving. Artificial Intelligence Series, North Holland, 1979.

8. Shapiro [E.].
Algorithmic Program Debugging. MIT Press, 1982.

9. Sterling [L.] and Shapiro [E.].
The Art of Prolog. Advanced Programming Techniques. MIT Press, 1986.

(This page intentionally left blank.)

C.1 Software Performance Report

BIM will use the comments submitted on this form for the improvement of the ***ProLog*** system and its documentation.

Is this manual understandable, usable and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Did you find inconsistencies between the manual and the behaviour of the system? Please specify.

Did you discover a bug in the system? Please add listings of programs and results.

Name : _____

Organization : _____

Address : _____

_____

_____

Telephone : _____

Date : _____