ProLog by BIM 4.0

October 1993

Reference Manual

Symbols

A

B

D

E

M

N

predicates - Sybase

Q

R

S

Y

Z

ProLog by BIM 4.0

October 1993

Reference Manual

Principal Components

Syntax

Built-in Predicates

Compiler Directives

Modules

External Language Interface

Debugger

Programming Environment

Windowing And Graphics Libraries

Database Interfaces

Prolog
And
Unix
Libraries

Programming Tools And Scripts

Appendix

Edition (15 October 1993)

This edition applies to Release 4.0 and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure that you are using the correct edition for the level of the product.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

BIM
Prolog department
Kwikstraat 4
B-3078 Everberg
Belgium

When you send information, you grant BIM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. The edition does not change when an update is incorporated.

Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes.

Edition		2.2
Edition	July 1988	2.3
Edition	March 1989	2.4
Update	February 1990	2.5
Edition	November 1990	3.0
Update	December 1991	3.1
Update	February 1992	3.1.1
Edition	October 1993	4.0

Credits and Acknowledgments

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 and service marks

The following terms, denoted by a double asterisk (**), used in this publication, are trademarks of:

ProLog by BIM	BIM
OSF/Motif, Motif	The Open Software Foundation, Inc.
X Toolkit Intrinsics (Xt)	Massachusetts Institute of Technology
X library (Xlib)	Massachusetts Institute of Technology
X11	Massachusetts Institute of Technology
ORACLE	Oracle Corp.
GNU Emacs	Free Software Foundation, Inc.
DEC	Digital Equipment Corporation
Quintus Prolog	Quintus Corporation
LPA, MacProlog	Logic Programming Associates, Ltd
DELPHIA-PROLOG	DELPHIA
SUN	SUN Microsystems
SYBASE	SYBASE Inc.
SunOS	SUN Microsystems
SPARC	SUN Microsystems
UNIX	AT & T

Chapter 1
Preface

This manual describes ***ProLog by BIM***, a system that implements a standard version of the Prolog language running under SunOS** on the SPARC**.

This manual provides reference information for the application programmer who will develop Prolog applications on the SPARC* platform. This manual is not intended to be a user manual or a tutorial for the Prolog language. Readers who are not familiar with Prolog should refer to the *Bibliography* for references to introductory material on Prolog.

The manual is structured as follows:

- Index (the index includes a complete list of ***ProLog by BIM*** predicates under the heading "predicates - ...").
- Introduction to the ***ProLog by BIM*** system.
- PART 1 - a description of the principal components of ***ProLog by BIM***.
- PART 2 - a description of the supported Prolog syntaxes.
- PART 3 - a description of all built-in predicates.
- PART 4 - a description of all directives.
- PART 5 - how to use modules in ***ProLog by BIM***.
- PART 6 - a description of the external language interface.
- PART 7 - how to use the debugger.
- PART 8 - how to use the windowing environment.
- PART 9 - a description of the windowing and graphics interfaces.
- PART 10 - a description of the database interface.
- PART 11 - a description of the Prolog and UNIX utility libraries
- PART 12 - a description of the Prolog programming tools and scripts.
- PART 13 - appendix: Error Messages - Bibliography - Reader's Comments.

Related to this work are the following manuals:

- CARMEN: a description of the Carmen GUI generator.
- User's Guide: a guide for advanced use of ***ProLog by BIM***.
- Installation Guide for ***ProLog by BIM***.

Chapter 2
Introduction to ProLog by BIM

ProLog by BIM is a high performance implementation of the Prolog language, combined with a powerful programming environment, especially designed for the SPARC running UNIX.

Prolog

Based on Horn clause logic, Prolog incorporates the high-level mechanisms required for the development of advanced information processing systems: flexible pattern matching, easy construction and manipulation of general data structures, and a search execution strategy based on backtracking.

A Prolog program consists of sets of facts and rules that describe the application in a concise, declarative and clear way. Execution of the program corresponds to deduction on basis of the facts, controlled by rules, to generate solutions for a given problem.

Initially, the Prolog language was exclusively used for artificial intelligence (AI) applications such as expert systems, knowledge bases, and natural language processing, areas where a simulation of human intelligence was required; applied research and advanced experiments have shown a far wider applicability of the language, especially in the world of industry. ***ProLog by BIM*** is designed to meet the requirements of industrial developers demanding a professional Prolog implementation.

ProLog by BIM

ProLog by BIM and its programming environment combine the potential of a symbolic processing language with the state of the art of compiler building and software engineering environments for developers of "real–life" applications.

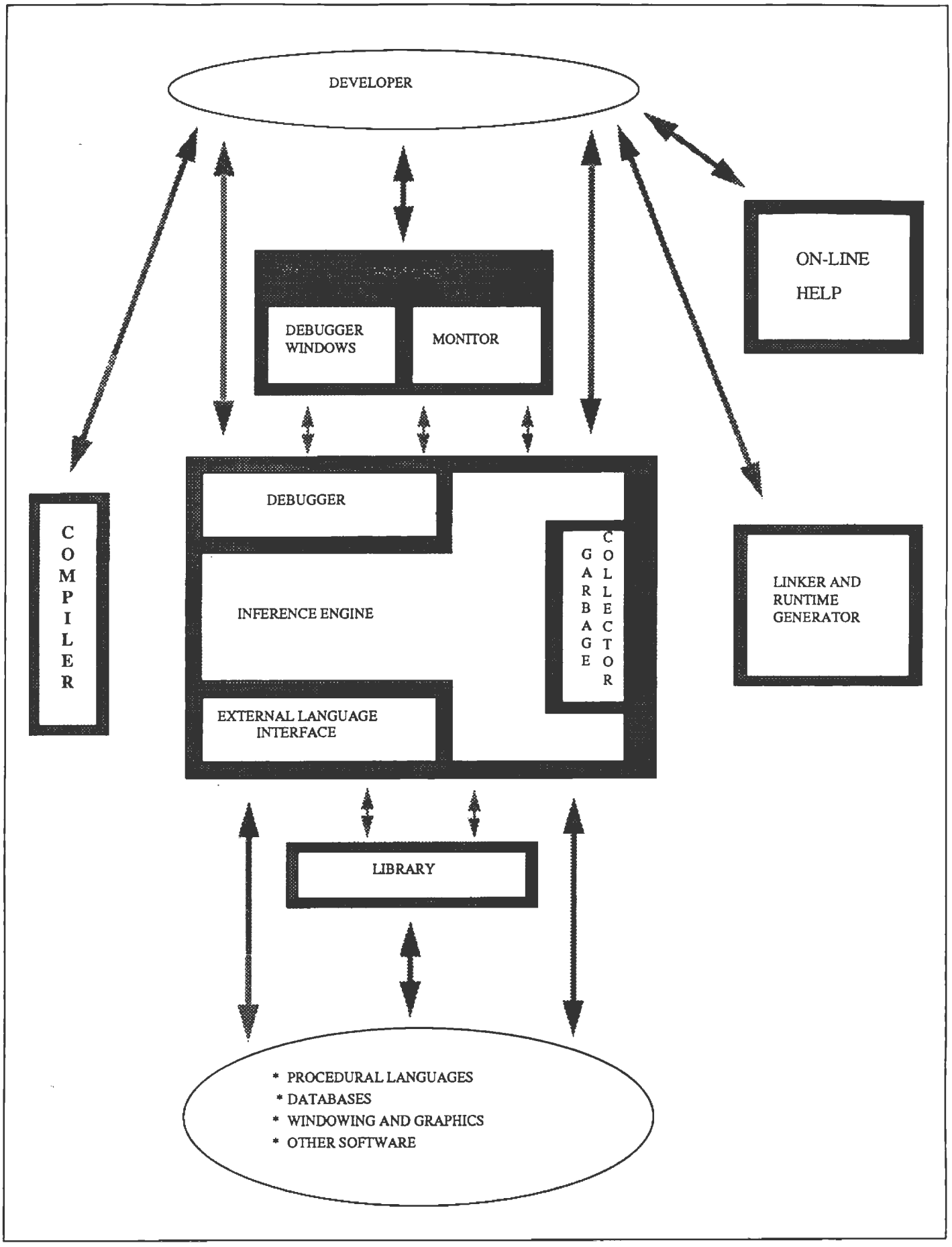
The environment of ***ProLog by BIM*** is composed of:

The inference engine

The inference engine constitutes the kernel of Prolog in which facts and rules can be asserted or loaded from file and goals can be executed.

The compiler

The compiler is based on the concepts of the Warren Abstract Machine (WAM). A Prolog source program is compiled to intermediary code, which can then be interpreted and debugged or further compiled to Assembler for faster processing. This approach guarantees a high performance, while still maintaining the flexibility needed for incremental application building and prototyping.


DEVELOPER
ON-LINE
HELP
DEBUGGER
WINDOWS
MONITOR
DEBUGGER
COMPILER
INFERENCE ENGINE
GARBAGE COLLECTOR
EXTERNAL LANGUAGE
INTERFACE
LINKER AND
RUNTIME
GENERATOR
LIBRARY
* PROCEDURAL LANGUAGES
* DATABASES
* WINDOWING AND GRAPHICS
* OTHER SOFTWARE

The garbage collector

The garbage collector controls all the data structures manipulated by the system and prevents memory overflows by supervising the automatic assignment and release of memory areas.

The debugger

The debugger allows the developer to monitor the execution of programs and to quickly localize possible errors.

The windowing environment

Composed of the debugger control and monitor windows, the windowing environment allows the developer to easily manipulate the inference engine, the debugger and the compiler, through the use of pop-up windows and control buttons.

The external language interface

The external language interface gives the developer full access to external software from within Prolog.

The library

The *external packages library* provides the developer with a large set of predicates giving unrestricted access to databases and windowing systems.

The *Prolog source library* contains a large set of predicates to be loaded into ***ProLog by BIM***. Most of the predicates are provided with the source code. They are mostly issued from the public domain. The author names (R.A. O'Keefe, K. Johnson, D. Bowen, D.H.D. Warren, F. Pereira, L. Byrd, A. Porto, M. Gehrs and many others) can be found in the library source files. *Neither BIM nor any of the authors are responsible for any of the provided programs.*

The linker

The linker allows the developer to create customized Prolog environments, and to generate stand-alone end-user applications.

The Graphical User Interface Generator

Carmen is a WYSIWYG user interface generator which directly generates Prolog code bypassing tedious low-level coding.

The on-line help system

The on-line help system gives an easy access to an on-line version of the manual.

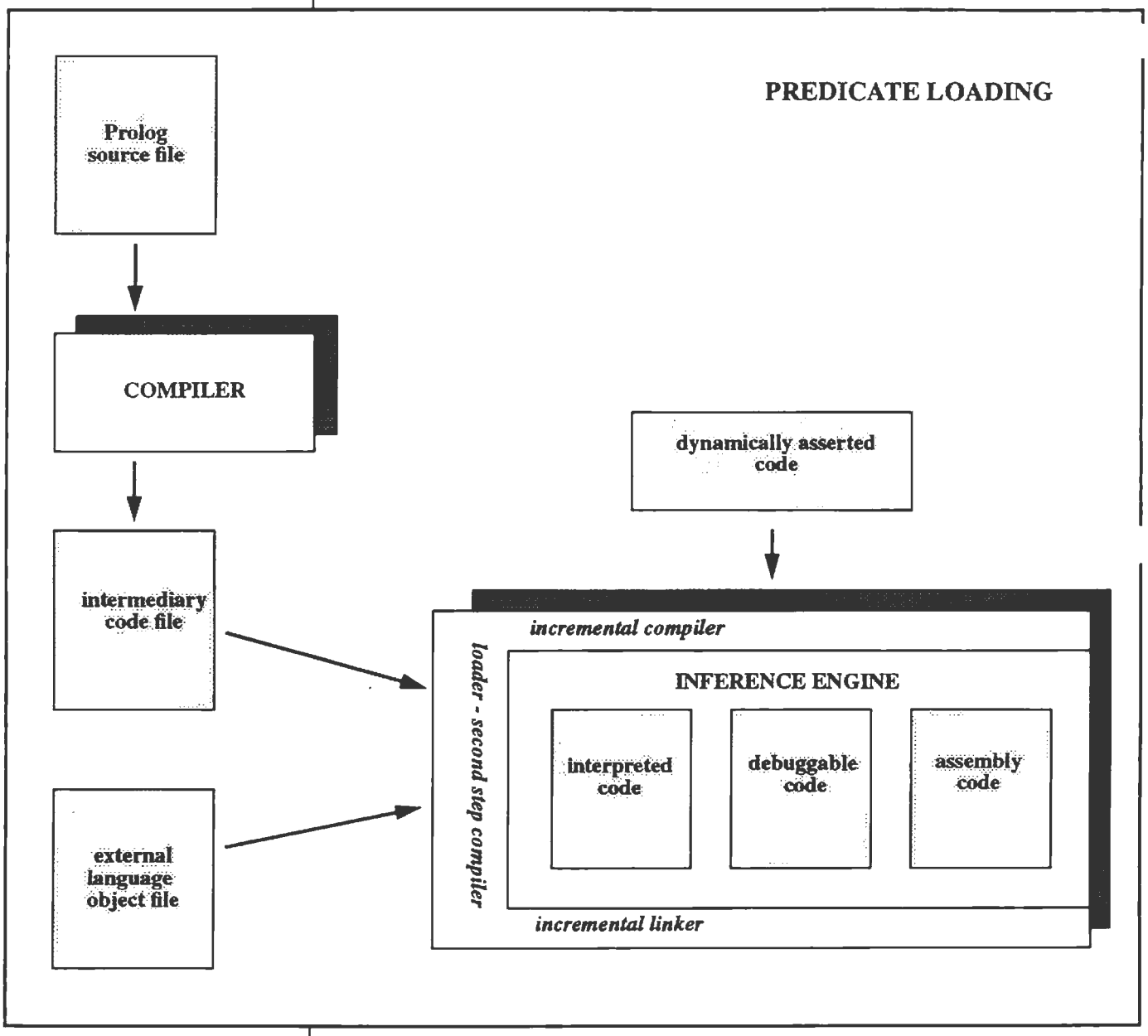
The license server

The license server implements a floating license scheme which enables to run ***ProLog by BIM*** on any machine in the network.

In order to be performed, Prolog predicates must be loaded into the inference engine.

Predicates can be entered during a user session by using assert predicates, or they can be loaded from Prolog files.

In the former case, predicates are compiled by the assert predicates into intermediary code and stored into the engine database.



In the latter case, Prolog source files are compiled, by the ***ProLog by BIM*** compiler, into intermediary code files which are, in turn, loaded into the inference engine. During the load phase, the Prolog loader distinguishes dynamic and debug predicates from static predicates. The first two are stored in intermediary code format into the Prolog database while static predicates are further compiled into assembly code.

The main difference lies in the fact that predicates in intermediary code can be dynamically asserted and retracted while predicates further compiled into assembly code will execute faster, but cannot be dynamically asserted or retracted.

During the load of Prolog files referring to routines defined in external languages, the corresponding external language object files are incrementally linked to the inference engine.

Chapter 3
Getting started

Installation

ProLog by BIM can be installed at any convenient location. See the "***ProLog by BIM*** *Installation Guide*" for further information. You need to know the path name of the installation directory in order to use ***ProLog by BIM***.

Preparing the user environment

Once ***ProLog by BIM*** is installed (please refer to the ***ProLog by BIM*** *Installation Guide* for further information), the user environment must be prepared. It is **mandatory** to set the environment variable BIM_PROLOG_DIR to the pathname of the installation directory and it is advisable to modify the environment variable PATH. How to set these variables depends on the shell you are using.
(typing "echo $SHELL" shows your default shell)

If you are using the Bourne-shell (sh):

To set the environment variable BIM_PROLOG_DIR (mandatory):

```
$ BIM_PROLOG_DIR=<root directory of ProLog by BIM>
$ export BIM_PROLOG_DIR
```

To modify the environment variable PATH:

```
$ PATH=$PATH:$BIM_PROLOG_DIR/bin
$ export PATH
```

It is advised to add the above commands to the file $HOME/.profile for convenience.

If you are using the C-shell (csh):

To set the environment variable BIM_PROLOG_DIR (mandatory):

```
% setenv BIM_PROLOG_DIR <root directory of ProLog by BIM>
```

To modify the environment variable PATH:

```
% set path = ($path $BIM_PROLOG_DIR/bin)
```

It is advised to add the above commands to the file ~/.cshrc for convenience.

There are some other environment variables you may want to set. See the information on libraries (BIM_PROLOG_LIB) and on licences (BIM_LICENSE_HOST). To set the defaults of the windowing environment, one can use the mechanisms as described in *"Programming Environment - Defaults"*.

Invoking ProLog by BIM

To invoke ***ProLog by BIM*** type:

```
1> $BIM_PROLOG_DIR/bin/BIMprolog
```

or if you modified the path you can use

```
1> BIMprolog
```

This command uses the default installation parameters. For modification of the default installation see the "***ProLog by BIM Installation Guide***".

Checking the installation

Compatibility with Edinburgh

If you have a correct installation and a valid license (i.e. the license server is running) and you invoke BIMprolog, a banner is shown followed by the Prolog system prompt.

If you prefer an Edinburgh compatible system, the prompt should be:

```
1> BIMprolog
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993

?-
```

Enter the query true and check the answer:

```
1> BIMprolog
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993

?- true.
Yes
?-
```

Please consult the " ***ProLog by BIM Installation Guide***" if you get a different behavior and want to use an Edinburgh compatible system as the default installation.

BIM installation

With the 'native' installation you get the prompt:

```
1> BIMprolog
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993

>
```

Switching the mode

The prompt "?- " indicates thatthe system is expecting queries. The prompt "> " indicates that the system accepts both definitions and queries, the latter must in this case be preceded with a "?-". Switching between the two modes is done with a please/2 option.

```
> ?- write(ok).
ok
> ?-please(querymode,on).
?- please(querymode,off).
>
```

Note that only the querymode is changed. Neither the syntax nor the reporting of solutions are influenced.

Note that the query

```
?- please(querymode,off).
>
```

is similar to the query

```
?- [user].
>
```

in other systems.

Starting the system and the monitor

In a window environment you should consider using the monitor. See the section on the monitor about what it can do for you.

To start the system and the monitor together use the command line option -Pem+

```
1> BIMprolog -Pem+
```

The Prolog system is started and the main monitor window will appear.

Alternatively, you can start the monitor afterwards with the query:

```
?- please(em,on).
```

Using the compiler

The system comes with a separate compiler. This compiler is invoked by the system whenever a file need to be compiled. It can also be invoked explicitly.

The compiler is invoked with the command:

```
1> $BIM_PROLOG_DIR/bin/BIMpcomp <list of files>
```

or if you have extended your path you can use:

```
1> BIMpcomp <list of files>
```

Starting the debugger

The development environment contains a GUI for the debugger. For a discussion of the debugger see "*Debugger*".

To start the debugger at system start-up time use the command line option -Ped+:

```
1> BIMprolog -Ped+
```

The debugger can be started from the monitor: switch the please option "ed" on with the mouse.

The debugger can be started with a query too:

```
?- please(ed,on).
```

Files must be compiled for debugging in order to use the debugger. You can compile files for debugging by invoking the compiler with the command line switch -d:

```
1> BIMpcomp -d <list of files>
```

Alternatively, you can add this switch in the call to the consult predicate:

```
?- ['-d example.pl'].
```

If the monitor is running, you can set the -d compiler switch and consult the file with a few mouse clicks.

Principal Components

Principal Components

Chapter 1
The Engine: BIMprolog

1.1 Invoking the engine

One main executable of ***ProLog by BIM*** is the inference engine which executes the Prolog code.

Starting

The ***ProLog by BIM*** 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 option>)* |
(<Prolog option>)* - (<user-defined options>)*

<Prolog option> ==>
-<category><name> |
-<category><name><value> |
<file specification>

<category> ==>
table options (T) |
please options (P) |
debug options (D) |
compiler options (C) |
special options

<name> ==>
A full description of the option names, category per category, is given below.

<value> ==>
+ (on) | - (off) | <number> | <number><scale>

<scale> ==>
K | k | M | m

<file specification> ==>
This can be any valid file name.

<user-defined options> ==>
All options after the delimiter (the "-" sign between blanks) are considered to be user-defined options. There is no limitation on the syntax.

When the engine is started, the table options, specified on the command-line, are gathered and handled.

Then, a user initialization is issued. In the default case, the engine searches for a file with name '.pro', first in the current working directory and if not found there, in the user's home directory. If such a file is found, it is loaded. This behavior can be overruled by the -f option. When the -f option is specified with an argument, the argument is treated as a file name, and that file is loaded. If no argument is specified, no user initialization takes place.

After this user initialization, the other command line arguments are handled in the order in which they are specified. Options are executed and files are loaded.

When all command line arguments are handled, the engine hands over control to the user.

The user-defined options are available to the application through the predicates **argc/1, argv/1, argv/2.**

Ending

The engine can be halted with the built-in predicates **halt/0,1** or by closing the input stream with ^D.

Exit status

The engine process terminates with an exit status that gives information about the reason for termination.

- When terminated with halt/1, the given argument is used as exit status code.
- When ending with halt/0 or ^D, a default success exit status is returned. In Unix Systems, this default exit status is 0.
- In all other cases, a special condition terminated the engine and this is represented by a corresponding exit code.

Exit status codes

Below is a list of possible exit status codes with conditions. Exit status codes between 0 and 99 are reserved for ***ProLog by BIM***. User-defined codes should use values outside that range in order to be portable across different releases of ***ProLog by BIM***.

Exit	Condition
1	Failed process set-up.
10	Terminated by a signal from the Operating System.
20	Non-sufficient system license (preceded by a message).
25	Unsuited environment for starting the engine.
31-33	Terminated during a restore operation from a saved state.
35	Failed execution of a run-time system top-level predicate.
41-43	Failed system initialization.
45	Failed user-defined initialization (see -f option).
81	Fatal read error.
82	Fatal overflow error.
83	Fatal memory fault error.

Help on the engine

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

```
% BIMprolog -help
```

1.2 Managing the internal tables

The ***ProLog by BIM*** engine has a set of internal tables to store the user rules and user data and to store system data when running a query. All these tables are automatically managed by the system. Whenever appropriate, they can expand or shrink as necessary.

Certain tables, whose size is to a higher degree dependent on the application or its execution, are made visible to and controllable by the user. Parameters of these tables can be changed by specifying table options on the command line -T, or in a running engine with the **table/2** built-in.

Command line table option

The syntax for the command line table options is given below in a BNF-like format.

<TableOption>	==>	-T<TableId><ParamSettingList>
<TableId>	==>	H | S | T | D | F | I | C | R | B
<ParamSettingList>	==>	<ParamSetting> | <ParamSetting>,<ParamSettingList>
<ParamSetting>	==>	<ParamId><ParamValue>
<ParamValue>	==>	<Integer> | <Integer><Scale>
<ParamId>	==>	b | e | s | l
<Scale>	==>	K | k | M | m

A parameter setting is given as the identification letter for the parameter immediately followed by the value. The value is either a simple integer number, or an integer number followed by a scale factor. The value denotes the number of entries that are required.

During system set-up, the table options are executed before any other options and before the loading of any file.

Table description

The following list gives a short description for each of the ***ProLog by BIM*** tables.

Heap (H)
: is 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.

Compiled Code (C)
: contains the compiled and fully optimized code which is generated for static predicates. Code that becomes obsolete is removed during garbage collection.

Interpreted Code (C)
: contains the code which is generated for dynamic predicates. Code that becomes obsolete is removed during garbage collection.

Data (D)
: contains one entry for each atom, real and pointer. Entries that become obsolete are removed during garbage collection.

Functors (F)
: contains one entry for each functor that has been used.

Record Keys (R)
: contains one entry for each tuple (key1,key2) used in a record predicate. Erased keys are removed immediately.

Backup Heap (B)
: contains structured terms that are recorded. Terms associated with erased keys are removed during garbage collection.

Text (T)
: contains string texts, essentially used in atoms and the external language interface. Strings that become obsolete are removed during garbage collection.

System Tables
: are tables whose size is independent of the user's application. These tables and their management are kept invisible to the user. They expand or shrink automatically, following the resource requirements of the system in a transparent way unless expansion becomes impossible due to a lack of available (virtual) memory. In this case an overflow occurs.

Table parameters

The control of a table is handled by the following parameters.

m (minimum)	Minimum size, fixed by the system.
b (base)	Base size of the table.
e (expansion)	Required degree of expansion (see below).
s (shrink)	Shrink requirement on garbage collection (see below).
l (limit)	Limit size as set by the user.
u (upperlimit)	Upper limit size, fixed by the system.

The **minimum size** of a table is determined by the system and cannot be changed by the user. It is the minimum required size for the engine to be able to run.

The **base size** is the actual size with which the table is first created at start-up of the engine. It cannot be less than the minimum size. Certain tables allow modification of this base size, resulting in a change of the actual size. Such tables are called resizable.

The **expansion** parameter indicates in what way the table management must be done. A table, incorporating a garbage collector, can either be expanded or cleaned up when it has not enough free storage space left. The expansion parameter indicates whether the system should preferably use garbage collection or expansion, to create more free space. Setting it to 0 means that garbage collection is highly preferred. A value of 100 indicates maximum preference for expansion. The advantage of garbage collection is less memo consumption. The advantage of expansion is, at first, higher execution speed. However, this rule should be taken with some care. Higher memory consumption (caused by continuous expansion), can result in speed slow down because of an increase in swapping.

The **shrink** parameter indicates whether memory areas that are freed on garbage collection, must be deallocated or not. The parameter can take two values: 1 for deallocation, 0 for no deallocation. This parameter has only effect on tables that are able to deallocate memory areas.

The **limit size** is the maximum size that the table can occupy. Once its size becomes larger than this limit, the table will not expand anymore. The next time that the table must be expanded to be able to continue, a fatal overflow error occurs.

The **upper limit size** of the table is determined by the system and cannot be changed. It depends on the internal table representation.

Table control

Not all parameters are applicable for all tables. Parameters that are specified but which are not applicable, are silently ignored. Certain parameter values will be interpreted as approximations, and can be adjusted for internal restrictions.

Example

```
% BIMprolog -THb10K -THe1 -TH15M
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993

?- statistics('H').
H Heap : 10240u alloc'd ( 51200b)  23u used ( 0%) 10217u avail
Yes
```

The base size for the heap is set to 10 KB. The expansion factor is set to 1, meaning that high preference is given to garbage collection. The limit size is set to 5 MB. With the statistics built-in predicates, the actual allocated and used size can be retrieved.

The predicate **table/2** can be used in different ways to interact with the ***ProLog by BIM*** tables (see **table/2** in "*Built-in Predicates - Engine Manipulation*")

Example

Table/2 is used to set and retrieve the values of the table settings.

```
?- table('H',_l).
  _l = [10240,1,0,5242880]
Yes
?- table('H',[_,_,_,10000000]),table('H',_L).
  _L = [10240,1,0,10000000]
Yes
```

Table/2 can be used to force a garbage collection.

```
?- table('H',collect).

*** Table Garbage Collection ***
H Heap  :  10240u alloc'd ( 51200b)  21u used ( 0%) 10219u avail
H Heap  :  10240u alloc'd ( 51200b)  14u used ( 0%  10226u avail
*** Table Garbage Collection *** Done in 0.000 s
Yes
```

For resizable tables, table/2 can be used to do a resize operation.

```
?- table('H',b15k).
*** Table Garbage Collection ***
H Heap : 10240u alloc'd ( 51200b)   31u used ( 0%) 10209u avail
H Heap : 15360u alloc'd ( 76800b)   14u used ( 0%) 15346u avail
*** Table Garbage Collection *** Done in 0.016 s
Yes
```

Default settings

The default settings of the parameters for each of the tables is shown in the table below. For each of the tables is shown if there exists a garbage collector(GC), if the table can be expanded (Exp) and if it is a resizable table (Rsz). Also the default values for the different parameters are listed.

Only the values for the parameters b, e, s and l can be changed. A shaded box means that the corresponding value can be changed either at installation time or in a running system. Changing the other values (the not shaded boxes) will be visible in the retrieval of the parameter values but will have no effect on the management of the table.

It is possible to change the default settings at installation time of ***ProLog by BIM*** (see "*Installation Guide*").

Table	GC	Exp	Rsz	m	b	e	s	l	u
H Heap	Y	Y	Y	4k	32k	25	0	1m	819m
S Stack	N	Y	Y	4k	32k	100	0	1m	819m
T Text	Y	Y	N	NA	0k	1	0	32m	32m
D Data	Y	Y	N	NA	0k	1	0	16m	16m
F Functor	N	Y	N	NA	0k	100	0	500k	8m
I Interpr Code	Y	Y	N	NA	0k	1	0	20m	20m
C Comp code	Y	Y	N	NA	0k	1	0	20m	20m
R Record keys	N	Y	N	1k	2k	100	0	1m	256m
B Backup heap	Y	Y	N	1k	8k	1	0	1m	819m

Changing the default settings

The default settings for the table parameters are chosen for optimal performance in average cases. The parameters of the tables can be set by using the engine command line option **-T**, or when the engine is running, using the **table/2** built-in. The usage of the table can be monitored by using the **statistics/4,3,1,0** predicates.

Common cases where one wants to change the settings are:

- If the initial size of the table is known to be insufficient one can set the base to a larger value preventing successive garbage collections and/or expansions.
- If the initial size of the table is known to be too large one can set the base to a smaller value. The smaller table will be better filled
- The limit parameter is set to a value which is on average the maximum that will ever be used. The limit parameter will prevent a table from growing indefinitely which could be the case with erroneous programs.
- If one has finished one section of his program which had a heavy usage of a particular table it is often a good idea to initiate a Garbage Collection and force a resize of the table.

The following constraints apply when changing the default settings:

- The base b cannot be set to a value smaller than the minimum m.
- The limit l must be bigger than the minimum m. Values higher than the upper limit are accepted, but the upper limit has preference.
- Once the table has been allocated, the base b can only be modified if the table is resizable.
- Once the table has been allocated, the limit l cannot be made smaller than the actual size.
- The expansion e must always be an integer between 0 and 100.
- The shrink s must always be 0 or 1.
- Other parameters must be positive (possibly scaled) integers. The meaning of the scale factors is:

K (k)	kilo	1024 entries
M (m)	mega	1,048,576 (= 1024*1024) entries

1.3 Please options

The behavior of the engine is controlled by a number of please switches. The default settings for the please switches are fixed at installation time and define the default behavior of the engine. These defaults can be changed by using the engine command line option **-P**, or in a running engine by using the **please/2** built-in (see *"Built-in Predicates - Engine Manipulation"*).

The set of please switches can be divided into three categories: binary switches, integer switches and string switches. Each category is explained below. For each switch its abbreviated name, its full name and its default value are given. The default value is specified for a compatibility installation and between brackets for a native installation (see *"Installation Guide"*). In your installation however, the values can be different. To know the current values in your system, start the engine and execute the query '?- please(_,_).'.

The syntax for the command-line please options is given below in a BNF-like format.

<PleaseOption>	==>	<PlsBinarySwitch> | <PlsIntegerSwitch> | <PlsStringSwitch>
<PlsBinarySwitch>	==>	-P<PlsBinaryId><PlsBinaryValue>
<PlsBinaryValue>	==>	+ | | -
<PlsIntegerSwitch>	==>	-P<PlsIntegerId><integer>
<PlsStringSwitch>	==>	-P<PlsStringId>=<string>

The binary switches can only take 2 values: on or off. '+' turns on the switch and '-' turns off the switch. When no value is specified, the value of the switch is toggled.

Please options with a binary value are:

ae : atomescape *on*
Controls the escape character.
The "\" (escape) character is active when atomescape is on.

c : compatibility *on (off)*
Controls the syntax accepted by the parser.
When on, the parser is compatible with the DEC-10 de-facto standard. In the other case, it accepts native ProLog syntax.

d : debugcode *off*
Indicates if debug code has to be generated for asserted predicates.
When on, newly asserted definitions for previously unspecified predicates are compiled to debug code.

e : eval *off (on)*
Controls in-line evaluation.
When on, the interactive compiler will expand ?/1 and '=?'/1 to evaluable expressions (see *"Built-in Predicates - Expression Evaluation"*). In the other case, they are treated as compound terms.

ed : envdebugger *off*
Controls the environment debugger (see "*Programming Environment*").

em : envmonitor *off*
Controls the environment monitor (see "*Programming Environment*").

h : hide *off*
Indicates if asserted predicates must be made hidden.
The hide switch only affects predicates without definitions or declarations. When on, asserting a definition for a previously unspecified predicate, will make that predicate hidden.

la : reloadall *on*
Controls the behavior of the reload and reconsult predicates.
When on, the reload predicates behave as the reload_predicates built-ins. Otherwise, the reload predicates are equal to the reload_file built-ins.

q : querymode *on (off)*
Controls the inputmode.
Querymode on means that terms entered will be interpreted as queries and executed. In this case the prompt will be '?-'. If querymode is off, the prompt will be '> ', and the terms that are entered are added to the in-core database. This mode is also known as the assertmode. In this mode, queries can be entered by preceding them with '?-' .

r : recovery *off*
Controls error recovery.
When off and an error <errno> is issued, an error message will be shown on stderr. If the error recovery is on, an **throw (<err-no>)** is executed.
(see also **err_catch/5**)

rf : readeoffail *off (on)*
Determines the behavior when read encounters end-of-file.
When on, the call of a read which reaches end-of-file fails. When off, such a call returns the value of the ra-switch or the rc-switch depending on the predicate used.

s : showsolution *on (off)*
Specifies whether the solution of a query has to be printed out or not.
When off, all solutions for the query will be generated. When on, the solutions of the queries are printed out and the system prompts the user after each solution for a <CR> to abort the query or a ";" to search the next solution. Queries that appear in a file will be executed with showsolution off.

sw : syswarn *on*
Controls the system diagnostic messages.
When on, general system messages will be printed. Syswarn is by default on in a development system and off in run-time systems.

tw : tablewarn *on*
Controls the table operation warnings.
When off, no table garbage collection messages will be printed.

w : warn *on*
Controls output of warnings and error messages.
When off, no error messages will be printed.
(see also "*Built-in Predicates - Error Handling*")

wf : writeflush *on*
Determines whether print operations have to be followed by a clearing of the output buffer.

wm : writemodule *off*
Determines if the explicit module qualification is to be specified when writing out a term.

wp : writeprefix *off*
Determines how operators are written out.
If writeprefix is on, operators will be written in regular functor notation. When writeprefix is off, operators will be written out according to their specificatio.

wq : writequotes *off*
Determines the usage of quotes in printing. These are necessary if the printed terms must be readable by ***ProLog by BIM***.
When off, nothing is quoted. When on, everything which needs quotes to be able to be read in again, is quoted.

Please options with an integer value are:

rc : readeofchar *-1*
Controls the end-of-file ascii-code.
The end-of-file ascii-code will be returned when a read of a character is attempted at the end of the file. This switch is only effective when the readeoffail switch is off.

tt : tabletime *0*
Returns or sets the time spent in table operations. The value is expressed in milliseconds.

wd : writedepth *-1*
Controls the printing of nested terms. The printing of a structured term is limited to the specified levels of nesting. All subterms of that level are printe' as "...". When the integer is a negative value, the limitation is removed.

Please options with a string argument are:

fr : formatreal *'%.15e'*
Controls the format in which a real is printed out. The atom specifies the output format for a real. It can be any format that may be used in a formatted print. For possible formats, see **printf/2,3**.

ra : readeofatom *end_of_file*
Controls the end-of-file atom. The end-of-file atom will be returned when a read of an atom is attempted at the end of the file. This switch is only effective when the readeoffail switch is off.

1.4 Debug options

The behavior of the debugger is controlled by a number of debug switches. The defaults of these switches can be changed by using the engine command line option **-D**, or in a running engine by using the **debug/2** built-in (see "*The Debugger - Overall Debugger Control*").

The syntax for the command-line debug options is given below in a BNF-like format.

<DebugOption>	==>	<DbgBinarySwitch> | <DbgIntegerSwitch>
<DbgBinarySwitch>	==>	-D<DbgBinaryId><DbgBinaryValue>
<DbgBinaryValue>	==>	+ | | -
<DbgIntegerSwitch>	==>	-D<DbgIntegerId><integer>

The binary switches can only take 2 values: on or off. '+' turns on the switch and '-' turns off the switch. When no value is specified, the value of the switch is toggled.

The table below gives the possible options, their default values and a short explanation (more on this can be found in "*The Debugger - General Concepts*").

c : cutearly	on	Controls the choice point destruction
i : indexed	on	Controls the usage of indexing.
p : prompt	on	Controls the activation of the prompt
td : tracedepth	-1	Allowed depth of the recorded trace.
tr : tracerecord	on	Recording of trace.
wd : writedepth	-1	Nesting depth for output of terms.
wm : writemodule	off	Explicit module qualification in output
wp : writeprefix	off	Usage of prefix functor form in output.
wq : writequotes	off	Usage of quotes in debugger output.

1.5 Controlling the compiler

The default option settings of the compiler are controlled by a number of compiler switches. The defaults of these switches can be changed by using the engine command line option **-C**, or in a running engine by using the **compiler/2** built-in (see "*Built-in Predicates - Program Loading*").

<CompilerOption>	==>	-C<CompOptionId><Value>
<Value>	==>	+ | | -

A full description of the compiler options is given in "*The Compiler - Compiler Options*".

1.6 Special option -f

At start-up, the ***ProLog by BIM*** system initially searches for a file with name '.pro', first in the current working directory and if not found there, in the user's home directory. If such a file is found, it is loaded. This behavior can be overruled by the -f option.

-f[*startfile*]
: When an argument is specified, the argument is treated as a file name, and that file is consulted instead of the *.pro* file. If no argument is given, no user initialization takes place.

Such a file can be used to load an initial set of predicates and to run a number of queries. One typical use is to tailor the ***ProLog by BIM*** engine to your own needs by changing the defaults for the please, compiler and/or debug switches.

Example

```
% BIMprolog -Dp- -Ps+ -Pq+
```

can be obtained by putting the following queries in the initialization file:

```
% cat .pro
?- debug(p,off).
?- please(s,on).
?- please(q,on).
```

1.7 Special option -r

ProLog by BIM offers the opportunity to save a session in a file. This saved state can be restored by invoking the engine with the special command line option -r followed by the file name of the saved state, or with **restart/1.**

-r *file*
Restores the ***ProLog by BIM*** session by using the saved state file *file*.

The default user-defined initialization is ignored during a restore start-up. Using the -f option explicitly, does enable user initialization immediately after the restore.

A full description of this save/restore feature can be found in *"Built-in Predicates - Engine Manipulations"*).

1.8 Consulting files

The files, mentioned on the command line, are loaded into the running engine in the order they appear. Whenever a loaded file contains a query, all solutions to that query are computed immediately when it is encountered. If a query appears in the middle of a file, it is computed before loading the rest of the file.

If a file needs to be compiled, it will be compiled first. The rules for compilation and recompilation are explained in *"Builtin Predicates - Program Loading"*.

The file names mentioned on the command line are used to locate the files. Any file name specification which is accepted by the operating system, can be given as command line option to the engine. The file name specification will be expanded by the operating system before it is interpreted by the engine. Besides that, the location of files can also be specified with the options -L and -H (see next section).

1.9 File name 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 symbols are expanded automatically by ***ProLog by BIM***.

The built-in **expand_path/2** does this expansion explicitly.

The following meta symbols are recognized. They can only be used at the beginning of the file name. The $VAR construction however, may appear anywhere in a file name.

Symbol	Expansion
~	Home directory of the user ($HOME)
~user	Home directory of "user"
$VAR	Value of environment variable VAR

The construction will be replaced by the value of the corresponding environment variable. If this variable is not defined, the construction is replaced by the name of the variable (without the $ sign).

-L*file* ***ProLog by BIM*** library directory

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 current working directory.

-H*file* ***ProLog by BIM*** home directory ($BIM_PROLOG_DIR)

Places where this expansion takes place:

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

Principal Components

Chapter 2
The Compiler: BIMpcomp

2.1 Introduction to the compiler

ProLog by BIM comes with a stand-alone compiler, ***BIMpcomp***. This compiler compiles a file from Prolog into intermediary object code. If the Prolog source file has extension .pro, the intermediary object code file has the same name but with extension .wic, otherwise that name is formed by extending the original file name with the suffix .wic.

All files must be compiled by the compiler prior to loading. When a file is consulted, ***ProLog by BIM*** invokes the compiler if needed, and then, if the compilation was successful, loads the .wic file. For clauses not in a file but who are entered directly or using the predicate assert, another compiler, integrated in the ***ProLog by BIM*** engine, is used.

A file does not need to be compiled before giving it to the engine. ***ProLog by BIM*** checks the .pro and the .wic file, if it exists, to see whether (re)compilation is needed, thus providing a sort of make-facility.

The compiler of ***ProLog by BIM*** may be used in makefiles like any other compiler. It can be used to compile files before starting ***ProLog by BIM***. First, this reduces the memory requirements as this precompilation avoids the need to fork the compiler process. Secondly, precompilation reduces the execution time.

The compiler can also be used to check the syntax of source files.

2.2 Basic rules of compilation

The compiler takes as input a **Prolog source file**, generates intermediate code for it and stores this code into an **intermediary object code file**. When the compilation is not successful an error **listing file** can be generated. The different files involved in the compilation process are:

file.pro	Prolog source file
file.wic	intermediary object code file
file.lis	listing file

The object code file is portable between the ***ProLog by BIM*** implementations of the same release.

The listing file is created whenever errors or warnings occur and the relevant option is set.

Each source file is compiled separately. When compiling one file, the compiler does not have any information about other source files, nor does it have any information of the current state of the engine. The compilation is guided by compiler options and by the previous compilation if the .wic file exists. Term expansion is not available.

A source file may contain any number of **queries** at any place in the file. Queries divide a file in segments. Each segment is compiled separately (as if it were provided in a separate source file). Such queries are compiled by the compiler but are only executed after the preceding intermediary code file segment is loaded. When the file is loaded in the engine the queries are executed before the clauses physically following it in the file are loaded and the queries may contain requests to consult other files. When loading (or compiling) the file *file.pro*, this file is not the current input stream. Queries that invoke read built-ins never read from the file being loaded.

Intermediary object files can be concatenated using any concatenation program that literally appends one file to the end of another (like the Unix program cat). The result is similar to having the separate source files included into one big file, with a (dummy) query inserted after each of the composing files.

2.3 Invoking the compiler

The compiler is called by:

```
% BIMpcomp . <CompilerOption>* <FileSpecification>*
```

<CompilerOption>	==>	(-<OptionName>[<ToggleValue>])
<OptionName>	==>	a | c | d | e | h | l | o | p | w | x | A
<ToggleValue>	==>	+ | -
<FileSpecification>	==>	any valid file name.

For the specification of the files on the command line, the *.pro* extension does not need to be given. If the *.pro* extension is omitted, the compiler looks for a file with a *.pro* extension. If such a file is not found, the given name is taken as the complete file name.

By default, the convention for the name of the intermediary object file is as follows. If the source file has an extension *.pro*, this extension is replaced by the extension *.wic*. If the source file does not have such an extension, the full source name is taken and extended with *.wic*.

If the compilation was not successful, the errors (syntactic or semantic) and warnings are by default printed on standard error. A listing file (whose name is the file name extended with *.lis*) can be generated by setting the -l option in the compilation command. ***BIMpcomp*** displays (on stderr) a message notifying the unsuccessful completion of th compilation (see "*Error Messages*").

The different compiler options are explained in a next section.

2.4 The intermediary object code

ProLog by BIM knows different kinds of Prolog code, each having different behavior and different characteristics. The distinction is made on the predicate level. ***ProLog by BIM*** distinguishes between predicates compiled to static, dynamic and debug code. On top of that, predicates can also be hidden.

The type of code that has to be generated by the compiler, can be controlled from the command line with compiler options or with compiler directives which are included in the source file (see "*Compiler Directives*").

Static code is the default. Without options nor declarations, all predicates in a source program are compiled to static code. Static code is the most optimized code that the compiler generates. When static code is loaded into the engine it is further compiled into performant native code taking into account all the particularities of the targe machine.
Predicates for which static code is generated, are called **static predicates**. Static predicates are limited in the following areas. All clauses of a static predicate need to be presented to the compiler at once. This means that they must be defined in the same segment of a file. Once loaded into the engine - by consulting the file - the static predicate cannot be modified any more (e.g. with retract or assert). Attempts to do so result in error messages and/or warnings. Only by deleting all the current definitions with the **abolish/1,2** built-ins, new definitions can be loaded for a predicate with the same name and arity.
The definition of a static predicate is always hidden and can therefore not be listed with the listing built-ins. Attempts to retrieve the definition of a static clause will fail.
Static predicates will run very fast. They can be indexed on up to three arguments. Indexing is by default performed on the first three arguments but this behavior can be altered by the index/2 and the mode/2 directives (see "*Compiler Directives*").

Dynamic code is code that is interpreted by the engine when it is executed.
Predicates for which this type of code is generated, are called **dynamic predicates**. Dynamic predicates can be modified at runtime by loading extra clauses (see **(re)load/1,2** in "*Built-in Predicates- Program Loading*") or by adding/removing

clauses with assert, retract (see "*Built-in Predicates- In-core Manipulation*").
The clauses of a dynamic predicate can be retrieved with clause and listing built-ins. They are indexed on one argument, by default the first. It is possible to ask for hashing on the indexed argument (see the **index/2** built-in and compiler directive).

Debug code is code which contains additional information for the debugger. Both static and dynamic predicates can be compiled into debug code.
Debug code is interpreted by the engine and the extra information is passed to the debugger (see also "*The Debugger*"). This overhead makes debug code the least performant code of ***ProLog by BIM*** and it should therefore only be used for predicates which are being debugged.
Debug code allows modification of the predicate during execution. This means that static predicates can behave differently when they are compiled for debugging compared to non-debug static code (i.e. assert, clause and retract will succeed on debug coded static predicates but are not allowed on non-debug coded static predicates).

Hidden code is code that can not be retrieved neither with the listing nor with the clause built-ins (see "*Built-in Predicates- In-core Database*"). Static predicates are always hidden. Dynamic predicates are by default not hidden. Debug coded predicates which are traced in the debugger are never hidden.

Besides the above mentioned predicates which are generated by the compiler, the following types also exist in ***ProLog by BIM***.

Built-in predicates are defined by the ***ProLog by BIM*** system and implement all the basic functionalities of the system. Built-in predicates are hidden and they cannot be redefined. Attempts to redefine a built-in predicate will result in error messages.The available built-ins are explained in a separate part of this manual (see "*Built-in Predicates*").

External predicates are predicates which are mapped to external routines by the External Language Interface. When an external predicate is called, its arguments will be converted to the external types and the external routine will be executed. When the routine returns, its output parameters are converted back to the ***ProLog*** internal format and unified with the arguments of the external predicate. (see "*External Language Interface*" for the compiler directives and built-in predicates which allow the management of external predicates).

Database predicates are mapped to relations of an external database. Calling a database predicate will invoke a call to the DBMS. The results of the database query will be returned to ***ProLog*** one by one on backtracking. Arguments are converted in the same way as for external predicates. The ***ProLog by BIM*** system comes with interfaces to a number of relational databases (see "*Database Interfaces*") and can also be interfaced with the ***ProLog by BIM*** dedicated database ***ClauseDB*** which is ideal for medium sized databases and databases with structured data (Please contact BIM if you want more information about this add-on product).

2.5 Compiler options

The behavior of the compiler is controlled by a number of compiler switches. These switches control the three main areas of the compiler: the parsing of the source file, the generation of the intermediary code file and the behavior of the compiler itself.

These compiler switches do not affect the behavior of the engine. Its behavior is controlled by the please switches (see "*Principal Components - The Engine*").

The following enumeration gives an overview of available options. For each option its abbreviated name, its full name and its default value are given (+ stands for on and - for off). The default value is specified for a compatibility installation and between brackets for a native installation (see "*Installation Guide*"). In your installation however, the values can be different. To know the default values in your system, start the compiler with option -H.

a : alldynamic -
All predicates in the source file will be compiled as dynamic predicates.

c : compatibility + (-)
Controls the syntax accepted by the parser.
When compatibility is on, the parser is compatible with the DEC-10 de-facto standard and **Definite Clause Grammar** rules are translated (see "*Syntax - DCG Rules* "). When off, the parser assumes native ***ProLog by BIM*** syntax.

d : debugcode -
All predicates in the source file will be compiled into debug code.

e : in line-evaluation - (+)
Controls in-line evaluation.
If in-line evaluation is on, the compiler interprets ?/1 and '=?'/1 as evaluable expressions and expands these calls (see *"Built-in Predicates - Expression Evaluation"*). In the other case, both terms are treated as functors.

o<file> : outputfile <base>.wic
Output is to be written in <file> instead of in the default file whose name is derived from the source file. The given file name should have .wic as suffix.

h : hidden code -
Hides all predicates defined in the file.

l : listing file -
Controls the rendering of the error messages:.
When off, error messages will be written to the standard error stream. When this option is on, error messages and warnings are redirected into a listing file whose name is the source file name extended with *.lis*. Error messages can be switched on or off with the w switch.

p : operator definitions -
Includes operator declarations in the object file.
When this option is on, loading the compiled file results in the replacement of the current operator definitions of the engine by the operator definitions of this file.
The operator definitions active in the engine are never considered during the compilation. The use of an include file with operator definitions should be considered.

w: warnings +
Controls output of warnings and error messages.
When off, no error messages will be returned. Error messages can be redirected with the l switch.

x : escape character +
Controls the escape character.
The "\"-escape character is active when the option is on.

An : Table size n = integer 1
Controls the size of the internal tables
If *n* is a positive integer, the startup sizes of the internal tables of the compiler are multiplied by *n* to speed up the compilation of larger programs. Otherwise, when *n* is a negative integer, the startup sizes of the internal tables are divided by *-n*, so that the compiler consumes less memory. If the size of the compiler tables is not big enough during compilation, the system increases the size automatically and the compilation is restarted.

Changing the default settings

The default settings for the compiler switches are fixed at installation time and define the default behavior of the compiler.

The default values of the switches can be changed in the engine by using the **compiler/2** built-in. When the compiler is thereafter called from within the engine it will take into account these new options.

The defaults from the installation can be overruled by explicitly setting the options when compiling/consulting the source file (see "*Built-in Predicates- Program Loading*").

Including compiler directives in the source file will overrule the settings of the compiler switches for the current compilation.

Principal Components

Chapter 3
The Linker: BIMlinker

ProLog by BIM offers the possibility to make customized executables or stand-alone applications. The first is useful in development while the latter facilitates the delivery of finished applications to customers.

The tool ***BIMlinker*** (the ***ProLog by BIM*** linker) takes care of this functionality. BIMlinker generates, with the given input, an intermediary file, compiles it and calls cc to make the final link. The three steps can be parameterized or replaced by other commands.

To have a good understanding of this chapter, some knowledge of the link facilities of the operating system and of the final linking tool is required.

3.1 Functionalities of BIMlinker

BIMlinker can generate two types of customized systems: customized development systems or run-time systems. Both types can also be "embedded", in which case the control of the running application is given to an external program.

A **customized development engine** is an executable that, besides all the functionalities of the ***ProLog by BIM*** development system, includes:

- Compiled Prolog code (.pro):
  extending the set of predicates with Prolog clauses.
- Code written in another language (*.o*):
  extending the set of predicates with external routines.
- Initializations:
  changing the default options of the inference engine.

A **runtime** is an optimized executable including the basic ***ProLog*** inference engine extended with the developer's application files. A runtime generally includes:

- Compiled Prolog files (*.wic*):
  the developer's Prolog application files and possible libraries.
- Compiled external language object files:
  the developer's application object files and possible libraries.
- The ***ProLog by BIM*** inference engine stripped of the parts not necessary to run the application.

The runtime does not include any development facilities (such as monitor, debugger, top level) nor does it depend on the ***ProLog by BIM*** installation.

An **embedded development system** is a development system in which the external program is the master. From the routine *main()*, defined in an external language, one can invoke the ***ProLog by BIM*** engine through the external language interface.

An **embedded run-time** application is a runtime in which the external program is the master instead of ***ProLog by BIM***. From the *main()* routine defined in the external program, one can call Prolog predicates through the external language interface.

3.2 Generic BIMlinker call

A generic form of a BIMlinker call is:

```
% BIMlinker [ <link_option> ] * -  [ <target> ] *
```

The linker options <link_option> must be given as first arguments. They are separated by a single, stand alone ' - ' (blank dash blank) from the other arguments which are options and target files for the engine. The BIMlinker options are fully explained in the next section.

The ***ProLog by BIM*** options <target> are the arguments after the ' - ' sign. They specify the ***ProLog by BIM*** command line options and the Prolog files to be linked. The accepted options are the same as for the BIMprolog command (see "*Principal Components - The Engine*").

The command line options effectively define the new ***ProLog by BIM*** option values for the generated system.

The Prolog files must be compiled before specifying them in a BIMlinker command. BIMlinker does not check if the wic-files are up to date.

The predicates defined in the wic-files will be loaded automatically each time the system is started up.

For a *runtime*, this code is restored from a data file that is generated by BIMlinker. As a result, the wic-files are no longer needed after generation of the run-time system.

In a *development system*, the code is still loaded from the wic-files. This means that these files cannot be removed.

If the wic-files, specified on the command line, contain external load declarations, they are processed by BIMlinker and the external object files mentioned will be linked into the generated executable. These external object files are no longer needed after generation of the run-time or development system. External load declarations specified in a consulted wic-file that is not given on the command line, will be processed at execution time. The external object files are not linked in the generated executable.

3.3 BIMlinker options

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

-cc cmd compile command for the link definition file

Default: SunOS 4: /bin/cc -c %S %I
SunOS 5: /usr/ccs/bin/cc -c %S %I

This option overrides the default value of the compile command (see below for more explanation on the compile command).

-cl cmd final link command

Default: SunOS 4: /bin/cc %S -o %O %I -lm -ldl
SunOS 5: /usr/ccs/bin/cc %S -o %O %I -lm -ldl -lsocket -lnsl

This option overrides the default value of the link command (see below for more explanation on the link command).

-de generation of a verbose run-time system

Default: no system warnings are displayed

When this option is given, a diagnostics verbose run-time system is generated. This is accomplished by producing a run-time system which starts off with the 'syswarn' please switch on. By default, a run-time system is created with its 'syswarn' please switch off, resulting in the suppression of system warnings during start-up.

-dl level

verbosity level of the linker

Default: 1

This option determines the amount of system diagnostics that is displayed by the linker. The level can vary between 0 (quiet) and 2 (very verbose). Value 2 can be useful for debugging the BIMlinker call.

-fl file

file name of link definition file

Default: /tmp/BP.xt.%U

This option overrides the default value of the link definition file name (see next section for more explanation on the link definition file).

-pe error_file

Path description for the error description file

Default: $BIM_PROLOG_DIR/install/errors.o

When creating a new executable, BIMlinker has to be provided with an **error description** object file. The error description object file is generated in two steps from a Prolog source file. First, a C source file is generated with

```
$BIM_PROLOG_DIR/install/BIMerrgen
```

Then the generated file must be compiled with a C-compiler.

The specification of the contents of the Prolog error description file can be found in *"Builtin Predicates - Error Handling and Recovery"*.

Example

If the error description file is named *new_errors.pro*, then

```
csh% $BIM_PROLOG_DIR/install/BIMerrgen new_errors
```

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

```
csh% cc -c new_errors.c
```

creating the object file *new_errors.o* which can be linked by

```
csh% BIMlinker -po newBIMprolog -pe new_errors
```

This will generate a "customized development system" where the default error messages are replaced by the error messages specified in new_errors.pro.

-pm main_file

Make an embedded system

Default: $BIM_PROLOG_DIR/install/main.o

When this option is specified, an embedded system is made and the option specifies the name of the *main file*. A main file is a file that contains the routine *main()* which is called as the top level routine when running the generated executable.

In order to enable ***ProLog by BIM*** to initialize its data structures, the *main()* routine must call the external routine **BIM_Prolog_initialize()**.

BIM_Prolog_initialize()

BIM_Prolog_initialize(argc, argv)
int argc;
char *argv[];

This routine initializes the ***ProLog*** part of the application. In a runtime, this is mainly loading the data from the saved data file. In a customized development system, it takes care of initializing the internal tables. In addition, it may be passed a list of command arguments that will be processed after the initialization phase. Such an argument list must have the same form as a command list passed to main(), but it may be constructed by the external routine. The first argument of the list must be the executable file name of the application. This name will be used for the restart facility (see **restart/0**). As a regular command argument list, it may contain a sequence of system option settings and target file names, possibly followed by a sequence of application options, separated from the system options by a - sign.

If the argument list contains options for the application that must be available during its execution, the argument list must remain living. Otherwise it may be discarded after the ***ProLog*** initialization.

From the *main()* routine defined in the external program, Prolog predicates can be invoked through the use of external language interface routines (see "*External Language Interface - Calling Prolog Predicates From C*"), or the ***ProLog*** top level can be started with the routine BIM_Prolog_main().

BIM_Prolog_main()

BIM_Prolog_main(argc, argv)
int argc;
char *argv[];

This routine starts the ***ProLog*** top level. This routine can only be called in a customized development system. In a run-time system, calling this routine will stop the runtime (no top level is available in a run-time application).

It is not possible to return to the calling routine from this routine. When the built-in halt/0 is called, the system stops completely.

-po exec_file

Path description for the generated executable

Default: -Hbin/BIMprolog

This option allows to change the name of the generated executable to *exec_file*. The name may include the file access path where the file is going to be placed. The executable is always created in the working directory. It is the user's responsibility to move this executable to the correct location.

-pr data_file

Generate a runtime

Default: generation of a development system

A run-time system is generated with data file *data_file*. It is created in the working directory. It is the user's responsibility to move this file to the correct location.

A *runtime* is an optimized executable including the essential Prolog inference engine and is linked with the developer's application files. A *runtime* generally includes:

- The ***ProLog by BIM*** inference engine stripped of the parts not necessary to run the application.
- Compiled Prolog files (*.wic*):
  the developer's Prolog application files and possible libraries.
- Compiled external language object files:
  the developer's application object files and possible libraries.

The application files will be saved in the *data file* file as mentioned in the option. This data file together with the executable constitute the run-time application.

-s opt

Specify symbol table treatment

Default: -s

The treatment of the symbol table of the generated executable, can be influenced by using this option. The value given is passed as option to the linker.

The following values are possible with the default linker command:

Option	Treatment
-s	Strip symbol table
-g	Produce dbx debugging info in the symbol table
-sb	Produce Sun Source Code Browser info (SunOS 4.x)
-xsb	Produce Sun Source Code Browser info (SunOS 5.x)

-vh home_var

Name of the home directory variable

Default: BIM_PROLOG_DIR

The environment variable *home_var* will be used to find the "home directory" of the new system.

The "*home directory*" of the generated executable is used to find the system files, such as the LICENSE file, the compiler BIMpcomp or the data file. The "home directory" is referred to with the prefix **-H** (see "*The internal working of BIMlinker - File names and expansion*").

Example

```
csh% BIMlinker -po newBIMprolog -vh NEW_BIM_PROLOG_DIR
```

Will generate a "*customized development system*" of which the "*home directory*" is defined with the environment variable NEW_BIM_PROLOG_DIR.

-vl libs_var

Name of the library directory path variable

Default: BIM_PROLOG_LIB

The environment variable *libs_var* will be used to find the "*library directory paths*".

The *library directory path* of the generated executable is used when expanding **-L** in file names (see *"The internal working of BIMlinker - File names and expansion"*).

Example

```
csh% BIMlinker -po newBIMprolog -vl NEW_BIM_PROLOG_LIB
```

Will generate a *customized development system* of which the *library directory path* is defined with the environment variable NEW_BIM_PROLOG_LIB.

3.4 The internal working of BIMlinker

BIMlinker will generate, with the given input, an executable and possibly a data file. The generated executable will automatically either load the data file (in case of a runtime) or the Prolog files which were specified in the BIMlinker command (when it is a customized development system).

To generate the executable, BIMlinker first produces a **link definition file**, which is a C source file, that has to be compiled. After compilation, the resulting object file is linked with the ***ProLog*** engine. The commands (compile command and link command) that are used and the name of the file (the link definition file) that is generated can be adapted according to the developer's environment.

Compile Command specification

The command that will be used to compile the link definition file, is determined in the following way. If the corresponding BIMlinker option is mentioned (-cc), the indicated value is used. Otherwise, if the corresponding environment variable is defined (BIM_XTC_CMD), its value is taken as command. If none of these are defined, a default value is used (default value, see explanation of BIMlinker option -cc).

A command definition is a text string which can contain references to parameters. These take the form of a % followed by a capital letter indicating the parameter that has to be substituted. To include a literal % sign, it must be doubled as %%. The following parameters are recognized:

Indicator	Meaning
%I	Input file(s)
%O	Output file
%S	Symbol table treatment option

The input file is the link definition source file, with a name as specified, and with extension '.c'.

The output file name is deduced from the input file name by stripping any leading paths and replacing the extension with '.o'. It should normally not be used in the compile command. Neither should another output file be specified in that command, since this would make it invisible for BIMlinker.

Link Command specification

The command that will be used to do the final linking is determined in the following way. If the corresponding BIMlinker option is mentioned (-cl), the indicated value is used. Otherwise, if the corresponding environment variable is defined (BIM_XTL_CMD), its value is taken as link command. If none of these are defined, a default value is used (default value, see explanation of BIMlinker option -cl).

A command definition is a text string which can contain references to parameters. These take the form of a % followed by a capital letter indicating the parameter that has to be substituted. To include a literal % sign, it must be doubled as %%. The following parameters are recognized:

Indicator	Meaning
%I	Input file(s)
%O	Output file
%S	Symbol table treatment option

The input file parameter is substituted by a sequence of files. First come the engine object files and the output file of the compile command. The rest of the input files are retrieved from the object files and libraries as mentioned in the extern_load directives.

The output file is the destination executable, as specified.

Link definition file name

The name of the link definition file is determined in the following way. If the BIMlinker option -fl is set, the indicated value is used. If the option is not mentioned and the environment variable BIM_XTL_FILE is defined, its value is taken. Otherwise, a default value is used.

A file name specification indicates the base name of the file. A '.c' suffix is appended for the source file and a '.o' suffix for the object file name. It can include a path, and possibly a parameter, indicated as:

Indicator	Meaning
%U	Unique file name substitute

The %U parameter is substituted by a character sequence that makes the file name become unique.

File names and expansion

The specification of the file names and their location in the BIMlinker command determine the way the files are searched for, when invoking the new executable.

File location

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

absolute location
Path starting with "/": this absolute path is searched for.
Path not starting with ".": this path, extended to an absolute path, is searched for.

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

environment dependent location
Path starting with "**-H**": relative to the application home directory.
Path starting with "**-L**": relative to one of the library directories.

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

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

The most flexible solution is the **environment dependent location**. The whole system can be moved to any directory. The location of the system files is determined by the *home directory* and *library directory path* environment variables.

Expansion

Environment dependent file names are expanded as follows:

-H*path* Expanded to $*APPL_HOME*/*path*,
with *APPL_HOME* the variable that indicates the application home directory, as set with the BIMlinker option **-vh**. If this variable is not explicitly set in the BIMlinker call it is BIM_PROLOG_DIR.

-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 environment variable that indicates the application library directory path, as set with the BIMlinker option **-vl**. If this variable is not explicitly set in the BIMlinker call it is BIM_PROLOG_LIB.
- Application home library $*APPL_HOME*/lib, with *APPL_HOME* the environment variable that indicates the application home directory, as set with the BIMlinker option **-vh**. If this variable is not explicitly set in the BIMlinker call it is BIM_PROLOG_DIR.
- ***ProLog by BIM*** home library $BIM_PROLOG_DIR/lib.
- Working directory.

Note:

A distinction must be made between the search for files specified in the BIMlinker or BIMpcomp commands and the search for files when running the executable generated by BIMlinker.

Search by BIMlinker and BIMpcomp:

- home directory: **$BIM_PROLOG_DIR**
- library directory path: **$BIM_PROLOG_LIB**

Search by an executable generated by BIMlinker:

- home directory: as set with the **-vh** option of BIMlinker at creation time of the executable.
- library directory path: as set with the **-vl** option of BIMlinker at creation time of the executable.

3.5 Customized development system for debugging

For **debugging** the external parts of the generated executable, the following conditions must be met.

- The external object files have to be compiled with the -g option.
- All debug information has to be retained in the new executable. This can be specified in the BIMlinker command with the -s-g option.

Example

Given the C file:

```
routine(in,out)
int in, *out;
{
      *out = in + 2;
      return(1);
}
```

And the Prolog file:

```
:-extern_load([routine],['debug.o']).
:-extern_predicate(routine(integer:i,integer:m)).
```

Creation of the debuggable executable is done with:

```
csh% BIMpcomp debug
csh% cc -g -c debug.c
csh% BIMlinker -s-g -po BIMprolog_debug - debug
Linking target file debug.wic
Final linking into BIMprolog_debug
```

The debugging session:

```
csh% dbx BIMprolog_debug
reading symbolic information ...
(dbx) stop in routine
[1] stop in routine
(dbx) run
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993

?- routine(3,_out).
[1] stopped in routine at line 4
     4     *out= in + 2;
(dbx) step
stopped in routine at line 5
     5     return(1);
(dbx) print *out
5
(dbx) assign *out = 8
(dbx) cont
  _out = 8
Yes
?- halt.
program exited
(dbx) quit
csh%
```

3.6 Creating customized development systems

A typical BIMlinker call to generate a *customized development system* is:

```
csh% BIMlinker -po exec_file - options wic_files
```

The new system is named *exec_file*, as indicated with BIMlinker option **-po**. It will have default options as given in *options* and when it is started up, the Prolog files *wic_files* are silently consulted.

It is not advisable to change the home directory name of a customized development system since such a system needs

The following two examples show BIMlinker calls from installation scripts (respectively BIMinstall and BIMinstallOM) of $BIM_PROLOG_DIR/install.

BIMprolog

The BIMprolog executable of the default ***ProLog by BIM*** installation can be generated with:

```
csh% BIMlinker -o -Hbin/BIMprolog - \
     'cat $BIM_PROLOG_DIR/install/BIMprolog.options'
```

The file BIMprolog.options contains the engine command line options of your installation. If the contents of this file is included in the BIMlinker command, then the generated development system will have the same defaults as BIMprolog.

BIMprologOM

The following call to BIMlinker generates the development system BIMprologOM with:

- The interfaces to the following windowing packages: OSF/Motif, Xt, Xlib, running under AIXwindows*.
- Turned into DEC-10 compatibility mode. (compatibility [on], in-line evaluation [off], querymode [on], showsolution [on], fail on end_of_file [off])

```
csh% BIMlinker -po -Hbin/BIMprologOM - \
     -Lwindowing/motif -Lwindowing/xt -Lwindowing/xlib \
     -Pc+ -Pe- -Pq+ -Ps+ -Pfr-
```

3.7 Creating run-time applications

BIMlinker allows the developer to generate stand-alone run-time versions of applications developed with ***ProLog by BIM***. The generated system consists of two files: an executable file and a data file.

Before generating a run-time system with BIMlinker, all external files and Prolog files must be compiled. It is recommended to use a *makefile* to both manage the compiled files and generate the run-time system.

BIMlinker call for run-time applications

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

```
csh% BIMlinker -po ApplRun -pr ApplData \
          -vh ApplHome -vl ApplLibs -  \
          options targets
```

The executable is named *ApplRun* and the data file *ApplData*. The environment variable that indicates the application home directory is named *ApplHome*, and the one for the library directories *ApplLibs*. The *options* are used as default ***ProLog by BIM*** 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.

Conventions for creating run-time applications

Some conditions must be met in order to turn an application into a run-time system. They apply to the top-level and to the program file organization.

Top-level condition

The **main/0** predicate.

A run-time system has no Prolog engine top level. The processing of such a system consists of the processing of a special predicate: **main/0**. As soon as **main/0** ends, tl run-time application 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. Starting a runtime that does not contain a definition for the predicate **main/0** will result in an error message.

Program file organization

There are 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 application is created.

execution time
Indicates the moment when the run-time application is processed.

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

1 All Prolog files that are given as targets to the ***ProLog by BIM*** linker are consulted at creation time.

2 The Prolog files that are consulted at creation time are saved into the run-tin. system. They are no longer needed at execution time.

3 The object files and libraries that are linked at creation time (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 no longer needed at execution time.

4 The Prolog files that are consulted at execution time, have to be accessible in a compiled form (*.wic*) at execution time.

5 The object files and libraries that are incrementally linked at execution time (if they appear in extern_load calls performed at execution time or as extern_load directives of files that are not given as targets to BIMlinker) have to be accessible in a compiled form (*.o*) at execution time.

6 Any object file that has to be part of the run-time system, must be declared with an external load directive in a Prolog file that is given as target to **BIMlinker**.

7 Queries in files that are consulted at creation time are executed at creation time. If they contain consults, these are also performed at creation time.

8 Queries in files that are consulted at execution time are ignored at creation time. They are only performed at execution time.

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 to put the whole system. This is accomplished through the use of an environment variable that contains the path of the application 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** abbreviation in the file name. An analogy can be made with the ***ProLog by BIM*** development system, where the home directory variable is by default named BIM_PROLOG_DIR.

Examples of run-time applications

The three following examples show the creation of run-time applications.

(minimal) Example 1

This example is in $BIM_PROLOG_DIR/demos/linker/RunTime1

The source program consists of the file RunTime1.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 RunTime1
% BIMlinker -vh RUNTIME1_HOME -pr -HRunTime1Data \
     -po -HRunTime1Run - -TCb0 RunTime1
```

The output of this BIMlinker call is:

```
Warning : Environment variable RUNTIME1_HOME is undefined.
Using system's home directory instead.
Linking target file RunTime1.wic
Final linking into RunTime1Run
Initializing data for RunTime1Run
Warning : Cannot expand -HRunTime1Data : unknown system home
directory.
Warning : Cannot expand -HRunTime1Run : unknown system home
directory.
```

Since no application home directory variable RUNTIME1_HOME was defined, the linker issues a warning. For this application, this variable is not used and therefore the warnings can be ignored.

Then the application can be moved to its destination home directory. Suppose it is placed in ~/RunTime1:

```
% mv RunTime1Run RunTime1Data ~/RunTime1
```

The application can be started:

```
% setenv RUNTIME1_HOME ~/RunTime1
% $RUNTIME1_HOME/RunTime1Run
```

This will produce the message:

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

Example 2

This example is in $BIM_PROLOG_DIR/demos/linker/RunTime2

The program consists of two source files: RunTime2.pro and RunTime2.c. The Prolog file contains the following declarations and definitions:

```
:- extern_load( [ date_time ] , [ 'RunTime2.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 contains 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 RunTime2.c
% BIMpcomp RunTime2
% BIMlinker -vh RUNTIME2_HOME -pr -HRunTime2Data \
     -po -HRunTime2Run - RunTime2
```

The output of this BIMlinker call is:

```
Warning : Environment variable RUNTIME2_HOME is undefined.
Using system's home directory instead.
Linking target file RunTime2.wic
Final linking into RunTime2Run
Initializing data for RunTime2Run
Warning : Cannot expand -HRunTime2Data : unknown system home
directory.
Warning : Cannot expand -HRunTime2Run : unknown system home
directory.
```

Since no application home directory variable RUNTIME2_HOME was defined, BIMlinker issues a warning. For this application, this variable is not used and therefore the warnings can be ignored.

The application can now be moved to its destination home directory. Assume it will be placed in ~/RunTime2.

```
% mv RunTime2Run RunTime2Data ~/RunTime2
```

The application can be started:

```
% $RUNTIME2_HOME/RunTime2Run
```

This will output something like:

```
Hello from the Run-Time System.
On this system, it is now Wed Oct 27 15:46:01 1993
Bye.
```

Example using XView

This example is in SBIM_PROLOG_DIR/demos/linker/RunTime3.

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 writes 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 any more 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_loa( [ 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 -vh RUNTIME3_HOME -pr -HRunTime3Data \
     -po -HRunTime3Run - -TDb8k -TFb4k main \
     -Lwindowing/xview -Lwindowing/xlib fixintro
```

The output of this BIMlinker call is similar to:

```
Warning : Environment variable RUNTIME3_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 RunTime3Run
Initializing data for RunTime3Run
Warning : Cannot expand -HRunTime3Run : unknown system home
directory.
Warning : Cannot expand -HRunTime3Data : unknown system home
directory.
```

The application can now be moved to its destination home directory (for example: ~/RunTime3):

```
% mv RunTime3Run RunTime3Data ~/RunTime3
% mv varintro.wic ~/RunTime3/lib
```

The application is started with:

```
% setenv RUNTIME3_HOME ~/RunTime3
% $RUNTIME3_HOME/RunTime3Run
```

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

```
Fixed intro to hanoi.
Variable intro to hanoi.
```

3.8 Creating embedded run-time applications

An *embedded run-time application* can be created with the following BIMlinker call.

```
% BIMlinker -po embed -pm main_file -pr data_file
                -vh ApplHome -vl ApplLibs - \
                options targets
```

The executable is named *embed* and the data file *data_file*. *main_file* contains the routi. main(). The environment variable that will indicate the application home directory is named *ApplHome*, and the one for the library directories *ApplLibs*. The *options* are used as default ***ProLog by BIM*** options in the embedded system and the *targets* are linked into the system. External code in these targets is linked into the executable file.

Conventions

Some conditions must be met in order to turn an application into an embedded run-time system.

- The call to **BIM_Prolog_initialize()** to ensure the ***ProLog by BIM*** initialization must be performed before calling any of the other ***ProLog by BIM*** external routines.
- The *embedded run-time application* consists of the same two files (executable and data file) as a run-time application. Since the differences between creation and execution time are also valid for the embedded runtimes, the application may also need some *.wic* files at execution time.

Example of embedded run-time applications

This example is in $BIM_PROLOG_DIR/demos/linker/EmbedRT.

As an example of an embedded run-time application, 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. The program 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 EmbedRT1.c
% BIMpcomp EmbedRT1
% BIMlinker -vh EMBEDRT1_HOME -pr -HEmbedRT1Data \
          -po -HEmbedRT1Run -pm EmbedRT1.o - EmbedRT1
```

The output of this BIMlinker call is:

```
Warning : Environment variable EMBEDRT1_HOME is undefined.
Using system home directory instead.
Linking target file EmbedRT1.wic
Final linking into EmbedRT1Run
Initializing data for EmbedRT1Run
Warning : Cannot expand -HEmbedRT1Data : unknown system home
directory.
Warning : Cannot expand -HEmbedRT1Run : unknown system home
directory.
```

Since no application home directory variable EMBEDRT1_HOME was defined, the linker issues a warning. For this application, this variable is not used and therefore the warnings can be ignored.

The application can now be moved to its destination home directory. It will be placed in ~/EmbedRT1:

```
% mv EmbedRT1Run EmbedRT1Data ~/EmbedRT1
```

The application can be started:

```
% $EMBEDRT1_HOME/EmbedRT1Run
```

The output of the program is:

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

Syntax

Syntax

Chapter 1
Syntax

1.1 Type ranges

Integers

The range of integers is from -268435456 up to 268435455 (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 greater than 10, digits greater than 9 are represented by characters a-z or A-Z.

Example

16'3a7 stands for the hexadecimal number 3a7.

Reals

Reals are manipulated in double precision: approximately 16 significant decimal digits.

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".

Example

0x1700da

Atoms

The length of atoms is restricted to 32767 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 64.

1.2 ProLog by BIM syntax rules

Native syntax

This section contains the complete ***ProLog by BIM*** native syntax, in BNF-format.

Notes on the used format:

- 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, other separators such as <LF>, tabs or special characters.
- Non-terminals are enclosed between <>.
- Terminals are in bold.

ProLog by BIM 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
M and N denote the precedence of operators allowed.

<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)>
where N ≠ 0

<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>
Range: the range of integers is 29 bits

<number> ⇒ <digit> |
<digit> <number>

<based number> ⇒ <base>'<alfanum><alfanum>* |
0'(<char>|')

<base> ⇒ integer between 1 and 36

<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, the \ has the same meaning as in a C string.

<alfanum+> ⇒ <letter> |
<digit> |
<underscore>

<alfanum> ⇒ <letter> |
<digit>

<restname> ⇒ <empty> |
<alfanum+><restname>

<char> ⇒ any printable character different from '

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 {or a} > }

A comment can be placed anywhere, and serves as a delimiter.

Differences with DEC-10 syntax

The ***ProLog by BIM*** native syntax differs from the DEC-10 Prolog syntax in th. following ways:

- All variables must start with an underscore. Names starting with an upper case are not interpreted as variables.
  Example _var is a correct variable name
  Term is not a variable name.
- A quoted string is never considered as an operator.
  Example If + is an infix operator, 1 + 2 is a valid term and
  1 '+' 2 is not a valid term.
- All operators should have the correct number of operands
  Example write(+) is erroneous.
- A postfix operator cannot be an operator of any other type.
- A dot "." followed by a space or an end-of line-character is not necessarily an endpoint.
  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**.
  Example
  0x0 stands for the null pointer
  0xabc for a pointer with integer value = 2748
- Certain constructions which do need quoting in DEC-10 syntax do not need quoting in native syntax.
  Example _r =? abc '||' de . in DEC-10 syntax
  _r =? abc || de . in native syntax
- Comments are specified between curly braces ({}), instead of a line beginning with % or text between /* and */
- The curly braces ({}) in DCG rules must be represented in the form '{}'(Term) .

DEC-10 syntax in ProLog by BIM

Programs written in the DEC-10 Prolog syntax can be interpreted and translated by the ***ProLog by BIM*** compiler. To be able to compile Prolog code correctly, the compiler must know in which syntax the code is written. This can be specified with the syntax switch (-c , -Cc) of the compiler and/or the engine (see "*Principal Components*").

The compiler accepts some extensions to the DEC-10 syntax.

- Explicit module qualification is possible (see *"Modules"*).
- It is possible to create pointers with the built-in predicate **pointertoint/2.**

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.

1.3 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 than or equal to that of the operator.

It is not advisable to change the predefined operators which are listed below, especially "," and "|".

Precedence	Type	Name	Usage
1200	xfx	:-	clauses ("if")
1200	xfx	-->	definite clause grammars
1200	fx	:-	directive
1200	fx	?-	queries
1150	fx	dynamic	directive
1150	fx	mode	directive
1100	xfy	;	clauses ("or")
1100	xfy	some	clauses
1050	xfy	->	clauses ("if then else")
1050	xfy	none	clauses
1000	xfy	,	clauses ("and")
1000	xfy	|	list
900	fy	\+	built-in ("not")
900	fy	not	built-in
800	fx	import	directive
800	fx	local	directive
800	fx	global	directive
750	xfx	from	directive

700	xfx	=	built-in
700	xfx	is	built-in
700	xfx	=?	built-in
700	xfx	=..	built-in
700	xfx	==	built-in
700	xfx	\==	built-in
700	xfx	=:=	built-in
700	xfx	=\=	built-in
700	xfx	?=	built-in
700	xfx	<	built-in
700	xfx	>	built-in
700	xfx	=<	built-in
700	xfx	>=	built-in
700	xfx	<>	built-in
700	xfx	@<	built-in
700	xfx	@>	built-in
700	xfx	@=<	built-in
700	xfx	@>=	built-in
700	xfx	\=	built-in
600	yfx	||	expressions
500	xfx	index	directive
500	yfx	+	expressions
500	yfx	-	expressions
500	yfx	/\	expressions
500	yfx	\/	expressions
500	fx	\	expressions
500	fx	+	expressions
500	fx	-	expressions
400	yfx	*	expressions
400	yfx	/	expressions
400	yfx	//	expressions
400	yfx	<<	expressions
400	yfx	>>	expressions
300	xfx	**	expressions
300	xfy	^	expressions
300	xfx	mod	expressions
300	xfx	rem	expressions
100	xfy	:	directive

For operator declaration, see "*Compiler Directives - Operators*" and "*Built-in Predicates - In-core Database*".

1.4 Directives

The compiler directives are explained in a separate part "*Compiler Directives*". A list of all compiler directives can also be found on the Quick Reference Card.

1.5 Built-in predicates

The Quick Reference Card contains a list of all the built-in predicates. These predicates are explained in the parts "*Built-in Predicates*", "*Modules*", "*Debugger*" and "*External Language Interface*".

1.6 DCG

DCG (**Definite Clause Grammar**) is a notational extension of Prolog that makes it easy to implement formal grammars.

Grammar rules are translated into Prolog clauses when they are compiled or consulted under the DEC-10 Prolog syntax (-c+ option for BIMprolog and BIMpcomp). They can also be used in native syntax (but in a slightly different syntax, see below).

The DCG formalism makes it possible to write clauses in a more compact way. DCG rules are most often used for writing parsers: when the grammar is given in BNF notation, it is very simple to turn it into a parser in Prolog using DCGs.

Example

Consider the grammar:

<expr>	==>	<number> |
		<number> '+' <expr> |
		<number> '-' <expr>
<number>	==>	<int> | <int> <number>
<int>	==>	'0' | '1' | '2' | '3' | '4' |
		'5' | '6' | '7' | '8' | '9'

in DCG form:

```
expr -->  number ;
          number , ['+'] , expr ;
          number , ['-'] , expr .
number --> int;
          int, number.
int --> ['0'], ! .
int --> ['1'], ! .
int --> ['2'], ! .
int --> ['3'], ! .
int --> ['4'], ! .
int --> ['5'], ! .
int --> ['6'], ! .
int --> ['7'], ! .
int --> ['8'], ! .
int --> ['9'], ! .
```

Now a query of the form

```
?- expr(['1','+','3','-','5'],[]).
```

will succeed.

The main pitfall in using DCGs is left recursion in the grammar rules. Also, the complexity of the parsing can be very bad because of the depth first strategy of Prolog.

The DCG formalism makes it possible to write clauses in a more compact way. The DCG translator will transform clauses with -->/2 as principal functor into normal Prolog clauses which are then added to the in-core database. In general, the transformation consists of adding two extra arguments to the head and all the goals of the clause, in a systematic way. These arguments serve as input and output argument respectively. If the head has a specific form, and for some goals, the transformation is different. Examples will make this clear.

It must be stressed that the transformation is performed on an attempt to add clauses to the database, either during compilation of a file or during assert.

Rules

Example showing the different DCG translation rules.

DCG rule	corresponding Prolog rule
`a --> b.`	`a(S0,S1) :-` `  b(S0,S1).`

`a --> b , c .`	`a(S0,S1) :-`
	`  b(S0,S2),`
	`  c(S2,S1).`

`(a,[b,c]) --> e,f.`	`a(S0,S1) :-`
	`  'C'(S1,b,S2),`
	`  'C'(S2,c,S3),`
	`  e(S0,S4),`
	`  f(S4,S3).`

`a --> [x,y].`	`a(S0,S1) :-`
	`  'C'(S0,x,S2),`
	`  'C'(S2,y,S1).`

`a --> b,{c},d.`	`a(S0,S1) :-`
	`  b(S0,S2),`
	`  c,`
	`  d(S2,S1)`

`a --> b, !, c.`	`a(S0,S1) :-`
	`  b(S0,S2),`
	`  !,`
	`  c(S2,S1)`

`a(f) --> b(h,i).`	`a(f,S0,S1) :- b(h,i,S0,S1) .`

These are the rules shown in the above examples.

- The goal ! is not transformed.
- A goal between {} has its brackets removed.
- A list L 'consumes' the given elements from one of the added arguments.
- The last example serves to point out that the extra arguments are added at the end of the given arguments.

The definition of 'C'/3 is:

```
'C'([X|Y],X,Y) .
```

'C'/3 is not redefinable. The compiler replaces occurences of 'C'/3 by explicit unification.

```
'C'(X,Y,Z) is replaced by X = [Y|Z]
```

This explicit unification can sometimes be absorbed in the head unification, hence the following alternatives for some of the examples:

`(a,[b,c]) --> e ,f.`	`a(S0,[b,c|S1]) :-`
	`  e(S0,S2),`
	`  f(S2,S1).`

`a --> [x,y].`	`a(S0,S1) :- S0 = [x,y|S1] .`

The semantics of both translations remains the same but the resulting Prolog rules will be translated more efficiently taking full advantage of the ***ProLog by BIM*** compilation techniques like indexing, ...

In native syntax mode, the curly brackets must be written as '{}'(goal) instead of {goal}.

Portability

There are some controversies related to DCGs: the DCG formalism is not part of the forthcoming ISO Prolog standard.

- !/0
  Some systems translate
  ```
  a --> !.
  ```
  to
  ```
  a(S0,S1) :- S0 = S1, !.
  ```
  others translate it to
  ```
  a(S0,S1) :- !, S0 = S1.
  ```
  ProLog by BIM chooses the first and translates it to
  ```
  a(S,S) :- !.
  ```
- Lists that are not ending on []
  Some systems translate
  ```
  a --> [b|f(9)].
  ```
  to
  ```
  a(S0,S2) :- 'C'(S0, b, S1), f(9, S1, S2).
  ```
  Others will fail on translating such a clause.
  ProLog by BIM raises an error for an illegal DCG rule.
- Numbers as goals in DCGs.
  ```
  1 --> 2 .
  ```
  This is by most systems added to the database as is.
  ProLog by BIM fails to DCG translate such a DCG rule and raises an error.
- Numbers between {}
  Some systems translate
  ```
  a --> {3}.
  ```
  to
  ```
  a(S0,S1) :- call(3), S1=S0.
  ```
  which when executed raises an error.
  For ***ProLog by BIM***, the DCG translation succeeds, but the resulting Prolog clauses cannot be translated and the system raises an error.
- The push down list.
  Some systems translate
  ```
  (a,[b]) --> c .
  ```
  to:
  ```
  a(S0,S1) :- 'C'(S1,b,S2), c(S0,S2).
  ```
  Others translate it to:
  ```
  a(S0,S1) :- c(S0,S2) , 'C'(S1,b,S2) .
  ```
  ProLog by BIM translates it into the former and optimizes it further into:
  ```
  a ( S0,[b|S1]):- c(S0,S1) .
  ```

The above mentioned controversies can most often be avoided, so that your DCG code will be portable.

Example of DCGs

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

```
:- option('-c+').
?- 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
```

Built-in Predicates

Built-in Predicates

Chapter 1
Introduction

1.1 Overview

This part of the manual gives an overview of most of the built-in predicates of ***ProLog by BIM***. Some of the built-in predicates are explained in more detail in a separate part of this manual. This is the case for the External Language Interface predicates (see "*External Language Interface*"), the debugging predicates (see "*Debugger*") and the module predicates (see "*Modules*").

In addition to the built-in predicates ***ProLog by BIM***, also contains an extensive set of library predicates. The library is divided into categories which are each described separately in the manual ("*Windowing and Graphics Libraries*", " *Database Interfaces*", "*Prolog and Unix Libraries*", "*Programming Tools and Scripts*")

1.2 Notation

The ***ProLog by BIM*** built-in predicates are described as follows:

functor/arity

predname(arg1, ...argn)

arg1: ...
...
argn: ...

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

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

A distinction is made between:

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

Possible types can be:

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

Built-in Predicates

Chapter 2
Engine Manipulation

This chapter explains the predicates available to the user to interact with the engine. Possible interactions are:

- Change the behavior of the ***ProLog by BIM*** engine during execution.
- Manage the internal tables.
- Stop, save or restart a ***ProLog by BIM*** session.
- Retrieve information regarding ***ProLog by BIM***.

2.1 Switches

A full description of the different engine switches and the possible manipulations is given in *"Principal Components - The Engine - Please options"*. This section explains in more detail the built-in predicate **please/2**.

please/2

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

arg1 : free or 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 value is changed to *arg2*. When both arguments are free, a list of all switches and their current values is given.

The following list gives an overview of the available switches. For each option its abbreviated name, its full name and a short explanation are given. More information can be found in "*Principal Components - The Engine - Please options*".

Changing the value of an option which controls the parser (atomescape, compatibility, in-line evaluation) only takes effect after the parsing of the current query.

ae : atomescape	Controls the escape character.
c : compatibility	Controls the syntax of the parser.
d : debugcode	Controls the generation of debug code.
e : eval	Controls in-line evaluation.
ed : envdebug	Controls the environment debugger.
em : envmonitor	Controls the environment monitor.
fr : formatreal	Controls the format of printed reals.
h : hide	Controls hiding of added definitions.
la : reloadall	Controls behavior of the reload and reconsult predicates.
q : querymode	Controls the top-level behavior.
r : recovery	Controls error recovery .
ra : readeofatom	Controls the end-of-file atom.
rc : readeofchar	Controls the end-of-file ascii-code.
rf : readeoffail	Controls the behavior of read at eof.
s : showsolution	Controls the printing of the solutions.
sw : syswarn	Controls system diagnostic messages.
tt : tabletime	Returns the time of table operations.
tw : tablewarn	Control the table messages.
w : warn	Controls the general error messages.
wd : writedepth	Controls the printing of nested terms.
wf : writeflush	Controls flushing after a write.
wm : writemodule	Contols explicit module qualification.
wp : writeprefix	Controls the output of operators.

wq : writequotes Controls the usage of quotes.

Example

```
?- please(ae,on).
Yes
?- write('at\
   om\n').
atom
Yes
?- please(ae,off).
Yes
?- write('at\
      ?- write('at\
               ^
*** SYNTAX 200 *** End of line character inside quoted string.
?- please(hide,on),
   assert(newpredicate(_1,_2)),
   listing(newpredicate/2).
Yes
?- please(hide,off),
   assert(anotherdefinition(par)),
   listing(anotherdefinition/1).

anotherdefinition(par) .
Yes
?- please(readeoffail,off),
   fopen(_File,zerolength,r),
   readln(_File,_Line).
  _Line = end_of_file
  _File = 0xd79e8
Yes
?- please(readeoffail,off),please(rc,26),
   fopen(_File,zerolength,r),
   get(_File,_Char).
  _Char = 26
  _File = 0xd7a10
Yes
?- please(readeoffail,on),
   fopen(_File,zerolength,r),
   readln(_File,_Line).
No
?- please(sw,on),reconsult('/tmp/p').
compiled /tmp/p.pro
reconsulted p.pro
Yes
?- please(sw,off),reconsult('/tmp/p').
Yes
?- please(tt,_t).
  _t = 32
Yes
?- please(tw,on),table('H',collect).

*** Table Garbage Collection ***
H Heap : 32768u alloc'd (163840b)   30u used (0%)   32738u
avail
H Heap : 32768u alloc'd (163840b)   20u used (0%)   32748u
avail
*** Table Garbage Collection *** Done in 0.016 s
Yes
?- please(tw,off),table('H',collect).
Yes
```

```
?- consult(nonexistent).
*** RUNTIME 600 *** File nonexistent.pro non-existent or not
a regular file.
Yes
?- please(w,off),consult(nonexistent).
Yes
?-
```

2.2 Table manipulation

This section describes the predicates that can be used for managing the internal tables. The list below shows the internal tables, with their identifier, that are visible to the user.

H	Heap
S	Stack
T	Text
D	Data
F	Functors
I	Interpr Code
C	Compiled Code
R	Record Keys
B	Backup Heap

A full description of the different tables and the possible manipulations is given in *"Principal Components - The Engine - Manipulating the ProLog by BIM tables"*.

table/2

The predicate table/2 can be used in two forms.

table(_TableId, _TableCommand)

arg1 : ground : atom
arg2 : ground : atom

Arg1 is the table identifier and *arg2* is the command. The indicated command is performed on table *arg1* if possible.

command	action
collect	invokes garbage collection, and invokes 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 format, the values of the table parameters can be retrieved and set at runtime.

Arg1 specifies the table: both the full name and the table identifier can be used.

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*, if this value is redefinable. 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	shrink
4	limit

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

Table statistics

The statistics/4,3,1,0 predicates can be used to retrieve information about the internal tables.

statistics/4

statistics(_TableId,_AllocedUnits,_UsedUnits,_AllocedBytes)

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

Gives statistics on the table *arg1*. Arguments *arg2* through *arg4* are instantiated to the allocated and used number of units, and to the allocated size in bytes respectively.

Example

```
?- statistics('H',_1,_2,_3).
  _3 = 163840
  _2 = 38
  _1 = 32768
```

statistics/3

statistics(all, _, _)
statistics(_TableId, _AllocedUnits, _UsedUnits)

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

Arg2 and *arg3* are instantiated to the allocated and used size of the table indicated by *arg1*.

If *arg1* is instantiated to the atom **all**, the statistics of all the tables are printed to the current output stream. *Arg2* and *arg3* remain uninstantiated.

Example

```
?- statistics('H',_1,_2).
  _2 = 33
  _1 = 32768
```

statistics/1

statistics(_TableId)

arg1 : ground : atom

The allocated, used and available size of table *arg1* is printed to the current output stream.

Example

```
?- statistics( H )
```

Could give as possible output :

```
H Heap  :32768u alloc'd (163840b) 23u used (0%) 32745u avail
```

The numbers which are printed out are respectively: the allocated size in units, the allocated size in bytes, the used size in units and as last one the available size in units.

statistics/0

statistics

This predicate is defined as:

statistics :- statistics(all, _, _).

Example

```
?- statistics.
H Heap       :  32768u alloc'd ( 163840b)     23u used ( 0%)
32745u avail
S Stack      :  32768u alloc'd ( 163840b)     92u used ( 0%)
32676u avail
T Text       :  40944u alloc'd (  49136b)  26768u used (65%)
14176u avail
D Data       :   6144u alloc'd (  77848b)   1924u used (31%)
4220u avail
F Functors   :   1024u alloc'd (  24576b)    972u used (94%)
52u avail
I Interp Code :  20472u alloc'd (  90080b)    130u used ( 0%)
20342u avail
C Compil Code : 388968u alloc'd (1564064b) 380836u used (97%)
8132u avail
R Record Keys :   2048u alloc'd (  32768b)      0u used ( 0%)
2048u avail
B Backup Heap :   8192u alloc'd (   8192b)      0u used ( 0%)
8192u avail
Yes
```

2.3 System control

Exit from ProLog by BIM

halt/1

halt(_ExitStatus)

arg1 : ground : integer

The engine process is terminated with exit status code *arg1* (see also "*Principal Components - The Engine*").

halt/0

halt

The engine process is terminated with default success exit status, which is 0 in Unix systems.

Saved state

ProLog by BIM offers the opportunity to save a session in a file. The complete state except for the list of open files, but including the external code is saved to a file on disk. A saved state can be restored by invoking the engine with the special command line option -r followed by the file name *arg1*, or with the built-in predicate **restart/1.**

save/1

save(_PhysFileName)

arg1 : ground : atom

Saves the current state of the session in the file *arg1*.

save/2

save(_PhysFileName, _Goal)

arg1 : ground : atom
arg2 : ground : term

The current state of the session is saved in the file *arg1*. In addition, the goal *arg2* is saved in the same file. This goal will automatically be executed when the saved state is restored. Before executing that goal, all command line arguments given with the restoring invocation, are first handled.

Example

```
?- save(savefile,
   write('saved state restored, goal executed')).
Yes

?-restart(savefile).
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993
restoring from state saved by pbb at Sun Dec 31 23:59:59 1999

saved state restored, goal executed
Yes
```

Restarting the engine

ProLog by BIM offers the possibility to restart the current session.

restart/0

restart

The session is terminated and restarted in exactly the same way as it was first started.

Example

```
% BIMprolog -Pq+ -Ps+ example
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993
compiled example.pro
consulted example.pro

?- restart .
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993
compiled example.pro
consulted example.pro

?-
```

restart/1

restart(_ArgumentList)
restart(_PhysFileName)

arg1 : ground : list of atom

The first call terminates the current session and restarts with the arguments specified in list *arg1*.

Example

```
% BIMprolog -Pq+ -Ps+
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993
compiled example.pro
consulted example.pro

?- restart([example2]).
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993
compiled example2.pro
consulted example2.pro

?-
```

The second alternative terminates the session and restarts by restoring the state saved in file *arg1*. This is equivalent to:

?-restart(['-r',_PhysFileName]).

Example

```
?- save(savefile).
Yes
?- restart(savefile).
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993
restoring from state saved by prolog at Sun Dec 31 23:59:59
1991
```

Information predicates

The following predicates exist to retrieve some general information about the ***ProLog by BIM*** version.

Information is associated to a number of keys.

Possible keys and associated information are:

Arg1	Arg2
serial	[your client number]
product	[ProLog by BIM]
version	[machine, os, release number, release date]
copyright	[copyright notice]
owner	[coordinates BIM]
distributor	[coordinates of your local distributor]

Example

```
?- information.
Serial Number : 0-C-1
Product :       ProLog by BIM
Version :       Sun4 - SunOS - 4.1  - 01-1993
Copyright :     (c) Copyright BIM - 1993
Owner :
   BIM
   Kwikstraat 4
   B-3078 Everberg
   Belgium
   tel +32 2 759 59 25
   fax +32 2 759 47 95
   prolog@sunbim.be
Distributor :
   BIM
   Kwikstraat 4
   B-3078 Everberg
   Belgium
   tel +32 2 759 59 25
   fax +32 2 759 47 95
   prolog@sunbim.be
Yes
```

information/2

information(_Key, _Identification)

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

The information associated with key *arg1* is returned in the list *arg2*.

information/1

information(_Key)

arg1 : ground : atom

The information associated with key *arg1* is printed to the current error output stream.

Example

```
?- information('owner').
   BIM
   Kwikstraat 4
   B-3078 Everberg
   Belgium
   tel +32 2 759 59 25
   fax +32 2 759 47 95
   prolog@sunbim.be
Yes
```

information/0

information

All information is printed to the current error output stream.

Built-in Predicates

Chapter 3
Execution Flow

3.1 Logical flow

fail/0

fail

Always fails.

true/0

true

Always succeeds.

3.2 Mark and cut

!/0

!

Discards all choice points made since the parent goal started execution.

mark/1

mark(_CutMark)

arg1 : free : internal

Sets a marker and instantiates *arg1* to it. The marker can be used in a subsequent call to **cut/1**.

cut/1

cut(_CutMark)

arg1 : ground : internal

All choice points created since the call to **mark/1** with the same argument, are cut away. The argument of **cut/1** must be the same as the argument of a previously executed **mark/1**. A **cut/1** must always refer to a **mark/1** that is performed either in the same clause, or in a parent clause. More than one **cut/1** can refer to the same **mark/1**.

Example

```
a :- mark(_X), b, c, cut(_X).
a :- true.
```

The call to **mark/1** instantiates _X to a marker value (this value is irrelevant to the Prolog programmer). **cut/1** uses this marker value to cut away the choice points created by b and c, but not that created by a.

Note:
The scope of different mark/1, cut/1 couples and of !/0 should not overlap. This can lead to unexpected behavior.

Example: scopes that overlap.

```
a :- b, mark(_A0), !, c, cut(_A0) .
b.
b.
c.
c.

?- a.
Yes ;
Yes
```

a has two solutions while only one solution is expected.

3.3 Catch and throw

catch/3

catch(_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 **catch/3** succeeds. If it fails, this call also fails.

If during the execution of the goal *arg1*, **throw/1** is called, the execution control is changed. An implicit cut and fail is performed up to the call of **catch/3**. Then the ball argument of the **throw/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 of the **catch/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 **catch/3**. An error occurs if there is no active call of **block/3** whose catcher matches against the ball of the **throw/1**.

throw/1

throw(_Ball)

arg1 : partial : term

Terminates execution of the most enclosing **catch/3** call whose catcher argument matches the ball *arg1*.

Example

The catch and throw predicates are useful for error handling. The following call installs an application specific error handler for executing _Goal:

```
catch( _Goal,
     app_error(_ErrNr, _Data),
     app_err_handler(_ErrNr, _Data) )
```

To raise an error during execution of this goal:

```
throw( 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.

3.4 Conditional flow

This section describes two constructs for defining conditional flows.

->/2

_IfGoal -> _ThenGoal
_IfGoal -> _ThenGoal; _ElseGoal

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

The *_Ifgoal* is evaluated. If a solution is found, the *_ThenGoal* is evaluated. If no solution is found for the *_IfGoal*, the *_ElseGoal* is evaluated. One solution at the most is calculated for the *_IfGoal*. For the *_ThenGoal* and the *_ElseGoal*, all solutions are computed one by one upon backtracking.

Example

```
?- assert(test(a)).
Yes
?- ( test(a) -> write(ok); write(nok) ).
okYes
?- ( test(b)  ->   write(ok); write(nok) ).
nokYes
```

In general, a clause containing an **if-then**[**-else**] construction in ***ProLog by BIM*** is transformed in the following way.

An **if-then-else** construction

```
G1 -> G2; G3
```

becomes

```
mark( _Mark0), ( call(G1), cut( _Mark0), G2 ; G3)
```

An **if-then** construction:

```
G1 -> G2
```

becomes

```
mark( _Mark1), ( call(G1), cut( _Mark1), G2)
```

If the _IfGoal is a simple test, the generated code can be optimized, avoiding the choice point to be created. More information can be found in "*User's Guide - Optimizations*".

Note that a !/0 in the _IfGoal can lead to unexpected behavior (see **cut/1** for further explanation).

none/2 some/2

_TestGoal none _NoneGoal some _SomeGoal

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

Goal *arg1* is evaluated. If this fails, goal *arg2* is evaluated. If *arg1* succeeded, goal *arg3* is evaluated. If upon backtracking another solution for goal *arg1* is found, goal *arg3* is evaluated again. If no other solution for goal *arg1* could be found, the whole construction fails.

The goal

```
_TestGoal none _NoneGoal
```

is equivalent to

```
_TestGoal none _NoneGoal some true
```

The goal

```
_TestGoal some _SomeGoal
```

is equivalent to

```
_TestGoal none true some _SomeGoal
```

Example

```
a(_) :- fail.

?- ( a(_X) none write(no_a) some write(a(_X)) ), fail.
no_aNo
```

a/1 fails, so "no_a" is written out.

```
b(1).
b(2).
b(3).

?- ( b(_x) none write(no_a) some write(B(_x)) ), fail.
B(1)B(2)B(3)No
```

The query displays for each success of b, its result and fails.

3.5 Loop

repeat/0

repeat

This backtrackable predicate always succeeds. It is defined as:

```
repeat.
repeat :- repeat.
```

3.6 Exit from query

exit/0

exit

Stops the current query, and returns to the top level of the ***ProLog by BIM*** engine.

abort/0

abort

The same as **exit/0**.

toplevel/1

toplevel(_PredicateName)

arg1 : ground : atom

The current query is terminated immediately and the top-level query **?- _PredicateName** is started.

One usage of this predicate is to call it at the end of a signal handler, when the suspended query must not be resumed.

Built-in Predicates

Chapter 4
Metalevel

4.1 Metacall

call/1

call(_Goal)

arg1 : partial : term

This predicate is called the **metacall**. The predicate calls the goal *arg1* as a subgoal of call/1. If *arg1* fails, the predicate fails. If *arg1* succeeds, the predicate succeeds. If *arg1* contains a !/0, it has no effect outside **call/1**, i.e. it is local to the goal *arg1*.

It could be defined with the clause:

```
call(_Goal) :-
     abolish(callpred(_)),
     assert((callpred(arg(_Goal)) :- _Goal)),
     callpred(arg(_Goal)).
```

The predicate fails with an error message if *arg1* is free.

4.2 Negation

\+/1

not/1

\+ _Goal

not _Goal

arg1 : partial : term

This predicate succeeds if *arg1* is not provable, otherwise it fails. No check is made to see whether *arg1* is ground or not.

4.3 All-solution predicates

The use of the all-solution predicates presented in this section, is discussed in more detail in the "*User's Guide - Optimizations*".

findall/3

findall(_Collect, _Goal, _SolutionList)

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

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

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 a findall/3 query gives:

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

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).
```

```
?- findall([_x,_y],a(_x,_y,_z),_List).
 _x = _11
 _y = _13
 _z = _18
 _List = [[5,2],[7,3],[2,5],[3,3],[9,4]]
Yes
```

findall/4

findall(_Collect, _Goal, _SolutionList, _EndList)

arg1 : any : term
arg2 : partial : term
arg3 : any : term
arg4 : free : variable

Same as **findall/3,** but the solution list *arg3* is an open-ended list with *arg4* being the end.

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 a findall query gives:

```
?- findall(point(_x,_y), a(_x,_y,h), _list, _end).
 _x =  _11
 _y =  _12
 _list = [point(3,2), point(3,3) | _9 ]
 _end =  _9
Yes
```

bagof/3

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

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

Arg3 is a list of all instances of the term *arg1* for which the goal *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 occurring in *arg2* and not in *arg1* (even on free variables). This predicate fails if *arg2* is not properly instantiated or when no solutions are found.

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 occurring in _a. This is called **existential quantification.**

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, the results for a bagof/3 query are:

```
?- 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 *"Built-in Predicates - Comparison"*).

Example

With the same setup as in bagof/3, setof/3 will return the following results.

```
?- setof([_x,_y],a(_x,_y,_z),_List).
 _x = _11
 _y = _13
 _z = h
 _List = [[3,3],[5,2]]
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
?- setof([_x,_y],_z^a(_x,_y,_z),_List).
 _x = _11
 _y = _13
 _z = _16
 _List = [[2,5],[3,3],[5,2],[7,3],[9,4]]
Yes
```

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 a query of setof/3 gives:

```
?- setof([_x,_y], _z^a(_x, _y,_z), _list).
   _x = _9
 _y = _11
 _z = _14
 _list = [[2,5],[3,3],[5,2],[7,3],[9,4]]
Yes
```

Built-in Predicates

Chapter 5
Term Manipulation

5.1 Test predicates

Mode test predicates

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 test predicates

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 either a real, an integer, an atom or a pointer.

list/1

list(_Term)

arg1 : any : term

Succeeds if arg1 is a non-empty list.

Example

```
?- list([_El|f(b)]).
Yes
?- list([]).
No
```

compound/1

compound(_Term)

arg1 : any : term

Succeeds if arg1 is a compound term, i.e. a non-atomic term.

Example

```
?- compound([_El|_L]).
Yes
?- compound(f(1,2)).
Yes
?- compound('this is an atom').
No
```

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
Compound term	functor

Example

```
?- term_type(ox123,_T).
  _T = pointer
Yes
?- term_type([_El|_T],_T).
  _T = functor
Yes
```

is_inf/1

is_inf(_Term)

arg1 : any : term

Succeeds if *arg1* is an infinite term (is cyclic).

Example

```
?- please(wd,4).
Yes
?- _inf = g( f( atom, _inf ), _noninf),
   is_inf( _inf).
 _inf = g(f(atom, g(f(...,...), _13)), _13)
 _noninf = _13
Yes
```

5.2 Conversions

This section presents a number of predicates which allow the conversion of terms from one ***ProLog by BIM*** type to another. Besides the predicates explained here, there exist also **sread/2** and **swrite/2**, predicates which allow the parsing of an atom into a term and the writing of a term into an atom (see "*Built-in Predicates - Input and Output*").

Conversion of simple 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.

Example

```
?- ascii( a, _ASCII).
 _ASCII = 97
Yes
?- ascii( _Char, 48).
 Char = 0
Yes
?- ascii( _Char, 0'a).
 _Char = a
Yes
?- ascii( '\n', _ASCII).
 _ASCII = 10
Yes
```

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.

Example

```
?- pointertoint(_Pointer, 32).
 _Pointer = 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.

Example

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

stringtoatom/2

stringtoatom(_StringPointer, _Atom)

arg1 : any : pointer
arg2 : any : atom

The pointer to a character array *arg1* corresponds to the atom *arg2*.

At least one of the arguments must be instantiated.

If *arg1* is free, it is instantiated to a pointer which addresses the character array representing the text of atom *arg2*. This array should in no case be overwritten. It is not guaranteed to always represent the same text.

If *arg2* is free, it is instantiated to an atom that has the character array, pointed to by *arg1*, as text. The array can be deallocated after this call.

If both arguments are instantiated, the character array, pointed to by *arg1*, is compared with the text of atom *arg2*.

This predicate may crash ***ProLog by BIM*** if it is used with a pointer that does not address a valid string.

Example

```
?- stringtoatom(_p, 'an_atom').
 _p = 0x202f8bed
Yes
?- stringtoatom(0x202f8bed, _a).
 _a = an_atom
Yes
```

Conversion of lists

The predicates **atomtolist/2** and **name/2** allow the conversion between an atomic type and a list of characters or ASCII codes. When a list is transformed into an atomic value, its type will be determined by the elements of the list. The resulting type can be :

- a pointer (if the list starts with the elements "0" and "x").
- an integer (if all the elements of the list *arg2* are numerical characters, which ranges between "0" and "9").
- a real (if the list is a combination of numeric characters, and a dot).
- an atom (in all other cases).

Example

```
?- please(wq,on). % use quotes where appropriate
Yes
?- atomtolist(_a,['1','2','e','f']).
 _a = '12ef'
Yes
?- atomtolist(_a,['1','2','e']).
 _a = 1.200000000000000e+01
Yes
?- atomtolist(_a,['1','2']).
 _a = 12
Yes
?- atomtolist(_a,['0',x,'1']).
 _a = 0x1
Yes
?- atomtolist(_a,['+','1','2','e']).
 _a = '+12e'
Yes
```

Due to internal conversions (on numbers), the mappings provided by these predicates is not a one-to-one mapping:

Example

```
?- atomtolist( _x, ['0','3']) , atomtolist( _x, ['3']).
?- name( _x, [48,51]) , name( _x, [51]).
```

Both queries succeed.

atomtolist/2

atomtolist(_Atomic, _ListOfChar)

arg1 : any : atomic
arg2 : any : list

Arg2 is the list built from the characters of *arg1*. Therefore, *arg2* is a list of atoms of length one.

At least one of the arguments must be ground.

Example

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

name/2

name(_Atomic, _ListOfAsciiCodes)

arg1 : any : atomic
arg2 : any : list of integer

As **atomtolist/2**, except that *arg2* is a list of ASCII codes instead of characters.

Example

```
?- please(wq, on).
?- name(_x, [51,52,53,101,49]).
 _x = 3.45000+e03
Yes
?- name('Prolog', _List).
 _List = [80,114,111,108,111,103]
Yes
```

asciilist/2

asciilist(_Atom, _ListOfAsciiCodes)

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 codes of the characters of the atom.

At least one of the arguments must be ground.

Example

```
?- please(wq, on). % use quotes where appropriate
?- asciilist(_Atom, [51,52,53,101,49]) .
 _Atom = '345e1'
Yes
?- asciilist('345e1', "345e1").
Yes
```

Conversion of terms

=../2

_Term =.. [_Functor _ArgList]

arg1 : partial or free
arg2 : any : list

This built-in predicate is called **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 []-terminated list from which the first element is of type atomic.
If *arg1* is atomic, the corresponding arg2 is a list, containing only *arg1*.

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 : term
arg2 : any : atomic
arg3 : any : integer

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

Example

```
?- functor(_Term, a, 3).
 _Term = a(_2, _3, _4)
Yes
?- functor(a(_, _, _), _Functorname, _Arity).
 _Functorname = a
 _Arity = 3
Yes
```

```
?- functor(1, _Functorname, _Arity).
 _Functorname = 1
 _Arity = 0
Yes
```

arg/3

arg(_ArgNumber, _Term, _Arg)

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

Unifies the *arg1*'th argument of *arg2* with *arg3*

If *arg1* <= 0 or *arg1* > arity of *arg2*, the predicate fails without giving an error message.

Example

```
?-arg(2,a(43, 12, 76), _Arg).
 _Arg = 12
Yes
?- arg(2,a(1,_2,3),two).
  _2 = two
Yes
```

namevars/4

namevars(_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.

If X denotes 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
.....
(X + last letter) + (arg3-1)
(X + last letter of arg4) + first letter + arg2
...

first letter is either A or a, and last letter is either Z or z, depending on the case of the last letter of arg4.

Example

```
?- namevars(a(_w, _x, _y, _z), 3, 5, abc).
 _w = abc3
 _x = abc4
 _y = abd3
 _z = abd4
Yes
```

namevars/3

namevars(_Term, _LowNum, _HighNum)

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

namevars/3 is defined as:

```
namevars( _x, _y, _z) :- namevars( _x, _y, _z, 'A').
```

Example

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

numbervars/3

numbervars(_Term, _Number1, _Number2)

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

All variables in term *arg1* are instantiated to a term of the form '$VAR'(_N) where _N is a number that starts from number *arg2* for the leftmost variable and increases by one for each subsequent variable. *Arg3* is unified with the next number that would be used if there were more variables in the term.

The numbers must be greater than or equal to 0.

When writing out a term, any terms of the form '$VAR'(_N) are written as a named variable, with the name defined by the number _N. The names used are _A to _Z for numbers 0 to 25, _A_1 to _Z_1 for numbers 26 to 51, and so on. In compatibility syntax, the leading underscores are omitted.

Example

```
?- numbervars(a(_w, _x, _y, _z), 24, _i).
 _w = _Y
 _x = _Z
 _y = _A_1
 _z = _B_2
_i = 28
Yes
```

copy/2

copy(_Term, _NewTerm)

arg1 : any : term
arg2 : any : term

A copy of *arg1* is made with different variables, and unified with *arg2*.

Example

```
?- _var1 = f(_v, atom , f( _v1, _v2)), copy(_var1, _var2).
 _var1 = f(_12, atom, f(_12, _17))
 _v1 = _12
 _v2 = _17
 _var2 = f(_75, atom, f(_75, _80))
Yes
```

varlist/2

varlist(_Term, _VarList)

arg1 : any : term
arg2 : any : list of variables

All variables appearing in term *arg1* are collected in the list *arg2*.

Example

```
?- varlist( f( _V1, atom, f( _V1, _V2)), _Vl).
 _V1 = _9
 _V2 = _14
 _Vl = [_9,_14]
Yes
```

5.3 Atom manipulation

atomlength/2

atomlength(_Atom, _Length)

arg1 : ground : atom
arg2 : any : 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

Arg3 is the concatenation of items *arg1* and *arg2*. At most one of the arguments may be free.

The instantiated arguments can be of type atom, integer, real or pointer. Non-atom arguments are automatically converted to atoms before concatenation.

Example

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

atomconcat/2

atomconcat(_ListOfAtomics, _ConcatAtom)

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

Arg2 is the atom that is constructed by concatenating all items of the list *arg1* in the given order.

The items of *arg1* can be of type atom, integer, real or pointer. Non-atom arguments are automatically converted to atoms before concatenation.

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*.

Example

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

atompart/4

atompart(_Atom, _AtomPart, _StartPos, _Length)

arg1 : ground : atom
arg2 : any : atom
arg3 : any : integer > 0
arg4 : any : integer > 0

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 respectively 1 and the remaining length of *arg1*, when they are not specified.

If *arg2* is instantiated and *arg3* is free, *arg3* will be instantiated to the starting position of the first occurrence of *arg2* in *arg1*.

Example

```
?- atompart('pattern matching','chi', _start, _length).
 _start = 12
 _length = 3
Yes
?- atompart('pattern matching', _part, _start ,_length).
 _part = 'pattern matching'
 _start = 1
 _length = 16
Yes
?- atompart('pattern matching', _part, 3, _length).
 _part = 'ttern matching'
 _length = 14
Yes
?- atompart('pattern matching', _part, 3, 4).
 _part = tter
Yes
?- atompart('pattern matching','ab', _, _).
No
```

atompartsall/3

atompartsall(_Atom, _AtomPart, _StartPos)

arg1 : ground : atom
arg2 : ground : atom
arg3 : any : 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).

Example

```
?- atompartsall('pattern matching', 'at', _pos).
 _pos = 2
Yes ;
 _pos = 10
Yes ;
No
```

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.

Example

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

The characters 'c' and 'h' do not occur in the string 'at'.
The character 'a' is the first character that does occur in the string 'at'.

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 position is smaller than 1, it takes the value of the beginning (1). If the start position is beyond the end, it takes the value of the end position.

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*.

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.

Example

```
?- _M = 'lower2UPPER' ,
      lowertoupper(_M, _U),
      lowertoupper(_L, _M).
 _M = lower2UPPER
 _U = 'LOWER2UPPER'
 _L = lower2upper
Yes
```

5.4 List manipulation

listconcat/3

listconcat(_List1,_List2,_List3)

arg1 : partial : list
arg2 : any : list
arg3 : free : list

Concatenates list *arg2* to the end of list arg1 and returns the resulting list in *arg3*.

Note
Arg2 does not really have to be a list. Whatever it is, it will be appended at the end of list *arg1*.

Built-in Predicates

Chapter 6
Expression Evaluation

6.1 Evaluation of expressions

An evaluable expression is a term that is constructed from numerical and/or textual constants and from computable functions.

ProLog by BIM has two classes of expression evaluation: general and arithmetic. The General Expression Evaluation works on numerical (integer and real) and textual (atom) data. Arithmetic Expression Evaluation only accepts numerical data but is more optimized. Therefore, it is recommended to use arithmetic expression evaluation whenever it is known that the arguments are numerical, and execution speed is critical.

The following table lists expression evaluation operators:

Operator	*General*	*Arithmetic*
Built-in predicate	=?/2	is/2
In-line evaluation	'=?'/1	?/1
Comparison		</2 >/2 =</2 >=/2
		<>/2 =\=/2 =:=/2

An expression can be evaluated by calling a built-in predicate, or by using an in-line evaluation operation.

Any in-line evaluation term is replaced by the result of the evaluation of its argument. This is accomplished by a compile time source transformation.

Example

Suppose the file expr.pro contains the clause:

```
cl( Var ) :- write(?( Var  + 1)), nl.
```

```
?- consult(expr).
compiling expr.pro
loaded expr.pro
Yes
?- cl(3).
4
Yes
```

The clause actually executed, looks like:

```
cl( Var ) :- X is Var + 1, write(X), nl.
```

An in-line evaluation appearing in the head of a clause has the same effect as if the expression were the first subgoal of the clause.

In-line evaluation can be suppressed with the compiler option -e-. In that case, the operator is treated as a normal functor.

Example

```
?- consult('-e- expr').
compiling -e- expr.pro
loaded expr.pro
Yes
?- cl(3).
?(3 + 1)
Yes
```

Computable functions

The table below lists all computable functions, grouped by operator definition and by arity. The argument types are indicated. A type "number" corresponds to "integer" or "real". Type "string" is synonym for "atom". Type "atomic" is either an atom, a number or a pointer.

Name/ Precedence	Result	Arguments	Function
Unary operators			
+ 500	number	number	unary plus
- 500	number	number	unary negation
\ 500	integer	integer	bitwise not (complement)
Binary operators			
+ 500	number	number, number	addition
+ 500	pointer	pointer, integer	pointer offset
- 500	number	number, number	subtraction
- 500	pointer	pointer, integer	pointer offset
- 500	integer	pointer, pointer	pointer offset
/\ 500	integer	integer, integer	bitwise and (conjunction)
\/ 500	integer	integer, integer	bitwise or (disjunction)
xor 500	integer	integer, integer	bitwise xor (exclusive or)
* 400	number	number, number	multiplication
/ 400	number	number, number	real division
// 400	integer	integer, integer	integer division
<< 400	integer	integer, integer	bitwise left arithmetic shift
>> 400	integer	integer, integer	bitwise right arithmetic shift
mod 300	integer	integer, integer	modulo division
rem 300	integer	integer, integer	remainder division
** 300	number	number, integer	exponentiation
^ 300	number	number, integer	exponentiation
|| 600	string	atomic, atomic	string concatenation
Constant functors			
pi	real		constant pi
cpu_time	real		expired cpu time
Unary functors			
int	integer	number	truncate to integer
trunc	integer	number	truncate to integer
round	integer	number	round to integer
real	real	number	expand to real
float	real	number	expand to real
string	string	atomic	convert to string
sign	integer	number	sign (-1, 0 or 1)
abs	number	number	absolute value
len	integer	string	length of string
code	integer	string	code of character
char	string	integer	character for code

sqrt	real	number	square root
exp	real	number	exponential
log	real	number	natural logarithm
log10	real	number	decimal logarithm
sin	real	number	sine
cos	real	number	cosine
tan	real	number	tangens
asin	real	number	arc sine
acos	real	number	arc cosine
atan	real	number	arc tangens
int_to_hex	string	integer	hex string image of integer
lower	string	string	lower case string conversion
upper	string	string	upper case string conversion
strip	string	string	strip string

Binary functors

gcd	integer	integer, integer	greatest common denominator
random	integer	integer, integer	random number from range
pow	real	number, number	power
atan2	real	number, number	arc tangens of arg1/arg2
int_to_hex	string	integer, integer	hex string image of integer
substring	string	string, integer	substring
strip	string	string, integer	strip string
min	term	term, term	standard order minimum
max	term	term, term	standard order maximum

Ternary functors

substring	string	string, integer, integer	substring
strip	string	string, integer, string	strip string

Notes

- Constant functors should be specified with name and ().

```
_Res is pi() + cputime() .
```

- Where necessary, integers are converted to reals before calculation.

```
?- _r is 5. + 4 .
 _r = 9.00000e+00
Yes
```

- Arithmetic on reals is always performed in double precision.
- Bitwise operations are performed on all 29 significant bits.

```
?- _x = 2'101, _y is \ _x, _z is _x /\ _y .
 _x = 5
 _y = -6
 _z = 0
Yes
```

- For pointer offset calculations, an offset must be an integer.
  Two pointers cannot be added.
- Real division (/) has a real result if at least one of the arguments is a real.

```
?- _x is 7 / 3  .
 _x = 2
Yes
```

- The arguments of the integer division (//) must be integers. Integer division truncates towards 0.
  For negative arguments it is defined as:
  $$\_x // \_y = sign(\_x) * sign(\_y) * (abs(\_x) // abs(\_y))$$

```
?-_x is -7 // 3 .
 _x = -2
Yes
```

- Remainder (**rem**) is defined as:
  $$\_x \text{ rem } \_y = \_x - (\_x // \_y) * \_y$$
- Modulo (**mod**) is defined as:
  $$\_x \text{ mod } \_y = \_x \text{ rem } \_y , \ sign(\_x) = sign(\_y)$$
  $$\_x \text{ mod } \_y = \_x \text{ rem } \_y + \_y , \ sign(\_x) \neq sign(\_y)$$
  Such that the following statements hold:
  $$0 =< \_x \text{ mod } \_y < \_y , \ \_y >= 0$$
  $$\_y < \_x \text{ mod } \_y =< 0 , \ \_y =< 0$$
- The function **int** truncates a number to the largest integer smaller than or equal to it. The following statement holds:
  $$\_x - 1 < int(\_x) =< \_x$$
- The function **trunc** is the same as **int** for a positive argument, and it is symmetric around 0:
  $$trunc(- \_x) = - trunc(\_x)$$
  The following statements hold:
  $$\_x - 1 < trunc(\_x) =< \_x , \ \_x >= 0$$
  $$\_x =< trunc(\_x) < \_x + 1 , \ \_x =< 0$$
- The function **round** is defined as:
  $$round(\_x) = sign(\_x) * trunc(abs(\_x) + 0.5)$$
  This function is symmetric around 0.
- Exponentiation with ** or ^ requires an integer exponent. The function **pow** must be used for a real (or integer) exponent.
  The result is of the same type as the base argument.
- The arguments of a string concatenation function (||) are first converted to strings with *string* if necessary.
- Conversion of a number to a string is as if it would be written out.

```
?- please(fr, '%.5e').
Yes
?- _y =? string(pi()).
 _y = '3.14159e+00'
Yes
```

- Conversion of an integer to a string in hexadecimal notation is performed by the function **int_to_hex**. An optional second argument specifies the field width, in which case the string is left padded with 0 s.

```
?- _x =? int_to_hex(543, 4),
   _s =? '16\''||_x,
   sread(_s, _i).
 _x = '021F'
 _s = '16\'021F'
 _i = 543
Yes
```

- The result of **random/2** is a random integer in the range of its arguments. The random number generator can be seeded with the built-in **srandom/1**.
- The arguments of **substring/3** are:

  arg1 : atom : original string
  arg2 : integer : starting index
  arg3 : integer : length of substring

- The arguments of **substring/2** are:

  arg1 : atom : original string
  arg2 : integer : starting index

  The index *arg2* starts from 0 for the first character. The length of the substring is the length of the original string minus the starting index.

- The arguments of **strip/3** are:

  arg1 : atom : original string
  arg2 : integer : strip position
  arg3 : atom : strip character set

- The characters in *arg3* are stripped from *arg1*. The value of *arg2* determines the location(s) where the strip must be performed, at the beginning, in the middle and/or at the end of *arg1*.

  Arg2 can be specified as a 3-bit binary number.
  - bit 0: strip ending
  - bit 1: strip middle
  - bit 2 : strip beginning

  Arg2 can also be specified as an integer, in this case only the value of the 3 last bits of this integer are considered (modulo 8).

arg2	B	M	E
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

  1 = strip
  0 = do not strip

  B= Beginning.
  A character is in the beginning of the string, if it and all characters preceding it, are in the character set *arg3*.

  M= Middle.
  A character is in the middle, if it is in the character set *arg3*, and if it is neither in the beginning nor in the end.

  E= Ending.
  A character is in the ending of the string, if it and all characters following it, are in the character set *arg3*.

- The arguments of **strip/2** are:

  arg1: atom: original string
  arg2: integer: strip position

  Blanks(' ') are stripped from *arg1*.

- The arguments of **strip/1** are:

  arg1: atom: original string

  Blanks in the beginning and the ending of *arg1* are stripped.

```
?- _x =? strip('  this atom will be stripped  ', 6, 'abcd ').
 _x = 'thistomwillestripped '
Yes
?- _x =? strip('   this long atom will be stripped   ', 2'110).
 _x = 'thisatomwillbestripped      '
Yes
?- _x =? strip('     this long atom will be stripped      ').
 _x = 'this atom will be stripped'
Yes
```

General evaluation

=?/2

_Result =? _Expression

arg1 : any : atomic
arg2 : ground : term

Term *arg2* is evaluated and the result is unified with *arg1*. The result is integer, real or atom.

The term *arg2* must be an expression, built with integer, real or atom constants and computable functions.

Note: The in-line equivalent of **=?/2** is:

'=?'/1

'=?'(_GeneralExpression)

arg1 : ground : expression

The general expression *arg1* is evaluated in-line. This is accomplished by a compile time source translation. The compiler option **-e-** can be used to disable this translation.

Example

```
?- _x = on,
   write('=?'(substring( (c||_x||catenati||_x),2,8) )).
ncatenat
 _x = on
Yes
```

Arithmetic evaluation

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 an arithmetic expression built with integer, real or atom constants and computable functions.

Note: The in-line equivalent of **is/2** is:

?/1

?(Arithmetic_Expression)

arg1 : ground : expression

The arithmetic expression *arg1* is evaluated in-line. This is accomplished by a compile time source translation. The compiler option **-e-** can be used to disable this translation.

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 ).
```

6.2 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*. One of the arguments can be free at the most. The offset can be positive or negative.

6.3 Random generator

The random generator available 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.

Example

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

Built-in Predicates

Chapter 7
Unification and Comparison

7.1 Unification

?=/2

_Term1 ?= _Term2

arg1 : any : term
arg2 : any : term

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

Example

```
?- f(_X,5) ?=  f(6,_Y), write(_X ?=_Y), nl.
 _13 ?= _17
Yes
?- 5 ?= 76  .
No
```

=/2

_Term1 = _Term2

arg1 : any : term
arg2 : any : term

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

arg1 and *arg2* may be infinite terms.

Example

```
?- _X = p(4, b).
 _X = p(4, b)
Yes
?- p(4, _Y) = p(_X, b).
 _X = 4
 _Y = b
Yes
?- p(4, a ) = p(4, b).
No
?- please(wd,2),
   _X = f(a,_X),
   _Y = f(a,_Y),
   _X = _Y.
 _X = f(a,f(a,f(...,...)))
 _Y = f(a,f(a,f(...,...)))
Yes
```

\=/2

_Term1 \= _Term2

arg1 : any : term
arg2 : any : term

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

Arg1 and *arg2* may be infinite terms.

Example

```
?- _X \= p(4, b).
No
?- p(4, _Y) \= p(_X, b).
No
?- p(4, a ) \= p(4, b).
Yes
```

mgu/3

mgu(_Term1, _Term2, _Term3)

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

Unifies *arg3* with the **Most General Unifier** of *arg1* and *arg2*, without actually unifying these terms. The MGU is represented as a list of terms of the form _Var = _Term, such that if all _Var are unified with the corresponding _Term, the terms *arg1* and *arg2* are effectively unified.

occur/2

occur(_Term1, _Term2)

arg1 : any : term
arg2 : any : term

This predicate implements "sound" unification. It unifies *arg1* with *arg2* if the unification does not create infinite terms, otherwise it fails.

Unification in ***ProLog by BIM*** handles unification of infinite terms correctly. It treats the terms as rational trees. However the inferences are unsound.

Example

```
successor(_N,s(_N)).
?- successor(X,X).
 _X = s(s(...)).
Yes
```

The answer to the question if there is a number equal to its successor has an answer, whereas in Peano arithmetic it should not.

Example

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

occurs/2

occurs(_Part, _Term)

arg1 : any : atomic
arg2 : any : term

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

Example

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

7.2 Equality

==/2

_Term1 == _Term2

arg1 : any : term
arg2 : any : term

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

For this comparison to succeed, any variables within the terms must be identical. No unification is performed.

Example

```
?- f(5,_X) == f(5,_X).
Yes
?- f(_Y) == f(_X).
No
?- 5 == 5.0 .
No
?- please(wd,3), _X = f(a,_X), _Y = f(a,f(a,_Y)), _X == _Y.
   _Y = f(a,f(a,f(...,...)))
   _X = f(a,f(a,f(...,...)))
Yes
```

\==/2

_Term1 \== _Term2

arg1 : any : term
arg2 : any : term

Equivalent to the negation of ==/2.

Example

```
?- 5.0 \== 5 .
Yes
?- 5 \== 5 .
No
?- f(_X) \== f(_Y).
Yes
```

7.3 Arithmetic comparison

The arguments of the built-in predicates described in this section, must be ground and built according to the rules for arithmetic expressions (see "*Built-in Predicates - Expression Manipulation*"). Atom comparison is not possible with 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*

7.4 Standard order comparison

There is no restriction on the arguments of the built-in predicates described in this section. They refer to the "standard order of terms".

Terms are ordered according to the following criteria:

- Variables @< atoms @< numbers @< pointers @< terms.
- Variables are ordered according to their age.
- 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.

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
```

compare/3

compare(_Order, _Term1, _Term2)

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

The terms *arg2* and *arg3* are compared along the standard order comparison rules (@ operators). The result, which is as in the following table, is unified with *arg1*.

Result	Order
<	_Term1 @< _Term2
=	_Term1 == _Term2
>	_Term1 @> _Term2

Example

```
?- compare(_r, funt(_t, _4), funt(_t , f(_))).
 _r = '<'
 _t = _10
 _4 = _11
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*.

Example

```
?- sort(  [2,abc,f(3),2], _X  ).
   _X = [abc2,f(3)]
Yes
```

keysort/2

keysort(_List, _SortedList)

arg1 : partial : list of (term-term)
arg2 : any : list of (term-term)

List *arg1* should contain elements of the form (_Key-_Value), where both _Key and _Value may be terms. The list 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.

Example

```
?- keysort([k3-val3,k1-val1,k3-val3b,k2-val2], _X).
   _X = [k1-val1,k2-val2,k3-val3,k3-val3b]
Yes
?- _Unsort = [f(k1)-23, f(g(g))-23, f(k1)-22, f(g(_g))-_g],
     keysort(_Unsorted, _Sorted).
 _Unsorted = [f(k1)-23, f(g(t))-23, f(k1)-22, f(g(_g))-_g]
 _Sorted = [f(k1)-23, f(k1)-22), f(g(_g))-_g, f(g(t))-23]
Yes
```

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

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

keysort/3

keysort(_List, _KeyPred, _SortedList)

arg1 : partial : list of (term-term)
arg2 : ground : atom
arg3 : any : 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(_KeyValueTerm, _Key)

arg1 : ground : term
arg2 : free : term

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

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* must consist of elements of the form *arg2* (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 *arg2* and the key *arg3*. The resulting list is unified with *arg4*. Duplicates are not removed.

Example

```
data( [table(Smith, John,  London),
      table(VanHalen, John, NewYork),
      table(Smith, Emmy, Brussel)] ).

?- data( _Data ),
   keysort(_Data,table(_K1,_K2,_),_K1-_K2,_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, then on the second argument.

Built-in Predicates

Chapter 8
In-core Database

8.1 In-core database manipulation

The dynamic predicates can be considered as a database with operations as retrieve, store and update. Retrieving information is done either by calling the predicate with free variables in the call or by using clause. Storing information is done with the assert predicate. Updating is done with a retract to remove and retrieve information and a subsequent assert. The effect of changes to the in-core database on pending queries -for which not all answers have been returned- must be taken into account. The draft standard prescribes what is called the logical update view. ***ProLog by BIM*** implements this view. Note that previous versions of ***ProLog by BIM*** used a different implementation.

The basic rule for the logical view on updates is that the clauses that are used for any call of either the predicate or the built-in operations assert, retract and clause, are those that exist at the time the operation on the predicate is initiated. Subsequent modifications will not influence the further execution.

Example

```
:- dynamic data/1.
data(1).
data(2).
data(3).

?- data(X), Xplus1 is X + 1, retract(( data(Xplus1) )).
 Xplus1 = 2
 X = 1
Yes;
 Xplus1 = 3
 X =2
Yes;
No
?- listing(data).
 data(1).
Yes
```

Notice how the system backtracks over all alternatives for the call **data/1**, even though the retract call removes clause after clause before they are used in that call to data/1.

Example

```
:- dynamic data/1.
data(1).
data(2).

?- data(X), Xplus1 is X + 1, assertz(data(Xplus1)).
 Xplus1 = 2
 X = 1
Yes ;
 Xplus1 = 3
 X = 2
Yes
?- listing(data).
 data(1) .
 data(2) .
 data(2) .
 data(3) .
Yes
```

Assertz/1 adds definitions at the end. The call to **data/1** does not use these new definitions, since they are created after initiating the call.

Predicate naming convention

For the assert, retract and clause predicates there exist many variations. Two extensions to the basic operation are added: the use of database references and the use of variable name association lists.

Database references are used to uniquely identify a clause. They complement the pattern matching used by the standard versions of the predicates. These references should only be used with these predicates and passively passed in the rest of the program. Their format and type can change between different releases and one should not depend on it. Their main use is to avoid multiple searches and as a communication means between the database update predicates. The derived predicate is formed by prepending the letter **r** to the name of the predicate and to add a new argument at the end which must be free or a valid database reference.

Variable name association lists are used to preserve the symbolic names that variables have in the source. The argument must be free or a []-terminated list with pairs *atom* = *var*. The functor =/2 must be used. For the predicates derived from clause and retract the system will build such a list. The predicate read also has a variant to produce such a list. An association list may be given as input to the predicates derived from assert and write. The name of the derived predicates is formed by prepending the letter **v** to the name of the base predicate. The predicate itself is extended at the end with an extra argument for the association list.

Both extensions can be used at the same time. In that case the predicate name is formed by prepending **rv** to the base name and by adding two arguments at the end: first the reference argument and last the association list argument.

Asserting clauses

This section enumerates the different versions of the assert predicates that are available in ***ProLog by BIM***. The assert predicate is used to add definitions to the in-core database.

assert/1

asserta/1

assertz/1

assert(_Clause)
asserta(_Clause)
assertz(_Clause)

arg1 : partial : clause

rassert/2

rasserta/2

rassertz/2

rassert(_Clause, _ClauseRef)
rasserta(_Clause, _ClauseRef)
rassertz(_Clause, _ClauseRef)

arg1 : partial : clause
arg2 : free : internal

vassert/2

vasserta/2

vassertz/2

vassert(_Clause, _VarNameList)
vasserta(_Clause, _VarNameList)
vassertz(_Clause, _VarNameList)

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

rvassert/3

rvasserta/3

rvassertz/3

rvassert(_Clause, _ClauseRef, _VarNameList)
rvasserta(_Clause, _ClauseRef, _VarNameList)
rvassertz(_Clause, _ClauseRef, _VarNameList)

arg1 : partial : clause
arg2 : free : internal
arg3 : partial : list of (atom = _var)

The clause *arg1* is asserted. The built-ins with suffix a assert the clause at the beginning of the predicate's clause chain. Those with z or without suffix, assert at the end of the chain.

In the argument _ClauseRef, a clause reference which uniquely identifies this clause (as long as it exists) is returned. A list which contains the names of the variables in the clause, can be passed in argument _VarNameList.

Retracting clauses

An enumeration is given of the different retract predicates that are built-in in ***ProLog by BIM***. The retract predicates are used to delete clauses from the in-core database.

retract/1

retract(_Clause)

arg1 : partial : clause

All clauses matching *arg1* are retracted one by one upon backtracking.

rretract/1

rretract(_ClauseRef)

arg1 : ground : internalt

The clause with reference *arg1* is retracted.

rretract/2

rretract(_Clause, _ClauseRef)

arg1 : any : clause	*partial : clause*
arg2 : ground : internal	*free : internal*

With the first use, the clause referenced by *arg2* is unified with *arg1*. If the unification succeeds, the clause is retracted, otherwise, the predicate fails.

With the second use (*arg2* is free), the predicate deletes the matching clauses and instantiates *arg2* with the clause reference, one by one upon backtracking.

The above defined predicates behave differently when used for hidden predicates. In that case, the unification of the argument _Clause is checked but the argument is not instantiated.

The next three predicates are used to delete all definitions of a predicate. These built-ins are not only applicable to dynamic predicates but can also be used on static predicates.

retractall/1

retractall(_ClauseHead)

arg1 : partial : term

For dynamic predicates, all clauses whose head match *arg1* are retracted at once. Only the real definitions are retracted. Any declarations (mode, index, hidden) remain effective.

For static predicates, the built-in behaves as **abolish/1**.

This predicate always succeeds.

```
?- assert(a),retractall(a),
 listing(a/0),
 predicate_type(a,_t).
 _t = dynamic
Yes
```

Although **a/0** does not have any more definitions in the in-core database.The predicate is still declared as dynamic.

```
?-retractall(b(_Var1,_Var2)).
```

abolish/1

abolish(_ClauseHead)

arg1 : partial : term

All definitions and declarations of the predicate with same principal functor as *arg1* are retracted. This predicate always succeeds.

```
?- assert(a),abolish(a),
 listing(a/0),
 predicate_type(a,_t).
No
```

predicate **a/0** is completely unknown to the system: it does not have any definitions or pseudo definitions any more.

```
?- abolish(b(_Var1,f(1,2))).
Yes
```

abolish/2

abolish(_Name, _Arity)

arg1 : ground : atom
arg2 : ground : integer

All definitions and declarations for the predicate with name *arg1* and arity *arg2* are retracted. This predicate always succeeds.

```
?- abolish(a,2).
Yes
```

Updating

update/1

update(_Clause)

arg1 : partial : clause

Asserts *arg1* after retraction of all clauses with same name and arity as the head of *arg1*.

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

Example

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

a(123) .
a(456) .
---
a(321) .
Yes
```

Retrieving clauses

The clause predicates that ***ProLog by BIM*** provides, are used to retrieve information from the in-core database.

clause/2

clause(_Head, _Body)

arg1 : partial : term
arg2 : any : term

rclause/3

rclause(_Head, _Body, _ClauseRef)

arg1 : partial : term	*any : term*
arg2 : any : term	*any : term*
arg3 : free : internal	*ground : internal*

vclause/3

vclause(_Head, _Body, _VarNameList)

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

rvclause/4

rvclause(_Head, _Body, _ClauseRef, _VarNameList)

arg1 : partial : term	*any : term*
arg2 : any : term	*any : term*
arg3 : free : internal	*ground : internal*
arg4 : free : list of (atom = _var)	*free : list of (atom = _var)*

A clause with head matching *_Head* is returned. Its head is unified with *_Head* and its body with *_Body*. The body of a fact is the atom true.

When the argument _ClauseRef is free, the clause reference of the matching clause is returned in _ClauseRef. When it is instantiated, the clause with that reference is returned (if unifiable with the provided _Head).

A mapping of variable = name is returned in _VarNameList. The variables in this list may get instantiated because of instantiations during the clause head unification.

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
```

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
```

The clause predicates behave differently when used for hidden predicates. In that case, the unification of the arguments _Head and _Body is checked but the arguments are not instantiated.

Referenced clause inspection

Each clause in the database has a "clause reference" and a "file specification". Given a clause reference, the file specification can be retrieved. Such a file specification has the form Path/Name. Path is the path to the directory in which the file resides. It is an absolute path with a trailing '/'. Name is the name of the file in that directory.

If the clause was added interactively (typed in or asserted during program execution), the special value 'user' is returned as file specification.

Whenever a file specification must be entered, it can be given either in the above form, or as a single atom including the whole path and file name. In both cases, usual file expansion symbols may be used.

rfile/2

rfile(_ClauseRef, _File)

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

Arg2 is unified with the source file specification in which the clause with reference *arg1* is defined. This is either a structure of the form Path/Name, or the atom 'user' in case the clause was defined interactively.

Example

```
?-rassert( rasserted_clause, _ref).
 _ref = 67501059
Yes
?-rfile(67501059, _file).
 _file = user
Yes
```

rdefined/1

rdefined(_ClauseRef)

arg1 : ground : integer

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

8.2 Program inspection

Listing of predicates

ProLog by BIM offers the possibility to list the predicates currently in the in-core database. Only dynamic non-hidden predicates, and predicates that are compiled for debugging will be listed. Static predicates, built-in predicates and all hidden predicates are not visible to the user.

Only clauses are listed, not the declarations. If a list of all active operator and dynamic definitions is required, use the predicates **all_directives/0,1**, defined in the next section.

listing/0

listing

All clauses of non-hidden dynamic predicates in the Prolog database are written to the current output stream. The output looks like a Prolog source program without directives. The operator definitions which are active at the moment of the listing, are used to format the output.Variables are written using their symbolic names if known. Otherwise variables appear as an underscore, followed by a name. This name may differ from call to call. The predicates supporting a variable name association list (**vassert, vread**) can be used to provide variable names.

Example

```
?- op( 100, xfx, a) .
?- assert( (X a Y :- write(ok)) ) .
?- listing.
 _a0 a _a1 :-
   write(ok) .
Yes
```

listing/1

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

arg1 : ground : atom
atom/integer
[]-terminated list of atom or atom/integer

Arg1 must be one of the following forms:

Atom: the clauses of the non-hidden dynamic predicates with functor name *arg1* are listed to the current output stream.

Atom/Integer: the clauses of the non-hidden dynamic predicate with functor name *_Predname* and arity *_Arity* are listed to the current output stream.

A []-terminated list of the above two forms: for each member of the list the corresponding **listing/1** instruction is executed.

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 defined by *arg1*. This file has to be open 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

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 clauses of non-hidden dynamic 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 specified by *arg1*.

file_listing/1

file_listing(_File)

arg1 : ground : atom/atom or atom

All non-hidden dynamic clauses defined in file *arg1* are listed on the current output stream. The file can be specified as Path/Name or as a single path. Specifying "user" as *arg1* will list all the asserted clauses.

Example

```
?- file_listing('example_a.pro').

a1( _44 ) :-
    a2 .

a2 .
Yes
```

Example

```
?- assert(a) .
Yes
?- file_listing(user)

a .
Yes
```

file_listing/2

file_listing(_LogFileName, _File)
file_listing(_FilePointer, _File)

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

All non-hidden dynamic clauses defined in file *arg2* are listed in file *arg1*. The file specification *arg2* must be of the form Path/Name or a single atom. Specifying "user" as *arg2* will list all the asserted predicates.

Example

```
?- consult(example1),
   fopen(LISTING, 'example1_list', w),
   file_listing(LISTING, 'example1.pro').
```

The file "*example1_list*" now contains the listing of clauses defined in "*example1.pro*".

Listing directives

To allow the compilation of the files which are the result of the previous listing predicates one should also include the result of the following predicates which enumerates all the necessary directives.

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*.

all_directives/0

all_directives

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

8.3 Run-time declarations

Dynamic declaration

dynamic/1

dynamic _PredicateDescriptor

arg1 : ground : atom/integer

The predicate described by name/arity *arg1*, is declared dynamic. It is impossible to make existing non-dynamic predicates dynamic with this built-in.

Example

```
?- dynamic a/5 .
Yes
?- dynamic_functor(a/5) .
Yes
```

Optimizations

Mode declarations, declaring how a predicate will be called, can be defined for dynamic predicates. They allow the generation of more efficient code. They are only checked if the program is executed in debug mode.

Dynamic predicates can be indexed on one of their arguments. The default for it is to be indexed on the first argument, regardless of how the dynamic predicate is created (defined in a file or asserted interactively). Optionally, it is possible to specify hashing on top of the basic indexing.

More information regarding mode declarations and indexing can be found in "*Compiler Directives*" and in "*Principal Components - The Compiler*".

Index and mode declarations for dynamic predicates can be specified in files with the same directives as those used for static predicates. The following built-in predicates may be used to define them interactively when the predicate does not have any definitions yet.

It is impossible to change the indexing or mode declarations of a predicate which already has definitions.

mode/1

mode _Term

arg1 : ground : term

Sets the modes for the dynamic predicate defined by the principal functor of *arg1*. The modes are set as indicated in the arguments of term *arg1* (see also "*Compiler Directives*").

index/2

_Name /_Arity index _IndexSpecification

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

The dynamic predicate described by *arg1* (in the form *_Name/_Arity*), is indexed on the argument specified in *arg2*. If *arg2* has the form *_Argnr/_Size*, additional hashing is done for the indexed argument. *_Size* defines the size of the hash table. When the value of *arg2* is 0, there will be no indexing for the predicate.

An indexed dynamic predicate can be rehashed with the following built-in predicate.

rehash/2

rehash(_Name/_Arity, _TableSize)

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

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

Hiding predicates

hide/1

hide(_Term)

arg1 : partial : term

The predicate defined by the principal functor of *arg1* is made hidden. This means that the definitions of the predicate are not visible to the user. They cannot be inspected with the **listing**, **clause** and **retract** predicates.

The predicate does not have any effect when *arg1* has not been declared as a predicate yet.

Once hidden, a predicate cannot be made visible any more. However, after complete removal of all definitions and declarations (e.g. using **abolish/1,2**), a new predicate with the same principal functor will be visible again.

The predicate always succeeds.

Operators

Operators can be defined at runtime with the following built-in predicate.

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 []-terminated 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.

A conflicting operator declaration overrides the previous declaration.

It is possible to override built-in operators, or to have two operators with the same name, provided that one is an infix operator (xfy, yfx, xfx) and the other a prefix operator (fx, fy, yf, xf).

To declare an operator in a file, one must include the directive **op/3** in the file. It is not sufficient to declare the operator interactively. (see *"Principal Components - The Compiler"* and *"Compiler Directives"*).

The list of predefined operators can be found in *"Syntax"*.

8.4 Examining the database

Existence

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 do not necessarily have a definition. If only a **dynamic/1, mode/1** or **index/2** declaration was issued for a dynamic predicate, it will exist as predicate, but without a definition.

Example

```
?- has_a_definition(halt).
Yes
```

Functor type

static_functor/1

static_functor(_Term)

arg1 : partial : term

Succeeds if *arg1* is declared as a static predicate.

dynamic_functor/1

dynamic_functor(_Term)

arg1 : partial : term

Succeeds if *arg1* is declared as a dynamic predicate or is compiled into debug code.

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 declared as an external predicate.

hidden_functor/1

hidden_functor(_Term)

arg1 : partial : term

Succeeds if *arg1* is declared as a hidden predicate.

builtin/1

builtin(_Term)

arg1 : partial : term

Succeeds if *arg1* is a built-in

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
built-in predicate	built-in
static predicate	static
dynamic or debug coded predicate	dynamic
database predicate	database
external predicate	external

predicate_info/3

predicate_info(_Term,_Type,_Info)

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

Succeeds if the principal functor of *arg1* is a functor of a predicate with type *arg2*. Depending on the predicate type, *arg3* is instantiated to a list containing information about the predicate.

Type	Info
built-in	[]
static	[]
dynamic	[_NrClauses,_IdxArg,_HtabSize]
database	[]
external	[]

With

_NrClauses	Number of active clauses
_IdxArg	Number of the indexed argument (0 if none)
_HtabSize	Size of the hash table (0 for none)

Existence or Enumeration

The following predicates enumerate a number of entities which are present in the ***ProLog by BIM*** system. Lots of the results you will get are internally defined by the system (e.g. functor names of built-in predicates) and will not be of great interest to the user. The order in which the results are returned is dependent on a number of factors and should not be relied upon.

current_atom/1

current_atom(_Atom)

arg1 : any : atom

Returns all atoms currently known in the ***ProLog by BIM*** system one at a time by backtracking, or succeeds once, if *arg1* is instantiated to an atom.

Example

```
?- please(wq,on),current_atom(_A).
 _A = '\n'
Yes ;
 _A = ' '
Yes ;
 _A = 'Goal'
Yes ;
 _A = 'B'
Yes
```

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

Unifies arg3 with the operators currently known in the ***ProLog by BIM*** system, *arg1* with their precedence and *arg2* with their type, one at a time by backtracking.

Example

```
?- please(c,off).
Yes
?- current_op(_P,_A,_O).
 _O = ':-'
 _A = xfx
 _P = 1200
Yes ;
 _O = '-->'
 _A = xfx
 _P = 1200
Yes

?- please(c,on).
Yes
?- current_op(_P,_A,(:-)).
 _A = xfx
 _P = 1200
Yes ;
 _A = fx
 _P = 1200
Yes
```

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 of all existing predicates, one at a time by backtracking.

Example

```
?- assert( (g(_1,_2) :- g(_1)) ).
Yes
?- current_predicate(g,_pred).
 _pred = g( _30, _31 )
Yes
```

current_functor/2

current_functor(_FuncName, _FuncTerm)

arg1 : any : atom
arg2 : any : term

Unifies *arg1* with the name of the functors currently known in the ***ProLog by BIM*** system, and *arg2* with the most general term corresponding to it, one at a time by backtracking.

Example

```
?- assert( (g(_1,_2) :- g(_1)) ).
Yes
?- current_functor(g, _functor).
 _functor = g( _30, _31 )
Yes ;
 _functor = g( _30 )
Yes ;
No
```

all_functors/1

all_functors(_FuncTerm)

arg1 : any : term

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

8.5 Managing cyclic terms

ProLog by BIM offers support for working with cyclic terms. Predicates exist to avoid creating cyclic terms (**occur/2, occurs/2**), or for testing whether such a cyclic term was created (**is_inf/1**). Moreover, cyclic terms can be recorded and returned in the answer list of a call to **findall/3** and its friends (see also "*User's Guide - Cyclic Terms*").

It can be useful to assert a clause which contains a cyclic term or to reduce a cyclic term to a canonical and finite form, e.g. to be used for printing. Therefore the following built-in predicates have been made available to the user:

canonical/3, which, given a (cyclic) term *_Tin* produces a non-cyclic term *_Tout* and a list of unifications *_Lun* to perform in order to construct from *_Tout* the given term *_Tin*.

canonical_clause/2 which given a (cyclic) clause *_Tin* produces a non-cyclic clause *_Tout* which can be asserted and which, when executed, behaves like *_Tin*.

Since cyclic terms are a special case of terms with sharing subterms, also two more general predicates have been implemented and made available:

cse/3 acts like canonical/3 except that *_Tout* will not contain any sharing subterms except for variables and atomic terms; and similarly,

cse_clause/2 is analogous to canonical_clause/2.

canonical/3

canonical(_Tin,_Tout,_Lun)

arg1 : any : term
arg2 : any : term
arg3 : any : list of terms

The predicate is meant to be called with *_Tout* and *_Lun* distinct free variables; its semantics is not defined otherwise, but no warning or error is raised.

When a call to canonical/3 succeeds, then *_Tout* is a non-cyclic term and *_Lun* a list of unifications of the form Var = Term with every Term non-cyclic and such that if all unifications of *_Lun* are performed, *_Tout* is identical to *_Tin*. *_Tin* can be a cyclic term. Moreover, the list *_Lun* is minimal and its arguments share no subterms. The minimality of *_Lun* means in particular that if *_Tin* is non-cyclic and contains no sharing subterms, *_Lun* will be the empty list [] and *_Tout* will be identical to *_Tin*.

Example

a non-cyclic term with no internal sharing:

```
?- _Tin = g(f(1),f(1)), canonical(_Tin,_Tout,_Lun).
 _Lun = []
 _Tout = g(f(1),f(1))
 _Tin = g(f(1),f(1))
```

a non-cyclic term with internal sharing:

```
?- _A = f(1), _Tin = g(_A,_A), canonical(_Tin,_Tout,Lun).
 _Lun = [_69 = f(1)]
 _Tout = g(_69,_69)
 _Tin = g(f(1),f(1))
 _A = f(1)
```

a cyclic term:

```
?- _Tin = f(_Tin), canonical(_Tin,_Tout,_Lun) .
 _Lun = [_Tout = f(_Tout)]
 _Tin = f(f(f(f(...))))
```

cse/3

cse(_Tin,_Tout,_Lun)

arg1 : any : term
arg2 : any : term
arg3 : any : list of terms

This predicate is a variation of **canonical/3**. It expects the same sort of arguments and the relation between the arguments is the same. The difference is that also the common identical subterms of the terms in the list *_Lun* are replaced by appropriate unifications. This becomes apparent in the following example (compare with the first example for **canonical/3**)

Example

a non-cyclic term with no internal sharing but with a common identical subterm:

```
?- _A = f(1), _Tin = g(_A,_A), cse(_Tin,_Tout,_Lun).
 _Lun = [_69 = f(1)]
 _Tout = g(_69,_69)
 _Tin = g(f(1),f(1))
 _A = f(1)
```

a non-cyclic term with internal sharing:

```
?- _A = f(1), _Tin = g(_A,_A), cse(_Tin,_Tout,_Lun).
 _Lun = [_69 = f(1)]
 _Tout = g(_69,_69)
 _Tin = g(f(1),f(1))
 _A = f(1)
```

a cyclic term:

```
?- _Tin = f(_Tin), cse(_Tin,_Tout,_Lun).
 _Lun = [_Tout = f(_Tout)]
 _Tin = f(f(f(f(...))))
```

canonical_clause/2

canonical_clause(_Clin,_Clout)

arg1 : partial : term
arg2 :free : term

The predicate is meant to be called with *arg2* a free variable; its semantics is not defined otherwise. *Arg2* is constructed from (a possibly cyclic) *arg1*, which is supposed to be a clause (no checking is done whether it is a valid clause). *Arg2* is a non-cyclic clause where all internal sharing is replaced by unifications (compare to *_Lun* in **canonical/3**).

Example

a non-cyclic clause with no internal sharing but with a common identical subterm:

```
?- _Clin = (head(f(_X)) :- goal1, goal(f(_X))),
   canonical_clause(_Clin, _Clout).
 _Clout = head(f(_X)) :- goal1, goal(f(_X))
 _Clin = head(f(_X)) :- goal1, goal(f(_X))
```

a non-cyclic clause with internal sharing:

```
?- _A = f(_X) ,
   _Clin = (head(_A) :- goal1, goal(_A)),
     canonical_clause(_Clin,_Clout).
 _Clout = head(_114) :- _114 = f(_X), goal1, goal(_114)
 _Clin = head(f(_X)) :- goal1, goal(f(_X))
 _A = f(_X)
```

a cyclic clause:

```
?- _X = f(_X), _Clin = (head(_X) :- goal1, goal(_X)),
   canonical_clause(_Clin, _Clout) .
 _Clout = head(_109) :- _109 = f(_109), goal1, goal(_109)
 _Clin = head(f(f(f(...)))) :- goal1, goal(f(f(...)))
 _X = f(f(f(f(f(...)))))
```

a clause with a cyclic body:

```
?- _X = (write(ok), _X),
     _Clin = (t :- X),
     canonical_clause(_Clin, _Clout).
 _Clout = t :- _82 = (write(ok) , _82) , _82
 _Clin = t :- write(ok), write(ok), write(...), ... , ...
 _X = write(ok), write(ok), write(ok), write(...), ... , ...
```

cse_clause/2

cse_clause(_Clin,_Clout)

arg1 : partial : term
arg2 : any : term

This predicate is a variation of **canonical_clause/2.** It expects the same sort of arguments and the relation between the arguments is the same. The difference is that also the common identical subterms of *arg2* are replaced by appropriate unifications. This becomes apparent in the first example (compare with the first example for canonical_clause/2):

Example

a non-cyclic clause with no internal sharing but with a common identical subterm:

```
?- _Clin = (head(f(_X)) :- goal1, goal(f(_X))) ,
   cse_clause(_Clin, _Clout) .
 _Clout = head(_102) :- _102 = f(X), goal1, goal(_102)
 _Clin = head(f(_X)) :- goal1, goal(f(_X))
```

a non-cyclic clause with internal sharing:

```
?- _A = f(_X),
   _Clin = (head(_A) :- goal1, goal(_A)),
   cse_clause(_Clin, _Clout) .
 _Clout = head(_114) :- _114 = f(X), goal1, goal(_114)
 _Clin = head(f(_X)) :- goal1 , goal(f(_X))
 _A = f(_X)
```

a cyclic clause:

```
?- _X = f(_X) ,
   _Clin = (head(_X) :- goal1, goal(_X)) ,
   cse_clause(_Clin, _Clout) .
 _Clout = head(_109) :- _109 = f(_109), goal1, goal(_109)
 _Clin = head(f(f(f(...)))) :- goal1, goal(f(f(...)))
 _X = f(f(f(f(f(...)))))
```

Built-in Predicates

Chapter 9
Program Loading

9.1 Program manipulation

Loading a program implies loading all Prolog files into the engine. Each file must be compiled to intermediary object code (which is stored in an object file) before it can be loaded. This compilation phase can be invoked explicitly by the developer or can be done implicitly by the engine.

The predicates that are defined in the loaded files will be added to the internal code tables. Code for dynamic predicates is added to the Interpreted Code table. Static predicates are further compiled into native code which is then added to the Compiled Code table (see also "*Principal Components - The Engine*").

There exist built-in predicates to invoke the compiler and the loader separately. The consult built-ins integrate the two steps by issuing a compile-and-load.

9.2 General concepts

This section describes general rules regarding file compilation and loading.

Compilation rules

The options specified in the compile goal determine the options to be used by the compiler. Three sets of options are considered:

- compiled: options with which the source was previously compiled (see **compiled/2**)
- engine: current engine compiler switch settings (see **compiler/2**)
- specified: explicitly mentioned options in the load goal

The options specified in the compile goal and the compiler options they imply are:

Goal	Compiler options
<none>	compiled option set
-	engine option set
-x -y ...	compiled option set, overridden by specified option set -x -y ...
- -x -y ...	engine option set, overridden by specified option set -x -y ...

In case there is no corresponding intermediate code file the compiled option set is assumed the same as the engine option set. The engine option set is originally the same as the compiler default settings, as determined at system installation time. It can be modified at invocation of the engine with the -C command option and during execution with the **compiler/2** built-in.

compiler/2

compiler(_SwitchName, _Value)

arg1 : ground : atom
arg2 : free or ground : atomic

Sets or retrieves the compiler switch with name *arg1*. If *arg2* is free, it is instantiated to the current value of the indicated switch. If *arg2* is ground, the switch is set to *arg2*. These settings are considered the engine compiler switch settings.
If one or both arguments do not have the required value the predicate will write out the current settings and fail.

Example

```
?- compiler(c,_X).
 _X = on
Yes
?- compiler(_,_).
Available switches in compiler/2 :
    a : alldynamic       (Toggle) off
    c : compatibility    (Toggle) off
    d : debugcode        (Toggle) off
    e : eval             (Toggle) on
    h : hide             (Toggle) off
    l : listing          (Toggle) off
    p : operators        (Toggle) off
    w : warn             (Toggle) on
    x : atomescape       (Toggle) on
    A : Alltablesize     (Number) 1
No
```

(Re)compilation

Source files are (re)compiled if at least one of the following conditions is met:

- The intermediate code file (<file>.wic) does not exist.
- The source file (<file>.pro or <file>) is more recent than the corresponding intermediate code file (<file>.wic).
- The indicated compiler options do not match the options with which the source file was previously compiled.

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

More information on compiler options can be found in "*Principal Components - The Compiler*".

Loading behavior

The loading of a predicate is performed in one of two modes: append or overwrite. For append mode, there are two alternatives.

append mode:
New clauses for the predicate are added at the end of the chain.

append* mode:
All clauses known in the engine as being loaded from the specified file, are removed. If this would lead to a dynamic predicate having all its definitions removed, its declaration will also be removed. The new clauses are then added at the end of the chain as in append mode.

overwrite mode:
All existing clauses and declarations for the predicates defined in the file, are first removed, (as with abolish/1,2) before the new clauses are loaded into the engine.

Which of the modes is used, depends on the predicate used to load.

Built-in	Mode
load/1	append
consult/1,2	append
reload_file/1	append*
reconsult_file/1,2	append*
reload_predicates/1	overwrite
reconsult_predicates/1,2	overwrite

Predicate conflicts

For a load in append mode, the predicate should not have been defined before, except if it was a dynamic predicate. Trying to add clauses to a static predicate, will result in a warning message and the clauses will not be loaded. If the load of a file can lead to a double declaration of a predicate (trying to load an existing predicate but with a different type), the loading of the clauses for this predicate is terminated and a warning is displayed.

A load in overwrite mode allows replacement of static, dynamic and external predicates with all possible types. Database and built-in predicates cannot be overwritten.

Operators

If the loaded file is compiled with the -p option, the current active operators in the engine are cleared and replaced by the operators active during compilation.

9.3 Built-ins for program loading

This section describes the built-ins available for loading files into the engine. It consists of built-ins for compiling, for loading and for consulting files. But first, a summary is given of how files and compiler options can be specified in the arguments of these predicates.

File specification

A file specification argument can have two forms:

- a single atom, giving the full file access path, or
- a structure of the form Path/Name.

The file name and the access path are expanded to find the location of the file (see "*Principal Components - The Engine*" for more information about file name expansion).

The name of the source file is used in the compile and consult built-ins. It is constructed as follows:

Specification	source file name
file.pro	file.pro
file	file if it exists, otherwise file.pro
file.wic	file.pro
file.suff	file.suff if it exists, otherwise file.suff.pro

The name of the object file is used in the load built-ins. It is constructed as follows:

Specification	object file name
file.pro	file.wic
file	file.wic
file.wic	file.wic
file.suff	file.suff.wic

Compiler option specification

Predicates that include the compile phase, can have compiler options. These are given:

- in the single atom that indicates the file access path, preceding that path.
- as a separate second argument, consisting of an atom that includes all required options.

Compile predicates

compile/1

compile(_FileName)

arg1 : ground : atom or atom/atom

File arg1 is compiled if necessary. Compiler options may be included in *arg1*.

compile/2

compile(_FileName,_Options)

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

File *arg1* is compiled, with compiler options *arg2*, if necessary.

compiled/2

compiled(_FileName,_Options)

arg1 : ground : atom or atom/atom
arg2 : free : atom

Succeeds if a compiled version of file *arg1* (that can be loaded into the engine) exists. The option set with which it was compiled is returned in *arg2* as an atom. That atom is in a form suitable for use in **compile/2** or in consult predicates. The current engine options are not considered. Instead, all existing compiler options are enumerated, with their respective value.

Load predicates

The load predicates search for the specified intermediary object file and try to load it. If the file cannot be loaded, the load predicates give an error message and succeed.

The load predicates do not perform a recompilation. If compiler options are specified, they are discarded.

load/1

load(_FileName)

arg1 : ground : atom or atom/atom

File *arg1* is loaded in append mode. All clauses in file arg1 are added.

reload_predicates/1

reload_predicates(_FileName)

arg1 : ground : atom or atom/atom

File *arg1* is loaded in overwrite mode. All clauses of predicates that are defined in file *arg1* are first deleted. The new clauses from file *arg1* are then added.

reload_file/1

reload_file(_FileName)

arg1 : ground : atom or atom/atom

File *arg1* is loaded in append* mode. All clauses that were defined in file *arg1* are deleted, as would be the case for **unload/1**. The new clauses from file *arg1* are then added.

reload/1

reload(_FileName)

arg1 : ground : atom or atom/atom

If the **reloadall** (la) please switch is on, this is the same as **reload_predicates/1**, otherwise it is the same as **reload_file/1.**

file_loaded/1

file_loaded(_FileName)

arg1 : ground : atom or atom/atom

Succeeds if file *arg1* has already been loaded.

ensure_loaded/1

ensure_loaded(_FileName)

arg1 : ground : atom or atom/atom

Checks whether file *arg1* has been loaded. If not, it is loaded as with **load/1**.

unload/1

unload(_FileName)

arg1 : ground : atom or atom/atom

All clauses that were defined in file *arg1* are deleted. If this would lead to a dynamic predicate having all its definitions retracted, its declarations will also be removed.

9.4 Consult predicates

consult/1

consult(_FileName)

arg1 : ground : atom or atom/atom

File *arg1* is compiled if necessary, and then loaded in append mode. Compiler options may be included in *arg1*. All clauses in file *arg1* are added.

Example

```
?- consult('-a+ ~prolog/file1').
compiling -a+ /bim53/prolog/file1.pro
loaded file1.pro
Yes
```

consult/1 can also be invoked by using the short hand notation:

```
[_FileName] or [_Path/_FileName]
```

Example

```
?- ['-a+ ~prolog/file1'].
compiled -a+ /bim53/prolog/file1.pro
loaded file1.pro
Yes
```

consult/2

consult(_FileName,_Options)

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

File *arg1* is compiled, with compiler options *arg2*, if necessary, and then loaded in append mode. All clauses in file *arg1* are added.

Example

```
?- consult('~prolog/file2', '-a+ -d-').
compiling -a+ -d- /bim53/prolog/file2.pro
loaded file2.pro
Yes
```

reconsult_predicates/1

reconsult_predicates(_FileName)

arg1 : ground : atom or atom/atom

File *arg1* is compiled if necessary, and then loaded in overwrite mode. Compiler options may be included in *arg1*. All clauses of predicates that are defined in file *arg1* are first deleted. The new clauses from file *arg1* are then added.

reconsult_predicates/2

reconsult_predicates(_FileName,_Options)

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

File *arg1* is compiled, with compiler options *arg2*, if necessary, and then loaded in overwrite mode. All clauses of predicates that are defined in file *arg1* are first deleted. The new clauses from file *arg1* are then added.

Example

file1.pro contains:

```
a(1).
b(2).
c(3).
```

```
?-consult('-a+ ~prolog/file1').
compiling -a+ /bim53/prolog/file1.pro
loaded file1.pro
Yes
```

assume file1.pro has been edited and now contains:

```
a(one).
b(two).
d(four).
```

```
?- reconsult_predicates('-a+ ~prolog/file1').
compiling -a+ /bim53/prolog/file1.pro
reloaded file1.pro
Yes
?- listing.
a(one).
b(two).
c(3).
d(four).
Yes
```

reconsult_file/1

reconsult_file(_FileName)

arg1 : ground : atom or atom/atom

File *arg1* is compiled if necessary, and then loaded in append* mode. Compiler options may be included in *arg1*.

All clauses that were defined in file *arg1* are deleted, as would be the case for **unconsult/1**. The new clauses from file *arg1* are then added.

Example

file1.pro contains:

```
a(1).
b(2).
c(3).
```

```
?- consult('-a+ ~prolog/file1').
compiling -a+ /bim53/prolog/file1.pro
loaded file1.pro
Yes
```

assume file1.pro has been edited and now contains:

```
a(one).
b(two).
d(four).
?- reconsult_file('-a+ ~prolog/file1').
compiling -a+ /bim53/prolog/file1.pro
reloaded file1.pro
Yes
?- listing.
a(one).
b(two).
d(four).
Yes
```

reconsult_file/2

reconsult_file(_FileName,_Options)

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

File *arg1* is compiled, with compiler options *arg2*, if necessary, and then loaded in append* mode. All clauses that were defined in file *arg1* are deleted, as would be the case for **unconsult/1**. The new clauses from file *arg1* are then added.

reconsult/1

reconsult(_FileName)

arg1 : ground : atom or atom/atom

If the **reloadall** (la) please switch is on, this is the same as **reconsult_predicates/1**, otherwise it is the same as **reconsult_file/1**.

reconsult/1 can also be invoked by using the short hand notation:

```
[- _FileName] or [- _Path/_FileName]
```

Example

```
?- [-'-a+ ~prolog/file1'].
compiled -a+ /bim53/prolog/file1.pro
reloaded file1.pro
Yes
```

reconsult/2

reconsult(_FileName,_Options)

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

If the **reloadall** (la) please switch is on, this is the same as **reconsult_predicates/2**, otherwise it is the same as **reconsult_file/2**.

ensure_consulted/1

ensure_consulted(_FileName)

arg1 : ground : atom or atom/atom

Checks whether file *arg1* has been loaded. If not, it is consulted as with **consult/1**.

ensure_consulted/2

ensure_consulted(_FileName,_Options)

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

Checks whether file *arg1* has been loaded. If not, it is consulted as with **consult/2**.

9.5 File predicate association

Retracting clauses on a file by file basis

retractfile/2

retractfile(_Term, _File)

arg1 : partial : term
arg2 : ground : atom/atom

All clauses for the predicate defined by the principal functor of *arg1*, and defined in file *arg2*, are retracted. The file can be specified as Path/Name or as a single path. Declarations for the predicate are not removed.

Example

```
?- file_listing('example_a.pro').
a1( _44 ) :-
   a2 .
a2 .
Yes

?- retractfile(a1(_), 'example_a.pro').
Yes
?- file_listing('example_a.pro').
a2 .
Yes
```

Inquiring association between predicate and files

multi_file/1

multi_file(_Term)

arg1 : partial : term

Succeeds if the predicate defined by the principal functor of *arg1* is defined in more than one file. Interactively asserted definitions are treated as if they were defined in a single file.

This predicate fails if *arg1* has either no definitions at all, or if it has all of its definitions in the same file.

Example

```
?- file_listing('example.pro').

example_clause( _44 ) :-
   write(first) .
Yes
?- dynamic_functor(example_clause(_)).
Yes
?- multi_file(example_clause(_)).
No
```

Asserting a new fact gives:

```
?- assertz( ( example_clause(_) :- write(second) ) ).
Yes
?- listing.

example_clause( _29 ) :-
   write(first) .
```

```
example_clause( _29 ) :-
   write(second) .
Yes
?- multi_file(example_clause(_)).
Yes
```

predicate_files/2

predicate_files(_Term, _FileList)

arg1 : any : term
arg2 : any : list of atom/atom or atom

Returns in *arg2* the list of files that contain clauses for the predicate defined by the principal functor of *arg1*. The files are specified as Path/Name, or as "user" if the predicate has interactively asserted clauses.

Only real definitions have an associated source file. Pseudo definitions like **dynamic/1, mode/1** and **index/2,** are not considered.

Example

```
?- file_listing('example.pro').

example_clause( _44 ) :-
   write(first) .
Yes
?- dynamic_functor(example_clause(_)).
Yes
?- assertz( ( example_clause(_):- write(second) ) ).
Yes
?- predicate_files(example_clause(_4), _f1).
 _f1 = [user,/usr/prolog/MAN_EXAMPLES/ example.pro]
Yes
```

file_predicates/2

file_predicates(_File, _PredicateList)

arg1 : ground : atom/atom or atom
arg2 : free : list of atom/integer

Returns in *arg2* the list of predicates for which there are clauses in file *arg1*. This list contains elements of the form Name/Arity. The file specification *arg1* must be of the form Path/Name or a single atom. Specifying "user" as *arg1* will return in *arg2* the list of predicates which have asserted clauses.

Only real definitions in the source file are taken into account. Pseudo definitions like **dynamic/1, mode/1** and **index/2,** are not considered.

Example

```
?- file_listing('example.pro').

example_clause( _44 ) :-
     write(first) .
Yes
?- file_predicates('example.pro', _1).
     _1 = [example_clause(_35)]
Yes
```

current_file/1

current_file(_File)

arg1 : any : atom or atom/atom

If *arg1* is instantiated, the predicate succeeds if arg1 is loaded in the system. Otherwise, it fails. The file can be specified as Path/Name or as a single path.
If *arg1* is free, it is instantiated to all loaded files, one by one on backtracking, and in the form Path/Name.

9.6 Loading libraries

The library of ***ProLog by BIM*** consists of the *Prolog Source Library* and the *External Packages Library.*

- The *Prolog Source Library* contains a large set of predicates which can be loaded into ***ProLog by BIM***. Since the Prolog source files, C files and LEX files are provided in source format, they can be adapted according to the needs of the developer.
- The *External Packages Library* is composed of compiled interface files to windowing and database packages.

Source files:

The source files of the ***Prolog Source Library*** are in the directory $BIM_PROLOG_DIR/src.

Compilation:

Before using the *Prolog Source Library*, it is necessary to compile the source files. This is done when installing ***ProLog by BIM***, or manually, by running the makefiles included in each subdirectory of the library. Please refer to the "***ProLog by BIM Installation Guide***" for further information about library installation.

Compiled files:

The compiled files of the *Prolog Source Library* and the *External Packages Library* can be found under the directory $BIM_PROLOG_DIR/lib.

Documentation:

The following parts of the manual describe the library of ***ProLog by BIM***:

External Packages Library:

- Windowing Interfaces.
- Database Interfaces.

Prolog Source Library:

- Prolog Utilities.
- UNIX Utilities.
- Tools.

Note:
The *Prolog source library* contains a large set of predicates. Most of the predicates are provided with the source code. They are mostly adapted from the public domain. The author names (R.A. O'Keefe, K. Johnson, D. Bowen, D.H.D. Warren, F. Pereira, L. Byrd, L. Peirera & A. Porto, M. Gehrs and many others) can be found in the library source files. *Neither BIM nor any of the authors are responsible for any of the provided programs.*

The *Prolog Source Library* and the *External Packages Library* can be loaded into ***ProLog by BIM*** by using the consult predicates.

Example

```
?- consult('-Lprolog/lists').
```

This command loads the compiled file of the lists library stored in the directory $BIM_PROLOG_DIR/lib/prolog.

This method requires the exact location of the compiled library files the user wants to load.

An alternative is to refer to the libraries with a symbolic name. ***ProLog by BIM*** maintains a database containing the associations "Symbolic name - Library location path". This database can be accessed through the use of the following predicates:

lib/1

lib(_Symbolic_Library_Name)

arg1 : ground : atom

If the library with symbolic name *arg1* exists, it is loaded with **ensure_loaded/1**.

If lib/1 is called with a symbolic name that is not defined in the association database, the symbolic name is considered as a file name that is searched for in the $BIM_PROLOG_DIR/lib directory: **ensure_loaded/1** is called with as argument the symbolic name prefixed by "-L".

Example

```
?- lib('lists').
```

This command loads the compiled file of the library identified by the symbolic name 'lists'.

```
?- lib('anylib') .
```

Tries to consult the file **"-Lanylib"** since there is no **anylib** symbol in the association database.

lib_mapping/2

lib_mapping(_Symbolic_Library_Name, _Library_Location_Path)

arg1 : any : atom
arg2 : any : atom

Retrieves the current library mappings.

If *arg1* is instantiated, the mapping for that library is returned. When there is a system and user-defined mapping, both values are returned by backtracking.

If *arg2* is instantiated, all libraries that are mapped to that file are returned, one at a time by backtracking.

If both arguments are free, the whole mapping database is returned, one by one at a time by backtracking.

The last case of two ground arguments checks whether the given library is mapped to the indicated path.

lib_map/2

lib_map(_Symbolic_Library_Name, _Library_Location_Path)

arg1 : ground : atom
arg2 : ground : atom

Stores the mapping of library with name *arg1*, to file with path *arg2*. Any existing mapping of this library is first erased. Any available shortcut symbol ($VAR, -L, -H, ~, ...) can be used in the path. See also "*Principal Components - The Engine - File name expansion*" for the list of symbols recognized in a file access path.

Example

```
?- lib_map(listlib,'~/my_prolog_lib/listlib' ).
Yes
```

Example

The following query loads the 'listut' library, if it is not yet loaded. By default, it will be loaded from the file -Hlib/prolog/listut, because this is recorded in the library mapping database as shown by the **lib_mapping/2** query.

```
?- lib( listut ) .
Yes
?- lib_mapping( listut , _Path ) .
_Path = $BIM_PROLOG_DIR/lib/prolog/listut
Yes
```

To override the 'listut' library with a user-defined version, the new version must have the same name, but may be located in another directory (for example: ~/my_lib/). This new location is entered with the query:

```
?- lib_map( listut , '~/my_lib/listut' ) .
```

The following query will then consult ~/my_lib/listut:

```
?- lib( listut ).
```

lib_reset/0

lib_reset

Resets all system default mappings in the library association database.

lib_reset/1

lib_reset(_Library)

arg1 : ground : atom

Resets the system default mapping for library *arg1* in the library association database. If the indicated library is not part of the ***ProLog by BIM*** library, it is simply removed from the association database.

lib_listing/0

lib_listing

Lists the current library association database with mappings of symbolic names to location paths.

Built-in Predicates

Chapter 10
Record Database

10.1 Introduction to the record database

The **record** predicates are most suitable for simulating global data. The predicates for managing the internal database exist in a version with two keys and 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).

For all the record database predicates holds that if the key argument is a structured term, the principal functor of this term (functor name/arity) will serve as the key.

The data that is recorded must be at least partially instantiated. A free variable is not allowed as data to be recorded.

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**.

With the predicates **record_init_queue/2, record_enqueue/3** and **record_dequeue/3, global queues** can be simulated.

The keys of a record are stored in the record table (R). If the associated term is structured it will be stored in the backup heap (B). More information on these tables can be found in "*Principal Components - The Engine - Table manipulation*".

10.2 Global values

record/3

record(_Key, _DomainKey, _Term)

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

If there is a term in the internal database associated with *arg1* and *arg2*, then this call fails and an error message is displayed. 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 : partial : 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 : partial : 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 : partial : 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
arg3 : 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.

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 information associated with the key pair *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

erase_all'

This predicate is defined as:

```
erase_all :- erase_all(0).
```

is_a_key/2

is_a_key(_Term1, _Term2)

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

This predicate is defined as:

```
is_a_key(_Term) :- is_a_key(_Term, 0).
```

current_key/2

current_key(_Key, _DomainKey)

arg1 : any : term
arg2 : any : 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. One should not rely on the order in which the solutions are returned (this depends on a lot of factors such as table size, hash function, previous queries,..).

current_key/1

current_key(_Key)

arg1 : any : term

This predicate is defined as:

```
current_key(_Key) :- current_key(_Key, 0).
```

10.3 Global stacks

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

record_push/3

record_push(_Key, _DomainKey, _Term)

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

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 : partial : term

This predicate is defined as:

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

record_pop/3

record_pop(_Key, _DomainKey, _ListHead)

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

The list associated with the keys *arg1* and *arg2* is split into head and tail. The head is unified with *arg3*. The tail replaces the original list as associated value for the keys.

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 : any : term

This predicate is defined as:

```
record_pop(_Key, _Term) :- record_pop(_Key, 0, _term).
```

10.4 Global arrays

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 : any : 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.

Example

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

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

recorded_arg/3

recorded_arg(_SelectList, _Key, _SelectArg)

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

Arg4 replaces the term, occurring 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 : partial : term

This predicate is defined as:

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

record_copy/6

record_copy(_FromKey, _FromDomain, _FromArg, _ToKey, _ToDomain, _ToArg)

arg1 : partial : term
arg2 : partial : term
arg3 : ground : list of integer
arg4 : partial : term
arg5 : partial : term
arg6 : ground : list of integer

The subterm of the term, recorded under keys *arg4*, *arg5*, at the position indicated by index list *arg6*, is replaced by the subterm of the term, recorded under keys *arg1*, *arg2*, at the position indicated by index list *arg3*. A position in a term is indicated by a list of indexes. The first index is the argument number of the principal term. The next is the argument number of that subterm, and so on.

Example

```
?- rerecord(a, b, f(11, 22, 33)) .
Yes
?- rerecord(c, d, g(x, y)) .
Yes
?- recorded(a, b, _x) .
 _x = f(11, 22, 33)
Yes
?- recorded(c, d, _x) .
 _x = g(x, y)
Yes
?- record_copy(a, b, [3], c, d, [2]) .
Yes
?- recorded(a ,b, _x) .
```

```
 _x = f(11, 22, 33)
Yes
?- recorded(c, d, _x) .
 _x = g(x, 33)
Yes
```

record_copy/4

record_copy(_FromKey, _FromArg, _ToKey, _ToArg)

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

Same as **record_copy/6** with as second keys the default value 0.

This predicate is defined as:

```
record_copy(_FromKey, _FromArg, _ToKey, _ToArg):-
     record_copy(_FromKey,0,_FromArg,_ToKey,0,_ToArg).
```

Example

```
?- rerecord(a, b, f(11, 22, 33)) .
Yes
?- rerecord(c, d, g(x, y)) .
Yes
?- recorded(a, b, _x) .
 _x = f(11, 22, 33)
Yes
?- recorded(c, d, _x) .
 _x = g(x, y)
Yes
?- record_copy(a, b, [3], c, d, [2]) .
Yes
?- recorded(a, b, _x) .
 _x = f(11, 22, 33)
Yes
?- recorded(c, d, _x) .
 _x = g(x, 33)
Yes
```

10.5 Queue data structure

A queue data structure is implemented on the record database. The operations on the queue are provided by three built-ins, to make a new queue, to enqueue and to dequeue a value. A queue is represented as a term of the form queue(_List,_Tail) where _List is an open-ended list containing all elements of the queue and with _Tail as its open end. A program should not rely on this representation, but rather use the queue as an abstract data type.

record_init_queue/2

record_init_queue(_Key, _DomainKey)

arg1 : partial : term
arg2 : partial : term

A new queue is constructed with keys *arg1*, *arg2*.

record_init_queue/1

record_init_queue(_Key)

arg1 : partial : term

This predicate is defined as:

```
record_init_queue(_Key) :- record_init_queue(_Key,0).
```

record_enqueue/3

record_enqueue(_Key ,_DomainKey, _Value)

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

The element *arg3* is enqueued on the queue identified by keys *arg1*, *arg2*.

record_enqueue/2

record_enqueue(_Key, _Value)

arg1 : partial : term
arg2 : partial : term

This predicate is defined as:

```
record_enqueue(_Key,_Value) :-
      record_enqueue(_Key,0,_Value).
```

record_dequeue/3

record_dequeue(_Key, _DomainKey, _Value)

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

One element is dequeued from the queue identified by keys *arg1*, *arg2* in *arg3*.

Example

```
?- record_init_queue (key1, key2).
Yes
?- record_enqueue (key1, key2, value).
Yes
?- record_enqueue (key1, key2, functor(_Var)).
 _Var = _10
Yes
?- record_dequeue (key1, key2, _V).
 _V = value
Yes
?- record_dequeue (key1, key2, _V).
 _V = functor(_38)
Yes
```

record_dequeue/2

record_dequeue(_Key, _Value)

arg1 : partial : term
arg2 : any : term

This predicate is defined as:

```
record_dequeue(_Key,_Value) :-
      record_dequeue(_Key,0,_Value).
```

Built-in Predicates

Chapter 11
Input - Output

11.1 Introduction

I/O predicates are used to read from or write to a file. The file can be indicated implicitly, using the "current" streams, or can be explicitly named, either using a logical file name or using a file pointer. Therefore, most I/O built-in predicates have two variants. The predicates that read from or write to explicitly indicated files have an arity that is 1 higher than the corresponding predicates that act on the current streams.

Current stream

There are three current streams: one for reading, one for writing and one for error messages. The three streams are initialized to user (acting as physical file name). Redirection is performed with see/1 (for input), and tell/1 (for output). ***ProLog by BIM*** writes its error messages to the current error output stream, which is initially associated with stderr. The current error output stream can be redirected with tell_err/1. The argument for these predicates is a physical file name.

Path names in physical file names can be given in an abbreviated format (for example ~, ~user, $ATOM,...). For more information see also "*The Engine - File name expansion*".

To close the current streams, use **seen/0**, **told/0** and **told_err/0**. If the current stream is **user**, these two predicates succeed, but do not close the streams.

To know the names of the current streams, use **seeing/1**, **telling/1** and **telling_err/1.**

Logical file name

A logical file name is an atom used inside the program to designate a "physical" 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 **fclose/1**.The logical files **stdin**, **stdout** and **stderr** are always open. They must not be closed.

File pointers

A file can also be referred to by its file pointer instead of a logical file name. File pointers can be obtained from an **fopen/3** call, from a call to **seeingptr/1**, **tellingptr/1** or **telling_errptr/1** or from the external routines. A file pointer can be shared between Prolog predicates and external routines. In this way, one can open files in an external routine and write/read this file from ***ProLog by BIM***, or external routines can write/read files opened in a Prolog predicate.

End-of-file

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, all calls to the read built-ins trying to read beyond the end-of-file fail. When this option is unset, the **read** built-ins return a special value, that is specified with another option of **please/2**. Built-ins that read characters return the *end-of-file character*, set with **readeofchar**, which defaults to the integer -1. Built-ins 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 by BIM***. No warning is given, although it can cause unexpected effects.
The maximum number of files that can be opened simultaneously is operating system dependent.

Related predicates

The library UnixFileSys also contains predicates which implement input/output functions. (see "*Prolog and Unix Libraries*").

11.2 Redirection of standard I/O streams

There are three current streams: one for reading, one for writing and one for error messages. The streams are initialized to **user** (the physical file name). Redirection is performed with **see/1** (for input), and **tell/1** (for output). ***ProLog by BIM*** writes its error messages to the current error output stream, which is initially associated with stderr. The current error output stream can be redirected with **tell_err/1**. The argument for these predicates is a physical file name.

Table warnings and system warnings are always written to stderr. These warnings can be turned off with the please switches *syswarn* and *tablewarn*. (see "*Built-in Predicates - Engine Manipulation*"). Error messages can be turned off with the please switch *warn* or (more selectively) with **error_message/2** (see "*Built-in Predicates - Error Handling*").

Standard input

see/1

see(_PhysFileName)
see(_FilePointer)

arg1 : ground : atom or pointer

Arg1 becomes the current input stream. The previous current input stream is not closed. Failure can occur if the maximum number of open files is reached, or if the file is protected.

If *arg1* is an atom, it must be a physical file name. If the file has already been opened as the current input stream, input will be read from the current file pointer position. If the indicated physical file has not yet been opened as current input stream, or has been closed again, it is opened for reading.

If *arg1* is a pointer, it is assumed to be the file pointer of an open file. Input will be read from that file.

seeing/1

seeing(_PhysFileName)
seeing(_FilePointer)

arg1 : any : atom or pointer

The current input stream is unified with *arg1*. If it was opened with a physical file name, this name is returned. If it was opened with a file pointer, *arg1* is unified with that file pointer. The name of the current input stream for **stdin** is **user**.

seeingptr/1

seeingptr(_FilePointer)

arg1 : any : pointer

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

seen/0

seen

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 output

tell/1

tell(_PhysFileName)
tell(_FilePointer)

arg1 : ground : atom or pointer

Arg1 becomes current output stream. 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.

If *arg1* is an atom, it must be a physical file name. 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.

told/0

told

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 file name, 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 **user**.

tellingptr/1

tellingptr(_FilePointer)

arg1 : any : pointer

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

Standard error

tell_err/1

tell_err(_PhysFileName)
tell_err(_FilePointer)

arg1 : ground : atom or pointer

All error messages from ***ProLog by BIM***, normally written on **stderr**, will be written on the file specified by *arg1*. ***ProLog by BIM*** may still write some (very urgent) error messages on **stderr**.
Its behavior is analogous to that of **tell/1**, but for the current error output stream.

told_err/0

told_err

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 or pointer

The current error output stream is unified with *arg1*. If it was opened with a physical file name, 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 **stderrt** is **user**.

telling_errptr/1

telling_errptr(_FilePointer)

arg1 : any : pointer

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

11.3 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*). The physical filename (*arg2*) is associated to the logical file name *arg1*. When free, *arg1* is 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*.

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 file name, the logical file mname 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 file name is given by the system.

No information is given concerning **stdin**, **stdout** and **stderr**.

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(_lst).
 _lst = [source.pro,SRC,w,data.pro,DATA,w,input,.6,r]
Yes
```

11.4 Reading

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 (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**). An error recovery is performed on syntax errors: 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.

Even if **read/2** fails because of the unification with arg2, a term will be read (and removed) from the input.

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 syntax error recovery is performed 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 by BIM*** will undo this effect). Therefore, when writing interactive programs, be careful to test for an eof.

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.

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 *arg2* as prompt. 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 *arg1* as prompt.

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*.

Reading terms from an atom

sread/2

sread(_Atom, _Term)

arg1 : ground : atom
arg2 : any : term

sread/2 reads term from the atom specified by *arg1*. The atom is parsed in the syntax which is in effect at the moment of the call (see "*Built-in Predicates - Engine Manipulation - Please/2*").

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)

Same 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)

Same 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

Same 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

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

Skipping characters

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

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

11.5 Writing

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 file name 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.

Example

The *ProLog by BIM* goal

```
?- 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

Same 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

Same as **write/2**, but atoms that need to be quoted are quoted, and spaces are inserted where appropriate.

Thus, terms written with **writeq/2** can be read with **read/1** or **read/2**. But:

- The term written does not end with a period.
- The same operator declarations that are active at the time of writing must be active when reading.
- If atoms containing control characters have been constructed by using **name/2** or **atomtolist/2**, an error message will be issued when rereading.

Example

The *ProLog by BIM* goal

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

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

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

(see also the example in **write/2**.)

writeq/1

writeq(_Term)

arg1 : any : term

Same 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

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

writem/1

writem(_Term)

arg1 : any : term

Same 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

Same 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

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

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.

Writing to atoms

swrite/2

swrite(_Atom, _Term)

arg1 : free : atom
arg2 : any : term

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

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

Same 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 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

The user can give more conversion specifications between the %-sign and the conversion character.

- *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

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.

Example

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

Invalid combinations might provoke crashes.

printf/2

printf(_Format, _Value)

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

Same 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**.

Writing out blank spaces

nl/1

nl(_LogFileName)
nl(_FilePointer)

arg1 : ground : atom or pointer

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

nl/0

nl

Same 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

Same 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

Same 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

Term *arg2* is written to file arg1 in a user-defined manner. If *arg2* is not a variable and if a matching definition of **portray/2** with same arguments exists, this is used to print the term. Otherwise **write/2** is used to print the principal functor of the term. These same rules are applied recursively for each argument of the term.

print/1

print(_Term)

arg1 : any

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

Example

```
> ?- print( 'a/1').
a/1
Yes
> portray( _x) :- writeq( _x).
> ?- print( 'a/1').
'a/1'
Yes
> ?- retractall( portray( _)).
Yes
> portray(_x):-printf('printed integer is %3d\n',_x).

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

vprint/3

vprint(_LogFileName,_Term,_NameVarList)

vprint(_FilePointer,_Term,_NameVarList)

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

Term *arg2* is written on file *arg1* in a user-defined manner. If *arg2* is not a variable and if a matching definition of **vportray/3** with same arguments exists, this is used to print the term. Otherwise **vwrite/3** is used to print the principal functor of the term. The same rules are applied recursively for each argument of the term.

vprint/2

vprint(_Term,_NameVarList)

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

Same as **vprint/3** but to the current output stream, and using **vportray/2** as user-defined print predicate or **vwrite/2** otherwise.

11.6 Clearing and file pointer positioning

Clearing the buffer

A general please option 'wf' exists for controlling the flushing behaviour of output operations (see "*Principal Components - The Engine - Please options*"). Besides that, one can use the following set of predicates for flushing specific streams.

flush/0

flush

The current output stream is flushed. This means that any unwritten data is written to the current output stream and that the corresponding buffer is cleared.

flush_err/0

flush_err

The current error output stream is flushed. This means that any unwritten data is written to the current error output stream and that the corresponding buffer is cleared.

flush/1

flush(_LogFileName)
flush(_FilePointer)

arg1 : ground : atom or pointer

The file, specified by *arg1*, is flushed. This means that any unwritten data is written to the file and that the corresponding buffer is cleared.

File pointer position

ftell/2

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

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

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 : internal

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

Note:
programs should not perform arithmetic on *_filePosition*.

Example

```
?- fseek('FILE1',0).
```

resets the file pointer to the beginning of the file.

11.7 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 beyond the last character of the file.

eof/0

eof

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

11.8 Miscellaneous predicates

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 used by read predicates is "@ ".

iprompt/1

iprompt(_PROLOG_for_AIX_Prompt)

arg1 : any : atom

If *arg1* is instantiated, the ***ProLog by BIM*** prompt is changed to *arg1*. If *arg1* is free, it is instantiated to the current ***ProLog by BIM*** prompt. The default prompt in assertmode, repectively querymode, is "> ", repectively "?-".

Built-in Predicates

Chapter 12
Interface to the Operating System

Besides the basic built-in predicates mentioned in this chapter, ***ProLog by BIM*** also offers a number of libraries which implement less straightforward functionalities (see "*Prolog and Unix Libraries*").

12.1 Interaction with the environment

UNIX system calls

system/2

system(_ReturnCode, _SystemCommand)

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

The command *arg2* is executed as a child process, without any shell in between (i.e. no expansion of environment variables, no search in $path for the executable). The return code of this execution is unified with *arg1*.

If *arg2* is a list, it is assumed to consist of the program name, followed by its arguments. As is the convention for calling executables, the program is called with as first argument its name followed by the given arguments.

When *arg2* is an atom, the program is invoked with as single argument, the name of the program.

Example

```
?- expand_path('$BIM_PROLOG_DIR/bin/BIMpcomp',_Path),
   system(_Ret,[_Path,file1]).
```

This activates the ***ProLog*** compiler to compile file1.

```
?- system(_Ret,'/bin/date').
```

Prints the date and time by executing /bin/date.

system/1

system(_SystemCommand)

arg1 : ground : atom or list of atom

Same as **system/2** but the return code is ignored.

shell/2

shell(_ReturnCode, _ShellCommand)

arg1 : any : integer
arg2 : ground : atom

The command *arg2* is executed in a shell as a child process. The return code of this execution is unified with *arg1*.

Which shell is used, is dependent on the OS environment in which the process is running.

If *arg2* is a list, it is assumed to consist of the program name, followed by its arguments. As is the convention for calling executables, the program is called with as first argument its name followed by the given arguments.

When *arg2* is an atom, the program is invoked with as single argument, the name of the program.

Example

```
?- shell(_r, [echo,'$BIM_PROLOG_DIR']).
Yes
```

Prints out the value of the shell variable BIM_PROLOG_DIR. This cannot be done with system/2 for two reasons. One is that 'echo' as such is unknown, unless the current directory is /bin. An absolute path should be used. The second reason is that the echo program requires to run in a shell.

shell/1

shell(_ShellCommand)

arg1 : ground : atom

Same as **shell/2** but the return code is ignored.

sh/0

sh

Invokes a Bourne shell.
Equivalent to **system**('/bin/sh').

csh/0

csh

Invokes a C shell.
Equivalent to **system**('/bin/csh').

Example

```
?- csh.
csh% date
Fri  Dec 31 23:59:59 MET 1999
csh% ^D
Yes
```

Environment variable manipulation

putenv/1

putenv(_VarNameValue)

arg1 : ground : atom

The environment variable setting *arg1* is stored. This setting must have the form "Name=Value".

Example

```
?- putenv('BIM_PROLOG_LIB=/usr/prolog/lib').
Yes
?- getenv('BIM_PROLOG_LIB', _Val).
 _Val = /usr/prolog/lib
Yes
```

setenv/2

setenv(_VarName, _VarValue)

arg1 : ground : atom
arg2 : ground : atom

The environment variable with name *arg1* is assigned the value *arg2*.

Example

```
?- setenv('BIM_PROLOG_DIR', '/usr/prolog/INSTALLATION').
Yes
?- getenv('BIM_PROLOG_DIR', _Val).
 _Val = /usr/prolog/INSTALLATION
Yes
```

getenv/2

getenv(_VarName, _Value)
arg1 : ground : atom
arg2 : any : atom

Arg2 is unified with the value of the environment variable with name *arg1*.

Example

```
?- getenv('HOME', _Val).
 _Val = /prolog/tests
Yes
```

Command level arguments

ProLog by BIM offers the possibility to specify and retrieve command level arguments. User-defined options are specified after the delimiter ("-" sign between blanks) and are passed literally as atoms (See also "*Principal Components - The Engine*").

Example

```
csh% BIMprolog - Useroption1 Useroption2
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993

?- argc(_NumberOfUseroptions).
  _NumberOfUseroptions = 2
Yes
?- argv(1, _ArgValue).
 _ArgValue = Useroption1
Yes
```

argc/1

argc(_Number)
arg1 : any : integer

Returns in *arg1* the number of user-defined command line arguments.

argv/1

argv(_ArgList)
arg1 : free : list of atoms

Returns in *arg1* the list of user-defined command line arguments. The arguments appear in the list in the same order as on the command line.

argv/2

argv(_Index, _ArgValue)
arg1 : ground : integer
arg2 : any : atom

Succeeds with *arg2* the *arg1*'th command line argument.

12.2 Path expansion

ProLog by BIM offers the following built-in predicates for file name expansion.

The following meta symbols are recognized.

~ , ~user
These symbols are expanded according the UNIX expansion rules. They may only appear at the beginning of the path.

$VAR
$VAR is replaced by the value of the environment variable. If the environment variable is not defined, it is replaced by the name of the variable. $VAR may appear anywhere in the path.

-Lfile
The file is searched for in the library directories.

-Hfile
The file is searched for in the home directory.

The last two meta symbols refer to directories which can be specified with environment variables.

More information on expansion rules can be found in "*Principal Components - The Engine*".

expand_path/2

expand_path(_Path, _ExpPath)

arg1 : ground : atom
arg2 : any : atom

The path *arg1* is expanded and unified with *arg2*.

absolute_path/2

absolute_path(_Path, _AbsolutePath)

arg1 : ground : atom
arg2 : any : atom

The path *arg1* is expanded to an absolute path and unified with *arg2*.

Expansion takes place in the same way as with **expand_path/2**. If the resulting path does not start with '/', it is further expanded by prefixing the current working directory to it.

Example

```
?- absolute_path('$BIM_PROLOG_DIR', _e).
 _e = /usr/prolog/INSTALLATION
Yes
?- absolute_path('~prolog', _e).
 _e = /usr/prolog/prolog
Yes
```

12.3 File existence

exists/1

exists(_PhysFileName)

arg1 : ground : atom

This succeeds if *arg1* is the name of an existing file. The path *arg1* is first expanded according to the ***ProLog by BIM*** expansion rules before it is tested on existence.

12.4 Time predicates

cputime/1

cputime(_CpuTimeUntilNow)

arg1 : free : real

Arg1 is instantiated to the number of seconds of processing unit time (cputime) of the engine process used since ***ProLog by BIM*** was started. The accuracy of **cputime/1** depends on the operating system.

time/1

time(_Goal)

arg1 : partial

Arg1 is considered as a goal and is executed. The processing time (cputime) in seconds taken to execute the goal *arg2* is written to the current output stream.

time/2

time(_Goal, _CpuTimeTillSuccess)

arg1 : partial : term
arg2 : free : real

As **time/1**, but *arg2* is instantiated to the processing unit time (cputime) needed to execute the goal *arg1*.

Built-in Predicates

Chapter 13
Signal Handling

13.1 Introduction

The signal handling mechanism of ***ProLog by BIM*** 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 by 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 explicitly reset to accept status. The UNIX signals are supported (such as SIGINT, SIGSEGV, SIGPIPE).

A number of signals have a default handler installed. This handler terminates the current query after printing out a message of the form:

```
*** SIGNAL *** SIGname
```

Example

For a complete example regarding signal handling see

```
$BIM_PROLOG_DIR/demos/general/interrupts.pro
```

Example

```
/* definition of the interrupt handler */
my_interrupt_handler('SIGINT', _Goal) :-
     status,
     write('Resuming ':_Goal),nl.

/* retrieving the status of the interrupt signal */
status :-
     signal('SIGINT',status(_Status,_Nb)),
     write('Pending signals ':_Nb),nl,
     write('Status ':_Status),nl,nl.

/* installing the interrupt handler */
init :-
     install_prolog_handler('SIGINT',my_interrupt_handler),
     signal('SIGINT',accept),   % the default is ignore !!
     main.
```

13.2 Installing signal handlers

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* instantiated to 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.

13.3 Controlling signal delivery

The delivery of signals can be controlled by the programmer with the following set of built-in predicates.

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

query about status.

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.

Example

```
?- signal('SIGKILL',status(_1,_2)).
 _1 = accept
 _2 = 0
Yes
```

wait/0

wait

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**.

Built-in Predicates

Chapter 14
Error Handling and Recovery

Error messages can be controlled in several ways and at different levels in ***ProLog by BIM***. The system has three types of error messages: general messages, system messages and table warning messages. In general, all error messages are written to the standard error stream. The control for system and table warning messages is limited. The messages can be switched on or off (see **please/2**). General error messages can also be switched off globally. They can also be redirected with the **tell_err/1** predicate (see "*Input-Output - Redirection of streams*").

This section describes predicates which offer a finer control over the error rendering and the contents of the messages that are rendered. Existing error messages can be changed as well as new error messages can be added.

14.1 Error rendering

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*. If an error occurs and its 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 with 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 : any : atom
arg2 : any : integer
arg3 : any : list

The last pending error is reported. *Arg1* is unified with 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.
If there is no pending error, the predicate fails.

After this inquiry, the error status is cleared.

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 list.

The argument list *arg3* must contain all specified class arguments, followed by all specified error arguments, in the right order and in the form as specified in the section on "*Description of error arguments*" (see later in this section). The corresponding *arg3* of **error_status/3** can be passed directly as *arg*3 to this predicate.

This predicate always exits with an error condition. It can be used to simulate the behavior of built-in predicates in case of error detection.

error_print/0

error_print

The message of the last error is printed out, regardless of the warn switch. If no error messages have been issued previously, one will be generated.

error_print/1

error_print(_ErrMessage)

arg1 : free : atom

The message of the last error is transformed to an atom and unified with *arg1*.

error_print/3

error_print(_ErrClass,_ErrNumber,_ErrArguments)

arg1 : ground : atom or integer
arg2 : ground : integer
arg3 : ground : list

Similar as **error_raise/3**, but this predicate does not raise an error condition. It only prints out the message and succeeds. Another difference is that it forces the printing of the error, regardless of any switches that could disable error messages.

It only exits with an error condition if an error occurred in the use of the predicate itself.

error_print/4

error_print(_ErrMessage,_ErrClass,_ErrNumber,_ErrArguments)

arg1 : free : atom
arg2 : ground : atom or integer
arg3 : ground : integer
arg4 : ground : list

Same as **error_print/3**, but the output is transformed to an atom and unified with arg1.

14.2 Error recovery

err_catch/5

err_catch(_Error, _CatchErrors, _Goal, _FailGoal, _SuccessGoal)

arg1 : free : integer
arg2 : ground : integer or list of integers or term
arg3 : partial : term
arg4 : partial : term
arg5 : partial : term

The **error recovery** mechanism is active when the binary please switch "error recovery" (r) is on.

Execution mechanism:

The goal *arg3* is executed.

If no error occurs, this is followed by the execution of goal *arg5*.

If an error occurs during the execution of *arg3*, this execution terminates. *Arg1* is unified with the error number. If that number appears in *arg2*, the goal *arg4* is executed. Otherwise control goes back to the next enclosing **err_catch/5** goal.

Error ranges (*arg2*) are specified in one of the following ways:

single error:	integer
error range:	[integer, integer]
range list:	[integer, integer] | [integer, integer] | ...

When the please option "error-recovery" (r) is off, the goal *arg3* is executed. If the goal succeeds, the goal *arg5* is executed. Otherwise the err_catch goal fails.

Example

```
one( _x ) :-
     error_message([479 - 499,800], off),
     err_catch( _e , ([479, 499] | 800),
          two( _x , _y ),
          failure( _e , _x ), success( _y )),
     write('\n  err_catch one  succeeded  \n'),
     error_message([479 - 499, 800], on) .

two( _x , _y ) :-
     err_catch( _ , 497,
          calc( _x , _y ),
          modulo_zero( _y ), true),
     write('\n  err_catch two  succeeded \n') .

calc( _m , _res ) :-
     _res  is 12 mod  _m  .

failure( _e , _x ) :-
     printf('ERROR %d CAUGHT :', _e ),
     write('result can not be computed for '),
     write( _x ) .

success( _x ) :-
     write('result is '),
     write( _x ) .

modulo_zero('mod_by_zero') :-
     write('ERROR 497 CAUGHT:\
           attempt to take modulo by zero ') .
Yes
```

Considering the previous definitions, the following cases can be distinguished:
1/ The query succeeds:

```
?- one(7).

     err_catch two  succeeded
result is 5
     err_catch one  succeeded
Yes
```

2/ The error is caught by the immediately enclosing **error_catch/5** which succeeds and the execution continues.

```
?- one(0).
ERROR 497 CAUGHT: attempt to take modulo by zero
     err_catch two  succeeded
result is mod_by_zero
     err_catch one  succeeded
Yes
```

3/ The error is not caught immediately, the control goes back to the next enclosing **err_catch/5** where the errno is in the range. The err_catch succeeds and the execution continues.

```
?- one(3.4).
ERROR 495 CAUGHT :result can not be computed for 3.400000e+00
     err_catch one  succeeded
Yes
```

14.3 Customizing error messages

The error messages of the ***ProLog by BIM*** system are defined by means of an error description file. This file contains a set of facts describing the error messages that must be handled and how they must be rendered. At the installation of ***ProLog by BIM***, such a description file must be provided for linking into the BIMprolog executable. At runtime, any program can modify the error messages by interactively loading a description file. The default error description file is:

```
$BIM_PROLOG_DIR/install/errors.pro
```

By instructing the ***ProLog by BIM*** linker (BIMlinker) to use another error description file, the error handling can be customized at installation time. To modify it at runtime, the built-in **error_load/1** must be used.

Loading the error description file

error_load/1

error_load(_ErrorFile)

arg1 : ground : atom

The current set of error descriptions is replaced by the contents of the error description file *arg1*.

The error description file

An error description file must contain facts for the predefined predicates err_class/5, err_ordinal/2, err_msg_range/2 and err_msg/3.

err_class/5

err_class(_ClassNr, _ClassId, _ClassText, _ClassArgs,_ClassMessage).

arg1 : ground : integer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : list of atom
arg5 : ground : atom

The error class with number *arg1* is associated to identifier *arg2*. Its rendering text is *arg3* and a global class message is given by the text *arg5* with arguments described in *arg4*. The class number *arg1* must be in the range 1..16. It is solely used by the system to uniquely identify the error classes. The logical class identifier *arg2* is the symbolic version of the numeric identifier *arg1*. Both identifiers can be used in the error handling built-in **error_raise/3**. The symbolic identifier is preferred as it is also returned by the system in **error_status/3**, and it enhances readability.

When an error of the specified class is issued, it is rendered in the following general lay-out:

```
*** CLASS number *** class message error message
```

The class name CLASS is the text specified in *arg3*. The number is the error number (as specified in **err_msg/3**). The next item is a global class message, which is the same for each error of this class. It consists of the text specified in *arg5*. This text may contain references to arguments denoted by %i where i is the number of the argument to be inserted. Arguments are numbered from 1. All arguments that can occur, must be described in *arg4* which is a positional list with a description for each argument (see "*Description of arguments*").

The system predefined error classes are given in the table below. The mapping between identifiers (numerical and symbolic) should not be modified, as well as the number of arguments specified (redefining the class identifier may break programs that use the error handling built-ins). If desired, the rendering text and global class message can be changed.

Index	Name
1	SYNTAX
2	SEMANTIC
3	COMPOVFL
4	WARNING
5	RUNTIME
6	BUILTIN
7	OVERFLOW
8	MODE

Example

With the standard system error descriptions, including

```
err_class(6,BUILTIN,BUILTIN,[functor],'%1 : ') .
```

error 307 occurring in built-in ascii/2 is rendered as :

```
*** BUILTIN 307 *** ascii/2 : The first argument must be free
or an atom.
```

The global class message only contains the name of the built-in, some white space and a colon.

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 must 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 defined error messages is specified. The first number of the range is *arg1* and the last is *arg2*. Any existing error message range that overlaps with this range, is discarded.

The range minimum must be greater than 0, and its maximum must be greater than the minimum. The system predefined error message range is from 100 to 9999.

All defined error messages must be in an existing range, except for the special error message with number 0.

At most 255 different ranges may be defined.

err_msg/3

err_msg(_ErrorNumber, _ErrorArguments, _ErrorMessage).

arg1 : ground : integer
arg2 : ground : list of atom
arg2 : ground : atom

The error message with number *arg1* is specified to have arguments as described by *arg2*, and as text *arg3*. Only error numbers within existing ranges (see **err_msg_range/2**) can be used for *arg1*. An exception is the number 0. This special message is a catchall: it is used for each error that has no associated message defined.

The text *arg3* may contain references to arguments denoted by %i where i is the number of the argument to be inserted. Arguments are numbered from 1, starting from the real error argument list (error class arguments are not accessible).

Description of error arguments

The arguments of an error message are described to enable the system to render the message properly and also to enable checking of the correct use of the error message. A description simply consists of the type of the argument, which can be any of:

TypeId	Actual	Description
ordinal	integer	an ordinal number
integer	integer	an integer number
string	pointer	a null-terminated character array
atom	atom	a Prolog atom
functor	term	a Prolog term functor

An ordinal must be presented as an integer in the range 1..5. It is rendered in a textual form, as defined with err_ordinal/2.

An integer can be any (positive or negative) integer and is rendered as is.

A string can be any pointer to a null-terminated character array. A Prolog atom can be converted into a string with the built-in **stringtoatom/2**.

An atom is given as a ***ProLog*** atom and it is rendered as its textual representation.

A functor can be represented in different forms:

- a structured term where the principal functor defines the functor
- a structured term of the form name/arity
- an atom of the form name_arity

It is always rendered in the form name/arity.

Example

The functor argument error(1,_,x) is rendered as error/3. It can also be given as error/3 or error_3.

Compiler Directives

Compiler Directives

Chapter 1
Directives

Directives are parser or compiler commands that influence the way clauses are parsed and translated. Directives overwrite the current settings of the compiler. Part 1 "*Principal Components - The Compiler*" has a complete section devoted to the generation of the code.

Definitions of predicates can be mixed with directives, but this complicates later understanding of how the program executes. Common practice is to place all the directives at the beginning of the file except for the index and mode declarations which are most often placed before the predicate definition and serve as a first comment.

Most of the directives mentioned in this chapter have an equivalent built-in predicate which allows the same functionality from the running engine for the dynamically defined predicates.

A directive has the following form

```
:- <body> .
```

It looks like a clause without a head.

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.

Multiple directives can be given in the form:

```
:- directive1, directive2, ... directive .
```

Certain predicate handling directives also accept multiple arguments, these are given as a sequence.

```
:- mode p1(i,o), p2(r,i) .
```

1.1 General directive

option/1

:- option(_OptionList)

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 with earlier releases.

Option	Description	Directives
a	Generate dynamic code	alldynamic
c	Parse in DEC-10 syntax	compatibility
d	Generate debug code	set[no]debug
e	Translate in-line evaluation	-
h	Hide predicates	[no]hide
p	Include operator definitions	-
w	Give warning messages	-
x	Allow \ as atom escape sign	-

Option settings can temporarily be modified by first saving the current set, modifying as desired and afterwards restoring from the saved set. Savings can be nested.

s	Save the current option settings.
r	Restore the options from the previously saved set.

Example

```
:- option('d+').
```

1.2 Dynamic and static

dynamic/1

:- dynamic(_Name /_Arity)

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.

Note

If the **dynamic/1** directive appears after a module declaration, then a **local/1** directive for *arg1* is implied. Such **dynamic/1** declaration is only useful for predicates defined in the same module and can be applied to both global and local predicates.

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

:- alldynamic

All predicates in the file following the **alldynamic/0** directive are dynamic.

1.3 Debugging

setdebug/0

:- setdebug

All predicates in the file following the **setdebug/0** directive are translated into debug code.

setnodebug/0

:- setnodebug

All predicates in the file following the **setnodebug/0** directive are translated into non-debug code.

For more information on debug code and debugging in general see *"The Debugger"*.

1.4 Compatibility

compatibility/0

:- compatibility

All predicates in the file following the **compatibility/0** directive will be parsed in the DEC-10 Prolog syntax.

1.5 Hiding

hide/0

:- hide

All predicates in the file following the **hide/0** directive are hidden. This influences the built-in predicates listing and clause.

nohide/0

:- nohide

All predicates in the file following the **nohide/0** directive are visible. This influences the built-in predicates listing and clause.

1.6 File inclusion

include/1

:- include(_FileName)

arg1 : ground : atom

File *arg1* is textually included in the source file it is defined in. This means that a "new" source file is composed by replacing the **include/1** directive with the contents of file *arg1*. The composed 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. The use of the save and restore settings of the **option/1** directive allow the included file to issue directives which will only affect the included file *arg1*.

Path name expansion on *arg1* is done automatically (~, ~<user>, $<variable>,-L,-H). (See the section on path name expansion.)

1.7 Optimization

The directives explained in this section are used by the compiler to optimize the generated code. More information on the generation of the code can be found in "*Principal Components - The Compiler*".

mode/1

:- mode _ModeDeclaration

arg1 : ground : term

This directive is used to indicate the intended input-output use of a particular predicate. The compiler can generate more efficient code with this information. **mode/1** is defined as a prefix operator. *Arg1* is a term where each argument represents the intended input-output use of the corresponding argument of the corresponding predicate.

There are three possible modes:

i (+)	: input :	the argument is always ground when the predicate is called
o (-)	: output :	the argument is always free when the predicate is called
? (?)	: any :	anything else

The symbols between parentheses can be used in compatibility mode (see "*Please options*" in "*Principal Components - The Engine*").

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.

Errors in mode declarations can lead to failing goals or (exceptionally) to machine exception (segmentation faults).

If the **mode/1** directive appears after a module declaration, a **local/1** directive for *arg1* is implied. Such **mode/1** declaration is only useful for predicates defined in the same module and can be applied to both global and local predicates.

Example

```
:- module(one) .
:- mode a(i,i,i,o)
:- global b/2 .
:- mode b(i,o).
```

This defines a local predicate **a/4** and a global predicate **b/2**.

index/2

:- _Name/_Arity index _IndexSpecification

arg1 : ground : atom/integer
arg2 : ground

The compiler uses this directive to generate special indexing instructions, on the arguments defined with *arg2*, for the predicate with name and arity specified by *arg1*. **index/2** is defined as an infix operator.

Static predicates

Arg2 must have the form: integer or (integer, integer, ...)

At most three arguments can be specified.

The order in which arguments are used for indexing is the same as the order given by the user in the index declaration, except that "i" mode arguments are used for indexing before "?" mode arguments and "o" mode arguments are never used for indexing.

The default indexing is on the first three 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 indexes dynamic predicates on their first argument.

Notes:

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 example, an index put on an output argument - as specified by a mode declaration for the same predicate - is meaningless.

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 therefore 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.

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

:- op(_Precedence,_Format,_Name)

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.

A list of all predefined operators is mentioned in *"Syntax"*.

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 .
```

The operator directives only effects the current file. Use of the -p compiler option provides a way to replace the operator definitions in the engine with the operator definitions of the file.

1.9 Modules

The directives on modules are explained in *"Modules"*.

1.10 External Language Interface

The directives for the External Language Interface are explained in "*External Language Interface*"

Modules

Modules

Chapter 1
Modules

This chapter describes the use of modules in ***ProLog by BIM***.

ProLog by BIM supports a module concept that makes it easy and natural to split a program into separate components - called modules - which interact only through a set of predicates that 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 by BIM*** module concept is flat, static and name/arity based (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).

Since 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.

1.1 Compiler directives

By default predicates belong to the global module. This section describes how this default behavior can be changed by using directives. The directives are put in a source file and are interpreted at compile time.

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* (they are local to the module). The predicates preceding 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 one **module/1** directive at the most in a file.

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 local character 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/1** directive declares *arg1* as a local functor. It must be defined after a module directive. The directive is used:

- To declare that *arg*1 belongs to the module. This is useful for example 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 built-in predicate. For this purpose, the **local/1** directive is even necessary. A warning is given by the compiler stating that you are redefining a built-in predicate (unless you use the "-w-" flag).

Example

version without emq

```
:- module(mod) .:
:- local g/1 .
a(_l) :-
     bagof(_x,g(_x),_l) .
```

version with emq

```
:- module(mod) .
:- local g/1 .
a$mod(_l) :-
bagof(_x,g$mod(_x),_l) .
```

Compare with:

```
:- module(mod) .
a(_l) :-
     bagof(_x,g(_x),_l) .
```

```
:- module(mod) .
a$mod(_l) :-
     bagof(_x,g$(_x),_l) .
```

Both columns are identical although the right column uses explicit module qualification (emq).

Implicit 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 a **module/1** directive, a **local/1** directive for that functor is implicit.

```
:- module(mod)
:- dynamic 1/2 .
```

A local directive for the predicate l/2 is implicit. All appearances of l/2, which do not have an emq, refer to l$mod/2.

A functor with an emq used after a module declaration implies an implicit **local/1** directive for that functor if the specified modules are equal.

```
:- module(mod)
m( 1 ,2 ) :- l$mod(2,3).
```

A local directive for the predicate **l/2** is implicit. All appearances of **l/2**, which do not have an emq, refer to l$mod/2 .

import/1

:- import _Name / _Arity from _ModuleName .

arg1 : ground : term

With the **import/1** directive with a predicate *arg1* from module *arg2* is known in the current module and can thus be written without emq.

Example

Assuming that the predicate **g/1** is defined as local in the module "bar":

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) .
a(_x) :- g(_x) .
```

```
:- module(mod) .
a$mod(_x) :- g$(_x) .
```

Without the import directive, the occurrence of **g/1** in the body of **a$mod/1** would be interpreted as a call to a global predicate **g/1**.

Implicit import directive:

A functor with an emq used after a module declaration implies an implicit **import/1** directive for that functor if the specified modules are not equal.

```
:- module(mod)
m( 1 ,2 ) :- i$other(2,3).
```

An import directive for the predicate **i/2** from the module other is implicit. All subsequent uses of **i/2** really refer to i$other/2 .

global/1

:- global _Name / _Arity .

arg1 : ground : term

The **global/1** directive defines *arg1* as a global functor. It is only useful when following a **module/1** directive. The global/1 directive is useful to declare that the predicate that is defined belongs to the global module.

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) .
```

Without the **global/1** directive **a/1** would belong to module mod.

1.2 Module built-in predicates

When a ***ProLog by BIM*** session starts, it starts in the global module and explicit module qualification can be used. None of the import, local or global declarations appearing in the consulted files have effect.

If a predicate is defined as being local to a module, it can only be called with explicit module qualification, or by referring to the predicate from within the module. Positioning within a module is done with the **module/1** built-in.

Example

```
?-module(modA) .
Yes
```

After the execution of this query, all predicates from modA are accessible without explicit module qualification.

Entering new definitions in no_querymode, will add them to the new module. The assert predicates will assert the term exactly as it is specified (terms without emq will always be added to the global module).

The following built-ins allow to interact with the module system during the execution of a query.

module/1

module(_ModName)

arg1 : any : atom

If instantiated, the current module becomes *arg1*. Changing the current module only takes effect after the interpretation of the current query. If free, *arg1* will be instantiated to the name of the current module.

Example

```
?- please(wm,on),assert(a$modA).
Yes
?- module(modA),a.
*** RUNTIME 405 *** Illegal call : unknown predicate a/0.
No

?- a.
Yes
```

The first call to a/1 refers to a$". The second query to a/1 refers to a$modA/1 because the module change has taken effect before the interpretation of that query.

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*.

Example

```
?- module(a$one(_x),_y), write(_y) .
one
```

module/3

module(_QualTerm, _ModName, _Term)

arg1 : any : term
arg2 : any : atom
arg3 : any : term

Arg3 is the global 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.

Example

```
?- module(a$one(b$two), one, a(b$two)) .
Yes
```

mod_unif/2

mod_unif(_Term1, _Term2)

arg1 : any : term
arg2 : any : term

Unifies the two arguments, as if they had no module qualification at any level (as if they were globals).

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 : term

The same as **write/2**, but all non-global names are written with their explicit module qualification. (see *"Built-in Predicates - Input-Output"*)

writem/1

writem(_Term)

arg1 : any : term

The same as **writem/2**, but on the current output stream. (see *"Built-in Predicates - Input-Output"*)

mlisting/1

mlisting(_ModName)

arg1 : ground : atom

Writes all the definitions of dynamic predicates that are local to the module named *arg1* on the current output stream.

1.3 Explicit module qualification

Atoms and functors are fully identified by their name, arity and the module in which they are defined. The **explicit module qualification (emq)** is the syntactic correct way to express this. The global module has the name '' (the empty atom).
The other ***ProLog by BIM*** items - integers, reals, pointers and variables - have no module qualification.

Example

```
atom$modA
functor$modA(_arg1,atom$modA,globalatom$)
globalatom$
globalfunctor$(_arg1,atom$modA,globalatom$)
```

denotes **atom** and **functor/3** as defined in the module modA, and **globalatom** and **globalfunctor/3** as globals.

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.

1.4 Resolving module qualification

The rules for resolving the module qualification are (these apply at compile time):

- If **foo/n** is explicitly qualified with a module, then **foo/n** belongs to the module it is qualified with.
- If **foo/n** has a definition after the module declaration it belongs to that module.
- If **foo/n** is not explicitly qualified but (only) has a local declaration or an implied local declaration, **foo/n** is local.
- Otherwise, if **foo/n** is imported from only one module with an import or an implied import declaration, **foo/n** belongs to that module.
- Otherwise, if foo/n already belongs to a single module, every unqualified foo/n belongs to that module.
- Otherwise, if **foo/n** is not imported, **foo/n** is global.
- Otherwise, an error against the module rules has been made.

Example

```
:- optione('a+').
:- module(modA).
test:- a$modB.
a :- c.
b :- _X = a, f(a)
```

A listing of this code gives:

```
test$modA :- a$modB .
a$modA :- c .
b$modA :- _X  = a$modA, f(a$modA) .
```

In case of ambiguity, explicit module qualifications must be used. Ambiguity arises in the following cases:

- When foo/n is imported from different modules.
- When foo/n is imported and has also a local declaration.
- When foo/n is declared global and also has local declaration.
- When foo/n is declared global and is also imported.

1.5 General built-ins and modules

This section explains the behavior of some general built-ins when used inside a module.

- The following predicates always create global atoms:

  numbervars/3
  numbervars/4
  namevars/3
  namevars/4
- The following predicates return global atoms in their first argument:

  ascii/2
  name/2
  atomtolist/2
  asciilist/2
  inttoatom/2
  pointertoaton/2
- The predicate **atomtolist/2** returns global atoms in its second argument.
- The predicate **read/1** interprets 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**.

External Language Interface

0.1 Overview of the external language interface

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 by BIM*** 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 by BIM*** programmer can invoke his 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 by BIM*** predicates can be invoked from the external functions.

To illustrate its versatility, the external language interface has been used to couple ***ProLog by BIM*** with existing graphics, windowing and database packages. Among these are: OSF/Motif**, OPEN_LOOK/XView**, Xlib**, Xt** and ORACLE**, Sybase** giving the application builder unrestricted access to graphics and windowing facilities as well as to the operational data in relational databases.

External Language Interface

Chapter 1
Linking External Routines

1.1 Incremental linking

Before a routine written in a language other than ***ProLog*** can be used from a ***ProLog*** program, it must be linked into the ***ProLog by BIM*** system and loaded. Using **BIMlinker** (see "*Principal Components - The Linker*"), one can create a new, customized ***ProLog by BIM*** 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 in a ***ProLog*** engine is performed by using the dynamic linking facilities of SunOS. As a result, it is restricted to the incremental linking of external routines

- into a dynamically linked engine
- from a shared object

The default installation of the engine creates dynamically linked executables. The second constraint, that the external routines must be defined in a shared object, is the user's responsibility.

It is possible to let ***ProLog*** handle some of the burden to create a shared object. When enumerating the list of object files and libraries that contain external functions to be linked in, ***ProLog*** will automatically generate a shared object and link it dynamically. Alternatively, the user can completely take over control to generate the shared object and instruct ***ProLog*** to use it. This saves the time to generate that shared object on each incremental linking of the package.

Note:
This functionality of incremental linking will not work for all examples on SunOS 4, due to problems in the dynamic linking facility of the underlying operating system. In case of problems, the underlying dynamic linker will issue an error message and in some cases the process will be stopped. The higher the SunOS number, the fewer problems will be encountered. In case of problems, one can always use BIMlinker, and make a customized executable incorporating the external routines.

Incremental linking can be achieved by consulting a file or interactively by executing some queries. 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 shared object or in which object files and/or libraries they can be found. Secondly, 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 shared object 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 immediately associated with the ***ProLog*** predicate.

Creating shared objects

If the incremental linker receives a set of object files and/or libraries instead of a shared object, it will generate a shared object which is then linked into the running engine. The generation of the shared object is performed in two steps. First a link definition file is created and compiled (with a C compiler). Next, that link definition object file is linked, together with the objects and libraries, into a shared object. The shared object is finally dynamically linked into the running engine. For the compilation command, the link command and the file name, default values are taken.

	SunOS 4.x	SunOS5.x
file name	/tmp/BP.dl.%U.so	/tmp/BP.dl.%U
compile cmd	/bin/cc -c %I	/usr/ccs/bin/ld -c %I
link cmd	/bin/ld -o %O %I	/usr/ccs/bin/ld -o %O %I

The default values can be overruled by setting environment variables

Variable	Value
BIM_DLL_FILE	Dynamic linker link definition file name
BIM_DLC_CMD	Dynamic linker compilation command
BIM_DLL_CMD	Dynamic linker link command

The values of these variables must be text strings which can contain references to parameters, in the form of a % followed by a single capital letter. To include a literal %, it must be doubled as %%.

Indicator	Parameter
%U	Unique file name substitute
%I	Input file(s)
%O	Output file

A file name specification indicates the base name of the file. A .c suffix is appended for the source file name and a .o suffix for the object file name. The final shared object file name is made by appending a .so suffix. The base name can include a path, and possibly a parameter in order to convert it to a unique file name.

The actual values of the input and output file parameters are different for each of the link and compile commands. For the compile command, the input file is the link definition source file, with a name as specified, and with extension '.c'. The output file name is deduced from the input file name by stripping any leading paths and replacing the extension with '.o'. It should normally not be used in the compile command. Neither should another output file be specified in that command, since this would make it invisible for the ***ProLog*** engine. For a link command, the input file parameter is substituted by a sequence of files. First comes the output file of the compile command, containing the link definitions. This is followed by the objects and libraries, as mentioned in the linker directive. Output file for the link command, is derived from the link definition file name by adding a.so suffix.

These environment variables do not influence the dynamic linker if it is directly fed with a shared object.

1.2 Linker directives and built-ins

The linker directives **extern_load/2,3** tell the incremental linker of ***ProLog by BIM*** which external routines must be linked and which shared object or object files and libraries must be opened to find them. In a file, they must be specified as compiler directive, and interactively as built-in predicates.

extern_load/3

:- extern_load(_Identifier, _Externals, _Objects)

extern_load(_Identifier, _Externals, _Objects)

arg1 : ground : atom
arg2 : ground : list of atom
arg3 : ground : atom or list of atom

An external object is loaded. It is identified by *arg1*. A list of requested external functions is enumerated in *arg2*. External definitions for these functions can be found as indicated by arg3. This is either a list of object modules and libraries, or a single shared object module. In the first case, ***ProLog*** generates a shared object that is linked against the given objects and libraries. That shared object is then dynamically linked. In the second case, the indicated shared object is dynamically linked.

extern_load/2

:- extern_load(_Externals, _Objects)

extern_load(_Externals, _Objects)

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

Same as **extern_load/3**, but without identifier for the module.

extern_load/1

:- extern_load(_Externals)

extern_load(_Externals)

arg1 : ground : list of atom

Same as **extern_load/3**, but without identifier, and the external functions are searched for in the running process itself.

extern_unload/1

extern_unload(_Identifier)

arg1 : ground : atom

The external module, identified by *arg1*, is unloaded. The external function code is removed from the process's address space. All external predicates that were mapped to functions in this module, become undefined. External modules that were not identified, cannot explicitly be unloaded. An **unload/1** implicitly performs an extern_unload/1 of all external modules that were loaded from the file, even those that were not identified.

extern_loaded/1

extern_loaded(_Identifier)

arg1 : ground or free : atom

If *arg1* is ground, the predicate succeeds if an external module with identifier *arg1* is loaded. If *arg1* is free, it is instantiated to all identifiers of a loaded external module one by one upon backtracking.

1.3 Predicate mapping declaration

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 a compiler directive, and interactively as a built-in predicate.

The syntax for the ***ProLog*** predicate and function declarations used in the mapping declarations can be described as follows:

<pred_decl>	⇒	*<pro_name>* [(*<args>*)]
<func_decl>	⇒	*<pro_name>* [(*<args>*)] : *<type>*
<args>	⇒	*<argument>* [, *<args>*]
<arg>	⇒	*<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>	⇒	**NoprotoC** | **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.

Example

```
:- extern_predicate( chmod(integer:r,string:i,integer:i) )
:- extern_predicate(
          lreadlink,
          readlink(integer:r,string:o,string:i) ).
```

are two examples of an extern_predicate declaration as they are defined in the UnixFileSys library.

The language specification C indicates ANSI C with function prototypes. Pre-ANSI C had no function prototypes, and should be mentioned as **NoprotoC**. The difference is important for type expansion: with prototypes, short and float are not propagated to int and double, as is the case when no prototypes are used.

extern_language/1

:- extern_language(_Language)

extern_language(_Language)

arg1 : ground : atom

The default language for the declarations that follow, is set to *arg1*. It must be one of the supported language identifiers (see *<language>*). The default language is C.

extern_predicate/3

:- extern_predicate(_Language, _ExternalName, _PredDecl)

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 prefixes and suffixes 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>* 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)

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)

extern_predicate(_PredDecl)

arg1 : ground : term

Same as **extern_predicate/3**, but with the default language and where the external name is the same as the ***ProLog*** predicate name.

extern_function/3

:- extern_function(_Language, _ExternalName, _FuncDecl)

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 prefixes 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>* 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)

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)

extern_function(_FuncDecl)

arg1 : ground : term

Same as **extern_function/3**, but with the default language and where the external name is the same as the ***ProLog*** function name.

1.4 Interactive linking

For interactive incremental linking, some additional built-ins 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 *arg1*, 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 the declarations given prior to this command.

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.

Arg2 is instantiated to the external name of the routine with which the ***ProLog*** predicate *arg1* is associated. *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. This can be done with any compiler which generates the correct file format.

All libraries needed to link in the object files must be mentioned in the object list.

ProLog can in general only link shared objects. If a list of objects and libraries is defined, ***ProLog*** will generate a shared object of all the objects and libraries which are passed to the linker. Therefore it generates a temporary file with C-code. To compile that file, a C-compiler must be available. If such a C-compiler is not available, the user must make the shared object himself and pass that to ***ProLog***.

External Language Interface

Chapter 2
Calling External Routines from Prolog

2.1 Parameter mapping declarations

An external routine can be mapped to a Prolog predicate or function by using the mapping declarations as explained in the previous chapter. This paragraph describes in detail the argument declarations which are part of the predicate and function declaration. The argument declarations describe how the parameters are passed from Prolog to the external routine.

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 real pointer atom	integer real pointer string or string : *s*

C types

The correspondences between argument types in Prolog and data types in the external language is listed in the following tables.

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 : $s > 0$	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* > 0	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 by BIM*** 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 correct 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 one *return* parameter at the most, 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 is simply 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 modified when exiting it, a failure occurs. 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, since they are arrays of characters.

Mode *e* is a variant of *i*. The actual argument is first evaluated (as if the argument were evaluated by the =?/2 operator), and then passed with *input* mode (see also "*Built-in Predicates - Expression Evaluation - General evaluation*").

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 simple (atomic) types.
- 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 (atomic) types.
- 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* and *bpterm* 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 to Prolog). 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 action is 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 the value already has a reference for the declared language, structure and type, 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 action is 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 action is 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 0.5-inch wide box.

Boxes with a thick border, represent the data that is actually passed as parameter (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. They must remain active at least *during exit*. There are no problems if the external data are made *static*. 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, these parameter passing modes, combined with dynamic data structures, should be avoided. These combinations are labeled with "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 (see part *"Syntax"*).

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 **int/long**

call

short : i **short**

call

In Pre-ANSI C, type short is propagated to int. This is also the case when no prototypes are used.

call

float : i **float**

call

In Pre-ANSI C, type float is propagated to double. This is also the case when no prototypes are used.

call

real / double : i **double**

call

pointer : i **BP_Pointer**

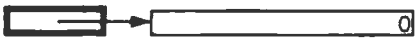
call

atom : i **BP_Atom**

call

string : i **BP_String**

call



The text zone is long enough to hold the actual string and a terminating 0 byte.

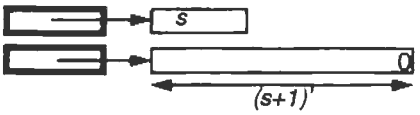
string : s : i (s > 0) **BP_String**

call

s' = even(s). The text zone is s' bytes.

string : 0 : i **BP_String**

call



(s+1)' = even(s+1). The text zone is (s+1)' bytes and s is set to the length of the actual string.

Note: There are two consecutive external arguments for this parameter.

bpterm : i **BP_Term**

call

untyped : i

call

C - Simple output

integer / long : o **int * / int ***

call

exit

short : o **short ***

call

exit

float : o **float ***

call

exit

real / double : o **double ***

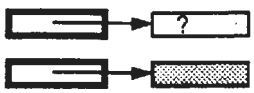
call

exit

pointer : o **BP_Pointer ***

call

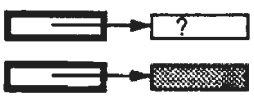
exit



atom : o **BP_Atom ***

call

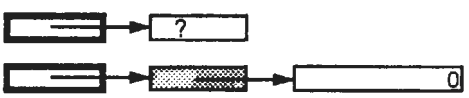
exit



string : o **BP_String ***

call

exit

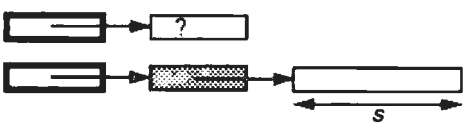


The text zone must be terminated with a 0 byte. It may be deallocated after *exit*.

string : s : o (s > 0) **BP_String ***

call

exit

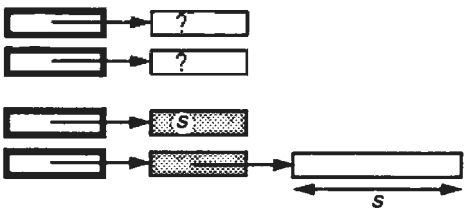


The whole length s of the text zone is considered as the resulting string. The text zone may be deallocated after *exit*.

string : 0 : o **int *** **BP_String ***

call

exit



The external routine has to indicate the length of the text zone at *exit*. The text zone may be deallocated after *exit*.

bpterm : o
untyped : o

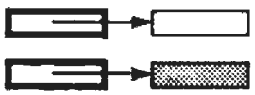
Not allowed.

C - Simple mutable

integer / long : m **int * / int ***

call

exit

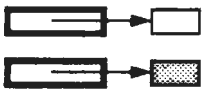


Default value : 0.

short : m **short ***

call

exit

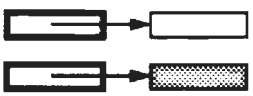


Default value : 0.

float : m **float ***

call

exit

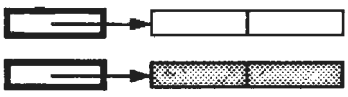


Default value : 0.

real / double : m **double * / double ***

call

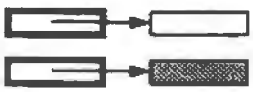
exit



Default value : 0.

pointer : m **BP_Pointer ***

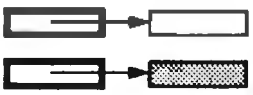
call *exit*



Default value : 0x0.

atom : m **BP_Atom ***

call *exit*



Default value : empty atom.

string : m **BP_String ***

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) **BP_String ***

call

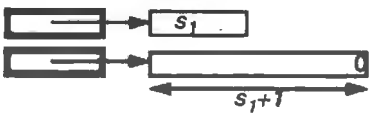
s' = even(s). The text zone contains the actual string (default empty), terminated with a 0 byte, truncated to s' bytes.

exit

The whole length s of the text zone is considered as the resulting string.

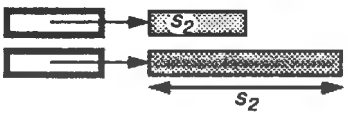
string : 0 : m **int * : BP_String ***

call



Defaults 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.

bpterm : m
untyped : m

Not allowed.

C - Simple return

integer / long : r **int / long**

exit

short : r **short**

exit

In Pre-ANSI C, type short is propagated to int. This is also the case when no prototypes are used.

exit

float : r **float**

exit

In Pre-ANSI C, type float is propagated to double. This is also the case when no prototypes are used.

exit

real / double : r **double / double**

exit

pointer : r **BP_pointer**

exit

atom : r **BP_Atom**

exit

string : r **BP_String**

exit

The text zone must contain the string, terminated with a 0 byte. It may be deallocated after *exit*.

string : s : r (s > 0) **BP_String**

exit

The whole length s of the text zone is considered as the resulting string.

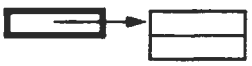
string : 0 : r
bpterm : r
untyped : r

Not allowed.

C - Array input

integer / long : array : i

call

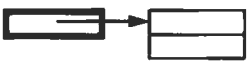


short : array : i

call

float : array : i

call

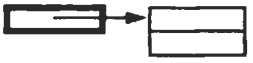


real / double : array : i

call

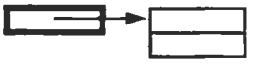
pointer : array : i

call



atom : array : i

call



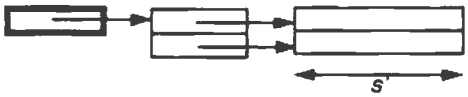
string : array : i

call

The text zones are long enough to hold the actual strings with a terminating 0 byte.

string : s : array : i (s > 0)

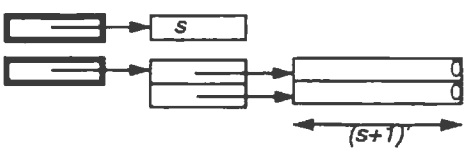
call



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.

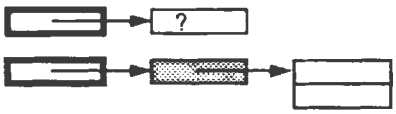
bpterm : array : i
untyped : array : i

Not allowed

C - Array output

integer / long : array : o

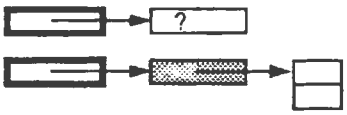
call
exit



The array may be deallocated after *exit*.

short : array : o

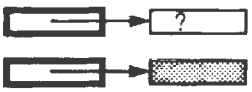
call
exit



The array may be deallocated after *exit*.

float : array : o

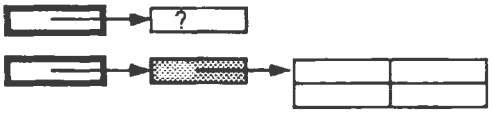
call
exit



The array may be deallocated after *exit*.

real / double : array : o

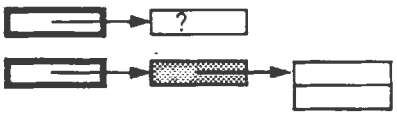
call
exit



The array may be deallocated after *exit*.

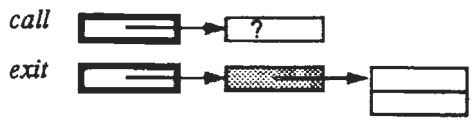
pointer : array : o

call
exit



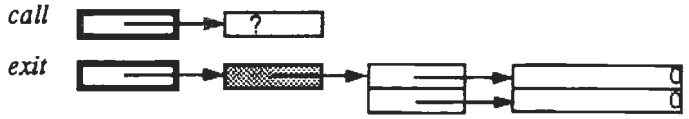
The array may be deallocated after *exit*.

atom : array : o



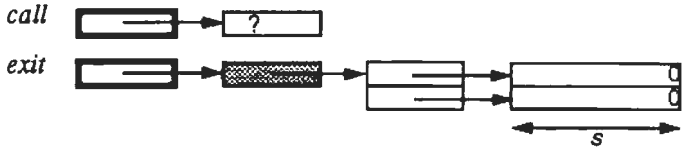
The array may be deallocated after *exit*.

string : array : o



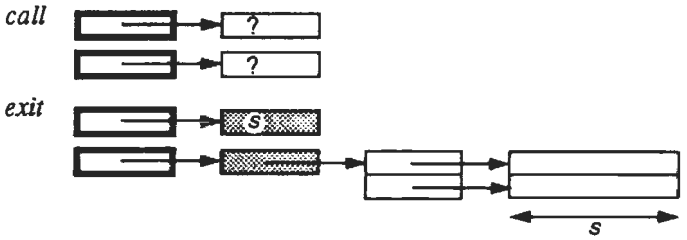
The text zones must end on a 0 byte. The pointer array and text zones may be deallocated after *exit*.

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



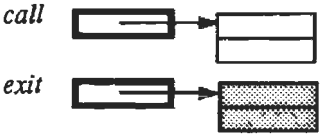
The external routine has to indicate the length of the text zones at *exit*. The pointer array and text zones may be deallocated after *exit*.

bpterm : array : o
untyped : array : o

Not allowed.

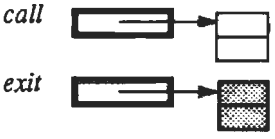
C - Array mutable

integer / long : array : m



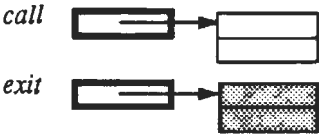
Default value : 0.

short : array : m



Default value : 0.

float : array : m



Default value : 0.

real / double : array : m

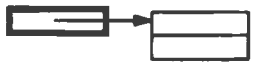
call

exit

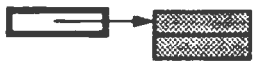
Default value : 0.

pointer : array : m

call



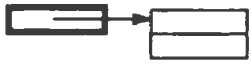
exit



Default value : 0x0.

atom : array : m

call

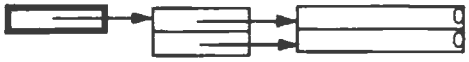


exit

Default value : empty atom.

string : array : m

call



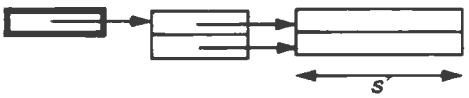
The text zones are long enough to hold the actual strings (default emtpy) with a terminating 0 byte.

exit

The text zones must end on a 0 byte. The text zones may be deallocated after *exit*.
Note: The text zones at *exit* are not necessarily the same as those at *call*.

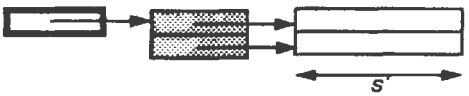
string : s : array : m (s > 0)

call



s' = even(s). The text zones are s' bytes long.

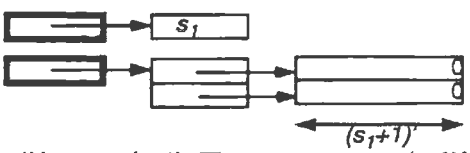
exit



The whole length s of the text zones are considered as the resulting strings. The text zones may be deallocated after *exit*.
Note: The text zones at *exit* are not necessarily the same as those at *call*.

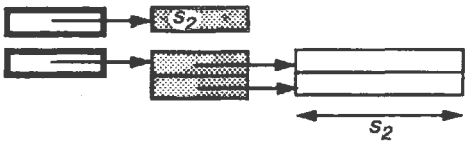
string : 0 : array : m

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*.
Note: The text zones at *exit* are not necessarily the same as those at *call*.

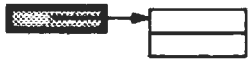
bpterm : array : m
untyped : array : m

Not allowed

C - Array return

integer / long : array : r

exit



The array may be deallocated after *exit*.

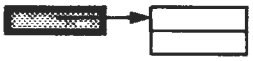
short : array : r

exit

The array may be deallocated after *exit*.

float : array : r

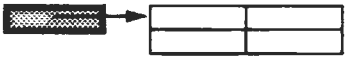
exit



The array may be deallocated after *exit*.

real / double : array : r

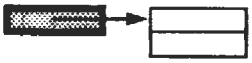
exit



The array may be deallocated after *exit*.

pointer : array : r

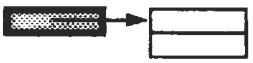
exit



The array may be deallocated after *exit*.

atom : array : r

exit



The array may be deallocated after *exit*.

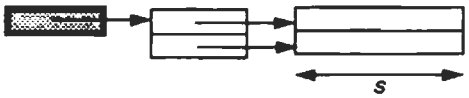
string : array : r

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
bpterm : array : r
untyped : 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.

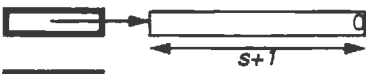
string : s : i (s > 0)

call 

Default : empty string.

The text zone contains the actual string, terminated with a 0 byte, truncated to s bytes.
Note: 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.
Note: There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

bpterm : i
untyped : i

Not allowed.

FORTRAN - Simple mutable

In FORTRAN simple output and simple mutable are equivalent.

integer / long : m

call 

exit 

Default value : 0.

short : m

call 

exit 

Default value : 0.

real / float : m

call 

exit 

Default value : 0.

double : m

call 

exit 

Default value : 0.

pointer : m

call 

Default value : 0x0.

exit 

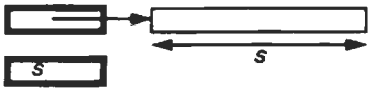
atom : m

call 

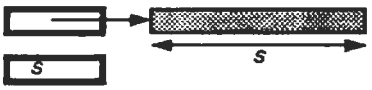
Default value : empty atom.

exit 

string : s : m (s > 0)

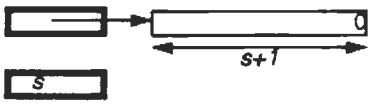
call 

The text zone contains the actual string (default empty), terminated with a 0 byte, truncated to s bytes.

exit 

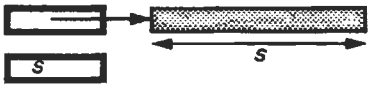
The whole length s of the text zone is considered as the resulting string.
Note: There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

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.
Note: There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

bpterm : m
untyped : m

Not allowed.

FORTRAN - Simple return

integer / long : r

exit 

short : r

exit 

real / float : r

exit 

double : r

exit 

pointer : r

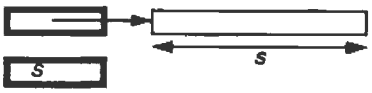
exit 

atom : r

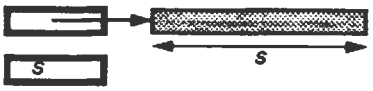
exit

string : s : r (s > 0)

call



exit



The whole length s of the text zone is considered as the resulting string.
Note: A FORTRAN *return string* is treated as a *mutable string* parameter.

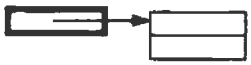
string : 0 : r
bpterm : r
untyped : r

Not allowed.

FORTRAN - Array input

integer / long : array : i

call



Default value : 0.

short : array : i

call

Default value : 0.

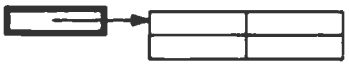
real / float : array : i

call

Default value : 0.

double : array : i

call



Default value : 0.

pointer : array : i

call

Default value : 0x0.

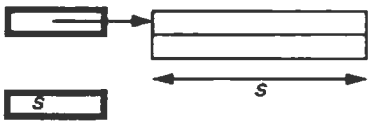
atom : array : i

call

Default value : empty atom.

string : s : array : i (s > 0)

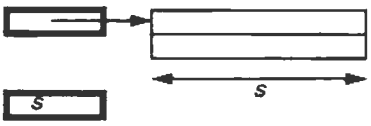
call



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.

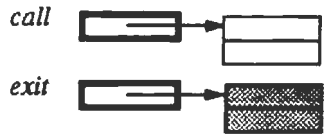
bpterm : array : i
untyped : array : i

Not allowed.

FORTRAN - Array mutable

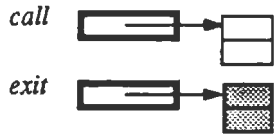
In FORTRAN array output and array mutable are equivalent.

integer / long : array : m



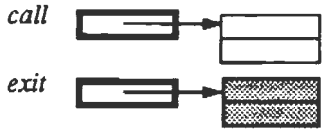
Default value : 0.

short : array : m



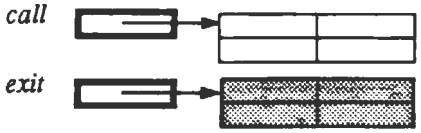
Default value : 0.

real / float : array : m



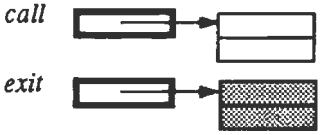
Default value : 0.

double : array : m



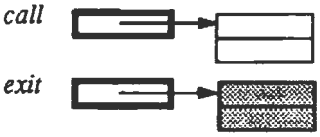
Default value : 0.

pointer : array : m



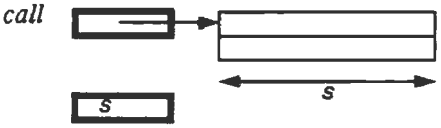
Default value : 0x0.

atom : array : m

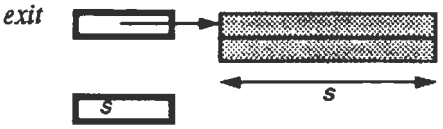


Default value : empty atom.

string : s : array : m (s > 0)



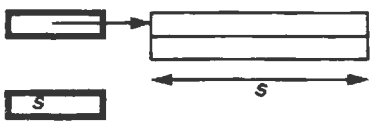
The text zones contain the actual strings (default empty), terminated with a 0 byte, truncated to s bytes.



The whole length s of the text zones are considered as the resulting strings.
Note: There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

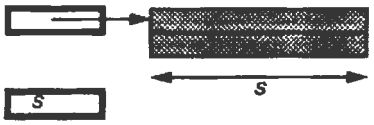
string : 0 : array : m

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.
Note: There are two external arguments for this parameter. The second one is at the end of the parameter list, following all regular parameters.

bpterm : array : m
untyped : array : m

Not allowed.

Pascal - Simple input

integer / long : i

call

short : i

call

float : i

call

real / double : i

call

pointer : i

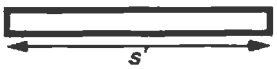
call

atom : i

call

string : s : i (s > 0)

call



s' = even(s). The text zone is s' bytes.

bpterm : i
untyped : i

Not allowed.

Pascal - Simple mutable

In Pascal simple output and simple mutable are equivalent.

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.

bpterm : m
untyped : m

Not allowed.

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*.

bpterm : r
untyped : r

Not allowed.

Pascal - Array input

integer / long : array : i

call

short : array : i

call

float : array : i

call

real / double : array : i

call

pointer : array : i

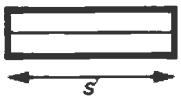
call

atom : array : i

call

string : s : array : i (s > 0)

call



s' = even(s). The text zones are exactly *s'* bytes long.

bpterm : array : i
untyped : array : i

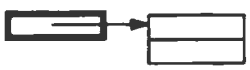
Not allowed.

Pascal - Array mutable

In Pascal array output and array mutable are equivalent.

integer / long : array : m

call



Default value : 0.

exit

short : array : m

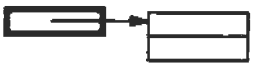
call

Default value : 0.

exit

float : array : m

call

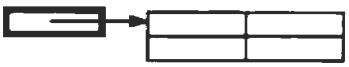


Default value : 0.

exit

real / double : array : m

call

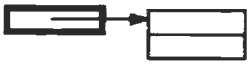


Default value : 0.

exit

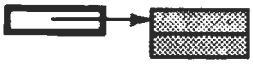
pointer : array : m

call



Default value : 0x0.

exit

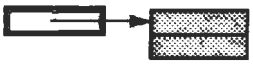


atom : array : m

call

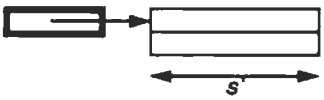
Default value : empty atom.

exit



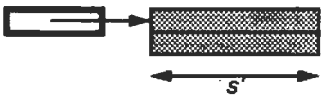
string : s : array : m (s > 0)

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.

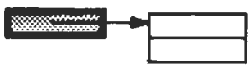
bpterm : array : m
untyped : array : m

Not allowed.

Pascal - Array return

integer / long : array : r

exit



The array may be deallocated after *exit*.

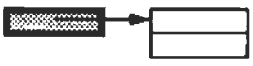
short : array : r

exit

The array may be deallocated after *exit*.

float : array : r

exit



The array may be deallocated after *exit*.

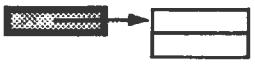
real / double : array : r

exit

The array may be deallocated after *exit*.

pointer : array : r

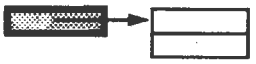
exit



The array may be deallocated after *exit*.

atom : array : r

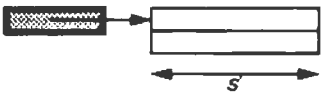
exit



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*.

bpterm : array : r
untyped : array : r

Not allowed.

2.4 Backtrackable external predicates

The ***ProLog by BIM*** external language interface provides facilities for simulating backtrackable external predicates. Such predicates behave in a way similar to normal non-deterministic Prolog predicates. For this purpose, a number of built-in predicates and external built-in routines are provided.

repeat/4

repeat(_Handle, _Id, _Size,_InitialValue)

arg1 : free : integer
arg2 : ground : term
arg3 : ground : integer
arg4 : ground : pointer

This predicate succeeds repeatedly upon backtracking. *Arg1* is unified with a unique handle for the backtrack point. It is identified by the user application with the term *arg2*.

At the first call, it reserves *arg3* units of 4 bytes. This space remains reserved until the backtrack point is cut away, either explicitly from an external routine, or implicitly by a Prolog cut operation. The maximal number of reserved units is 256.

If *arg4 is* a non-zero pointer, it is assumed to contain initial data for the reserved units. At the first call, this data will be copied into the reserved space. On backtracking, *arg4* is ignored. As a result, the memory to which *arg4* might point, can be deallocated.

repeat/5

repeat(_Handle, _ID, _Size, _InitialValue, _CutCallback)

arg1 : free : integer
arg2 : ground : term
arg3 : ground : integer
arg4 : ground : pointer
arg5 : ground : atom or pointer

Same as **repeat/4** with an additional cut callback handler. This handler is either a Prolog predicate with name *arg5* and arity 1, if *arg5* is an atom, or an external function at address *arg5* if it is a pointer.

When the backtrack point for this repeat is cut away, the callback handler is called with the backtrack point handle as argument. Since the backtrack point must still be available to the callback handler, it is only cut away after return from the callback.

BIM_Prolog_get_repeat_id()

BP_Term BIM_Prolog_get_repeat_id(handle)
int handle;

The identifier of the external repeat backtrack point, indicated by handle, is returned. An error message is issued and zero is returned if the handle is illegal.

BIM_Prolog_get_repeat_units()

BP_Pointer BIM_Prolog_get_repeat_units(handle)
int handle;

A pointer to the reserved units of the external repeat backtrack point, with identifier handle is returned. The pointer can be used to retrieve or modify the reserved units. It is up to the external routine to ensure proper use of the reserved space. Nothing should be written (or read) into adjacent memory. An error message is issued and a zero pointer is returned if the handle is illegal.

BIM_Prolog_repeat_iterate()

int BIM_Prolog_repeat_iterate(handle)
int *handle;

This routine iterates over the external repeat backtrack points. This function can be used in a while-loop to iterate over all external repeat backtrack points. On first invocation, the handle whose address is passed, must be initialized to 0. On each iteration, it will be set to the next external repeat backtrack point, starting from the topmost and going down the stack. As long as the function can return another backtrack point, it returns TRUE. When no more external backtrack points are available, it returns FALSE. The value of the handle is left undefined at that moment.

Example

This example shows how a function 'fun' can be applied to all external backtrack points.

```
typedef void (*Function)( int handle );
apply( Function fun )
{
   int handle;
   handle = 0;
   while( BIM_Prolog_repeat_iterate(&handle) )
      fun(handle);
}
```

BIM_Prolog_cut_repeat()

int BIM_Prolog_cut_repeat(handle)
int handle;

The external repeat backtrack point, with identifier handle is cut away. Any reserved units are deallocated as well. If a callback handler is installed with this backtrack point, it is activated immediately. The routine returns TRUE if the cut succeeded. Otherwise an error message is issued and FALSE is returned.

2.5 Example: Prolog calling externals

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 side:

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
```

The Prolog side:

The following Prolog declarations are placed in a file which is compiled and loaded

```
:- extern_load( [add] , ['add.o] ).
:- extern_function( add( real:e , real:e ) : real ).
```

The arguments are declared to be of type *evaluate*, so that expressions can be used as arguments of the function.

The use:

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 enumeration type

Suppose there is an existing library that contains a routine that returns an enumeration type value (for example 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.

The C side:

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 of BP_Atom 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() ] );
}
```

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.

The Prolog side:

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)).
```

The use:

Retrieving the error status is done as follows:

```
?- get_error_status( _Status ).
 _Status = NoError
Yes
```

Mutable parameter

As an illustration of a mutable parameter, consider the C function *strcpy()* which copies a string.

The C side:

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 output array must be large enough to hold the string, and the terminating null byte.

The Prolog side:

The Prolog declarations for this external predicate are:

```
:- extern_load( [ strcpy ], [ 'strcpy.o' ] ).
:- extern_predicate( strcpy( string : m , string : i ) ).
```

Because the first argument is declared as a *mutable string*, the argument is passed without an extra reference (contrary to *output* mode).

The use:

If the predicate is called with a free first argument, the external routine receives a pointer to a character array that is big enough to hold the largest possible Prolog atom, as the first argument.

```
?- strcpy( _String , 'Input String' ).
 _String = Input String
Yes
```

Array parameters

This example shows the use of array parameters. An external predicate that calculates the vector product of two vectors with real coordinates is defined.

The C side:

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 side:

The Prolog declarations to use this function are:

```
:- extern_load( [vector_product ],[ 'vector.o' ] ).
:- extern_predicate( vector_product(real:r, real:array:i,
                                    real:array:i, integer:i) ).
```

The use:

A possible call:

```
?- vector_product( _P, v( 1.0 , 2.0 ), v( 3.0 , 4.0 ), 2 ).
 _P = 11.0
Yes
```

The external predicate is called with two terms of arity 2 and an integer, indicating the number of elements in the terms.

Backtracking external predicate

The next example illustrates the general framework for backtrackable external predicates with a simple iterator over an external table.

An external program manages a table with names. The interface to this program provides predicates to access the table from Prolog and 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;

free_iterator( handle )
int handle;
{
     free( BIM_Prolog_get_repeat_units(handle) );
}

int table_iterate_start( pattern , iterator , identifier ,
address)
char * pattern;
Iterator * iterator;
char ** identifier;
pointer * address;
{
     Iterator iter;
     iter = ( Iterator ) malloc( sizeof(IteratorRecord) );
     if ( ! iter ) return( 1 );
     iter->index = 0;
     strcpy( iter->pattern , pattern );
     *iterator = iter;
     *identifier = TABLE_ITERATOR;
     *address = &free_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
          return( 1 );
}
```

An external iterator 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 choice point related information. It must be used to retrieve the next solution. When an iterator is invoked, with *table_iterate_start()*, a choice point identifier is also passed to the calling Prolog predicate. 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 any more, the most recent table iterator choice point 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 non-deterministic 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, pointer:o) ).
:- extern_predicate( table_iterate_next( integer:r,
           pointer:i, string:o) ).

table_pattern_match( _Pattern, _Match ) :-
     table_iterate_start( 0, _Pattern, _Iterator,
                _Identifier, _CutRoutine),
     repeat(_Handle, 'ITERATOR', 1, _Iterator, _CutRoutine ),
     table_iterate_next(_Ret, _Iterator, _Match),
     ( _Ret == 1 ->
          !
     ;
          true
     ).
```

Calling this predicate with the first argument instantiated to a pattern, will return all matching names in the second argument, one at a time by backtracking.

External Language Interface

Chapter 3
Calling Prolog Predicates From C

When using the routines and types explained in this chapter, one must include the follwing file in its source files.

```
$BIM_PROLOG_DIR/include/BPextern.h
```

3.1 Access to Prolog predicates

It is possible to call predicates that are defined in ***ProLog by BIM*** from an external routine. To do this, the predicate name and arity must be known. With this information, the predicate *handle* can be obtained. The predicate can be called using this handle and passing the desired parameters.

The following two functions can be used to retrieve the handle of a predicate.

BIM_Prolog_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 by BIM*** atom, which is returned in its internal representation. That atom is necessary to retrieve the predicate handle. The *protect* flag may be FALSE for this purpose, because the atom is only needed to retrieve the handle and may be destroyed afterwards.

BIM_Prolog_string_get_predicate()

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; 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.

BIM_Prolog_string_get_name_arity()

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: the first way returns only one solution; the second returns multiple solutions. They are used, respectively, for deterministic and non-deterministic predicates. Nevertheless, both searching for only one solution of a non-deterministic predicate and searching for different solutions of a deterministic predicate are allowed.

Single solution call (deterministic call)

The following function is used to find one single solution of a predicate with a known handle:

BIM_Prolog_call_predicate()

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 "*Calling External Routines from ProLog by BIM - 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.

BIM_Prolog_setup_call()

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 on its solutions.

If *functor* is not a legal functor 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 analogous 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, with addresses that 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.

BIM_Prolog_next_call()

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.

BIM_Prolog_terminate_call()

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.

If no iterator is active, an error message is issued and the function returns FALSE. Otherwise it returns TRUE. An iterator must be terminated in order to continue iteration over previously created iterators.

Printing messages

An external routine can display a message directly by using a C routine, or it can use the ***ProLog by BIM*** printing routine. In this way it will be compatible with the rest of the application concerning the redirection of the output stream.

BIM_Prolog_write()

BIM_Prolog_write(message)
char * message;

The routine will act the same way as the ***ProLog by BIM*** predicate **write/1**.

Printing error messages

An error message from an external routine can be printed directly on the error stream (*stderr*), but it can also go via the ***ProLog by BIM*** error printing routine. In this way it is compatible with the rest of the application where it concerns the warning switch and the redirection of the error stream.

BIM_Prolog_error_message()

BIM_Prolog_errormessage(error_message)
char * error_message;

The error message with text *error_message* is rendered on the current error stream, if the warning switch is on.

The built-in **error_raise/3** is also available as external function:

BIM_Prolog_error_raise()

int BIM_Prolog_error_raise(class , errnr , args...)
int\BP_Atom handle;
int errnr;

This function has a similar behavior as **error_raise/3**. As class argument, a (small) integer may be given, directly indicating the error class. As an alternative, an atom may be given as well. It should be one of the existing error class identifiers. The second argument must be the number of an existing error message. The rest of the arguments are taken as parameters to the error message. They must have the correct type (see also **error_raise/3**).

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 an 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

Specifier	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 runtime.

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. 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 corresponds to an array at the external side of the interface, and to a list or a flat term at the Prolog side. 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 responsibility 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. The External Language Interface will perform the allocations for the arguments.

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. The external routine is responsible for performing the necessary allocations.

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 0.5-inch wide box.

Boxes with a thick border represent the data that is actually passed as parameter (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 only exist 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 by BIM***, 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 with free(3) when no longer needed. ***ProLog by BIM*** 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

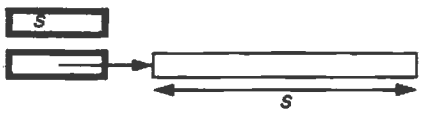
BP_T_STRING | BP_S_SIMPLE | BP_M_IN

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.
Note: 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

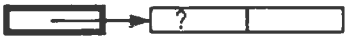
BP_T_FLOAT | BP_S_SIMPLE | BP_M_OUT

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

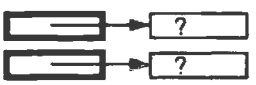
call

exit

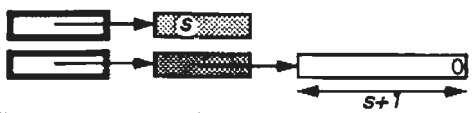
The text zone contains the string terminated with a 0 byte. It is short-term memory and should not be overwritten.

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.
Note: 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

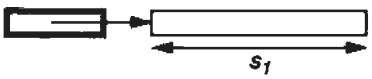
BP_T_ATOM | BP_S_SIMPLE | BP_M_MUTE

call

exit

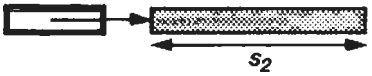
BP_T_STRING | BP_S_SIMPLE | BP_M_MUTE

call



Default for s: MAX_ATOM. The text zone must be long enough to hold the string.

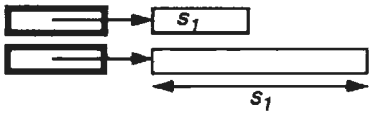
exit



The text zone contains the string, terminated with a 0 byte. And $s_2 \leq s_1$.

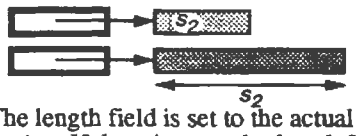
BP_T_STRINGS | BP_S_SIMPLE | BP_M_MUTE

call



The text zone must be at least s bytes long.

exit

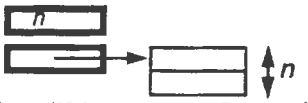


The length field is set to the actual length of the text zone that has been filled. And $s_2 \leq s_1$. If there is enough place left, the string is terminated with a 0 byte.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

Array input

BP_T_INTEGER / BP_T_LONG | BP_S_ARRAY | BP_M_IN

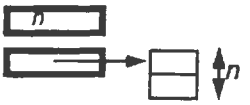
call



Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_SHORT | BP_S_ARRAY | BP_M_IN

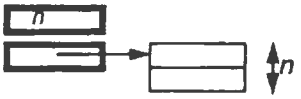
call



Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_FLOAT | BP_S_ARRAY | BP_M_IN

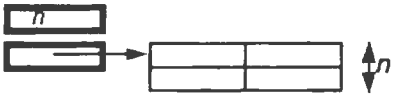
call



Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_REAL / BP_T_DOUBLE | BP_S_ARRAY | BP_M_IN

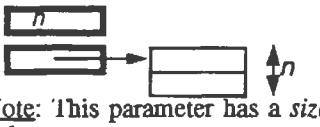
call



Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_POINTER | BP_S_ARRAY | BP_M_IN

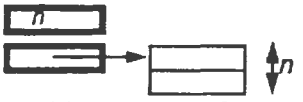
call



Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_ATOM | BP_S_ARRAY | BP_M_IN

call



Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRING | BP_S_ARRAY | BP_M_IN

call

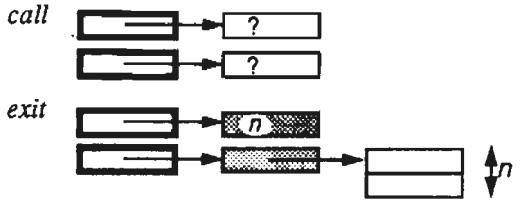
The text zones must hold the actual strings with a terminating 0 byte.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRINGS | BP_S_ARRAY | BP_M_IN

Not allowed.

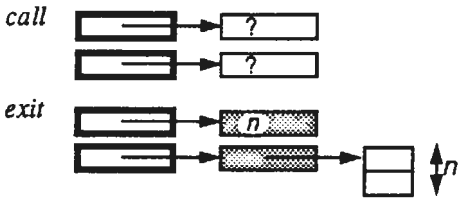
Array output

BP_T_INTEGER / BP_T_LONG | BP_S_ARRAY | BP_M_OUT



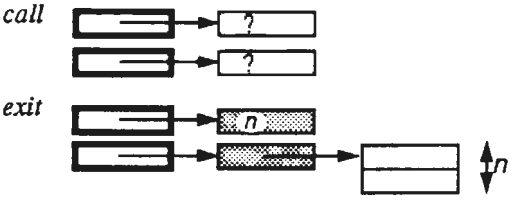
The array is long-term memory and must be freed when no longer needed.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_SHORT | BP_S_ARRAY | BP_M_OUT



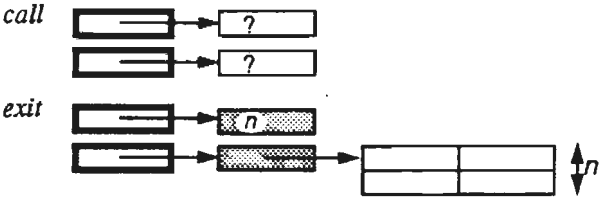
The array is long-term memory and must be freed when no longer needed.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_FLOAT | BP_S_ARRAY | BP_M_OUT



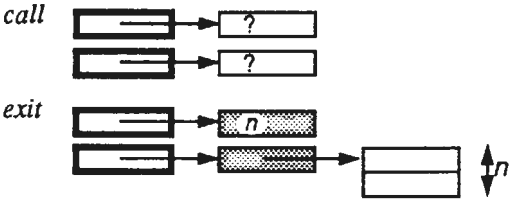
The array is long-term memory and must be freed when no longer needed.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_REAL / BP_T_DOUBLE | BP_S_ARRAY | BP_M_OUT



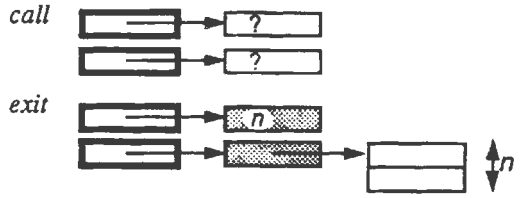
The array is long-term memory and must be freed when no longer needed.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_POINTER | BP_S_ARRAY | BP_M_OUT



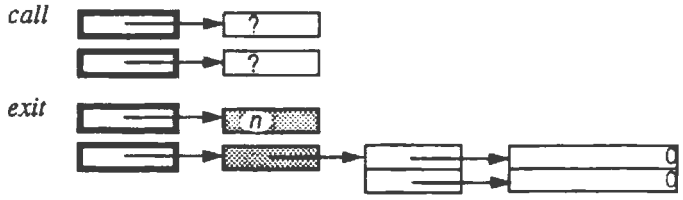
The array is long-term memory and must be freed when no longer needed.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_ATOM | BP_S_ARRAY | BP_M_OUT



The array is long-term memory and must be freed when no longer needed.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRING | BP_S_ARRAY | BP_M_OUT



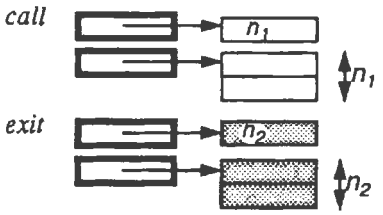
The text zones contain the strings with a terminating 0 byte. They are short-term memory and should not be overwritten. The array is long-term memory and must be freed when no longer needed.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRINGS | BP_S_ARRAY | BP_M_OUT

Not allowed.

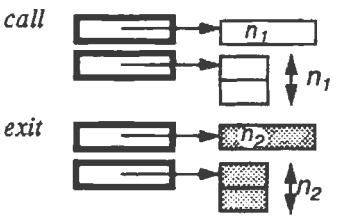
Array mutable

BP_T_INTEGER / BP_T_LONG | BP_S_ARRAY | BP_M_MUTE



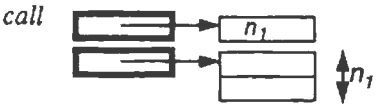
The length is set to the actual returned array length, with $n_2 \leq n_1$.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_SHORT | BP_S_ARRAY | BP_M_MUTE

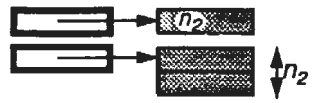


The length is set to the actual returned array length, with $no_2 \leq no_1$.
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_FLOAT | BP_S_ARRAY | BP_M_MUTE



exit

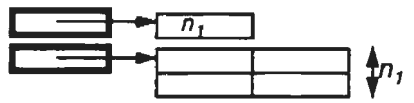


The length is set to the actual returned array length, with $n_2 \leq n_1$.

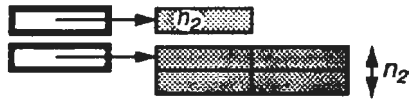
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_REAL / BP_T_DOUBLE | BP_S_ARRAY | BP_M_MUTE

call



exit

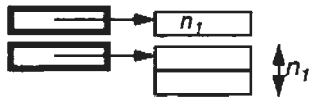


The length is set to the actual returned array length, with $n_2 \leq n_1$.

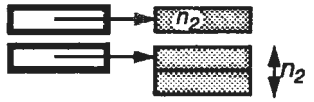
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_POINTER | BP_S_ARRAY | BP_M_MUTE

call



exit

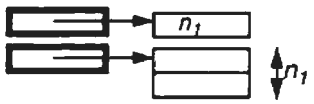


The length is set to the actual returned array length, with $n_2 \leq n_1$.

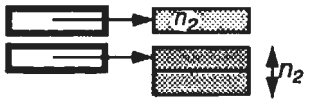
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_ATOM | BP_S_ARRAY | BP_M_MUTE

call



exit

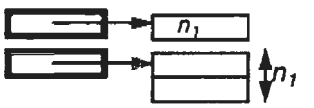


The length is set to the actual returned array length, with $n_2 \leq n_1$.

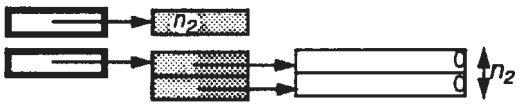
Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRING | BP_S_ARRAY | BP_M_MUTE

call



exit



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.

Note: This parameter has a *size* descriptor argument, immediately preceding the value argument.

BP_T_STRINGS | BP_S_ARRAY | BP_M_MUTE

Not allowed.

External Language Interface

Chapter 4
Simulating External Built-ins

4.1 Simulation of external built-ins

An alternative to writing external predicates is to write external routines that behave as built-in predicates. These have the advantage of being faster to access, but the disadvantage of being less flexible and less robust.

It is not allowed to call ***ProLog by BIM*** predicates from within an external built-in routine. It is the programmer's responsibility to ensure this.

A second consequence is, that the external routine itself is responsible for retrieving the parameters from the ***ProLog*** argument registers and instantiating them accordingly (using provided external functions).

External built-in predicates cannot fail on return due to a failed unification of output or return parameters. The external routine must return a status value being TRUE for success and FALSE for failure. One can make the external predicate succeed/fail depending on the unifications of its arguments, by returning the logical AND of all unification results.

To define external built-ins, one of the following three directives or built-ins can be used.

extern_builtin/3

:- extern_builtin(_Language, _ExternalName, _PredDecl)

extern_builtin(_Language, _ExternalName, _PredDecl)

arg1 : ground : atom
arg2 : ground : atom
arg3 : ground : term of the form _Name / _Arity

The external routine with external name *arg2*, and written in language *arg1*, is mapped to a ***ProLog*** predicate as described in *arg3* which is of the form *_Name / _Arity*. The routine *arg3* is treated as an external built-in.

extern_builtin/2

:- extern_builtin(_Language, _PredDecl)

:- extern_builtin(_ExternalName , _PredDecl)

extern_builtin(_Language , _PredDecl)

extern_builtin(_ExternalName , _PredDecl)

arg1 : ground : atom
arg2 : ground : term

Same as **extern_builtin/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_builtin/1

:- extern_builtin(_PredDecl)

extern_builtin(_PredDecl)

arg1 : ground : term

Same as **extern_builtin/3**, but with the default language and where the external name is the same as the ***ProLog*** predicate name.

4.2 Access of argument registers

To access and unify the argument registers passed to an external built-in, a set of routines is provided. Files in which the external routines and special types are used, must include the following file:

```
$BIM_PROLOG_DIR/include/BPextern.h
```

Argument retrieval

The following functions retrieve the values of the argument registers and return them as a ***ProLog by BIM*** term.This term can further be handled as if it is a **BP_Term** type parameter.

BIM_Prolog_get_argument()

BIM_Prolog_get_argument(nr , term)
int nr;
BP_Term *term;

Retrieves the *arg1*'th argument into the address *arg2*. No range checking is performed on the argument number.

BIM_Prolog_get_argument_list()

BIM_Prolog_get_argument_list(term1 { , termi } , 0)
BP_Term *termi;

This routine has a variable number of arguments. The routine retrieves a list of argument registers, starting from register 1 and stores the values in the addresses given as *arg1*, arg2, .. up to the last argument, before the terminating 0.

Type and value retrieval

The type and value of a ***ProLog by BIM*** argument can be retrieved directly with the following set of functions.

BIM_Prolog_get_argument_type()

int BIM_Prolog_get_argument_type(nr)
int nr;

Returns the *arg1*'th argument's type. No range checking is performed on the argument number.

BIM_Prolog_get_argument_value()

int BIM_Prolog_get_argument_value(nr , type , value)
int nr;
int type;
union *value;

Retrieves the *arg1*'th argument, after conversion to type *arg2* into the address *arg3*. In normal cases, TRUE is returned. If the argument is incompatible with the indicated type, an error message is issued and FALSE is returned. No range checking is performed on the argument number.

BIM_Prolog_get_argument_type_value()

BIM_Prolog_get_argument_type_value(nr , type , value)
nt nr;
int *type;
union *value;

Retrieves the *arg1*'th argument. Its type is stored in the address *arg2* and its converted value in the address *arg3*. No range checking is performed on the argument number.

BIM_Prolog_get_argument_value_integer()

BIM_Prolog_get_argument_value_real()

BIM_Prolog_get_argument_value_atom()

BIM_Prolog_get_argument_value_string()

BIM_Prolog_get_argument_value_pointer()

int BIM_Prolog_get_argument_value_<type>(nr , value)
int nr;
<C-type> *value;

Retrieves the *arg1*'th argument, after conversion to the type as specified in the routine name, and stores the value into the address *arg2*. If this succeeds, TRUE is returned. If the argument is incompatible with the indicated type, an error message is issued and FALSE is returned. No range checking is performed on the argument number.

The correspondence between the ***ProLog*** type and <C-Type> is given in the table below.

integer	int
real	double
atom	BP_Atom
string	BP_String
pointer	unsigned int

Argument unification

The following set of unification functions are provided for the ***ProLog by BIM*** argument registers.

BIM_Prolog_unify_argument_value()

int BIM_Prolog_unify_argument_value(nr , type , value)
int nr;
int type;
union value;

Unifies the *arg1*'th argument with the value *arg3* of type *arg2*. It returns TRUE if unification succeeded and FALSE if it failed. No range checking is performed on the argument number.

BIM_Prolog_unify_argument_value_integer()

BIM_Prolog_unify_argument_value_real()

BIM_Prolog_unify_argument_value_atom()

BIM_Prolog_unify_argument_value_string()

BIM_Prolog_unify_argument_value_pointer()

int BIM_Prolog_unify_argument_value_<type>(nr , value)
int nr;
<C-Type> value;

Unifies the arg1'th argument with the value arg2 of type as specified in the routine name. It returns TRUE if unification succeeded and FALSE if it failed. No range checking is performed on the argument number.

The correspondence between the ProLog type and <C-Type> is given in the table above.

BIM_Prolog_unify_argument_term()

int BIM_Prolog_unify_argument_term(nr , term)
int nr;
BP_Term term;

Unifies the *arg1*'th argument with the term *arg2*. The routine returns TRUE if unification succeeded and FALSE if it failed. No range checking is performed on the argument number.

External Language Interface

Chapter 5
External Manipulation of Prolog Terms

5.1 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 a handle that represents the term on the system's heap. The data type for such a handle is *Term*. The term should only be manipulated using the external ***ProLog by BIM*** routines.

To make use of the routines for external manipulation of Prolog terms, one must include the following file:

```
$BIM_PROLOG_DIR/inlcude/BPextern.h
```

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 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 terms is the list, which has two arguments, and whose functor is always **./2**.

In order to manipulate terms and subterms, 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 by BIM*** 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

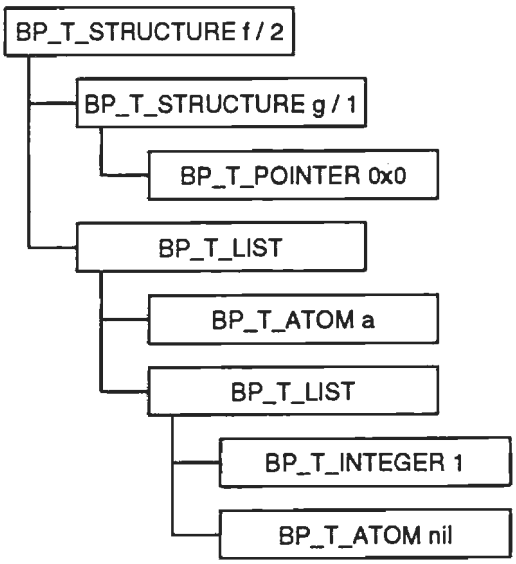
The table below gives a list of all defined term types (base and variants), with the corresponding C data type and a description of the value of a term of that type.

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

Note: The number of a variable is the same number as 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])



5.2 Term decomposition

The standard method to decompose a term, is

first identify its type,

and then retrieve its value.

An example can be found at the end of this chapter.

Retrieving the type of a term

BIM_Prolog_get_term_type()

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 valid term, an error message is issued and the special value BP_T_ILLEGAL is returned.

Retrieving the value of a term

BIM_Prolog_get_term_value()

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.

Note: 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 they should not be overwritten. It is not certain that this string will remain at the same memory address during the entire program lifetime. As a result, if the text is needed by the external routines for a longer period, it should be copied to external private memory.

BIM_Prolog_get_term_type_value()

int BIM_Prolog_get_term_type_value(term, type, value)
BP_Term term;
int *type;
union *value;

Retrieve the type and value of <term>. The type is stored in the address <type> and the converted value in the address <value>.

Retrieving an argument of a term

BIM_Prolog_get_term_arg()

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.

5.3 Term construction

A term can be constructed in the external routines in two main 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 consumption may cause 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 *"Life time of terms - Protecting a term"*)

For an example see at the end of this chapter.

Ensuring heap space for constructing a term

BIM_Prolog_term_space()

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

BIM_Prolog_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

BIM_Prolog_unify_term_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 succeeds, 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 *"Representation of terms"* for a correspondence 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 free variables as its arguments.

Unifying two terms

BIM_Prolog_unify_terms()

int BIM_Prolog_unify_terms(term1 , term2)
BP_Term term1;
BP_Term term2;

The two terms *term1* and *term2* are unified. If unification succeeds, 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.

Instantiation of a term of a compound function

BIM_Prolog_unify_argument_term()

BIM_Prolog_unify_argument_term(term , nr , type , value)
BP_Term term;
int nr;
int type;
union value;

The <nr>th argument with the term <term> is unified with a value <value> of type <type>. Returns TRUE if unification succeeded. Otherwise it returns FALSE and <term> is left unchanged. For further details, refer to BIM_Prolog_unify_term_value().

5.4 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 subterm 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 "*Term Construction*").

Protecting a term

BIM_Prolog_protect_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

BIM_Prolog_unprotected_term()

BP_Term BIM_Prolog_unprotect_term(term)
BP_Term term;

The term *term* is made 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.

Testing term protection

To check whether a term is a protected term or not, the following test function can be used:

BIM_Prolog_is_protected_term()

int BIM_Prolog_is_protected_term(term)
BP_Term term;

If the argument *term* is a protected term, the function returns TRUE. Otherwise, it returns FALSE.

An iterator can be used to loop over all protected terms. This is useful for debugging purposes.

During iteration, no terms should be protected or unprotected. Such operations might confuse the iterator. An iterator must externally be declared as a variable of type BP_Iterator. It must be initialized before usage.

BIM_Prolog_protected_terms_init()

BIM_Prolog_protected_terms_init(iterator)
BP_Iterator *iterator;

The iterator whose address is passed as argument, is initialized.

BIM_Prolog_protected_terms_next()

int BIM_Prolog_protected_terms_next(iterator)
BP_Iterator *iterator;

The next protected term of the iterator whose address is passed as argument, is returned. If all protected terms have been handled by this iterator, 0 is returned.

5.5 Conversion of simple terms

A number of external functions is provided for converting between different representations of ***ProLog by BIM*** internal data.

Conversion from string to atom

BIM_Prolog_string_to_atom()

BP_Atom BIM_Prolog_string_to_atom([protect ,] string)
int protect;
BP_String string;

BIM_Prolog_mod_string_to_atom()

BP_Atom BIM_Prolog_mod_string_to_atom([protect,] string , module)
int protect;
BP_String string;
BP_String module;

The null-terminated character array *string* is converted to a ***ProLog by BIM*** atom (with module qualification *module*, for BIM_Prolog_mod_string_to_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 by BIM***. 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 by BIM***. 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 by BIM***.

Conversion from sized string to atom

BIM_Prolog_strings_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 by BIM*** 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

BIM_Prolog_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

BIM_Prolog_save_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 by BIM***, 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 by BIM***'s string tables.

5.6 Example: term decomposition

The C routine *print_term()*, defined below, prints out the term it receives from ***ProLog by BIM***, in the same style as the built-in predicate **write/1**.

To use it from ***ProLog by BIM***, 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) == 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;
            }
            if ( BIM_Prolog_get_term_value (term, BP_T_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.

5.7 Example: 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 by BIM*** 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, BP_T_STRUCTURE,
                &functor) || functor != attribute_3 )
      goto get_attribute_exception;
   BIM_Prolog_get_term_arg(term, 1, &arg1);
   if ( ! BIM_Prolog_get_term_value(arg1, BP_T_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, BP_T_INTEGER,
                int_val);
   else if ( look_up_attribute_real(attr_name, &real_val) )
      BIM_Prolog_unify_term_value(arg2, BP_T_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.

External Language Interface

Chapter 6
Manipulation of External Data

The predicates in this chapter provide easy access from Prolog to data created in an external language. Data structures that are created and managed in Prolog can also easily be accessed by an external language program.

If performance is a key issue, one is advised not to use the high-level predicates.

6.1 External type definition

An external type is specified as described by the following rules:

<TypeSpec>	=>	<TypeInstance> | <BasicType> | <ComplexType>
<TypeInstance>	=>	<TypeID>
<BasicType>	=>	**integer | short | long | real | float | double | char | atom | pointer**
<ComplexType>	=>	<StructureType> | <UnionType> | <ArrayType> | <PointerType>
<StructureType>	=>	**structure**(<FieldName>- <TypeSpec> { , <FieldName> - <TypeSpec> })
<UnionType>	=>	**union**(<FieldName> - <TypeSpec> { , <FieldName> - <TypeSpec> })
<ArrayType>	=>	**array**(<size> , <TypeSpec>)
<PointerType>	=>	**pointer**(<TypeSpec>)
<TypeID>	=>	<atom>
<FieldName>	=>	<atom>
<Size>	=>	<integer>

The following predicate defines a type.

extern_typedef/2

extern_typedef(_TypeID, _TypeSpec)

arg1 : ground : TypeID
arg2 : ground : TypeSpec

A type is defined with identifier *arg1* and specification *arg2*. The specification may still be incomplete. It may refer to type identifiers that are not yet defined.

6.2 External variable allocation

The following two predicates allocate and deallocate external variables.

extern_allocate/2

extern_allocate(_VarID, _Type)

arg1 : ground : atomic
free : pointer
arg2 : ground : TypeID or TypeSpec

An external variable of type *arg2* is allocated. If *arg1* is ground, the variable is associated to that name. If it is free, it is instantiated to the address of the external variable.

The type *arg2* must be completely defined.

This predicate fails if the variable could not be allocated.

6.3 External variable manipulation

extern_deallocate/1

extern_deallocate(_VarID)

arg1 : ground : atom or pointer

The external variable with identifier *arg1* is deallocated.

A field of an external variable is specified as described by the following rules:

<VarSpec>	=>	<VarID> | <VarID> <VarSelection>
<VarSelection>	=>	^ <VarSelector> |
		^ <VarSelector> <VarSelection>
<VarSelector>	=>	<FieldName> | <Index>
<Index>	=>	<integer>
<VarID>	=>	<atomic>

The *VarSelection* must be consistent with the type of the variable.

A *FieldName VarSelector* selects one of the fields of a *StructureType* or *UnionType*, while an *Index VarSelector* selects an element of an *ArrayType* or dereferences a *PointerType* with a given offset. The first element of an array has index 0.

An external variable can be assigned a new (partial) value, its value can be retrieved, and its address or type can be retrieved.

extern_set/2

extern_set(_VarSpec, _Value)

arg1 : ground : VarSpec
arg2 : ground : atomic

The field specified by *arg1* is assigned the value *arg2*. The variable defined by the VarID of *arg1* must have been allocated in Prolog.

extern_set/3

extern_set(_Type,_VarSpec, _Value)

arg1 : ground : TypeID or TypeSpec
arg2 : ground : VarSpec
arg3 : ground : atomic

The VarID of *arg2* is cast to type *arg1*. The field specified by *arg2* is assigned the value *arg3*.

extern_get/2

extern_get(_VarSpec, _Value)

arg1 : ground : VarSpec
arg2 : free : atomic

The value of the field specified by *arg1* is retrieved and unified with *arg2*. The variable defined by the VarID of *arg1* must have been allocated in Prolog.

extern_get/3

extern_get(_Type,_VarSpec, _Value)

arg1 : ground : TypeID or TypeSpec
arg2 : ground : VarSpec
arg3 : free : atomic

The variable of *arg2* is cast to type *arg1*. The value of the field specified by *arg2* is retrieved and unified with *arg3*.

extern_address/2

extern_address(_VarSpec, _Address)

arg1 : ground : VarSpec
arg2 : free : pointer

The address of the field specified by *arg1* is calculated and unified with *arg2*.

extern_type/2

extern_type(_VarSpec, _Type)

arg1 : ground : VarSpec
arg2 : free : BasicType

The type of the field specified by *arg1* is looked up and unified with *arg2*.

6.4 External memory allocation

A set of lower level predicates are provided for allocation and deallocation of external memory blocks.

extern_malloc/2

extern_malloc(_Address, _Size)

arg1 : free : pointer
arg2 : ground : integer

A block of external memory of *arg2* bytes is allocated. *Arg1* is instantiated to its address.

This block may be deallocated with **extern_free/1**.

extern_free/1

extern_free(_Address)

arg1 : ground : pointer

The block of memory at address *arg1* is deallocated. It should have been allocated before with **extern_malloc/2**.

Nothing is tested as to whether this constraint is satisfied.

6.5 External memory access

Direct external memory access is possible with the following lower level predicates. The indicated type determines the type of the external data at the given address, and how it must be converted to ***ProLog by BIM*** data.

Type	External	Prolog
integer	int	integer
short	short	integer
long	long	integer
real	float	real
float	float	real
double	double	real
pointer	unsigned int	pointer
atom	BP_Atom	atom
char	char	integer

extern_peek/3

extern_peek(_Type, _Address, _Value)

arg1 : ground : atom
arg2 : ground : pointer
arg3 : any : atomic

The value of type *arg1* is retrieved from the location at address *arg2*, and unified with *arg3*.

extern_poke/3

extern_poke(_Type, _Address, _Value)

arg1 : ground : atom
arg2 : ground : pointer
arg3 : ground : atomic

The value *arg3* is stored as an entity of type *arg1* in the location at address *arg2*.

6.6 Example: manipulating external data

In this example, an external routine is defined that fills an array with the powers of a specified root. The creation of and access to the array are done in Prolog.

The Prolog code in file *powers.pro* is:

```
:- extern_load([init_powers], ['powers.o']).

:- extern_predicate( init_powers(pointer:i) ).

program :-
   {* Declare the structure of a series of powers. *}
   extern_typedef( powers, structure( root  - integer,
      power - array( 20 , integer )) ),

   {* Allocate and initialize the structure
        for the powers of 2.  *}
   extern_allocate( powers_of_2 , powers ),
   extern_set( powers_of_2^root , 2 ),
   extern_address( powers_of_2 , _Address ),
   init_powers( _Address ),

   {* Retrieve the value of the third power of 2
     with extern_get/3 *}
   extern_get( powers, powers_of_2^power^3, _ValuePower3 ),
   write( 'Value of 2 ** 3 with extern_get/3 ':
         _ValuePower3 ), nl, nl,

   {* Retrieve the value of the tenth power of 2
     with extern_peek/3*}
   extern_address( powers_of_2^power^10,_AddressPower10),
   extern_peek( integer, _AddressPower10 , _ValuePower10 ),
   write( 'Value of 2 ** 10 with extern_peek/3' :
         _ValuePower10 ), nl, nl,

   {* Deallocation of the structure for the powers of 2.  *}
   extern_deallocate( powers_of_2 ).

?- program .
```

The C code for the routine *init_powers()* in file *powers.c* is:

```
typedef struct {
   int root ;
   int power[20];
      /* series of the powers of the given root */
} PowerSeries;

init_powers( instance )
PowerSeries * instance;
{
int i, acc ;
   acc = (*instance).power[0] = 1 ;
   for (i=1; i < 20 ; i++)
      acc = (*instance).power[i] = (*instance).root * acc ;

   return(0) ;
}
```

Execution of the program gives the following result:

```
Value of 2 ** 3 with extern_get/3 : 8

Value of 2 ** 10 with extern_peek/3 : 1024
```

Debugger

Debugger

Chapter 1
Functionalities

To make the development of Prolog programs easy, the ***ProLog by BIM*** system includes an advanced programming environment. This debugging environment is based on state-of-the-art Prolog debugging techniques, and exploits all features of currently available multi-windowing and high resolution screen supporting workstations, which increases ease of use and improves the productivity of the ***ProLog by BIM*** programmer.

Debugging strategies

ProLog by BIM incorporates several debugging strategies:

The execution-time debugging package allows to follow and investigate a running program. It encorporates two debugging techniques. Trace-oriented debugging, based on the box model, focusses on the ports of the active predicates which are represented in trace lines. Source-oriented debugging improves 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 debugging strategy is the "post-execution analysis". After the execution in debug mode has produced an incorrect result or fails, the trace information can be analyzed. 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-execution debugging facility.

Debugging in the window environment

The multi-windowing programming environment integrates the ***ProLog by BIM*** system and its debugger with a view on the source text, 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. For example: its contents can be saved for later inspection. Interaction with the system is not restricted to typing commands. Whenever possible, appropriate buttons are provided, so that actions require as little manipulation as possible. The current ***ProLog by BIM*** session, the original source, error messages, data files, and all control panels are readily available and tightly interlinked. More about debugging in the windowing environment can be found in "*Windowing Environment*" and in the "*User's Guide*".

Experience with similar multi-window programming environments for conventional procedural languages has shown a considerable positive impact on achieving higher productivity and higher quality of end-user applications.

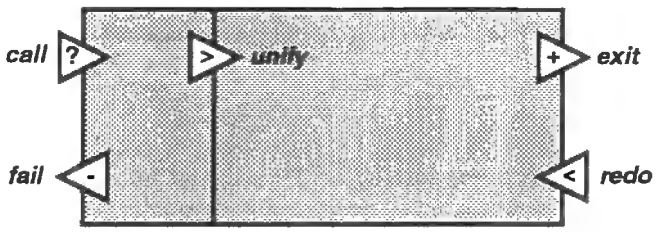
Debugger

Chapter 2
General Concepts of the Debugger

Debugger interaction from the engine is performed with a number of built-in predicates. The user can interact with the debugger from the engine top level or from the debugger command level. Interacting from the debugger command level is performed through commands. Control can be passed from the engine to the debugger and back. The debugger is based on a five-port box model. Each port can be traced and it is possible to interact with the debugger at each port. One can step from one port to another in the execution of a query or take larger steps to advance more rapidly. This chapter explains the box model, how one can compile a predicate into debug code and some general interaction with the debugger.

2.1 Box model

A box with five ports is associated with each predicate. 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 where the debugger information can be displayed. It is possible to indicate at which ports information is required, and even to halt execution at certain ports.



The 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; for example, 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.

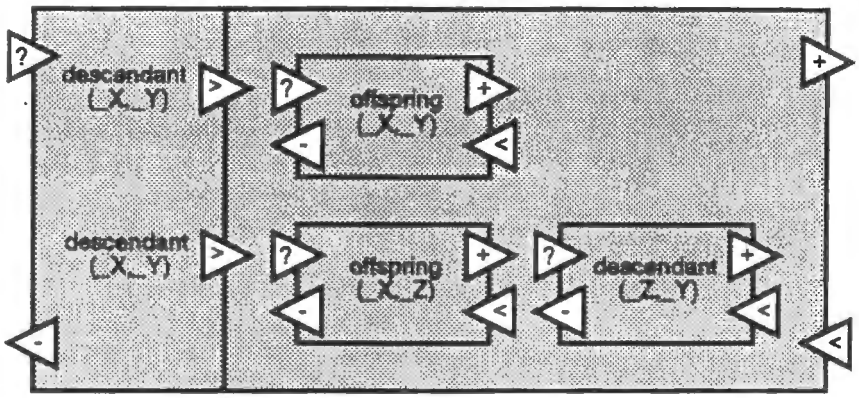
Whenever a port name must be given, one can choose between the full name or an abbreviation, which is the first letter of the name.

A small example is used to give a graphical representation of the layout of different boxes.

```
descendant( _X , _Y ) :-
     offspring( _X , _Y ) .
descendant( _X , _Y ) :-
     offspring( _X , _Z ),
     descendant( _Z , _Y ) .
```

The box for the top most invocation of descendant/2, together with the boxes for its subgoals, are shown in the next figure.

Lay-out of boxes for nested and successive subgoals.



The boxes for the subgoals in both clauses are positioned inside the box for the top invocation of descendant/2. The boxes for the two successive subgoals in the second clause are positioned one after the other This positioning also suggests how the ports are coupled to each other. It does not show all couplings. This would be too difficult in a two-dimensional picture.

2.2 Preparing a program for debugging

Before a predicate can be debugged, it must be compiled to debug code. Debug code 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 of other code.

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 interpreted and acts 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. However, this can be confusing when following the execution or analyzing the trace afterwards.

Compiling a complete file for debugging

There are several ways to compile a predicate to debug code. One is to compile the complete file containing the predicate into debug code. The effect is that all predicates defined in the file, are compiled to debug code.

When a file is explicitly compiled with BIMpcomp one has to specify the -d+ option

```
% BIMpcomp -d+ file.pro
```

will compile all the predicates defined in file.pro into debug code

With **compiler/2** one can switch the compiler debug flag on: all files which are compiled from within the engine will be compiled into debug code.

```
?- compiler(d,on).
Yes
?- compile(file).
compiling -d+ file.pro
Yes
```

By specifying -d+ in the compile and consult predicates one can compile one particular file into debug code.

```
?- consult(file, '-d+').
compiled -d+ file.pro
loaded f.pro

?- compile(file,'-d+').
compiled -d+ file.pro
Yes
```

Compiler directives for debugging

Using compiler directives in the source file offers better 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 directive is provided for this purpose:

option/1

:- option(_CompOption)

If *arg1* equals "d+", then the compiler generates debug code generation for the predicates which follow. If *arg1* equals "d-", then the compiler doesn't generate debug code generation for the predicates which follow.

Note that this predicate can also be called with other values for *arg1*. The complete explanation of **option/1** is given in "*Directives - General directive*".

To compile a complete file to debug code using this method, it is sufficient to place a single '**:- option(['d+']).**' directive at the beginning of the file.

Interactive definition of debug predicates

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 built-in **please/2**:

```
?- please (d, on).
```

All interactively entered clauses for the same predicate must be compiled to the same type of code.

2.3 Debugger interaction from within the engine

One can interact with the debugger from the top level of the engine by using one of the following built-ins.

Issuing debugger commands from within the engine

debug/1

debug(_Command)

arg1 : ground : atom or list of atoms

The atom (or the list) *arg1* is treated as a (list of) 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).

Debugger options

The debugger has a certain number of options which can be set from within the engine.

The set of debug switches can be divided into two categories: binary switches and integer switches. Each category is explained below. For each switch its abbreviated name, its full name and its default value are given. To know the current values of the different debug switches, execute the query '?- debug(_,_).'.

Debug options with a binary value are:

c : cutearly on
Controls the choice point destruction under debugger execution.
When on, choice points are cut away immediately. When off, choice points are only marked when cut. Upon backtracking this is mentioned as a 'fail due to cut'.

i : indexed on
Controls the usage of indexing in debug mode.
When on, clauses for which unification failure can be determined with indexing, will not be tried and will not be visible in the trace. Otherwise, all the clauses will be tried.

p : prompt on
Indicates whether a command must be requested after activation of the debugger and after the loading of a file. When on, a prompt for command is given.

tr : tracerecord on
Specifies whether a trace must be recorded, during a keep-trace execution. Using this option, the recording of the trace can be suspended temporarily.

wm : writemodule off
Controls explicit module qualification in debugger output.

wp : writeprefix off
Determines how operators are printed in the debugger output
If writeprefix is on, operators will be written in regular functor notation. When writeprefix is off, operators will be written out according to their specification.

wq : writequotes off
Controls output of quotes in debugger output.

Debug options with an integer value are:

td : tracedepth 10000
Defines the allowed depth of the recorded trace.
When the integer is a negative value, the limit is removed.

wd : writedepth -1
Controls the printing of nested terms in the debugger output.
The printing of a structured term is limited to the specified levels of nesting. All sub-terms of that level are printed as "...". When the integer is a negative value, the limit is removed.

The default settings of these options can be changed from the "*Debugger Control Window*' of the monitor (See "*Windowing Environment*"), with certain debugger commands (see further) or with the debug/2 built-in.

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*. If both arguments are free, the current values of debug options are printed to the current output stream.

2.4 General debugger commands

Help can be requested from the debugger with the following commands.

help (h or ?)

Prints an overview of available commands.

help *command* (h or ?)

Prints more information for <command>.

The appearance of the window debugger can be modified with the following commands. For more information regarding the window debugger see "*Windowing Environment*".

button *command*

Appends a button for <command> in the debugger window.

menu *command*

Appends a menu item for <command> in the debugger window.

unbutton *command*

Deletes the button for <command> in the debugger window.

unmenu *command*

Deletes the menu item for <command> in the debugger window.

The engine can be called from the debugger command line:

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, for example, to request a listing, or to set or remove spy points during program execution.

From within the debugger, the control can be returned to the engine.

run

Leaves debugger command mode and returns control to the engine allowing the user to invoke the predicate to be debugged.

Quitting the debugger can be done with the following command:

quit (q)

Aborts and returns to the top level of the system.

Defining new 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. The <modif> allows modification of the type.

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 <modif>, being one of:

- *optional*: the argument is optional.

  If no actual argument is present for an optional argument, the empty list is passed as the argument.
- *list*: the command can have several (at least one) arguments of the same type, separated by blanks or commas.

  A list of arguments is passed to the predicate as a Prolog list.

Without argument, command prints a list of user-defined commands.

With only one argument, any existing definition for that command is destroyed.

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.

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'
```

Debugger

Chapter 3
Execution-Time Debugging

The execution-time debugging is based on a five-port box model. It is possible to step from one port to another in the execution of a query. One can also take larger steps to advance more rapidly. Investigation of the state of the execution by the user is possible during this stepping.

3.1 Trace-oriented versus source-oriented

Debugging in a **trace-oriented** way is possible by using spy points. This debugging gives a trace of ports of the predicates. Several built-in predicates are provided for setting and removing spy points. A **spy point** on a port of a predicate is an indication to the system that it must suspend the control flow at that port. The user can select whether a port must be shown or not. A trace line will be printed out for all the shown ports. A port can be leashed or not. An unleashed port is shown, but execution continues. If the port is leashed, the system halts execution and waits for a user command.

Programs can also be debugged in a **source-line oriented** fashion, instead of using the trace-oriented approach. This means that the debugging process follows the structure of the program as it was written. Source-oriented debugging is based on program lines rather than on ports. **Break points** can be set on program lines and indicate to the system that the control flow must be suspended when that line is reached. A break point is similar to a leashed spy point. The execution always stops at a break point and the corresponding line of the program source is printed out.

The source-oriented debugger and trace-oriented debugger are tightly integrated. Both methods can be mixed. Both source-oriented and trace-oriented commands can be executed from a break point or from a leashed spy point. The command determines where the execution will be halted and what will be shown at that point.

3.2 Invoking, interrupting and leaving the debugger

The following built-in predicates are used to invoke and leave the debugger. If they are called from a predicate, they will only affect the queries following the command, and not the current one. From the moment the debugger is started, all subsequent queries are executed in debugging mode.

The **debug/0** and **trace/0** predicates initially switch to debugger command mode (as on a leashed port). In this mode the system waits for break point setting, a request for information, or the **run** command which will return the control to the engine and allow the user to invoke the predicate to be executed.

trace/0

trace

Invokes the debugger and starts it in full tracing mode. The **run** command here has the same effect as **creep** during a suspended execution.

debug/0

debug

Invokes the debugger without tracing. This assumes spies or break points have been placed on the ports or source lines. The execution of the program will be suspended when one of those points is encountered. The **run** command has the same effect here as **nextp** during suspended execution.

nodebug/0

nodebug

notrace/0

notrace

Both predicates turn the debugger off.

Interrupting a running program

It is possible to *interrupt* a running program if the debugger is active. When interrupted, a choice is given to either terminate the execution (for example: 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 *"Built-in Predicates - Signal handling"* on how to change the default handler (the default handler returns to the top level with an error message).

3.3 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 compiled to debug code. Parts of it can be omitted in the execution-time debugger by using a command that skips certain ports, and during trace recording by simply setting off the recording.

The trace line

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 that 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 (for example: 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, 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.

The predicate

Writing out the arguments of a predicate is done according to the settings of the writedepth, writemodule, writeprefix and writequotes debugger switches.

These settings can be controlled by the debug/2 predicate or by the following commands:

Depth *n* (D)

Sets the writedepth for the debugger to <n> levels. Its default value is 10. This is analogous to **debug(writedepth, n)**.

Module (M)

Switches the toggle for the printing of module qualifications in trace lines. This is analogous to **debug(writemodule,_)**.

Prefix (P)

Switches the toggle for the use of operators in printing trace lines. This is analogous to **debug(writeprefix,_)**.

Quotes (Q)

Switches the toggle for the usage of quotes in trace lines. This is analogous to **debug(writequotes,_)**.

Trace (T)

Sets recording of the trace on or off. Allows in corporation with the post-execution debugger (see also "*Post Execution Debugger - Trace recording control*").

The port

The symbols that identify the type 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 built-in predicates, the second character is the same as the first.
- For user-defined predicates or built-ins defined in Prolog, a possible second character gives some extra information.

The 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 built-in
++	exit of a built-in
--	fail of a built-in
<<	redo of a built-in
!!	fail of a built-in because alternatives are cut away

Finally, the alternatives of an OR-list are indicated with ";".

3.4 Trace-oriented debugging

Several built-in predicates are provided for controlling spy points in the debugger.

Setting and removing spy points

Normally, when setting a spy, one must indicate on what predicate it must be set and on which ports of its box. Since one frequently wants to set spies on the same set of ports for several predicates, it is possible 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 the debugger is left, the same spy points remain set and become active again when the debugger is reinvoked in the same session.

showspy/0

showspy

Prints the current spy points 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

showspydefault

Prints the current spydefault setting on the current output stream.

spy/0

spy

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 spy points 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 spy-points 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 can be abbreviated to their first letter. Any existing spy-points are left unchanged.

nospy/0

nospy

Removes the spy points 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 spy points 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 spy points from all ports indicated in *arg2* of the predicates given in *arg1*. The predicates must be given as a 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 to be viewed during debugging. The non-selected ports will never be shown and execution will never be suspended at these 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

showleash

Prints the currently leashed ports on the current output stream. The call, unify, and redo ports are leashed by default.

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.

Trace-oriented commands

When execution is suspended in the trace-oriented debugger 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, as specified between brackets after each command. A sequence of commands can be given on the same command line separating them with " ; ".

The default command, which is assumed when the empty string is entered, depends on whether the predicate is user-defined or built-in. For user-defined predicates the **creep** command is executed, while for built-ins the **go on** command is used.

creep (c)

Resumes execution with full tracing until the next leash port.

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 when the user only wants to see the call port of predicates (for example: built-ins). 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 spy point or the next port of the same box, whichever comes first.

nextp (n)

Resumes execution without tracing until the next spy point.

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 appears when the output ports are not leashed, there the debugger continues in skip mode. This command entered on an exit or fail port has the same effect as **creep**.

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.

3.5 Source-oriented debugging

When execution is suspended in the source-oriented debugger the user has to give a command to determine how execution should continue.

Source-oriented commands

The following set of commands is available for source-oriented debugging.

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 "_".

The **step** command is used as default command. It is executed when the empty string is entered,

back

Goes back to the previous level in the execution tree.

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.

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.

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> is printed out. This variable must be defined in the environment of the predicate that is being displayed. 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 spy point 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 spy point to become a break point.

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.

Debugger

Chapter 4
Post-Execution Debugging

Post-execution debugging allows to record a trace of the execution and to investigate it afterwards.

4.1 Trace recording control

Before starting **post-execution debugging**, the query must have been executed and its trace recorded. This can be done in conjunction with the stepping debugger or without tracing anything during execution. Afterwards, the recorded trace can be analyzed.

The debugger does not usually record the trace of a program. To do so, one must indicate this before entering the query, in which case the following query (and only the first one) will have its trace recorded. A recorded trace can be analyzed 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 analyze the trace with the built-in analysis algorithm.

Another method to suppress 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 analyze the trace with the built-in analysis algorithm. When _x is negative, the limit on the depth is removed.

The following predicate controls the trace recording process:

keeptrace/0

keeptrace

A trace of the execution of the following query will be recorded. After completing the query execution, **analyze/0** is automatically activated.

See also "*Execution-Time Debugging - Controlling the stepping debugger*" (actions on leashed ports) in this part.

4.2 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 built-in predicate.

Zooming trough a trace

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.

Algorithmic debugging

The analysis algorithm is invoked with:

analyze/0

analyze

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. The head of each considered goal predicate is displayed with 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 other solutions. By default, for succeeded subgoals only the solution is investigated and not the failures.

Commands for the post-mortem-debugger

Interaction with the post-execution debugger is possible by using one of the following commands.

Commands can be aliased with **alias.** New commands can be defined by the command **command**. A sequence of commands can be given on the same command line separating them with " ; ". See also "*Debugger - General Concepts - Overall debugger control*".

advance (a)

If the current goal predicate has failed, 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 through solutions or failures. 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.

back (b)

Leaves this level and goes back to the caller. On the top level (the query), this command is ignored.

call (c)

The goal predicate will be displayed at its call port from now on. This means that the head and all subgoals will have their arguments unified as they were just before the call of the goal. If for example, 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 more detailed output for the goal predicate. Both the call port and the exit port of the subgoals (and the head) are displayed. 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 displayed as 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 a command. If all subgoals succeeded, nothing happens.

Failures (F)

If a subgoal has succeeded, the analyzer will only display its solution when the subgoal is investigated. You may want to see the failures that occurred before the solution was found. This command indicates to the analyzer whether to investigate previous failures.

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)

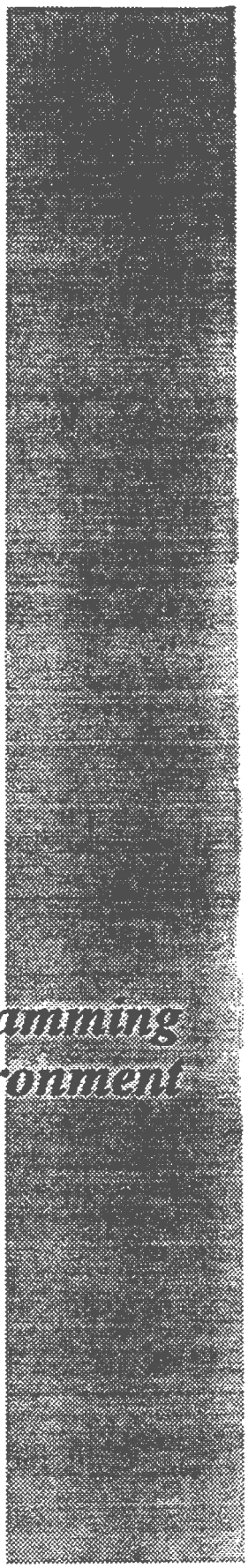
Ends analysis and returns to the top level of the system.

zoomln *FromLine, ToLine*

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.

zoomld *FromLine, Depth*

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.



Programming Environment

Programming Environment

Chapter 1
Environment

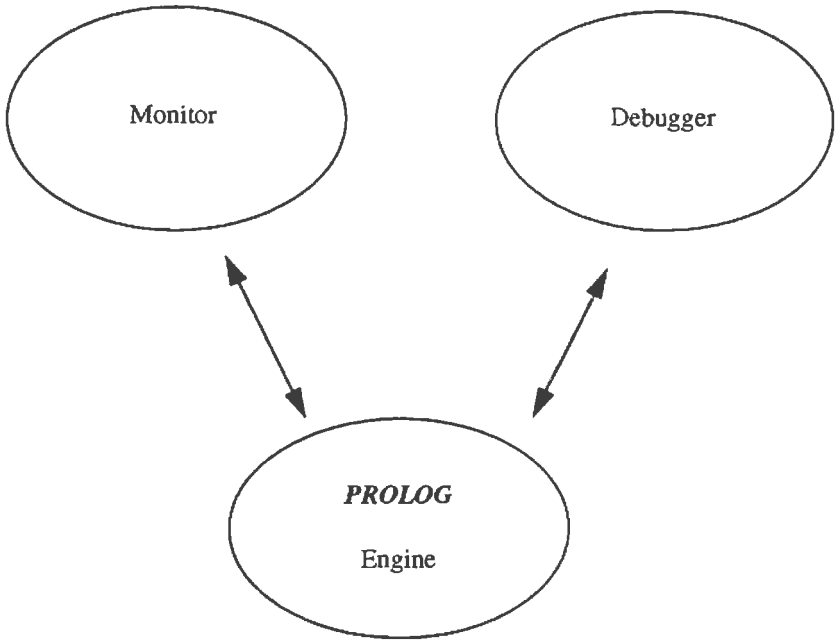
1.1 Basics

The ***ProLog by BIM*** system supports two look-and-feels: Motif and OPEN LOOK. ***ProLog by BIM*** chooses the most appropriate look-and-feel depending on the environment it is running in. One can change this choice with the environment variable BIM_PROLOG_LAF. When set, this variable should have one of the following values:

OL (ol)	OPEN LOOK (XView)
M (m)	OSF/Motif

It is perfectly possible to mix the look-and-feel used for the ***ProLog by BIM*** environment with a window manager that is compliant to the other look and feel.

The ***ProLog by BIM*** 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 window interaction.



The master window is the one running the ***ProLog by BIM*** engine. This is the frame in which the system is started. This is usually an xterm (Motif) or a commandtool (OPEN LOOK).

The other windows are not active by default. They can be activated and deactivated separately at any time. The activity of these windows is controlled by the **please/2** built-in predicate with switches **envmonitor (em)** and **envdebug (ed)** for the monitor and debugger windows 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+
```

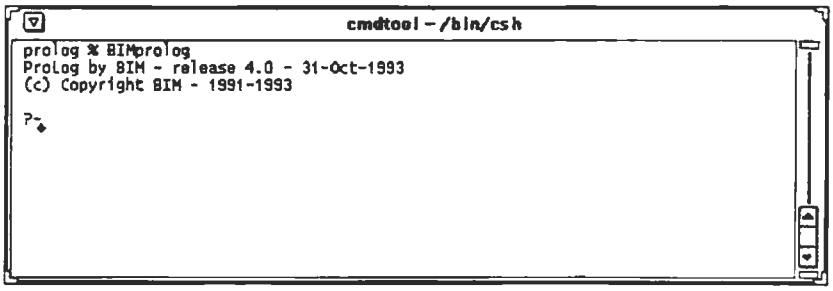
When a window is activated, the master process will start up a new process that creates the window frame. Upon creation, the defaults database is consulted to get the user's preferred window lay-out. When running, the monitor and debugger windows interact with their master.

These windows can be closed, resized and moved around the screen like any other frame of your window system.

In the following chapters, the three processes with their associated frames are described. First comes the master window, then the monitor window and finally the debugger window.

1.2 Master window

The master window is the window in which the ***ProLog by BIM*** engine was started up.



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.

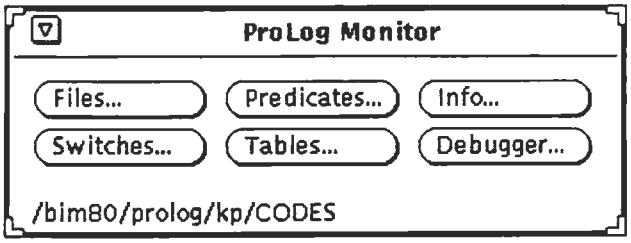
1.3 Monitor window

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 implemented access to a number of predicates 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 terminate the execution.

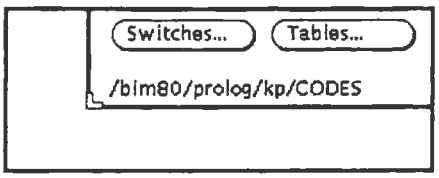
The base frame of the monitor window contains a set of pop-up buttons, that open a subwindow with available commands on the indicated topic. Each topic is described separately.



Help

The OPEN LOOK monitor window is integrated with the OPEN LOOK help mechanism. Touching the Help key on the keyboard, will pop-up a window with help information for the field under the cursor. The file name with the help information is located in $BIM_PROLOG_DIR/help. This directory must be included in the HELPPATH environment variable.

Working directory



The current working directory of the engine is indicated in the bottom left-hand corner of the monitor.

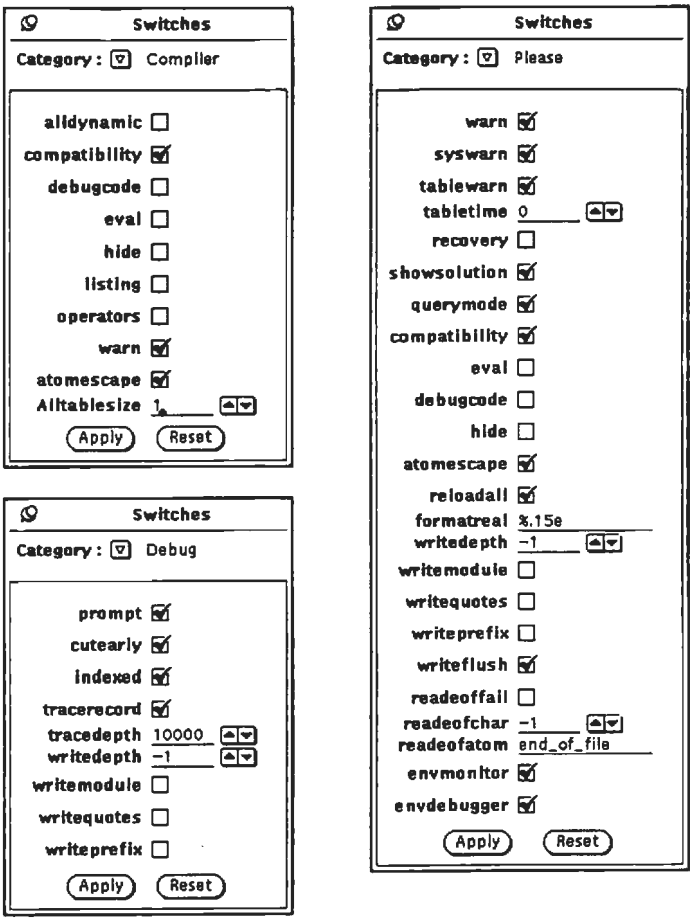
The working directory can be changed in the file browser (see "*Files subwindow*").

Info subwindow

The **Info** button displays information regarding the system and who to contact in case of problems (see **information/0,1,2**).

Switches subwindow

Switches...



By pressing the **Switches** button, a graphical overview of the engine switches is provided. The **Category** toggle offers a choice between the please, the debug and the compiler switches (see **please/2**, **debug/2** and **compiler/2**).

Each field in the three windows represents an option and its current value. These values can be changed by the following actions. For binary switches, clicking on the corresponding toggle button will switch the value. For the other switches, edit the value field. All changes made become only active when the **Apply** button is pressed.

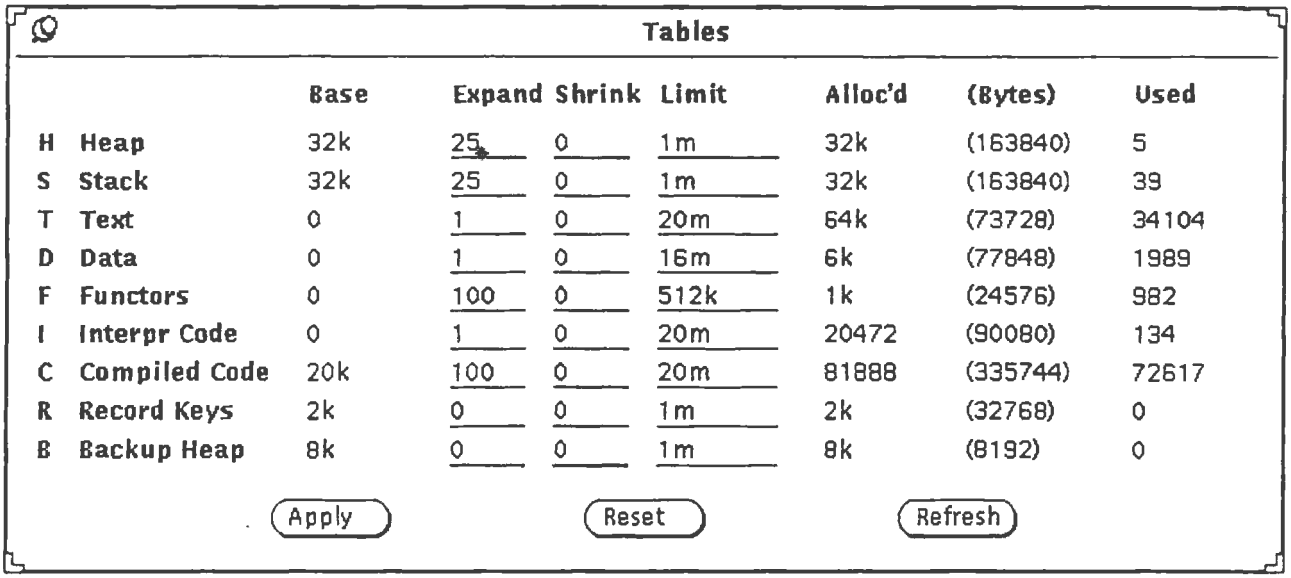
Once can reset the window by clicking the **Reset** button. This action will undo all changes that are not yet applied.

Tables

Tables...

By pressing the **Tables** button, the tables window is popped up.

This **Tables** window graphically represents the built-in predicate **table/2**. The window reflects the current values of the table size options of the different visible tables used by the ***ProLog by BIM*** system.



The fields **Expand, Shrink** and **Limit** can be edited. All changes made to these fields only become active when the **Apply** button is pushed.

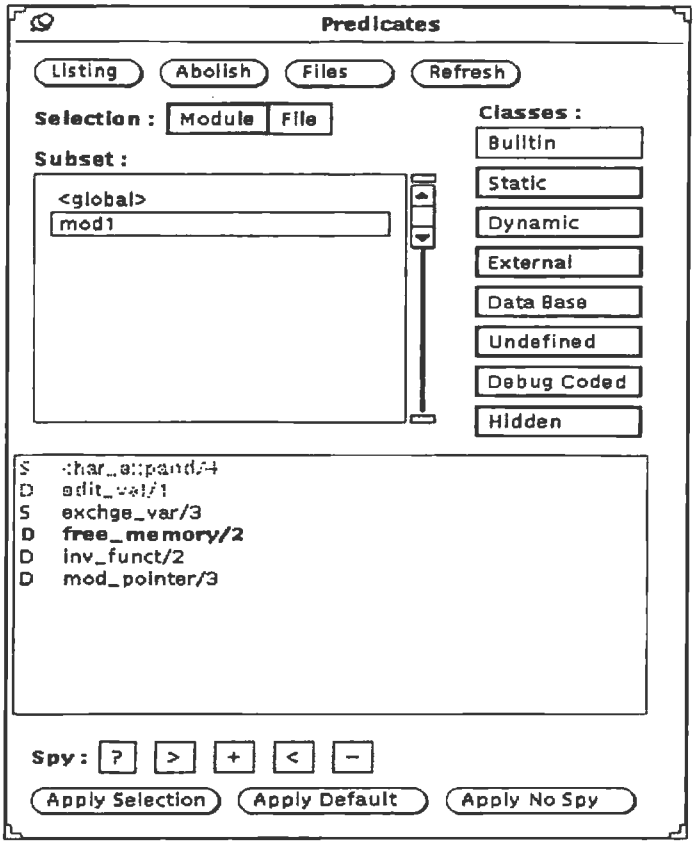
The original state of the window can be reset with the **Reset** button.

Pushing the **Refresh** button shows the values of the different parameters like they are now. The **Refresh** button can be pushed at any moment even when a query is active.

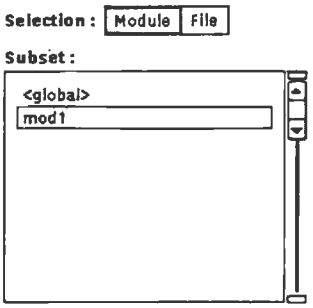
Predicates

Predicates...

The **Predicates** pop-up window provides an easy way for retrieving information about the predicates defined in the system.



The **Subset** window shows, depending on the value of the **Selection** switch, all currently defined modules or all currently loaded files in the system.



The **Classes** choice lists the existing predicate classes. The first six classes are disjunctive collections of predicates. For example: a built-in predicate is assumed to be only built-in and neither static nor dynamic. 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.

A selection of predicates is made by choosing one of the **Selection** topics, clicking a value in the **Subset** window, clicking one or more options in the **Classes** choice list and pushing the **Refresh** button. Now the set of matching predicates is displayed in the **Predicates Display**.

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.

Abbreviation	*Predicate class*
S	Static
D	Dynamic
B	Built-in
DB	Database
X	External

Font	*Predicate mode*
bold	Debug coded
dimmed	Hidden

The example in the figure has static and dynamic predicates selected. Hidden predicates are also requested. 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 hidden dynamic predicate.

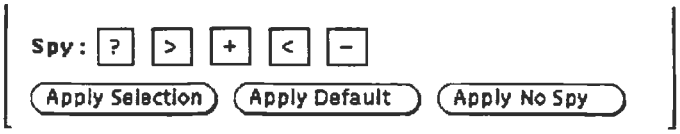
The four buttons result in the following actions.

The **Refresh** button should be used when the selection of predicates has been changed or when important changes to the ***ProLog by BIM*** database are made. Adding new predicates or deleting existing ones does not change the list of selected predicates automatically.

Pushing the **Listing** button invokes the built-in **listing/1** for each of the selected predicates in the Predicates Display. Only dynamic and debug coded predicates (see "*Built-in Predicates - In-core Database Manipulation*").

The button labeled **Abolish** invokes the built-in **abolish/1** for each of the selected predicates (see "*Built-in Predicates - In-core Database Manipulation*").

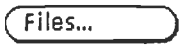
The **Files** button gives, for each of the selected predicates, the list of files which contain definitions of the predicate (see **predicate_files/2**). This list is printed out in the command tool.



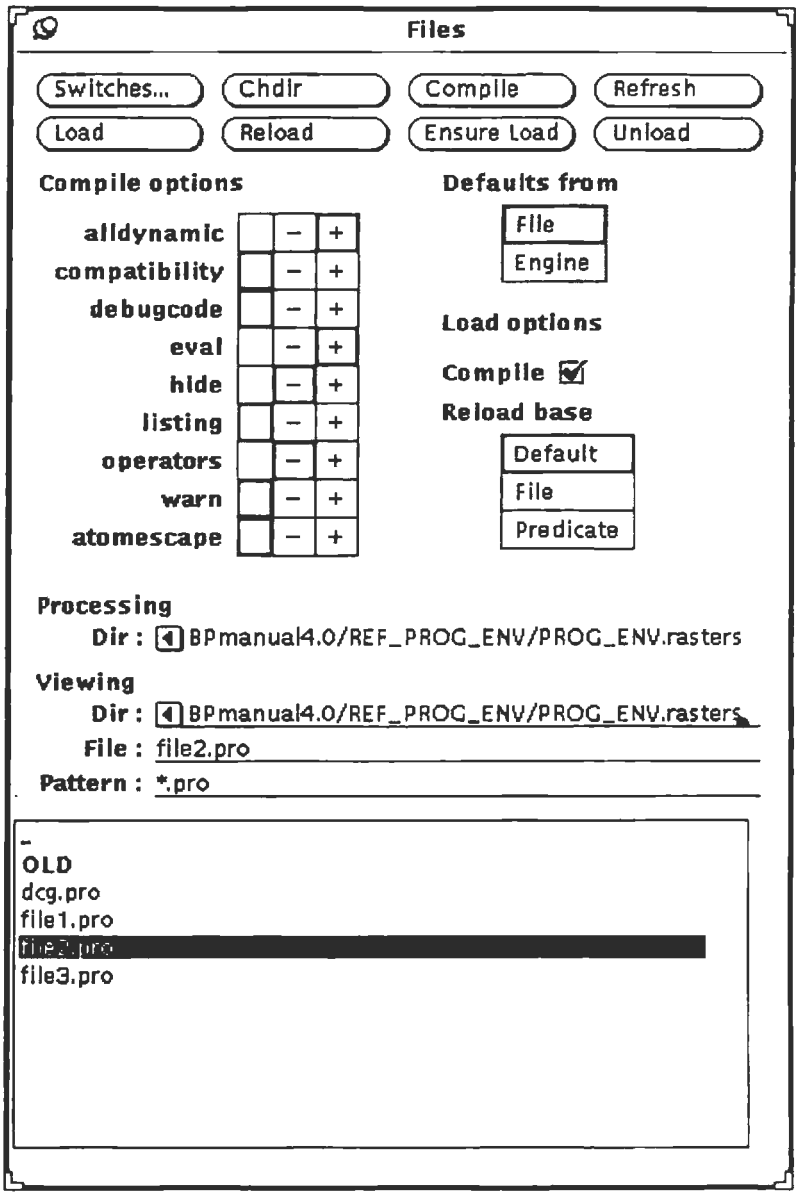
The lowest part of the Predicates window contains buttons for manipulating spies on predicates. When a single predicate is selected from the Predicate display, its current spy ports are shown.

The setting/unsetting of spy ports can be done in three ways. Clicking the **Apply Selection** button will set spy ports on the selected predicates as indicated with the **Spy** toggle selection. The **Apply Default** button sets spy ports as determined by the current engine default spy port setting (can be changed in the Debugger Control window). **Apply No Spy** removes all spy ports for the selected predicates (see "*Debugger*" for more information on spy ports).

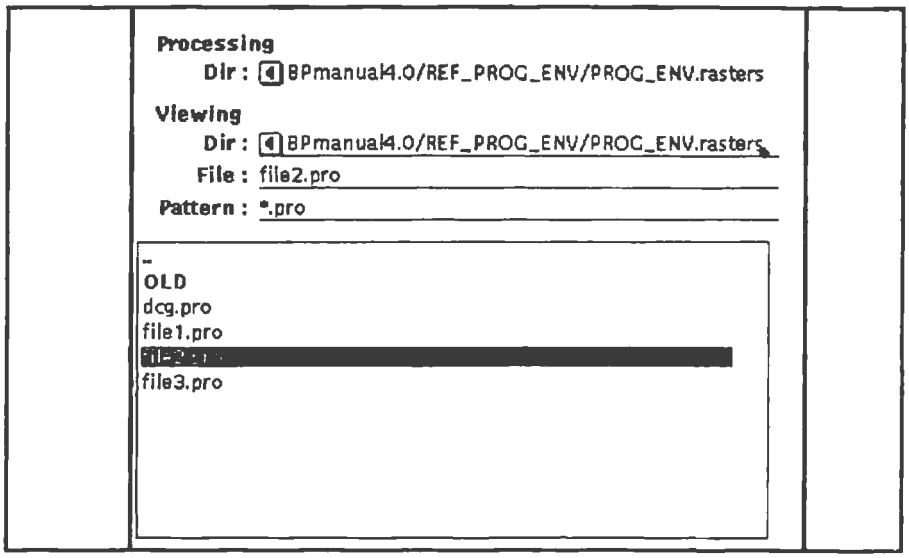
Files



This button pops up the **Files** window that provides an easy way to browse, compile and load files.



The upper part of the window provides some control facilities for compiling, loading and consulting files. The lower part contains the list of source files and subdirectories in the specified directory.

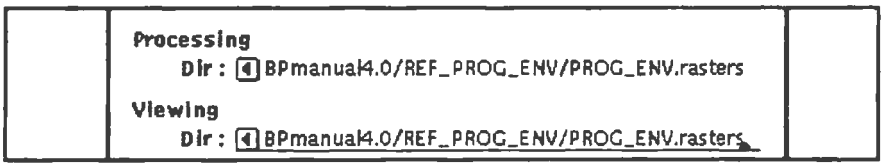


Everything below the label Viewing displays information about the current viewing environment. The field **Dir** contains the absolute path of the directory being viewed. The field **File** contains the last selected file name. The field **Pattern** specifies which file names are displayed. The pattern consists of a text containing at most one wildcard (*). The **Files Display** shows all the subdirectories and all the files in the currently viewed directory. The files must match the Pattern.

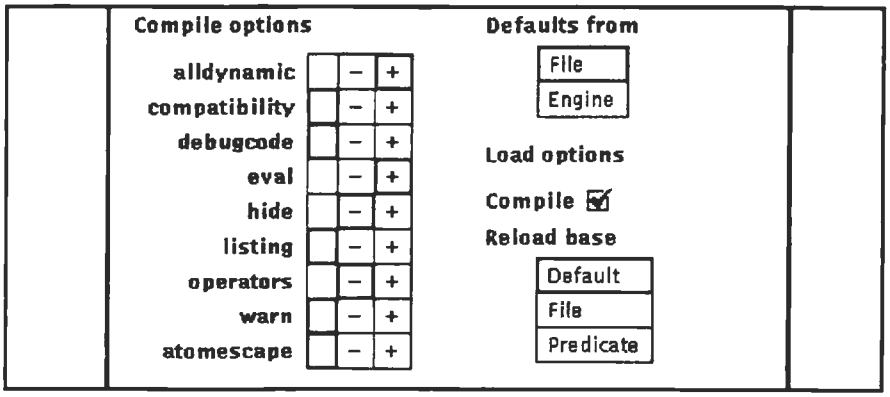
One can browse through the directory structure by clicking (only 1 click!) on a directory name in the display or by editing the Dir field and pushing the Refresh button (refresh is invoked when pushing the <Return> key on the keyboard).

A file that is selected (click 1 time on the file name) is shown in the File field.

Changing the viewing directory has no impact on the working direcory. The working directory can be changed to the viewing directory by pushing the **Chdir** button (top of the window).



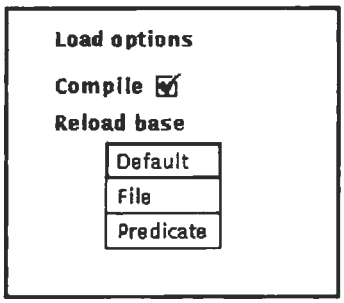
The **Compile options** list shows the compiler options. These options can be given a value by clicking one of the corresponding toggle values. '+' means on, '-' means off. The options not specified (empty field) are taken from the defaults as specified in the **Defaults from** choice button. If **File** is chosen, the options of the previous compilation of the file are taken. If **Engine** is in effect, the current values of the engine compiler switches are used. The **Switches** button pops up a window that shows the current values of the compiler switches. It is the same window as can be viewed with the Switches button of the main monitor window.



The button **Compile** calls **compile/2** with the specified options and the selected file as arguments. No loading is performed.

The load options are taken into account when activating one of the button **Load, Reload, Ensure Load** and **Unload**. When the **Compile** field is off, clicking one of these buttons calls the corresponding load built-in predicate. When **Compile** is switched on, the corresponding consult built-in predicate is called (thus performing a compilation if needed). By default, the **Compile** switch is on, meaning that a consult action will be performed.

A further specification of the **Reload** button is possible with the **Reload base** choice. Selecting one of the values changes reload(reconsult) into reload_file or reload_predicates (reconsult_file/reconsult_predicates)



The example below shows the actions that must be done in order to specify a rather complicated consult.

Example

```
reconsult_file('- -a+ -e- ~prolog/environment/file2.pl').
```

can be specified by:

1 entering '~prolog' in the **Viewing Dir** field and clicking on refresh.

2 modifying the **Pattern** field to *.pl.

3 clicking on the subdirectory 'environment' in the **Files Display**

4 clicking on 'file2'

The file is now selected. The compiler options are specified by:

5 clicking on '+' of the alldynamic and '-' of the eval toggles.

6 Selecting 'Engine' in the **Defaults from** is the equivalent of the - sign.

The predicate which will be called is selected by:

7 Ticking 'Compile' in the **Load option**, selecting 'File' in the **Reload base** and pushing the **Reload** button makes sure that the reconsult_file predicate is called.

The **Unload** button corresponds to the unload predicate.

The **Compile** button allows an explicit call to the compiler with the specified file and the options set as in 'compiler options' and 'Defaults from'.

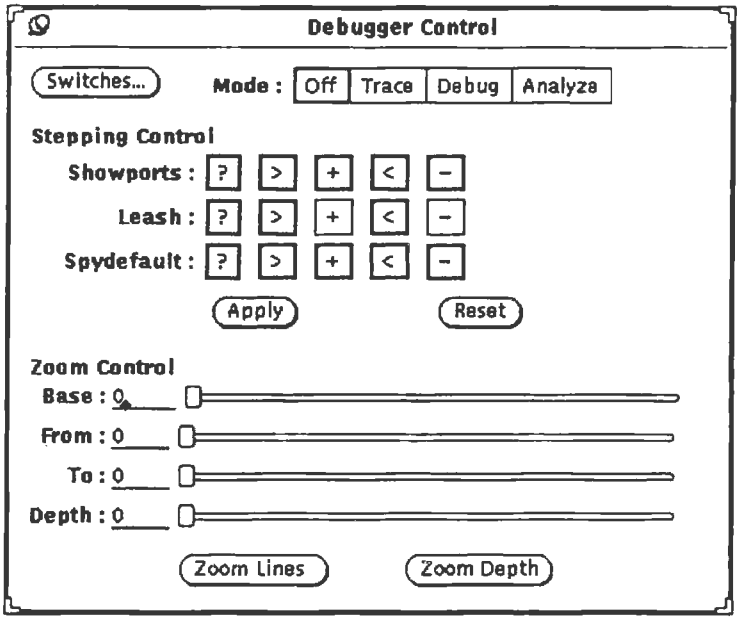
The **Refresh** button will refresh all the selections made. The button is typically used when a file has been created or when the pattern has been adapted.

The **Switches** button visualizes the compiler switches subwindow which enables global specification of the option with which the compiler will be called.

Debugger

Debugger...

The **Debugger** pop-up window is divided into three logical regions: one for the selection of the debug mode, one for the spy point defaults and one for trace zooming facilities.

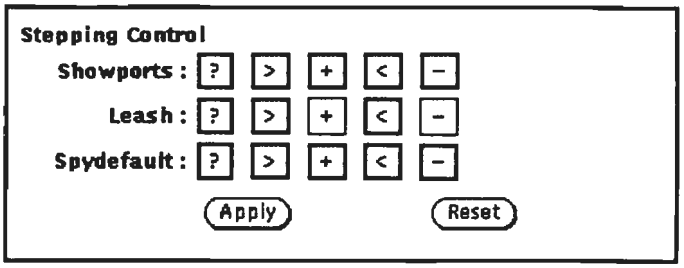


Each of these regions is described separately in the following paragraphs. The Switches button pops up the debug switches subwindow and allows control of the general behavior of the debugger (same window as can be viewed from the Switches button of the monitor main window).

Selection of the debug 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 built-in predicates **trace/0** or **debug/0** are invoked. The last mode is **Analyze**, which is the mode that results from calling the built-in **keeptrace/0**.

Stepping control



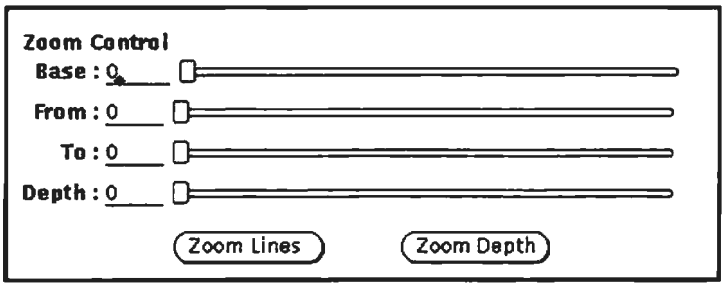
The three items provide the built-in predicates **showports/1**, **leash/1** and **spydefault/1**. The five ports are represented by their symbolic abbreviations:

?	call port
>	unify port
+	exit port
<	redo port
-	fail port

Changing the settings can be done by clicking on the port that must be switched.

Trace zooming

The lower part of the debugger control panel allows zooming through the trace. This corresponds to the built-ins **zoomln/2** and **zoomld/2**. These can be activated by pressing the appropriate buttons.



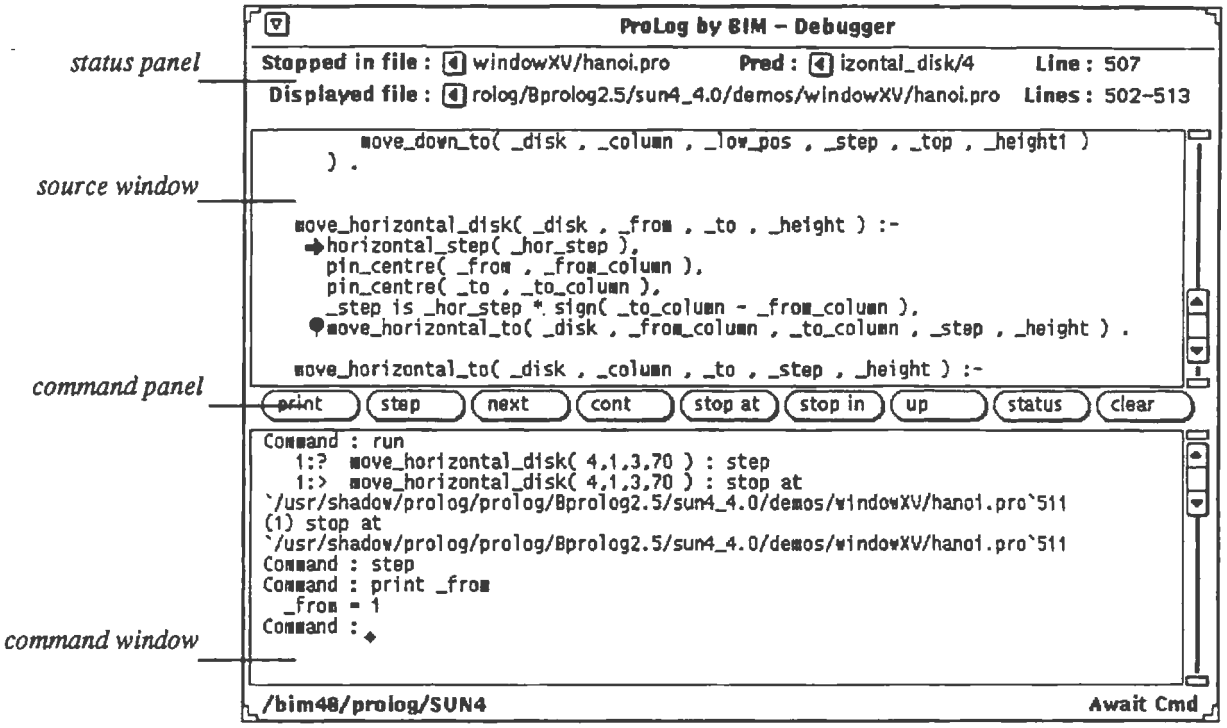
The arguments that will be used are retrieved from the sliders. The **From** and **To** sliders give the first and second argument for **zoomln/2**. The **From** and **Depth** sliders are used for the arguments of **zoomld/2**.

The **Base** slider is used to set a base value for the **From** and **To** sliders. Since it would be impossible to represent the whole range of trace lines in a single slider (there can be 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 making the other sliders range from 12000 through 12300. They are currently set to 12100 and 12200.

1.4 Debugger window

Window lay-out



The debugger window consists of:

status panel

This panel holds status information.

source window

A text window containing the program source (It cannot be edited).

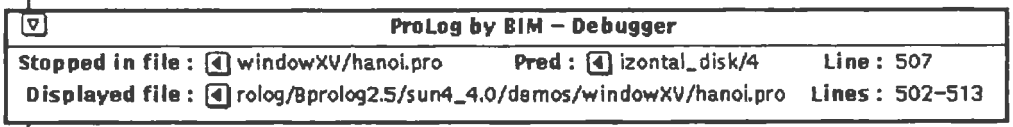
command panel

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). In the Motif version, this window is split into two parts: the bottom part accepts input and the output is displayed in the upper part.

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 which 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, which on the menu, and in what order, is customizable through the defaults facilities. The buttons and menu items can be changed temporarily during a session with the command button, unbutton, menu, unmenu.

For commands that need 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 does not contain special characters).

A line number is formed by selecting the line in the source window.

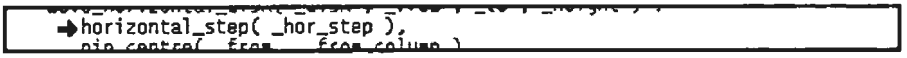
When a predicate is needed as an argument, it should be selected with its name/arity. If only the name is selected, a "/" will be appended and the arity must be entered manually.

There is one button, labeled ***invest*** in ***analyze*** mode that operates 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 a file other 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.



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.

At any time during a suspension of the execution, the line where that suspension occurred, can be displayed in the source window. For that, the *show* command can be used. This displays the line in the window, indicated by a hollow arrow.

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 still wishes to see the corresponding source line, the ***show*** command can be used.

Another use of the ***show*** command is to redisplay the current predicate line 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.

```
Command : print _from
  _from = 1
```

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*** commands 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.

Programming Environment

Chapter 2
Defaults

2.1 Motif window environment resources

The resources are specified in the standard X11** method.

Widgets are named along their function and type. Not every widget has a unique name, and therefore cannot always be adapted independently.

The widget hierarchy is given as an indication.

Monitor

Application Name: ProLogMonitor
Application Class: ProLog
Application Resources:

Name	Type	Description
x	integer	X position of Monitor frame
y	integer	Y position of Monitor frame
infoX	integer	X position of Info window
infoY	integer	Y position of Info window
switchesX	integer	X position of Switches window
switchesY	integer	Y position of Switches window
tablesX	integer	X position of Tables window
tablesY	integer	Y position of Tables window
predicatesX	integer	X position of Predicates window
predicatesY	integer	Y position of Predicates window
predicatesLines	integer	number of lines in Predicates window
filesX	integer	X position of Files window
filesY	integer	Y position of Files window
filesLines	integer	number of lines in Files window
debuggerX	integer	X position of Debugger Control window
debuggerY	integer	Y position of Debugger Control window

Widget Hierarchy:

```
MessageWindow
BasePane
        ActionArea
                Files...
                Predicates...
                Info...
                        Switches...
                Tables...
                Debugger...
        StatusForm
                WorkingDirLabel
                StatusLabel
InfoFrame
        InfoPane
                InfoPanel
                ActionArea
                        Ok
```

TablesFrame
    TablesPane
        TablesPanel
        ActionArea
            Apply
            Reset
            Refresh
            Close

SwitchesFrame
    SwitchesPane
        SwitchesCategoryForm
            SwitchesCategoryMenu
        SwitchesPanel
        ActionArea
            Apply
            Reset
            Close

PredicatesFrame
    PredicatesPane
        ActionArea
            Listing
            Abolish
            Files
            Refresh
        PredicatesControlPanel
            PredicatesControl
        PredicatesDisplay
        PredicatesSpyPanel
            PredicatesControl
        ActionArea
            Apply Selection
            Apply Default
            Apply No Spy

FilesFrame
    FilesPane
        ActionArea
            Switches...
            Chdir
            Compile
            Refresh
            Load
            Reload
            Ensure Load
            Unload
        FilesOptionsPanel
            FilesOptions
        FilesStatusPanel
            FilesStatus
        FilesDisplay

DebuggerFrame
    DebuggerPane
        ActionArea
            Switches...
            Close
        DebuggerModePanel
            DebuggerControl
        DebuggerSteppingPanel
            DebuggerControl

ActionArea
Apply
Reset
DebuggerZoomPanel
DebuggerControl
ActionArea
Zoom Lines
Zoom Depth

Debugger

Application Name: ProLogDebugger
Application Class: ProLog
Application Resources:

Name	Type	Description
x	integer	X position of Debugger frame
y	integer	Y position of Debugger frame
width	integer	Number of columns
srcLines	integer	Number of lines in Source window
cmdLines	integer	Number of lines in Command window
steppingButton	string	Description of buttons for stepping
analyzeButton	string	Description of buttons for analyze
steppingMenu	string	Description of menu for stepping
analyzeMenu	string	Description of menu for analyze

Widget Hierarchy:

MessageWindow
BasePane
InfoPanel
InfoLabel
SourceText
CommandPanel
CommandButton
SteppingMenu
MenuButton
AnalyzeMenu
MenuButton
CommandHistoryPanel
CommandHistory
CommandInput
StatusPanel
StatusLabel

For the placement of the frames to be effective, the Motif Window Manager must allow the application to position its windows. This can be done with the following resource:

Mwm*clientAutoPlace: False

Both applications use the ISO 8859.1 character set. As a result, a font list resource setting should include this character set.

Example

To change the background color of the BasePane of the Monitor to red, specify the following resource:

```
ProLogMonitor*BasePane*background: red
```

2.2 XView window environment resources

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

ProLogMonitor.font string
Font for text.

ProLogMonitor.font.scale enumeration
Scaling of the chosen font (value: small, medium, large, extralarge)

ProLogMonitor.x integer
Horizontal offset of frame on screen

ProLogMonitor.y integer
Vertical offset of frame on screen.

ProLogMonitor.infoX integer
Horizontal offset of info pop-up from monitor frame

ProLogMonitor.infoY integer
Vertical offset of info pop-up from monitor frame

ProLogMonitor.switchesX integer
Horizontal offset of switches pop-up from monitor frame

ProLogMonitor.switchesY integer
Vertical offset of switches pop-up from monitor frame

ProLogMonitor.tablesX integer
Horizontal offset of tables pop-up from monitor frame

ProLogMonitor.tablesY integer
Vertical offset of tables pop-up from monitor frame

ProLogMonitor.predicatesX integer
Horizontal offset of predicates pop-up from monitor frame

ProLogMonitor.predicatesY integer
Vertical offset of predicates pop-up from monitor frame

ProLogMonitor.filesX integer
Horizontal offset of files pop-up from monitor frame

ProLogMonitor.filesY integer
Vertical offset of files pop-up from monitor frame

ProLogMonitor.debuggerX integer
Horizontal offset of debugger pop-up from monitor frame

ProLogMonitor.debuggerY 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
ProLogDebugger.font Font for text windows	string
ProLogDebugger.font.scale Scaling of text font (value: small, medium, large, extralarge)	enumeration
ProLogDebugger.width Width of the frame in columns	integer
ProLogDebugger.srclines Number of lines in the source window	integer
ProLogDebugger.cmdlines Number of lines in the command window	integer
ProLogDebugger.x Horizontal offset of frame on screen	integer
ProLogDebugger.y Vertical offset of frame on screen	integer
ProLogDebugger.steppingbutton Stepping: Choose active buttons	string
ProLogDebugger.analyzebutton Analyze: Choose active buttons	string
ProLogDebugger.steppingmenu Stepping: Choose active menu items	string
ProLogDebugger.analyzemenu Analyze: Choose active menu items	string

2.3 Debugger resources

The values of the stepping and analyze button and menu resources of the Debugger must be strings. 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 codes.

Table 1:

	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			

Example

The default setting of the stepping buttons can be specified by:

PrologDebugger*steppingButton: F1 E3 E2 E1 D1 D2 E5 D3 D4

The following buttons appear now in the command panel of the environment debugger:

Windowing And Graphics Libraries

0.1 Overview

To be both easy to use and effective in an industrial environment, Prolog applications need flexible interfaces to windowing and graphics packages. ***ProLog by BIM*** interfacing facilities offer unrestricted access to the OSF/Motif**, OPENLOOK/Xview**, **, and Xt** libraries by providing a one to one mapping between Prolog predicates and library routines.

Interaction windows, buttons, menus and graphics can now be defined and manipulated from within Prolog programs, offering exactly the same functionality in user interfacing as is available in procedural languages.

Also included in the ***ProLog by BIM*** package is the graphical user interface generator Carmen. This tool allows the definition of the interface of the applications without having to do tedious coding. Carmen will generate the appropriate calls to the interface predicates from the specification of the interface (for more information see "*Carmen*").

Windowing and Graphics Libraries

Chapter 1
Interface to OSF / Motif

The interface to the *Motif* package is based on the *Xt* interface. Although it is sufficient to use the Motif interface only, both packages can be used together.

This chapter describes the ***ProLog by BIM*** binding for the *OSF/Motif* package. All predicates and attributes are briefly described. Only a basic explanation of the Motif functions is given. A complete description of this package can be found in the O'Reilly series *"OSF/Motif User's Guide"*.

1.1 Availability of Motif

The Motif predicates are defined in the ***ProLog by BIM*** library **-Lwindowing/motif**. Since the interface is based on the Xt interface, the Xt package must also be loaded. It is defined in the ***ProLog by BIM*** library **-Lwindowing/xt**. A third library, on which the Xt package is based, is the X11 library, defined in the ***ProLog by BIM*** library **-Lwindowing/xlib**. It is not mandatory to load it. However, it is useful to do so, because it provides all lower level access possibilities to the window system.

To load the packages together with ***ProLog by BIM***:

```
% BIMprolog -Lwindowing/motif -Lwindowing/xt -Lwindowing/xlib
```

To consult the packages interactively, from a running system:

```
?- lib( motif ), lib( xt ), lib( xlib ) .
```

Note that the order of consulting the packages is important: first the highest level package -Lwindowing/motif, then -Lwindowing/xt and finally -Lwindowing/xlib.

Note:

Due to the limitations of the SunOs 4.x dynamic linking package it is not possible to incrementally load the packages as explained above. Please use an extended executable as explained below.

If an extended executable containing the motif interface has been built during the installation procedure it can be started with:

```
% BIMprologOM
```

1.2 Widget classes

The table below lists the widget classes that are provided by Motif. These can be used in the same way as the widget classes provided by Xt.

Widget class
- xmMenuShellWidgetClass
- xmDialogShellWidgetClass
- xmFileSelectionBoxWidgetClass
- xmCommandWidgetClass
- xmSelectionBoxWidgetClass
- xmMessageBoxWidgetClass
- xmFormWidgetClass
- xmScaleWidgetClass
- xmFrameWidgetClass
- xmPanedWindowWidgetClass
- xmDrawingAreaWidgetClass
- xmBulletinBoardWidgetClass
- xmRowColumnWidgetClass
- xmMainWindowWidgetClass
- xmScrolledWindowWidgetClass
- xmDrawnButtonWidgetClass
- xmTextWidgetClass
- xmScrollBarWidgetClass
- xmListWidgetClass

xmToggleButtonWidgetClass
xmPushButtonWidgetClass
xmCascadeButtonWidgetClass
xmLabelWidgetClass
xmArrowButtonWidgetClass
xmSeparatorWidgetClass
xmToggleButtonGadgetClass
xmPushButtonGadgetClass
xmCascadeButtonGadgetClass
xmLabelGadgetClass
xmArrowButtonGadgetClass
xmSeparatorGadgetClass

1.3 Initialization and termination

Initialization of a Motif user interface is done entirely using Xt (see "*Interface to Xt - Initialization and termination*"). The toolkit must be initialized first. Usually the next step is to create an application context, open the display and create an application shell widget. From then on, Motif widgets can be created inside the application shell widget (see below "*Widget manipulations*").

1.4 Widget manipulations

All Motif widgets can be manipulated as any other Xt widget, using the Xt widget manipulation predicates. The Motif interface mainly provides a number of useful predicates for manipulation of Motif widgets.

Widget creation

For each Motif widget a convenience predicate is provided for creation of the widget. It exists in two forms (arity 4 and 5): one form (arity 4) takes the attributes as a single list of attribute name/value pairs. The arity 5 form takes the attributes in an Xt argument list, with the number of attributes as an extra argument.

XmCreateMenuShell/5

XmCreateMenuShell(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmMenuShell widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateMenuShell/4

XmCreateMenuShell(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmMenuShell widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateDialogShell/5

XmCreateDialogShell(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateDialogShell/4

XmCreateDialogShell(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateFileSelectionBox/5

XmCreateFileSelectionBox(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmFileSelectionBox widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateFileSelectionBox/4

XmCreateFileSelectionBox(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmFileSelectionBox widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateFileSelectionDialog/5

XmCreateFileSelectionDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmFileSelectionBox child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateFileSelectionDialog/4

XmCreateFileSelectionDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmFileSelectionBox child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateCommand/5

XmCreateCommand(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmCommand widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateCommand/4

XmCreateCommand(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmCommand widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateSelectionBox/5

XmCreateSelectionBox(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmSelectionBox widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateSelectionBox/4

XmCreateSelectionBox(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmSelectionBox widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateSelectionDialog/5

XmCreateSelectionDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmSelectionBox child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateSelectionDialog/4

XmCreateSelectionDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmSelectionBox child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreatePromptDialog/5

XmCreatePromptDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmSelectionBox child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreatePromptDialog/4

XmCreatePromptDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmSelectionBox child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateMessageBox/5

XmCreateMessageBox(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmMessageBox widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateMessageBox/4

XmCreateMessageBox(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmMessageBox widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateMessageDialog/5

XmCreateMessageDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmMessageBox child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateMessageDialog/4

XmCreateMessageDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmMessageBox child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateErrorDialog/5

XmCreateErrorDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmMessageBox child with error symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateErrorDialog/4

XmCreateErrorDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmMessageBox child with error symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateInformationDialog/5

XmCreateInformationDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmMessageBox child with information symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateInformationDialog/4

XmCreateInformationDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmMessageBox child with information symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateQuestionDialog/5

XmCreateQuestionDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmMessageBox child with question symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateQuestionDialog/4

XmCreateQuestionDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmMessageBox child with question symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateWarningDialog/5

XmCreateWarningDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmMessageBox child with warning symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateWarningDialog/4

XmCreateWarningDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmMessageBox child with warning symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateWorkingDialog/5

XmCreateWorkingDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmMessageBox child with working symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateWorkingDialog/4

XmCreateWorkingDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmMessageBox child with working symbol is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateForm/5

XmCreateForm(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmForm widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateForm/4

XmCreateForm(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmForm widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateFormDialog/5

XmCreateFormDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmForm child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateFormDialog/4

XmCreateFormDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmForm child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateScale/5

XmCreateScale(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmScale widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateScale/4

XmCreateScale(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmScale widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateFrame/5

XmCreateFrame(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmFrame widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateFrame/4

XmCreateFrame(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmFrame widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreatePanedWindow/5

XmCreatePanedWindow(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmPanedWindow widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreatePanedWindow/4

XmCreatePanedWindow(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmPanedWindow widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateDrawingArea/5

XmCreateDrawingArea(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDrawingArea widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateDrawingArea/4

XmCreateDrawingArea(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDrawingArea widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateBulletinBoard/5

XmCreateBulletinBoard(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmBulletinBoard widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateBulletinBoard/4

XmCreateBulletinBoard(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmBulletinBoard widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateBulletinBoardDialog/5

XmCreateBulletinBoardDialog(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmDialogShell widget with a xmBulletinBoard child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateBulletinBoardDialog/4

XmCreateBulletinBoardDialog(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmDialogShell widget with a xmBulletinBoard child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateRowColumn/5

XmCreateRowColumn(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmRowColumn widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateRowColumn/4

XmCreateRowColumn(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmRowColumn widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateRadioBox/5

XmCreateRadioBox(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmRowColumn widget with radio box behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateRadioBox/4

XmCreateRadioBox(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmRowColumn widget with radio box behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateOptionMenu/5

XmCreateOptionMenu _Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmRowColumn widget with option menu behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateOptionMenu/4

XmCreateOptionMenu(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmRowColumn widget with option menu behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateMenuBar/5

XmCreateMenuBar(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmRowColumn widget with menu bar behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateMenuBar/4

XmCreateMenuBar(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmRowColumn widget with menu bar behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreatePulldownMenu/5

XmCreatePulldownMenu(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmRowColumn widget with pulldown menu behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreatePulldownMenu/4

XmCreatePulldownMenu(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmRowColumn widget with pulldown menu behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreatePopupMenu/5

XmCreatePopupMenu(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmRowColumn widget with pop-up menu behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreatePopupMenu/4

XmCreatePopupMenu(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmRowColumn widget with pop-up menu behavior is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateMainWindow/5

XmCreateMainWindow _Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmMainWindow widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateMainWindow/4

XmCreateMainWindow(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmMainWindow widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateScrolledWindow/5

XmCreateScrolledWindow(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmScrolledWindow widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateScrolledWindow/4

XmCreateScrolledWindow(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmScrolledWindow widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateText/5

XmCreateText(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmText widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateText/4

XmCreateText(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmText widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateScrolledText/5

XmCreateScrolledText(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmScrolledWindow widget with a xmText child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateScrolledText/4

XmCreateScrolledText(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmScrolledWindow widget with a xmText child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateScrollBar/5

XmCreateScrollBar(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmScrollBar widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateScrollBar/4

XmCreateScrollBar(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmScrollBar widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateList/5

XmCreateList(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmList widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateList/4

XmCreateList(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmList widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateScrolledList/5

XmCreateScrolledList(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmScrolledWindow widget with a xmList child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateScrolledList/4

XmCreateScrolledList(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmScrolledWindow widget with a xmList child is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateToggleButton/5

XmCreateToggleButton(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmToggleButton widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateToggleButton/4

XmCreateToggleButton(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmToggleButton widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreatePushButton/5

XmCreatePushButton(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmPushButton widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreatePushButton/4

XmCreatePushButton(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmPushButton widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateCascadeButton/5

XmCreateCascadeButton(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmCascadeButton widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateCascadeButton/4

XmCreateCascadeButton(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmCascadeButton widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateLabel/5

XmCreateLabel(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmLabel widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateLabel/4

XmCreateLabel(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmLabel widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateArrowButton/5

XmCreateArrowButton(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmArrowButton widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateArrowButton/4

XmCreateArrowButton _Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmArrowButton widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateSeparator/5

XmCreateSeparator(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmSeparator widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateSeparator/4

XmCreateSeparator(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmSeparator widget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateToggleButtonGadget/5

XmCreateToggleButtonGadget(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmToggleButton gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateToggleButtonGadget/4

XmCreateToggleButtonGadget(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmToggleButton gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreatePushButtonGadget/5

XmCreatePushButtonGadget(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmPushButton gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreatePushButtonGadget/4

XmCreatePushButtonGadget(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmPushButton gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateCascadeButtonGadget/5

XmCreateCascadeButtonGadget(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmCascadeButton gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateCascadeButtonGadget/4

XmCreateCascadeButtonGadget(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmCascadeButton gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateLabelGadget/5

XmCreateLabelGadget(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmLabel gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateLabelGadget/4

XmCreateLabelGadget(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmLabel gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateArrowButtonGadget/5

XmCreateArrowButtonGadget(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmArrowButton gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateArrowButtonGadget/4

XmCreateArrowButtonGadget(_Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmArrowButton gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

XmCreateSeparatorGadget/5

XmCreateSeparatorGadget(_Widget, _Parent, _Name, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer

A new xmSeparator gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed in the last two arguments. The attribute list is *arg4*, and the number of attributes in that list is *arg5*.

XmCreateSeparatorGadget/4

XmCreateSeparatorGadget _Widget, _Parent, _Name, _AttributeList)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : list

A new xmSeparator gadget is created. Its handle is returned as *arg1*. The parent widget is *arg2*. The widget name is *arg3*. An attribute list is passed as *arg4*. This list consists of a sequence of attribute names and their corresponding values.

Window manager check

XmIsMotifWMRunning/2

XmIsMotifWMRunning(_Answer, _Shell)

arg1 : free : atom (Boolean)
arg2 : ground : pointer

Checks whether the Motif Window Manager is running on the screen that contains the shell widget *arg2*. The Boolean answer is returned in *arg1*.

File selection box

XmFileSelectionBoxGetChild/3

XmFileSelectionBoxGetChild(_Child, _Widget, _Component)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom (XmFileSelectionBoxDialogComponent)

The child widget of widget *arg2*, for component *arg3* is returned in *arg1*.

XmFileSelectionDoSearch/2

XmFileSelectionDoSearch(_Widget, _DirectoryMask)

arg1 : ground : pointer
arg2 : ground : pointer

A directory search for widget *arg1* is initiated. The directory mask is updated with *arg2*.

Command

XmCommandAppendValue/2

XmCommandAppendValue(_Widget, _Value)

arg1 : ground : pointer
arg2 : ground : pointer

The xmString value *arg2* is appended at the end of the command in widget *arg1*.

XmCommandError/2

XmCommandError(_Widget, _Error)

arg1 : ground : pointer
arg2 : ground : pointer

The xmString error message *arg2* is entered in the history area of the widget *arg1*.

XmCommandGetChild/3

XmCommandGetChild(_Child, _Widget, _Component)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom (XmCommandDialogComponent)

The child widget of widget *arg2*, for component *arg3* is returned in *arg1*.

XmCommandSetValue/2

XmCommandSetValue(_Widget, _Value)

arg1 : ground : pointer
arg2 : ground : pointer

The xmString value *arg2* replaces the command in widget *arg1*.

Selection box

XmSelectionBoxGetChild/3

XmSelectionBoxGetChild(_Child, _Widget, _Component)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom (XmSelectionBoxDialogComponent)

The child widget of widget *arg2*, for component *arg3* is returned in *arg1*.

Message box

XmMessageBoxGetChild/3

XmMessageBoxGetChild(_Child, _Widget, _Component)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom (XmMessageBoxDialogComponent)

The child widget of widget *arg2*, for component *arg3* is returned in *arg1*.

Scale

XmScaleGetValue/2

XmScaleGetValue(_Widget, _Value)

arg1 : ground : pointer
arg2 : free : integer

The scale value of widget *arg1*, is returned in *arg2*.

XmScaleSetValue/2

XmScaleSetValue(_Widget, _Value)

arg1 : ground : pointer
arg2 : ground : integer

The scale value of widget *arg1*, is set to *arg2*.

Row column

XmGetMenuCursor/2

XmGetMenuCursor(_Cursor, _Display)

arg1 : free : pointer
arg2 : ground : pointer

The menu cursor currently used by the application on display *arg2* is returned in *arg1*.

XmSetMenuCursor/2

XmSetMenuCursor(_Display, _Cursor)

arg1 : ground : pointer
arg2 : ground : pointer

The menu cursor for the application is set to *arg2* on display *arg1*.

XmMenuPosition/2

XmMenuPosition(_MenuWidget, _Event)

arg1 : ground : pointer
arg2 : ground : pointer

The menu widget *arg1* is positioned as indicated in event *arg2*.

XmOptionButtonGadget/2

XmOptionButtonGadget(_Gadget, _MenuWidget)

arg1 : free : pointer
arg2 : ground : pointer

The cascade button gadget for option menu *arg2*, is returned in *arg1*.

XmOptionLabelGadget/2

XmOptionLabelGadget(_Gadget, _MenuWidget)

arg1 : free : pointer
arg2 : ground : pointer

The label gadget for option menu *arg2*, is returned in *arg1*.

Main window

XmMainWindowSep1/2

XmMainWindowSep1(_Separator, _Widget)

arg1 : free : pointer
arg2 : ground : pointer

The first separator widget of widget *arg2*, is returned in *arg1*.

XmMainWindowSep2/2

XmMainWindowSep2(_Separator, _Widget)

arg1 : free : pointer
arg2 : ground : pointer

The second separator widget of widget *arg2*, is returned in *arg1*.

XmMainWindowSetAreas/6

XmMainWindowSetAreas(_Widget, _MenuBar, _Command, _HorizontalScrollbar, _VerticalScrollbar, _WorkRegion)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : pointer
arg4 : ground : pointer
arg5 : ground : pointer
arg6 : ground : pointer

The components of widget *arg1* are installed. The Menu Bar is *arg2*, the Command widget is *arg3*, the Horizontal and Vertical Scroll bars are *arg4* and *arg5*, and the Work Region is *arg6*.

Scrolled window

XmScrolledWindowSetAreas/4

XmScrolledWindowSetAreas (_Widget, _HorizontalScrollbar, _VerticalScrollbar, _WorkRegion)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : pointer
arg4 : ground : pointer

The components of widget *arg1* are installed. The Horizontal and Vertical Scroll bars are *arg2* and *arg3*, and the Work Region is *arg4*.

Text

XmTextClearSelection/2

XmTextClearSelection(_Widget, _Time)

arg1 : ground : pointer
arg2 : ground : pointer

The primary selection in widget *arg1* is cleared. The time of the event that triggered this request should be passed as *arg2*.

XmTextGetSelection/2

XmTextGetSelection(_Selection, _Widget)

arg1 : free : atom
arg2 : ground : pointer

The primary selection of widget *arg2* is returned in *arg1*.

XmTextSetSelection/4

XmTextSetSelection(_Widget, _First, _Last, _Time)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : pointer

The primary selection of widget *arg1* is set to the text in positions from *arg2* to *arg3*. The time of the event that triggered this request should be passed as *arg4*.

XmTextGetEditable/2

XmTextGetEditable(_Editable, _Widget)

arg1 : free : atom (Boolean)
arg2 : ground : pointer

Checks whether widget *arg2* is editable. The Boolean answer is returned in *arg1*.

XmTextSetEditable/2

XmTextSetEditable(_Widget, _Editable)

arg1 : ground : pointer
arg2 : ground : atom (Boolean)

The edit permission of widget *arg1* is set to *arg2*.

XmTextGetMaxLength/2

XmTextGetMaxLength(_Length, _Widget)

arg1 : free : integer
arg2 : ground : pointer

The maximum allowed length for text in widget *arg2* is returned in *arg1*.

XmTextSetMaxLength/2

XmTextSetMaxLength(_Widget, _Length)

arg1 : ground : pointer
arg2 : ground : integer

The maximum allowed length for text in widget *arg1* is set to *arg2*.

XmTextGetString/2

XmTextGetString(_String, _Widget)

arg1 : free : atom
arg2 : ground : pointer

The string value of widget *arg2* is returned in *arg1*.

XmTextSetString/2

XmTextSetString(_Widget, _String)

arg1 : ground : pointer
arg2 : ground : atom

The string value of widget *arg1* is set to *arg2*.

XmTextReplace/4

XmTextReplace(_Widget, _First, _Last, _Text)

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

The text from position *arg2* to position *arg3* in widget *arg1* is replaced by *arg4*.

Scrollbar

XmScrollBarGetValues/5

XmScrollBarGetValues(_Widget, _Value, _SliderSize, _Increment, _PageIncrement)

arg1 : ground : pointer
arg2 : free : integer
arg3 : free : integer
arg4 : free : integer
arg5 : free : integer

The scrollbar state of widget *arg1* is returned. Its current value is *arg2*, its slider size *arg3*, the button increment *arg4* and the page increment *arg5*.

XmScrollBarSetValues/6

XmScrollBarSetValues(_Widget, _Value, _SliderSize, _Increment, _PageIncrement, _Notify)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : integer
arg5 : ground : integer
arg6 : ground : atom (Boolean)

The scrollbar state in widget *arg1* is set. Its current value is *arg2*, its slider size *arg3*, the button increment *arg4* and the page increment *arg5*. A change in value is indicated in *arg6*. If this is True, the ValueChanged callback will be activated.

List

XmListAddItem/3

XmListAddItem(_Widget, _Item, _Position)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : integer

The xmString item *arg2* is added to the list *arg1* at position *arg3*. If it matches an item on the selected items list, it is selected.

XmListAddItemUnselected/3

XmListAddItemUnselected(_Widget, _Item, _Position)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : integer

The xmString item *arg2* is added to the list *arg1* at position *arg3*.

XmListDeleteItem/2

XmListDeleteItem(_Widget, _Item)

arg1 : ground : pointer
arg2 : ground : pointer

The xmString item *arg2* is deleted from the list *arg1*.

XmListSetItem/2

XmListSetItem(_Widget, _Item)

arg1 : ground : pointer
arg2 : ground : pointer

The xmString item *arg2* in the list *arg1*, is made the first visible item.

XmListSetBottomItem/2

XmListSetBottomItem(_Widget, _Item)

arg1 : ground : pointer
arg2 : ground : pointer

The xmString item *arg2* in the list *arg1*, is made the last visible item.

XmListSelectItem/3

XmListSelectItem(_Widget, _Item, _Notify)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : atom (Boolean)

The xmString item *arg2* in the list *arg1*, is added to the selected item set. If *arg3* is True, the Selected callback is activated.

XmListDeselectItem/2

XmListDeselectItem(_Widget, _Item)

arg1 : ground : pointer
arg2 : ground : pointer

The xmString item *arg2* in the list *arg1*, is removed from the selected item set.

XmListDeselectAllItems/1

XmListDeselectAllItems(_Widget)

arg1 : ground : pointer

The selected item set of the list *arg1*, is made empty.

XmListItemExists/3

XmListItemExists(_Exists, _Widget, _Item)

arg1 : free : atom
arg2 : ground : pointer
arg3 : ground : pointer

Arg1 indicates existence of the xmString item *arg3* in the list *arg2*.

XmListDeletePos/2

XmListDeletePos(_Widget, _Position)

arg1 : ground : pointer
arg2 : ground : integer

The item at position *arg2* is deleted from the list *arg1*.

XmListSetPos/2

XmListSetPos(_Widget, _Position)

arg1 : ground : pointer
arg2 : ground : integer

The item at position *arg2* in the list *arg1*, is made the first visible item.

XmListSetBottomPos/2

XmListSetBottomPos(_Widget, _Position)

arg1 : ground : pointer
arg2 : ground : integer

The item at position *arg2* in the list *arg1*, is made the last visible item.

XmListSetHorizPos/2

XmListSetHorizPos(_Widget, _Position)

arg1 : ground : pointer
arg2 : ground : integer

The scrollbar of the list *arg1*, is moved to position *arg2*.

XmListSelectPos/3

XmListSelectPos(_Widget, _Position, _Notify)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : atom (Boolean)

The item at position *arg2* in the list *arg1*, is added to the selected item set. If *arg3* is True, the Selected callback is activated.

Toggle button

XmToggleButtonGetState/2

XmToggleButtonGetState(_State, _Widget)

arg1 : free : atom (Boolean)
arg2 : ground : pointer

*Arg*1 is set to the state of widget *arg2*, which is True if it is selected, and False otherwise.

XmToggleButtonSetState/3

XmToggleButtonSetState(_Widget, _State, _Notify)

arg1 : ground : pointer
arg2 : ground : atom (Boolean)
arg3 : ground : atom (Boolean)

The state of widget *arg1*, is set to *arg2*. If *arg3* is True, the ValueChanged callback is activated.

Cascade button

XmCascadeButtonHighlight/2

XmCascadeButtonHighlight(_Widget, _Highlight)

arg1 : ground : pointer
arg2 : ground : atom (Boolean)

Sets the highlight state of widget *arg1* to *arg2*.

1.5 Strings

A number of predicates handle a special Motif string type. In ***ProLog by BIM*** these strings are manipulated by their handle.

XmStringCreate/3

XmStringCreate(_String, _Text, _CharacterSet)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom (XmStringCharSet)

A new xmString is created. Its handle is returned as *arg1*. The text of the string is *arg2*, and it is in character set *arg3*.

XmStringCreateLtoR/3

XmStringCreateLtoR(_String, _Text, _CharacterSet)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom (XmStringCharSet)

A new left-to-right xmString is created. Its handle is returned as *arg1*. The text of the string is *arg2*, and it is in character set *arg3*.

XmStringGetLtoR/4

XmStringGetLtoR(_Return, _String, _CharacterSet, _Text)

arg1 : free : atom (Boolean)
arg2 : ground : pointer
arg3 : ground : atom (XmStringCharSet)
arg4 : free : atom

The text of xmString *arg2* for character set *arg3* is retrieved in *arg4*. The predicate returns a Boolean True or False in *arg1* if the retrieval succeeded or failed.

XmStringFree/1

XmStringFree(_String)

arg1 : ground : pointer

The xmString *arg1* is deleted and the memory it used is freed.

1.6 Callbacks

The interface maps callback predicates to callback C routines, since the Xt toolkit can only handle C routines as callbacks. This mapping is performed as transparently as possible to the application programmer. A single callback can be passed immediately. A list of callbacks must be constructed before being passed to the toolkit.

1.7 Attribute manipulations

Motif attributes are treated in the standard Xt manner.

The table below lists all defined attributes with their names and value types. The types and mapping to ***ProLog by BIM*** are explained in the next section.

Name	Value type
XmNaccelerators	XtTranslations
XmNancestorSensitive	Boolean
XmNbackground	Pixel
XmNbackgroundPixmap	Pixmap
XmNborderColor	Pixel
XmNborderPixmap	Pixmap
XmNborderWidth	Dimension
XmNcolormap	Colormap
XmNdepth	integer
XmNdestroyCallback	callback(XmAnyCallback)
XmNheight	Dimension
XmNmappedWhenManaged	Boolean
XmNscreen	pointer
XmNsensitive	Boolean
XmNtranslations	XtTranslations
XmNwidth	Dimension
XmNx	Position
XmNy	Position

XmNallowShellResize	Boolean
XmNcreatePopupChildProc	callback (XmCreatePopupChildProc)
XmNgeometry	String
XmNoverrideRedirect	Boolean
XmNpopdownCallback	callback(XmAnyCallback)
XmNpopupCallback	callback(XmAnyCallback)
XmNsaveUnder	Boolean
XmNheightInc	integer
XmNiconMask	Pixmap
XmNiconPixmap	Pixmap
XmNiconWindow	Window
XmNiconX	integer
XmNiconY	integer
XmNinitialState	enum(XmWmState)
XmNinput	Boolean
XmNmaxAspectX	integer
XmNmaxAspectY	integer
XmNmaxHeight	integer
XmNmaxWidth	integer
XmNminAspectX	integer
XmNminAspectY	integer
XmNminHeight	integer
XmNminWidth	integer
XmNtitle	String
XmNtransient	Boolean
XmNwaitForWm	Boolean
XmNwidthInc	integer
XmNwindowGroup	XID
XmNwmTimeout	integer
XmNdeleteResponse	enum(XmDeleteResponse)
XmNkeyboardFocusPolicy	enum(XmKeyboardFocusPolicy)
XmNmwmDecorations	integer
XmNmwmFunctions	integer
XmNmwmInputMode	integer
XmNmwmMenu	String
XmNshellUnitType	enum(XmUnitType)
XmNiconic	Boolean
XmNiconName	String
XmNargc	integer
XmNargv	pointer
XmNbottomShadowColor	Pixel
XmNbottomShadowPixmap	Pixmap
XmNforeground	Pixel
XmNhelpCallback	callback(XmAnyCallback)
XmNhighlightColor	Pixel
XmNhighlightPixmap	Pixmap
XmNshadowThickness	short

XmNtopShadowColor	Pixel
XmNtopShadowPixmap	Pixmap
XmNunitType	enum(XmUnitType)
XmNuserData	pointer
XmNallowOverlap	Boolean
XmNautoUnmanage	Boolean
XmNbuttonFontList	pointer
XmNcancelButton	Widget
XmNdefaultButton	Widget
XmNdefaultPosition	Boolean
XmNdialogStyle	enum(XmDialogStyle)
XmNdialogTitle	pointer
XmNlabelFontList	pointer
XmNmapCallback	callback(XmAnyCallback)
XmNmarginHeight	short
XmNmarginWidth	short
XmNnoResize	Boolean
XmNresizePolicy	enum(XmResizePolicy)
XmNshadowType	enum(XmShadowType)
XmNtextFontList	pointer
XmNtextTranslations	XtTranslations
XmNunmapCallback	callback(XmAnyCallback)
XmNapplyCallback	callback(XmAnyCallback)
XmNapplyLabelString	pointer
XmNcancelCallback	callback(XmAnyCallback)
XmNcancelLabelString	pointer
XmNdialogType	enum(XmDialogType)
XmNhelpLabelString	pointer
XmNlistItemCount	integer
XmNlistItems	pointer
XmNlistLabelString	pointer
XmNlistVisibleItemCount	integer
XmNminimizeButtons	Boolean
XmNmustMatch	Boolean
XmNnoMatchCallback	callback(XmAnyCallback)
XmNokCallback	callback(XmAnyCallback)
XmNokLabelString	pointer
XmNselectionLabelString	pointer
XmNtextAccelerators	XtTranslations
XmNtextColumns	integer
XmNtextString	pointer
XmNdirMask	pointer
XmNdirSpec	pointer
XmNfileSearchProc	proc(XmFileSearchProc)
XmNfilterLabelString	pointer
XmNlistUpdated	Boolean
XmNcommand	pointer
XmNcommandChangedCallback	callback(XmAnyCallback)
XmNcommandEnteredCallback	callback(XmAnyCallback)
XmNhistoryItems	pointer

XmNhistoryItemCount	integer
XmNhistoryMaxItems	integer
XmNhistoryVisibleItemCount	integer
XmNpromptString	pointer
XmNdefaultButtonType	enum (XmMessageBoxDefaultButton)
XmNmessageAlignment	enum(XmAlignment)
XmNmessageString	pointer
XmNsymbolPixmap	Pixmap
XmNbottomAttachment	enum(XmAttachment)
XmNbottomOffset	integer
XmNbottomPosition	integer
XmNbottomWidget	Widget
XmNleftAttachment	enum(XmAttachment)
XmNleftOffset	integer
XmNleftPosition	integer
XmNleftWidget	Widget
XmNresizable	Boolean
XmNrightAttachment	enum(XmAttachment)
XmNrightOffset	integer
XmNrightPosition	integer
XmNrightWidget	Widget
XmNtopAttachment	enum(XmAttachment)
XmNtopOffset	integer
XmNtopPosition	integer
XmNtopWidget	Widget
XmNfractionBase	integer
XmNhorizontalSpacing	integer
XmNrubberPositioning	Boolean
XmNverticalSpacing	integer
XmNdecimalPoints	short
XmNfontList	Boolean
XmNhighlightThickness	short
XmNscaleHeight	Dimension
XmNscaleWidth	Dimension
XmNshowValue	Boolean
XmNtitleString	pointer
XmNtraversalOn	Boolean
XmNallowResize	Boolean
XmNpaneMaximum	integer
XmNpaneMinimum	integer
XmNskipAdjust	Boolean
XmNrefigureMode	Boolean
XmNsashHeight	Dimension
XmNsashIndent	Position
XmNsashShadowThickness	integer
XmNsashWidth	Dimension
XmNseparatorOn	Boolean
XmNspacing	integer

XmNexposeCallback	callback(XmAnyCallback)
XmNinputCallback	callback(XmAnyCallback)
XmNresizeCallback	callback(XmAnyCallback)
XmNadjustLast	Boolean
XmNadjustMargin	Boolean
XmNentryAlignment	enum(XmAlignment)
XmNentryBorder	short
XmNentryCallback	callback(XmRowColumnCallback)
XmNentryClass	pointer
XmNisAligned	Boolean
XmNisHomogeneous	Boolean
XmNmapCallback	callback(XmRowColumnCallback)
XmNmenuAccelerator	String
XmNmenuHelpWidget	Widget
XmNmenuHistory	Widget
XmNmnemonic	char
XmNnumColumns	short
XmNorientation	enum(XmOrientation)
XmNpacking	enum(XmPacking)
XmNpopupEnabled	Boolean
XmNradioAlwaysOne	Boolean
XmNradioBehavior	Boolean
XmNresizeHeight	Boolean
XmNresizeWidth	Boolean
XmNrowColumnType	enum(XmRowColumnType)
XmNsubMenuId	Widget
XmNunmapCallback	callback(XmRowColumnCallback)
XmNwhichButton	integer
XmNmenuCursor	String
XmNclipWindow	Widget
XmNhorizontalScrollBar	Widget
XmNscrollBarDisplayPolicy	enum(XmScrollBarDisplayPolicy)
XmNscrollBarPlacement	enum(XmScrollBarPlacement)
XmNscrolledWindowMarginHeight	Dimension
XmNscrolledWindowMarginWidth	Dimension
XmNscrollingPolicy	enum(XmScrollingPolicy)
XmNverticalScrollBar	Widget
XmNvisualPolicy	enum(XmVisualPolicy)
XmNworkWindow	Widget
XmNcommandWindow	Widget
XmNmainWindowMarginHeight	Dimension
XmNmainWindowMarginWidth	Dimension
XmNmenuBar	Widget
XmNshowSeparator	Boolean
XmNaccelerator	String
XmNacceleratorText	pointer
XmNalignment	enum(XmAlignment)
XmNlabelInsensitivePixmap	Pixmap
XmNlabelPixmap	Pixmap
XmNlabelString	pointer
XmNlabelType	enum(XmLabelType)

XmNmarginBottom	short
XmNmarginLeft	short
XmNmarginRight	short
XmNmarginTop	short
XmNrecomputeSize	Boolean
XmNstringDirection	enum(XmStringDirection)
XmNactivateCallback	callback(XmAnyCallback)
XmNarmCallback	callback(XmAnyCallback)
XmNdisarmCallback	callback(XmAnyCallback)
XmNpushButtonEnabled	Boolean
XmNautoShowCursorPosition	Boolean
XmNcursorPosition	pointer
XmNeditable	Boolean
XmNeditMode	enum(XmEditMode)
XmNfocusCallback	callback(XmAnyCallback)
XmNlosingFocusCallback	callback(XmAnyCallback)
XmNmaxLength	integer
XmNmodifyVerifyCallback	callback(XmAnyCallback)
XmNmotionVerifyCallback	callback(XmAnyCallback)
XmNtopCharacter	pointer
XmNpendingDelete	Boolean
XmNselectionArray	pointer
XmNselectThreshold	integer
XmNblinkRate	integer
XmNcolumns	short
XmNcursorPositionVisible	Boolean
XmNrows	short
XmNwordWrap	Boolean
XmNscrollHorizontal	Boolean
XmNscrollLeftSide	Boolean
XmNscrollTopSide	Boolean
XmNscrollVertical	Boolean
XmNdecrementCallback	callback(XmAnyCallback)
XmNdragCallback	callback(XmAnyCallback)
XmNincrement	integer
XmNincrementCallback	callback(XmAnyCallback)
XmNinitialDelay	integer
XmNmaximum	integer
XmNminimum	integer
XmNorientation	enum(XmOrientation)
XmNpageDecrementCallback	callback(XmAnyCallback)
XmNpageIncrement	integer
XmNprocessingDirection	enum(XmProcessingDirection)
XmNrepeatDelay	integer
XmNshowArrows	Boolean
XmNsliderSize	integer
XmNtoBottomCallback	callback(XmAnyCallback)
XmNtoTopCallback	callback(XmAnyCallback)
XmNvalueChangedCallback	callback(XmAnyCallback)

XmNautomaticSelection	Boolean
XmNbrowseSelectionCallback	callback(XmAnyCallback)
XmNdefaultActionCallback	callback(XmAnyCallback)
XmNdoubleClickInterval	integer
XmNextendedSelectionCallback	callback(XmAnyCallback)
XmNitemCount	integer
XmNitems	pointer
XmNlistMarginHeight	Dimension
XmNlistMarginWidth	Dimension
XmNlistSpacing	short
XmNmultipleSelectionCallback	callback(XmAnyCallback)
XmNselectedItemCount	integer
XmNselectedItems	pointer
XmNselectionPolicy	enum(XmSelectionPolicy)
XmNsingleSelectionCallback	callback(XmAnyCallback)
XmNvisibleItemCount	integer
XmNlistSizePolicy	enum(XmListSizePolicy)
XmNfillOnSelect	Boolean
XmNindicatorOn	Boolean
XmNindicatorType	enum(XmIndicatorType)
XmNselectColor	Pixel
XmNselectInsensitivePixmap	Pixmap
XmNselectPixmap	Pixmap
XmNset	Boolean
XmNvisibleWhenOff	Boolean
XmNarmColor	Pixel
XmNarmPixmap	Pixmap
XmNfillOnArm	Boolean
XmNshowAsDefault	short
XmNcascadingCallback	callback(XmAnyCallback)
XmNcascadePixmap	Pixmap
XmNmappingDelay	integer
XmNarrowDirection	enum(XmArrowDirection)
XmNmargin	short
XmNseparatorType	enum(XmSeparatorType)
XmNvalue	StringInteger

1.8 Special types

The following sections describe the predefined special types that are specific to Motif.

Simple types

A number of simple types are mapped directly to ***ProLog by BIM*** simple terms.

Type	Prolog type
String	atom
Dimension	integer
Position	integer
Pixel	integer
XID	pointer
Widget	pointer
Window	pointer
StringInteger	atom or integer

Enumeration types

Enumeration types are mapped to ***ProLog by BIM*** atoms. Values of such a type are represented by the corresponding symbolic name, both in going to the external predicate and in coming back from it. The table below lists the defined Motif specific enumeration types and their symbolic values.

Type	Values
XmStringCharSet	XmSTRING_DEFAULT_CHARSET XmSTRING_ISO8859_1 XmSTRING_OS_CHARSET
XmSelectionBoxDialogComponent	XmDIALOG_APPLY_BUTTON XmDIALOG_CANCEL_BUTTON XmDIALOG_DEFAULT_BUTTON XmDIALOG_HELP_BUTTON XmDIALOG_LIST XmDIALOG_LIST_LABEL XmDIALOG_OK_BUTTON XmDIALOG_SELECTION_LABEL XmDIALOG_SEPARATOR XmDIALOG_TEXT XmDIALOG_WORK_AREA
XmFileSelectionBoxDialogComponent	XmDIALOG_APPLY_BUTTON XmDIALOG_CANCEL_BUTTON XmDIALOG_DEFAULT_BUTTON XmDIALOG_FILTER_LABEL XmDIALOG_FILTER_TEXT XmDIALOG_HELP_BUTTON XmDIALOG_LIST XmDIALOG_OK_BUTTON XmDIALOG_SELECTION_LABEL XmDIALOG_TEXT

XmMessageBoxDialogComponent	XmDIALOG_CANCEL_BUTTON XmDIALOG_DEFAULT_BUTTON XmDIALOG_HELP_BUTTON XmDIALOG_MESSAGE_LABEL XmDIALOG_OK_BUTTON XmDIALOG_SEPARATOR XmDIALOG_SYMBOL_LABEL
XmCommandDialogComponent	XmDIALOG_COMMAND_TEXT XmDIALOG_PROMPT_LABEL XmDIALOG_HISTORY_LIST

Callback handler types

The Motif interface has a number of predefined external callback handler types.

Type	Usage
XmAnyCallback	General callback handler
XmCreatePopupChildProc	Handler at pop-up of shell
XmRowColumnCallback	Row Column callback handler

Structured types

External structured types must be created and deleted explicitly. Once created (either from Prolog or from an external routine), its fields can be modified or retrieved randomly. This is done with the Xt package. The defined Motif specific structures are listed below.

External data types

The following table lists the defined external data types.

Type	Fields	Prolog field type
XmAnyCallbackStruct		
	reason	pointer
	event	pointer
XmArrowButtonCallbackStruct		
	reason	pointer
	event	pointer
	click_count	integer
XmDrawingAreaCallbackStruct		
	reason	pointer
	event	pointer
	window	pointer
XmDrawnButtonCallbackStruct		
	reason	pointer
	event	pointer
	window	pointer
	click_count	integer
XmPushButtonCallbackStruct		
	reason	pointer
	event	pointer
	click_count	integer

XmRowColumnCallbackStruct

reason	pointer
event	pointer
widget	pointer
data	pointer

XmScrollBarCallbackStruct

reason	pointer
event	pointer
value	integer
pixel	integer

XmToggleButtonCallbackStruct

reason	pointer
event	pointer
set	integer

XmListCallbackStruct

reason	pointer
event	pointer
item	pointer
item_length	integer
item_position	integer
selected_items	pointer
selected_item_count	integer
selected_item_positions	pointer
selection_type	char

XmSelectionBoxCallbackStruct

reason	pointer
event	pointer
value	pointer
length	integer

XmCommandCallbackStruct

reason	pointer
event	pointer
value	pointer
length	integer

XmFileSelectionBoxCallbackStruct

reason	pointer
event	pointer
value	pointer
length	integer
mask	pointer
mask_length	integer
dir	pointer
dir_length	integer
pattern	pointer
pattern_length	integer

XmScaleCallbackStruct

reason	pointer
event	pointer
value	integer

XmTextVerifyCallbackStruc

reason	pointer
event	pointer
doit	integer
currInsert	integer
newInsert	integer
startPos	integer
endPos	integer
text	pointer

Windowing and Graphics Libraries

Chapter 2
Interface to X Toolkit Intrinsics (Xt)

Certain user interface toolkits are based on the *X Toolkit Intrinsics* (or briefly *Xt*) package. Interfaces from ***ProLog by BIM*** to such toolkits consist of an interface to part of this package. Only the relevant portion for application programmers is provided in ***ProLog by BIM***.

This chapter describes the ***ProLog by BIM*** binding for the *X Toolkit Intrinsics* package. All predicates and attributes are briefly described. Only a basic explanation of the Xt functions is given. A complete description of this package can be found in *X Toolkit Intrinsics - C Language Interface*. An additional series of predicates is defined for manipulating data of special external types. These predicates do not have associated Xt routines.

At the end of this chapter, some instructions are given for developers of interfaces to toolkits that are based on Xt.

2.1 Availability of Xt

The Xt predicates are defined in the ***ProLog by BIM*** library **-Lwindowing/xt**.

To load the package together with ***ProLog by BIM***:

```
% BIMprolog -Lwindowing/xt
```

To consult the package interactively, from a running system:

```
?- lib( xt ) .
```

Since the Xt package is not a stand-alone toolkit, it is not very useful on its own. It is usually combined with an Xt based toolkit. These interfaces should be ready to consult the Xt package automatically when it is not yet loaded. This is the case with all Xt based toolkit interfaces provided with ***ProLog by BIM***.

Note:

Due to the limitations of the SunOs 4.0 dynamic linking package it is not possible to incrementally load the packages as explained above. Please use an extended executable as explained below.

If an extended executable containing the Motif interface has been built during the installation procedure it will also contain the interface to Xt and can be started with:

```
% BIMprologOM
```

2.2 Widget classes

The table below lists the widget classes that are provided by Xt. An Xt based toolkit defines additional widget classes.

Widget Class
shellWidgetClass
overrideShellWidgetClass
wmShellWidgetClass
transientShellWidgetClass
topLevelShellWidgetClass
applicationShellWidgetClass

2.3 Initialization and termination

Initialization of an Xt based user interface consists in its most general form of initializing the toolkit, creating an application context, opening the display and creating an application shell widget.

Toolkit initialization

XtToolkitInitialize/0

XtToolkitInitialize

Initializes the X Toolkit.

Application contexts

XtCreateApplicationContext/1

XtCreateApplicationContext(_AppContext)

arg1 : free : pointer

A new application context is created. Its handle is returned as *arg1*.

XtDestroyApplicationContext/1

XtDestroyApplicationContext(_AppContext)

arg1 : ground : pointer

The application context with handle *arg1* is destroyed.

Displays

XtDisplayInitialize/9

XtDisplayInitialize(_AppContext, _Display, _AppName, _AppClass, _Options, _NrOptions, _Argc, _Argv, _NewArgc)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : atom
arg5 : ground : pointer
arg6 : ground : integer
arg7 : ground : integer
arg8 : ground : pointer
arg9 : free : integer

The display with handle *arg2* is initialized and added to the application context with handle *arg1*. The application name and class are *arg3* and *arg4*. An option list can be passed as *arg5* with *arg6* giving the number of options in the list. The last three arguments specify command level arguments. The actual number of arguments must be passed in *arg7*, the argument list itself in *arg8*. Any Xt arguments are stripped of the argument list. The remaining number of arguments is returned as *arg9*.

XtOpenDisplay/10

XtOpenDisplay(_Display, _AppContext, _DisplayName, _AppName, _AppClass, _Options, _NrOptions, _Argc, _Argv, _NewArgc)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : atom
arg5 : ground : atom
arg6 : ground : pointer
arg7 : ground : integer
arg8 : ground : integer
arg9 : ground : integer
arg10 : free : integer

The display described by *arg3* is opened, initialized and added to the application context with handle *arg2*. The display handle is returned as *arg1*. The application name and class are *arg4* and *arg5*. An option list can be passed as *arg6* with *arg7* giving the number of options in the list. The last three arguments specify command level arguments. The actual number of arguments must be passed in *arg8*, the argument list itself in *arg9*. Any Xt arguments are stripped of the argument list. The remaining number of arguments is returned as *arg10*.

XtCloseDisplay/1

XtCloseDisplay(_Display)

arg1 : ground : pointer

The display with handle *arg1* is closed.

Application shell widget

XtAppCreateShell/7

XtAppCreateShell(_Widget, _AppName, _AppClass, _WidgetClass, _Display, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : atom
arg5 : ground : pointer
arg6 : ground : pointer
arg7 : ground : integer

A new application shell widget is created. Its handle is returned as *arg1*. The application name and class are *arg2* and *arg3*. The widget class is specified in *arg4* (see "*Widget classes*"). It is created on the display with handle *arg5*. An attribute list is passed in the last two arguments. The attribute list is *arg6*, and the number of attributes in the list is *arg7*.

XtAppCreateShell/6

XtAppCreateShell(_Widget, _AppName, _AppClass, _WidgetClass, _Display, _AttributeList)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : atom
arg5 : ground : pointer
arg6 : ground : list

A new application shell widget is created. Its handle is returned as *arg1*. The application name and class are *arg2* and *arg3*. The widget class is specified in *arg4* (see "*Widget classes*"). It is created on the display with handle *arg5*. An attribute list is passed as *arg6*. This list consists of a sequence of attribute names and their corresponding values.

XtVaAppCreateShell/6

XtVaAppCreateShell(_Widget, _AppName, _AppClass, _WidgetClass, _Display, _AttributeList)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : atom
arg5 : ground : pointer
arg6 : ground : list

This is equivalent to **XtAppCreateShell/6**.

Convenience initialization predicate

XtAppInitialize/11

XtAppInitialize(_Widget, _AppContext, _AppClass, _Options, _NrOptions,_Argc, _Argv, _NewArgc, _Resources, _Args, _NrArgs)

arg1 : free : pointer
arg2 : free : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer
arg6 : ground : integer
arg7 : ground : pointer
arg8 : free : integer
arg9 : ground : list of atom
arg10 : ground : pointer
arg11 : ground : integer

The X Toolkit is initialized. A new application context is created. Its handle is returned as *arg2*. A display is opened, initialized and added to the application context. Options can be given as a list *arg4* and number of options *arg5*. Command line arguments are given as actual number of arguments *arg6* and argument list *arg7*. The remaining number of arguments in the list, after stripping of Xt arguments, is returned in *arg8*. A list of fallback resources can be specified as *arg9*. An application shell widget is created with attribute list *arg10* and number of attributes *arg11*. The handle of this widget is returned in *arg1*.

XtAppInitialize/10

XtAppInitialize(_Widget, _AppContext, _AppClass, _Options, _NrOptions,_Argc, _Argv, _NewArgc, _Resources, _AttributeList)

arg1 : free : pointer
arg2 : free : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer
arg6 : ground : integer
arg7 : ground : pointer
arg8 : free : integer
arg9 : ground : list of atom
arg10 : ground : list

The X Toolkit is initialized. A new application context is created. Its handle is returned as *arg2*. A display is opened, initialized and added to the application context. Options can be given as a list *arg4* and number of options *arg5*. Command line arguments are given as actual number of arguments *arg6* and argument list *arg7*. The remaining number of arguments in the list, after stripping of Xt arguments, is returned in *arg8*. A list of fallback resources can be specified as *arg9*. An application shell widget is created with attribute list *arg10*. The handle of this widget is returned in *arg1*. The attribute list consists of a sequence of attribute names and their corresponding values.

XtVaAppInitialize/10

XtVaAppInitialize(_Widget, _AppContext, _AppClass, _Options,_NrOptions,_Argc, _Argv, _NewArgc, _Resources, _AttributeList)

arg1 : free : pointer
arg2 : free : pointer
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : integer
arg6 : ground : integer
arg7 : ground : pointer
arg8 : free : integer
arg9 : ground : list of atom
arg10 : ground : list

This is equivalent to **XtAppInitialize/10**.

2.4 Widget manipulation

Widget creation

XtCreateWidget/6

XtCreateWidget(_Widget, _Name, _Class, _Parent, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : pointer
arg6 : ground : integer

A new widget is created. Its handle is returned as *arg1*. The widget name and class are *arg2* and *arg3* (see "*Widget classes*"). The parent widget is *arg4*. An attribute list is passed in the last two arguments. The list itself is *arg5* and the number of attributes in the list is *arg6*.

XtCreateWidget/5

XtCreateWidget(_Widget, _Name, _Class, _Parent,_AttributeList)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : list

A new widget is created. Its handle is returned as *arg1*. The widget name and class are *arg2* and *arg3* (see "*Widget classes*"). The parent widget is *arg4*. An attribute list is passed as *arg5*. This list consists of a sequence of attribute names and their corresponding values.

XtVaCreateWidget/5

XtVaCreateWidget(_Widget, _Name, _Class, _Parent,_AttributeList)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : list

This is equivalent to **XtCreateWidget/5**.

XtCreateManagedWidget/6

XtCreateManagedWidget(_Widget, _Name, _Class, _Parent, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : pointer
arg6 : ground : integer

A new widget is created and managed. Its handle is returned as *arg1*. The widget name and class are *arg2* and *arg3* (see "*Widget classes*"). The parent widget is *arg4*. An attribute list is passed in the last two arguments. The list itself is *arg5* and the number of attributes in the list is *arg6*.

XtCreateManagedWidget/5

XtCreateManagedWidget(_Widget, _Name, _Class, _Parent, _AttributeList)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : list

A new widget is created and managed. Its handle is returned as *arg1*. The widget name and class are *arg2* and *arg3* (see "*Widget classes*"). The parent widget is *arg4*. An attribute list is passed as *arg5*. This list consists of a sequence of attribute names and their corresponding values.

XtVaCreateManagedWidget/5

XtVaCreateManagedWidget(_Widget, _Name, _Class, _Parent, _AttributeList)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : list

This is equivalent to **XtCreateManagedWidget/5**.

Widget destruction

XtDestroyWidget/1

XtDestroyWidget(_Widget)

arg1 : ground : pointer

The widget with handle *arg1* is destroyed.

Composite widget managing

XtManageChild/1

XtManageChild(_Widget)

arg1 : ground : pointer

The widget with handle *arg1* is managed in its composite parent.

XtManageChildren/2

XtManageChildren(_WidgetList, _NrChildren)

arg1 : ground : pointer
arg2 : ground : integer

The widget list with handles *arg1*, and *arg2* elements, is managed in its composite parent.

XtUnmanageChild/1

XtUnmanageChild(_Widget)

arg1 : ground : pointer

The widget with handle *arg1* is unmanaged from its composite parent.

XtUnmanageChildren/2

XtUnmanageChildren(_WidgetList, _NrChildren)

arg1 : ground : pointer
arg2 : ground : integer

The widget list with handles *arg1*, and *arg2* elements, is unmanaged from its composite parent.

Widget realization and mapping

XtRealizeWidget/1

XtRealizeWidget(_Widget)

arg1 : ground : pointer

The widget with handle *arg1* is realized.

XtUnrealizeWidget/1

XtUnrealizeWidget(_Widget)

arg1 : ground : pointer

The widget with handle *arg1* is unrealized.

XtMapWidget/1

XtMapWidget(_Widget)

arg1 : ground : pointer

The widget with handle *arg1* is mapped.

XtUnmapWidget/1

XtUnmapWidget(_Widget)

arg1 : ground : pointer

The widget with handle *arg1* is unmapped.

XtSetMappedWhenManaged/2

XtSetMappedWhenManaged(_Widget, _Set)

arg1 : ground : pointer
arg2 : ground : atom (Boolean)

The switch for mapping the widget when it is managed, of the widget with handle *arg1* is set to *arg2* (see "*Special types - Boolean*" for type Boolean).

2.5 Pop-ups

Pop-up windows are composed of a pop-up shell widget and one child widget. The pop-up shell is created with one of the following predicates. Its child is created with the general widget creation predicates. A pop-up is mapped and unmapped with dedicated predicates.

Pop-up shell creation

XtCreatePopupShell/6

XtCreatePopupShell(_Widget, _Name, _Class, _Parent, _Args, _NrArgs)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : pointer
arg6 : ground : integer

A new pop-up shell widget is created. Its handle is returned as *arg1*. The widget name and class are *arg2* and *arg3* (see "*Widget Classes*" on page 9-61). An attribute list is passed in the last two arguments. The list itself is *arg5* and the number of attributes in the list is *arg6*.

XtCreatePopupShell/5

XtCreatePopupShell(_Widget, _Name, _Class, _Parent, _AttributeList)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : list

A new pop-up shell widget is created. Its handle is returned as *arg1*. The widget name and class are *arg2* and *arg3* (see "*Widget classes*"). An attribute list is passed as *arg5*. This list consists of a sequence of attribute names and their corresponding values.

XtVaCreatePopupShell/5

XtVaCreatePopupShell(_Widget, _Name, _Class, _Parent, _AttributeList)

arg1 : free : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer
arg5 : ground : list

This is equivalent to **XtCreatePopupShell/5**.

Mapping pop-ups from the application

XtPopup/2

XtPopup(_Widget, _GrabKind)

arg1 : ground : pointer
arg2 : ground : atom (XtGrabKind)

The pop-up shell widget with handle *arg1* is popped-up with grab kind *arg2* (see "*Special types - Enumeration types*" on for type XtGrabKind.).

XtPopupSpringLoaded/1

XtPopupSpringLoaded(_Widget)

arg1 : ground : pointer

The pop-up shell widget with handle *arg1* is popped up as a spring-loaded pop-up.

XtPopdown/1

XtPopdown(_Widget)

arg1 : ground : pointer

The pop-up shell widget with handle *arg1* is popped down.

Mapping pop-ups from a callback

XtCallbackNone/3

XtCallbackNone(_Widget, _PopUpWidget, _CallData)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : pointer

The pop-up shell widget with handle *arg2* is popped up with no grabbing from the callback from the widget with handle *arg1*. Data that must be passed with this callback are given in *arg3*.

XtCallbackNonexclusive/3

XtCallbackNonexclusive(_Widget, _PopUpWidget, _CallData)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : pointer

The pop-up shell widget with handle *arg2* is popped up with non-exclusive grabbing, from the callback from the widget with handle *arg1*. Data that must be passed with this callback are given in *arg3*.

XtCallbackExclusive/3

XtCallbackExclusive(_Widget, _PopUpWidget, _CallData)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : pointer

The pop-up shell widget with handle *arg2* is popped up with exclusive grabbing, from the callback from the widget with handle *arg1*. Data that must be passed with this callback are given in *arg3*.

XtCallbackPopdown/3

XtCallbackPopdown(_Widget, _PopdownID, _CallData)

arg1 : ground : pointer
arg2 : ground : pointer (XtPopdownIDRec)
arg3 : ground : pointer

The pop-up shell widget, popped-up from a callback, as described in *arg2* (see "*Special types - External data types*" for type XtPopdownIDRec), is popped down. Data that must be passed with this callback are given in *arg3*.

2.6 Conversion predicates

A number of predicates is provided for converting different identifiers of objects, and for retrieving associated objects from an object.

Identifier conversion

XtName/2

XtName(_Name, _Widget)

arg1 : free : atom
arg2 : ground : pointer

Arg1 is instantiated to the name of the widget with handle *arg2*.

XtNameToWidget/3

XtNameToWidget(_Widget, _RefWidget, _Name)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom

Arg1 is instantiated to the handle of the widget with name *arg3*, starting from the reference widget with handle *arg2*.

Retrieval of associated objects

XtWidgetToApplicationContext/2

XtWidgetToApplicationContext(_AppContext, _Widget)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the application context handle of the widget with handle *arg2*.

XtDisplay/2

XtDisplay(_Display, _Widget)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the display handle of the widget with handle *arg2*.

XtParent/2

XtParent(_Parent, _Widget)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the parent widget handle of the widget with handle *arg2*.

XtScreen/2

XtScreen(_Screen, _Widget)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the screen handle of the widget with handle *arg2*.

XtWindow/2

XtWindow(_Window, _Widget)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the window handle of the widget with handle *arg2*.

XtDisplayOfObject/2

XtDisplayOfObject(_Display, _Object)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the display handle of the object with handle *arg2*.

XtScreenOfObject/2

XtScreenOfObject(_Screen,_Object)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the screen handle of the object with handle *arg2*.

XtWindowOfObject/2

XtWindowOfObject(_Window,_Object)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the window handle of the object with handle *arg2*.

2.7 Geometry management

A number of geometry management predicates is provided.

Geometry requests

XtMakeGeometryRequest/4

XtMakeGeometryRequest(_Result, _Widget, _Request, _Reply)

arg1 : free : atom (XtGeometryResult)
arg2 : ground : pointer
arg3 : ground : pointer (XtWidgetGeometry)
arg4 : ground : pointer (XtWidgetGeometry)

A geometry request is made for the widget with handle *arg2*. The requested geometry is passed in *arg3*. The reply geometry is returned in *arg4*. The result of the request is returned as *arg1*.

XtMakeResizeRequest/6

XtMakeResizeRequest(_Result, _Widget, _ReqWidth, _ReqHeight, _RepWidth, _RepHeight)

arg1 : free : atom (XtGeometryResult)
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : integer
arg5 : free : integer
arg6 : free : integer

A resize request is made for the widget with handle *arg2*. The requested width and height is passed in *arg3* and *arg4*. The reply width and height is returned in *arg5* and *arg6*. The result of the request is returned as *arg1*.

XtQueryGeometry/4

XtQueryGeometry(_Result, _Widget, _Intended, _Preferred)

arg1 : free : atom (XtGeometryResult)
arg2 : ground : pointer
arg3 : ground : pointer (XtWidgetGeometry)
arg4 : ground : pointer (XtWidgetGeometry)

A geometry query is made for the widget with handle *arg2*. The intended geometry change is passed in *arg3*. The preferred geometry is returned in *arg4*. The result of the request is returned as *arg1*.

Geometry changes

XtMoveWidget/3

XtMoveWidget(_Widget, _X, _Y)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer

The widget with handle *arg1* is moved to position *arg2*, *arg3*.

XtResizeWidget/4

XtResizeWidget(_Widget, _Width, _Height, _BorderWidth)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : integer

The widget with handle *arg1* is resized to width *arg2*, height *arg3* and border width *arg4*.

XtConfigureWidget/6

XtConfigureWidget(_Widget, _X, _Y, _Width, _Height, _BorderWidth)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : integer
arg5 : ground : integer
arg6 : ground : integer

The widget with handle *arg1* is moved to position *arg2*, *arg3* and resized to width *arg4*, height *arg5* and border width *arg6*.

XtResizeWindow/1

XtResizeWindow(_Widget)

arg1 : ground : pointer

The widget with handle *arg1* is resized to its current dimensions.

2.8 Event management

The event management predicates include a set of predicates for querying and manipulating the event queue. Another set can be used to register or unregister non-X event handlers. Some predicates handle the event sensitivity of widgets.

Event queue manipulation

XtAppPending/2

XtAppPending(_InputMask, _AppContext)

arg1 : free : integer (XtInputMask)
arg2 : ground : pointer

This predicate returns immediately if there is an event pending for the application context with handle *arg2*. The bit mask specifying which types of events are pending, is returned as *arg1* (see "*Special types - Bit mask types*" for bit mask type XtInputMask.). If no event is pending, the output buffers are flushed and the predicate returns with *arg1* instantiated to zero.

XtAppPeekEvent/3

XtAppPeekEvent(_Result, _AppContext, _Event)

arg1 : free : atom (Boolean)
arg2 : ground : pointer
arg3 : ground : pointer

If there is an event pending for the application context with handle *arg2*, *arg3* is filled in with the next event and that event is removed from the queue. If no event is pending, the output buffers are flushed and the predicate blocks until input is available. If that input is an event, *arg3* is filled in with that event and the event is removed from the queue. Otherwise, the input is for an alternate source and the predicate returns *False* as *arg1*. *Arg1* is *True* if an event is returned.

XtAppNextEvent/2

XtAppNextEvent(_AppContext, _Event)

arg1 : ground : pointer
arg2 : ground : pointer

If there is an event pending for the application context with handle *arg1*, *arg2* is filled in with that event, which is removed from the queue. If no event is pending, the output buffers are flushed and the predicate blocks until an event is available. Any non-X event input is also handled in the mean time.

XtAppProcessEvent/2

XtAppProcessEvent(_AppContext, _InputMask)

arg1 : ground : pointer
arg2 : ground : integer (XtInputMask)

Processes the next event of type *arg2* for the application context with handle *arg1*. If no such event is available, the predicate waits until an event of the specified type occurs (see "*Special types - Bit mask types*" for bit mask type XtInputMask.).

XtDispatchEvent/2

XtDispatchEvent(_Result, _Event)

arg1 : free : atom (Boolean)
arg2 : ground : pointer

The event with handle *arg2* is dispatched to an appropriate handler. If there is no such handler, *arg1* is *False*, otherwise it is *True*.

XtAppMainLoop/1

XtAppMainLoop(_AppContext)

arg1 : ground : pointer

This predicate continuously reads and dispatches events for the application context with handle *arg1*.

Non-X event handlers

XtAppAddInput/6

XtAppAddInput(_XtID, _AppContext, _Source, _Condition, _Predicate, _Data)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : integer (XtInputCondition)
arg5 : ground : atom
arg6 : ground : pointer

An input handler is registered. Its Xt identifier is returned in *arg1*. The application context in which it is registered has handle *arg2*. The input source has file descriptor *arg3*. An input condition can be given in *arg4* (see "*Special types*" for bit mask type XtInputCondition.) The handler predicate name is *arg5*. Client-specific data to be passed to the handler is given in *arg6*.

XtRemoveInput/1

XtRemoveInput(_XtID)

arg1 : ground : pointer

The input handler with Xt identifier *arg1* is unregistered.

XtAppAddTimeOut/5

XtAppAddTimeOut(_XtID, _AppContext, _Interval, _Predicate, _Data)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : atom
arg5 : ground : pointer

A time-out handler is registered. Its Xt identifier is returned in *arg1*. The application context in which it is registered, has handle *arg2*. The time-out interval is *arg3* milliseconds. The handler predicate name is *arg4*. Client-specific data to be passed to the handler is given in *arg5*.

XtRemoveTimeOut/1

XtRemoveTimeOut(_XtID)

arg1 : ground : pointer

The time-out handler with Xt identifier *arg1* is unregistered.

XtAppAddWorkProc/5

XtAppAddWorkProc(_XtID, _AppContext, _Predicate, _Data)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : pointer

A background work procedure is registered. Its Xt identifier is returned in *arg1*. The application context in which it is registered has handle *arg2*. The work procedure predicate name is *arg3*. Client-specific data to be passed to the handler, is given in *arg4*.

XtRemoveWorkProc/1

XtRemoveWorkProc(_XtID)

arg1 : ground : pointer

The background work procedure with Xt identifier *arg1* is unregistered.

Event sensitivity

XtSetSensitive/2

XtSetSensitive(_Widget, _Sensitive)

arg1 : ground : pointer
arg2 : ground : atom (Boolean)

The event sensitivity of the widget with handler *arg1* is set to *arg2* (see "*Special types - Boolean*" or type Boolean.).

XtIsSensitive/2

XtIsSensitive(_Sensitive, _Widget)

arg1 : free : atom (Boolean)
arg2 : ground : pointer

The event sensitivity of the widget with handler *arg2* is returned as *arg1* (see "*Special types - Boolean*" for type Boolean.).

2.9 Callbacks

The interface maps callback predicates to callback C routines, as the Xt toolkit can only handle C routines as callbacks. This mapping is performed as transparently as possible to the application programmer. A single callback can be passed immediately. A list of callbacks must be constructed before being passed to the toolkit.

Callback registration and unregistration

XtAddCallback/4

XtAddCallback(_Widget, _Name, _Predicate, _Data)

arg1 : ground : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer

A callback handler for the widget with handle *arg1* is added to its callback list with name *arg2*. The name of the callback predicate is *arg3* and the client-specific data that must be passed to it, is *arg4*.

XtAddCallbacks/3

XtAddCallbacks(_Widget, _Name, _Callbacks)

arg1 : ground : pointer
arg2 : ground : atom
arg3 : ground : pointer (XtCallbackList(_N))

A list of callback handlers for the widget with handle *arg1* is added to its callback list with name *arg2*. The list of callbacks is *arg3*. (see "*Special types* - External data types" for type XtCallbackList.)

XtRemoveCallback/4

XtRemoveCallback(_Widget, _Name, _Predicate, _Data)

arg1 : ground : pointer
arg2 : ground : atom
arg3 : ground : atom
arg4 : ground : pointer

The callback handler with predicate name *arg3* and client-specific data *arg4* are removed from the callback list with name *arg2* of the widget with handle *arg1*.

XtRemoveCallbacks/3

XtRemoveCallbacks(_Widget, _Name, _Callbacks)

arg1 : ground : pointer
arg2 : ground : atom
arg3 : ground : pointer (XtCallbackList(_N))

The list of callback handlers *arg3* is removed from the callback list with name *arg2* of the widget with handle *arg1* (see "*Special types - External data types*" for type XtCallbackList).

XtRemoveAllCallbacks/2

XtRemoveAllCallbacks(_Widget, _Name)

arg1 : ground : pointer
arg2 : ground : atom

All callback handlers are removed from the callback list with name *arg2* of the widget with handle *arg1*.

2.10 Attribute manipulation

Attribute settings are collected in an attribute list. This is an array of tuples, whose elements can be set randomly. Instead of using the *general special type predicates* (see "*External Data Manipulation*") to change the element values, the *standard Xt predicates* (such as Xt_get, Xt_set) should be used for setting the attributes in such a list. The reason is that the interface will perform the necessary conversions on these settings when the standard Xt predicate is used, while the general purpose external type set predicate does not convert its arguments.

Attribute lists can also be passed as a ***ProLog by BIM*** list. This list must be a sequence of attribute names and values, optionally terminated with 0. The Xt interface provides an additional predicate to test the correctness of such an attribute list.

To retrieve values of attributes, the names and associated ***ProLog by BIM*** variables must be passed as a list to the standard Xt predicate for retrieving resource values. The version that takes an array address and number of array elements, is also provided, but in this case, the user is responsible for allocating buffer memory, assigning addresses, and retrieving and converting values from the buffer memory.

Attribute list elements

XtSetArg/3

XtSetArg(_Argument, _Attribute, _Value)

arg1 : ground : pointer ^ integer (ArgList(_N))
arg2 : ground : atom
arg3 : ground : term

The attribute, specified by *arg1*, is set to name *arg2* and value *arg3*. The attribute specification *arg1* is a term of the form *_ArgList ^ _Index* with *_ArgList* an Arglist structure (see "*Special types - External data types*" for type Arglist) and *_Index* an index in that array (starting from 0).

Setting an attribute list

XtSetValues/3

XtSetValues(_Widget, _ArgumentList, _NrArguments)

arg1 : ground : pointer
arg2 : ground : pointer (ArgList(_N))
arg3 : ground : integer

The attribute list of *arg3* attributes, with handle *arg2*, is applied to the widget with handle *arg1*. (See "*Special Types* - External data types" on page 9-89 for type Arglist.)

XtSetValues/2

XtSetValues(_Widget, _AttributeList)

arg1 : ground : pointer
arg2 : ground : list

The attribute list of *arg2*, is applied to the widget with handle *arg1*. This list consists of a sequence of attribute names and their corresponding values.

XtVaSetValues/2

XtVaSetValues(_Widget, _AttributeList)

arg1 : ground : pointer
arg2 : ground : list

This is equivalent to **XtSetValues/2**.

Getting an attribute list

XtGetValues/3

XtGetValues(_Widget, _ArgumentList, _NrArguments)

arg1 : ground : pointer
arg2 : ground : pointer (ArgList(_N))
arg3 : ground : integer

The values of the attribute list of *arg3* attributes, with handle *arg2*, are retrieved from the widget with handle *arg1*. (See "*Special Types - External data types*" on page 9-89 for type Arglist.)

The attribute list must contain tuples consisting of attribute name and address to store the value.

XtGetValues/2

XtGetValues(_Widget, _AttributeList)

arg1 : ground : pointer
arg2 : ground : list

The values of the attribute list of *arg2*, are retrieved from the widget with handle *arg1*. This list consists of a sequence of attribute names and corresponding free variables. Upon return, the variables will be instantiated to the attribute value.

XtVaGetValues/2

XtVaGetValues(_Widget, _AttributeList)

arg1 : ground : pointer
arg2 : ground : list

This is equivalent to **XtGetValues/2**.

Attribute list checking

xt_avlist_check/1

xt_avlist_check(_AttributeList)

arg1 : ground : list

The attribute list *arg1* is verified. This predicate gives error messages about incorrect attribute lists. The attribute list is a list of symbolic attribute names and value settings.

xt_avlist_whatis/1

xt_avlist_whatis(_Type)

arg1 : ground : atom

An explanation is given of what attribute type *arg1* represents.

2.11 Special types

The Xt interface includes predicates for external data type manipulation. Some of these types can be mapped directly to ***ProLog by BIM*** types. Others require external space. This must be allocated explicitly by creating the data. The following sections describe the predefined special types.

Boolean

The Boolean type is an enumeration type which represents True/False values. It is mapped to ***ProLog by BIM*** atoms. The table below lists the recognized values.

Symbol	Value
True	1
TRUE	1
False	0
FALSE	0

On return from an external predicate, a Boolean value is always translated to the symbolic values *True* or *False* (only first letter in capital).

Bit masks

Bit masks are represented in ***ProLog by BIM*** by integers. The bits can be manipulated with the standard built-in functions. To manipulate them with symbolic names, a set of predicates is defined in the Xt interface.

xt_mask_encode/3

xt_mask_encode(_Type, _SymbolList, _Mask)

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

The symbolic bit mask list *arg2* of type *arg1* is encoded into a bit mask *arg3*. The list must contain symbols that are legal bit mask fields for the indicated type. (See page 9-86 for a list of recognized types and symbols).

xt_mask_decode/3

xt_mask_decode(_Type, _Mask, _SymbolList)

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

The bit mask *arg2* of type *arg1* is decoded into a symbolic bit mask list *arg3* (see below for a list of recognized types and symbols).

xt_mask_set/4

xt_mask_set(_Type, _Symbol, _InMask, _OutMask)

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

The bit with symbol *arg2* in bit mask *arg3* of type *arg1* is set on. The new bit mask is returned as *arg4* (see below for a list of recognized types and symbols).

xt_mask_reset/4

xt_mask_reset(_Type, _Symbol, _InMask, _OutMask)

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

The bit with symbol *arg2* in bit mask *arg3* of type *arg1* is set off. The new bit mask is returned as *arg4* (see below for a list of recognized types and symbols).

xt_mask_test/3

xt_mask_test(_Type, _Symbol, _Mask)

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

The bit with symbol *arg2* in bit mask *arg3* of type *arg1* is tested. This predicate succeeds if it is set on, and fails otherwise (see below for a list of recognized types and symbols).

Bit mask types

Below is a table of defined bit mask types with their symbolic field names.

Type	Fields
XtGeometryMask	CWX
	CWY
	CWWidth
	CWHeight
	CWBorderWidth
	CWSibling
	CWStackMode
	XtCWQueryOnly
XtStackMode	Above
	Below
	TopIf
	BottomIf
	Opposite
	XtSMDontChange
XtInputCondition	XtInputReadMask
	XtInputWriteMask
	XtInputExceptMask
XtInputMask	XtIMXEvent
	XtIMTimer
	XtIMAlternateInput
	XtIMAll

Enumeration types

Enumeration types are mapped to ***ProLog by BIM*** atoms. Values of such a type are represented by the corresponding symbolic name, as well in going to the external predicate, as in coming back from it. The table below lists the defined enumeration types and their symbolic values.

Type	Values
XtGrabKind	XtGrabNone
	XtGrabNonexclusive
	XtGrabExclusive
XtGeometryResult	XtGeometryYes
	XtGeometryNo
	XtGeometryAlmost
	XtGeometryDone

Callback handler types

The Xt interface has a number of predefined external callback handler types. These are not usually used by an application program.

Type	Usage
XtInputCallback	Input handler
XtTimerCallback	Time-out handler
XtWorkProc	Background work procedure

Structured types

External structured types must be created and destroyed explicitly. Once created, their fields can be modified or retrieved randomly. First, the external structure manipulation predicates are described. Then, the defined structures are listed.

External structure manipulation

xt_create_data/2

xt_create_data(_Handle, _Type)

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

An external data structure of type *arg2* is created. *Arg1* is instantiated to its handle.

xt_free_data/1

xt_free_data(_Handle)

arg1 : ground : pointer or term

The external data structure with handle *arg1* is destroyed.

xt_set/2

xt_set(_Variable, _Value)

arg1 : ground : term
arg2 : ground : term

The external variable described by *arg1* is assigned the value *arg2*. A variable specification consists of an external data handler possibly with a selection of fields. This selection has the form of a ^ *_Field* sequence.

xt_set/3

xt_set(_Cast, _Variable, _Value)

arg1 : ground : term
arg2 : ground : term
arg3 : ground : term

The external variable described by *arg2* is casted to type *arg1* and assigned the value *arg3*. A variable specification consists of an external data handler possibly with a selection of fields. This selection has the form of a ^ *_Field* sequence.

xt_get/2

xt_get(_Variable, _Value)

arg1 : ground : term
arg2 : free : term

The value of the external variable described by *arg1* is returned in *arg2*. A variable specification consists of an external data handler possibly with a selection of fields. This selection has the form of a ^ *_Field* sequence.

xt_get/3

xt_get(_Cast, _Variable, _Value)

arg1 : ground : term
arg2 : ground : term
arg3 : free : term

The value of the external variable described by *arg2* is casted to type *arg1* and returned in *arg3*. A variable specification consists of an external data handler with possibly a selection of fields. This selection has the form of a ^ *_Field* sequence.

External data types

The following table lists the defined external data types.

Type	Fields	Prolog Field Type
Arg	name	atom
	value	atomic
ArgList(_N)	Array of _N Arg	
XtCallbackRec	callback	atom
	closure	pointer
XtCallbackList(_N)	Array of _N XtCallbackRec	
XtPopdownIDRec	shell_widget	pointer
	enable_widget	pointer
XtWidgetGeometry	request_mode	integer (XtGeometryMask)
	x	integer
	y	integer
	width	integer
	height	integer
	border_width	integer
	sibling	pointer
	stack_mode	integer (XtStackMode)

2.12 Instructions for toolkit interface developers

When developing an interface for an Xt based toolkit, the following instructions should be taken into account if the Xt interface is used. The Xt toolkit interface provides some facilities that are handy for Xt based toolkit interfaces. This includes treatment of callbacks, widget classes, attribute lists and external data structures.

Availability

To use the Xt toolkit interface, a module interface file must be included:

```
:- include( '-Hinclude/xt.mod' ) .
```

The Xt interface must be consulted after consulting the toolkit interface.

A run-time system should include both the toolkit interface and the Xt interface. Both must be indicated as targets to **BIMlinker**. The Xt interface must appear as last target.

For a successful incremental link of the toolkit interface, the target list of the **extern_load** directive, must include the Xt interface extensions library, and the Xt library:

```
'-Lwindowing/XtExt.a' , '-lxt'
```

The libraries must appear after the toolkit library, and before the X11 library.

Initialization

The toolkit interface must provide two levels of initialization: a re-entrant initialization of external data, and initialization of Prolog data, which may assume the previous level has been executed.

External top-level initialization

win_sys_initialize/0

win_sys_initialize

This predicate must be defined in the toolkit interface (in the global module). It must initialize the external part of the interface. It will be called at runtime, either from the start-up initialization predicate, or in a run-time system, from the application. This splitting up of initialization of the external and internal part is necessary for run-time systems, as the external data is not saved during generation of such a run-time system. Therefore it must be possible to re-initialize it when the run-time system is started up.

This predicate must call the following predicate.

xt_init_extern/0

xt_init_extern

This Xt interface predicate initializes the external part of the interface.

It should be called from the toolkit external initialization predicate.

Internal top-level initialization

xt_initialize_toolkit/0

xt_initialize_toolkit

This predicate must be defined in the toolkit interface (in the global module). It must initialize the internal part of the interface. It will be called when the Xt interface is consulted. In a run-time system, it will only be called at system generation time, not when it is started up.

Example

An OSF/Motif interface, based on the Xt toolkit interface, might define the initialization predicates as follows:

```
:- global win_sys_initialize/0 .
:- extern_predicate( om_init_extern ) .
win_sys_initialize :-
     xt_init_extern,           { Xt tookit external data }
     om_init_extern .
          { OSF/Motif external data }
om_initialize :-
     om_init_types,                { External data types }
     om_init_packages,             { Widget classes }
     om_init_callback,             { Callbacks }
     om_init_proc .                { External procedures }
:- global xt_initialize_toolkit/0 .
xt_initialize_toolkit :-
     { OSF/Motif internal data initialization }
     om_initialize .
```

External data type initialization

All external data types that must be recognized by the Xt external data structure manipulation predicates, must be initialized. This is accomplished by declaring the external types with the **extern_typedef/2** built-in predicate.

Widget classes initialization

All widget classes that must be treated by the Xt package, must be initialized with the following Xt interface predicate. This must be called at start-up time.

xt_record_packages/3

xt_record_packages(_TypeNames, _ClassHandles, _NrClasses)

arg1 : ground : term of atom
arg2 : ground : term of pointer
arg3 : ground : integer

The *arg3* widget classes are registered with the Xt interface. The symbolic names of the classes must be passed in *arg1* as a term of arity *arg3*. The external class handles must be passed in a parallel term *arg2*.

Callback initialization

All callbacks that must be treated by the Xt package, must be initialized with the following Xt interface predicate. This must be called at start-up time.

xt_record_callbacks/3

xt_record_callbacks(_TypeNames, _CallbackHandlers, _NrCallbacks)

arg1 : ground : term of atom
arg2 : ground : term of pointer
arg3 : ground : integer

The *arg3* callback handler types are registered with the Xt interface. The symbolic names of the callback handlers must be passed in *arg1* as a term of arity *arg3*. The external handlers must be passed in a parallel term *arg2*. Each callback predicate of one of the mentioned types, will be mapped to the corresponding external callback handler, which is then responsible for calling the predicate.

Externally callable predicate initialization

All predicates which are callable from the toolkit, and that must be treated by the Xt package, must be initialized with the following Xt interface predicate. This must be called at start-up time.

xt_record_procs/3

xt_record_procs(_TypeNames, _ProcHandlers, _NrProcs)

arg1 : ground : term of atom
arg2 : ground : term of pointer
arg3 : ground : integer

The *arg3* predicate proc types are registered with the Xt interface. The symbolic names of the predicate procs must be passed in *arg1* as a term of arity *arg3*. The external handlers must be passed in a parallel term *arg2*. Each predicate proc of one of the mentioned types, will be mapped to the corresponding external proc handler, which is then responsible for calling the predicate.

Attribute definitions

The attributes must be defined in a uniform way in order to be handled by the Xt interface package. Each attribute must be a fact, describing how it can be converted to an external form. The enumeration type attributes must also be defined.

The file containing attribute and enumeration definitions must include the Xt attribute and enumeration definition file at the beginning of the file:

```
:- include( '-Hsrc/windowing/xt.def.pro' ) .
```

Attribute definitions

xt_attribute/3

xt_attribute(_Symbol, _Code, _ArgumentType)

arg1 : ground : atom
arg2 : ground : atom
arg3 : ground : term

This predicate defines attribute conversions. The symbolic name of the attribute is *arg1*. Its corresponding external code is *arg2*, which is normally the same as *arg1* but without the leading prefix. The conversion of its value is defined by *arg3* (see list below). There must be one fact for each attribute.

xt_encode_attributes/3

xt_encode_attributes(_AttributeList, _ArgList, _NrArgs)

arg1 : ground : list
arg2 : free : pointer (ArgList(_N))
arg3 : free : integer

The attribute list *arg1* is converted to an external attribute list. The external argument list is returned as *arg2*, the number of arguments in that list as *arg3*. These two parameters can be passed directly to **XtWidgetCreate/6** or **XtSetValues/3** or any equivalent predicates.

The table below lists the recognized attribute value conversions.

Convertor	Actual type
integer	integer
short	integer
char	atom (one character atom)
Dimension	integer
Position	integer
pointer	pointer
XID	pointer
Pixmap	pointer
Colormap	pointer
Window	pointer
Widget	pointer
XtTranslations	pointer
String	atom
Boolean	atom (Boolean)
boolean	atom (Boolean)
enum(_*Type*)	atom (value of enumeration type_*Type*)
callback(_*Type*)	pointer (XtCallbackList(_*N*))
proc(_*Type*)	atom (predicate name)

Attribute enumeration types

xt_enumeration/3

xt_enumeration(_EnumerationType, _Symbol, _Code)

arg1 : ground : atom
arg2 : ground : atom
arg3 : ground : pointer or integer

This predicate defines values and conversions for the enumeration type *arg1*. The symbolic name of a value is *arg2*. Its corresponding external code is *arg3*. There must be exactly one fact for each value of each enumeration type.

This predicate can also be called with *arg2* or *arg3* free to convert an enumeration value.

Windowing and Graphics Libraries

Chapter 3
Interface to XLib

3.1 Availability of Xlib

It is assumed that the reader has access to a Xlib Programming and Reference Manual.

The Xlib predicates are defined in the ***ProLog by BIM*** library **-Lwindowing/xlib.**

To load the package together with ***ProLog by BIM***:

```
% BIMprolog -Lwindowing/xlib
```

To consult the package interactively, from a running system:

```
?- lib( xlib ) .
```

As the Xlib package is not a stand-alone toolkit, it is not very useful on its own. It is usually combined with an Xlib-based toolkit. These interfaces should consult the Xlib package automatically when it is not yet loaded. This is the case with all Xlib based toolkit interfaces provided with ***ProLog by BIM***.

Note:

Due to the limitations of the SunOs 4.0 dynamic linking package it is not possible to incrementally load the packages as explained above. Please use an extended executable as explained below.

If an extended executable containing the Xlib interface has been build during the installation procedure it can be started with:

```
BIMprologX
```

If an extended executable containing the Motif interface has been built during the installation procedure it also contains the interface to the Xlib routines and can be started with:

```
BIMprologOM
```

3.2 Xlib predicates

Correspondence between C routines and predicates

The predicates in ***ProLog by BIM*** have the same name as the corresponding Xlib C routines. The order and the type of the arguments of a predicate are the same as for the corresponding C routine (except for structures and a few exceptions, see below).

Routines that have a return value in C have an additional first argument for the predicate in ***ProLog by BIM***. That first value is instantiated to the return value.

Symbolic constants

The symbolic names (such as CWBackPixel and CWCursor) can be used in ***ProLog by BIM*** too. In ***ProLog by BIM***, they are treated as atoms, and before sending them to the corresponding Xlib routine, they are converted to the correct value. This is completely transparent to the user of the package. The integer value may also be used, although this is not encouraged. Boolean values are represented with *True* and *False*.

Function parameters

Parameters that are function pointers in C (for example: the value of the third parameter in the XIfEvent routine) must be passed just with the name of the predicate. The predicate passed as parameter should then have the same arity as the number of arguments the function parameter has in C (plus one for a return value if present).

Structures as parameters

The ***ProLog by BIM*** interface to Xlib is completely based on the C language interface to Xlib. The ***ProLog by BIM*** Xlib interface has a series of special predicates to handle the C structures.

Declaring structures in ProLog by BIM

The structures that are used as parameters for the external routines, must be declared first. These structures must be destroyed when they are no longer needed or before they become inaccessible.

declare/2

declare(_Struct, _Type)

arg1 : free : term
arg2 : ground : atom

Declares the structure *arg1*. Argument *arg1* is instantiated to a structure of type *arg2*. The concrete representation of *arg1* after the instantiation is opaque.

undeclare/1

undeclare(_Struct)

arg1 : ground : term

Arg1 must be a previously declared structure. The structure *arg1* is destroyed and can no longer be used.

Example

```
declare(_Struct,XWindowAttributes),
undeclare(_Struct).
```

These calls instantiate _Struct to a structure of type XWindowAttributes and immediately destroy it.

Assigning values to the fields

assign/3

assign(_Struct, _Field, _Value)

arg1 : ground : term
arg2 : ground : atom
arg3 : ground : atomic

The value of *arg3* is assigned to the field *arg2* of the structure *arg1*. This is an incremental assignment. It does not matter whether there were previously assigned values to this field or not.

assignfields/2

assignfields(_Struct, _NameValueList)

arg1 : ground : term
arg2 : ground : list of name/value

Arg2 is a list of Fieldname/Value pairs. Each value is assigned to the correct field of the structure *arg1*.

Example

```
declare(_Attributes,XWindowAttributes),
...
assign(_Attributes,width,40),
assign(_Attributes,height,45),
assignfields(_Attributes,[x/10,y/40]),
...
undeclare(_Attributes).
```

The first call declares a structure of type XWindowAttributes. The next two calls assign a single value to one field of the structure. In the fourth call, a value is assigned to two fields of the structure. Finally, when the structure is no longer needed, the last call undeclares it.

assign/2

assign(_StructTo, _StructFrom)

arg1 : ground : term
arg2 : ground : term

This predicate is used to assign the contents of one structure to another structure. The contents of *arg2* is assigned to *arg1*. *Arg1* and *arg2* must be of the same type.

Example

```
declare(_Attributes1,XWindowAttributes),
declare(_Attributes2,XWindowAttributes),

assignfields(_Attributes1,[x/10,y/40]),
assign(_Attributes2,_Attributes1),

undeclare(_Attributes1),
undeclare(_Attributes2).
```

Obtaining fields of a structure

getfield/3

getfield(_Struct, _Field, _value)

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

Arg3 is unified with the value of the field specified by arg2, in the structure *arg1*.

getfields/3

getfield(_Struct, _FieldList, _NameValueList)

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

Arg2 is a list of field names of the structure *arg1*. *Arg3* is unified with a list containing name/value pairs (in the same order as they are listed in *arg2*).

Example

```
declare(_Attributes,XWindowAttributes),

assign(_Attributes,width,40),
assign(_Attributes,height,45),
assignfields(_Attributes,[x/10,y/40]),

getfield(_Attributes,y,_Yvalue),
getfields(_Attributes,[width,height,x],_ValueList),

write(_ValueList),
undeclare(_Attributes).
```

This example writes [width/40,height/45,x/10] on the standard output.

Using arrays of structures

There are functions that deal with arrays of structures. It is possible to declare arrays of a known size. The individual components are structures.

declare/3

declare(_ArrayOfStruct, _Type, _Size)

arg1 : free : term
arg2 : ground : term
arg3 : ground : integer

This predicate is used to declare an array of structures. *Arg1* is instantiated with an array of length *arg3*. *Arg2* describes the type of the elements.

Example

```
declare(_WAttrArr,Array(XWindowAttributes),4).
```

declares an array of 4 elements with type XWindowAttributes

When the array is no longer needed, or in any case, before it becomes unreachable, it should be deallocated with **undeclare/1**.

Example

```
declare(_WAttrArr,Array(XWindowAttributes),4),
undeclare(_WAttrArr).
```

Examples

Assign/3 and **assignfields/2** also handle array assignment. The field is specified as a list: its first element is the number of the element in the array; its second element is the field name.

Example

```
declare(_WAttrArr,Array(XWindowAttributes),4),
assign(_WAttrArr,[0,x],10)
undeclare(_WAttrArr).
```

The value 10 is assigned to the field x of the first component in the array.

Example

```
declare(_WAttrArr,Array(XWindowAttributes),4),
assignfields(_WAttrArr,[ [0,x]/10, [2,y]/23 ]),
undeclare(_WAttrArr).
```

The value 10 is assigned to the field x of the first element of the array. The value 23 is assigned to the field y of the third element in the array.

Example

```
declare(_WAttrArr,Array(XWindowAttributes),4),
declare(_WAttr,XWindowAttributes),

assignfields(_WAttr,[x/10,y/20]),
assignfields(_WAttrArr,[3/_WAttr]).
```

A complete structure is assigned to the last element in the array.

Getfield/3 and **getfields/3** work in the same way. Consider the following example:

Example

```
declare(_WAttrArr,Array(XWindowAttributes),4),
getfield(_WAttrArr,0,_WAttr),
assignfields(_WAttr,[x/1,y/2]).
```

In this case _WAttr is just another name for _WAttrArr[0]. _WAttr is not a new variable containing the values of _WAttrArr[0]. Any assignment to _WAttr will be an assignment to _WAttrArr[0].

Note:

- In C, all arrays start with index 0.

Example

```
declare(_WAttrArr,Array(XWindowAttributes),4),
```

declares an array with four components, _WAttrArr[0] to _WAttrArr[3].

- Symbolic constants have to be handled carefully.

Example

the backing_store field of the XWindowAttributes is declared as an integer. The only values that may be assigned to it are defined as NotUseFul, WhenMapped and Always.

Example

```
#define NotUseFul  0
#define WhenMapped 1
#define Always     2
```

Also in ***ProLog by BIM***, these symbolic names can be used:

```
...
assign(_Attributes,backing_store,NotUseFul),
... .
```

The symbolic name will always be the result of the operation.

```
...
getfield(_Attributes,backing_store,_Value),
... .
```

_Value will be instantiated to one of the atoms NotUseFul, WhenMapped and Always. Even when first assigning a pure integer value to the field, and then later on using **getfield/3** to obtain the value, the result will be the symbolic name.

Obtaining the type of a declared structure or array

gettype/2

gettype(_Struct, _Type)

arg1 : ground : term
arg2 : any : atom

Arg2 is unified with the type of the structure *arg1*.

For example:

```
...
declare(_AttrArr,Array(XWindowAttributes)),
gettype(_AttrArr,_Type),
... .
```

_Type will be instantiated to Array(XWindowAttributes).

3.3 Example

A window is created in the example below. Inside the window, the text "Hello World" is placed. The program exits when the left button is pressed inside the window.

```
{*********************************************************
 Xlib, ProLog by BIM Language Interface.
 Demo: Hello World
*********************************************************}
DoEvents(_Dpy,_Event):-
     {An Expose Event --> draw the Text}
     type(_Event,XExposeEvent),


getfields(_Event,[window,display],[window/_W,display/_Dpy]),
```

```
        XDefaultGC(_Gc,_Dpy,0),

        _String = 'Hello World',
        atomlength(_String,_StrLen),
        XDrawString(_Dpy,_W,_Gc,55,55,_String,_StrLen).
        DoEvents(_Dpy,_Event):-
        {Was it a button press event ?}
        type(_Event,XButtonPressedEvent),

        getfields(_Event,[button,display],
             [button/Button1,display/_Dpy]),
        XCloseDisplay(_Dpy),

        Cexit(0).
    main:-
        {Open the display}
        XOpenDisplay(_Dpy,NULL),

        {Call some information functions}
        XDefaultRootWindow(_Root,_Dpy),
        XDefaultScreen(_Scrn,_Dpy),
        XWhitePixel(_White,_Dpy,0),
        XBlackPixel(_Black,_Dpy,0),
        XDefaultGC(_Gc,_Dpy,0),

        {Declare a record and assign some values to it}

        declare(_SetWAttr,XSetWindowAttributes),
        assignfields(_SetWAttr,[background_pixel/_White,
                                border_pixel/_Black,
                                override_redirect/True]),

        {Create the one and only window in this program}
        XCreateWindow(_Mywindow,_Dpy,_Root,
                                10,10,200,100,6,
                                CopyFromParent,
                                InputOutput,
                                CopyFromParent,
                                (CWBackPixel|CWBorderPixel),
                                _SetWAttr),
        XMapRaised(_Dpy,_Mywindow),
        XSelectInput(_Dpy,_Mywindow,
                        (ButtonPressMask|ExposureMask)),

        {Place the font in the Gc}
        declare(_GCValues,XGCValues),
        XLoadFont(_Font,_Dpy,
             '-adobe-new century schoolbook-bold-r-normal
                                --12-120-75-75-p-77-iso8859-1'),
        assignfields(_GCValues,[font/_Font]),
        XChangeGC(_Dpy,_Gc,(GCFont),_GCValues),

        {Get rid of the declared struct}
        undeclare(_SetWAttr),
        undeclare(_GCValues),

        {Declare the event struct for the main loop}
        declare(_Event,XAnyEvent),

        {Select the Events}
        repeat,

    XNextEvent(_Dpy,_Event,_EventOut),
                                DoEvents(_Dpy,_EventOut).
    {*********************************************}
```

3.4 Additional notes

The use of masks

Bitmasks, used in some Xlib routines, can be specified in ***ProLog by BIM*** by the use of symbolic constants. These bitmasks can be combined by or-ing them and placing this term within parentheses.

Example

```
(CWBlackPixel|CWBorderPixel|CWCursor)
```

This construction can be used whenever a bitmask must be given. Also the integer representation is accepted, but the programmer is advised not to use it.

Getfield/3 and **getfields/3** return for bitmasks an integer value.

Example

```
declare(_Hints,XWMHints),
assign(_Hints,[flags/(InputHint|IconPositionHint),

input/True,icon_x/10,icon_y/30]),
getfield(_Hints,flags,_Value),
integer(_Value).
```

Event handling predicates

Events are treated in a specific way in Xlib. A routine that takes an event as input can change this event (for example: changing this type). Due to Prolog's single assignment, it is not possible to maintain it that way.

Therefore, each routine that takes an event of type XAnyEvent as input parameter, has an additional argument. This argument is instantiated with the correct event upon return from the Xlib routine. The input argument and the return argument share some fields (any change made to one of the fields in one argument, will be seen in the other argument).

Example

```
declare(_EventIn,XAnyEvent),
XNextEvent(_Dpy,_EventIn,_EventOut).
```

_EventIn and _EventOut share the fields type, serial, send_event, display and window. The other fields, specific for the type of the event returned, are only available through _EventOut.

Arrays of basic types

In the ***ProLog by BIM*** Xlib Interface, the following array types are defined:

Type	Used In
Array(Atom)	XListProperties, XRotateProperties
Array(Window)	XQueryTree, XRestackWindows
Array(long)	XAllocColorCells, XAllocColorPlanes, XStoreColors,XFreeColors
Array(char)	XSetDashes, XStoreBytes, XStoreBuffer, XFetchBytes, XFetchBuffer, XQueryKeymap, XSetPointerMapping, XChangeProperty
Array(KeySym)	XRebindKeysym, XGetKeyboardMapping
Array(Colormap)	XListInstalledColormaps

Array(string)	XListFonts,XListFontsWithInfo, XSetFontPath,XGetFontPath, XSetStandardProperties, XSetCommand

These types can be used in **declare/3**.

Note:
The type Array (char) is only used in routines where the parameter is not treated as a null terminated string but as an array of bytes. The values assigned to it should not be characters but integers lower than 256. The values obtained from it are integers lower than 256.

Example

```
?- declare(_Buffer,Array(char),100),
   assign(_Buffer,23,65),
   getfield(_Buffer,23,_Value).
_Value = 65
Yes
```

Arrays of null terminated strings must be declared in the normal way. Consider the following example. The XSetFontPath routine uses an array of strings to establish a search path for fonts.

Example

```
XOpenDisplay(_Dpy,NULL),
declare(_FontPathArr,Array(string),3),

assignfields(_FontPathArr,
      [
      0/'/usr/lib/X11/fonts/misc/',
      1/'/usr/lib/X11/fonts/75dpi/',
      2/'/usr/lib/X11/fonts/100dpi/']),
XSetFontPath(_Dpy,_FontPathArr,3),
XFreeFontPath(_FontPathArr),
undeclare(_FontPathArr),

XCloseDisplay(_Dpy).
```

Hierarchy in the variables

It is possible to declare arrays of arrays. It is also possible for arrays and structures to be a field of a structure.

Example

an array of arrays of XWindowAttributes structures:

```
declare(_ArrayArray,Array(Array(XWindowAttributes)),2),
declare(_Array1,Array(XWindowAttributes),20),
declare(_Array2,Array(XWindowAttributes),50),

assignfields(_ArrayArray,[0/_Array1, 1/_Array2]),
assignfields(_ArrayArray,[
                                  [0,1,x]/10,
                                 [1,30,y]/30
                                  ]).
```

In **assign/3**, **assignfield/3**, **getfield/3**, and **getfields/3**, the complete "path" to the value to be accessed must be specified. When assigning fields or components, the values should have the correct type. When obtaining fields and components, the values are automatically converted to the correct type.

3.5 List of defined predicates

XFillArcs/5
XActivateScreenSaver/1
XAddHost/2
XAddHosts/3
XAddPixel/2
XAddToSaveSet/2
XAllPlanes/1
XAllocColor/4
XAllocColorCells/8
XAllocColorPlanes/12
XAllocNamedColor/6
XAllowEvents/3
XAutoRepeatOff/1
XAutoRepeatOn/1
XBell/2
XBitmapBitOrder/2
XBitmapPad/2
XBitmapUnit/2
XBlackPixel/3
XBlackPixelOfScreen/2
XCellsOfScreen/2
XChangeActivePointerGrab/4
XChangeGC/4
XChangeKeyboardControl/3
XChangeKeyboardMapping/5
XChangePointerControl/6
XChangeProperty/8
XChangeSaveSet/3
XChangeWindowAttributes/4
XCheckIfEvent/6
XCheckMaskEvent/5
XCheckTypedEvent/5
XCheckTypedWindowEvent/6
XCheckWindowEvent/6
XCirculateSubwindows/3
XCirculateSubwindowsDown/2
XCirculateSubwindowsUp/2
XClearArea/7
XClearWindow/2
XClipBox/2
XCloseDisplay/1

XDisplayHeightMM/3
XDisplayName/2
XDisplayOfScreen/2
XDisplayPlanes/3
XDisplayString/2
XDisplayWidth/3
XDisplayWidthMM/3
XDoesBackingStore/2
XDoesSaveUnders/2
XDrawArc/9
XDrawArcs/5

XDrawImageString/7
XDrawImageString16/7
XDrawLine/7
XDrawLines/6
XDrawPoint/5
XDrawPoints/6
XDrawRectangle/7
XDrawRectangles/5
XDrawSegments/5
XDrawString/7
XDrawString16/7
XDrawText/7
XDrawText16/7
XEmptyRegion/2
XEnableAccessControl/1
XEqualRegion/3
XEventMaskOfScreen/2
XEventsQueued/3
XFetchBuffer/4
XFetchBytes/3
XFetchName/4
XFillArc/9
XFillPolygon/7
XFillRectangle/7
XFillRectangles/5
XFindContext/5
XFlush/1
XForceScreenSaver/2
XFree/1
XFreeColormap/2
XFreeColors/5
XFreeCursor/2
XFreeFont/2
XFreeFontInfo/3

XImageByteOrder/2
XInsertModifiermapEntry/4
XInstallColormap/2
XInternAtom/4
XIntersectRegion/3
XKeycodeToKeysym/4
XKeysymToKeycode/3
XKeysymToString/2
XKillClient/2
XLastKnownRequestProcessed/2
XListFonts/5
XListFontsWithInfo/6
XListHosts/4
XListInstalledColormaps/4
XListProperties/4
XLoadFont/3
XLoadQueryFont/3
XLookupColor/6
XLookupKeysym/3
XLookupString/6
XLowerWindow/2

XMapRaised/2
XMapSubwindows/2
XMapWindow/2
XMaskEvent/4
XMatchVisualInfo/6
XMaxCmapsOfScreen/2
XMinCmapsOfScreen/2
XMoveResizeWindow/6
XMoveWindow/4
XNewModifiermap/2
XNextEvent/3
XNextRequest/2
XNoOp/1
XOffsetRegion/3
XOpenDisplay/2
XParseColor/5
XParseGeometry/6
XPeekEvent/3
XPeekIfEvent/5
XPending/2
XPlanesOfScreen/2
XPointInRegion/4
XPolygonRegion/4
XProtocolRevision/2

XSetClipMask/3
XSetClipOrigin/4
XSetClipRectangles/7
XSetCloseDownMode/2
XSetCommand/4
XSetDashes/5
XSetErrorHandler/1
XSetFillRule/3
XSetFillStyle/3
XSetFont/3
XSetFontPath/3
XSetForeground/3
XSetFunction/3
XSetGraphicsExposures/3
XSetIOErrorHandler/1
XSetIconName/3
XSetIconSizes/4
XSetInputFocus/4
XSetLineAttributes/6
XSetModifierMapping/3
XSetNormalHints/3
XSetPlaneMask/3
XSetPointerMapping/4
XSetRegion/3
XSetScreenSaver/5
XSetSelectionOwner/4
XSetSizeHints/4
XSetStandardColormap/4
XSetStandardProperties/8
XSetState/6
XSetStipple/3

XSetSubwindowMode/3
XSetTSOrigin/4
XSetTile/3
XSetTransientForHint/3
XSetWMHints/3
XSetWindowBackground/3
XSetWindowBackgroundPixmap/3
XSetWindowBorder/3
XSetWindowBorderPixmap/3
XSetWindowBorderWidth/3
XSetWindowColormap/3
XSetZoomHints/3
XShrinkRegion/3
XStoreBuffer/4

XrmPutLineResource/2
XrmPutResource/4
XrmPutStringResource/3
XrmQGetResource/6
XrmQPutResource/5
XrmQPutStringResource/4
XrmQuarkToString/2
XrmStringToBindingQuarkList/3
XrmStringToQuark/2
XrmStringToQuarkList/2
XrmUniqueQuark/1

Windowing and Graphics Libraries

Chapter 4
Interface to OPEN LOOK / XVIEW

4.1 Preface

The XView routines are available from ***ProLog by BIM*** as external predicates. This chapter treats the differences between the C language binding and the ***ProLog by BIM*** binding for these routines. It does not describe the XView routines. More information about that, can be found in the *"XView Reference Manual: Summary of the XView Api"* by Sun Microsystems,Inc. and *"XView Programming Manual"* by O'Reilly, Ass.

4.2 Availability of XView

The XView predicates are defined in the ***ProLog by BIM*** library program : **-Lxview**.

The next command calls ***ProLog by BIM*** with the XView package and the user's program *my_xview_application.pro*.

```
% BIMprolog -Lwindowing/xview my_xview_application
```

The package can also be consulted interactively:

```
?- consult( '-Lwindowing/xview' ) .
```

When this ***ProLog by BIM*** package is used frequently, it is better to link it externally with **BIMlinker**. The standard installation procedure provides in an externally linked system that includes the Xlib and XView interfaces. This program is called **BIMprologXV**. Using it in the above example, gives the following command:

```
% BIMprologXV my_xview_application
```

Note:

Due to the limitations of the SunOs 4.0 dynamic linking package it is not possible to incrementally load the packages as explained above. Please use an extended executable as explained below.

If an extended executable containing the Xview interface has been built during the installation procedure it will also contain the interface to xview and can be started with:

```
BIMprologXV
```

4.3 XView predicates

Correspondence between C routines and predicates

The predicates in ***ProLog by BIM*** have the same name as the corresponding XView C routines. Moreover, some XView macros are available in ***ProLog by BIM*** as predicates.

The order and the type of the arguments of a predicate are the same as for the corresponding C routine.

Routines that have a return value in C have an additional first argument for the predicate in ***ProLog by BIM***. That first argument is instantiated to the return value.

Symbolic constants

The symbolic names (like *FRAME, CANVAS_MARGIN* etc.) can be used in ***ProLog by BIM*** as well. In ***ProLog by BIM*** they are treated as atoms and before sending them to the corresponding XView routine, they are converted to the correct value. This is completely transparent to the user of the package.

Boolean values are represented with *TRUE* and *FALSE*.

Parameter lists

Whenever an XView routine in C has a null-terminated sequence of parameters, the corresponding ***ProLog by BIM*** predicate has a single parameter, which must be a list ending with the integer zero.

Example

the C-call

```
base_frame = xv_create( NULL, FRAME,
     FRAME_ LABEL, "The Towers of Hanoi" ,
     XV_X, 15, XV_Y, 15,
     0 );
```

becomes in Prolog

```
xv_create( _base_frame, NULL, FRAME, [
     FRAME_LABEL, 'The Towers of Hanoi' ,
     XV_X, 15, XV_Y, 15,
     0] ),
```

Function parameters

Parameters that are function pointers in C (e.g. the value of the *WIN_EVENT_PROC* attributes and the like), can be given in two ways. When the purpose is to attach a Prolog predicate to that parameter, the predicate can be given directly in the form *name/arity.*

The next call sets the *WIN_EVENT_PROC* of a canvas to the predicate event_handler/2.

```
canvas_set( _canvas, [ WIN_EVENT_PROC, event_handler/2, 0] )
```

Note that the arity of the predicate is to be determined from the specification in the XView description. The Prolog programmer is still free to use a lower arity. In this case the last arguments are not passed to the predicate. So the following call is also legal:

```
canvas_set( _canvas, [ WIN_EVENT_PROC,
simple_event_handler/0, 0] )
```

Another possibility is to attach a C-defined routine instead of a ProLog predicate. In that case a pointer to the routine (retrieved from some external predicate, or by using **extern_name_address/3**) may be given.

A useful application of this is to turn off any event procedure using:

```
canvas_set( _canvas, [ WIN_EVENT_PROC, 0x0, 0] )
```

The XView interface detects through the type of the argument which of both methods (a pointer or a predicate description) is used.

Finally a note on a special case of call back predicates: the *notifier predicates*. These predicates are expected to return an exit code to the notifier (as *NOTIFY_IGNORED* or *NOTIFY_DONE*). This is accomplished from ***ProLog by BIM*** by instantiating the first argument of the predicate to the exit code. This means that such predicates have one argument more than the corresponding C routines would have.

Example

Here an input handler for a pipe is set.

```
notify_set_input_func( _client, pipe_reader/3, _pipe_in_fd )
 ...
pipe_reader( NOTIFY_DONE , _client , _fd ) :- ...
```

As can be seen, the pipe_reader/3 predicate has three arguments rather than two as described in the XView specification.

Function call parameters

Normally, ***ProLog by BIM*** does not evaluate arguments; they are always passed as structures. As a consequence, parameters of XView predicates cannot be function calls.

Example

This illustrates the above. In C one can write

```
panel_set( _item , PANEL_ITEM_X , xv_col(_panel,1) , 0 );
```

Translating this to Prolog gives

```
xv_col( _col , _panel , 1 ),
panel_set( _item , [ PANEL_ITEM_X , _col , 0 ] )
```

The call of xv_col() is performed out of the attribute list and the resulting column value is used as attribute value.

Variable typed return values

Some routines give a return value whose type depends on their input parameters (e.g. xv_get() and others). In C, the programmer has to use type casts to determine the type of the returned value.

The ***ProLog by BIM*** XView interface detects the type by inspecting the attribute and gives a result of the appropriate type.

Example

The following calls in C

```
x = (int) xv_get( win , XV_X );
label = (char *) xv_get( base_frame , FRAME_LABEL );
```

are equivalent to the Prolog calls

```
xv_get( _x , _win , XV_X )
xv_get( _label , _base_frame , FRAME_LABEL )
```

In the first goal, *_x* will have type integer and in the second, *_label* will be an atom.

Routines that return a value whose semantics is determined by the user, always return a pointer. If it should be interpreted as an integer, then the **pointertoint/2** built-in can be applied to it. If it represents an atom, it can be converted with **stringtoatom/2**.

Complex retrieval

The retrieval of the contents of a text subwindow is a more complex matter than for other attributes. That contents is stored in a buffer that must be provided by the program that wants to retrieve it. A buffer can be created in ***ProLog by BIM*** using one of the additional interface predicates (which are described in the next section).

Example

```
create_char_array( _Buffer , ?(_BufSize+1) ),
xv_get( _NextPos , _Textsw ,
      TEXTSW_CONTENTS , _StartPos , _Buffer , _BufSize ),
stringtoatom( _Buffer , _Text ),
xv_free( _Buffer )
```

In this example, a buffer is created of *_BufSize+1* characters (one character extra for a terminating 0 character). The contents of text subwindow *_Textsw*, is retrieved, starting at position _StartPos and for the size of the buffer. Next, the buffer is converted to an atom *_Text*, and finally, the buffer is deallocated.

If multiple retrievals of text subwindow contents are planned, it might be better to allocate the buffer only once and not deallocate it. Using the same buffer for each retrieval will save a lot of memory management overhead.

4.4 Additional interface predicates

Attribute list checking

Attribute/value lists are converted in ***ProLog by BIM*** from symbolic form to numerical form before they are passed to XView. This transformation checks the types of attribute values, but does not give any messages when errors are detected. Instead it fails. For debugging purposes, two predicates are provided to check an attribute/value list.

avlist_check/1

avlist_check(_AttributeValueList)

arg1 : ground : list

The attribute/value list *arg1* is checked for type consistency. If an error is detected, a diagnostic message is given. It indicates the part of the list that was right, the attribute whose argument was wrong and the expected type for that argument.

avlist_whatis/1

avlist_whatis(_AttributeArgumentType)

arg1 : ground : atom

An explanation is given of what is meant with *arg1*. This is an attribute argument type as indicated by avlist_check/1.

Character array

A character array is a contiguous array of single byte integers. Its elements are represented as small integers with values from 0 to 255. They are indexed on a zero-base. Character arrays are used in the colormaps, or to retrieve the contents of a text subwindow, but they can also be used for other purposes.

create_char_array/2

create_char_array(_ArrayPointer, _ArraySize)

arg1 : free : pointer
arg2 : ground : integer

A character array of *arg2* elements is created. *Arg1* is instantiated with a pointer to the array.

create_char_array/3

create_char_array(_ArrayPointer, _ArraySize, _ArrayValue)

arg1 : free : pointer
arg2 : ground : integer
arg3 : ground : list of integer

A character array of *arg2* elements is created. It is initialized with the elements of *arg3*. *Arg1* is instantiated with a pointer to the array. The integers in *arg3* are truncated to small integers with a modulo 256 operation.

set_char_array/3

set_char_array(_ArrayPointer, _Index, _Value)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer

The element at index *arg2* of character array *arg1*, is changed to *arg3*.

set_char_array/4

set_char_array(_ArrayPointer, _IndexFrom, _IndexTo, _Values)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : list of integers

The range of elements from index *arg2* to index *arg3* of character array *arg1*, are changed to the elements of *arg4*.

get_char_array/3

get_char_array(_ArrayPointer, _Index, _Value)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : free : integer

Arg3 is instantiated to the element at index *arg2* of character array *arg1*.

get_char_array/4

get_char_array(_ArrayPointer, _IndexFrom, _IndexTo, _Values)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : free : list of integer

Arg4 is instantiated to the range of elements from index *arg2* to index *arg3* of character array *arg1*.

Short array

A short array is a contiguous array of short integers. They are indexed on a zero-base. Short arrays are can be used for bitmap descriptions of server images, but also for any other purpose.

create_short_array/2

create_short_array(_ArrayPointer, _ArraySize)

arg1 : free : pointer
arg2 : ground : integer

A short array of *arg2* elements is created. *Arg1* is instantiated with a pointer to the array.

create_short_array/3

create_short_array(_ArrayPointer, _ArraySize, _ArrayValue)

arg1 : free : pointer
arg2 : ground : integer
arg3 : ground : list of integer

A short array of *arg2* elements is created. It is initialized with the elements of *arg3*. *Arg1* is instantiated with a pointer to the array. The integers in *arg3* are truncated to short integers with a modulo 2^16 operation.

set_short_array/3

set_short_array(_ArrayPointer, _Index, _Value)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer

The element at index *arg2* of short array *arg1*, is changed to *arg3*.

set_short_array/4

set_short_array(_ArrayPointer, _IndexFrom, _IndexTo, _Values)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : list of integers

The range of elements from index *arg2* to index *arg3* of short array *arg1*, are changed to the elements of *arg4*.

get_short_array/3

get_short_array(_ArrayPointer, _Index, _Value)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : free : integer

Arg3 is instantiated to the element at index *arg2* of short array *arg1*.

get_short_array/4

get_short_array(_ArrayPointer, _IndexFrom, _IndexTo, _Values)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : free : list of integer

Arg4 is instantiated to the range of elements from index *arg2* to index *arg3* of short array *arg1*.

String array

A string array is a contiguous array of strings. It is indexed on a zero-base. The text of the strings (which are character arrays) is copied into dedicated memory.

create_string_array/2

create_string_array(_ArrayPointer, _ArraySize)

arg1 : free : pointer
arg2 : ground : integer

A string array of *arg2* elements is created. *Arg1* is instantiated with a pointer to the array.

create_string_array/3

create_string_array(_ArrayPointer, _ArraySize, _ArrayValue)

arg1 : free : pointer
arg2 : ground : integer
arg3 : ground : list of atom

A string array of *arg2* elements is created. It is initialized with the elements of *arg3*. *Arg1* is instantiated with a pointer to the array.

set_string_array/3

set_string_array(_ArrayPointer, _Index, _Value)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : atom

The element at index *arg2* of string array *arg1*, is changed to *arg3*.

set_string_array/4

set_string_array(_ArrayPointer, _IndexFrom, _IndexTo, _Values)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : list of atom

The range of elements from index *arg2* to index *arg3* of string array *arg1*, are changed to the elements of *arg4*.

get_string_array/3

get_string_array(_ArrayPointer, _Index, _Value)

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

Arg3 is instantiated to the element at index *arg2* of string array *arg1*.

get_string_array/4

get_string_array(_ArrayPointer, _IndexFrom, _IndexTo, _Values)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : free : list of atom

Arg4 is instantiated to the range of elements from index *arg2* to index *arg3* of string array *arg1*.

XView and X rectangle structures

A rectangle list is passed to an X canvas repaint predicate, from the notifier. It contains an array of X rectangles. These can be retrieved to use them in Xlib calls, are to investigate their fields. The normal sequence of using the predicates below, is first **get_xrectlist/3** with the rectangle list that was passed, to get the rectangle array. Then one element from this rectangle array is retrieved with **get_xrect_array/3**, and finally, this X rectangle can be decomposed with **get_xrectangle/5**.

get_xrectlist/3

get_xrectlist(_RectList, _Count, _RectArray)

arg1 : ground : pointer
arg2 : free : integer
arg3 : free : pointer

The Xv_xrectlist structure, pointed to by *arg1* is decomposed. Its count field is unified with *arg2*, and its rectangle array with *arg3*.

get_xrect_array/3

get_xrect_array(_RectArray, _ArrayIndex, _Rectangle)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : free : pointer

The *arg2*'nd element (zero-based) of the rectangle array *arg1* is unified with *arg3*.

No range checking is performed on the index.

get_xrectangle/5

get_xrectangle(_Rectangle, _X, _Y, _Width, _Height)

arg1 : ground : pointer
arg2 : free : integer
arg3 : free : integer
arg4 : free : integer
arg5 : free : integer

The XRectangle structure *arg1* is decomposed. The parameters *arg2, arg3, arg4* and *arg5* are instantiated to its x, y, width and height components respectively.

XView single color structure

create_singlecolor/4

create_singlecolor(_SingleColor, _Red, _Green, _Blue)

arg1 : free : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : integer

An Xv_singlecolor structure is created and initialized with *arg2*, *arg3* and *arg4* as red, green and blue components respectively. *Arg1* is instantiated to a pointer to the structure.

set_singlecolor/4

set_singlecolor(_SingleColor, _Red, _Green, _Blue)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : integer

The value of the Xv_singlecolor structure *arg1* is changed to *arg2*, *arg3* and *arg4* as red, green and blue components.

get_singlecolor/4

get_singlecolor(_SingleColor, _Red, _Green, _Blue)

arg1 : ground : pointer
arg2 : free : integer
arg3 : free : integer
arg4 : free : integer

The value of the Xv_singlecolor structure *arg1* is retrieved. *Arg2*, *arg3* and *arg4* are instantiated to its red, green and blue components.

Timeval structure

create_timeval/3

create_timeval(_TimeVal, _Seconds, _uSeconds)

arg1 : free : pointer
arg2 : ground : integer
arg3 : ground : integer

A timeval structure is created and initialized to *arg2* seconds and *arg3* microseconds. *Arg1* is instantiated to a pointer to the structure.

set_timeval/3

set_timeval(_TimeVal, _Seconds, _uSeconds)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : integer

The value of the timeval structure *arg1* is changed to *arg2* seconds and *arg3* microseconds.

get_timeval/3

get_timeval(_TimeVal, _Seconds, _uSeconds)

arg1 : ground : pointer
arg2 : free : integer
arg3 : free : integer

The value of the timeval structure *arg1* is retrieved. *Arg2* is instantiated to the seconds and *arg3* to the microseconds.

Itimerval structure

create_itimerval/3

create_itimerval(_ITimerVal, _Interval, _Value)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : pointer

An itimerval structure is created and initialized with *arg2* as the interval and *arg3* as the value of the timer. *Arg1* is instantiated to a pointer to the structure.

The two arguments, *arg2* and *arg3* must be pointers to timeval structures.

set_itimerval/3

set_itimerval(_ITimerVal, _Interval, _Value)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : pointer

The value of the itimerval structure *arg1* is changed to *arg2* as the interval and *arg3* as the value of the timer.

The two arguments, *arg2* and *arg3* must be pointers to timeval structures.

get_itimerval/3

get_itimerval(_ITimerVal, _Interval, _Value)

arg1 : ground : pointer
arg2 : ground : pointer
arg3 : ground : pointer

The value of the itimerval structure *arg1* is retrieved. *Arg2* is instantiated to its interval and *arg3* to its value.

The two arguments, *arg2* and *arg3* must be pointers to timeval structures. The structures they are pointing to, will be changed.

Structure deallocation

Structures that are not longer needed, can be deallocated to give the used memory free again.

xv_free/1

xv_free(_Structure)

arg1 : ground ; pointer

The structure, pointed to by *arg1* is deallocated.

4.5 Additional interface tools

BIMimage is a tool to convert an image description between XView or X format and ***ProLog by BIM*** representation.

Synopsis

```
% BIMimage [ -r ] [ -icon ] [ -xbm ][ -n name ] [ infile [
  outfile ]]
```

Description

BIMimage converts image descriptions between the standard XView or X image formats and ***ProLog by BIM*** readable predicate format. The XView image format uses 16 bit data in the bitmap, while the X image format uses 8 bit data for the bitmap representation.

Without options, the input file is considered an XView or X image file. The exact format is detected when reading the file. The output file will contain a description of the same image as a predicate. The name of the predicate is derived from the input file name. The path is removed and all '.' are replaced by '_'.

The -n option sets the predicate name to *name* instead of the default name that is derived from the input file name.

With -r, the reverse conversion is done. The input file should contain a ***ProLog by BIM*** predicate describing an image. A standard XView or X image file is generated as output. The exact format, XView or X, is derived from the input image description.

The option -icon forces output in XView icon format

The option -xbm forces output in X bitmap format

When *outfile* or *infile* are omitted, they are replaced by *stdout* and *stdin*.

Format

The ***ProLog by BIM*** format for an image description is an arity 5 predicate with following specification:

_ImagePredicate/5

_ImagePredicate(_Type, _Width, _Height, _Depth, _Bitmap)

arg1 : ground : atom
arg2 : ground : integer
arg3 : ground : integer
arg4 : ground : integer
arg5 : ground : list of integer

The type specifier *arg1* is either *X* for standard X format (8 bit data), or *XView* for standard XView image format (16 bit data).

Arg2, arg3 and *arg4* are the image width, height and depth in pixels.

The bitmap is given in *arg5* as a list of 8 bit or 16 bit integers, depending on *arg1*.

Example

If an XView image file (which could have been created with **iconedit**) is named *demo.icon* the following call converts it to a predicate:

```
% BIMimage demo.icon demo.pro
```

This creates a ***ProLog by BIM*** file *demo.pro* containing the predicate demo_icon/5.

Server images

From an image description predicate, a *Server Image* object must be created to use the image. The following predicate is a general purpose predicate, which given the name of an image description predicate, creates a server image.

create_server_image/2

create_server_image(_ServerImage, _ImageDescription)

arg1 : free : pointer
arg2 : ground : atom

A server image is created as described by the predicate with name *arg2*. The resulting object handle is returned in *arg1*.

```
create_server_image( _ServerImage , _ImageDescription ) :-
     _Goal =.. [ _ImageDescription,_Type,_Width,
                 _Height,_Depth,_Bitmap ],
     call( _Goal ),
     create_server_image_bitmap_type( _Type , _Bits ),
     xv_create( _ServerImage , XV_NULL , SERVER_IMAGE , [
          XV_WIDTH , _Width ,
          XV_HEIGHT , _Height ,
          SERVER_IMAGE_DEPTH , _Depth ,
          _Bits , _Bitmap ,
          0 ] ) .

create_server_image_bitmap_type( XView , SERVER_IMAGE_BITS )
.
create_server_image_bitmap_type( X , SERVER_IMAGE_X_BITS ) .
```

Database Interfaces

Database Interfaces

Chapter 1
Introduction to the Database Interfaces

1.1 Rationale behind database interfaces

The integration between Prolog systems and relational databases has been a research and development issue for many years. One of the crucial limitations of early Prolog systems was the lack of the ability to build and manipulate persistent data structures. This proved to be the major obstacle for the development of real-life applications in Prolog.

On the other hand, relational databases did not offer a full-fledged language for programming advanced applications. Query languages like SQL are easy means to accessing data for non-expert users, but lack important features for effective application programming.

The only viable approach seemed to be a pragmatic integration of Prolog and relational databases. Thanks to the availability of professional and mature systems, both Prolog and relational databases, effective integration became possible.

The ***ProLog by BIM*** interfaces to the relational database system offer the combined power of both paradigms for the benefit of the advanced information system builder.

1.2 Overview of the database interfaces

But also in a tight coupling of ***ProLog*** and relational databases a number of options remain open. Different levels of access to the database systems from ***ProLog*** are possible.

Transparent access

A first possibility gives the Prolog programmer a completely transparent access to the relational database. A second approach gives the Prolog programmer direct access to the procedural language interface procedures of the relational database. Finally, a third approach implements a truly embedded SQL in Prolog.

The transparent access mode gives the ***ProLog*** programmer an extremely easy means for interaction with the data stored in the databases. The only database-specific command is the ***ProLog*** predicate to open a particular database. From then on all the relations in the relational database are known to the ***ProLog*** system and accessing and manipulating them is achieved in exactly the same way as if they were in-core predicates of ***ProLog***.

This is an extremely useful feature in an evolutionary prototyping approach. One can start off with a prototype implementation of the application using the in-core database features of Prolog, while testing can be done with operational data in the relational database. Delivery in the final user environment is trivial.

Access through the external interface

Another way of giving access to the database from ***ProLog*** is through its external language interface. As described elsewhere, it allows the ***ProLog*** programmer to invoke any available external function from within a Prolog program. The external functions are linked (externally or incrementally) to the ***ProLog*** executable, while binding the external function definitions to the Prolog calls. As such user-built external functions can be added to extend the set of built-in functions of the ***ProLog*** system, but also complete libraries (such as the database host language interface functions) can be made available. Using these procedures, the ***ProLog*** programmer can access virtually all of the functionality of the database system, unlocking its complete power.

Embedded SQL

A last and final mode of operation is the embedded SQL in Prolog. This mode represents the ultimate merge of the logic programming paradigm and SQL. The SQL strings are forwarded to the database for processing, results can be printed on the screen or can be collected in ***ProLog*** variables for further processing. This represents the most advanced and most flexible scheme available to the ***ProLog*** application builder.

Usage of the database interfaces

Not all interfacing modes are necessarily available in all of the interfaces of the different databases supported by ***ProLog***. But when they are available, the different modes can be used alternatively in a same program. A last note concerns the fact that ***ProLog*** incorporates appropriate garbage collectors to allow for data-intensive or "infinite" computations to be run effectively.

Database Interfaces

Chapter 2
Generic Database Interface

2.1 Introduction to the generic database interfaces

ProLog provides an interface to external relational databases, like Sybase and Oracle, allowing manipulation of their contents in the most flexible way. Note however that a good knowledge about the use of the database at hand is a prerequisite.

The interfaces are implemented with the external language interface. The ***ProLog*** user can choose which external database to access when starting ***ProLog***. An effort has been made to unify the syntax for the different database interface predicates. In this release, it is not possible to access two database systems simultaneously.

The predicates, defined for all interfaces, can be divided into three main classes: *system predicates*, *relation predicates* and *schema predicates*. *The system predicates* open and close the database. The *relation predicates* are predicates which retrieve, insert, modify or delete records. The *schema predicates* give the user more information about the schema of the current opened database.

2.2 System predicates

The system predicates open and close the database. These predicates are database dependent, for details on the respective database systems see the following chapters.

open*_relational_database_systemname_*db

Before any operation can be done on the database or its relations, the database has to be opened. This predicate can have arguments, like the name of the database, or other parameters needed when opening the database.

In the interfaces that ***ProLog*** provides, only one database can be open at a time.

*relational_database_systemname_*dbstatus/1

arg1 : any : integer

The argument is 0 if a database is open, otherwise –1.

close*_relational_database_systemname_*db/0

Closes the database and deletes all internal datastructures needed by the interface.

2.3 Relation predicates

The *relation predicates* are predicates which retrieve, insert, modify or delete records.

The relations of the external database are represented in ***ProLog*** as Prolog terms with the same arity as the database relations. Some databases provide a *combination* field type (i.e. the field has one or more subfields). These fields are represented as lists in ***ProLog***.

In the examples, we shall work with the relation 'persons', with fields 'code', 'name' and 'town'. This relation is represented in ***ProLog*** as

```
persons( _code, _name, _town)
```

retrieve/1

arg1 : partial : term (a database goal)

Accesses the database to solve the goal. Only tuples that unify with the goal are retrieved from the database. This predicate backtracks on different solutions, or fails when there is no solution.

Example

```
?- retrieve( persons(_code, _name, _town) ),
   write([ _code, _name, _town]), nl.
[ 532, Bart, Heverlee]
[ 276, Renilde, Antwerpen]
[ 661, Alice, Brussels]
[ 139, Martina, Heverlee]
[ 735, Dani, Brussels]
Yes
?- retrieve( persons( _code, _name, Brussels) ),
   write([ _code, _name]), nl.
[ 661, Alice]
[ 735, Dani]
Yes
```

Due to differences in internal representations of reals, it is not recommended to retrieve records on an exact value of a real, but rather on a range of reals (see **retrieve/2**).

retrieve/2

arg1: partial : term (a database goal)
arg2 : partial : flat list of <cond> on the variables of arg1

with <cond> ==> <variable> <operator> <variable> | <variable> <operator> <constant>

and <operator> ==> < | > | =< | >= | <> | =

Accesses the database to solve a goal, under additional conditions specified in the list *arg2.*

Example

```
?- retrieve(  persons(_code, _name, _town),
                      [ _code > 500, _code < 700]
           ),
   write([ _code, _name, _town]), nl.
[ 532, Bart, Heverlee]
[ 661, Alice, Brussels]
Yes
```

printr/1

arg1 : ground : atom (a database relation name)

Prints all the records of the database relation.

Example

```
?- printr( persons ).
persons( 532, Bart, Heverlee)
persons( 276, Renilde, Antwerpen)
persons( 661, Alice, Brussels)
persons( 139, Martina, Heverlee)
persons( 735, Dani, Brussels)
Yes
```

insert/1

arg1 : ground : term (a database tuple)

Adds facts to the external database.

Example

```
?- printr( persons ).
persons( 532, Bart, Heverlee)
persons( 276, Renilde, Antwerpen)
Yes
?- insert( persons( 661, Alice, Brussels) ).
Yes
```

```
?- printr( persons ).
persons( 532, Bart, Heverlee)
persons( 276, Renilde, Antwerpen)
persons( 661, Alice, Brussels)
Yes
```

delete/1

arg1 : partial : term (a database goal)

Retracts facts, that unify with the goal, from the external database.
This predicate backtracks on different solutions, or fails when there is no solution.

deleteall/1

arg1 : partial : term (a database goal)

Retracts all the facts that unify with the goal from the external database.
This predicate always succeeds.

2.4 Schema predicates

The *schema predicates* give the user more information about the schema of the current opened database.

schema/0

Lists all the database functors of the currently open database on the standard output.

Example

```
?- schema.
towns( _6, _7)
persons( _6, _7, _8)
Yes
```

schema/1

arg1 : ground : atom (a database relation name)

Displays the schema of the relation on current output stream. The lay-out of this schema depends on the database system.

2.5 Transparent access

Once a database is open with open<...>db, the *ProLog* system distinguishes relations defined in the in-core database from the ones defined in the currently open external database.

For external relations the following automatic conversion is performed:

- _goal ==> retrieve (_goal)
- assert (_goal) ==> insert (_goal)
- retract (_goal) ==> delete (_goal)
- retractall (_goal) ==> deleteall (_goal)

This mechanism is fully transparent to the user and is particularly useful in the migration from a prototype application using in-core relations to an operational system retrieving data from a commercial relational database.

Database Interfaces

Chapter 3
Interface to ORACLE

3.1 Introduction to the ORACLE interface

Different levels of access to the ***ORACLE*** relational database system from ***ProLog by BIM*** are possible.

A first possibility provides the ***ProLog by BIM*** programmer with a set of Prolog predicates not only allowing explicit access to and modification of the ***ORACLE*** database contents, but also the ability to query the schema of the tables and their corresponding columns. Database tables can be easily mapped to user-defined Prolog relations.

A second approach gives the ***ProLog by BIM*** programmer a completely transparent access to the relational database. This transparent access mode gives the programmer an extremely easy means to interact with the data stored in the ***ORACLE*** databases. The only database-specific commands are the ***ProLog by BIM*** predicates to open a particular database and to name the database relations that are going to be accessed. From then on, all accessible relations in the ***ORACLE*** relational database are known to the ***ProLog by BIM*** system. The access and manipulation of these relations is achieved in exactly the same way as if they were in-core predicates of ***ProLog by BIM***.

This is an extremely useful feature in an evolutionary prototyping approach. One can start off with a prototype implementation of the application using the in-core database features of Prolog, while testing can be done with operational data in the ***ORACLE*** relational database.

The final mode of operation is the embedded SQL in Prolog. The SQL strings are sent to the ***ORACLE*** database for processing, and results can be displayed on the screen or collected in ***ProLog by BIM*** variables for further processing.

3.2 Preliminary notes

The ***ProLog by BIM*** interface to the ***ORACLE*** relational database system allows manipulation of its contents in a very flexible way. However, a good knowledge about the use of the ***ORACLE*** database system is a prerequisite.

Please refer to the installation guide to check the supported ***ORACLE*** version.

The ***ORACLE*** database system must be completely installed and running.

The UNIX environment of the current user must contain the variables giving access to ***ORACLE*** (ORACLE_SID and ORACLE_HOME) and the current user must be authorized to access the ***ORACLE*** database server. Please refer to your database administrator for further information.

3.3 Calling the ORACLE interface

If a linked version of ***ProLog by BIM*** and its interface to ***ORACLE*** has been created during the installation, the interface can be called with:

```
% BIMprolog.Oracle
```

The ***ORACLE*** database can be opened and manipulated as described in the following paragraphs (see "*System predicates*").

On systems running SunOS 5, ***ProLog by BIM*** and its ***ORACLE*** interface can be called with the UNIX command:

```
% BIMprolog -Ldatabases/Oracle
```

3.4 Datatypes

ProLog by BIM converts the following ***ORACLE*** datatypes into ATOMS (0..MAX_ATOM).

CHAR(n)	1..255 characters
VARCHAR(n)	1..255 characters
DATE	9 characters (DD-MON-YY)
LONG	1..65535 characters

ProLog by BIM converts the following datatype into INTEGER or REAL according to the effective value returned by ORACLE.

NUMBER	1..38 digits from 1.0 X 10-29 to 9.99x10124 NUMBER[[+nb of digits(1..38)][+scale-84..127]

Notes:

- ***ProLog by BIM*** INTEGERS range from -2^{28}upto2^{28}-1.
- ***ProLog by BIM*** REALS are supported in double precision.

3.5 Overview of the ORACLE interface

The database interface predicates can be divided into four main classes:

system predicates
schema predicates
relation predicates
SQL predicates

The *system predicates* open and close the database.

The *schema predicates* give the user more information about the schema of the current opened database and the accessible tables.

The *relation predicates* are predicates which retrieve, insert, modify or delete records.

The *SQL predicates* allows the use of the database query language from within ***ProLog by BIM***.

3.6 System predicates

openORACLEdb/0

openORACLEdb

Opens the ***ORACLE*** database with default values for USER and PASSWORD: the values of the UNIX environment variables ORACLE_USER and ORACLE_PWD.

openORACLEdb/1

openORACLEdb(_OptionList)

arg1 : ground : list of atoms

The argument is a list of classes of options immediately followed by their value. The different classes are:

'USER' or 'U'
Specifies the user name of the ***ORACLE*** login procedure

'PASSWORD' or 'P'
Specifies the password of the user in the ***ORACLE*** login procedure

'SHOW_ERRORS' or 'SE' **(on/off)**
Determines whether the ***ORACLE*** error messages have to be displayed or not.

closeORACLEdb/0

closeORACLEdb

The ***ORACLE*** database is no longer accessible from ***ProLog by BIM***.

statusORACLEdb/1

statusORACLEdb(_Status)

arg1 : any : integer

Arg1 is 0 when the database is open, -1 otherwise.

3.7 Schema predicates

Once the database is open, the database relations present in the ***ORACLE*** system must be connected to Prolog relations before using them transparently. The following predicates help you list what database tables can be accessed from within ***ProLog by BIM***, how to connect them with Prolog names, and then, list the relations already connected.

accessible/0

accessible

Lists all the database relations accessible to the current user on the current output stream.

accessible/1

accessible(_Owner)

arg1 : ground : atom

Lists all the database relations owned by *arg1* and accessible to the current user on the current output stream.

accessible/4

accessible(_Owner, _TableName, _Arity, _Type)

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

Collects, by backtracking, Owner (*arg1*), TableName (*arg2*), Arity (*arg3*) and Type (*arg4*) of each table accessible to the current user.

def_dbalias/2

def_dbalias(_PrologName, _OracleName)

arg1 : any : atom- Prolog name
*arg2 : any : atom- **ORACLE** name prefixed by owner*
*- **ORACLE** name not prefixed by owner if owner=user*

Establishes a link between an accessible ***ORACLE*** relation (*arg2*) and a Prolog name (*arg1*) for easy transparent manipulation.

dbalias/2

dbalias(_PrologName, _OracleName)

arg1 : any : atom- Prolog name
*arg2 : any : atom- **ORACLE** name prefixed by owner*

Succeeds for each valid couple Prolog name (*arg1*) - ***ORACLE*** name (*arg2*).

schema/0

schema

Lists all the database relations that have been aliased.

Example

```
?- schema.
Yes
?- accessible(_Owner, 'BEER', _Arity, _Type).
 _Owner = SYSTEM
```

```
 _Arity = 2
 _Type = TABLE
Yes
?- def_dbalias('b', 'SYSTEM.BEER').
Yes
?- dbalias(_PrologName, _OracleName).
 _PrologName = b
 _OracleName = SYSTEM.BEER
Yes
?- schema.
b/2
Yes
```

schema/1

schema(_PrologName)

arg1 : ground : atom - Prolog name of a database relation.

Writes the schema of the relation *arg1* to the current output stream.

schemalist/1

schemalist(_RelationList)

arg1 : any : list of atom/integer

Arg1 is instantiated to a list of the existing aliased relations in the currently opened database. Relations are specified by their Prolog name/arity.

Example

```
?- schemalist(_PrologNameList).
 _PrologNameList = [b / 2,num / 1]
Yes
```

schemalist/3

schemalist(_PrologName, _Arity, _ArgumentList)

arg1 : any : atom - Prolog name of a database relation
arg2 : any : integer - Arity
arg3 : any : list of (atom,atom) - Argument list

Arg2 and *arg3* are respectively instantiated to the arity and the list of arguments of the relation specified in *arg1*. If *arg1* is free, it is instantiated by backtracking to the existing aliased relations.

Example

```
?- schemalist(_PrologName, _Arity, _ArgumentList).
 _PrologName = b
 _Arity = 2
 _ArgumentList = [(RATE , NUMBER),(NAME , CHAR)]
Yes ;
 _PrologName = num
 _Arity = 1
 _ArgumentList = [(ARG1 , LONG)]
Yes
```

3.8 Relation predicates

retrieve/1

retrieve(_Goal)

arg1 : partial : term (a database goal)

Accesses the database to retrieve the tuples that unify with *arg1*. This predicate backtracks on different solutions or fails when there is no solution.
The interface transforms ***ORACLE*** NULL values into empty atoms in ***ProLog by BIM***.

insert/1

insert(_Fact)

arg1 : ground : term (a database fact)

Adds a fact to the ***ORACLE*** database.

delete/1

delete(_Goal)

arg1 : partial : term (a database goal)

Retracts facts that unify with the goal, from the ***ORACLE*** database.
This predicate backtracks on different solutions and fails when there is no more solution.

deleteall/1

deleteall(_Goal)

arg1 : partial : term (a database goal)

Retracts all the facts that unify with the goal.
This predicate always succeeds.

printr/1

printr(_RelName)

arg1 : ground : atom (a database relation name)

Prints out all the tuples of the database relation *arg1* on the current output stream.

3.9 Transparent access

Once a database is open with **openORACLEdb/0-1**, the ***ProLog by BIM*** system distinguishes relations defined in the Prolog database from the ones aliased in the currently open ***ORACLE*** database.

For external relations, the following automatic conversion is performed:

- _Goal ==> retrieve (_Goal)
- assert (_Goal) ==> insert (_Goal)
- retract (_Goal) ==> delete (_Goal)
- retractall (_Goal) ==> deleteall (_Goal)

This mechanism is fully transparent to the user and is particularly useful in the migration from a prototype application using Prolog relations to an operational system retrieving data from a relational database system like ***ORACLE***.

3.10 SQL predicates

sql/1

sql(_SqlCall)

arg1 : ground : atom

Sends a SQL call to ***ORACLE***.

sql/2

sql(_SqlCall, _ResultList)

arg1 : ground : atom
arg2 : any : list

Sends a SQL call to ***ORACLE*** and gets the results one at a time by backtracking in the list *arg2*.

Example

```
?- schema(b).
b/2
stands for : SYSTEM.BEER - 0 records
   NAME                 CHAR          10
   RATE                 NUMBER        22
Yes
?- assert(b(a,1)),
   assert(b(b,2)),
   sql(commit).
Yes
?- printr(b).
b(a,1)
b(b,2)
Yes
?- assert(b(c,3)).
Yes
?- printr(b).
b(a,1)
b(b,2)
b(c,3)
Yes
?- sql(rollback).
Yes
?- printr(b).
b(a,1)
b(b,2)
Yes
```

3.11 Data dictionary

In order to improve the performance and the functionalities of the ***ORACLE*** interface, the structure information of the accessible tables is automatically loaded in the Prolog internal database. This data dictionary can be queried through the following predicates.

info_functors/4

info_functors(_TableName, _Void, _PrologName, _Arity)

arg1 : atom : prefixed table name
arg2 : free : <unused parameter>
arg3 : integer : Prolog name
arg4 : integer : arity

One tuple per accessible ***ORACLE*** relation.

info_arguments/5

info_arguments(_TableName, _ArgNb, _ArgType, _ArgLen, _ArgName)

arg1 : atom : prefixed table name
arg2 : integer : argument number
arg3 : atom : argument type
arg4 : atom : argument length
arg5 : atom : argument name

One tuple per argument of an accessible ***ORACLE*** relation.

Example

```
?- schema(b).
b/2
stands for : SYSTEM.BEER - 2 records
NAME                 CHAR           10
RATE                 NUMBER         22

Yes
?- info_functors(_PrefixedName, _, b, _Arity).
 _PrefixedName = SYSTEM.BEER
 _Arity = 2
Yes

?- info_arguments('SYSTEM.BEER',_ArgNb,
                  _ArgType,_ArgLgth,_ArgName).
 _ArgNb = 2
 _ArgType = NUMBER
 _ArgLgth = 22
 _ArgName = RATE
Yes ;
 _ArgNb = 1
 _ArgType = CHAR
 _ArgLgth = 10
 _ArgName = NAME
Yes
```

3.12 Error handling in ORACLE

showORACLEerrors/1

showORACLEerrors(_Status)

arg1 : any : atom (on/off)

Determines whether the ***ORACLE*** error messages have to be displayed or not.

Example

```
?- printr(b).
b(a,1)
b(b,2)
Yes
?- b(1,a).
Oracle error : code is -1722, op is 3072
ORA-01722: invalid number

No
?- showORACLEerrors(off).
Yes
?- b(1,a).
No
```

err_last/2

err_last(_ErrorKey,_ErrorMessage)

arg1 : any : atom
arg2 : any : atom

Returns the key and message of the last issued ***ORACLE*** error. Fails if no ***ORACLE*** error has been issued or if err_last_reset/0 cleared any memorized error.

The possible Key and associated messages are defined by the err_key_message/2 predicate

err_key_message/2

err_key_message(_ErrorKey,_ErrorMessage)

arg1 : any : atom
arg2 : any : atom

err_last_reset/0

err_last_reset

Clears the last memorized ***ORACLE*** error (if any). This predicate always succeeds.

Database Interfaces

Chapter 4
Interface to Sybase

4.1 Introduction

ProLog provides two levels of interface to the relational database Sybase: a high-level interface for the casual user, and a low-level allowing the direct exploitation of all the advanced features of the Sybase system by the ***ProLog*** user.

The high level is a user-friendly interface that provides about 20 predicates to access the Sybase database in an easy way. The high-level interface can be divided into the following three classes:

The predicates of the generic interface, defined for all interfaces, which can be divided into three main classes: *system predicates*, *relation predicates* and *schema predicates*.

- The system predicates open and close the database.
- The relation predicates are predicates which retrieve, insert, modify or delete records.
- The schema predicates give the user more information about the schema of the current opened database.

The transparent access allows to act on the database predicates in the same way as on the dynamic predicates of the in-core database.

The SQL level allows access to Sybase via standardized SQL calls.

The low-level interface is the implementation in ***ProLog*** of most of the Sybase DB-LIBRARY routines. This rich set offers the programmer the ability to create his own optimized calls to the Sybase database. In addition to the DB-Library predicts the interface itself provides a few predicates which allow easy access to the RDBMS data. The names of these predicates are in lower case preceded by DB.

4.2 Calling the Sybase interface

When during the installation of ***ProLog*** and its interface to Sybase a linked version is created, this interface can be called as

```
% BIMprolog.Sybase
```

See the installation manual or ask your system manager for further information.

On systems running SunOS5, ***ProLog*** and its interface to Sybase can be called with the UNIX-command

```
% BIMprolog -Ldatabases/Sybase
```

If the ***ProLog*** prompt appears on the screen, the Sybase database can be consulted as described in the following chapters.

If the file '-Ldatabases/Sybase.pro' cannot be found, the interface from ***ProLog*** to Sybase is not installed.

On systems runningSunOS 5, if desired, only the low-level interface between ***ProLog*** and Sybase can be called as follows:

```
% BIMprolog -Ldatabases/SybaseLow -Ldatabases/SybaseLowData
```

Supported version

Please refer to the installation manual to check the supported version of Sybase.

4.3 General remarks

Simultaneously opened DBprocesses

The ***ProLog*** interface communicates with the Sybase DataServer via Sybase DBprocesses.

- Deterministic predicates (non-backtrackable) open a DBprocess, send a call to the DataServer, get the result(s), close the DBprocess and succeed (or fail).
- Non-deterministic predicates (backtrackable) open a DBprocess, send a call to the DataServer, get the first solution and succeed (or fail). The other solutions are retrieved one at a time by backtracking. The DBprocess is closed only if all the solutions are exhausted or if the predicate cannot be accessed any longer by backtracking (Generally because of a use of a cut <!>).
  Since there is a maximum number of simultaneously opened DBprocesses on the Sybase DataServer, one is advised to make good use of the cut <!> to avoid this maximum DBprocesses limitation.

Note:

- The low-level predicates **dbgetmaxprocs/1** and **dbsetmaxprocs/1** allow you to access and modify this maximum number of DBprocesses.
- To avoid dead-lock problems the maximum number of open DBprocesses allowed in ***ProLog*** is the Sybase DataServer maximum number minus 5.

Example

```
?- dbgetmaxprocess(_x), write(_x).
55Yes
```

50 open processes are allowed.

Null values

Null values in the database are returned as free variables at the Prolog side.

Opened database

The database which is opened by default is the default database of the user.

4.4 Datatypes

The ProLog-Sybase interface supports

Table 1:

Sybase	Range	ProLog	Range	Note
int	-231.. 231-1	integer	-228.. 228-1	reduced
smallint	-215.. 215-1	integer	-228.. 228-1	
tinyint	0..255	integer	-228.. 228-1	
float	mach.dependent	real	machine dependent	double precission
real	mach_dependent	real	machine dependent	single precission
char(n)	1..255 char	atom	0..MAX_ATOM char	
varchar(n)	1..255 char	atom	0..MAX_ATOM char	
text	< 2 gigabytes	atom list of atoms	0..MAX_ATOM char 0..MAX_ATOM char	if length< MAX_ATOM if length > MAX_ATOM
binary(n)	1..255 bytes	atom	0..MAX_ATOM char	Leading '0x'
varbinary(n)	1..255 bytes	atom	0..MAX_ATOM char	Leading '0x'
bit	[0,1]	integer	-228.. 228-1	
money	$+- 9 billions	atom	0..MAX_ATOM char	Leading '$'
smallmoney	$+- 9billions	atom	0,,MAX_ATOM char	Leading '$'
datatime	< 31/12/9999	arom	0..MAX_ATOM char	
smalldatetime	< 31/12/9999	atom	0..MAX_ATOM char	

Further information on datatypes can be found in the *"TRANSACT-SQL User's Guide"* of Sybase.

4.5 High level

Database structure information

The following mechanism has been implemented to improve the performance of the high level ***ProLog*** Sybase Coupling:

When opening a Sybase database, its structure information is automatically loaded in the ***ProLog*** internal database. This structure information is issued from the Sybase system tables (sysobjects, syscolumns, systypes) and is composed of three predicates:

info_functors/5, info_arguments/5 and **info_types/3.**

Note:

- Only the information about the tables selected by the options of openSYBASEdb or refreshSYBASEdb is loaded.
- After high-level SQL calls that modify the database schema (create a new relation, drop a usertype, ...), the user should refresh the database structure information by using **refreshSYBASEdb/0,1** .

info_arguments/5

info_arguments(_RelId,_ArgNb,_ArgUTypeNb,_ArgLen,_ArgName)

arg1 : any : integer : relation identifier
arg2 : any : nteger : argument number
arg3 : any : integer : argument usertype number
arg4 : any : integer : argument length
arg5 : any : atom : argument name

info_functors/4

info_functors(_RelId,_RelType,_RelName,_RelArity)

arg1 : any : integer : relation identifier
arg2 : any : integer : relation type
arg3 : any : integer : relation name
arg4 : any : integer : relation arity

info_types/3

info_types(_UTypeNb, _UTypeName, _PhysType)

arg1 : any : integer : usertype number
arg2 : any : atom : usertype name
arg3 : any : integer : physical storage datatype

Example

```
relation : People_weight(_name, _birthdate,_weight)
--> info_functors (8098093,U,People_weight,3).
...
--> info_arguments (8098093,1,2,30,_name).
--> info_arguments (8098093,2,12,8,_birthdate).
--> info_arguments (8098093,3,7,4,_weight).
...
--> info_types(2,varchar,39).
...
--> info_types(7,int,56) .
...
--> info_types(12,datetime,61) .
```

System predicates

The system predicates open and close the database and are responsible for keeping the database structure information up to date.

openSYBASEdb/0

openSYBASEdb

Opens the Sybase database with as default USER and PASSWORD, the values of the UNIX environment variables SYBASE_USER and SYBASE_PWD.

openSYBASEdb/1

openSYBASEdb(_OptionList)

arg1 : ground : list of options

Opens the Sybase database. The database structure is loaded in the ***ProLog*** in-core database. The argument is a list of classes of options immediately followed by its value. The different classes are:

Table 2:

Option	Values	Default	Function
'USER' 'U'			Specifies the user name of the Sybase login procedure
'PASSWORD' 'P'		no password	Specifies the password of the user in the Sybase login procedure
'HOST' 'H'			Specifies the hostmachine, where the Sybase datafiles are installed, in the Sybase login procedure
'APPLICATION' 'A'		the value of 'USER'	Specifies the application name in the Sybase login procedure
'LOAD_OBJECTS 'LO'	system, user or all	user	Specifies which relations have to be loaded: system relations (which contain information about the Sybase database itself, e.g. users, relations, indexes,...), user relations (relations defined by the user himself) or both
'LOAD_USER' 'LU'	_user [_user1, ...] all "	the value of 'USER'	_user (specified user) [_user1, ...] (specified users) all (all users) " (your login name) Specifies whose relations have to be loaded. If these relations are from the user who owns this database,'dbo' (database owner) should be specified.
'SHOW_ERRORS' 'SE'	on/off	on	Determines whether the SYBASE error messages have to be displayed or not. (see also "*Error handling*")
'DATABASE' 'DB'		the default user database	Specifies the database which will be opened

See the installation manual or ask your system manager for further information:

```
?-openSYBASEdb([U, prolog, LU, dbo, P, sybase]).
```

This call opens the Sybase database for user 'prolog' with password 'sybase', and loads all the user relations of the database owner (dbo).

The environment variable DSQUERY determines which server will be accessed. Setting this variable with setenv/2 allows to determine which server will be accessed. DBuse/1 allows to specify the database which will be accessed. **refreshSYBASEdb/0,1** should be used to update the low-level structure information if the value of DSQUERY is changed after the opening of the database.

closeSYBASEdb/0

closeSYBASEdb

The Sybase database is no longer accessible from ***ProLog***.

refreshSYBASEdb/0

refreshSYBASEdb

Refreshes the database structure information.

refreshSYBASEdb/1

refreshSYBASEdb(_OptionList)

arg1 : ground : list of options

Refreshes the database structure information according to arg1. The argument is a list of classes of options immediately followed by its value (for a list of possible values see openSYBASEdb).

statusSYBASEdb/1

statusSYBASEdb(_Status)

arg1 : any : integer

The argument is 0 when the database is open, -1 otherwise.

helpSYBASEdb/0

helpSYBASEdb

Displays help information regarding the SYBASE interface on the standard output stream.

caseSYBASEsensitive/1

caseSYBASEsensitive(_OnorOff)

arg1 : any : atom ('on' or 'off')

When "off", no distinction is made between upper-case and lower-case letters in like-format retrieve queries sent to Sybase. Default=on.

Example

```
?- printr(beer).
beer(Jupiler,4)
beer(Palm,5)
beer(Pils,9)
beer(Grimbergen,8)
Yes
?- caseSYBASEsensitive(on).
Yes
?- beer('%GrimBERGEN%':_name,_rate).
No

?- caseSYBASEsensitive(off).
Yes
?- beer('%GrimBERGEN%':_name,_rate).
 _name = Grimbergen
 _rate = 8
Yes
```

Error handling

showSYBASEerrors/1

showSYBASEerrors(_Status)

arg1: any : atom (on\|off) or term (on(_errno)/off(_errno))

Determines whether the Sybase error messages have to be displayed.

Example

```
?- schema(beer).
beer/2    user table - owner : dbo - 3 records
        arg1                varchar     10
        arg2                int          4
```

```
Yes
?- printr(beer).
beer(Orval,8)
beer(Leffe,9)
beer(Palm,5)
Yes
?- showSYBASEerrors(_X).
 _X = on
Yes
?- beer(2,2).
*** SYBASE 1322 *** : Call to SYBASE dbsqlexec() Failed.
*** Error Occured in clause : retrieve/1
*** SYBASE message :
*** General SQL Server error:
Check messages from the SQL Server.
*** DataServer messages :
*** Msg 257, Level 16, State 1:
*** Implicit conversion from datatype 'int' to 'varchar' is
not allowed.
*** Use the CONVERT function to run this query.
No
?- showSYBASEerrors(off).
Yes
?- beer(2,2).
No
?- showSYBASEerrors(on).
Yes
?- showSYBASEerrors(off(1322)).
Yes
?- beer(2,2).
No
?- showSYBASEerrors(on(1322)).
Yes
?- beer(2,2).
*** SYBASE 1322 *** : Call to SYBASE dbsqlexec() Failed.
*** Error Occured in clause : retrieve/1

*** SYBASE message :
*** General SQL Server error:Check messages from the SQL
Server.
*** DataServer messages :
*** Msg 257, Level 16, State 1:
*** Implicit conversion from datatype 'int' to 'varchar' is
not allowed.
*** Use the CONVERT function to run this query.
No
```

Schema predicates

The schema predicates give the user more information about the schema of the current opened database.

schema/0

schema

Lists all the database relation names on the current output stream.

Example

```
?- schema.
Person/4
birthday/2
salary/2
bimcustomer/23
basicpart/3
composed_of/3
assembly/2
```

```
names/4
beer/3
taxes/2
Yes
```

schema/1

schema(_RelName)

arg1 : ground : atom (a database relation name)

Writes the schema of the relation to the current output stream.

Example

```
?- schema(beer).
beer/3
   user table - owner : dbo - 7 records
   name          varchar    15
   strength      smallint   2
   price         smallint   2
Yes
```

The relation name is followed by a code, specifying its type ('U' for user relation or 'S' for system relation. See also option 'LOAD_OBJECTS' of the predicate **openSYBASEdb/1**.

schemalist/1

schemalist(_RelSchema)

arg1 : any : list of atom/integer (relation/arity)

The list *arg1* is instantiated to the existing relations in the currently opened database.

Example

```
?- schemalist(_Schemalist).
 _SchemaList = [beer/1, b/2, taxes/4]
Yes
```

schemalist/3

schemalist(_RelName, _Arity, _ArgumentList)

arg1 : any : atom (a database relation name)
arg2 : any : integer (arity)
arg3 : any : list of (atom,atom) <argument name, type>

Arg2 and *arg3* are respectively instantiated to the arity and the list of arguments of the relation specified in *arg1*. If *arg1* is free, it is instantiated by backtracking to the different relations in the database.

Relation predicates

The relation predicates are predicates which retrieve, inert or delete records. The following variants exist:

- retrieve/1, retreive_first/1,retrieve_no/1, retrieve/2, retrieve_orderby/2, printr/1, join/1
- insert/1
- delete/1, deleteall/1

Retrieval

Retrieval can be done on portions of patterns such as the keyword "LIKE" in SQL (see SYBASE - TRANSACT - SQL USER'S Guide - LIKE keyword).

The SYBASE wildcards may be used respecting the SQL syntax:

Wildcard	Meaning
%	any string of zero or more characters
_	any single character

[]	any single character within the specified range (e.g., [a-f]) or set (e.g., [abcdef])
[^]	any single character not within the specified range (e.g., [^a-f]) or set (e.g., [^abcdef])

The argument you want to retrieve must bear the following format:

arg1 : arg2

arg1 : ground : atom (the character string containing the wildcards)
arg2 : free : atom (the argument that will be instantiated by the retrieve)

Example

```
?- retrieve( beer('%a%':_name,_,_) ),
   write(_name),nl,
   fail.
Orval
Maes
Stella
Chimay
No
```

Please consult the remark in the beginning of this chapter for further information on simultaneously opened DBprocesses.

retrieve/1

retrieve(_Goal)

arg1 : partial : term (a database goal)

Accesses the database to retrieve the tuples that unify with *arg1*. This predicate backtracks on different solutions or fails when there is no solution.

The interface transforms Sybase NULL values into free arguments in ***ProLog***.

Example

```
?- retrieve( beer(_name,_strength,_) ),
   write(_name),tab(1),
   write(_strength),
   write('  degrees'),nl,
   fail.
Orval           10   degrees
Maes            5    degrees
Leffe           10   degrees
Jupiler         4    degrees
Stella          4    degrees
Chimay          10   degrees
Kriek           8    degrees
No
```

retrieve_first/1

retrieve_first(_Goal)

arg1 : partial : term

Accesses the SYBASE database to retrieve the first tuple that unifies with *arg1*. *Arg1* is unified with the first tuple of the relation. The predicate fails if there is no solution.

retrieve_not/1

retrieve_not(_Goal)

arg1 : partial : term (a database goal)

Equals to "\+ retrieve(_Goal)" but more time efficient.

retrieve_orderby/2

retrieve_orderby(_Goal, _OrderbyAtom)

arg1 : partial (a database goal)
arg2 : ground : atom (' arg# asc|desc , ... ')

Accesses the database to retrieve tuples that unify with the goal. Tuples are ordered in an ascending or descending way according to *arg2*. The format for the atom *arg2* is :

'arg# asc|desc[,arg# asc|desc]...'

Example

```
?- retrieve_orderby( beer(_Name, _Rate), '1 asc, 2 desc').
 _Name = Leffe
 _Rate = 4
Yes ;

 _Name = Maes
 _Rate = 8
Yes ;

 _Name = Maes
 _Rate = 7
Yes ;

 _Name = Palm
 _Rate = 9
Yes
```

Tuples are ordered on argument 1 (_Name) ascendingly.
Tuples with identical _Name are ordered descendingly on _Rate.

printr/1

printr(_RelName)

arg1 : ground : atom (a database relation name)

Prints out all the tuples of the database relation, on the current output stream.

Example

```
?- printr(beer).
beer(Orval,10,30)
beer(Maes,5,25)
beer(Leffe,10,38)
beer(Jupiler,4,23)
beer(Stella,4,22)
beer(Chimay,10,36)
beer(Kriek,8,28)
```

Joining external relations

join/1

join(_DbcallList)

arg1: partial : list of database goals

Accesses the database to solve the conjunction of database goals.

Example

```
?- join( [beer(_Name,_Rate), taxes(_Rate,_Percent)] ).
 _Name = Palm
 _Rate = 5
 _Percent = 20
Yes ;
```

```
 _Name = Orval
 _Rate = 8
 _Percent = 23
Yes
```

Insertion

insert/1

insert(_Fact)

arg1 : partial : term (a database fact)

Adds a fact to the Sybase Database.

Free arguments are treated by SYBASE as NULL or DEFAULT VALUES according to the SYBASE table definition.

Example 1

```
?- printr(beer).
beer(Orval,10,30)
beer(Maes,5,25)
beer(Leffe,10,38)
beer(Jupiler,4,23)
beer(Stella,4,22)
beer(Chimay,10,36)
beer(Kriek,8,28)
Yes
?- insert( beer(Kwak,8,32) ),
   printr(beer).
beer(Orval,10,30)
beer(Maes,5,25)
beer(Leffe,10,38)
beer(Jupiler,4,23)
beer(Stella,4,22)
beer(Chimay,10,36)
beer(Kriek,8,28)
beer(Kwak,8,32)
Yes
```

Deletion

delete/1

delete(_Goal)

arg1 : partial : term (a database goal)

Retracts facts that unify with the goal, from the Sybase Database.
This predicate backtracks on different solutions and fails when there are no more solutions.

Note:

- Since **delete/1** uses the SQL delete command of Sybase, duplicates of a tuple are always deleted immediately, although backtracking occurs on multiple solutions.
- Please consult the remark in the beginning of this chapter for further information on simultaneously openDBprocesses.

Example

```
?- printr(beer),
   delete( beer(_name,8,32) ),
   write(_name),nl,fail.
beer(Orval,10,30)
beer(Maes,5,25)
beer(Leffe,10,38)
beer(Jupiler,4,23)
beer(Stella,4,22)
```

```
beer(Chimay,10,36)
beer(Kriek,8,28)
beer(Kwak,8,32)
beer(Kwak,8,32)
Kwak
Kwak
Yes
?-printr(beer).
beer(Orval,10,30)
beer(Maes,5,25)
beer(Leffe,10,38)
beer(Jupiler,4,23)
beer(Stella,4,22)
beer(Chimay,10,36)
beer(Kriek,8,28)
Yes
```

Example 2

```
?- printr(beer),
   delete(beer(_name,8,32)),
   write(_name),nl,!.
beer(Orval,10,30)
beer(Maes,5,25)
beer(Leffe,10,38)
beer(Jupiler,4,23)
beer(Stella,4,22)
beer(Chimay,10,36)
beer(Kriek,8,28)
beer(Kwak,8,32)
beer(Kwak,8,32)
Kwak       <-- only printed once
Yes
?- printr(beer).
beer(Orval,10,30)
beer(Maes,5,25)
beer(Leffe,10,38)
beer(Jupiler,4,23)
beer(Stella,4,22)
beer(Chimay,10,36)
beer(Kriek,8,28)
           <-- but all occurences are deleted
Yes
```

deleteall/1

deleteall(_Goal)

arg1 : partial : term (a database goal)

Retracts all the facts that unify with the goal.
This predicate always succeeds.

4.6 Embedded SQL predicates

The coupling with Sybase offers a true embedded SQL. This presents the most straightforward access to the full power of the Sybase system. SQL statements can be embedded in the ***ProLog*** source, while results are displayed on the current output stream or unified with a ***Prolog*** variable for further processing in the ***Prolog*** program.

The SQL predicates mentioned below access the database which has been identified with the system predicate openSYBASEdb.

The following possibilities to use SQL exist.

Table 3:

Return of results	unique command	step by step
on current output stream	sql/1	sqlcmd/1, sqlexec/0
in a variable list	sql/2	sqlcmd/1, sqlexec/1

See also the remark in the beginning of this chapter for further information on simultaneously opened DBprocesses.

sql/1

sql(_SQL_String)

arg1 : ground : atom (SQL_string)

Sends the SQL_string to the Sybase DataServer for execution.
The possible results returned by the DataServer are displayed in the Sybase format on the current output stream.

Example

```
?- sql('select name,price from beer').
Orval           30
Maes            25
Leffe           38
Jupiler         23
Stella          22
Chimay          36
Kriek           28
Yes
```

sql/2

sql(_SQL_String, _VarList)

arg1 : ground : atom (SQL_string)
arg2 : any : list

This is the logical extension of **sql/1** : the results of the sql call are received one at a time by backtracking, and unified with *arg2*.

Please consult the remark in the beginning of this chapter for further information on simultaneously openDBprocesses.

Example 1

```
?- sql('select name,price from beer',_list),
   write(_list),nl,
   fail.
[Orval,30]
[Maes,25]
[Leffe,38]
[Jupiler,23]
[Stella,22]
[Chimay,36]
[Kriek,28]
No
```

Example 2

```
?- sql('select name,price from beer',[_name,_price]),
   write(_name:_price),
   write(' BF.'),nl,
   fail.
Orval : 30 BF.
Maes : 25 BF.
```

```
Leffe : 38 BF.
Jupiler : 23 BF.
Stella : 22 BF.
Chimay : 36 BF.
Kriek : 28 BF.
No
```

sqlcmd/1

sqlcmd(_SQL_String)

arg1 : ground : atom (SQL_string)

Puts the SQL_string (or part of an SQL_string) into the Sybase command buffer.

sqlexec/0

sqlexec

Sends the contents of the Sybase COMMAND BUFFER to the DataServer for execution. The possible results returned by the DataServer are displayed in the Sybase format on the current output stream.

Example

```
?- sqlcmd('select name,(price + price * tax_rate)'),
   sqlcmd('from beer,taxes where strength=alcohol_rate'),
   sqlexec.
Jupiler    27.370000
Stella     26.180000
Maes       29.750000
Kriek      35.000000
Orval      39.900000
Leffe      50.540000
Chimay     47.880000
Yes
```

sqlexec/1

sqlexec(_VarList)

arg1 : any : list

Sends the contents of the Sybase COMMAND BUFFER to the DataServer for execution. The possible results returned by the DataServer are received one by one by backtracking and unified with *arg1*.

See also the remark in the beginning of this chapter for further information on simultaneously opened DBprocesses.

Example

```
?- sqlcmd('select name,(price + price * tax_rate)'),
   sqlcmd(' from beer,taxes where strength=alcohol_rate'),
   sqlexec( [_name,_customer_price] ),
   write(_name), tab(1),
   write(' costs '),
   write(_customer_price),
   write(' BF.'), nl,
   fail.
Jupiler    costs 27.37 BF.
Stella     costs 26.18 BF.
Maes       costs 29.75 BF.
Kriek      costs 35.0 BF.
Orval      costs 39.9 BF.
Leffe      costs 50.54 BF.
Chimay     costs 47.88 BF.
No
```

statusSYBASEcmd/1

statusSYBASEcmd(_Status)

arg1 : any : integer

The argument is -1 if the Sybase command buffer is empty, 0 otherwise.

DBuse/1

'DBuse'(_DatabaseName)

arg1 : ground : atom

The database which is accessed in the SQL commands is changed to database arg1. No implicit refreshSYBASE/0,1 is performed and the low-lowel database structure information is not updated. refreshSYBASE/0,1 should be called explicitly when accessing the database through the transparent access routines.

4.7 Low level

The low-level interface is the implementation in ***ProLog*** of most of the Sybase DB-LIBRARY routines. In addition to the DB-Library predicates, the interface provides a few predicates which allow easier access to the RDBMS data.

Low-level interface predicates

The low-level interface is the implementation in ***ProLog*** of most of the Sybase DB-LIBRARY routines. In addition to the DB-Library predicates the interface provides a few predicates which allow easier access to the RDBMS data. The names of these predicates are in lower case preceded by DB.

Low-level set-up

DBconvertstatus/1

'DBconvertstatus'(_DBconvertstatus)

arg1 : free or ground : atom (atom/integer)

Indicates if the return code values of the calls to Sybase routines have to be integers or converted to atoms.

The possible values for *arg1* are: 'integer' and 'atom'.
If *arg1* is integer, the return values will be returned as integers.
If *arg1* is atom, the return values will be returned as atoms.
If *arg1* is free, it will be instantiated to the current value of the convertstatus.

DB_ColWidth/1

'DB_ColWidth'(_Width)

arg1 : free or ground : integer

If *arg1* is free, it will be instantiated to the value of the variable ColWidth (see also **dbprrow/1** and **dbprhead/1**), otherwise it will change its value.

Access predicates

DBaccess/3

'DBaccess'(_DBproc, _ColNb, _Value)

arg1: ground : pointer
arg2: ground : integer
arg3: free

Arg3 is instantiated to the value of the data in the *arg2*'th column. The type of the data is automatically detected.

DBaccess/4

'DBaccess'(_DBproc, _ColNb, _Value, _Type)

arg1: ground : pointer
arg2: ground : integer
arg3: free
arg4: ground:atom/integer (depending on DBconvertstatus)

Arg3 is instantiated to the value of the data in the *arg2*'th column. *Arg4* specifies the type of the data.

Processing command batch results

DBgetvalue/5

'DBgetvalue'(_DBproc, _ColNb, _BoundVariable, _BindType, _Value)

arg1 : ground : pointer
arg2 : ground : integer
arg3 : ground : pointer
arg4 : ground : atom/integer (depending on DBconvertstatus)
arg5 : free

Gets the value of a bound variable (see **dbbind/6**).

Low-level error handling

BP_dberrhandle/1

'BP_dberrhandle'(_Name)

arg1: ground : atom

Installs as Sybase error handler, the global predicate whose functor is *arg1* and arity equals 7. The arguments passed from Sybase to this handler are:

1. *dbproc: pointer*
2. *severity: integer*
3. *dberr: integer*
4. *oserr: integer*
5. *dberrstr: atom*
6. *oserrstr: atom*
7. *return_code: integer (must be instantiated in the handler and returned to Sybase with one of the following values : 0:INT_EXIT , 1:INT_CONTINUE or 2:INT_CANCEL)*

Any existing error handler can be de-installed by specifying an empty atom in *arg1*.

BP_dbmsghandle/1

'BP_dbmsghandle'(_Name)

arg1: ground : atom

Installs as Sybase message handler, the global predicate whose functor is *arg1* and arity=8. The arguments passed from Sybase to this handler are:

1. *dbproc: pointer*
2. *msgno: integer*
3. *msgstate: integer*
4. *severity: integer*
5. *msgtext: atom*
6. *srvname: atom*
7. *procname: atom*
8. *line: integer*

The return code sent to Sybase is always 0 and thus does not appear in this predicate header. Any existing message handler can be de-installed by specifying an empty atom in *arg1*.

DB-library predicates

The following DB-library routines are supported by the interface. The order of the parameters is exactly the same as in the corresponding library routine, possibly preceded by a returncode. Further information on these routines can be found in the "*Sybase DB-Library Reference Manual*".

Examples of how to use the low-level interface

Example 1

```
send_sql(_sqlstring) :-
     dblogin(_login),
     ( _login == 0x0 ->
          write('** dblogin failed **'),
          fail
     ;
          true
     ),
     'DBSETLUSER'(_login, 'Rand'),
     dbopen(_dbproc, _login, ''),
     ( _dbproc == 0x0 ->
          write('** dbopen failed **'),
          fail
     ;
          true
     ),
     dbcmd(_dbproc, _sqlstring),
     dbsqlexec(_return, _dbproc),
     ( _return == 'SUCCEED' ->
          true
     ;
          write('** dbsqlexec failed **'),
          dbclose(_dbproc),
          fail
     ),
     dbresults(_return2, _dbproc),
     ( _return2 == 'SUCCEED' ->
          true
     ;
          write('** dbresults failed **'),
          dbclose(_dbproc),
          fail
     ),
     dbprrow(_dbproc),
     dbclose(_dbproc).
```

Example 2

```
get_first_usertype (_usertype, _name, _type) :-
     dblogin(_login),
     ( _login == 0x0 ->
          write('** dblogin failed **'),
          fail
     ;
          true
     ),
     'DBSETLUSER'(_login, 'Rand'),
     dbopen(_dbproc, _login, ''),
     ( _dbproc == 0x0 ->
          write('** dbopen failed **'),
          fail
     ;
          true
     ),
     dbcmd(_dbproc,
           'select usertype,name,type from systypes '),
```

```
        dbsqlexec(_return, _dbproc),
        ( _return == 'FAIL' ->
             write('** dbsqlexec failed **'),
             dbclose(_dbproc),
             fail
        ;
             true
        ),
        dbresults(_return2, _dbproc),
        ( _return2 == 'FAIL' ->
             write('** dbresults failed **'),
             dbclose(_dbproc),
             fail
        ;
             true
        ),
        dbnextrow(_return3, _dbproc),
        ( _return3 == 'NO_MORE_ROWS' ->
             write('** No rows for this call **')
        ;
             DBaccess(_dbproc, 1, _usertype),
             DBaccess(_dbproc, 2, _name),
             DBaccess(_dbproc, 3, _type)
        ),
        dbclose(_dbproc).
```

The DB-library routine mappings

dbadata/4

dbadata(_Pointer, _Dbproc, _ComputeID, _ColumnNb)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : integer

Returns a pointer to the data for a particular column in a compute.

dbadlen/4

dbadlen(_Length, _Dbproc, _ComputeID, _Column)

arg1 : free : integer (length)
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : integer

Gets the length of the data for a particular column in a compute.

dbaltcolid/4

dbaltcolid(_Colid, _Dbproc, _ComputeID, _ColumnNb)

arg1 : free : integer (colid)
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : integer

Finds the operand column id for a particular column in a compute.

dbaltbind/7

dbaltbind(_Return, _Dbproc, _ComputeID, _ColumnNb, _VarType, _VarLen, _VarAddr)

arg1 : any : returnvalue
arg2 : qround : pointer
arg3 : ground : integer
arg4 : ground : integer
arg5 : ground : integer
arg6 : ground : integer
arg7 : free

dbaltlen/4

dbaltlen(_NbOfColumns, _Dbproc, _ComputeID, _ColumnNb)

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

Finds the maximum length of the data for a particular column in a compute.

dbaltop/4

dbaltop(_OperatorType, _Dbproc, _ComputeID, _ColumnNb)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : integer

Finds the type of aggregate operator for a particular column in a compute.

dbalttype/4

dbalttype(_OperatorType, _Dbproc, _ComputeID, _ColumnNb)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : integer

Finds the datatype for a particular column in a compute.

dbbind/6

dbbind(_Returncode, _Dbproc, _ColumnNb, _Type, _Length, _Variable)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : atom/integer (depending on DBconvertstatus)
arg5 : ground : integer
arg6 : ground : pointer

Binds the DataServer query from a column number *arg3* to a program variable *arg6* with the type indicated in *arg4* and with a maximum length of *arg5*. Since the program variable only contains the pointer to the bound variable, use the additional interface predicate **DBgetvalue/3** to access the value.

The different types that can be used as *arg4* are:

INTBIND	(int)
SMALLBIND	(smallint)
TINYBIND	(tinyint)
FLT8BIND	(float)
CHARBIND	(char)
NTBSTRINGBIND	(varchar)

BINARYBIND(	binary)
BITBIND	(bit)
MONEYBIND	(money)
DATETIMEBIND	(datetime)

dbbylist/4

dbbylist(_Return, _Ddbproc, _Computeid, _Size)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : integer
arg4 : free : integer

dbcancel/2

dbcancel(_ReturnCode,_Dbproc)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer

Cancels the current command batch.

dbcanquery/2

dbcanquery(_Returncode, _Dbproc)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer

Cancels any pending rows from the most recently executed query.

dbchange/2

dbchange(_NewDbName, _Dbproc)

arg1 : free : atom
arg2 : ground : pointer

Checks to see if the database context has changed.

dbclose/1

dbclose(_Dbproc)

arg1 : ground : pointer

Closes and deallocates a DBPROCESS structure.

dbclrbuf/2

dbclrbuf(_Dbproc, NbOfRows)

arg1 : ground : pointer
arg2 : ground : integer (number of rows to drop)

Drops rows from the row buffer.

dbclropt/4

dbclropt(_Returncode, _Dbproc, _Option, _Parameter)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg4 : ground: atom/integer (depending on DBconvertstatus)
arg5 : ground : atom (parameter)

Clears an option set by **dbsetoption/4**.

dbcmd/3

dbcmd(_Returncode, _Dbproc, _Text)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : atom

As **dbcmd/2** but returns a status.

DBCMDROW/2

'DBCMDROW'(_Returncode, _Dbproc)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer

Tells if the current command is a command that returns rows.

dbcolbrowse/3

arg1: any : atom/integer (depending on DBconvertstatus)
arg2: ground : pointer
arg3: ground : integer

dbcollen/3

dbcollen(_Length, _Dbproc, _ColumnNb)

arg1 : free : integer
arg2 : ground : pointer
arg3 : ground : integer

Arg1 is instantiated to the maximum length (in bytes) of the variable in the *arg3*'th column of the DataServer result.

dbcolname/3

dbcolname(_ColumnName, _Dbproc, _ColumnNb)

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

Arg1 is instantiated to the *arg3*'th column name of the DataServer query.

dbcolsource/3

dbcolsource(_ColName, _Dtsproc, _ColumnNb)

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

dbcoltype/3

dbcoltype (_Type, _Dbproc, _ColumnNb)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground: pointer
arg3 : ground: integer

Arg1 is set to the column type (SYBINT1, SYBINT2, SYBINT4, SYBFLT8, SYBCHAR, SYBBINARY, SYBDATETIME or SYBMONEY) of the data in the *arg3*'th column of the DataServer query or to -1, if the column number is not in range.

DBCOUNT/2

'DBCOUNT'(_NbOfRows, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Displays the number of rows affected by a SQL command.

DBCURCMD/2

'DBCURCMD'(_CommandNb, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Gives the number of the current command.

DBCURROW/2

'DBCURROW'(_RowNb, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Gives the number of the first row in the row buffer.

dbdata/3

dbdata(_Data, _Dbproc, _ColumnNb)

arg1 : free : pointer
arg2 : ground: pointer
arg3 : ground: integer

dbdatlen/3

dbdatlen(_Length, _Dbproc, _ColumnNb)

arg1 : free : integer
arg2 : ground: pointer
arg3 : ground: integer

arg1 is instantiated to the actual length (in bytes) of the variable in the *arg3*'th column of the DataServer query

DBDEAD/2

'DBDEAD'(_DbBool, _Dbproc)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer

Determines whether a particular DBprocess is dead.

dberrhandle/2

dberrhandle(_PrevHandler, _Handler)

arg1 : free : pointer
arg2 : ground : pointer

dbexit/0

dbexit

Closes and deallocates all open DBPROCESS structures.

DBFIRSTROW/2

'DBFIRSTROW'(_RowNb, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

dbfreebuf/1

dbfreebuf(_Dbproc)

arg1 : ground : pointer

Clears the command buffer of the *arg1* DBPROCESS structure.

dbfreequal/1

dbfreequal(_QualPtr)

arg1: ground : pointer

dbgetchar/3

dbgetchar(_Char, _Dbproc, _CharNb)

arg1 : free : atom
arg2 : ground : pointer
arg3 : ground : integer (number of the char to find)

Returns the nth character in the command buffer.

dbgetmaxprocs/1

dbgetmaxprocs(_MaxNbOfDbproc)

arg1 : free : integer

Determines the current maximum number of simultaneously open DBprocesses.

dbgetoff/4

dbgetoff(_CharacterOffset, _Dbproc, _OffsetType, _Where)

arg1 : free : integer
arg2 : ground : pointer
arg3 : ground : integer/atom
arg4 : ground : integer (where in the buffer)

Checks for the existence of SQL keywords in the command buffer.

dbgetrow/3

dbgetrow(_Returncode, _Dbproc)

arg1 : any : atom/integer
(REG_ROW / NO_MORE_ROWS / BUF_FULL or compute_id)
arg2 : ground : pointer

Sets the current row.

DBGETTIME/1

'DBGETTIME'(_NbOfSeconds)

arg1 : free : integer

Gets the number of seconds that DB-Library will wait for a SQL Server response to a SQL command.

dbgetuserdata/2

dbgetuserdata(_Return, _Dbproc)

arg1: free : pointer
arg2: ground : pointer

dbhasretstat/2‘

dbhasretstat((_Dbbool, _Dbproc)

arg1: free : integer
arg2: ground:pointer

dbinit/1

dbinit(_Returncode)

arg1 : free : atom/integer (depending on dbconvertstatus)

DBIORDESC/2

'DBIORDESC'(_FileDescriptor, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Provides program access to the UNIX file descriptor used by a DBprocess to read data coming from the DataServer.

DBIOWDESC/2

'DBIOWDESC'(_FileDescriptor, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Provides program access to the UNIX file descriptor used by a DBprocess to write data coming from the DataServer.

DBISAVAIL/2

'DBISAVAIL'(_DbBool, _Dbproc)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer

Tells if the DBprocess is available for general use.

dbisopt/4

dbisopt(_Returncode, _Dbproc, _Option, _Parameter)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : atom/integer (depending on DBconvertstatus)
arg4 : ground : atom (parameter)

Checks if the DataServer or DB-LIBRARY option *arg3* has been set to the value *arg4*.

DBLASTROW/2

'DBLASTROW'(_RowNb, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Returns the number of the last row in the row buffer.

dblogin/1

dblogin(_LoginRecord)

arg1 : free : pointer (LOGINRECORD)

Allocates a login parameter structure for use in **dbopen/3**.

This structure may be filled with **DBSETL.../2** calls.

DBMORECMDS/2

'DBMORECMDS'(_Returncode, _Dbproc)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer

Indicates whether there are more commands to be processed.

dbmoretext/4

dbmoretext(_Retcode, _Dbproc, _Sizetext)

arg1: any : integer/integer (depending on DBconvertstatus)
arg2: ground : pointer
arg3: ground : integer
arg4: ground : pointer

dbmsghandle/2

dbmsghandle(_PrevHandler, _Handler)

arg1 : free : pointer
arg2 : ground : pointer

dbname/2

dbname(_DbName, _Dbproc)

arg1 : free : atom
arg2 : ground : pointer

Gets the name of the current database.

dbnextrow/2

dbnextrow(_Returncode, _Dbproc)

arg1 : any : atom/integer
(REG_ROW / NO_MORE_ROWS / BUF_FULL or compute_id)
arg2 : ground : pointer

Reads in the next row.

dbnumalts/3

dbnumalts(_NbOfColumns, _Dbproc, _ComputeID)

arg1 : free : integer
arg2 : ground : pointer
arg3 : ground : integer

Returns the number of columns in a particular compute row.

dbnumcols/2

dbnumcols(_NbOfColumns, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Arg1 is instantiated to the number of columns of the DataServer result.

dbnumcompute/2

dbnumcompute(_NbOfComputes, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Returns the number of COMPUTE clauses in the select statement.

DBNUMORDERS/2

'DBNUMORDERS'(_NbOfColumns, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Returns the number of columns named by a SQL SELECT command's ORDER BY clause.

dbnumrets/2

dbnumrets(NbOfRetPar, _Dbproc)

arg1: free : integer
arg2: ground : pointer

dbopen/3

dbopen(_Dbproc, _LoginRecord, _Server)

arg1 : free : pointer
arg2 : ground : pointer (LOGINRECORD)
arg3 : ground : atom

Creates and initializes a DBPROCESS structure with the LOGINRECORD structure which is created with **dblogin/1**. The DBprocess is connected to the DataServer named *arg3* (default is 'DSQUERY').

dbordercol/3

dbordercol(_ColumnNb, _Dbproc, _Columnid)

arg1 : free : integer
arg2 : ground : pointer
arg3 : ground : integer (ORDER BY column id)

Returns the nth column in the most recently executed query's ORDER BY clause.

dbprhead/1

dbprhead(_Dbproc)

arg1 : ground : pointer

Prints out the column headings for the rows returned by the DataServer. The length of the lines can be set with the predicate **DB_ColWidth/1**.

dbprrow/2

dbprrow(_Dbproc)

arg1 : free : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer

Prints out all the rows returned by the DataServer. The length of the lines can be set with the predicate **DB_ColWidth/1**.

dbprtype/2

dbprtype(_String, _Token)

arg1 : free : atom
arg2 : ground : integer/integer (depending on DBconvertstatus)

Converts a DataServer type token to a readable string.

dbqual/4

dbqual(_Return, _Dbproc, _Tabnum, _Tabname)

arg1: free : pointer
arg2: ground : pointer
arg3: ground : integer
arg4: ground : atom

DBRBUF/2

'DBRBUF'(_Returncode, _Dbproc)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer

Determines whether the DB-LIBRARY network buffer contains any unread bytes.

dbreadpage/5

dbreadpage(_Int, _Dbproc, _Dbname, _PageNb, _Buf)

arg1 : free : integer
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : integere
arg5 : ground : pointer

dbresults/2

dbresults(_ReturnCode, _Dbproc)

arg1 : any : atom//integer (SUCCEED/FAIL/NO_MORE_RESULTS or 1/0/2 depending on DBconvertstatus)
arg2 : ground : pointer

Sets up the results of the next query.

dbretdata/3

dbretdata(_RetData, _Dbproc, _Retnum)

arg1 : free : pointer
arg2 : ground : pointer
arg3 : ground : integer

dbretlen/3

dbretlen(_Retlen, _Dbproc, _Retnum)

arg1 : free : integer
arg2 : ground : pointer
arg3 : ground : integer

dbretname/3

dbretname(_Retname, _Dbproc, _Retnum)

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

dbretstatus/2

dbretstatus(_Retstatus, _Dbproc, _Retnum)

arg1 : free : integer
arg2 : ground : pointer

dbrettype/3

dbrettype(_Rettype, _Dbproc, _Retnum)

arg1 : free : integer
arg2 : ground : pointer
arg3 : ground : integer

DBROWS/2

'DBROWS'(_Returncode, _Dbproc)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer

Indicates whether the current command is going to return rows.

DBROWTYPE/2

'DBROWTYPE'(_Returncode, _Dbproc)

arg1 : any : integer/atom
(compute_id or REG_ROW / NO_MORE_ROWS / BUF_FULL)
arg2 : ground : pointer

Reports the type of the current row.

dbrpcinit/4

dbrpcinit(_Retcode, _Dbproc, _Rpcname, _Options)

arg1 : any : integer/atom (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : integer/atom (depending on DBconvertstatus)

dbrpcparam/8

dbrpcparam(_Retcode, _Dbproc, _Paramname, _Status, _Type, _Maxlen, _Datalen, _Value)

arg1 : any : integer/atom (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : atom
arg4 : ground : integer/atom (depending on DBconvertstatus)
arg5 : ground : integer/atom (depending on DBconvertstatus)
arg6 : ground : integer
arg7 : ground : integer
arg8 : ground : pointer

dbrpcsend/2

dbrpcsend(_Retcode, _Dbproc)

arg1 : any : integer
arg2 : ground : pointer

dbrpwclr/1

dbrpwclr(_Loginrec)

arg1: ground : pointer

dbrpwset/5

dbrpwset(_Retcode, _Loginrec, _Surname, _Password, _Dirlen)

arg1: free : integer
arg2: ground : pointer
arg3: ground : atom
arg4: ground : atom
arg5: ground : integer

dbsetavail/1

dbsetavail(_Dbproc)

arg1 : ground : pointer

Marks this DBprocess as being available for general use.

dbsetifile/1

dbsetifile(_InterfaceFile)

arg1 : ground : atom

Specifies the SYBASE interface file that dbopen() will use to connect to a DataServer.

DBSETLAPP/3

'DBSETLAPP'(_ReturnCode, _LoginRecord, _ApplicationName)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : pointer (LOGINRECORD)
arg3 : ground : atom

Sets the application name of the LOGINRECORD structure to *arg2*.

DBSETLHOST/3

'DBSETLHOST'(_ReturnCode, _LoginRecord, _HostName)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer (LOGINRECORD)
arg3 : ground : atom

Sets the host name of the LOGINRECORD structure to *arg2*.

dbsetlogintime/2

dbsetlogintime(_Returncode, _NumberOfSeconds)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : integer (sets the number of seconds (arg2) that DB-LIBRARY waits for a DataServer response to a request for a DBPROCESS connection)

DBSETLPWD/3

'DBSETLPWD'(_ReturnCode, _LoginRecord, _Password)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer (LOGINRECORD)
arg3 : ground : atom

Sets the users password of the LOGINRECORD structure to *arg2*.

DBSETLUSER/3

'DBSETLUSER'(_ReturnCode, _LoginRecord, _UserName)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer (LOGINRECORD)
arg3 : ground : atom

Sets the users name of the LOGINRECORD structure to *arg2*.

Example

```
?- dblogin(_login),
   'DBSETLUSER'(_login, 'Rand'),
   'DBSETLPWD'(_login,'PKV12'),
   dbopen(_dbproc,_login,''),
                { ! default is 'DSQUERY' ! }
   ...,
   dbclose(_dbproc).
Yes
```

dbsetnull/5

dbsetnull(_ReturnCode, _Dbproc, _Bindtype, _Bindlen, _Bindval)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : atom/integer (depending on DBconvertstatus)
arg4 : ground: integer
arg5 : ground : atom

dbsetmaxprocs/1

dbsetmaxprocs(_ReturnCode, _MaxNbOfDbproc)

arg1 : any : atom/integer (depending on DBconvertstatus)
arg2 : ground : integer

Sets the maximum number of simultaneously open DBprocesses.

dbsetopt/4

dbsetopt(_Returncode, _Dbproc, _Option, _Parameter)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : atom/integer (depending on DBconvertstatus)
arg4 : ground: atom

Sets a DataServer or DB-LIBRARY option. *Arg4* has to be an atom.

Example

```
?- ..., dbsetopt(_dbproc,'DBBUFFER','500'), ...
```

The different options are:

DBPARSEONLY
DBSHOWPLAN
DBNOEXEC
DBIGNOFLOW
DBIGNDIVZERO
DBARITHABORT
DBARITHNULL
DBOFFSET
DBSTAT
DBSTORPROCID
DBROWCOUNT
DBBUFFER
DBNOAUTOFREE

dbsettime/2

dbsettime(_ReturnCode, _NbOfSeconds)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : integer

Sets the number of seconds for this DBprocess to wait for a DataServer response.

dbsetuserdata/2

dbsetuserdata(_Dbproc, _Data)

arg1: ground : pointer
arg2: ground : pointer

dbstrcpy/5

dbstrcpy(_ReturnCode, _Dbproc, _Start, _NumBytes, _Dest)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : integer
arg4 : ground : integer
arg4 : ground : pointer

dbsqlexec/2

dbsqlexec(_Returncode, _Dbproc)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer

Sends the contents of the command buffer (of the *arg2* DBPROCESS structure) to the DataServer for execution.

dbsqlok/2

dbsqlok(_Returncode, _Dbproc)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer

Verifies the correctness of a command batch.

dbsqlsend/2

dbsqlsend(_Returncode, _Dbproc)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer

Sends a command to the DataServer and does not wait for a response.

dbstrlen/2

dbstrlen(_Length, _Dbproc)

arg1 : free : integer
arg2 : ground : pointer

Returns the length, in characters, of the command buffer.

dbtabbrowse/3

dbtabbrowse(_Dbbool, _Dbproc, _Tabnum)

arg1: any : integer/atom
arg2: ground : pointer
arg3: ground : integer

dbtabcount/2

dbtabcount(_Ret, _Dbproc)

arg1: free : integer
arg2: ground : pointer

dbtabname/3

dbtabname(_Ret, _Dbproc, _Tabnum)

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

dbtabsource/4

dbtabsource(_Ret, _Dbproc, _Calnum, _Tabnum)

arg1: free : atom
arg2: ground : pointer
arg3: ground : integer
arg4: free : atom

dbtsnewlen/2

dbtsnewlen(_Ret, _Dbproc)

arg1: free : integer
arg2: ground : pointer

dbtsnewval/2

dbtsnewval(_Dbbinary, _Dbproc)

arg1: free : pointer
arg2: ground : pointer

dbtsput/6

dbtsput(_Retcode, _DBproc, _Newts, _Newtslen, _Tabnum, _Tabname)

arg1: any : integer/atom (depending on dbconvertstatus)
arg2: ground : pointer
arg3: ground : pointer
arg4: ground : integer
arg5: ground : integer
arg6: ground : atom

dbtxptr/3

dbtxptr(_Dbbinary, _Dbproc, _Column)

arg1: free : pointer
arg2: ground : pointer
arg3: ground : integer

dbtxtimestamp/3

dbtxtimestamp(_Dbbinary, _Dbproc, _Column)

arg1: free : pointer
arg2: ground : pointer
arg3: ground : integer

dbtxtsnewval/2

dbtxtsnewval(_Dbbinary, _Dbproc)

arg1: free : pointer
arg2: ground : pointer

dbtxtsput/4

dbtxtsput(_Retcode, _Dbproc, _Newtxts, _Calnum)

arg1: any : integer/atom (depending on DBconvertstatus)
arg2: ground : pointer
arg3: ground : pointer
arg4: ground : integer

dbuse/3

dbuse(_Returncode, _Dbproc, _DbName)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : pointer
arg3 : ground : atom

Uses a particular database.

dbvarylen/3

dbvarylen(_Dbbool, _Dbproc, _Column)

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

dbwillconvert/3

dbwillconvert(_Returncode, _SrcType, _DesType)

arg1 : any : atom//integer (depending on DBconvertstatus)
arg2 : ground : integer/atom (depending on DBconvertstatus)
arg3 : ground : integer/atom (depending on DBconvertstatus)

Determines wether a specific datatype conversion is available with DB-LIBRARY

dbwritepage/6

dbwritepage(_RetCode, _Dbproc, _Dbname, _Pageno, _Size, _Buf)

arg1: any : integer/atom
arg2: ground : pointer
arg3: ground : atom
arg4: free : pointer
arg5: free : integer
arg6: ground : pointerr

dbwritetext/9

dbwritetext(_RetCode, _Dbproc, _Objname, _Textptr, _Textptrlen, _Timestamp, _Log, _Size, _Text).

arg1: free : integer/atom
arg2: ground : pointer
arg3: ground : atom
arg4: free : pointer
arg5: free : integer
arg6: ground : pointer
arg7: ground : integer/atom
arg8: free : integer
arg9: ground : pointer

Prolog And Unix Libraries

Prolog and Unix Libraries

Chapter 1
Prolog

This directory contains a number of subdirectories, each of which contains various predicates for manipulating data structures. To load these files, use the predicate:

```
?- lib(<filename>) .
```

or on the command line:

```
csh% BIMprolog -Lprolog/<filename>
```

Note:

The following *autonumbered titles* refer to the **UNIX library directory.**
Subtitles are the name of the **files** to be loaded with the ?- lib (<file name>) goal.

1.1 lists

Contains a number of files that define most of the standard routines for manipulating various kinds of lists.

difference_lists

Definitions for manipulating ***difference lists***. The main advantage of this approach to using lists is that the concatenation of lists can be done in unit time. This greatly improves the efficiency of programs that heavily require the manipulation of lists.

diff_create/1

diff_create(_DiffList)

arg1 : free : difference list

Creates a difference list *arg1* having the form _H - _T.

diff_concat/3

diff_concat(_DiffList1, _DiffList2, _DiffList3)

arg1 : partial : difference list
arg2 : partial : difference list
arg3 : free : difference list

This concatenates two difference lists, *arg1* and *arg2*, producing a new difference list *arg3*.

diff_append/3

diff_append(_Element, _DiffList1, _NewDiffList)

arg1 : free : term
arg2 : partial : difference list
arg3 : free : difference list

This appends a new element *arg1* to the end of difference list *arg2*, producing a new difference list *arg3*.

diff_remove_front/3

diff_remove_front(_DiffList1, _Head, _NewDiffList)

arg1 : partial : difference list
arg2 : free : term
arg3 : free : difference list

Removes the front element *arg2* of difference list *arg1*, creating a new difference list in *arg3*.

diff_close/1

diff_close(_DiffList1)

arg1 : partial : difference list

Closes the list *arg1*, so that normal list operations will function correctly.

diff_convert/2

diff_convert(_List, _DiffList)

arg1 : partial : list
arg2 : free : difference list

Converts the list *arg1* from a normal list into a difference list in *arg2*.

diff_member/2

diff_member(_Term, _DiffList)

arg1 : any : term
arg2 : partial : difference list

A difference list version of the member function. *Arg1* is a member of *arg2*.

diff_not_member/2

diff_not_member(_Term, _DiffList)

arg1 : any : term
arg2 : partial : difference list

A difference list version of the member function that fails if the element *arg1* is in the list *arg2*.

diff_length/2

diff_length(_DiffList, _Length)

arg1 : partial : difference list
arg2 : any : integer

Arg2 returns the length of the list *arg1* (not including the tail variable of the list).

diff_list/1

diff_list(_DiffList)

arg1 : partial : difference list

Succeeds if *arg1* is in difference list form.

diff_elements/2

diff_elements(_Elem, _DiffList)

arg1 : free : term
arg2 : partial : difference list

Returns in *arg1* successive elements on backtracking of a difference list *arg2* in a manner similar to the normal **member/2** predicate.

diff_empty/1

diff_empty(_DiffList)

arg1 : any : difference list

Succeeds if the difference list *arg1* is empty, otherwise it fails.

diff_not_empty/1

diff_not_empty(_DiffList)

arg1 : any : difference list

Succeeds if the difference list *arg1* is not empty, otherwise it fails.

Example

Create a difference list structure.

```
?- diff_create(_L).
 _L = _32 - _32
Yes
```

Append element 1 to the created difference list.

```
?- diff_create(_L), diff_append(1, _L, _NL).
_L = [1 | _61] - [1 | _61]
_NL = [1 | _61] - _61
Yes
```

Create a second difference list.

```
?- diff_create(_L2), diff_append(2, _L2, _NL).
_L2 = [2 | _61] - [2 | _61]
_NL = [2 | _61] - _61
Yes
```

Combining both previous queries to append the second difference list to the first difference list.

```
?- diff_create(_L), diff_append(1, _L, _NL),
   diff_create(_L2), diff_append(2, _L2, _NL2),
   diff_concat(_NL, _NL2, _Result).
_L = [1,2 | _129] - [1,2 | _129]
_NL = [1,2 | _129] - [2 | _129]
_L2 = [2 | _129] - [2 | _129]
_NL2 = [2 | _129] - _129
_Result = [1,2 | _129] - _129
Yes
```

_Result is the new list.

lazy

A **lazy list** is a pair consisting of a normal Prolog list (but usually ending with an unbound variable) and a goal which may be used to generate new elements. The idea is that [X0,X1,X2,...] /**R** should satisfy the goals X0 **R** X1, X1 **R** X2, and so on. These objects should only be used as arguments to the lazy_list predicates.

make_lazy/3

make_lazy(_First, _Pred, _Lazy_List)

arg1 : partial : term
arg2 : ground : atom
arg3 : any : lazy list

Creates a lazy list *arg3* with first element *arg1*. *Arg2* is the functor name of the goal with arity 2, which generates new elements.

head_lazy/2

head_lazy(_Lazy_List, _Head)

arg1 : partial : lazy list
arg2 : any : term

Returns in *arg2* the head of the lazy list *arg1*.

tail_lazy/2

tail_lazy(_Lazy_List, _Tail_Lazy_List)

arg1 : partial : lazy list
arg2 : any : lazy list

Returns in *arg2* the tail of the lazy list *arg1*.

member_check_lazy/2

member_check_lazy(_Thing, _LazyList)

arg1 : partial : term
arg2 : partial : lazy list

Checks if *arg2* is a number of the lazy list *arg1*.

Example

```
?- assert( (succ(_Head,_Next) :- _Next is _Head + 1 ) ).
Yes
?- make_lazy(0,succ,_List),
   head_lazy(_List,_H),
   tail_lazy(_List,_T).
 _List=[0 | _1]/succ
 _H=[0]
 _T=[1|_3]/succ
Yes
```

listut

Lists processing utilities. This directory contains most of the standard Prolog definitions that can be used to perform membership tests and to append lists. More information on the behavior of the predicates can be found in the sources. The following predicates are defined:

append/3

append(_L1, _L2, _L3)

arg1 : any : list
arg2 : any : list
arg3 : any : list

This is the standard append function. The use of this function is not encouraged, because difference lists are much more efficient. *Arg3* is *arg2* appended to *arg1*.

correspond/4

correspond(_X, _Xlist, _Ylist, _Y)

arg1 : any : term
arg2 : ground : list
arg3 : partial : list
arg4 : any : term

Succeeds when *arg2* and *arg3* are lists, *arg1* is an element of *arg2*, *arg4* is an element of *arg3*, and *arg1* and *arg4* are in similar places in their lists.

delete/3

delete(_List, _Elem, _Residue)

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

Arg3 is instantiated to the list *arg1* in which all elements equal to *arg2* are deleted.

last/2

last(_Last, _List)

arg1 : any : term
arg2 : partial : list

Arg1 is unified with the last element of list *arg2*.

nextto/3

nextto(_X, _Y, _List)

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

Succeeds when *arg1* and *arg2* appear side by side in list *arg3*.

nmember/3

nmember(_Elem, _List, _Index)

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

Succeeds when *arg1* is the *arg3*'th member of *arg2*. It may be used to select a particular element, or the elements and indices together.

nmembers/3

nmembers(_Indices, _List, _ListOfElements)

arg1 : partial : list
arg2 : partial : list
arg3 : partial : list

Like **nmember/3** except that it looks for a list of arguments. *Arg1* and *arg3* must not both be free.

Example

```
?- nmembers([3,5,1], [a,b,c,d,e,f,g,h], [c,e,a])
Yes
```

nth0/3

nth0(_N, _List, _Elem)

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

Succeeds when *arg3* is the *arg1*'th member of *arg2*, counting the first as element 0. (That is, discards the first *arg1* elements and unifies *arg3* with the next element.)

nth0/4

nth0(_N, _List, _Elem, _Rest)

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

Unifies *arg3* with the *arg1*'th element of *arg2*, counting from 0, and *arg4* with the other elements. It can be used to select the *arg1*'th element of *arg2* (yielding *arg3* and *arg4*), or to insert *arg3* after the *arg1*'th (counting from 1) element of *arg4*, when it yields *arg2*.

Example

```
?- nth0(2, _List, c, [a,b,d,e])
 _List = [a,b,c,d,e].
Yes
```

nth1/3

nth1(_N, _List, _Elem)

arg1 : ground : integer
arg2 : partial : list
arg3 : any : term or free

Is the same as **nth0/3** except that it counts from 1.

nth1/4

nth1 (_N, _List, _Elem, _Rest)

arg1 : ground : integer
arg2 : any : list or free
arg3 : any : term or free
arg4 : any : list or free

Nth1/4 is the same as **nth0/4** except that it counts from 1. **Nth1/4** can be used to insert element *arg3* before the *arg1*th element of the list *arg4*.

numlist/3

numlist(_Lower, _Upper, _List)

arg1 : ground : integer
arg2 : ground : integer
arg3 : any.: list

Succeeds when *arg3* is [_Lower, _Lower+1, ..., _Upper-1, _Upper].

Arg1 and *arg2* must be integers, not expressions. If *arg2* < *arg1* the predicate fails rather than producing an empty list.

perm/2

perm(_List, _Perm)

arg1 : partial : list
arg2 : any : list

Succeeds when *arg1 and arg2* are permutations of each other.

Note:

The purpose of perm is to generate permutations.

When simply testing whether a list is a permutation of another, it is more efficient to use **keysort/2** or **list_to_bag**.

Warning:

Note that the number of permutations of an N-element list is N!, that is 5040 possible permutations for a seven-element list.

perm2/4

perm2(_A, _B, _C, _D)

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

Succeeds when the set {*arg1*, *arg2*} equals the set {*arg3*, *arg4*}. This predicate is defined as:

```
perm2(_A, _B, _C, _D) :-
     perm([_A, _B], [_C, _D]).
```

This predicate allows the user to write pattern matches on commutative operators.

remove_dups/2

remove_dups(_List, _Pruned)

arg1 : partial : list
arg2 : any : list

Removes duplicate elements from *arg1*.

If the list *arg1* contains non-ground elements, the result is unpredictable.

reverse/2

reverse(_List, _Reversed)

arg1 : partial : list
arg2 : any : list

The order of the elements in list *arg1* is reversed and the result is unified with *arg2*.

rev/2.

rev/2 is a synonym for **reverse/2**.

same_length/2

same_length(_List1, _List2)

arg1 : any : list
arg2 : any : list

Succeeds when *arg1* and *arg2* are both lists and have the same number of elements.

If both arguments are free, **same_length/2** will generate two lists of the same length; that is, of length 0,1,2, ... by backtracking.

If one of the arguments is free, the other one will be instantiated to a list of the same length.

select/4

select(_X, _Xlist, _Y, _Ylist)

arg1 : any : term
arg2 : partial : list
arg3 : any : term
arg4 : partial : list

Succeeds when *arg2* and *arg4* are lists of the same length, only differing by one element at the same position in each list. *Arg1* and *arg3* are the differing elements of list *arg2* and list *arg4*, respectively.

shorter_list/2

shorter_list(_Short, _Long)

arg1 : any : list
arg2 : partial : list

Succeeds when *arg1* is a list strictly shorter than *arg2*. *Arg2* does not have to be a proper list, provided that it is long enough. This can be used to generate lists shorter than *arg2*, lengths 0, 1, 2...n will be tried, but backtracking will terminate with a list that is one element shorter than *arg2*. It cannot be used to generate lists longer than *arg1*.

subseq/3

subseq(_Sequence, _SubSequence, _Complement)

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

Succeeds when *arg2* and *arg3* are both sub-sequences of the list *arg1* (the order of corresponding elements being preserved). Every element of *arg1* which is not in *arg2* must be in *arg3* and vice versa.

The following relation holds:

length(_Sequence) = length(_SubSequence) + length(_Complement),

This was written to generate subsets and their complements together, but can also be used to combine two lists in all possible ways.

Example

```
?- subseq([1,2,3,4], [1,3,4], [2]).
Yes
```

subseq0/2

subseq0(_Sequence, _SubSequence)

arg1 : partial : list
arg2 : any : list

Succeeds when *arg2* is a sub-sequence of *arg1*. *Arg2* and *arg1* may be equivalent.

Example

```
?- subseq0([a,b], [a]).
Yes
?- subseq0([a,b], [a,b]).
Yes
```

subseq1/2

subseq1(_Sequence, _SubSequence)

arg1 : partial : list
arg2 : any : list

Succeeds when *arg2* is a sub-sequence of *arg1* containing at least one element less.

sumlist/2

sumlist(_Numbers, _Total)

arg1 : ground : list of integers
arg2 : any : integer

Succeeds when *arg1* is a list of integers, and *arg2* is their sum.

order

Defines the "ordered" predicates.

ordered/1

ordered(_List)

arg1 : partial : list

Succeeds when *arg1* is a list of terms [T1,T2,...,Tn] so that
for all i in 2..n Ti-1 @=< Ti.

ordered/2

ordered(_Function, _List)

arg1 : ground : term
arg2 : partial : list

Ordered(P, [T1,T2,...,Tn]) succeeds if the following conjunction of goals succeeds: P(T1,T2) & P(T2,T3) & ...

This predicate can be used to generate sequences.

Example

```
?- _L = [1,_,_,_,_], ordered(times(2),_L) .
 _L = [1,2,4,8,16].
Yes
```

len_/2

len_(_Length, _List)

arg1 : ground : integer
arg2 : free : list

Generates a list of a given length.

assoc_lists

This file contains a binary tree implementation of "association lists". This stores association pairs in a tree structure; the empty tree is "t".

Note:
The keys must be ground, but the associated values do not need to be ground.

put_assoc/4

put_assoc(_Key, _Old_Tree, _Value, _New_Tree)

arg1 : ground : term
arg2 : partial : tree
arg3 : any : term
arg4 : any : tree

Create a new assoc_list by replacing an existing pair by a new one.

Example

To store the pair "foo-moo" in an empty tree:

```
put_assoc(foo,t,moo,_T)
```

To add to tree _T the pair "bee-flea" giving the tree _U:

```
put_assoc(bee,_T,flea,_U)
```

get_assoc/3

get_assoc(_Key, _Tree, _Val)

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

If *arg3* is free, it will be instantiated to the value associated with *arg1* in the assoc_list *arg2*.

If *arg1* is free, **get_assoc/3** will find each *arg1-arg3* pair by backtracking.

assoc_to_list/2

assoc_to_list(_Assoc, _List)

arg1 : any : tree
arg2 : any : list

If *arg2* is free, then *arg1* converts the assoc_list to a list of the form [_Key1-_Val1, _Key2-_Val2...]. If *arg1* is free, **assoc_to_list/2** produces the shortest possible *arg1* assoc_list.

map_assoc/3

map_assoc(_Pred, _In_tree, _Out_tree)

arg1 : ground : atom
arg2 : partial : tree
arg3 : any : tree

Calls *arg1*(_X,_Y) for each _X on *arg2*, and constructs with _Y the *arg3* assoc_list.

lists

This file contains general list manipulation predicates.

is_list/1

is_list(_List)

arg1 : any : list

Succeeds if *arg1* is a list.

non_empty_list/1

non_empty_list(_List)

arg1 : any : list

Succeeds if *arg1* is a non-empty list.

empty_list/1

empty_list(_List)

arg1 : any : list

Succeeds if *arg1* is the empty list (nil).

in_list/2

in_list(_Var, _List)

arg1 : any : term
arg2 : partial : list

Succeeds if *arg1* is a member of the list *arg2*.

not_in_list/2

not_in_list(_Obj, _List)

arg1 : any : term
arg2 : partial : list

Succeeds if *arg1* is not in the list *arg2*.

element/2

element(_Element, _List)

arg1 : free : term
arg2 : partial : list

Arg1 is instantiated to the value of the elements of the list *arg2* by backtracking.

element_number/3

element_number(_List, _Element, _Count)

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

Arg1 is instantiated by backtracking to the value of the elements of the list *arg2*, and *arg3* to the corresponding position of the element.

list_max/2

list_max(_List, _Val)

arg1 : partial : list
arg2 : free : number

Arg2 returns the maximum (integer or real) value in the list *arg1*.

list_min/2

list_min(_List, _Val)

arg1 : partial : list
arg2 : free : number

Arg2 returns the minimal (integer or real) value in the list *arg1*.

front/3

front(_Element, _List, _Rlist)

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

Simply adds *arg1* to the front of the list *arg2* to form *arg3*.

length/2

length(_List, _Length)

arg1 : partial : list
arg2 : any : integer

Arg2 returns the length of the list *arg1*.

rremove/3

rremove(_Element, _List, _Tail)

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

Instantiates *arg1* to the first element of list *arg2* and unifies the rest of the list with *arg3*.

multil

Multiple-list routines.

mlmaplist/2

mlmaplist(_Pred, _Lists)

arg1 : ground : atom
arg2 : partial : multiple-list

Applies *arg1* to argument tuples which are successive slices of the multiple-list *arg2*.

For example:

```
?- mlmaplist(tidy, [_Untidy, _Tidied]) .
```

Applies tidy([_U, _T]) to each successive [_U, _T] pair from _Untidy and _Tidied.

mlmaplist/3

mlmaplist(_Pred, _Lists, _Extra)

arg1 : ground : atom
arg2 : partial : multiple-list
arg3 : any : term

Is similar to **mlmaplist/2,** but passes the *arg3* argument to *arg1*, as well as the slices from the multiple-list *arg2*.

mlmaplist/4

mlmaplist(_Pred, _Lists, _Init, _Final)

arg1 : ground : atom
arg2 : partial : multiple-list
arg3 : any : term
arg4 : any : term

Is similar to **mlmaplist/2,** but has an extra accumulator feature. *Arg3* is the initial value of the accumulator, and *arg4* is the final result. *Arg1* (Slice, AccIn, AccOut) is invoked to update the accumulator.

mlmember/2

mlmember(_Elems, _Lists)

arg1 : any : term
arg2 : partial : multiple-list

Is the multi-list analog of **member/2**.

Example

```
?- ml_member([a,d,g], [[a,b,c],[d,e,f],[g,h,i]]) .
Yes
?- ml_member(_X, [[a,b,c],[d,e,f],[g,h,i]]) .
 _X = [a,d,g]
Yes ;
 _X = [b,e,h]
Yes ;
 _X = [c,f,i]
Yes ;
No
```

projec

Routines to select the nth argument of each element of a list.

keys_and_values/3

keys_and_values([_K1-_V1,...,_Kn-_Vn], [_K1,...,_Kn], [_V1,...,_Vn])

arg1 : any : list
arg2 : any : list
arg3 : any : list

Succeeds when *arg1* is a list of key-value pairs, *arg2* the corresponding list of keys and *arg3* the list of respective values. It is meant for splitting a list of key-value pairs (such as **keysort/2**) into separate lists of keys and of values. It may be used for building a list of pairs from a pair of lists.

Example

To sort a list without having duplicates removed:

```
?- keys_and_values(_RawPairs, _RawKeys, _),
   keysort(_RawPairs, _OrdPairs),
   keys_and_values(_OrdPairs, _OrdKeys, _).
```

project/3

project(_ListOfTerms, _ArgNr, _ListOfArgs)

arg1 : partial : list of terms
arg2 : ground : integer
arg3 : any : list of terms

The *arg2*th elements of the terms in the list *arg1* are collected in a list that is unified with *arg3*. When *arg2* is 0, the functor names of the terms are considered.

Example

```
?- project ( [s (1,2,3), t (4,5,6), u (a,b,c) ], 3, _res).
 _res = [3, 6, c]
Yes
```

1.2 symbols

Contains programs for manipulating symbols.

gensym

gensym/2

gensym(_Template, _Symbol)

arg1 : ground : atom
arg2 : free : atom

A program for generating unique symbols. The user supplies a template which will be used as the starting point for generating the new atoms. When a new template is supplied, the predicate starts a new numbering sequence for this template but memorizes the counters of the previous templates.

Example

```
?- gensym(ta, _A), gensym(ta, _B),
   gensym(tb, _C), gensym(ta, _C).
 _A = ta0
 _B = ta1
 _C = tb0
 _D = ta2
Yes
```

maxatomlength

maxatomlength/2

maxatomlength(_List, _Length)

arg1 : partial : list
arg2 : free : integer

Returns the length of the longest atom in the list or -1 if the list contains no atoms.

Example

```
?- maxatomlength([a,ab,abc,ab,a], _L).
 _L = 3
Yes
```

1.3 global

Consists of programs to manipulate global data values, using the ***ProLog by BIM*** record database. The predicates can be subdivided into two groups. The first manages sets of objects, and the second provides a global single value variable.

obj_set

Predicates that manipulate "**sets**" of objects in the record database of ***ProLog by BIM***. The user may add to, delete from or test these "**sets**" without having to perform the associated record, erase, and so on. The user only has to know the "set identifier".

scrub_obj_set/1

scrub_obj_set(_Set_Name)

arg1 : ground : atom

Cleans the recorded object set *arg1*.

obj_set_get/2

obj_set_get(_Set_Name, _Objs)

arg1 : ground : atom
arg2 : free : list

Arg2 returns the elements of the object set *arg1*. The set is returned as a list that is not in any particular order.

obj_set_insert

obj_set_insert(_Set_Name, _Obj)

arg1 : ground : atom
arg2 : partial : term

Inserts *arg2* into the set of recorded objects *arg1*. This will only insert the object once, and duplicates will not be inserted. A duplicate is a term which can be unified with the object.

obj_set_delete/2

obj_set_delete(_Set_Name, _Obj)

arg1 : ground : atom
arg2 : any : list

Deletes *arg2* from the object set *arg1*.

in_obj_set/2

in_obj_set(_Set_Name, _Obj)

arg1 : ground : atom
arg2 : partial : term

Tests whether *arg1* is already in the object set *arg1*.

not_in_obj_set/2

not_in_obj_set(_Set_Name, _Obj)

arg1 : ground : atom
arg2 : partial : term

Tests whether *arg2* is not yet in the object set *arg1*. Fails if it is in the set. Currently, the object tests are based on whether the objects are unifiable.

obj_equal/2

obj_equal(_Obj1, _Obj2)

arg1 : any : term
arg2 : any : term

Tests whether the objects *arg1* and *arg2* are equal. Currently, the test is based on whether the objects are unifiable.

obj_not_equal/2

obj_not_equal(_Obj1, _Obj2)

arg1 : any : term
arg2 : any : term

Tests whether the objects *arg1* and *arg2* are not equal. Currently, the test is based on whether the objects are unifiable.

Example

Insert object 1 into the set identified as set1.

```
?- obj_set_insert(set1, 1).
Yes
```

Add element 2 to the same set.

```
?- obj_set_insert(set1, 2).
Yes
```

Insert element 1 into the set. This succeeds but it is already a member of the object set. It is not inserted a second time.

```
?- obj_set_insert(set1, 1).
Yes
```

Delete object 2 from the set identified as set1.

```
?- obj_set_delete(set1, 2).
Yes
```

See if the object is a member of the set.

```
?- in_obj_set(set1, 2).
No
```

?- in_obj_set(set1, 1).
Yes

Retrieve the whole set as a list.

```
?- obj_set_get(set1, _Objs).
     _Objs = [1]
Yes
```

destructive

These effectively simulate global variables, which may be created, destroyed, incremented or decremented.

global_new_id/1

global_new_id(_New_Id)

arg1 : free : atom

Arg1 returns a new global variable identifier.

global_set/2

global_set(_Id, _Value)

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

Sets global variable *arg1* to a certain value *arg2*.

global_get/2

global_get(_Id, _Value)

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

Arg2 returns the value of global variable *arg1*.

global_inc/1

global_inc(_Id)

arg1 : ground : atom

Increments by one the global integer variable *arg1*.

global_dec/1

global_dec(_Id)

arg1 : ground : atom

Decrements by one the global variable *arg1*.

global_list/1

global_list(_List)

arg1 : free : list

Arg1 returns a list containing all the global variable identifiers that have been created.

global_destroy/1

global_destroy(_Id)

arg1 : ground : atom

Erases the global variable *arg1*.

1.4 sets

Manipulates sets of objects. A set is a data structure that contains no duplicate element.

setutl

The elements in the sets are not ordered.

member/2

member(_Elem, _List)

arg1 : any : term
arg2 : partial : list

Standard membership predicate. Succeeds if *arg1* is in *arg2*. If *arg1* is free, the successive elements will be returned by backtracking.

memberchk/2

memberchk(_Elem, _List)

arg1 : ground : term
arg2 : partial : list

Almost the same as the **member/2** predicate above, except only a known element is tested for.

nonmember/2

nonmember(_Elem, _List)

arg1 : ground: term
arg2 : partial : list

Succeeds if *arg1* does not occur in *arg2*.

pairfrom/4

pairfrom(_Set, _Elem1, _Elem2, _Residue)

arg1 : partial : list
arg2 : any : term
arg3 : any : term
arg4 : any : list

Succeeds when *arg1* is a list. *Arg2* is in *arg1* and *arg3* appears in the list after *arg2*. *Arg4* is the list *arg1* minus the two elements.

select/3

select(_Elem, _List, _Rem)

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

Succeeds when *arg1* occurs in *arg2*. *Arg3* is the list minus *arg1*.

add_element/3

add_element(_Elem, _Set1, _Set2)

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

Succeeds when *arg2* and *arg3* are sets represented as unordered lists and *arg3* is the union of set *arg2* and singleton {*arg1*}.

del_element/3

del_element(_Elem, _List, _Rem)

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

Removes *arg1* from *arg2* and places the remainder in *arg3*.

disjoint/1

disjoint(_Set)

arg1 : partial : list

Succeeds if *arg1* contains no duplicate element.

disjoint/2

disjoint(_Set1, _Set2)

arg1 : partial : list
arg2 : partial : list

Succeeds if *arg1* and *arg2* have no element in common.

intersect/2

intersect(_Set1, _Set2)

arg1 : partial : list
arg2 : partial : list

Succeeds if *arg1* and *arg2* have an element in common.

subset/2

subset(_Set1, _Set2)

arg1 : partial : list
arg2 : partial : list

Succeeds if each element of *arg1* is in *arg2*.

seteq/2

seteq(_Set1, _Set2)

arg1 : partial : list
arg2 : partial : list

Succeeds if *arg1* and *arg2* are subsets of each other.

listtoset/2

listtoset(_List, _Set)

arg1 : partial : list
arg2 : any : list

Succeeds when *arg1* and *arg2* are lists and *arg2* contains the same elements as in *arg1*, but with no duplicate.

intersect/3

intersect(_Set1, _Set2, _Inter)

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

Arg3 is instantiated to the intersection of set *arg1* and set *arg2*.

subtract/3

subtract(_Set1, _Set2, _Diff)

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

Succeeds if *arg3* is the elements of set *arg1* that are not in set *arg2*.

symdiff/3

symdiff(_Set1, _Set2, _Diff)

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

Succeeds if *arg3* is the symmetric difference of set *arg1* and set *arg2*. Each element occurs in *arg1* or *arg2* but not in both.

union/3

union(_Set1, _Set2, _Union)

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

Succeeds when *arg3* contains the elements of *arg1* that do not occur in *arg2*, followed by the elements in *arg2*.

ordset

Manipulates **ordered sets**. The ordering is defined by the @< family of term comparison predicates, which is also the ordering used by **sort/2** and **setof/3**.

The benefit of the ordered representation is that the elementary set operations can be done in time proportional to the sum of the argument sizes rather than their product.

list_to_ord_set/2

list_to_ord_set(_List, _Set)

arg1 : partial : list
arg2 : any : list

Succeeds when *arg2* is the ordered representation of the unordered set *arg1*.

merge/3

merge(_List1, _List2, _Merged)

arg1 : partial : list
arg2 : partial : list
arg3 : free: list

Arg3 is instantiated to the merge of the two given ordered sets *arg1* and *arg2*. When the input sets contain duplicates, each copy of an element is preserved in the output (see also **ord_union/3**).

Example

```
?- merge[1,2,5,7], [3,5,7], _ x] .
 _x = [1,2,3,5,5,7,7]
Yes
```

ord_disjoint/2

ord_disjoint(_Set1, _Set2)

arg1 : partial : list
arg2 : partial : list

Succeeds when the two ordered sets have no element in common.

ord_insert/3

ord_insert(_Set1, _Element, _Set2)

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

The ordered set *arg3* is made by inserting element *arg2* in the ordered set *arg1*. If *arg2* is already a member of *arg1*, it is not added.

ord_intersect/2

ord_intersect(_Set1, _Set2)

arg1 : partial : list
arg2 : partial : list

Succeeds when the two ordered sets have at least one element in common.

ord_intersect/3

ord_intersect(_Set1, _Set2, _Intersection)

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

Arg3 is unified with the ordered representation of the intersection of *arg1* and *arg2*, provided that *arg1* and *arg2* are ordered sets.

ord_seteq/2

ord_seteq(_Set1, _Set2)

arg1 : partial : list
arg2 : partial : list

Succeeds when the two arguments represent the same set.

ord_subset/2

ord_subset(_Set1, _Set2)

arg1 : partial : list
arg2 : partial : list

Succeeds when every element of the ordered set *arg1* appears in the ordered set *arg2*.

ord_subtract/3

ord_subtract(_Set1, _Set2, _Difference)

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

Succeeds when *arg3* contains the elements of *arg1* that are not in *arg2*.

ord_symdiff/3

ord_symdiff(_Set1, _Set2, _Difference)

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

Succeeds when *arg3* is the symmetric difference of ordered set *arg1* and ordered set *arg2*.

ord_union/3

ord_union(_Set1, _Set2, _Union)

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

Succeeds when *arg3* is the union of ordered set *arg1* and ordered set *arg2*. When an element occurs in both sets, only one element is retained.

intsets

This provides a conceptual interface to the bitstrings package.

intset_create/1

inset_create(_IntSet)

arg1 : free : inset

Creates an emty integer set.

intset_member/2

inset_member(Int, _IntSet)

arg1 : ground : integer
arg2 : ground : intset

Succeeds if *arg1* is a member of the integer set *arg2*; fails otherwise.

intset_add/3

inset_add(Int, _IntSet, NewIntSet)

arg1 : ground : integer
arg2 : ground : intset
arg3 : free : intset

Adds integer *arg1* to the intset *arg2* to produce a new intset *arg3*.

intset_del/3

inset_del(Int, _IntSet, NewIntSet)

arg1 : ground : integer
arg2 : ground : intset
arg3 : free : intset

Removes integer *arg1* from the intset *arg2* to produce a new intset *arg3*.

intset_union/3

inset_union(IntSet1, _IntSet2, NewIntSet)

arg1 : ground : intset
arg2 : ground : intset
arg3 : free : intset

Returns as *arg3* the union of the two intsets given in *arg1* and *arg2*.

intset_intersection/3

inset_intersection(IntSet1, _IntSet2, NewIntSet)

arg1 : ground : intset
arg2 : ground : intset
arg3 : free : intset

Returns as *arg3* the intersection of the two intsets given in *arg1* and *arg2*.

intset_to_list/2

inset_to_list(_IntSet, _List)

arg1 : ground : intset
arg2 : free : list of integers

Converts intset *arg1* into a list of integers.

intset_from_list/2

inset_from_list(_List, _IntSet)

arg1 : ground : list of integers
arg2 : free : intset

Converts a list of integers into the intset representation used by these predicates.

1.5 trees

tree

Documentation on this library can be found in the corresponding source file.

1.6 queues

queues

A queue is an abstract data type. The representation of the queues here is different from the representation of the queues in the ***ProLog by BIM*** kernel.

queue_create/1

queue_create(_Queue)

arg1 : free : queue

Creates a new queue. The queue is defined as a term having one argument. The argument is a difference list.

queue_add/3

queue_add(_QL, _Elem, _NQL)

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

Adds to a queue *arg1* the object *arg2* producing a new queue *arg3*.

queue_remove/3

queue_remove(_QL, _Elem, _NQL)

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

Arg2 is unified with the front element of queue *arg1* and *arg3* is the rest of the queue.

queue_length/2

queue_length(_QL, _Length)

arg1 : partial : queue
arg2 : any : free or integer

Arg2 returns the length of the queue *arg1*.

1.7 arrays

Contains programs that manipulate arrays of objects from within Prolog.

arrays

With these predicates, an array is represented in the following notation {1,2,3} is shown initially as array([1|_1],[2|_2],[3|_3])+0 where the [1|_1] and so on are lists in which the most recent value for the element comes last, and the +0 is a zero update count.

list_to_array/2

list_to_array(_List, _Array)

arg1 : ground : list
arg2 : free : array

The list *arg1* is converted into the array *arg2*.

array_to_list/2

array_to_list(_Array, _List)

arg1 : partial : array
arg2 : any : list

Converts the array *arg1* into the list *arg2*.

array_length/2

array_length(_Array, _Length)

arg1 : partial : array
arg2 : free : integer

Arg2 is the length of *arg1*.

store/4

store(_Index, _OldArray, _Element, _NewArray)

arg1 : ground : integer
arg2 : partial : array
arg3 : any : term
arg4 : partial : array

Stores *arg3* in array *arg2* at index *arg1* and unifies the result with *arg4*.

fetch/3

fetch(_Index, _Array, _Element)

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

Arg3 is unified with the *arg1*th element of array *arg2*. If *arg1* is not in the range of the array, the predicate fails.

logarr

Extendable arrays accessible in logarithmic time. The following predicates are defined:

new_array/1

new_array(_A)

arg1 : free : logarray

Returns a new empty array in *arg1*.

is_array/1

is_array(_A)

arg1 : partial : logarray

Checks whether *arg1* is an array.

aref/3

aref(_Index, _Array, _Element)

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

Unifies *arg3* with *arg2*[*arg1*], or fails if *arg2*[*arg1*] has not been set.

arefa/3

arefa(_Index, _Array, _Element)

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

Is similar to **aref/3**, except that it unifies *arg3* with a new array if *arg2*[*arg1*] is undefined. This is useful for multi-dimensional arrays implemented as arrays of arrays.

arefl/3

arefl(_Index, _Array, _Element)

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

Is similar to **aref/3**, except that *arg3* appears as "[]" for undefined cells.

array_to_list/2

array_to_list(_Array, _List)

arg1 : partial : logarray
arg2 : free : list

Returns a list of pairs Index-Element of all the elements of *arg1* that have a value.

aset/4

aset(_Index, _Array, _Element, _NewArray)

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

Unifies *arg4* with the new array in which *arg2*[*arg1*] has been set to *arg3*.

1.8 maps

Implements and manipulates "**finite maps**".

map

A finite map is a function from terms to terms with a finite domain. This definition actually assumes that the domain consists of ground terms. The following predicates are defined.

is_map/1

is_map(_Map)

arg1 : partial (?) : term

Succeeds if *arg1* is a map.

list_to_map/2

list_to_map(_List, _Map)

arg1 : partial : list
arg2 : any : map

Takes a list *arg1* with elements in the form _Key-_Value and transforms it into a finite map *arg2*. _Keys may not be duplicated.

map_agree/2

map_agree(_Map1, _Map2)

arg1 : partial : map
arg2 : partial : map

Succeeds if *arg1* and *arg2* have a key and value in common, or if they have no keys.

map_compose/3

map_compose(_Map1, _Map2, _Composition)

arg1 : partial : map
arg2 : partial : map
arg3 : any : map

Composes *arg1* and *arg2* into composition.

map_disjoint/2

map_disjoint(_Map1, _Map2)

arg1 : partial : map
arg2 : partial : map

Succeeds if *arg1* and *arg2* have no domain elements in common.

map_domain/2

map_domain(_Map, _Domain)

arg1 : partial : map
arg2 : any : set

Unifies *arg2* with the ordered set representation of the domain of the finite map *arg1*.

map_exclude/3

map_exclude(_Map, _Set, _Restricted)

arg1 : partial : map
arg2 : partial : set
arg3 : any : map

Constructs a restriction of the map *arg1* by dropping members of the *arg2* from the *arg3* map's domain.

That is, *arg3* and *arg1* agree, but domain *(arg3)* = domain(*arg1*) \ *arg2*. *Arg2* must be an ordered set as defined in the ordset library file.

map_include/3

map_include(_Map, _Set, _Restricted)

arg1 : partial : map
arg2 : partial : set
arg3 : any : map

Constructs a restriction of the map *arg1* by dropping everything which is NOT a member of *arg2* from the restricted map's domain.

That is, the *arg3* and original *arg1* agree, but domain(*arg3*) = domain(*arg1*) ∩ *arg2*. *Arg2* must be an ordered set as defined in the ordset library file.

map_invert/2

map_invert(_Map, _Inverse)

arg1 : partial : map
arg2 : any : map

Unifies *arg2* with the inverse of a finite invertible *arg1*.

map_map/3

map_map(_Predicate, _Map1, _Map2)

arg1 : partial : (?)
arg2 : partial : map
arg3 : any : map

Composes *arg2* with the predicate, *arg1* so that K-V2 is in *arg3* if K-V1 is in *arg2* and *arg3*(V1,V2) is true.

map_range/2

map_range(_Map, _Range)

arg1 : partial : map
arg2 : any : set

Unifies *arg2* with the ordered set representation of the range of the finite map *arg1*.

Note:
The cardinality of the domain and the range are seldom equal, except of course for invertible maps.

map_to_assoc/2

map_to_assoc(_Map, _Assoc)

arg1 : partial : map
arg2 : any : assoc_list

Converts a map *arg1* from its compact form to an assoc_list *arg2*.

map_to_list/2

map_to_list(_Map, _KeyValList)

arg1 : partial : map
arg2 : any : list

Converts a map *arg1* from its compact form to a list *arg2* of Key-Val pairs.

map_union/3

map_union(_Map1, _Map2, _Union)

arg1 : partial : map
arg2 : partial : map
arg3 : any : map

Forms the union of the two given maps. That is*arg3*(X) = *arg1*(X) if it is defined, or *arg2*(X) if that is defined. But when both are defined, both must agree (See **map_update/3** for a version where *arg2* overrides *arg1*).

map_update/3

map_update(_Base, _Overlay, _Updated)

arg1 : partial : map
arg2 : partial : map
arg3 : any : map

Combines the finite maps *arg1* and *arg2* as map_union does, except that when both define values for the same key, the *arg2* value is taken regardless of the *arg1* value. This is useful for changing maps (also known as the "mu function").

map_update/4

map_update(_Map, _Key, _Val, _Updated)

arg1 : partial : map
arg2 : partial : term
arg3 : partial : term
arg4 : any : map

Computes an *arg4* map, which is the same as *arg1*, except that the image of *arg2* is *arg3*, rather than the image it had under *arg1*, if any.

map_value/3

map_value(_Map, _Arg, _Result)

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

Applies the finite map *arg1* to an argument *arg2*, and unifies *arg3* with the answer. It fails if *arg2* is not in the domain of *arg1*, or if the value does not unify with *arg3*.

portray_map/1

portray_map(_Map)

arg1 : partial : map

Writes the finite map *arg1* to the current output stream in a readable form.

Note:
A map written out this way cannot be read back. The point of this predicate is that you can add a clause "**portray(X) :- is_map(X), !, portray_map(X).**" to get maps displayed in a readable format by **print/1**.

1.9 bags

Manipulates "**bags**" in Prolog.

bagutl

A bag B is a function from a set dom(B) to the non-negative integers. See the source file for more information.

In this module bags are implemented using the following representation:

bag	is an empty bag
bag (E, M, B)	is bag B extended with a new element E, which occurs with multiplicity M and which is preceded by all elements of B in standard order.

is_bag/1

is_bag(_Bag)

arg1 : partial : term

This predicate succeeds when the term *arg1* is a bag as defined above.

bag_union/3

bag_union(_Bag1, _Bag2, _Union)

arg1 : partial : bag
arg2 : partial : bag
arg3 : any : bag

This predicate succeeds when the bag *arg3* is the union of the bags *arg1* and *arg2*.

bag_inter/3

bag_inter(_Bag1, _Bag2, _Intersection)

arg1 : partial : bag
arg2 : partial :bag
arg3 : any : bag

This predicate succeeds when the bag *arg3* is the intersection of the bags *arg1* and *arg2*.

bagmax/2

bagmax(_Bag, _Max)

arg1 : partial : bag
arg2 : any : term

This predicate unifies *arg2* with the element of bag *arg1*, which has the largest number of occurrences.

bagmin/2

bagmin(_Bag, _Min)

arg1 : partial : bag
arg2 : any : term

This predicate unifies *arg2* with the element of bag *arg1*, which has the smallest number of occurrences.

lengthbag/3

lenghtbag(_Bag, _Total, _Distinct)

arg1 : partial : bag
arg2 : any : integer
arg3 : any : integer

This predicate unifies the integers *arg2* and *arg3*, respectively, with the total number of elements, and number of distinct elements found in the bag *arg1*.

memberbag/3

memberbag(_E, _M, _Bag)

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

This predicate succeeds when the term *arg1* appears with multiplicity *arg2* in the bag *arg3*. This predicate can be used to test or generate members of the bag.

memberchkbag/3

memberchkbag(_E, _M, _Bag)

arg1 : partial : term
arg2 : partial : integer
arg3 : partial : bag

This predicate succeeds when the term *arg1* appears with multiplicity *arg2* in the bag *arg3*. This predicate cannot be used as a generator.

checkbag/2

check_bag(_Pred, _Bag)

arg1 : partial : term
arg2 : partial : bag

This predicate applies the predicate given as *arg1* to all elements of the bag *arg2*. For instance, if *arg1* is p(_X, _Y) and *arg2* is bag(_E, _M, _B) p(_X, _Y, _E, _M) will be called.

mapbag/3

mapbag(_Pred, _Bagin, _Bagout)

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

This predicate maps the predicate given in *arg1* to all elements of the bag *arg2* onto all elements of the bag *arg1* and builds a bag unified with *arg3* which contains the mapping results. For instance, if *arg1* is map(_X), and *arg2* is bag(_E, _M, _B), then *arg3* will contain bag(_Emapped, _M, _Bmapped) if map(_X, _E, _Emapped) succeeded.

make_sub_bag/2

make_sub_bag(_Bag, _SubBag)

arg1 : partial : bag
arg2 : free : bag

This predicate generates on backtracking all sub bags of the bag *arg1*.

test_sub_bag/2

test_sub_bag(_Sub, _Bag)

arg1 : partial : bag
arg2 : partial : bag

This predicate succeeds if the bag *arg1* is a sub bag of the bag *arg2*.

bag_to_list/2

bag_to_list(_Bag, _List)

arg1 : partial : bag
arg2 : any : list

This predicate transforms the bag *arg1* into a list containing _M occurrences of _E if bag (_E, _M,...) is in *arg1*, the resulting list is unified with *arg2*.

list_to_bag/2

list_to_bag(_List, _Bag)

arg1 : partial : list
arg2 : any : bag

This predicate makes a bag from the list *arg1* and unifies the result with *arg2*.

bag_to_set/2

bag_to_set(_Bag, _Set)

arg1 : partial : bag
arg2 : any : set

This predicate reduces the bag *arg1* to the set (represented as a list) of all elements of the bag. The resulting set is unified with *arg2*. The occurrences information contained in the bag is lost and thus there cannot be a reverse function.

portray_bag/1

portray_bag(_Bag)

arg1 : partial : term

This predicate prints a bag to the current output stream.

1.10 heaps

Implements "**heaps**" in Prolog. Heaps are collections of objects with no specific ordering. They may contain duplicates.

heaps

More information on how heaps are represented internally can be found in the source file.

list_to_heap/2

list_to_heap(_List, _Heap)

arg1 : partial : list
arg2 : free : heap

Takes a list *arg1* of Key-Value pairs (such as keysort could be used to sort) and converts it into a heap *arg2*.

heap_to_list/2

heap_to_list(_Heap, _List)

arg1 : partial : heap
arg2 : free : list

Returns the current set of Key-Value pairs in the heap *arg1* as a list *arg2*, sorted into ascending order of Keys.

heap_size/2

heap_size(_Heap, _Size)

arg1 : partial : heap
arg2 : any : integer

Succeeds when *arg2* is the number of elements in heap *arg1*.

add_to_heap/4

add_to_heap(_OldHeap, _Key, _Value, _NewHeap)

arg1 : partial : heap
arg2 : partial : term
arg3 : partial : term
arg4 : any : heap

Unifies *arg4* with the heap *arg1* to which the Key-Value pair *arg2-arg3* has been inserted.

Note:
The insertion is not stable; that is, if you insert several pairs with the same key, the order in which you retrieve them is not necessarily the order in which they were inserted. If you need a stable heap, you could add a counter to the heap and include the counter at the time of insertion in the key.

get_from_heap/4

get_from_heap(_OldHeap, _Key, _Value, _NewHeap)

arg1 : partial : heap
arg2 : free : term
arg3 : free : term
arg4 : free : heap

Returns the Key-Value pair (*arg2 - arg3*) from *arg1* with the smallest Key, and also a heap *arg4* which is the heap *arg1* with that pair deleted.

min_of_heap/3

min_of_heap(_Heap, _Key, _Value)

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

Unifies the pair *arg2-arg3* with the pair at the top of the heap *arg1* (which of course has the smallest Key). This predicate fails if the heap is empty.

min_of_heap/5

min_of_heap(_Heap, _Key1, _Value1, _Key2, _Value2)

arg1 : partial : heap
arg2 : any : term
arg3 : any : term
arg4 : any : term
arg5 : any : term

Returns the smallest (Key1) and second smallest (Key2) pairs in the heap, without deleting them. It fails if the heap does not have at least two elements.

1.11 graphs

Implements "**graphs**" in Prolog.

graphs

Graphs can have two representations.

The P-representation of a graph is a list of (from-to) vertex pairs, where the pairs can be in any order. This form is useful for input/output operations. The representation of a link in the Pform, A-B does not imply the existence of the link B-A. You have to put that in separately.

The S-representation of a graph is a list of (vertex-neighbors) pairs, where the pairs are in standard order (as produced by keysort) and the neighbors of each vertex are also in standard order (as produced by sort). This form is useful for many calculations.

In all the operations in this file, vertex labels are assumed to be ground terms, or at least to be sufficiently instantiated so that no two of them have a common instance.

vertices/2

vertices(_Graph, _Vertices)

arg1 : partial : S-graph
arg2 : any : list

Strips off the neighbors lists of an S-representation to produce a list *arg2* of the vertices of the graph *arg1* (It is a characteristic of S-representations that every vertex appears, even if it has no neighbors).

p_to_s_graph/2

p_to_s_graph(_Pform, _Sform)

arg1 : partial : P-graph
arg2 : any : S-graph

Converts a P- to an S-representation.

s_to_p_graph/2

s_to_p_graph(_Sform, _Pform)

arg1 : partial : S-graph
arg2 : any : P-graph

Converts an S- to a P-representation.

warshall/2

warshall(_Graph, _Closure)

arg1 : partial : S-graph
arg2 : any : S-graph

Takes the transitive closure of a graph in S-form.

Note:
This is not the reflexive transitive closure.

s_to_p_trans/2

s_to_p_trans(_Sform, _Pform)

arg1 : partial : S-graph
arg2 : any : P-graph

This predicate takes an S-graph in *arg1* and returns a P-graph corresponding to the transposition of *arg1*.

p_member/3

p_member(_X, _Y, _P_Graph)

arg1 : any : term
arg2 : any : term
arg3 : partial : P-graph

Succeeds when (*arg1,arg2*) is an existing arc in the graph whose P-form is given in *arg3*.

s_member/3

s_member(_X, _Y, _S_Graph)

arg1 : any : term
arg2 : any : term
arg3 : partial : S-graph

Succeeds when (*arg1,arg2*) is an existing arc in the graph whose S-form is given in *arg3*.

compose/3

compose(_G1, _G2, _Composition)

arg1 : partial : S-graph
arg2 : partial : S-graph
arg3 : any : S-graph

Returns in *arg3* the composition of two S-form graphs *arg1* and *arg2*, which need not have the same set of vertices.

p_transpose/2

p_transpose(_P_Graph, _Transposed)
arg1 : partial : P-graph
arg2 : any : P-graph

Arg2 is the transposition of the graph in P-form *arg1*.

s_transpose/2

s_transpose(_S_Graph, _Transposed)
arg1 : partial : S-graph
arg2 : any : S-graph

Arg2 is the transposition of the graph in S-form *arg1*.

1.12 terms

The files in this directory manipulate Prolog structures.

depth

Finds or checks the depth of a term. The following two predicates are defined.

depth_of_term/2

depth_of_term(_Term, _Depth)
arg1 : partial : term
arg2 : any : integer

Succeeds when *arg2* is the depth of the term *arg1*, where the depth is computed according to the following definition:

```
depth(Var) = 0
depth(Const) = 0
depth(F(T1,...,Tn)) = 1+max(depth(T1),...,depth(Tn))
```

depth_bound/2

depth_bound(_Term, _Bound)
arg1 : partial : term
arg2 : ground : integer ≥ 1

Succeeds when the term *arg1* has a depth smaller than the bound *arg2*.

struct

These routines view a term as a data structure. In particular, they handle Prolog variables in the terms as objects.

subst/3

subst(_Substitution, _Term, _Result)
arg1 : partial : term
arg2 : partial : term
arg3 : any : term

applies a substitution, where

<substitution> ::= <OldTerm> = <NewTerm> |
<Substitution> & <Substitution> |
<Substitution> # <Substitution>

The last two possibilities only make sense when the input *arg1* is an equation, and the substitution is a set of solutions. The "conjunction" of substitutions refers to back-substitution, and the order in which the substitutions are done may be crucial. If the substitution is ill-formed, **subst/3** fails.

occ/3

occ(_Subterm, _Term, _Times)

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

Returns in *arg2* the number of times that the subterm *arg1* occurs in the term *arg2*. The subterm must be ground.

variables/2

variables(_Term, _VarList)

arg1 : partial : term
arg2 : any : list

Returns in *arg2* a list whose elements are the variables occurring in term *arg1*, each appearing exactly once in the list. Is the same as the built-in **varlist/2**.

copy_ground/3

copy_ground(_Term, _Copy, _Substitution)

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

Makes a copy of the term *arg1* where variables are replaced by ground terms. *Arg3* is a substitution which can be applied (using **subst/3**) to the copy to recover the original.

flat

Flattens various binary trees to lists and converts back.

The <operator>_to_list predicates take a binary tree (where leaf nodes are anything not labelled by the operator) and flatten it to a list. They also omit "units" of that operator; that is, if the operator is "&" (respectively "|", "+" or "*"), the constant "true" (respectively "false", "0" or "1") will not appear in the list.

The list_to_<operator> predicates take a list and turn it back into a tree.

All those predicates are defined in terms of the next four predicates.

binary_to_list/5

binary_to_list(_Tree, _Operator, _Unit, _Before, _After)

arg1 : partial : binary tree
arg2 : ground : atom
arg3 : ground : term
arg4 : any : list
arg5 : any : list

This predicate transforms the binary tree *arg1* built with terms of the form *arg2* into a difference list *arg4-arg5* containing the leaves of the tree.

The unit for the operator (*arg2*) is specified in *arg3*. Units are removed from the result.

binary_to_list/4

binary_to_list (_True, _Operator, _Before, _After)

arg1 : partial : binary tree
arg2 : ground : atom
arg3 : any : list
arg4 : any : list

The same as **binary_to_list/5**, but for operators with no unit.

list_to_binary/4

list_to_binary(_List, _Operator, _Unit, _Tree)

arg1 : partial : list
arg2 : ground : atom
arg3 : ground : term
arg4 : any : binary tree

Succeeds when *arg4* is the binary tree built with the elements of the list *arg1* and the operator *arg2*.

If the list is empty, *arg4* is unified with *arg3* (the unit for the operator).

list_to_binary/3

list_to_binary(_List, _Operator, _Tree)

arg1 : partial : list
arg2 : partial : atom
arg3 : any : binary tree

The same as **list_to_binary/4** but for an operator which has no unit. If the list is empty, this predicate fails.

and_to_list/2

and_to_list(_Conjunction, _List)

arg1 : partial : term
arg2 : free : list

is implemented by:

```
and_to_list(_Conjunction, _List):-
     binary_to_list(_Conjunction, '&', true, _List, [])
```

list_to_and/2

list_to_and(_List, _Conjunction)

arg1 : partial : list
arg2 : free : term

is implemented by:

```
list_to_and(_List, _Conjunction):-
     list_to_binary(_List, '&', true, _Conjunction)
```

or_to_list/2, list_to_or/2, plus_to_list/2, list_to_plus/2, times_to_list/2, list_to_times/2

are similarly implemented.

flatten/2

flatten(_List, Flatlist)

arg1 : partial : list (of lists)
arg2 : free : list

is implemented by:

```
flatten(_List, _FlatList):-
     binary_to_list(_List, '.', [], _FlatList, [])
```

cyclic

Predicates that manipulate terms that might have cycles within them.

list_to_tree/2

list_to_tree(_List, _Tree)

arg1 : partial : list of pairs Key-Data
arg2 : free : tree

Returns in *arg2* a tree with next and previous pointers instead of nil elements. Terms returned by this predicate are called linked trees hereafter.

make_base_tree/2

make_base_tree(_List, _Tree)

arg1 : partial : list of pairs Key-Data
arg2 : free : an open ended tree

Returns in *arg2* the open-ended tree corresponding to the list *arg1*.

link_tree/1

link_tree(_Tree)

arg1 : partial : an open ended tree

Modifies the tree *arg1* with next and previous pointers instead of free vars.

tree_to_list_a/2

tree_to_list_a(_Tree, _AscendList)

arg1 : partial : a linked tree
arg2 : free : list

Returns in *arg2* an ascending list of the Key-Data pairs of the tree *arg1*.

tree_to_list_d/2

tree_to_list_d(_Tree, _DecendList)

arg1 : partial : a linked tree
arg2 : free : list

Returns in *arg2* a descending list of the Key-Data pairs of the tree *arg1*.

min/2

min(_Tree, _Node)

arg1 : partial : a linked tree
arg2 : free : term

Returns in *arg2* the smallest node in the tree *arg1*.

max/2

max(_Tree, _Node)

arg1 : partial : a linked tree
arg2 : free : term

Returns in *arg2* the largest node in the tree *arg1*.

previous/2

previous(_Node, _PrevNode)

arg1 : partial : some node of a linked tree
arg2 : free : term

Returns in *arg2* the predecessor of the linked tree node *arg1*.

next/2

next(_Node, _NextNode)

arg1 : partial : some node of a linked tree
arg2 : free : term

Returns in *arg2* the successor of the linked tree node *arg1*.

write_rdbl/1

write_rdbl(_Tree)

arg1 : partial : a linked tree

This predicate displays a linked tree in a more readable form.

same_functor

These predicates are used to perform equality tests between pairs of functors. They remove the need to know in advance which one of the arguments will be ground.

same_functor/4

same_functor(_Term1, _Term2, _Name, _Arity)

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

Succeeds if *arg1* and *arg2* have the same name *arg3* and arity *arg4*; fails otherwise.

same_functor/3

same_functor(_Term1, _Term2, _Arity)

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

Succeeds if *arg1* and *arg2* have the same arity *arg4*; fails otherwise.

same_functor/2

same_functor(_Term1, _Term2)

arg1 : ground : term
arg2 : ground : term

Succeeds if *arg1* and *arg2* have the same functor; fails otherwise.

The following are a series of predicates which arebased on the documentation from the Quintus prolog library manual.

change_args

The basic use of this predicates is as a rewrite facility to change patterns of arguments.

change_arg/5

change_arg(_Index, _Term, _OldTerm, _NewTerm, _ReplaceMent)

arg1 : ground : integer
arg2 : partial : term
arg3 : partial : term
arg4 : partial : term
arg5 : partial : term

This will replace attribute *arg1* in term *arg2* with a replacement term *arg5* to produce term *arg4*. The term replaced is returned in *arg3*. Sufficient arguments must be specified to identify the part of the structure to change.

change_arg/4

change_arg(_Index, _OldTerm, _NewTerm, _NewArg)

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

This will replace attribute *arg1* in term *arg2* with *arg4* to produce term *arg3*. As with **change_arg/5** *arg1* must be specified, but the other arguments may be partially instantiated.

swap_args/6

swap_args(_Index1, _Index2, _OldTerm, _Arg1, _NewTerm, _Arg2)

arg1 : ground : integer
arg2 : ground : integer
arg3 : partial : term
arg4 : partial : term
arg5 : partial : term
arg6 : partial : term

Succeeds when term arg3 and term arg5 are identical, except that in arg3 index arg1 is arg4, index arg2 is arg6 and in arg5 index arg1 is arg6, and index arg2 is arg4. Fails otherwise.

swap_args/4

swap_args(_Index1, _Index2, _OldTerm, _NewTerm)

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

This is identical to **swap_args/6**, except that *arg4* and *arg6* are not present.

change_name/4

change_name(_Name, _OldTerm, _NewName, _NewTerm)

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

Replaces the name of the functor *arg2* with a new name *arg3* to create a new term *arg4*. Sufficient of the arguments must be instantiated to allow the name and arity of the functor to be obtained.

functors

functor_copy/2

functor_copy(_OldTerm, _NewFunctor)

arg1: input: partial term
arg2: output: partial term

This creates a new functor in *arg2* based on the specification of the functor found in *arg1*. The specification may be of the form name/arity, name, or name(...). The attributes of the new functor are all new variables. The predicate reports an error if the functor cannot be copied.

functor_spec/1

functor_spec(_Functor)

arg1 : partial : term

Succeeds if *arg1* is a Prolog object which can be recognised as specifying a functor (i.e. name/arity, name, and name(...) are all recognised as functor specifications).

1.13 variables

vars

Handle variables and include membership tests on lists of variables.

var_check/2

var_check(_Term, _VarList)

arg1 : partial : term
arg2 : any : list

Succeeds only if all the variables in the term *arg1* are also in the list *arg2*.

var_replace/2

var_replace(_Term, _NewTerm)

arg1 : partial : term
arg2 : free : term

Returns a copy of a term where all the variables are replaced by $0..$X; that is, the new term is ground whereas the original was partial.

var_undo/2

var_undo(_Term, _NewTerm)

arg1 : partial : term
arg2 : free : term

This predicate returns in *arg2* a copy of the term *arg1* in which all occurrences of $0..$X are replaced by real variables. This is the opposite of **var_replace/2**.

terms

term_flatten/2

term_flatten(_Term, _List)

arg1: partial : term
arg2: free : list

Flattens the term *arg1* to a list *arg2* of atomic values.

1.14 numbers

Has a variety of files which contain predicates for manipulating numbers. These include:

arith

Defines a family of predicates for arithmetic on integers.
They provide a limited reversibility feature so that:

```
succ(3, _X) unifies _X with 4
```

while

```
succ(_X, 3) unifies _X with 2.
```

In some cases, code written with those predicates is clearer than using the built-in **is/2**.

ge/2

ge(_X, _Y)
arg1 : ground : integer
arg2 : ground : integer

Succeeds if *arg1* is greater than or equal to *arg2*.

gt/2

gt(_X, _Y)
arg1 : ground : integer
arg2 : ground : integer

Succeeds if *arg1* is strictly greater than *arg2*.

le/2

le(_X, _Y)
arg1 : ground : integer
arg2 : ground : integer

Succeeds if *arg1* is smaller than or equal to *arg2*.

lt/2

lt(_X, _Y)
arg1 : ground : integer
arg2 : ground : integer

Succeeds if *arg1* is strictly smaller than *arg2*.

plus/3

plus(_A, _B, _S)
arg1 : any : integer
arg2 : any : integer
arg3 : any : integer

This predicate succeeds when all three arguments are integers and the relation *arg1+arg2 = arg3* holds. If any two arguments are instantiated, the remaining one is solved for. At least two of the three arguments must be instantiated.

succ/2

succ(_Pred, _Succ)
arg1 : any : integer
arg2 : any : integer

This predicate succeeds when the integer *arg2* is the arithmetic successor of the integer *arg1*. At least one of the arguments must be bound to an integer.

times/3

times(_A, _B, _P)
arg1 : any : integer
arg2 : any : integer
arg3 : any : integer

This predicate succeeds when all three arguments are integers and satisfy *arg1*arg2 = arg3*.

divide/4

divide(_A, _B, _Q, _R)

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

This predicate succeeds when all four arguments are integers satisfying the following constraints:

```
_A = _B *_Q +_R
_A * _R≥ 0
0 ≤ |_R/_B| < 1
```

The predicate can be used with non-ground arguments.

between

Generates integers.

between/3

between(_Lower, _Upper, _NewInt)

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

Succeeds when the integer arguments satisfy *arg1* ≤ *arg3* ≤ *arg2*.

gen_arg/3

gen_arg(_N, _Term, _Arg)

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

This predicate works like **arg/3**, except that *arg1* needs not to be instantiated. In that case, all solutions are generated by backtracking.

gen_int/1

gen_int(_X)

arg1 : free : integer

Instantiates *arg1* to 0, then 1, -1, 2, -2, 3, -3, ...

gen_nat/1

gen_nat(_X)

arg1 : any : integer

If *arg1* is ground it will succeed if *arg1* is a natural number, otherwise it fails.
If *arg1* is free, it instantiates *arg1* to 0, then 1, 2, 3, 4... .

gen_nat/2

gen_nat(_L,_N)

arg1 : any : integer
arg2 : any : integer

If both arguments are ground it will succeed if *arg2* *arg1*

If *arg2* is free it instantiates *arg2* to *arg1*, then *arg1*+1, *arg1*+2...

If *arg1* is free it succeeds with *arg1* = *arg2*, then loops.

palip

PALIP is a package supporting arithmetic on integers whose range is a priori not limited. The only limitation is the available machine memory and the patience of the user.

All of PALIP is contained in one module called long_integers. All predicates are local, so the correct import declarations have to be issued to the compiler. To make this a bit easier, the necessary import declarations have been grouped in a library file named palip.mod. A declaration:

```
:- include('$BIM_PROLOG_DIR/include/palip.mod') .
```

in a ***ProLog by BIM*** file, makes the relevant predicates available.

To execute predicates defined in the PALIP package, one has to load the library either with the -L option of ***ProLog by BIM*** or with the **lib/1** predicate, but one must also select the palip module.

Example

```
?- lib(palip) , module(palip).
Yes
```

As a rule, the predicates fail if the arguments are of the wrong type. If arguments are of the wrong mode, ***ProLog by BIM*** error messages, or infinite loops are possible.

The representation of a long integer

A long integer - even if it is "small", for example 3 or -17, is represented as a term with principal functor **n/2**; its first argument represents the sign of the long integer, either "+" for positive, or "-" for negative; the second argument of the term is a list representing the absolute value of a long integer. The elements of this list are positive integers, and the i'th element of the list equals ((absolute value of long integer) / (10000 ^ (i-1))) modulo 10000 except that unnecessary "leading" zeros are never present.

Example

```
123456789 is represented by n('+',[6789,2345,1])
-10000 is represented by n('-',[0,1])
```

A special case is the value 0: it is always represented by n('+',[0]).

I/O and conversion predicates

multi_write/1

multi_write(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer

Writes its argument as an integer on the current output stream.

Example

```
?- multi_write(n('-',[123,456,789,0,444])) .
-4440000078904560123
Yes
```

multi_read/1

multi_read(_LongInt1, _LongtInt2, _LongInt3)

arg1 : free : long integer

Reads one line from the current input stream and transforms it to the internal representation of a long integer.

multi_integer_to_mp/2

multi_integer_to_mp(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : integer
arg2 : free : long integer

Transforms the integer *arg1* to a long integer *arg2*.

multi_mp_to_integer/2

multi_mp_to_integer(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : free : integer

Transforms the long integer *arg1* to an integer *arg2*.

General arithmetic

multi_compare/3

multi_compare(_LongInt1, _LongtInt2, _Atom)

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

Unifies *arg3* with the result of comparing *arg1* and *arg2*: the result can be **gt** for greater than, **lt** for less than, **eq** for equal.

multi_add/3

multi_add(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : ground : long integer
arg3 : free : long integer

Adds *arg1* to *arg2* and unifies the sum with *arg3*.

multi_sub/3

multi_sub(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : ground : long integer
arg3 : free : long integer

Subtracts *arg1* from *arg2* and unifies the difference with *arg3*.

multi_mul/3

multi_mul(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : ground : long integer
arg3 : free : long integer

Multiplies *arg1* with *arg2* and unifies the product with *arg3*.

multi_div/4

multi_div(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : ground : long integer
arg3 : free : long integer
arg4 : free : long integer

Divides *arg1* by *arg2* and unifies the quotient with *arg3* and the remainder with *arg4*.

multi_pow/3

multi_pow(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : ground : positive long integer
arg3 : free : long integer

Calculates *arg1* to the power *arg2* and unifies the result with *arg3*.

multi_gcd/3

multi_gcd(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : ground : long integer
arg3 : free : long integer

Calculates the greatest common divisor of *arg1* and *arg2* and unifies the result with *arg3*.

multi_sqrt/2

multi_sqrt(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : free : long integer

Calculates the square root of *arg1* and unifies the result with *arg2*.

multi_factorial/2

multi_factorial(_LongInt1, _LongtInt2, _LongInt3)

arg1 : ground : long integer
arg2 : free : long integer

Calculates the factorial of *arg1* and unifies the result with *arg2*.

multi_factor/2

multi_factor(_LongInt, _LongtInt2, _FactorList)

arg1 : ground : long integer
arg2 : free : list

Calculates the list of factors of *arg1* and unifies this list with *arg2*.

Example

```
?- multi_factor( n('+',[0,0,0,7]), _L).

_L = [p(n(+,[2]),n(+,[12])),p(n(+,[5]),n(+,[12])),p(n
(+,[7]),n(+,[1]))]

meaning:

2 is 12 times a factor of 7000000000000
5 is 12 times a factor of 7000000000000
7 is once a factor of 7000000000000
```

An interactive calculator mimicking bc

bc/0

bc

Reads from the current input stream line per line and attempts to interpret each line as an arithmetic expression, evaluates this expression and writes the result on the current output stream.

Note: This predicate issues cryptic error messages.

The predicate accepts expressions of the form:

Integer
Expr1 + Expr2
Expr1 - Expr2
Expr1 * Expr2
Expr1 / Expr2
Expr1 ^ Expr2

Expr1 % Expr2
+ Expr
- Expr
V Expr (this means square root of Expr)

long

The file long.pro contains an implementation of an arbitrary precision rational arithmetic package. Full documentation can be found in the file. The PALIP package is recommended for work on arbitrary length integers.

number_tests

This file provides four predicates testing integers for simple qualities.
They all take a single argument which must be an integer. Those predicates are intended to improve the readability of programs.

number_positive/1
number_negative/1
number_odd/1
number_even/1

arg1 : ground : integer

primes

This file provides a trivial primality tester.

number_prime/1

number_prime(_N)

arg1 : ground : integer

Succeeds if the integer *arg1* is a prime number, but fails otherwise.

number_not_prime/1

number_not_prime(_N)

arg1 : ground : integer

Succeeds if the integer *arg1* is not a prime number, but fails otherwise.

random

Random number generator. ***ProLog by BIM*** already has a built-in random number generator, but this is a Prolog version which is compatible with the DEC-10 Prolog library version.

random/2

random(_N, _I)

arg1 : ground : integer
arg2 : free : integer

Given an integer *arg1* >= 1, the predicate unifies *arg2* with a random integer between 0 and *arg1* - 1.

randomise/0

randomise

Restarts the random sequence from the beginning.

randomise/1

randomise(_Seed)

arg1 : ground : integer

Instantiates the seed for the randomizer to your own favorite value provided in *arg1*.

random/3

random(_List, _Elem, _Rest)

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

Given a non-empty list *arg1*, this predicate unifies *arg2* with a random element picked from *arg1* and unifies *arg3* with the list of the other elements of the list.

rand_perm/2

rand_perm(_List, _Perm)

arg1 : partial : list
arg2 : free : list

Instantiates *arg2* with a random permutation of the elements of list *arg1*.

1.15 text

Contains various text processing programs. Most of the routines generate nroff/troff output commands that can then be formatted for any printer. The routines are provided for generating reports to be printed. Reports that are generated directly using the *write* and *nl* predicates would not benefit from the formatting possibilities available, without requiring a great deal of work.

text

By loading this file, all of the files for text processing described below will be loaded.

Example

```
?- lib(text).
```

tbl

This package allows one to generate a table description which can be processed by the tbl tool of the AIX text processing package.

tbl/2

tbl(_Headers, _Lists)

arg1 : partial : list
arg2 : partial : list

Both arguments are lists of lists, *arg1* is a list of column definitions where a column definition is a list of atoms taken from ['c','l','r','s',...].
Arg2 is a list of row data represented by a list of atoms, one for each column.

Example

```
?- tbl([[c,s,s],[l,l,l]], [['table example'], [row1, 1, 1],
[row2, 2,3 ]]).

.TS
center,box;

c s s
l l l .

table example
row1    1       1
row2    2       3

.TE
Yes
```

title

title/1

title(_Title)

arg1 : partial : list or atom

Outputs a title suitable for nroff/troff. Each element of *arg1* is printed on a new line.

sub_title/1

sub_title(_Title)

arg1 : partial : list or atom

Outputs a subtitle suitable for nroff/troff. Each element of *arg1* is printed on a new line.

para

This package allows the generation of paragraphs and indented paragraphs from lists of atoms. The output is suitable for nroff/troff.

para/1

para(_List)

arg1 : partial : list

This predicate outputs the list *arg1* of atoms as a nroff/troff paragraph.

indent/2

indent(_Atom, _List)

arg1 : ground : atom
arg2 : partial : list

This predicate outputs the list *arg2* of atoms as an indented paragraph labeled by the atom *arg1*.

Example

```
?- para(['this is a nroff/troff paragraph',
          'Element of the list will',
          'be output as a separate line']).

\&.PP
this is a nroff/troff paragraph
Element of the list will
be output as a separate line
Yes

?- indent('label',
          ['this is a nroff/troff paragraph',
          'Element of the list will',
          'be output as a separate line']).

\&.IP "label"
this is a nroff/troff paragraph
Element of the list will
be output as a separate line
Yes
?-
```

conv

Routines that convert various string forms. To capitalize an atom or convert all capitals into lower case.

capitalise_atom/2

capitalise_atom(_Atom, _UpperCaseAtom)

arg1 : ground : atom
arg2 : any : atom

This predicate succeeds if the atom *arg2* is the atom *arg1* with all lower case letters converted to upper case.

capitalise_list/2

capitalise_list(_AsciiList, _UpperCaseAsciiList)

arg1 : ground : list
arg2 : any : list

This predicate succeeds when the list *arg2* of integers contains the elements of the list *arg1* of integers where all ASCII codes of lower case letters have been inverted to their upper case equivalent.

lower_case_atom/2

lower_case_atom(_Atom, _LowerCaseAtom)

arg1 : ground : atom
arg2 : any : atom

This predicate succeeds if the atom *arg2* is the atom *arg1* with all upper case letters converted to lower case.

lower_case_list/2

lower_case_list(_AsciiList, _LowerCaseAsciiList)

arg1 : ground : list
arg2 : any : list

This predicate succeeds when the list *arg2* of integers contains the elements of the list *arg1* of integers where all ASCII codes of upper case letters have been converted to their lower case equivalent.

Example

```
?- capitalise_atom('Convert_To_Upper', _U).
     _U = CONVERT_TO_UPPER
Yes

?- lower_case_atom('CONVERT_to_LOWER', _L).
     _L = convert_to_lower
Yes
```

1.16 meta

Predicates for helping the user in meta-programming.

applic

Application routines based on **apply/2**.

apply/2

apply(_Pred, _Args)

arg1 : partial : term
arg2 : partial : list

This predicate is the main one in this set. It is a variant of call/1 in which some of the arguments may be already in the term *arg1*, and the rest of the elements are passed in the list *arg2*.

Example

```
?- apply(foo, [X,Y]).
```

is similar to:

```
?- call(foo(X,Y)).
```

callable/1

callable(_Goal)

arg1 : partial : term

Tests whether the term *arg1* is callable; that is, whether the principal functor of *arg1* is an atom.

checkand/2

checkand(_Pred, _And)

arg1 : partial : term
arg2 : partial : term

This predicate succeeds if there is a pattern of calls to *apply (arg1, _X)* that succeeds for all non-free leaf nodes _X of the and-binary tree *arg2*.

mapand/3

mapand(_Pred, _Old, _New)

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

This predicate succeeds if there is a pattern of calls *apply (arg1, [_X, _Y])* that succeeds for all non-free leaf nodes _X of the &-binary tree *arg2*. *Arg3* is unified with the corresponding and-binary tree of _Y leaf nodes.

maplist/3

maplist(_Pred, _OldList, _NewList)

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

This predicate succeeds when all the calls to *apply (arg1, _X, _Y)* succeed for all the elements (_X, _Y) of lists *arg2* and *arg3*, respectively.

maplist/2

maplist(_Pred, _List)

arg1 : partial : term
arg2 : partial : list

This predicate succeeds if all calls *apply (arg1, _X)* succeed for each element _X of the list *arg2*.

checklist/2

checklist(_Pred, _List)

arg1 : partial : term
arg2 : partial : list

This predicate succeeds if there is a pattern of calls to *apply (arg1, _ X)* which succeeds for each element _X of the list *arg2*.

Example

```
goal(_X) :-
      var(_X),
      _X = 1 .
```

```
?- checklist(goal, [_X, _X]).
 _X = 1
Yes
?- maplist(goal, [_X, _X]).
No
```

afew/2

afew(_Pred, _List)

arg1 : partial : term
arg2 : partial : list

This predicate succeeds when at least one call to *apply (arg1, _ X)* succeeds for the elements _X of list *arg2*. Alternative elements can be checked by backtracking.

somechk/2

somechk(_Pred, _List)

arg1 : partial : term
arg2 : partial : list

This predicate succeeds as soon as one element _El of list *arg2* allows the call *apply (arg1, _El)*. Only the first solution is explored.

sublist/3

sublist(_Pred, _List, _SubList)

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

This predicate succeeds when *arg3* is the longest sublist of *arg2*, so that for all elements _E in *arg2* the call *apply (arg1, [_E])* succeeds.

convlist/3

convlist(_Pred, _OldList, _NewList)

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

This predicate unifies *arg3* with the list of all elements _E, so that the call *apply(arg1, [_X, _E])* succeeds for the elements _X of the list *arg2*.

exclude/3

exclude(_Pred, _List, _SubList)

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

This predicate unifies *arg3* with the list of elements _El from *arg2* for which the call *apply (arg1, _E)* does not succeed.

clause

This set of predicates is intended to help with the writing of programs manipulating other programs.

clause_split/3

clause_split(_Clause, _Head, _Body)

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

Takes the clause given as the first argument and returns the head and body as the second and third arguments respectively. If the body does not exist a variable is returned.

clause_real/1

clause_real(_Clause)

arg1 : partial : term

Succeeds if the term *arg1* represents a Prolog clause.

clause_rule/1

clause_rule(_Clause)

arg1 : partial : term

Succeeds if the term *arg1* represents a Prolog rule.

clause_fact/1

clause_fact(_Clause)

arg1 : partial : term

Succeeds if the term *arg1* represents a Prolog fact.

directive/1

directive(_Directive)

arg1 : partial : term

Succeeds if the term *arg1* represents a Prolog directive.

make_clause/3

make_clause(_Head, _Body, _Clause)

arg1 : partial : term
arg2 : partial
arg3 : partial

Constructs a Prolog clause in *arg3* from the terms *arg1* and *arg2*.
Facts can be constructed with *make_clause(_head, true, _fact)*.

decon

Constructs and extracts Prolog control structures.

prolog_bounded_quantification/3

prolog_bounded_quantification(_Form, _Generator, _Test)

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

Succeeds when *arg2* and *arg3* are respectively the generator and test of the term represented by the control structure *arg1*. Handles forall (Gen, Test).

prolog_clause/3

prolog_clause(_Clause, _Head, _Body)

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

Succeeds when *arg2 :- arg3* unifies with *arg1*.

Note:
It is not used to recognize whether a term is a clause or not; almost any Prolog term can serve as a (unit) clause. It is designed to build a clause given the head and body, or to extract a clause.

prolog_conjunction/2

prolog_conjunction(_Conjunction, _ListOfConjuncts)

arg1 : any : term
arg2 : any : list

Succeeds when the term *arg1* is the conjunction of all terms in the list *arg2*. This code wraps call(_) around variables, flattens disjunctions to (A',(B',(C',(D',E')))) form, and drops "true" disjuncts.

prolog_disjunction/2

prolog_disjunction(_Disjunction, _ListOfDisjuncts)

arg1 : any : term
arg2 : any : list

Succeeds when the term *arg1* is the disjunction of all terms in the list *arg2*. This code wraps call(_) around variables, flattens disjunctions to (A,(B,(C,(D,E)))) form, and drops "false" disjuncts.

prolog_if_branch/3

prolog_if_branch(_Branch, _Hypothesis, _Conclusion)

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

Handles the syntax of individual arms of if-then-elses. Succeeds when *arg1* unifies with the term *(arg2Æarg3)*.

prolog_negation/2

prolog_negation(_NegatedForm, _PositiveForm)

arg1 : any : term
arg2 : any : term

Recognizes and/or generates negations, succeeds when unifies with \+*(arg2)*.

foreach

These predicates act in a manner similar to a while loop.

foreach/2

foreach(_Goal1, _Goal2)

arg1 : partial : term
arg2 : partial : term

For each solution to the goal given in *arg1*, the goal given in *arg2* is called. If the goal *arg2* ever fails then the **foreach/2** call fails.

foreach/3

foreach(_Goal1, _Goal2, _Goal3)

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

The same as **foreach/2** except that if the second goal ever fails the goal given in *arg3* is called.

goals

Manipulates user meta-call goals.

expand_goal/3

expand_goal(_Goal, _Expand, _New)

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

This predicate adds the arguments given in the list *arg2* to the arguments already contained in the term *arg1*. The resulting term is unified with *arg3*.

call_it/1

call_it(_Goal)

arg1 : partial : term

Calls the goal *arg1*, but fails if the goal is a variable or non-callable term.

apply_to_subgoals/2

apply_to_subgoals(_Goal1, _Goal2)

arg1 : partial : term
arg2 : partial : term

This predicate calls the term *arg2* expanded with the subterms of *arg1* (that represents the body of a clause).

Example

```
?- expand_goal(goal(_Arg1), [_Arg2, _Arg3], _NewGoal).
 _Arg1 = _10
 _Arg2 = _11
 _Arg3 = _13
 _NewGoal = goal(_10,_11,_13)
Yes
?- apply_to_subgoals((goal1, goal2, goal3), write), nl.
goal1goal2goal3
Yes
```

idback

Intelligent backtracking database interpreter.

This code comes from the paper:
"An Interpreter of Logic Programs Using Selective Backtracking."
Luis Moniz Pereira & Antonio Porto,
Lisbon University report 3/80 CIUNL July 1980
presented at the Debrecen Logic Programming Workshop 1980.

For more information, see the extensive documentation found in the source file.

calls

Contains some meta-level predicates not defined in ***ProLog by BIM***.

once/1

once(_Goal)

arg1 : partial : term

Succeeds only once for the goal *arg1*.

succeed/1

succeed(_Goal)

arg1 : partial : term

Always succeeds, even if call *(arg1)* fails.

sols_exists/1

sols_exists(_Goal)

arg1 : partial : term

Succeeds if call *(arg1)* succeeds, no bindings are made to the variables of *arg1*.

unique/1

unique(_Goal)

arg1 : partial : term

Succeeds only if there is exactly one solution goal *arg1*.

at_most/2

at_most(_Max_Sols, _Goal)

arg1 : partial : term
arg2 : ground : integer

Succeeds if there are fewer than *arg1* solutions to the goal *arg2*.

find_sols/3

find_sols(_Max_Sols, _Goal, _Actual_Sols)

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

Succeeds when the integer *arg3* is the number of solutions for the goal *arg2*. An upper limit to the search is given by the integer *arg1*.

debug_not

debug_not/1

debug_not(_Goal)

arg1 : partial : term

This predicate implements a version of **not/1**. Free variables in the goal *arg1* (except setof/bagof dummy variable and those introduced by the existential quantifier ^), trigger an error message and a break level is entered.

It is purely intended as a debugging aid.

occurs

Routines for checking number/place of occurrence. ***ProLog by BIM*** already contains built-ins for performing the occur checks. They should be used in preference to the ones defined in this file.

contains/2

contains(_Kernel, _Expression)

arg1 : partial : term
arg2 : partial : term

Succeeds when the term *arg1* occurs somewhere in the term *arg2*. It must be used only as a test; to generate subterms use **subterm/2**.

freeof/2

freeof(_Kernel, _Expression)

arg1 : partial : term
arg2 : partial : term

Succeeds when the term *arg1* does not occur anywhere in the term *arg2*.

Note:

If *arg2* contains an unbound variable it will fail as *arg2* might occur there.

Since there is an infinity of Kernels not contained in any Expression, and since an infinity of Expressions do not contain any Kernel, it is recommended to limit the use of this predicate to tests.

patharg/3

patharg(_Path, _Exp, _Term)

arg1 : ground : list of integers
arg2 : partial : term
arg3 : any : term

Unifies the term *arg3* with the subterm of *arg2* found by following the path in *arg1*. It may be viewed as a generalization of **arg/3**. In order to discover a path to a known subterm, use the predicate **position/3**.

position/3

position(_Term, _Exp, _Path)

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

Succeeds when the term *arg1* occurs in term *arg2* at the position defined by the path *arg3*. It may be at other places too, so the predicate is prepared to generate them all. The path is a generalized Dewey number, as usual.

Example

```
?- position(x, 2*x^2+2*x+1=0, [1, 1, 2, 2]) .
Yes
?- position(x, 2*x^2+2*x+1=0, [1, 1, 1, 2, 1]) .
Yes
```

replace/4

replace(_Path, _OldExpr, _SubTerm, _NewExpr)

arg1 : ground : list of integers
arg2 : partial : term
arg3 : partial : term
arg4 : any : term

This predicate succeeds when the term *arg4* is identical to the term *arg2* except at the position specified by the path *arg1*, where the term *arg3* now is.

Example

```
?- replace([1,1,2,2], 2*x^2+2*x+1=0, y, 2*x^2+2*y+1=0).
Yes
```

tidy

This package provides an extendable simplifier for arithmetic and relational expressions.

tidy/2

tidy(_Old, _New)

arg1 : ground : term
arg2 : any : term

This predicate simplifies the expression *arg1* and returns the simpler form in *arg2*.

tidy_withvars/2

tidy_withvars(_Old, _New)

arg1 : ground : term
arg2 : any : term

This predicate is the same as **tidy/2**, except that it first transforms the expression into a ground form before simplifying.

tidy_stmt/2

tidy_stmt(_Old, _New)

arg1 : ground : term
arg2 : any : term

This predicate simplifies the relational parts of expressions.

tidy_expr/2

tidy_expr(_Old, _New)

arg1 : ground : term
arg2 : any : term

This predicate simplifies the arithmetic parts of expressions.

unify

test_unify/2

test_unify(_Term1, _Term2)

arg1 : partial : term
arg2 : partial : term

Prolog version of a test unification routine, unifies *arg1* with *arg2*.

test_unify/3

test_unify(_Term1, _Term2, _Bindings)

arg1 : term
arg2 : term
arg3 : list

Prolog version of a test unification routine. Instantiates *arg3* to the list of bindings made.

1.17 sorting

samsort

samsort/2

samsort(_List, _Sorted_List)

arg1 : partial : list
arg2 : free : list

This predicate sorts the list *arg1* in standard term order and unifies the result with *arg2*.

Existing ordering in the input is used to improve efficiency.

1.18 compatibility

In this section, a number of compatibility packages are introduced. They provide predicates which can be found in other Prolog implementations, thus helping in the conversion of programs to ***ProLog by BIM*** (see also "*Programming Tools and Scripts - Scripts*").

The following files and predicates are available:

DEC10_library

Some predicates that differ in ***ProLog by BIM*** compared to DEC-10:

- length/2
- dec10_abolish1
- dec10_open/1
- '^'/2
- 'I'/2

LPA_library

Some predicates for LPA's** MacProlog** (London UK):

- one/1
- forall/2
- varsin/2
- '++'/3
- '--'/3
- '**'/3
- mod/3
- sqrt/2
- abs/2
- int/2
- sign/2
- ln/2
- pwr/3
- sin/2
- cos/2
- tan/2
- pi/1
- irand/1
- charof/2
- append/3

- on/2
- length/2
- make_list
- calc_length/3
- revers/2

SWI_modules

Some predicates for SWI Prolog module manipulation (University of Amsterdam, The Netherlands):

- context_module/1
- use_module/1
- use_module/2

DELPHIA_libary

Some predicates for DELPHIA-PROLOG** programs (Seyssinet, France):

- diff/2
- equal/2
- unify/2
- make/3
- concat/3
- succed/0
- no/1
- calc/2
- test/1
- list/0
- quit/0
- append/1
- append/2
- insert/1
- insert/2
- last/1
- line/0
- stat/0

1.19 Kernel

Contains predicates defining some parts of a Prolog system in Prolog.

prolog_read

Defines the read used in DEC-10 style systems (taken directly from the public domain Prolog library). This has a definition similar to that of ***ProLog by BIM***'s **read/2** predicate when in "compatibility" mode. The main difference here is that the transformation of Definite Clause Grammars is not performed before returning the term to the user. This transformation can be achieved using the "src_to_src" transformation (see later).

prolog_read/1

prolog_read(_Term)

arg1 : partial : term

This predicate reads a term on the current input stream and unifies it with *arg1*.

prolog_read/2

prolog_read(_Term, _VarList)

arg1 : partial : term
arg2 : free : list

This predicate reads a term on the current input stream, unifies it with *arg1* and also returns a list *arg2* of terms AtomicName = Variable for each variable in the read term.

distfix

This file implements a term reader which can accept an extended syntax for Prolog terms. For instance, one can define the list **[S, is, the, set, of, X, such, that,** P] to be converted by the reader to **setof(X,P,S)**.

distfix_read/2

distfix_read(_Answer, _Variables)

arg1 : partial : term
arg2 : free : list

This predicate reads a term on the current input stream, unifies it with *arg1* and also returns a list *arg2* of terms AtomicName = Variable for each variable in the read term.

rdtoks

A tokenizer for Prolog.

read_tokens/2

read_tokens(_TokenList, _Dictionary)

This predicate implements a tokenizer for Prolog. It reads on the current input stream and returns a list of atomic tokens in *arg1* and a list of Atom = Var pairs in *arg2*. *Arg2* can be compared to a dictionary of variables found in the term.

rdtok

Another tokenizer.

read_tokens_1/2

read_tokens_1(_TokenList, _VarList)

arg1 : free : list
arg2 : free : list

This predicate also tokenizes a term read from the current input stream. The list of tokens returned in *arg1* consists of elements such as:

```
var(_Var, _Name)-- the name is retained for error messages
integer(_Int)     -- 18-bit integers or xwd(_,_) pairs
atom(_Name)       -- most atoms
string(_ChList)   -- for string constants "..."
Punct             -- for special punctuation ( ) [ ] {* *} , |
```

src_to_src

src_to_src/3

src_to_src(_Mode, _In, _Out)

The name of the source file is given in *arg2*, the name of the transformed file is specified in *arg3*. *Arg1* specifies the syntax mode of the PROLOG files. *Arg1* can have the value "native" or "compatibility" or can be free. If *arg1* is free, the current mode is selected and *arg1* will be instantiated to it.

The predicate reads terms from the input file, applies **term_expansion/2** to them and writes the transformed terms to the output file, which can then be consulted. The predicate **term_expansion/2** must be provided by the user.

This predicate unifies its second argument with the transformed version of its first argument.

Example

```
?- src_to_src(native, 'prolog_file.pro', 'result.pro').
?- consult('result.pro').
```

listing

Some additional routines for listing definitions. Only the predicate name and arity are displayed.

list_external/0

list_external

Lists routines that are external C routines.

list_static/0

list_static

Lists routines that are static.

list_dynamic/0

list_dynamic

Lists routines that are dynamic.

1.20 errors

error_report

This is a small package for analysing mode and predicate attribute errors. The motivation is simply that writing checking code for each predicate is tedious, but that once an error occurs some performance cost is acceptable if the error message is more helpful.

Consider the case of a predicate with the following definition:

```
:- mode pred(i, i, i, o).
pred(_A1, _A2, _A3, _V) :-
   atom(_A1),
   atomic(_A2),
   integer(_A3),
   var(_V),
   ...
```

This has all the required checks, but to produce reasonable error messages if it fails, the initial set of type tests can be difficult and also bug prone. The basic routine given below therefore takes a template which describes the form of the attributes and which compares the call structure with it. ie.

error_diagnose/2

error_diagnose(_Pattern, _Goal)

arg1 : ground : term
arg2 : partial : term

where Pattern is of the form goal(mode(type), ...). Mode can be one of input, output or unspecified - with unspecified meaning match on anything (ie. no type); output also has no "type" associated with it. The type declaration can be one of integer, real, atom, atomic, list or compound. Thus the call for the above predicate would become

Example

```
error_diagnose( pred(input(atom),
                     input(atomic),
                     input(integer),
                     output),
                pred(_A, _B, _C, _D) ).
```

The resulting message(s) have the same format as used by ***ProLog*** and so can be controlled using the normal error message handling routines; although this is not recommended. Multiple error messages are possible if multiple faults are discovered.

Additionally, sometimes it is required to check that a specific argument has a more specific form.

Example

```
pred_2(_Arg1, _Arg2) :-
     boolean_atom(_Arg1),
     boolean_int(_Arg2),
     !,
....
boolean_atom(_A) :-
     atom(_A),
     boolean_atom1(_A).
boolean_atom1(true).
boolean_atom1(false).
boolean_int(0).
boolean_int(1).
```

Once again displaying a sensible error message can lead to more complex code which could be avoided by using the above error diagnosis routine. The above code can be transformed into the following code.

```
pred_2(_Arg1, _Arg2) :-
     boolean_atom(_Arg1),
     boolean_int(_Arg2),
     !,
...
pred_2(_Arg1, _Arg2) :-
     error_diagnose( pred_2(input(boolean_atom),
                            input(boolean_int)),
                     pred_2(_Arg1, _Arg2)).
```

With this predicate if _Arg1 is instantiated, then the test boolean_atom/1 will be called and if it does not succeed, an appropriate error message will be displayed indicating that the first argument of pred_2/2 must be of type boolean_atom. This is a more helpful message than simply indicating that argument 1 must be atomic or an atom.

1.21 bits

bitstrings

These definitions allow the easy manipulation of bitstrings which are represented as a list of integers. This is useful in some cases where the main operations are set membership. The definitions provide a form of abstraction which means that replacing these predicates with others should be very easy if the integrity of use is preserved.

bitstring_create/1

bitstring_create(_EmptyBitString)

arg1 : free : list

Create an empty bitstring. It is provided solely for allowing the representation to be changed easily.

bitstring_set/3

bitstring_set(_Bit, _BitStringIn, _BitStringOut)

arg1 : ground : integer
arg2 : ground : bitstring
arg3 : free : bitstring

This sets a given bit in the bitstring to return a new bitstring. If the bitstring is too short then it will be extended to allow the bit to be set. The bits are numbered from 1 to N, where N is the length of the bitstring.

bitstring_length/2

bitstring_length(_BitString, _Length)

arg1 : ground : bitstring
arg2 : free : integer

This predicate determines the length of the bitstring, rounded to the nearest number of bits representable in an integer (in this case 27 bits).

bitstring_is_set/2

bitstring_is_set(_Bit, _BitStringIn)

arg1 : ground : bitstring
arg2 : ground : bitstring

This predicate succeeds only if *arg1* is set to 1 in *arg2*; fails otherwise.

bitstring_is_not_set/2

bitstring_is_not_set(_Bit, _BitStringIn)

arg1 : ground : bitstring
arg2 : ground : bitstring

This predicate succeeds only if *arg1* is set to 0 in *arg2*; fails otherwise.

bitstring_unset/3

bitstring_unset(_Bit, _BitStringIn, _BitStringOut)

arg1 : ground : integer
arg2 : ground : bitstring
arg3 : free : bitstring

This sets a given bit in the bitstring *arg2* to 0 and returns the new bitstring in *arg3*.

bitstring_invert/3

bitstring_invert(_Bit, _BitStringIn, _BitStringOut)

arg1 : ground : integer
arg2 : ground : bitstring
arg3 : free : bitstring

This inverts a given bit in the bitstring *arg2* and returns the new bitstring in *arg3*.

bitstring_and/3

bitstring_and(_BitString1, _BitString1,_BitStringOut)

arg1 : ground : bitstring
arg2 : ground : bitstring
arg3 : free : bitstring

This ANDs the bitstrings *arg1* and *arg2* together to produce a new bitstring. If *arg1* and *arg2* are of a different length the shortest is extended to the length of the longest bitstring.

bitstring_or/3

bitstring_or(_BitString1, _BitString1,_BitStringOut)

arg1 : ground : bitstring
arg2 : ground : bitstring
arg3 : free : bitstring

This ORs the bitstrings *arg1* and *arg2* together to produce a new bitstring. If *arg1* and *arg2* are of a different length the shortest is extended to the length of the longest bitstring.

bitstring_print/3

bitstring_print(_BitString)

arg1 : ground : bitstring

This displays on stdout the bit pattern as a sequence of 0 and 1's. This is mainly of use during the debugging of the bitstring operations.

Prolog and Unix Libraries

Chapter 2
Unix

The directory $BIM_PROLOG_DIR/src/unix contains a set of Prolog files providing interaction with and manipulation of the environment of the UNIX system. The compiled files can be consulted by either specifying them on the command line:

```
% BIMprolog -Lunix/<filename>
```

or by the query:

```
?- lib(<filename>).
```

Note:

The following *autonumbered titles* refer to the **UNIX library directory**.
Subtitles are the name of the **files** to be loaded with the ?- lib (<filename>) goal.

2.1 files

This directory contains predicates for manipulating files. The files are:

UnixFileSys

The library **UnixFileSys** provides an interface to (low-level) routines from the C library that are related to the file system. For the semantics and limitations of the system calls and C-library routines, see the *"SunOS™ Reference Manual"* - sections 2 and 3.

To use these predicates in a program, the module interface file **$BIM_PROLOG_DIR/include/UnixFileSys.mod** should be included in the program file. This is because the predicates are local to the module **UnixFileSys** (to prevent name clashes with existing built-ins or other predicates). The module interface file has all the necessary import declarations to use the library predicates in a program.

Example

```
:- include('-Hinclude/UnixFileSys.mod').
```

To execute examples such as the ones in this chapter, the ***ProLog by BIM*** system has to be linked with the UnixFileSys Library and one has to work in the correct module.

Example

```
csh % BIMprolog -Pq+ -Ps+ -Lunix/UnixFileSys
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993
compiled -w /prolog/lib/unix/UnixFileSys.pro
linking -Lunix/UnixFileSys.o -lc
consulted UnixFileSys.pro

?- module('UnixFileSys').
Yes
```

File manipulation

chmod/2

chmod(_Path, _Mode)

arg1 : ground : atom
arg2 : ground : integer

The mode of the file with name *arg1*, is changed to *arg2*. This predicate fails if the change failed.

Example

```
?- chmod ('a.pro', 8'777).
Yes
?- shell ('ls -l a.pro').
-rwxrwxrwx 1 prolog 919  Nov 1  0  8:45  a.pro*
Yes
```

fchmod/2

fchmod(_Fd, _Mode)

arg1 : ground : integer
arg2 : ground : integer

The mode of the file with descriptor *arg1*, is changed to *arg2*. This predicate fails if the change failed.

Example

```
?- shell('ls -ld UFS_test_DM').
Drwxr-xr-x   3 prolog     512 Aug 08 08:38 UFS_test_DM/
Yes
?- opendir(_d,UFS_test_DM),
   readdir(_de1,_d),
   get_dir(_d,_fd,_,_,_,_,_),
   get_dirent(_de1,_,_,_,_,_name),
   fchmod(_fd,8'100).
 _d = 0x200da100
 _de1 = 0x202fdc00
 _fd = 4
 _name = .
Yes
?- shell('ls -ld UFS_test_DM').
D--x------   3 prolog     512 Aug 08 08:38 UFS_test_DM/
Yes
```

umask/2

umask(_OldUmask, _NewUmask)

arg1 : free : integer
arg2 : ground : integer

The file mode creation mask is unified with *arg1*, and then changed to *arg2*.

rename/2

rename(_OldPath, _NewPath)

arg1 : ground : atom
arg2 : ground : atom

The link with name *arg1* is renamed as *arg2*. This predicate fails if an error occurred.

Example

```
?- shell('ls FILE*'),nl,
   rename(FILE,FILE_old),
   shell('ls FILE*').
FILE

FILE_old
Yes
```

link/2

link(_Path, _Link)

arg1 : ground : atom
arg2 : ground : atom

A hard link with name *arg2* is created to the file with name *arg1*. This predicate fails if the link could not be created.

Example

```
?- shell('ls -l FILE*'),nl,
   link(FILE,FILEhardlink),
   shell('ls -l FILE*'),nl.
Frw-r--r--   1 prolog        5 Aug 08 18:58 FILE

Frw-r--r--   2 prolog        5 Aug 08 18:58 FILE
Frw-r--r--   2 prolog        5 Aug 08 18:58 FILEhardlink

Yes
```

symlink/2

symlink(_Path, _Link)

arg1 : ground : atom
arg2 : ground : atom

A symbolic link with name *arg2* is created to the file with name *arg1*. This predicate fails if the link could not be created.

Example 1

```
?- shell('ls -l FILE*'),nl,
   symlink(FILE,FILEsymlink),
   shell('ls -l FILE*'),nl.
Frw-r--r--   1 prolog         5 Aug 08 18:58 FILE

Frw-r--r--   1 prolog         5 Aug 08 18:58 FILE
Lrwxrwxrwx   1 prolog       4 26 ao 14:09 FILEsymlink@ -> FILE
Yes
```

Example 2

```
?- symlink('subdir1/file3',linkfile3).
Yes
?- shell(ls).
file1       file2       linkfile3@  subdir1/    subdir2/
Yes
```

unlink/1

unlink(_Path)

arg1 : ground : atom

The file with name *arg1* is removed from the directory. Its resources are retained until all links to it are closed.

Example

```
?- shell('ls -l FILE*'),nl,
   unlink(FILEhardlink),
   shell('ls -l FILE*'),nl.
Frw-r--r--   2 prolog         5 Aug 08 18:58 FILE

Frw-r--r--   2 prolog         5 Aug 08 18:58 FILEhardlink
Frw-r--r--   1 prolog         5 Aug 08 18:58 FILE

Yes
```

readlink/2

readlink(_Path, _Link)

arg1 : free : atom
arg2 : ground : atom

Arg1 is instantiated to the contents of the symbolic link with name *arg2*. This predicate fails if an error occurred.

access/2

access(_Path, _Mode)

arg1 : ground : atom
arg2 : ground : integer or term

Succeeds if the process has *arg2* access type to the file with name *arg1*. The access mode *arg2* can be an integer or an or-composition of symbols from the table below.

Symbol	Description
R_OK	Read permission
W_OK	Write permission
X_OK	Execute permission
F_OK	File existence

Example

```
?- shell('ls -l FILE'),nl.
Fr--r--r--   1 prolog          5 Aug 08 18:58 FILE

Yes
?- access(FILE,R_OK).
Yes
?- access(FILE,X_OK|W_OK).
No
```

Structure stat

The information of a file is represented by the UNIX structure **stat**. This structure includes the following members:

st_dev;	device inode resides on
st_ino;	this inode's number
st_mode;	protection
st_nlink;	number of hard links to the file
st_uid;	user ID of owner
st_gid;	group ID of owner
st_rdev;	the device type, for inode that is device
st_size;	total size of file, in bytes
st_atime;	file last access time
st_mtime;	file last modify time
st_ctime;	file last status change time
st_blksize;	optimal blocksize for file system i/o ops
st_blocks;	actual number of blocks allocated.

See the *"SunOS™ Reference Manual"* for more information.

Example

```
?- create_stat(_stat),
   stat(file1,_stat),
   get_stat(_stat,_d,_i,_m,_nl,_u,_g,_rd,_s,
            _at,_mt,_ct,_bs,_nb),
   get_stat_mode(_m,_sm),
   destroy(_stat).
_stat = 0x107594
_d = -32252
_i = 20558
_m = 33188
_nl = 1
_u = 12
_g = 101
_rd = -30914
_s = 1
_at = 0x26ee4f46
_mt = 0x26ee4f5d
_ct = 0x26ee4f5d
_bs = 8192
_nb = 2
_sm = [S_IFREG,S_IRUSR,S_IWUSR,S_IRGRP,S_IROTH]
Yes
```

create_stat/1

create_stat(_Stat)

arg1 : free : pointer

A new **stat** structure is created. *Arg1* is instantiated to its handle.

destroy_stat/1

destroy_stat(_Stat)

arg1 : ground : pointer

The **stat** structure with handle *arg1*, is deallocated.

stat/2

stat(_Path, _Stat)

arg1 : ground : atom
arg2 : ground : pointer

The **stat** structure, pointed to by *arg2*, is filled with the file status of the file with name *arg1*. This predicate fails if the information retrieval failed.

lstat/2

lstat(_Path, _Stat)

arg1 : ground : atom
arg2 : ground : pointer

Same as **stat/2**, unless *arg1* is a symbolic link, in which case the status information of the link is returned instead of the referenced file.

fstat/2

fstat(_Fd, _Stat)

arg1 : ground : integer
arg2 : ground : pointer

The **stat** structure, pointed to by *arg2*, is filled with the file status of the file with descriptor *arg1*. This predicate fails if the information retrieval failed.

get_stat/14

get_stat(_Stat, _Dev, _Inode, _Mode, _NLinks, _Uid, _Gid, _RDev, _Size, _ATime, _MTime, _CTime, _BSize, _NBlocks)

arg1 : ground : pointer
arg2 : free : integer : device number
arg3 : free : integer : file inode number
arg4 : free : integer : protection mode
arg5 : free : integer : number of hard links
arg6 : free : integer : owner user id
arg7 : free : integer : owner group id
arg8 : free : integer : device id (for special files)
arg9 : free : integer : total file size
arg10 : free : pointer : last access time
arg11 : free : pointer : last modify time
arg12 : free : pointer : last status change time
arg13 : free : integer : preferred block size
arg14 : free : integer : actual allocated number of blocks

The fields of **stat** structure *arg1* are retrieved.

get_stat_mode/2

get_stat_mode(_Mode, _SymMode)

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

The file status mode *arg1* is converted to a list of symbolic flags, giving the file type, possible execution set flags and access rights for user, group and others.

Directory manipulation

mkdir/2

mkdir(_Path, _Mode)

arg1 : ground : atom
arg2 : ground : integer

A new directory is created with name *arg1*. Its mode is initialized from *arg2*. This predicate fails if the creation failed.

Example

```
?- shell('ls -ld NEW*'),nl,
   mkdir(NEW_DIR,8'750),
   shell('ls -ld NEW*'),nl.
Frw-r--r--   1 prolog          5 Aug 08 18:58 NEW_FILE

Drwxr-x---   2 prolog        512 Aug 09 14:47 NEW_DIR/
Frw-r--r--   1 prolog          5 Aug 08 18:58 NEW_FILE

Yes
```

rmdir/1

rmdir(_Path)

arg1 : ground : atom

The directory with name *arg1* is removed. This predicate fails if the remove failed.

Example

```
?- shell('ls -ld NEW*'),nl,
   rmdir(NEW_DIR),
   shell('ls -ld NEW*'),nl.
Drwxr-x---   2 prolog         512 26 ao 14:47 NEW_DIR/
Frw-r--r--   1 prolog           5 Aug 08 18:58 NEW_FILE
Frw-r--r--   1 prolog           5 Aug 08 18:58 NEW_FILE

Yes
```

chdir/1

chdir(_Path)

arg1 : ground : atom

The directory with name *arg1* is made the current working directory. This predicate fails if the change failed.

Example

```
?- shell(pwd),
   shell('ls -ld NEW*'),nl,
   chdir(NEW_DIR),
   shell(pwd).
/usr/shadow/prolog/tests/UnixFileSys
Drwxr-x---   2 prolog         512 26 ao 14:47 NEW_DIR/
Frw-r--r--   1 prolog           5 Aug 08 18:58 NEW_FILE

/usr/shadow/prolog/tests/UnixFileSys/NEW_DIR
Yes
```

Structure dir

The information of a directory is represented by the UNIX structure **DIR**. This structure includes the following members:

dd_fd;	file descriptor of directory.
dd_loc;	logical offset in directory.
dd_size;	size of buffer.
dd_blksize;	this file system's block size.
dd_curoff;	real offset in directory corresponding to dd_loc.
dd_buf;	malloc'd buffer depending on fb size.

opendir/2

opendir(_Dir, _Path)

arg1 : free : pointer
arg2 : ground : atom

The directory with name *arg2* is opened for reading. Its handle is returned in *arg1*. This predicate fails if the open failed.

telldir/2

telldir(_Position, _Dir)

arg1 : free : pointer
arg2 : ground : pointer

Arg1 is instantiated to the current position in directory *arg2*.

seekdir/2

seekdir(_Dir, _Position)

arg1 : ground : pointer
arg2 : ground : pointer

The position in directory *arg1* is set to *arg2*.

closedir/1

closedir(_Dir)

arg1 : ground : pointer

Directory *arg1* is closed.

get_dir/7

get_dir(_Dir, _Fd, _Loc, _Size, _BSize, _Off, _Buf)

arg1 : ground : pointer
arg2 : free : integer
arg3 : free : integer : logical offset in directory
arg4 : free : integer : size of buffer
arg5 : free : integer : this filesystem's block size
arg6 : free : integer : real offset in directory corresponding to dd_loc
arg7 : free : pointer : malloc'd buffer depending on fb size

The fields of **DIR** structure *arg1* are retrieved.

Example

```
?- shell('ls -a subdir2').
./ ../     subsubdir/
Yes
?- opendir(_d,subdir2),
   get_dir(_d,_fd,_l,_s,_bs,_o,_b),
   closedir(_d).
 _d = 0x107578
 _fd = 6
 _l = 0
 _s = 0
 _bs = 8192
 _o = 0
 _b = 0x27c788
Yes
```

Structure dirent

The information of a directory entry is represented by the UNIX structure **dirent**. This structure includes the following members:

d_offset;	offset next directory entry.
d_ino;	unique number of the file.
d_reclen;	directory record length in bytes.
d_namlen;	length of the file name.
d_name;	file name.

readdir/2

readdir(_DirEntry, _Dir)

arg1 : free : pointer
arg2 : ground : pointer

The next entry from directory *arg2* is read, and *arg1* is instantiated to its handle. This predicate fails if the read failed, or if there were no more entries.

get_dirent/6

get_dirent(_DirEnt, _Off, _FileNo, _RecLen, _NamLen, _Name)

arg1 : ground : pointer
arg2 : free : integer : offset of next disk directory entry
arg3 : free : integer : file number of entry
arg4 : free : integer : length of this record
arg5 : free : integer : length of name
arg6 : free : atom : name

The fields of structure *arg1* are retrieved.

Example

```
?- opendir(_d,subdir2),
   readdir(_de,_d),
   get_dirent(_de,_o,_fn,_rl,_nl,_nm),
   closedir(_d).
_d = 0x10755c
_de = 0x266000
_o = 12
_fn = 4314
_rl = 16
_nl = 1
_nm = .
Yes
```

File I/O system calls

open/4

open(_Fd, _Path, _Flags, _Mode)

arg1 : free : integer
arg2 : ground : atom
arg3 : ground : integer or term
arg4 : ground : integer

The file with name *arg2* is opened for action *arg3*, and *arg1* is instantiated to its file descriptor. *Arg3* may be an integer or an or-composition of symbols from the table below. If a file has to be created, its mode is taken from *arg4*. This predicate will fail if the open failed.

Symbol	Description
O_RDONLY	Opens for reading only.
O_WRONLY	Opens for writing only.
O_RDWR	Opens for reading and writing.
O_NDELAY	Opens delay control.
O_SYNC	Controls synchronization of subsequent writes.
O_APPEND	Opens for append at end-of-file.
O_CREAT	Creates file if necessary.
O_TRUNC	Truncates the file to zero if it exists.
O_EXCL	Creates only if file does not exist.

open/3

open(_Fd, _Path, _Flags)

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

open(_Fd, _Path, _Flags) is defined as **open(_Fd, _Path, _Flags, 0)**

close/1

close(_Fd)

arg1 : ground : integer

The file with descriptor *arg1* is closed.

dup/2

dup(_NewFd, _OldFd)

arg1 : free : integer
arg2 : ground : integer

The file with descriptor *arg2* is duplicated. *Arg1* is instantiated to the resulting file descriptor, which is the lowest available one.

dup2/2

dup2(_OldFd, _NewFd)

arg1 : ground : integer
arg2 : ground : integer

The file with descriptor *arg1* is duplicated into file descriptor *arg2*.
If *arg2* was already in use, it is first closed.

getdtablesize/1

getdtablesize(_DescTableSize)

arg1 : free : integer

Instantiates *arg1* to the size of the process descriptor table.

lseek/4

lseek(_Position, _Fd, _Offset, _Whence)

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

The file pointer of the file with descriptor *arg2* is set to offset *arg3*, as specified by *arg4*. The resulting position is returned in *arg1*. The seek mode *arg4* can be one from the table below. The predicate fails if the seek failed.

Symbol	Description
L_SET	Absolute offset.
L_INCR	Incremental offset.
L_XTND	Extended offset beyond end-of-file.

read/4

read(_NRead, _Fd, _Buffer, _NBytes)

arg1 : free : integer
arg2 : ground : integer
arg3 : ground : pointer
arg4 : ground : integer

Arg4 bytes are read from the file with descriptor *arg2* into the buffer pointed to by *arg3*. *Arg1* is instantiated to the number of bytes that have actually been read. The predicate fails if the read has failed.

write/4

write(_NWritten, _Fd, _Buffer, _NBytes)

arg1 : free : integer
arg2 : ground : integer
arg3 : ground : pointer
arg4 : ground : integer

Arg4 bytes are written from the buffer pointed to by *arg3* to the file with descriptor *arg2*. *Arg1* is instantiated to the number of bytes that have actually been written. The predicate fails if the write has failed.

fcntl/4

fcntl(_Ret, _Fd,_Command, _Arg)

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

A file control command *arg3* is executed on the file with descriptor *arg2*. Any argument for the command is given in *arg4*, and if there is a result for the operation, it is returned in *arg1*.

Defined commands (*arg3*) are given in the table below.

Symbol	Description
F_DUPFD	Duplicates descriptor *arg2* into lowest available descriptor greater than or equal to *arg4*.
F_GETFD	Gets the close-on-exec flag for descriptor *arg2*.
F_SETFD	Sets the close-on-exec flag for descriptor *arg2* to *arg4*.
F_GETFL	Gets descriptor *arg2*'s status flags.
F_SETFL	Sets descriptor *arg2*'s status flags to *arg4*.
F_GETOWN	Gets process id or process group id currently receiving SIGIO and SIGURG signals. Process groups are negative.
F_SETOWN	Sets process or process group to receive SIGIO and SIGURG signals. Process groups must be negative.

list_preds

list_predicates/2

list_predicates(_Dir, _List)

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

Lists in *arg2* all the predicates found in a given directory structure.

check_files

check_files/1

check_files(_Dir)

arg1 : ground : atom

Reads each file in the directory *arg1*. This is used as a quick test to check that at least the Prolog files can be read.

bundle

bundle/2

bundle(_File, _Output_File)

arg1 : ground : atom
arg2 : ground : atom

Takes the file *arg1* and produces the expanded version of the file where all the **lib/1** declarations are replaced by the contents of the library files (comments are removed). The expanded file is written into *arg2*.

dir

This file contains various predicates for manipulating UNIX directory trees from within ***ProLog by BIM***. It also contains the predicates necessary to search a directory structure for a given file. Definitions include:

dir_list/2

dir_list(_Dirname, _List)

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

Lists the objects in a directory. *Arg1* is the directory name. The list of objects in the directory *arg1* ('.' '..' and '.*' are not included) is returned in *arg2*.

sub_dirs/2

sub_dirs(_Dirname, _List)

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

The list of subdirectories found in directory *arg1*, is returned in *arg2*.

is_directory/1

is_directory(_Name)

arg1 : ground : atom

Succeeds if *arg1* is a directory name.

is_subdirectory/3

is_subdirectory(_Dirname, _Name)

arg1 : ground : atom
arg2 : ground : atom

Succeeds if its *arg2* is the name of a subdirectory of *arg1*.

obj_path/3

obj_path(_Dirname, _Filename,_Object)

Arg3 is the expanded object name, constructed from directory name *arg1* and file name *arg2*.

ftw/3

ftw(_Path, _Function,_Resultslist)

arg1 : ground : atom
arg2 : ground : goal
arg3 : free : list of atom

Instantiates *arg3* to the list of objects that satisfy a test *arg2*. The objects are searched for in the directory of root *arg1*. The user-supplied goal is expanded by two arguments: the input and the output. This routine is similar to ftw in the C libraries.

ft_search/3

ft_search(_Path, _Function, _Result)

arg1 : ground : atom
arg2 : ground : goal
arg3 : free : atom

As for **ftw/3**, but returns successive solutions on backtracking. The user-supplied goal is expanded by two arguments: the input and the output.

filename

Contains predicates for splitting a file name. The definitions in this file are:

suffix/2

suffix(_Filename, _Suffix)

arg1 : ground : atom
arg2 : any : atom

Arg2 will be unified with the suffix of file name *arg1*.

Example

```
?- suffix('filename.pro', _Suffix).
 _Suffix = pro
Yes
```

filename/2

filename(_Name, _Filename)

arg1 : ground : atom
arg2 : any : atom

Arg2 is unified with the file name part of atom *arg1*. The directory specification is stripped.

The directory specification is stripped.

Example

```
?- filename('/tmp/filename.pro', _Filename).
 _Filename = filename.pro
Yes
```

directory/2

directory(_Name, _Directory)

arg1 : ground : atom
arg2 : any : atom

Arg2 is unified with the directory part of file name *arg1*.

Example

```
?- directory('/tmp/filename.pro', _Directory).
 _Directory = /tmp
Yes
```

basename/2

basename(_Name, _Basename)

arg1 : ground : atom
arg2 : any : atom

Similar to **filename/2**, but also the suffix of the file name is stripped.

Example

```
?- basename('/tmp/filename.pro', _Basename).
 _Basename = filename
Yes
```

filename_parts/5

filename_parts(_Name, _Directory, _Filename, _Basename, _Suffix)

arg1 : ground : atom
arg2 : any : atom
arg 3 : any : atom
arg4 : any : atom
arg5 : any : atom

A combination of the previous predicates. *Arg1* is the file name to be decomposed. *Arg2* is unified with the directory name, *arg3* with the file name, *arg4* with the base name and *arg5* with the suffix.

Example

```
?- filename_parts('/tmp/filename.pro', _Directory, _Filename,
                  _Base, _Suf).
 _Directory = /tmp
 _Filename = filename.pro
 _Base = filename
 _Suf = pro
Yes
```

tmp_file

Generates temporary file names. The file name will be based on the template "tmpnam.xxx". The system ensures that no existing file bears the same name.

tmpnam/1

tmpnam(_Filename)

arg1 : free : atom

The file name returned in *arg1* includes a directory specification. If the environment variable TMP-DIR exists, its value is taken as a directory. Otherwise, the current directory is taken.

Example

```
?- tmpnam(_x).
 _x = /usr/prolog/test/tmpnam.xxx0
Yes
```

tmpnam/2

tmpnam(_Filename, _Tmpdir)

arg1 : free : atom
arg2 : ground :atom

The returned argument *arg1* is the new file name in the directory as specified by the path name *arg2*.

files+

files_exist/1

files_exist(_Files)

arg1 : ground : list of atoms

Succeeds if all the files in the list exist.

files_delete/1

files_delete(_Files)

arg1 : input : list of atoms

Unlinks all the files in the list. This will fail if one of the files does not exist.

files_load_if_exists/1

files_load_if_exists(_Files)

arg1 : ground : list of atoms

All files in list *arg1* that exist are reconsulted into the ProLog system.

files_which_exist/2

files_which_exist(_List, _Files)

arg1 : ground: list of atoms
arg2 : free: list of atoms

This scans the list *arg1* and returns a new list *arg2* which contains those files in *arg1* which exist. No order is specified for the returned list.

keep

Saves a predicate into a file. Comments are discarded. The definitions are:

keep/2

keep(_Predicate, _File)

arg1 : ground : atom
ground : term : functor/arity
arg2 : ground : atom

The predicates specified by *arg1*, are saved in file *arg2*. If file *arg2* exists, it is overwritten. If *arg1* is a term (functor/arity), the specified predicate is stored. If *arg1* is an atom, all predicates with functor name *arg1* are stored.

keep/1

keep(_Predicate)

The predicate *arg1* is saved in the file used by the previous **keep/2**. If no **keep/2** was called yet, a message is displayed and the predicate fails.

backup

backup/1

backup(_Filename)

arg1 : ground : atom

Creates a backup copy of the file *arg1*. The file to be backed up is copied into a new file having the same name as the original with an appended extension character "~".

edit

Calls a specified editor with a given file from ***ProLog by BIM***. The file is reconsulted when the editing is finished.

edit/1

edit(_File)

arg1 : ground : atom

Calls the editor defined in **editor/1** with the file *arg1* from Prolog. The (possibly modified) file is reconsulted when the editing is finished.

editor/1

editor(_Editor)

arg1 : ground : atom

Fact controlling which editor is going to be called by **edit/1** (default vi).

save_proc/2

save_proc(_Procedure, _Filename)

arg1 : ground : atom
: ground : term (functor/arity)
arg2 : ground : atom

Saves the procedure *arg1* in the file *arg2*. Similar to **keep/2**.

file

Routines for returning a list of clause functors defined in a file, or the directives.

file_pred/2

file_pred(_File, _Predicates)

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

Returns in the list *arg2* the clauses defined in file *arg1*.

file_dir/2

file_dir(_File, _List)

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

Returns in the list *arg2* the directives defined in file *arg1*.

record_file

Reads a Prolog file into the record database of ***ProLog by BIM***. The file is represented as a list. This can be used when manipulating a Prolog program. However, it requires a large amount of storage space. Before appending a file's term, it will be passed to a user-defined function that may process it further.

record_file/3

record_file(_Key, _Filename, _Usergoal)

arg1 : ground : atom
arg2 : ground : atom
arg3 : ground : goal

Arg1 is the record database key under which it will be recorded, *arg2* is the file name and *arg3* is the user-defined goal which will be applied to each term read from the file. The predicate fails if the key already exists.

2.2 time

UnixTime

The library **UnixTime** provides an interface to (low-level) routines from the C library that are related to the file system. For the semantics of the system calls and C-library routines, see *"SunOS™ Reference Manual"* - sections 2 and 3.

To use these predicates in a program, the module interface file **$BIM_PROLOG_DIR/include/UnixTime.mod** should be included in the program file. This is because the predicates are kept local to the module **UnixTime** (to prevent name clashes with existing built-ins or other predicates). The module interface file has all the necessary import declarations to use the library predicates in a program.

Example

```
:- include('-Hinclude/UnixTime.mod').
```

To execute the following examples, the ***ProLog by BIM*** system has to be linked with the UnixTime Library and one has to work in the correct module.

Example

```
csh % BIMprolog -Pq+ -Ps+ -Lunix/UnixTime
ProLog by BIM - release 4.0 - 15-Oct-1993
(c) Copyright BIM - 1991-1993
compiled -w /prolog/lib/unix/UnixTime.pro
linking -Lunix/UnixTime.o -lc
consulted UnixTime.pro

?- module(UnixTime).
Yes
```

time/1

time(_Clock)

arg1 : free : pointer

Arg1 is instantiated to the current time. This is expressed in seconds since the epoch (00:00:00 GMT 01 jan 1970).

ctime/2

ctime(_AsciiTime, _Clock)

arg1 : free : atom
arg2 : ground : pointer

The time *arg2*, in seconds since the epoch, is returned in textual form in *arg1*.

Example

```
?- time(_t),
   ctime(_ct,_t).
 _t = 0x386d356f
 _ct = Fri Dec 31 23:59:59 1999
Yes
```

dysize/2

dysize(_Days, _Year)

arg1 : free : integer
arg2 : ground : integer

Arg1 is instantiated to the number of days in year *arg2*.

Example

```
?- dysize(_Days,1999).
 _Days = 365
Yes
```

Structure timeb

The time information can be represented by the UNIX structure **timeb**. This structure includes the following members:

time;	time in seconds since the epoch (00:00:00 GMT 01 jan 1970)
millitm;	milliseconds
timezone;	timezone in minutes westward from Greenwich
dstflag;	indicator daylight saving time.

ftime/1

ftime(_TimeB)

arg1 : free : pointer

A **timeb** structure is filled with the current time. Its handle is returned in *arg1*. Successive calls of this predicate destroy the previous timeb information.

get_timeb/5

get_timeb(_TimeB, _Time, _MTime, _TimeZone, _Dst)

arg1 : ground : pointer
arg2 : free : pointer : time in seconds since the epoch
arg3 : free : integer : milliseconds
arg4 : free : integer : timezone in minutes westward from Greenwich
arg5 : free : integer : daylight saving time indicator

The fields of the **timeb** structure arg1 are retrieved.

Structure tm

The time information can be represented by the UNIX structure **tm**. This structure includes the following members:

tm_sec;	seconds (0 - 59)
tm_min;	minutes (0 - 59)
tm_hour;	hours (0 - 23)
tm_mday;	day of month (1 - 31)
tm_mon;	month of year (0 - 11)
tm_year;	year - 1900
tm_wday;	day of week (Sunday = 0)
tm_yday;	day of year (0 - 365)
tm_isdst;	1 if DST in effect
tm_zone;	abbreviation of timezone name
tm_gmtoff;	offset from GMT in seconds.

asctime/2

asctime(_AsciiTime, _Time)

arg1 : free : atom
arg2 : ground : pointer

The time arg2, which is a **tm** structure, is returned in textual form in *arg1*.

localtime/2

localtime(_LocalTime, _Clock)

arg1 : free : pointer
arg2 : ground : pointer

The time *arg2*, in seconds since the epoch, is broken down in a **tm** structure, whose handle is returned as *arg1*. It is corrected for the local time system.

Example

```
?- time(_t),
   localtime(_lt,_t),
   asctime(_at,_lt).
_t = 0x386d356f
_lt = 0x107098
_at = Fri Dec 31 23:59:59 1999
Yes
```

gmtime/2

gmtime(_GmTime, _Clock)

arg1 : free : pointer
arg2 : ground : pointer

The time *arg2*, in seconds since the epoch, is broken down in a **tm** structure, whose handle is returned as *arg1*. It is expressed as Greenwich Mean Time.

Example

```
?- time(_t),
   gmtime(_gmt,_t),
   asctime(_at,_gmt).
_t = 0x386d356f
_gmt = 0x107098
_at = Fri Dec 31 22:59:59 1999
Yes
```

get_tm/12

get_tm(_Tm, _Sec, _Min, _Hour, _MDay, _Month, _Year, _WDay, _YDay, _IsDst, _Zone, _GmtOff)

arg1 : ground : pointer
arg2 : free : integer : seconds
arg3 : free : integer : minutes
arg4 : free : integer : hour
arg5 : free : integer : day of month
arg6 : free : integer : month
arg7 : free : integer : year
arg8 : free : integer : day of week
arg9 : free : integer : day of year
arg10 : free : integer : daylight saving time indicator
arg11 : free : atom : always instantiated to the empty atom''
arg12 : free : integer : always instantiated to 0

The fields of the **tm** structure *arg1* are retrieved.

Example

```
?- time(_t).
   localtime(_lt,_t),
   get_tm(_lt,_s,_m,_h,_d,_mo,_y,_g,_dy,_p,_q,_l).
_t = 0x386d356f
_lt = 0x200241b4
_s = 59
_m = 59
_h = 23
_d = 31
_mo = 11
_y = 99
_g = 5
_dy = 364
_p = 0
_q =
_l = 0
Yes
```

Structure timeval and timezone

The timeval information can be represented by the UNIX structure **timeval**. This structure includes the following members:

tv_sec;	seconds since 00:00:00 01 jan 1970
tv_usec;	microseconds.

The timezone information can be represented by the UNIX structure **timezone**. This structure includes the following members:

tz_minuteswest;	west of Greenwich
tz_dsttime;	type of dst correction to apply.

The **timezone** structure indicates the local time zone (measured in minutes westward from Greenwich), and a flag that indicates the type of Daylight Saving Time (DST) correction to apply.

Note: this flag does not indicate whether Daylight Saving Time is currently in effect.

The flag indicating the type of Daylight Saving Time correction should have one of the following values (as defined in <sys/time.h>):

0	DST_NONE:	Daylight Saving Time not observed
1	DST_USA:	United States DST
2	DST_AUST:	Australian DST
3	DST_WET:	Western European DST
4	DST_MET:	Middle European DST
5	DST_EET:	Eastern European DST
6	DST_CAN:	Canadian DST.

gettimeofday/2

gettimeofday(_TimeVal, _TimeZone)

arg1 : free : pointer
arg2 : free : pointer

The current time is retrieved in a **timeval** and a **timezone** structure. *Arg1* is instantiated to the handle for the timeval and *arg2* for the timezone. Successive calls override the contents of these structures. The predicate fails if it was impossible to retrieve the time.

get_timeval/3

get_timeval(_TimeVal, _Sec, _uSec)

arg1 : ground : pointer
arg2 : free : integer : seconds
arg3 : free : integer : microseconds

The fields of the **timeval** structure *arg1* are retrieved.

get_timezone/3

get_timezone(_TimeZone, _MinWest, _DstZone)

arg1 : ground : pointer
arg2 : free : integer : minutes west from Greenwich
arg3 : free : atom : Daylight Saving Time zone

The fields of the **timezone** structure *arg1* are retrieved.

Example

```
?- gettimeofday(_tv,_tz),
   get_timeval(_tv,_s,_micros),
   get_timezone(_tz,_mw,_dstz).
_tv = 0x272b80
_tz = 0x272b88
_s = 0x26ee4ba2
_micros = 0xc8322
_mw = -60
_dstz = DST_MET
Yes
```

calendar

The program ***calendar*** manipulates dates. The possible formats for dates are [24,12,89], 24-dec-89 and 12/24/89. The defined predicates are:

legaldate/1

legaldate(_Date)

arg1 : ground : date

Checks if *arg1* describes a possible date. The possible formats for dates are [24,12,89], 24-dec-89 and 12/24/89.

weeknumber/2

weeknumber(_Date, _Week)

arg1 : ground : date
arg2 : any : integer

Unifies *arg2* with the week number of date *arg1*.

daytodate/2

daytodate(_Day, _Date)

arg1 : free : atom
arg2 : ground : date

Converts the date *arg2* into an atom representing the day.

dateoffset/3

dateoffset(_Date, _Offset, _NewDate)

arg1 : ground : date
arg2 : ground : integer
arg3 : any : date

Calculates the date *arg3* of the day which is *arg2* days from the date *arg1*. The offset *arg2* is controlled by the fact **suppress_week_end/1** and the predicate **addholiday/1**.

suppress_week_end/1

suppress_week_end(_WeekEndSuppressToggle)

arg1 : ground : atom : on/off

Fact controlling the "offset"-days of **dateoffset/3**. If arg1 is **off** (default) the offsets in dateoffset will be calendar days. If *arg1* is **on**, only working days are counted.

Example

```
?- update(suppress_week_end(on)),
   dateoffset(11/13/89, 5 , _date ),
   daytodate( _day , _date ).
 _date = 11 / 20 / 89
 _day = Monday 20 Nov 89
Yes
```

addholiday/1

addholiday(_Date)

arg1 : ground : date

Arg1 will not be counted any more in the "offset"-days of **dateoffset/1.**

Example

```
?- addholiday(11/20/89).
Yes
?- dateoffset(11/13/89, 5 , _date ),
   daytodate( _day , _date ).
 _date = 11 / 21 / 89
 _day = Tuesday 21 Nov 89
Yes
```

2.3 general

information

This directory contains predicates providing general interaction with UNIX.

date/1

date(_Date)

arg1 : free : atom

Returns the date in *arg1*.

whoami/1

whoami(_User)

arg1 : free : atom

Returns login name of user in *arg1*.

path/1

path(_Path_List)

arg1 : free : list of atom

Gets the path of the user returned as list.

current_dir/1

current_dir(_Dir)

arg1 : free : atom

Returns in *arg1* the current directory.

termcap/1

termcap(_Termcap_List)

arg1 : free : list of atom

Retrieves and splits up the termcap entry.

2.4 shell

command

This file contains predicates interfacing with the UNIX command shell.

command_gen/2

command_gen(_Partlist, _Command)

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

Arg2 is instantiated to the UNIX command generated from *arg1*, the list of subparts. The spaces required between the parts are inserted for the user.

command_call/1

command_call(_PartList)

arg1 : ground : list of atom

Creates the command as an atom and passes it to the system for execution.

2.5 system

uname

This is an interface to the function uname(2) which is part of UNIX. It allows the easy recovery of operating system delated which could be of use if such things as alternative libraries must be loaded, or some variations in interfaces.

uname/2

uname(_Option, _Value)

arg1: input: atom
arg2: free : atom

The allowed options are:

sysname	the system name
nodename	the node name
release	the release of the operating system
version	the version of the os
machine	machine form

More information on the details of this can be found in the manual page for uname(2).

2.6 io

Contains some predicate definitions for a variety of I/O operations, some of which may be found in other Prolog systems. The various files are:

ask

Asks the user for a response to questions and checks the answer. The answer is terminated by a newline rather than the normal "." of Prolog. Each definition takes a prompt as the first argument and returns the input object as the second argument. I/O streams are left unaffected. The definitions include:

ask_user/2

ask_user(_Prompt, _Object)

arg1 : ground : atom
arg2 : free : term

Places the prompt *arg1* on /dev/tty, reads any Prolog term from /dev/tty and instantiates *arg2* to it.

yes_or_no/2

yes_or_no(_Prompt, _Answer)

arg1 : atom
arg2 : free : atom

Repeats the question *arg1* until a "yes" or a "no" is input by the user. Also accepted are "Y", "N", "y" or "n" as abbreviations. *Arg2* is instantiated to either " yes" or "no".

ask_for_integer/2

ask_for_integer(_Prompt, _Answer)

arg1 : ground : atom
arg2 : free : integer

The question *arg1* is repeated until an integer is read. *Arg2* is instantiated to the integer.

getfile

getfile/2

getfile(_Prompt, _Filename)

arg1 : ground : atom
arg2 : free : atom

Gets a file name from the user. It uses the predicates defined in ask. *Arg1* is the prompt the *arg2* will be instantiated to the file name read.

change_io

Changes the I/O streams for ***ProLog by BIM***. Allows the stream "/dev/tty" to be changed without effecting it in ***ProLog by BIM***. The two definitions here are:

change_input/2

change_input(_Orig, _New)

arg1 : atom or free
arg2 : atom or free

The first argument is the input stream, and the second is the new input stream. If the current input stream is /dev/tty it is not changed.

change_output/2

change_output(_Orig, _New)

arg1 : atom or free
arg2 : atom or free

The first argument is the current output stream and the second argument is the new output stream. If the output stream is /dev/tty it is not changed.

portray_str

portray/1

portray(_List)

arg1 : list of chars

Portrays lists of characters as strings enclosed in ". Variables or $VAR/1 will be preceded by the pipe symbol "|".

printchars

printchars/1

printchars(_List)

arg1 : ground : list of integers

Prints a list of integers (ASCII codes) on the output stream as a string of characters.

putstr

putstr/1

putstr(_Obj)

arg1 : term or atom

If *arg1* is an atom, it is printed on the current output stream. If *arg1* is the structure with the form s(_File, _Start, _Length), it is taken as being a description of the file, file position and length of the string to be output and manipulated accordingly.

read

Predicates to read from a file and return end_of_file rather than failing as ***ProLog by BIM***'s read does by default. This can also be achieved using a **please/2** option setting (see "*Built-in Predicates - Switches*"). The definitions here include:

read_term/2

read_term(_Term, _Vars)

arg1 : term or free
arg2 : term of free

Similar to **vread/2**, except that the atom *end_of_file* is returned when the end-of-file is reached.

get_char0/1

get_char0(_Char)

arg1 : char or free

The same as **get0/1** except that 26 is returned when the end-of-file is reached.

get_char/1

get_char(_Char)

arg1 : char or free

The same as **get/1**, except that 26 is returned when the end-of-file is reached.

tty

This file defines predicates to read objects from a terminal. The input and output streams are left unchanged on returning. The definitions include:

ttynl/0

Produces a newline on the stnadard output stream (user).

ttyget0/1

ttyget0(_Ch)

arg1 : free

Same as **get_char0**, except from the standard input stream (user).

ttyget/1

ttyget(_Ch)

arg1 : free

As for **ttyget0/1**.

ttyput/1

ttyput(_Ch)

arg1 : integer

Same as **put/1** except that the output is to the standard output stream (user).

ttytab/1

ttytab(_V)

arg1 : free

Outputs a tab on the standard output stream (user).

ttyflush/0

ttyflush

Flushes the standard output stream (user).

ttyskip/1

ttyskip(_Ch)

arg1 : char

Skip chars on the standard input stream (user).

ttyread/1

ttyread(_Term)

arg1 : free

Reads from the standard input stream (user).

ttyread/2

ttyread(_Term, _VarNames)

arg1 : free
arg2 : free

As **read_term/2** but from the standard input stream (user).

tty_readline/1

tty_readline(_Line)

arg1 : free

Reads a line terminated by a newline from the standard input stream.

tty_message/1

tty_message(_Message)

arg1 : free

Sends a message to the standard output stream (user).

copy_stream

Copies characters from one stream onto the output stream, stopping when EOF is reached.

copy_stream/0

copy_stream

Copies from input to output until EOF is reached.

append_contents/1

append_contents(_File)

arg1 : ground : atom

Appends the contents of the file *arg1* to the current output steam.

2.7 ndbm

ndbm

The following package provides a very limited interface to the ndbm(3) package of SunOS. Ndbm maintains key/content pairs in a database. Both the key and the contents may be of arbitrary size. The basic set of operations are closely related to the record/recorded operations which are part of ***ProLog by BIM***, but which only operate on the internal record database (and so are not persistent). Access to these routines is only via keys, which must be a unique combination.

These routines should only be used in a number of cases:

- where there is too much information to be placed in the record database area of ***ProLog***,
- where the data is required to persist from one session to the next, but where corruption of the data is not important if the system crashes.

ndbm(3), does not have a form of protection against crashes. If the system crashes, the data might well be lost. Use ClauseDB if such security is needed or access on non-keys is required.

ndbm_open/4

ndbm_open(_DBName, _Flags, _Mask, _DBIdentifier)

arg1 : ground : atom
arg2 : ground : integer
arg3 : ground : integer
arg4 : free : dbidentifier

Opens the database indicated by *arg1*, according to flags *arg2* and masks *arg3*. A database identifier is returned which is used in subsequent operations. The values which might be given to *arg2* and *arg3* can be found in the documentation for ndbm(3).

ndbm_record/3

ndbm_record(_DBIdentifier, _Key, _Record)

arg1 : ground : dbidentifier
arg2 : ground : atom
arg3 : free : term

Records the contents of term *arg3* under key *arg2*. If the record already exists under that key, then the record operation will fail.

ndbm_rerecord/3

ndbm_rerecord(_DBIdentifier, _Key, _Record)

arg1 : input : dbidentifier
arg2 : input : atom
arg3 : input : atom

The previous record identified by *arg2* in the database identified by *arg1* is replaced with the record indicated by *arg3*. The previous record value is lost if it existed.

ndbm_recorded/3

ndbm_recorded(_DBIdentifier, _Key, _Record)

arg1 : ground : dbidentifier
arg2 : ground : atom
arg3 : free : term

Retrieves the record associated with *arg2* from the database indicated by *arg1*. The predicate fails if the record does not exists, or *Arg3* and the record are not unifiable.

ndbm_close/1

ndbm_close(_DBIdentifier)

arg1 : ground : dbidentifier

Closes the database associated with *arg1*.

ndbm_erase/2

ndbm_erase(_DBIdentifier, _Key)

arg1 : ground : dbidentifier
arg2 : ground : atom

Erases any record from the database which is associated with *arg2* in database *arg1*.

ndbm_current_key/2

ndbm_current_key(_DBIdentifier, _Key)

arg1 : grund : dbidentifier
arg2 : free : variable or atom

Retrieves a key *arg2* from database *arg1*. Keys are not retrieved in any particular order.

2.8 menus

menu

Documentation on this libraries can be found in the corresponding source file.

dir_menu

Documentation on this libraries can be found in the corresponding source file.

curses

Documentation on this libraries can be found in the corresponding source file.

Programming Tools And Scripts

Programming Tools and Scripts

Chapter 1
Tools

The directory **$BIM_PROLOG_DIR/src/tools** contains the sources of Prolog tools. These tools can be consulted either by specifying them on the command line:

```
% BIMprolog -Ltools/<toolname>
```

or by the query:

```
?- lib(<toolname>) .
```

1.1 plint

This tool can be consulted with:

```
?- lib(plint) .
```

The directory contains a lint checker for spotting problems, such as undeclared predicates, useless clauses, misspellings of variable names and singleton variables.

plint/1

plint(_Files)

Arg1 is the list of files to be checked.

1.2 ptags

This tool can be consulted with:

```
?- lib(ptags) .
```

The directory contains a program to generate "tags" files for use with the "vi" or "emacs" text editors. If the program is given a directory name, it will generate a tags file for all the subdirectories and the Prolog files found within them.

For a description of the use with GNU-Emacs**, see the section describing the emacs mode available with ***ProLog by BIM***.

The "tags" file is used by "vi" with the "-t" option to find the file in which a predicate is held.

ptags/2

ptags(_Args, _Files)

Arg1 is the list of arguments and *arg2* is the list of file names or directories to be searched for Prolog files to be tagged.

1.3 timing

This directory contains goal timing predicates, returning the elapsed time rather than processor time. Note that elapsed time is system load dependent.

getelapsedtime

This tool can be loaded with:

```
?- lib(getelapsedtime) .
```

time_command/2

time_command(_Goal, _Time)

arg1 : partial : term (goal)
arg2 : free : integer

Arg2 is instantiated to the time (in milliseconds) taken to execute the goal *arg1*.

timing

This tool can be loaded with:

```
?- lib(timing) .
```

elapsed_time/1

elapsed_time(_Time)

arg1 : free : (pointer: pointer)

As **cpu_time/1**, *arg1* is instantiated to the number of seconds and microseconds since the epoch, Jan. 1, 1970, 00:00, GMT.

elapsed/1

elapsed(_Goal)

arg1 : partial : term (goal)

As **time/1**, the predicate gives the elapsed time in seconds on stdout (integer: integer).

elapsed/2

elapsed(_Goal,_Time)

arg1 : partial : term (goal)
arg2 : free : (integer: integer)

As **time/2**, *arg2* is instantiated to the elapsed time in seconds taken to execute the goal *arg1*.

1.4 tracing

medic

This tool can be loaded with:

```
?- lib (medic) .
```

Gives the user an easy way to find mode errors. It provides a new consulting routine called "mode_chk", which reads a file.

mode_chk/1

mode_chk(_File)

The effect of this is similar to the effect of **reconsult_predicates/1**. When a mode conflict is detected, an error message is printed and the system stops on a break. Typing ^Z resumes program execution.

Error detection can be enabled for each predicate.

To disable the detection for a predicate, use the command:

well/2

well(_Functor, _Arity)

To enable the detection again, use:

sick/2

sick(_Functor, _Arity)

1.5 inspecting

This directory contains programs for inspecting Prolog source files. ***Vcheck*** searches for singleton variables in a Prolog source file and ***count*** displays a "count" of the predicates.

count

This tool can be consulted with:

```
?- lib(count) .
```

Provides the user with information (number of predicates and number of clauses in each predicate) about a valid Prolog file. Any consulted file is also considered.

count/1

count(_ListOfFiles)

Arg1 are the files to be counted.

vcheck

This tool can be consulted with:

```
?- lib(vcheck) .
```

This file defines a predicate **vcheck(-File)** which may be used as a debugging tool. It reads the clauses of the file, one at a time, and reports clauses that contain unique variables.

vcheck/0

vcheck

Requests the file name and then checks it for problems.

1.6 xref

This tool can be consulted with:

```
?- lib (xref).
```

This is a cross referencing program for Prolog code. See the XREF.HLP file in this directory for information on how to use the program.

1.7 profiler

This tool can be consulted with:

```
?- lib (profiler) .
```

This program allows the generation of run-time call statistics. The program to be profiled must be *consulted* using the predicate **profile/1** and the results can be obtained at the end of execution of the program by simply issuing a command. Certain functions have not been taken into account. These are index, dynamic and module declarations. The program produces a number of tables of statistics:

- **Procedure call information**:
  This table consists of a list of predicate name/arity together with the number of times each predicate was called. The list is ordered by the number of calls, starting with the highest number.
- **Procedure clause call information**:
  This table shows which clauses in each procedure were entered and how many times. It also shows the number of times that clauses succeeded, failed and are backtracked into.
- **Prolog built-ins call information**:
  This is similar to table 1, except that the predicates described are the ***ProLog by BIM*** built-ins.
- **Procedure call instantiation information**:
  Shows for each predicate the number of times it was called with each Prolog attribute type; that is, the number of times a particular argument was an integer, a variable and a list.
- **Predicate cost estimates**:
  Produces a cost estimate for each predicate based on a static analysis of the cost of entering a particular clause (that is, the cost of unifying the heads). The predicates are sorted according to the estimate of the total cost for each predicate. These estimates are very rough and should be considered carefully when modifying a program.
- **Missing mode declarations list**:
  Provides a list of mode declarations which are either different or are not defined. If the results where gathered without the full range of test data, then the assessments could be inaccurate.

Take care when evaluating the program performance with these results, because the program may operate very differently if the input test data is changed.

The file defines the following predicates:

profile/1

profile(_FileName)

arg1 : ground : atom

This predicate loads the Prolog file for profiling. It generates an intermediate file which will be placed in the directory, identified by the shell variable TMP_DIR (or the current directory if it is not defined). The temporary file will be deleted once reconsulted. This predicate acts in a similar way to **reconsult_predicates/1**, but *arg1* must be the full file name and path name of the user's Prolog file. Use the **please/2** option to change the syntax which will be accepted by this predicate (native syntax is the default).

show_results/1

show_results(_FileName)

arg1 : ground : atom

This predicate has to be called after execution of the program to be profiled. The results are sent to the file *arg1*. For displaying the results on the standard output stream use, **show_results/0**.

show_results/0

show_results

This predicate has to be called after execution of the program to be profiled. The results are written to the standard output stream.

Example

Taking the triangles puzzle from the library examples directory. Consulting the program for profiling using the call:

```
?- profile(
   'Libraries/bim_prolib/demos/puzzles/triangles.pro').
compiling file for profiling -
Libraries/bim_prolib/demos/puzzles/triangles.pro
compiling /bim35/rca/pam/.tmp/tmpnam.xxx0
consulted tmpnam.xxx0
Yes
```

Then call the program as usual.

```
?- arrange(_S, 17).
 _S = [side1(3,8,4,2),side2(2,9,5,1),side3(1,7,6,3)]
Yes
```

Put the results of the execution in a file.

```
?- show_results('/tmp/file').
Yes
```

The results file will then have the form:

********* Predicate List

Called	Name/Arity
42493	fill_remainder/5
35062	remove_nos/3
2045	fill_in_sides/3
448	choose_number/3
2	create_sides/1
2	allocate_points/3
1	arrange/2

********* Predicate Clause Call Information

arrange/2

ClauseNo	Entered	Unifie	Failed	Succeeded	Redo	Cost
1	1	1	0	1	0	2

allocate_points/3

ClauseNo	Entered	Unified	Failed	Succeeded	Redo	Cost
1	2	2	1	385	384	6
2	1	1	0	385	384	7

create_sides/1

ClauseNo	Entered	Unified	Failed	Succeeded	Redo	Cost
1	1	1	0	1	0	40

choose_number/3

ClauseNo	Entered	Unified	Failed	Succeeded	Redo	Cost
1	385	385	0	385	0	1155
2	63	63	60	1155	1152	441

remove_nos/3

ClauseNo	Entered	Unified	Failed	Succeeded	Redo	Cost
1	27285	27285	0	27285	0	190995
2	27276	27276	27251	42601	42576	300036

fill_in_sides/3

ClauseNo	Entered	Unified	Failed	Succeeded	Redo	Cost
1	385	385	384	1	0	2695
3	1659	1659	1656	3	0	11613

fill_remainder/5

ClauseNo	Entered	Unified	Failed	Succeeded	Redo	Cost
1	12337	1275	0	1275	0	61685
2	30156	30156	30150	2550	2544	271404
3	30150	30150	30144	2550	2544	271350

********* Prolog Builtins Call Information

Called	Name/Arity
60306	</2
40834	is/2
21719	atomic/1
21713	var/1
1660	=../2
1	=/2

********* Prolog Argument Call Argument Instantiation Information

arrange / 2

argno	functor	real	int	atom	list	[]	var
1	0	0	0	0	0	0	1
2	0	0	1	0	0	0	0

allocate_points / 3

argno	functor	real	int	atom	list	[]	var
1	0	0	0	0	1	0	1
2	1	0	0	0	0	0	1
3	0	0	1	0	0	0	1

create_sides / 1

argno	functor	real	int	atom	list	[]	var
1	0	0	0	0	1	0	1

choose_number / 3

argno	functor	real	int	atom	list	[]	var
1	0	0	0	0	448	0	0
2	0	0	0	0	63	385	0
3	0	0	0	0	0	0	448

remove_nos / 3

argno	functor	real	int	atom	list	[]	var
1	0	0	0	0	0	0	35062
2	0	0	0	0	27285	7777	0
3	0	0	0	0	0	0	35062

fill_in_sides / 3

argno	functor	real	int	atom	list	[]	var
1	385	0	0	0	1659	1	0
2	0	0	0	0	2044	1	0
3	0	0	2045	0	0	0	0

fill_remainder / 5

argno	functor	real	int	atom	list	[]	var
1	0	0	0	0	30156	12337	0
2	0	0	42493	0	0	0	0
3	0	0	0	0	41171	1322	0
4	0	0	0	0	0	0	42493
5	0	0	42493	0	0	0	0

********* Prolog cost estimates

Cost	Name/Arity
81855	remove_nos/3
61685	fill_remainder/5
1155	fill_in_sides/3
1155	choose_number/3
6	allocate_points/3
2	arrange/2
1	create_sides/1

********* Missing Mode Declarations

```
:- mode arrange(o,i) .
:- mode choose_number(?,?,o) .
:- mode remove_nos(o,?,o) .
:- mode fill_in_sides(?,?,i) .
:- mode fill_remainder(?,i,?,o,i) .
```

Here, for example, it can be seen that **fill_remainder/5** is a predicate that contains a large amount of backtracking and is called the most often.

1.8 deterministic

A tool which determines the deterministic behavior of a program.

This tool can be consulted with:

?- lib(deterministic) .

The program to be profiled must be *consulted* using the predicate **check_deterministic/1** and the results can be obtained at the end of execution of the program by simply issuing a command. This predicate does not take the extended indexing into account.

The file defines the following predicates:

check_deterministic/1

check_deterministic(_FileName)

arg1 : ground : atom

This predicate loads the Prolog file for determinicy checking. It generates an intermediate file which will be placed in the directory, identified by the shell variable TMP_DIR (or the current directory if it is not defined). The temporary file will be deleted once reconsulted. This predicate acts in a similar way to **reconsult_predicates/1**, but *arg1* must be the full file name and path name of the user's Prolog file. Use the **please/2** option to change the syntax which will be accepted by this predicate (native syntax is the default).

show_results/1

show_results(_FileName)

arg1 : ground : atom

This predicate has to be called after execution of the program to be profiled. The results are sent to the file *arg1*. For displaying the results on the standard output stream use, **show_results/0**.

show_results/0

show_results

This predicate has to be called after execution of the program to be profiled. The results are written to the standard output stream. The different sections of these results can also be obtained with the following predicates:

show_non-determinstic/0,
show_non_determinstic_children/0,
show_determinstic_after_succes/0,
show_determinstic/0 and show_diff/0 .

show_results_table/1

show_results_table(_Filename)

arg1 : ground : atom

Same as show_results/1 but in a different format.

show_results_table/0

show_results_table

Same as show_results/0 but in a different format.

Example

```
-option('c+').
        /*****************************************************/
        /*deterministic.example.test :                       */
        /*contains some example for the deterministic tools*/
        /*****************************************************/
a([],[]):-!.
a([_H|_T],[_Ht|_Tt]):-
        a(_H,_Ht),
        a(_T,_Tt).
a(_I,nonlist(_I)):-
        _I=[_|_].


c(_I,_I).
c(_I,[_I|_I]).

b(_L):-write(_L),nl.

d(_LI,_EI):-
        c(_EI,_E),
        a([_E|_LI],_L),
        b(_L).

?-d([1,2,3],a).
```

% BIMprolog

The tool is loaded with: -L tools/deterministic

```
?- check_deterministic('deterministic.test').
compiling file for determinicy testing - deterministic.test
compiling /bim80/prolog/lm/PROLOG_LIB/tmpnam.xxx52
reloaded tmpnam.xxx52
[nonlist(a),nonlist(1),nonlist(2),nonlist(3)]
[[nonlist(a) | nonlist(a)],nonlist(1),nonlist(2),nonlist(3)]
?- show_results_table.
alternatives left after:
        head    success name/arity
        no      no      'b'/1
        no      yes     'd'/2
        yes     no      'a'/2
        yes     yes     'c'/2
Yes
?- show_non_deterministic.
Non-Deteministic Predicates: alternative remain after success
        Name/Arity
        'd'/2
        'c'/2
Yes
?- show_deterministic.
Deterministic Predicates: no choicepoint is placed
        Name/Arity
        'a'/2
        'b'/1
Yes
```

```
?- show_deterministic_after_success.
Deterministic Predicates:
        Name/Arity
        'b'/1
Yes
?- show_non_deterministic_children.
Non-Deterministic Predicates: alternative for subgoals remain
after success
        Name/Arity
        'd'/2
        'c'/2
Yes
?- show_results.
        50% of program called non-deterministically
Non-Deterministic Predicates: alternative remain after
success
        Name/Arity
        'd'/2
        'c'/2
Deterministic Predicates: no choicepoint is placed
        Name/Arity
        'a'/2
        'b'/1
Yes
```

1.9 beautify

The beautifier for Prolog source files enhances the readability of Prolog programs.

The options that can be specified are:

indentation/1
Arg1 specifies the indentation (default 3).

blanks_after_directive/3
Arg1 specifies the blank lines between the same directives (def: 0).
Arg2 specifies the blank lines between two different directives (def: 1).
Arg3 specifies the blank lines between a directive and a non-directive (def: 2).

blanks_after_predicate/3
Arg1 specifies the blank lines between clauses of the same predicate (def: 1).
Arg2 specifies the blank lines between two different predicates (def: 2).
Arg3 specifies the blank lines between a predicate and a non-predicate (def: 2).

blank_term_depth/1
Arg1 specifies the levels that have blanks between arguments (def: 1).

or_join/1
Arg1 specifies the **or_join** behavior. Possible values are 0, 1 or 2 (def: 0).

join/1
Arg1 specifies the subgoals that will be written on the same line as the previous subgoal (default: !, nl, nl(_), tab(_), tab(_,_), spaces(_), spaces(_,_)).

The predicates are:

fbeautify/2

fbeautify(_SourceFile, DestFile)

arg1 : ground : atom : physical source file name
arg2 : ground : atom : physical beautified file name

Beautifies the source file and writes it on the destination file. The '*.pro*' extension may be omitted in *arg1* and *arg2*.

fbeautify/1

fbeautify(_SourceFile)

arg1 : ground : atom : physical source file name

Beautifies the source file and writes it on the destination file with the same name as the source but with ".b" inserted before the '*.pro*' extension. The '*.pro*' extension can be omitted in *arg1*.

fbeautify_write/3

fbeautify_write(_FileName, _Term, _VarNameList)

arg1 : ground : atom : logical beautified file name
arg2 : any : term : structure
arg3 : ground : list of (_var=atom) : list of variable names

The structure *arg2* is written in a "beautiful" form on the logical file *arg1*. Its variables will be written out with names as defined in list *arg3* (this is typically the result of a **vread**).

set/0

set

Lists the current option settings.

set/1

set(_OptionValue)

arg1 : ground : term : option value

If *arg1* is a viable option, it is set. For a single value option the previous value is replaced by the new one. For a multiple value option, the new value is added to the previous set.

unset/1

unset(_OptionValue)

arg1 : ground : term : option value

If *arg1* is a viable multiple value option, it is removed from the value set.

1.10 emacs

This directory contains an updated version of the "prolog-mode", which comes with GNU-emacs. In addition to this, some additional features have been added (some of which are enhancements of other public domain functions). The mode is loaded into GNU-emacs either by loading the file *prolog.el* via the emacs startup file or by using the GNU-emacs "load-file" command. More information on the emacs mode can be found in the emacs info file in %BIM_PROLOG_DIR/help or in the ps file $BIM_PROLOG_DIR/PS/emacs.ps.

Programming Tools and Scripts

Chapter 2
Scripts

Scripts are designed to help porting programs or identify specific problems in various syntax modes or Prolog systems. No guarantee can be given that the scripts can be used to solve all problems. Before amending the scripts, it is recommended to keep a backup copy of the original file.

The scripts are in the directory $BIM_PROLOG_DIR/scripts.

2.1 ChangeDirectives

Changes the "mode" and "public" declarations. Note that the declarations in ***ProLog by BIM*** are much stronger than those in DEC-10 Prolog, so run-time problems may still occur, though these will be mostly warning messages.

2.2 ChangeModules

Changes the module notations as used in SWI-Prolog** (University of Amsterdam, The Netherlands). It might also be useful for other Prolog implementations.

ChangeModules changes the **module/2** declarations into **module/1** and **global/2** directives. It also changes the explicit module qualifications from the colon format:

<module-name>:<predicate><args>

to the dollar notation:

<predicate name>$<module-name><args>

Example

```
:- module( example,[ module_example/3 ]).
                              % Comment module_example

module_example(_ModName, _Arg1, _Arg2) :-
     _ModName:my_pred(_Arg1, _Arg2).
```

is transformed in:

```
:- module( example ).
:- global  module_example/3  .     % Comment module_example
module_example(_ModName, _Arg1, _Arg2) :-
   module(_my_pred_ModName_modqual, _ModName,
          my_pred(_Arg1, _Arg2) ),
   _my_pred_ModName_modqual.
```

2.3 Change_FromDEC10

Changes some predicate calls into their ***ProLog by BIM*** equivalents. The predicates changed are the most obvious ones, and generally the conversion at this point is done only for predicates that have the same function but have different predicate names. Problems may still occur if the argument forms are not the same. Variables and real numbers are also changed.

Converts the following:

floor(	into	trunc(
Var is cputime	into	cputime(_Var)
see(user)	into	see('/dev/tty').
tell(user)	into	tell('/dev/tty').
close(	into	fclose(
tab(	into	spaces(
open(	into	dec10_open(
ttyflush	into	flush

2.4 ChangeOrPipes

Changes pipe characteristics outside lists. This form of disjunction is used in DEC-10 Prolog.

2.5 ChangeSyntax_FromCompat

Changes files that compile in compatibility mode to files that compile in native mode. Changes comment forms and variable forms. Variables are renamed to have an underscore as their first character. This program will terminate if DCG clauses are detected. DCG clauses can only be compiled in DEC-10 syntax mode, so there is no point in continuing to further translate the file.

2.6 ChangeExtern_FromForeign

Changes the external language declarations. This is specifically aimed at translating the Quintus Prolog** external language interface directives into extern_declarations suitable for the ***ProLog by BIM*** interface.

2.7 ChangeC_FromQP

Changes the C-to-Prolog string routines from Quintus Prolog (Mountain View, USA) into the corresponding ***ProLog by BIM*** procedures.

2.8 ChangeC_FromPbB2.5

This is a filter to convert C programs written for ProLog by BIM version 2.5.

2.9 Change_FromPbB3.0

Lex file to change the main differences with ProLog by BIM version 3.0.

2.10 SpotPitfalls

Signals potential problems where predicates may have different actions or return different values in ***ProLog by BIM***. Obviously, there are many possible differences. This program only spots the more common ones. Where a difference is detected, the goal call and its line number are displayed.

2.11 SpotDiffs_WithPbB3.0

Signals potential problems in Prolog source code written for ***ProLog by BIM*** version 3.0. The offending goal call and the line number will be displayed.

2.12 SpotDiffs_WithPbB3.1

Signals potential problems in Prolog source code written for ***ProLog by BIM*** version 3.1. The offending goal call and the line number will be displayed.

2.13 SpotUsage _InternalDB

Flags all calls to update the internal database of ***ProLog by BIM***. The goal call and the line number are displayed on the standard output stream.

2.14 SpotUsage _ClauseDB

Spots calls to the low-level ClauseDB routines. Each identified call is displayed along with the line number associated with it.

2.15 SpotUsage _PTB

Spots calls to the low-level PTB routines. Each identified call is displayed along with the line number associated with it.

2.16 Full_conversion

A shell script which combines the above "change" lex script programs.

Appendix

Appendix

Appendix

Chapter 1
Messages from the Engine

1.1 Error classes

There are seven classes of error messages in the ***ProLog by BIM*** system:

SYNTAX	syntax error message
SEMANTIC	semantic error message
WARNING	warning message
OVERFLOW	overflow message
BUILTIN	incorrect use of built-in predicate
RUNTIME	run-time error message
MODE	incorrect use of mode declaration

Two extra classes are reserved to implementation error messages: **INTERNAL** and **PANIC**. Any occurrence of such messages should be reported to BIM, if possible with the context in which the error occurred.

1.2 Error format

The error message format is:

```
*** <class> <error_number> *** <error_message>
```

Example

```
*** BUILTIN 483 *** attempt to divide by zero
```

1.3 Warnings

Warnings are the only error messages which do not stop the process of parsing, compilation or execution of a program. However, a warning issued from a directive means that this directive has been ignored.

1.4 Error ranges

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
240-299	Compiler errors (only from BIMpcomp)
300-599	Built-in errors
600-699	Engine errors
700-799	Table errors
800-899	External language interface errors
900-999	Messages from the debugger

1.5 Error listing

Error messages appear on the screen by default. An option exists for the ***ProLog by BIM*** compiler (the -l option) to print error messages in a listing file (see "*Principal Components - The Compiler*". Error messages from the engine can be redirected by using the **tell_err/1** predicate and its variants (see "*Built-in Predicates - Input-Output*").

Parser warnings

100	Identifier too long, truncated to %1 characters.
101	Illegal operator type ; directive ignored.
102	Illegal operator precedence ; range is from 1 to 1200 ; directive ignored.
107	Non-existing directive.
108	Illegal operator name in directive.
109	Illegal procedure name.

110 Illegal arity.

112 Illegal module name.

113 Specify the argument for a dynamic declaration as name/arity ; directive ignored.

114 Only one module declaration allowed.

115 Possible erroneous use of atom nil instead of [].

120 Local redefinition of a built-in predicate overrides standard definition.

121 Atom %1 starts with upper case.

124 Illegal mode declaration ; directive ignored.

Parser overflows

133 The arity of the structure is too high (maximum is 255).

190 Not enough resources to expand operator stack.

191 Not enough resources to expand operand stack.

208 Too many operator definitions.

221 Not enough resources to expand token stack.

Parser syntax

140 This module qualification is not allowed.

192 Digit expected after point in real number.

193 Digit expected in exponent of real number.

194 Illegal character inside quoted string.

195 Illegal character.

196 Term expected.

197 "," or ")" expected.

198 "]" expected.

199 Unexpected character.

200 End of line character inside quoted string.

201 Operand expected, operator found.

206 String too long.

207 Number too long.

211 Unexpected end of file inside comment.

212 Infix or postfix operator expected, prefix operator found.

213 Precedence of prefix operator too high use parentheses.

214 Precedence of infix operator too high use parentheses.

215 Precedence of postfix operator too high use parentheses.

216 Precedence too low use parentheses.

217 Ambiguous combination of operators use parentheses.

218 Conflicting operator types.

219 Integer too small or too big.

220 Real too small or too big.

224 Unexpected end of file.

225 "}" expected.

226 ")" expected.

227 Invalid pointer.

Parser semantic

130 Too many include files.

131 Include file could not be opened.

141 Invalid import declaration.

142 Invalid local directive.

143 Local declaration conflicts with previous declarations.

144 Invalid module qualification.

145 Module qualification needed.

146 Invalid global declaration.

150 Invalid dynamic declaration.

152 Mode declaration not allowed: name conflict.

160 Invalid index declaration.

162 Index declaration not allowed: name conflict.

170 Invalid extern directive.

172 A return parameter must be the first argument.

173 Wrong type combined with output or return mode.

174 No extern_load declaration for function or predicate.

175 External declaration not allowed: name conflict.

176 The routine list must be a list of atoms.

177 The load module list must be a list of atoms.

178 Invalid extern_function declaration.

179 Invalid extern_predicate declaration.

180 Invalid identifier in xtern_load declaration.

181 Duplicate declarations of an external function must be grouped.

182 Invalid extern_builtin declaration.

183 Unrecognized language specifier.

202 Attempt to redefine a built-in predicate or an external predicate.

203 Variable not permitted as head of a clause.

204 Integer not permitted as head of a clause.

205 Real not permitted as head of a clause.

209 Integer not permitted as goal of a body.

210 Real not permitted as goal of a body.

228 Pointer not permitted as head of a clause.

229 Attempt to redefine a static predicate.

230 Illegal dcg rule.

231 Pointer not permitted as goal of a body.

Compiler errors

240	This clause contains an uninstantiated metacall.
241	This clause needs too many registers.
242	This clause contains a call with too many arguments (maximum is 32).
243	The head of this clause has too many arguments (maximum is 32).
244	No query is allowed between the definitions of a predicate.
246	This clause cannot be asserted dynamically
247	This clause contains too many disjunctions (limit is 255).
248	This clause has too many variable names in it.
249	There are too many definitions of this predicate.
250	This program contains too many constants.
251	This program contains too many functors.
252	This clause is too long to translate.
253	This program contains too many procedures.
254	This file cannot be compiled: it is too long.
255	Illegal terminal for dcg: must terminate on [].
260	Module qualification needed on line %1 .
262	Illegal call in body.
263	This clause cannot be asserted dynamically
264	Error in assignment or comparison.
265	Missing external declaration for external function %1.
266	Useless unification or assignment containing %1.
267	Output argument %1 not free.
268	Input argument %1 not ground.
269	Real not permitted as goal of a body.
270	Integer not permitted as goal of a body.
271	Unable to allocate memory.
272	Illegal head of clause.
273	Pointer not permitted as goal of a body.
274	Definitions of a predicate should be contiguous.
275	Arithmetic expression too large too compile.
276	Always succeeding unification.
277	Always failing unification.
278	Named variable %1 occurs logically only once.
279	Odd instantiation in call to %1.
280	Division by zero.
281	Variable not instantiated in arithmetic expression.
282	Type error in arithmetic expression.
299	Read error %1.

Built-in errors

300	The %1 argument must be a free variable.
301	The %1 argument must be an integer.
302	The %1 argument must be a real.
303	The %1 argument must be a pointer.
304	The %1 argument must be an atom.
305	The %1 argument must be atomic (integer, real, pointer or atom).
306	The %1 argument must be free or an integer.
307	The %1 argument must be free or a real.
308	The %1 argument must be free or a pointer.
309	The %1 argument must be free or an atom.
310	The %1 argument must be free or atomic (integer, real, pointer or atom).
311	The %1 argument must be a []-terminated list.
312	The %1 argument must be a compound term (non-atomic).
313	The %1 argument must be an atom or a compound term.
320	The %1 argument must be instantiated.
325	The %1 argument must be a positive integer.
326	The %1 argument must be an integer or an atom.
330	The %1 argument must be a list of integers.
331	The %1 argument must be a list of single-character atoms.
332	The %1 argument must be a list of atomic (integer, real, pointer or atom) elements.
335	The %1 argument must be an integer or a list of integers.
336	The %1 argument must be an atom or a list of atoms.
337	The %1 argument must be atomic (integer, real, pointer or atom) or a list of atomic elements.
350	The %1 argument must be a predicate descriptor of the form name/arity or name(...).
351	Bad built-in descriptor argument. Expected name_arity.
360	At least one of the arguments must be instantiated.
390	Overflow in constructing a list for the %1 argument.
400	Meta-call : zero-arity functor %1() cannot be called.
401	Meta-call : integer %1 as goal non permitted.
402	Meta-call : real %1 as goal non permitted.
403	Meta-call : pointer %1 as goal non permitted.
404	Uncallable predicate.
405	Illegal call : unknown predicate %1/0.
406	Illegal call : unknown predicate %1.
407	Execution stopped because argument %2 not ground.
408	Bad argument type.
410	No clause/retract operations allowed on static predicate %1.
411	No clause/retract operations allowed on built-in predicate %1.
412	No clause/retract operations allowed on predicate %1.

413 Functor %1 is not a dynamic predicate and cannot be updated.

414 Clause for predicate %1 too big to decompile.

415 Can only retrieve predicates from real files - not from user.

419 Illegal DCG rule.

420 Atomic term %1 cannot be asserted as data base predicate.

421 Predicate %1 already has a definition and cannot be asserted as data base predicate.

422 Variable not permitted as head of a clause.

423 Number (integer, real or pointer) not permitted as head of a clause.

424 Attempt to assert built-in predicate %1.

425 Attempt to assert the static predicate %1.

426 Attempt to assert external predicate %1.

427 Number (integer, real or pointer) not permitted as goal.

428 Wrong variable-name association list.

430 A clause source file specification must be given as Path/Name.

431 Source file %1 is not loaded.

432 Library %1 in file %2 could not be loaded.

435 Bad clause reference %1.

440 Predicate %1 is not suitable for dynamic hashing.

441 Cannot change declaration of non-dynamic predicate %1.

442 Cannot change indexing of dynamic predicate %1 with definitions.

443 Cannot change mode of dynamic predicate %1 with definitions.

444 Cannot make existing predicate %1 dynamic.

450 The argument descriptor and recorded term have an incompatible structure.

451 The argument descriptor must be a flat list of integer indices.

452 The key(s) must be instantiated.

453 Key %1 is already in use.

454 Key %1 is already in use.

455 Add to non-existing key %1 is not allowed.

456 Add to non-existing key %1 is not allowed.

457 Heap overflow. (Table H)

458 Key pair %1,%2 is already in use.

459 Key pair %1,%2 is already in use.

460 Key pair %1,%2 is already in use.

461 Key pair %1,%2 is already in use.

462 Add to non-existing keys %1 and %2 is not allowed.

463 Add to non-existing keys %1 and %2 is not allowed.

464 Add to non-existing keys %1 and %2 is not allowed.

465 Add to non-existing keys %1 and %2 is not allowed.

470 The arguments must be a pointer, an integer and a pointer. Only one may be free.

475 Arithmetic expression contains an infinite term.

476 Integer result overflow.

477 The %2 argument of function %1 must be an integer.

478 The %2 argument of function %1 must be a positive integer.

479 The %2 argument of function %1 must be a real.

480 The %2 argument of function %1 must be an integer or a real.

481 The %2 argument of function %1 must be an atom.

482 Function %1 : a pointer offset must be an integer.

483 Integer overflow in function %1.

484 Division by zero in function %1.

485 Function %1 : the first argument must be less than the second.

486 Function %1 : the constructed atom is too long (%2 characters).

487 Function %1 : out of range.

488 Arithmetic comparison is only allowed between pointers or a combination of integers and reals.

489 Illegal computable expression : free variable or list or structure.

490 Arithmetic expression evaluation must result in integer or real.

491 Arithmetic expression in %1 argument must be integer or real.

495 Non-integer argument in compiled arithmetic expression (operator %1).

496 Integer overflow in compiled arithmetic expression (operator %1).

497 Division by zero in compiled arithmetic expression (operator %1).

498 Non-arithmetic result in compiled arithmetic expression.

499 Non-arithmetic argument in compiled arithmetic expression (operator %1).

500 The mode for opening %1 must be w, r or a.

501 Use an atom as logical filename or a pointer.

502 Logical file %1 not in use.

503 Logical file %1 is already in use.

504 Use an atom as current stream name or a pointer.

505 Unable to open %1 : too many files open already.

506 Cannot open %1.

507 Standard file %1 cannot be closed.

508 File %1 is open for output, not for input.

509 File %1 is open for input, not for output.

510 Variable name list must be []-terminated list of (name=_var).

520 The second argument must be a global atom.

521 The third argument must be a compound term with a principal functor belonging to the global module.

525 The second argument must be a []-terminated list with at most %1 elements and first element an atom.

530 The second argument of op/3 must be an atom of fx, fy, xfx, xfy, yfx, xf, yf.

531 The first argument must be an integer in the range 1..1200 or free, the second an operator type and and the third an atom.

535 The first argument must be free or a single character.

540 An atom to be constructed is too long (%1 characters).

541 The %1 argument must be a list of characters, not longer than the longest atom allowed.

542 The %1 argument must be a list of integer character codes, not longer than the longest atom allowed.

543 At most one argument may be free, the others must be atoms.

544 One of the arguments must be an atom and the other free.

545 The given or calculated start position and length arguments must be positive.

550 The first argument must be a symbolic signal name.

551 The second argument must be one of status/2, ignore, accept, suspend, status, raise or clear.

552 The arguments of status/2 must be free.

553 The signal argument must be a signal name or a list of signal names.

554 A Prolog handler was installed outside of the system"s control.

555 An external handler was installed outside of the system"s control.\nThe previous Prolog handler was %1.

560 Memory fault : too many arguments (%1) for program.

565 Command line argument numbers are in the range from 1 to %1.

570 %1 is not an acceptable key ; the first argument must be one of : B C D F H I R S T.

571 %1 is not an acceptable key ; the first argument must be one of : serial version copyright owner distributor.

590 Except for this one, no error occurred previously.

591 Error in loading of error set : %1.

592 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.

593 Error number %1 is out of defined error ranges.

594 Too many error arguments. Ignoring overflow.

595 Too few error arguments. Missing %1.

596 Non-expected type for %1 error argument.

Engine errors

600 File %1 non-existent or not a regular file.

601 Error during compilation of %1.

602 Cannot execute ProLog compiler in file %1.

605 Loader error : not enough space to dynamically allocate the temporary table of functors.

606 Loader error : not enough space to dynamically allocate the temporary table of constants.

607 Loader error : not enough space to dynamically allocate the temporary table of modules.

610 Loader error : try to load built-in predicate %1.

611 Loader error : try to load data base predicate %1.

612 Loader error : try to add to static predicate %1.

613 Loader error : try to add dynamic clauses to defined non-dynamic predicate %1.

614 Loader error : try to add to external predicate %1.

615 Loader error : try to add incompatible code type (debug) clauses to dynamic predicate %1.

616 Loader error : try to add incompatibly indexed clauses to dynamic predicate %1.

617 Loader error : try to add incompatible mode clauses to dynamic predicate %1.

618 Specify file as Path/Name or as a single atom (with options).

630 Object file %1 is incompatible with ProLog (wrong version or machine type).

631 Unable to open object file %1.

632 Unable to refind the previous position in object file %1.

633 Skipping bad compiler option %1.

634 File %1 cannot be consulted with this system.

640 Restore option needs a file name to restore from.

641 Problem in consulting pre-linked file %1.

642 Not enough memory to build up compiler command. Need %1 bytes.

643 Failed fork of program %1.

644 Unable to restart : no main control.

645 Unable to contact license server on host %1. Please contact your system administrator.

646 No licenses available. Please contact your system administrator.

647 Problem in calling license server: %1.

650 Unable to open file %1 for state saving.

651 Unable to open saved state file %1.

652 Initialized data file %1 not found.

660 File %1 has already been consulted ; source line information may be incorrect.

661 The %1 argument of %2 is not of type %3.

670 Input mode error in argument %1 of %2.

671 Output mode error in argument %1 of %2.

672 Undefined input argument in %1.

673 Non-undefined output argument in %1.

691 Not enough resources for starting the %1 window.

692 Could not fork off the %1 window.

693 Monitor window has died.

694 Debugger window has died.

695 Cannot execute monitor %1.

696 Cannot execute debugger %1.

Table errors

710 String table overflow (Fatal).

711 Hash table overflow (Fatal).

712 Module table overflow (Fatal).

713 Data table overflow (Fatal).

714 Functor table overflow (Fatal).

715 Internal data base key table overflow (Table option R).

716 Internal data base heap overflow (Table option B).

717 Operator table overflow (Fatal).

718 Predicate table overflow (Fatal).

719 Clause table overflow (Fatal).

720 Native predicate table overflow (Fatal).

721 Interpreted code overflow (Fatal).

722 Compiled code table overflow (Table option C).

723 Program source table overflow (Fatal).

724 Execution stopped : fatal overflow of the stack (Table option S).

725 Execution stopped : fatal overflow of the heap (Table option H).

730 Not enough resources for allocating or expanding the string table (requested %1 byte).

731 Not enough resources for allocating or expanding the hash table (requested %1 byte).

732 Not enough resources for allocating or expanding the module table (requested %1 byte).

733 Not enough resources for allocating or expanding the data table (requested %1 byte).

734 Not enough resources for allocating or expanding the functor table (requested %1 byte).

735 Not enough resources for allocating or expanding the internal data base tables (Table R, B) (requested %1 byte).

738 Not enough resources for allocating or expanding the predicate table (requested %1 byte).

739 Not enough resources for allocating or expanding the clause table (requested %1 byte).

740 Not enough resources for allocating or expanding the native predicate table (requested %1 byte).

741 Not enough resources for allocating or expanding the interpreted code table (requested %1 byte).

742 Not enough resources for allocating or expanding the compiled code table (Table C) (requested %1 byte).

743 Not enough resources for allocating or expanding the program source table (requested %1 byte).

744 Not enough resources for allocating or expanding the stack (Table S) (requested %1 byte).

745 Not enough resources for allocating or expanding the heap (Table H) (requested %1 byte).

746 Not enough resources for expanding infinite term handling table (requested %1 byte).

747 Not enough resources for allocating a new block of protected data (requested %1 byte). Data unit will not be protected.

748 Not enough resources for allocating a (new) predicate hash table (requested %1 byte). Predicate will not be (re)hashed.

750 Text table limits exceeded.

751 Data table limits exceeded.

752 Functor table limits exceeded.

753 Interpreted code table limits exceeded.

754 Native code table limits exceeded.

755 Stack limits exceeded.

756 Heap limits exceeded.

770 Not enough resources for allocating a new block of variable names (requested %1 byte). Not all variables will be returned.

771 Not enough resources for allocating the switch search tables (requested %1 byte).

780 A structure is too deeply nested for garbage collection.

781 A structure is too deeply nested.

External interface errors

800 Arg%2 of external predicate %1 is not of declared type %3, at call.

801 Uninstantiated input parameter for external predicate %1, arg%2.

802 Null-reference for arg%2 on exit from external predicate %1.

803 Untyped parameter of external predicate %1, arg%2 has illegal type.

804 Array parameter of external predicate %1, arg%2 must be a non-empty, []-terminated list or a structured term.

805 List parameter of external predicate %1, arg%2 must be a flat []-terminated list.

806 Unification of arg%2, declared as %3, has failed at exit from external predicate %1.

807 String:0 parameter of external predicate %1, arg%2 must be an atom at call.

808 Evaluation of arg%2 of external predicate %1 failed.

809 External routine %1 is not a predicate.

810 External routine %1 is not a function.

811 Calling a non-linked external predicate.

820 Trying to terminate or find a next solution for a non-active iteration.

821 External call of predicate %1 : too high arity %2.

822 External call of predicate %1, argument %2 : bad structure %3 for BP_T_BPTERM or BP_T_VOID.

823 External call of predicate %1, argument %2 : BP_T_STRINGS not allowed in structure BP_S_ARRAY.

824 External call of predicate %1, argument %2 : illegal mode %3.

825 External call of predicate %1, argument %2 : illegal type %3.

826 External call of predicate %1, argument %2 : untyped parameter has illegal type.

827 External call of predicate %1, argument %2 : wrong output term for BP_S_ARRAY.

828 External call of predicate %1, argument %2 : bad type %3 for BP_S_ARRAY output.

829 External call of predicate %1, argument %2 : could not allocate array of %3 elements.

836 Not enough memory for external call of predicate %1 (needed %2 bytes).

837 External output arguments cannot be protected during garbage collection.

838 Not enough memory for call of an external routine associated to predicate %1 (needed %2 bytes).

839 An atom to be constructed from a string is too long (%1 characters).

840 The string %1 could not be saved.

841	%1 : arg%2 : %3 is not an atom.
842	%1 : arg%2 : %3 is not a functor.
843	%1 : arg%2 : %3 is not a legal term or argument.
844	%1 : %3 is not a valid type for term or argument %2.
845	%1 : cannot retrieve argument %3 from term %2.
846	Too many external protected terms.
847	External protected terms will be destroyed.
848	Predicate arity %1 out of range [1..%2].
850	External module table overflow (Fatal).
851	Not enough resources for allocating or expanding the external module table (requested %1 byte).
852	Not enough resources for building an external module (requested %1 byte).
853	Unable to open process symbol table for dynamic linking. System message :\n%2
854	Unable to open symbol table of object %1 for dynamic linking. System message :\n%2
855	Parameter substitution of %1 to %2 (partially) failed.
856	External routine %1 not loaded. System message :\n%2
857	Unload of external module %1 failed ; no such module.
858	Compilation of link definition file failed ; return code %1.
859	Generation of linkable object failed ; return code %1.
860	Cannot write link definition source file %1.
861	Bad predicate declaration in %1 argument for interactive incremental link.
865	Bad number of reserved units in external backtrack point : %1.
866	%1 : Bad handle %2 for external backtrack point.
867	The %1 argument must be the (atom) name of a Prolog callback predicate or the (pointer) address of an external callback function.
870	Unrecognized language specifier : %1.
871	External function must be declared as function:type.
872	Bad argument declaration : %1.
873	First enter external predicate and function declarations.
874	First enter an extern_load specification.
875	No predicate or function declaration specified for external %1.
876	External routine %1 not specified in an extern_load.
880	There is no specification for the indicated type of the variable.
881	Expected value of type %1.
882	No external access possible to non-atomic data.
883	Bad variable identifier.
884	Bad variable selector.
885	Index variable selector impossible.
886	Field variable selector impossible.
887	Bad type for direct external memory access.
888	Not enough memory to allocate %1 bytes of external memory.
889	Bad enumeration value %1 for direct external memory access.

Debugger errors

900 Bad (list of) port name(s) in %1 argument.

901 Give predicates in the form "name/arity".

902 Unknown predicate %1/%2.

903 Unknown predicate %1.

904 Bad predicate specification : %1

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 (%1 lines recorded).

908 No trace has been recorded for analysis.

909 Too many subgoals in predicate to analyze.

910 Not enough memory to keep more than %1 lines in the trace (aborting keeptrace).

911 The file %1 does not contain the required source line information.

912 No executable code at line %2 in file %1.

913 Source file %1 has not yet been consulted.

914 Not enough memory to set more break points.

915 Too many defined aliases.

916 Too many user defined commands.

917 Bad argument type specifier : %1.

918 Bad argument type modifier : %1.

919 Cannot read source file %1.

920 No spies set on %1. Specify predicate as name/arity.

921 Predicate %1 is not compiled for debugging.

Appendix

Chapter 2
Bibliography

Bratko [I.].
Prolog Programming for Artificial Intelligence. Addison-Wesley Publishing Company, 1986.

Campbell [J.A.].
Implementations of Prolog. Ellis Horwood Ltd, 1984.

Clocksin [W.F.] and Mellish [C.S.].
Programming in Prolog. Springer Verlag, 1981.

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).

Gray [P.M.D.]. and Lucas [R.J.].
Prolog and databases. Ellis Horwood Ltd, 1988.

Hogger [C.J.].
Introduction to Logic Programming. Academic Press, 1984.

Kowalski [R.].
Logic for Problem Solving. Artificial Intelligence Series, North Holland, 1979.

Shapiro [E.].
Algorithmic Program Debugging. MIT Press, 1982.

Sterling [L.] and Shapiro [E.].
The Art of Prolog. Advanced Programming Techniques. MIT Press, 1986.

Appendix

Chapter 3
Reader's comments

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this it the form to use.

When you send your comments to BIM, you grant BIM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Please return this form to

BIM
Kwikstraat 4
B-3078 Everberg
Belgium

or your local ***ProLog by BIM*** distributor.

Name	Address
Company or Organization	Address
Phone	Address

ProLog by BIM

User's Guide

Description

This manual describes a number of Prolog programming techniques. It does not provide general answers, but concentrates on how to *solve problems with Prolog by BIM.*

The techniques that are described, are mainly explained with small examples. They only illustrate the one particular technique that is being explained.

Some of the techniques covered are general for Prolog programming and can be used with any Prolog system. Others, however, are specific to ***ProLog by BIM***. Either they are solved with ***ProLog by BIM***-specific features, not available in other systems, or they discuss ***ProLog by BIM***-specific features on their own.

Chapter 1 describes the distinction between program and data, the consequences of this difference and the built-in predicates that exploit these two aspects of Prolog clauses. The specific time/space trade-off is dealt with in depth.

Chapter 2 treats the relation between several term manipulation predicates, to what extent they are interchangeable and the trade-offs involved.

Chapter 3 deals with cyclic terms: when they are created, how they can be avoided, at what cost, and how they can be used safely to the programmer's advantage.

Chapter 4 contains many examples explaining the all-solution predicates. Alternatives to these predicates, mainly different data representation schemes, are illustrated.

Chapter 5 deals with the difficult topic of optimizing a Prolog program. Some of the described optimizations are generally applicable to any Prolog implementation; some are specific to ***ProLog by BIM***.

Chapter 6 illustrates the external language interface of ***ProLog by BIM*** by solving a typical problem in six different ways.

Chapter 7 covers the debugging tools of ***ProLog by BIM***. It presents the five port box model and gives some hints on how to use the source-oriented debugger and post-execution analyzer for rapid debugging. There are also some notes on customization of the debugger.

Audience

This document is intended for the advanced Prolog programmer who wants to get more out of ***ProLog by BIM***.

It assumes that the reader is familiar with Prolog and has at least a basic knowledge of programming in Prolog. Readers who do not have this background, are referred to the following texts:

L. Sterling & E. Shapiro, *The Art of Prolog*, MIT Press, 1986

C. J. Hogger, *Introduction to Logic Programming*, Academic Press, 1984

I. Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1986

F. Kluzniak & S. Szpakowicz, *Prolog for Programmers*, Academic Press, 1987

This manual does not provide an introduction to ***ProLog by BIM***. Information on how to get started with ***ProLog by BIM*** is found in the ***ProLog by BIM*** *Reference Manual.*

Typographical conventions

*Italic*s are used for:

- New terms when they are introduced.
- ***ProLog by BIM*** predefined terms and atoms.

Boldface is used for:

- Built-in predicates, mentioned in the text.
- Names of programs.

User's Guide

Chapter 1
Program Manipulation and Global Data

In principle, the distinction between *program* and *data* does not exist in Prolog: *data* can be represented as facts or clauses and called the same way as *program* clauses. Moreover, the existence of the predicate **clause/2** - which in ***ProLog by BIM*** works on dynamic and static predicates - helps in the (for logic rightful) confusion between *program* and *data*. Still, most applications written in Prolog consist of a set of clauses that are not changed during execution and it makes sense to refer to these clauses as the *program*. The clauses that are changed during execution can be thought of as *data*. It should first be noted that most problems can be solved without modifying the Prolog database during execution: programs making excessive use of this feature of modifying the Prolog database are not written according to the logic programming paradigm and are probably less declarative. Especially if the program is deterministic - if it exhibits no deep backtracking - it is better to implement changing data as an extra argument to the predicates. An example will follow later. If it is necessary to use global (and changing) data, ***ProLog by BIM*** has alternative predicates to **assert/retract**: the **record** predicates. Finally, if large amounts of data are to be manipulated by a program or this data has to survive the execution of an application, the data can be stored in the external ***ProLog by BIM*** ClauseDB. If this data already exists in a commercial database, there is a big chance ***ProLog by BIM*** offers you a transparent way to access it. For more information on the *ClauseDB* and external databases, the reader is referred to the appropriate manuals.

1.1 The dynamic Prolog database

The database predicates which modify the Prolog database, are **assert** and **retract**. The modifications they make are permanent, in the sense that the modifications are not undone on backtracking. In some applications, these predicates are used to modify the program, but most often they are used to simulate global data, for example: the setting of flags controlling execution or global counters. In ***ProLog by BIM***, the use of **assert/retract** should be reserved for the (rare) cases where one really wants to change the program dynamically. This because in ***ProLog by BIM***, asserted predicates are partially compiled in order to make their execution faster, but obviously, compilation reduces the speed of assertion.

Predicates can only be modified by **assert/retract** if they are *dynamic*. There are two ways to make a predicate dynamic:

By *declaring* to the compiler that the predicate is to be dynamic:

- with the directive **dynamic/1**
- with the directives **mode/1** or **index/2** without giving definitions for the predicate
- by compiling the predicate for debug.

By *creating* a predicate at runtime, when no definitions for the predicate already exist, with:

- a call to **mode/1** or **index/2**, or,
- the first assert of a clause for that predicate.

A predicate that is not dynamic, is called *static*: **assert/retract** fail (with an error message) on static predicates. However, static predicates can be modified: **abolish/1** and **retractall/1** can delete all clauses of a static predicate. After that, it is possible to load a static or a dynamic predicate with the same name and arity. In the current release of ***ProLog by BIM***, it is unsafe to change a static predicate into a dynamic one in this way. Changes from dynamic to static - by reconsult for instance - are fully supported.

Getting more out of dynamic predicates

Keep dynamic predicates partially static

On the average, static predicates are about 10 times faster than dynamic predicates (but this can vary from 2 to several orders of magnitude). Therefore it pays to make as many clauses static as possible. This is true, for instance, if conceptually a predicate is dynamic, but there is still a subset of its clauses that does not change, as in the following example:

The fixed subset and known at compile time:

```
:- dynamic a/1 .
:- a/1 index 1/1997 .
a(1) .
a(2) .
...
a(1000) .
?- assert(a(1001)) .
?- a(208) .
```

During the execution, additional facts will be added. Instead of declaring a/1 dynamic and use assert(a(_X)) in the program, an alternative approach is to add one definition to a/1, which is a catch-all for the dynamically added definitions:

```
:- dynamic dynamic_a/1 .
:- dynamic_a/1 index 1/97 .
static_a(1) .
static_a(2) .
...
static_a(1000) .
a(_X) :- var(_X), !, ( static_a(_X) ; dynamic_a(_X) ).
a(_X) :- static_a(_X), ! .
a(_X) :- dynamic_a(_X) .
?- assert(dynamic_a(1001)) .
?- a(208) .
```

The above principle of keeping as much of the predicate static as possible can even be applied in the case of retract: then, a/n is replaced by a set of facts initially known, and one definition for a/n which catches the deleted facts; moreover, the goal retract((a(_X,...):-_)) must be replaced in the program by asserta((exists_a(_X) :- !, fail)):

Known clauses at compile time:

```
:- dynamic a/n .
:- a/n index 1/1997 .
a(1,...) :- ... .
a(2,...) :- ... .
...
a(1000,...) :- ... .
?- retract( (a(201,...)) :- _ ) [, !] .
?- a(109,...) .
```

This is replaced by:

```
:- dynamic exists_a/1 .
:- exists_a/1 index 1/97 .
exists_a(_) .
initial_a(1,...) :- ... .
initial_a(2,...) :- ... .
...
initial_a(1000,...) :- ... .
a(_X,...) :- exists_a(_X), initial_a(_X,...) .
?- asserta( (exists_a(201) :- !, fail) ) .
?- a(109,...) .
```

Declarations for improved performance

The execution performance of dynamic code can be improved by indexing and hashing. By default, no hashing is performed, and indexing is only on the first argument of a clause. If indexing is required on any other argument, the programmer must declare this to the system by means of an index declaration or goal. The size of the hash table should be chosen carefully and when the number of clauses changes, rehashing is advisable.

The referenced database

There are versions of **assert/retract** with an extra argument: the clause reference. The clause reference is used to refer to a clause without the need to specify its form, or build its term representation. Moreover, when **rretract/2** is called with the reference instantiated, it does not create a choicepoint - which **retract/1** always does. So, the combination **rassert/ rretract** is attractive, as in the following piece of code:

```
p(...) :-
        ...
        rassert(...,_Ref),
        ...
        p(...),
        ...
        rretract(_Ref),
        ... .
```

Programs should not rely on the type of the reference, or its value. In particular, no ordering is guaranteed on references. Although arithmetic on references is possible in the current release, the values are meaningless and non-portable across new releases and possibly even across different runs of a ***ProLog by BIM*** program under the same release.

1.2 Global data with the record predicates

The term *record predicates* refers to the predicates **record/2, rerecord/2, recorded/2, erase/1** and their variants and enhancements.

Basics

These are the predicates which should be used whenever global data is to be manipulated. However, it is important to remember there is often a more declarative way of solving problems than with global data.

The basic facilities for manipulating the global data are:

- storing data: **record**
- updating data: **rerecord**
- retrieving data: **recorded**
- deleting data: **erase.**

Every predicate with one key has a variant with two keys. For simple usage, a single key is sufficient. The key is used to identify a piece of information. A key can be any non-free Prolog term, of which only the principal functor is significant. These predicates can be used to implement switches:

```
print_debug :-
        recorded(debuginfo,_Debug),
        write_debug_info(_Debug),
        rerecord(debuginfo,[]) .
```

Global counters can also be implemented with the record predicates:

```
initialize_number :-
      rerecord(cntr,0) . { record might already exist }
get_number(_Old) :-
      recorded(cntr,_Old),
      _New is _Old + 1,
      rerecord(cntr,_New) .
```

Using **rerecord/2** in this way overwrites any existing value. **Record/2** fails if *cntr* was previously used as a key.

It is possible to incorporate the initialize_number in get_number by making it more expensive.

```
get_number(_Old) :-
      recorded(cntr,_Old), !,
      _New is _Old + 1,
      rerecord(cntr,_New) .
get_number(0) :-
      record(cntr,1) .
```

Double keys give more flexibility for the choice of the key and suggest the division of the stored information in sub-domains. A similar effect may be achieved by module qualifying the keys.

Multi-dimensional data structures

Double keys can be used to simulate a two-dimensional data structure. However, for multi-dimensional data structures, **rerecord_arg/3** and **recorded_arg/3** (and their variants with double key) are more appropriate. These new predicates allow selective update/retrieval of information. The following sequence of queries illustrates this:

Example

```
?- record(table,t(t(1,2,3),t(4,5,6),t(7,8,9))) .
Yes
?- recorded(table,_X) .
_X = t(t(1,2,3),t(4,5,6),t(7,8,9))
Yes
?- recorded_arg([1],table,_FirstRow) .
_FirstRow = t(1,2,3)
Yes
?- rerecord_arg([3,3],table,nine) .
Yes
?- recorded(table,_X) .
_X = t(t(1,2,3),t(4,5,6),t(7,8,nine))
Yes
```

The example looks very much like array manipulation in procedural languages. The first argument of the **record_arg** predicates, specifies the (multi- dimensional) index of the argument that has to be updated/retrieved.

Example

```
?- record(tree,t(t(t([],1,t(t([],4,[]),8,[])),2,t(t([],9,[]),
          0,[])))) .
Yes
?- recorded(tree,_X) .
_X = t(t(t(nil,1,t(t(nil,4,nil),8,nil)),2,t(t(nil,9,nil),
          0,nil))
Yes
?- recorded_arg([1,3,1,2],tree,_Lrlv).
_Lrlv = 4
Yes
```

```
?- rerecord_arg([1,3,1,2],tree,t([],5,[])) .
Yes
?- recorded_arg([1,3,1,2],tree,_Lrlv).
_Lrlv = t(nil,5,nil)
Yes
?- recorded(tree,_X) .
_X = t(t(nil,1,t(t(nil,t(nil,5,nil),nil),8,nil)),2,
          t(t(nil,9,nil),0,nil))
Yes
```

An attempt to retrieve or update a non-existent part of a recorded object, results in simple failure:

```
?- record(test,f(a,b,c)) .
Yes
?- recorded_arg([4],test,_X) .
No
?- rerecord_arg([2,8],test,hello) .
No
```

The stack: a special case of a global data structure

A pair of predicates is defined to simulate a global stack: **record_push/2** and **record_pop/2** .

Initialize an empty stack:

```
rerecord(stack,[])
```

Push an element:

```
record_push(stack,_Info)
```

Pop an element:

```
record_pop(stack,_Info)
```

Example writing a customized predicate to gather solutions.

```
all_sol(_Goal,_Solution,_SolutionList) :-
     rerecord(solutions,[]),
     call(_Goal), record_push(solutions,_Solution), fail .
all_sol(_Goal,_Solution,_SolutionList) :-
     recorded(solutions,_SolutionList),
     erase(solutions) .

p(1) .
p(2) .
?- all_sol(p(_X),_X,_L) .
_X = _12
_L = [2,1]
```

Or partially evaluated:

```
all_sol_p(_SolutionList) :-
     rerecord(solutions,[]),
     p(_Solution), record_push(solutions,_Solution), fail .
all_sol_p(_SolutionList) :-
     recorded(solutions,_SolutionList),
     erase(solutions) .

p(1) .
p(2) .
?- all_sol_p(_L) .
_L = [2,1]
```

The order of solutions differs from the one delivered by the built-in predicate **findall/3** because of the operation of **record_push/2**.

The speed of the above all_sol_p/1 might not be better than the speed of **findall/3** or **bagof/3**, but the purpose of the above predicate is to show a framework in which to develop solution gathering predicates using the record predicates. Anyway, gathering solutions impairs efficiency, and one should always question its use. In the above example, p([1,2]) as a fact would remove the need for gathering the solutions of p/1 at once.

The queue: a special case of a global data structure

A pair of predicates is defined to simulate a global stack: **record_init_queue/2, record_enqueue/2** and **record_dequeue/2.**

Initialize an empty queue:

```
record_init_queue(q1,[])
```

Enqueue an element:

```
record_push(q1,_Info)
```

Dequeue the first element:

```
record_pop(q1,_Info)
```

1.3 Exploring alternatives

A program with a counter

Three versions of a predicate do_n_times/2 will be given: the predicate calls _N times the _Goal and then fails; the _Goal must succeed exactly once for this example to make sense:

Version 1

```
do_n_times(_N,_Goal) :-
        _N > 0 ,
        call(_Goal) ,
        do_n_times(?(_N - 1),_Goal) .
```

Version 2

```
do_n_times(_N,_Goal) :-
        rerecord(counter,_N) ,
        do_n_times(_Goal) .
do_n_times(_Goal) :-
        recorded(counter,_N) , _N > 0 ,
        call(_Goal) ,
        rerecord(counter,?(_N - 1)) ,
        do_n_times(_Goal) .
```

Version 3

```
do_n_times(_N,_Goal) :-
        retractall(counter(_)) ,
        assert(counter(_N)) ,
        do_n_times(_Goal) .
do_n_times(_Goal) :-
        retract(counter(_N)) , _N > 0 ,
        call(_Goal) ,
        assert(counter(?(_N - 1))) ,
        do_n_times(_Goal) .
```

For a call to do_n_times/2 with the _Goal instantiated to true, version 2 takes twice as much time as version 1, and version 3 takes 10 times as much time as version 1. Version 3 also uses up code space, which must be garbage collected from time to time.

Communication of information between alternatives

Here follow three versions of a simple random number generator: from the previous random number, it calculates the next one:

Version 1:

```
random_gen(_X) :- rand(_X,1) .
rand(_Newvalue,_Prev) :-
        _N is (_Prev * 17) mod 37 ,
        (
                _Newvalue = _N
        ;
                rand(_Newvalue,_N)
        ) .
```

Version 2:

```
random_gen(_X) :-
        rerecord(previous_value,1) ,
        rand(_X) .
rand(_Newvalue) :-
        recorded(previous_value,_Prev) ,
        _Newvalue is (_Prev * 17) mod 37 ,
        rerecord(previous_value,_Newvalue) .
rand(_Newvalue) :- rand(_Newvalue) .
```

Version 3:

```
random_gen(_X) :-
        assert(previous_value(1)) ,
        rand(_X) .
rand(_Newvalue) :-
        retract(previous_value(_Prev)) ,
        _Newvalue is (_Prev * 17) mod 37 ,
        assert(previous_value(_Newvalue)) .
rand(_Newvalue) :- rand(_Newvalue) .
```

Version 1 is 5 times faster than version 2 and 100 times faster than version 3.

Global data when failure driven loops are not good enough

Consider a predicate with the following structure:

```
a(_In,_Out) :- test(_In) , ! , _In = _Out .
a(_In,_Out) :- transform(_In,_X) , a(_X,_Out) .
```

in which the call to transform creates a lot of heap garbage, while the arguments of a/2 are small. A failure driven loop does not help, unless data flows across alternatives and this can be achieved with global data:

```
a(_In,_Out) :- test(_In) , ! , _In = _Out .
a(_In,_Out) :- transform(_In,_X) ,
                rerecord(transformed,_X) , fail .
a(_In,_Out) :- recorded(transformed,_X) , a(_X,_Out).
```

Before deciding on such a program structure, the programmer should carefully consider the trade-off between garbage collection of the heap, garbage collection of the record heap and readability of the program.

A trade-off between updating and retrieving

The predicate decide/2 below, decides between the use of the Prolog database and the record database for storing global data: the first argument to decide/2 is a typical term to be stored and retrieved, the second argument is the relative frequency of updating and retrieving (given as a real number):

```
decide(_Term,_Frequ) :-
     measure_store_with_assert(_Term,_Time_assert) ,
     measure_retrieve_by_call(_Term,_Time_call) ,
     measure_store_with_rerecord(_Term,_Time_rerecord) ,
     measure_retrieve_by_recorded(_Term,_Time_recorded) ,
     _T1 is _Frequ*_Time_assert + _Time_call ,
     _T2 is _Frequ*_Time_rerecord + _Time_recorded ,
     (
          _T1 >= _T2 , ! ,write('use record predicates\n')
     ;
          write('use assert/retract\n')
     ) .
measure_store_with_assert(_Term,_Time_assert) :-
     abolish(fact(_)) ,
     cputime(_Start) ,
     (
          succeed(10000),
          assert(fact(_Term)),
          retract(fact(_Term)),
          fail
     ;
          true
     ),
     cputime(_End) ,
     _Time_assert is _End - _Start .
measure_retrieve_by_call(_Term,_Time_call) :-
     abolish(fact(_)) ,
     assert(fact(_Term)) ,
     cputime(_Start) ,
     (
          succeed(10000) ,
          fact(_Term) ,
          fail
     ;
          true
     ) ,
     cputime(_End) ,
     _Time_call is _End - _Start .
measure_store_with_rerecord(_Term,_Time_rerecord) :-
     cputime(_Start) ,
     (
          succeed(10000) ,
          rerecord(fact,_Term) ,
          fail
     ;
          true
     ) ,
     cputime(_End) ,
     _Time_rerecord is _End - _Start .
measure_retrieve_by_recorded(_Term,_Time_recorded) :-
     rerecord(fact,_Term) ,
     cputime(_Start) ,
     (
          succeed(10000) ,
          recorded(fact,_) ,
          fail
     ;
```

```
            true
        ) ,
        cputime(_End) ,
        _Time_recorded is _End - _Start .
succeed(_N) :- _N > 0 .
succeed(_N) :- _N > 1 , succeed(?(_N - 1)) .
```

For instance: ?- decide([1,2,3,4],0.01) . tells you to use **assert/retract**, while ?- decide([1,2,3,4],0.1) . tells you to use record predicates. After running this program a couple of times, it becomes obvious that complicated terms, when retrieved more often than updated, are best asserted, while smaller terms which must be updated a lot, are best kept in the record database.

Cyclic data

Cyclic data are beyond logic. Still, since Prolog does unification without *occurs check*, programs can create cyclic - or infinite - terms, and all predicates can deal with them. Here is a simple program using a cyclic term:

```
print_days(_N,[_Day|_Rest_days]) :-
        _N < 366 ,
        write(_N = _Day) , nl ,
        print_days(?(_N+1),_Rest_days) .
```

The query:

```
?- _Weekdays =
[monday,tuesday,wednesday,thursday,friday,saturday,
        sunday|_Weekdays] , print_days(1,_Weekdays) .
```

produces:

```
1 = monday
2 = tuesday
3 = wednesday
4 = thursday
5 = friday
6 = saturday
7 = sunday
8 = monday
...
362 = friday
363 = saturday
364 = sunday
365 = monday
```

For some reason, it might be necessary to store a cyclic structure. However this is not possible with **assert**: the query

```
?- _Weekdays = [monday,tuesday,wednesday,thursday,
        friday,saturday,sunday|_Weekdays] ,
        assert(fact(_Weekdays)) .
```

will cause an overflow of the internal table.

However it is possible to store cyclic structures using records:

Example

```
?- _Weekdays = [monday,tuesday,wednesday,thursday,
        friday,saturday,sunday|_Weekdays] ,
        rerecord(fact,_Weekdays) .
```

works correctly, as does a later

```
?- recorded(fact,_L) .
_L = [monday,tuesday,wednesday,thursday,
friday,saturday,sunday,...]
```

(the write depth was set on 8 before this query ...)

There is a way to **assert** the cyclic structure:

```
?- assert( ( rule(_X) :-
_X =[monday,tuesday,wednesday,thursday,
friday,saturday,sunday|_X] ) ) .
```

a later call like ?- rule(_L) . delivers

```
_L = [monday,tuesday,wednesday,thursday,friday,saturday,
sunday,..
```

The transformation from the original cyclic clause to the clause which can be asserted can be performed with canonical_clause/2 or cse_clause/2 (See "*Built-ins - In-core database manipulation*").

User's Guide

Chapter 2
Term Manipulation

This chapter describes a number of built-in predicates that perform manipulations on Prolog terms.

2.1 The relation between arg/3, functor/3 and =../2

The built-ins **arg/3, functor/3** and **=../2** can be defined in terms of each other, as shown in the following paragraphs.

Arg/3 in function of =../2

Suppose **arg/3** were not built-in, but **=../2** is. Then **arg/3** can be defined as follows:

```
my_arg(_Argnr,_Term,_Arg) :-
     integer(_Argnr), nonvar(_Term),
     _Term =.. [_|_Arglist],
     find_arg(_Argnr,_Arg,1,_Arglist).
find_arg(_Argnr,_Arg,_CurrArgNr,[_CurrArg|_Arglist]) :-
     ( _Argnr == _CurrArgNr ->
          _Arg = _CurrArg
     ;
          _CurrArgNr < _Argnr,
          _NextArgN is _CurrArgNr+1,
          find_arg(_Argnr,_Arg,_NextArgN,_Arglist)
     ).
```

Functor/3 in function of =../2

Assume **functor/3** is not a built-in predicate, and **=../2** is built-in. Then **functor/3** can be defined as:

```
functor(_Term,_Name,_Arity) :-
     ( var(_Term) ->
          atomic(_Name),
          integer(_Arity), 0 =< _Arity, _Arity < 256,
          length(_Arglist,_Arity),
          _Term =.. [_Name|_Arglist]
     ;
          _Term =.. [_Name|_Arglist],
          length(_Arglist,_Arity)
     ).
length(_L,_N) :-
     /* mode length(+,-) : compute length _N */
     /* mode length(-,+) : construct list _L */
```

=../2 in function of functor/3 and arg/3

Finally, suppose **=../2** is not built-in, but **functor/3** and **arg/3** are. Then **=../2** can be defined as:

```
_Term =.. [_Name|_ArgList] :-
     ( var(_Term) -> length(_ArgList,_Arity) ; true ),
     functor(_Term,_Name,_Arity) ,
     check_args(_Pos,_Arity,_Term,_ArgList).
check_args(_N,_Arity,_Term,[_ArgN|_ArgList]) :-
     ( _N > _Arity ->
          _ArgList = []
     ;
          arg(_N,_Term,_ArgN),
          _ArgList = [_ArgN|_NewArgList],
          check_args(?(_N+1),_Arity,_Term,_NewArgList)
     ).
```

Discussion of functor/2, arg/3 and =../2

The main difference between the actual implementation as built-in predicates and the above implementation is that **arg/3** and **functor/3** run in constant time in ***ProLog by BIM*** - at least in the mode (i,?,?). The time it takes to get the Nth argument of a term with arity M by means of the built-in **arg/3**, is independent of N and M. This means that an array (a data structure with constant time access to its entries) can be simulated by a Prolog term, using **arg/3** for access. On the other hand, constant time updates are only possible using global data and the record predicates (Multi-dimensional data structures on page 12). The important difference is that the above implementation of **functor/3** and **arg/3** always uses an amount of heap proportional to the arity of the term: this is due to the second argument of **=../2** (=../2 in function of functor/3 and arg/3 on page 22).

2.2 Using arg/3, functor/3 and =../2 in a program

Different versions of a predicate **subterm/2** that succeeds if and only if its first argument is identical to the second argument or a part of the second argument:
subterm/2 is a variant of the **member/2** predicate. The purpose of the different examples is to show how the use of one predicate can be exchanged for another.

Version 1 with =../2

```
subterm(_S,_T) :- _S == _T .
subterm(_S,_T) :-
     nonvar(_T),
     _T =.. [_|_ListArgs],
     subterm_list(_S,_ListArgs) .
subterm_list(_S,[_Arg|_ListArgs]) :-
     subterm(_S,_Arg) .
subterm_list(_S,[_Arg|_ListArgs]) :-
     subterm_list(_S,_ListArgs) .
```

Version 2 with functor/3 and arg/3

```
subterm(_S,_T) :- _S == _T .
subterm(_S,_T) :-
     nonvar(_T),
     functor(_T,_,_Arity),
     subterm_arg(_S,_T,_Arity) .
subterm_arg(_S,_T,_ArgNr) :-
     _ArgNr > 0 ,
     arg(_ArgNr,_T,_Arg),
     subterm(_S,_Arg) .
subterm_arg(_S,_T,_ArgNr) :-
     _ArgNr > 0 ,
     subterm_arg(_S,_T,?(_ArgNr-1)) .
```

Comparison of both versions

In version 1, **=../2** has been used to construct the list of arguments of a term. Then this list is traversed in the usual way: treating the head and the tail in separate clauses. In version 2, **functor/3** caters for the arity of the term, and then **arg/3** is called with _ArgNr between this arity and 0 to get at the arguments of the term. Of course, this loop for _ArgNr involves arithmetic. A test showed that only for small arities (at most 3) of the term, it can be advantageous to avoid **=../2** at the cost of arithmetic. In *"5.15 Replacing arithmetic with list processing"* on page 69, you can find more about this in other circumstances.

The disadvantage of **=../2** is its global stack usage: the list of arguments of argument 1 to **=../2** is constructed on the global stack. Version 1 uses at least twice as much global stack as version 2.

Since **arg/3** fails when its first argument is out of range, version 2 above can be rewritten as:

```
subterm(_S,_T) :- _S == _T .
subterm(_S,_T) :- subterm_arg(_S,_T,1) .
subterm_arg(_S,_T,_ArgNr) :-
     arg(_ArgNr,_T,_Arg),
     (
          subterm(_S,_Arg)
     ;
          subterm_arg(_S,_T,?(_ArgNr+1))
     ).
```

User's Guide

Chapter 3
Cyclic Terms

Unification in first order logic is restricted to finite terms. This means in particular that an attempt to unify _X with f(_X) must fail, since no substitution exists which makes _X and f(_X) equal and finite terms. Still, most Prolog implementations "allow" the unification of _X and f(_X), thereby creating a cyclic (sometimes also called infinite) term.

All the built-in predicates of ***ProLog by BIM*** can be called with cyclic terms. The system also has a number of predicates which can be used to work with cyclic terms.

The following sections describe in detail what to expect from unifications involving cyclic terms. Also, it is indicated how cyclic terms can be avoided and what price is paid for this.

3.1 When are cyclic terms created?

Unification of two terms is defined in such a way that it reduces to a set of simultaneous unifications of a variable, say _X, with a term. A cyclic term is created whenever, in such a unification, the term is not a variable and contains _X. This is illustrated with an example:

The unification:

```
foo(a,_X,f(_X)) = foo(_Y,f(_Y),_Z)
```

is reduced to:

```
_Y = a , _X = f(_Y) , _Z = f(_X)
```

and this can be reduced - by repeatedly applying a substitution - to:

```
_Y = a , _X = f(a) , _Z = f(f(a))
```

and no cyclic term is created.

The unification:

```
foo(_X) = foo(f(_X))
```

is reduced to:

```
_X = f(_X)
```

In this case, f(_X) contains _X with which it is unified. This detection is called the *occurs check*. It is this check which is omitted in ***ProLog by BIM*** - as in most commercial Prolog implementations. The cost of the *occurs check* is too high and most programs do not attempt to create cyclic terms anyway. So, not performing the *occurs check* seems to be a gain rather than a loss.

Now, the question is:

what is _X after the above unification?

The answer is:

_X is a compound term with principal functor f/1 and argument _X.

There are other interesting questions:

What does _X look when it is written?

Can _X be stored in the Prolog or record or external database?

Can _X be unified with itself or other cyclic terms?

What if _X is the argument of **arg/3** or **functor/3** or ... ?

Section *Built-ins and cyclic terms on page 31* answers most of these questions.

3.2 Avoiding cyclic terms

Why avoid cyclic terms at all?

The example below shows that the omission of the *occurs check* makes the meaning of a Prolog program different from the first order logic program it resembles.

In the example, natural numbers are represented by 0 and the **suc/1** functor applied to a natural number. So, 0 means zero, suc(0) means one, suc(suc(0))) three. In this way, all natural numbers can be represented with only two different symbols. A predicate smaller/2 is defined, which is meant to succeed if the first argument as a natural number is smaller than the second argument, and to fail otherwise:

```
smaller(_X,suc(_X)) .    { _X is smaller than the successor of _X }
smaller(_X,suc(_Y)) :-   { _X is smaller than the successor of _Y }
     smaller(_X,_Y) .              { if _X is smaller than _Y }
```

This appears to be all right and Prolog returns meaningful answers to queries such as:

```
?- smaller(suc(suc(suc(0))),suc(suc(suc(suc(0))))) .
Yes                                  { 3 smaller than 4 ? }

?- smaller(suc(0),suc(0)) .          { is 1 smaller than 1 ? }
No
?- smaller(_X,suc(suc(0))) .
          { is there an _X smaller than 2 and if so, which _X ? }
_X = suc(0)
Yes ;
_X = 0
Yes ;
No
```

But, consider the query:

```
?- smaller(suc(_X),_X) .
```

It asks: "is there an _X such that the successor of _X is smaller than _X and if so, which _X ?"

From all that is known about the relation "smaller than" on natural numbers, the answer should be a firm "No". However, Prolog without the *occurs check* answers:

```
_X = suc(suc(suc(suc(suc(...)))))
Yes
```

which could be read: "infinity" is smaller than its successor. Indeed, _X is a cyclic term and can be interpreted as the infinite natural number.

The conclusion is: if a Prolog program has to have the same meaning as its first order logic equivalent, then cyclic terms should certainly never be created.

How to avoid cyclic terms

ProLog by BIM provides the built-in **is_inf/1** which allows to test wether a given term is cyclic. Furthermore ***ProLog by BIM*** has two additional built-in predicates whose purpose is to make it possible to avoid cyclic terms completely: **occur/2** and **occurs/2**. The former performs unification with *occurs check*. The latter checks whether its first argument, which must be a variable or an atom, is in the second argument.

Therefore, **occur/2** behaves as if defined by:

```
occur(_X,_Y) :-
      _X = _Y ,
      ( is_inf(_X) ->
            fail
      ;
            true
      ).
```

By replacing all implicit and explicit unifications in a Prolog program by an appropriate call to **occur/2**, this transformed program will never create cyclic terms. Unification is performed at several places in a program:

- The head of a clause
- Calls to **=/2**
- Calls to other built-in predicates

Examples:

Transforming the head of a predicate

Consider the following transformation:

original clauses:

```
append([_X|_L1],_L2,[_X|_L3]) :-
      append(_L1,_L2,_L3) .
foo(_X,g(_X),[_X]) .
```

transformed clauses:

```
append([_X|_L1],_L2,[_Y|_L3]) :-
      occur(_X,_Y) ,
      append(_L1,_L2,_L3) .
foo(_X,g(_Y1),[_Y2]) :-
      occur(_X,_Y1) ,
      occur(_X,_Y2) .
```

The rule is: if a variable _X occurs more than once in the head of a clause, replace all occurrences but one by a new variable, say _Yi, and add in the beginning of the body calls to **occur/2** of the form:

```
occur(_X,_Y1) , occur(_X,_Y2) , ...
```

Transforming calls to =/2

Simply replace the call _X = _Y by occur(_X,_Y) , whatever the form of _X and _Y.

So:

```
a(_X,_Y) :- tr(_X,_Z) , _Y = [_Z] .
```

becomes:

```
a(_X,_Y) :- tr(_X,_Z) , occur(_Y,[_Z]) .
```

Transforming calls to other built-in predicates

The rule here is the following:

> Every argument of the built-in predicate that can be bound to a non-atomic term, must be replaced by a new variable. This new variable must be unified with *occurs check* with the original argument after the call to the built-in predicate.

This rule can be relaxed a bit.

Examples

> A call like arg(_X,_Y,_Z) could bind _Z to a non-atomic term, so this call must be replaced by arg(_X,_Y,_New) , occur(_New,_Z).

The relaxations in the rule are as follows:

> If the argument is syntactically a variable, occurring for the first time, then this transformation needs not to be done.
>
> If the argument is atomic itself, this transformation needs not to be done, since there is no danger of binding this atomic term to a non-atomic one.
>
> If the argument is a partially instantiated term, only variables inside this term must be replaced by new variables.

Some illustrations of these relaxations:

```
write_arg(_N,_T) :- arg(_N,_T,_Arg) , write(_Arg) .
```

must not be transformed, because _Arg appears for the first time as third argument of **arg/3** - counting from the left to the right of course. The call to **write/1** needs no transformation, because **write/1** does not bind its argument.

```
check_arg(_N,_T,_Arg) :- atomic(_Arg) , arg(_N,_T,_Arg) .
```

must not be transformed, _Arg in the third argument of **arg/3** is known to be atomic.

```
solve(_H) :- clause(_H,_B) , solve(_B) .
```

This is a special case: the general rule says to transform the above clause to:

```
solve(_H) :- clause(_NewH,_B) , occur(_H,_NewH) ,solve(_B) .
```

but, **clause/2** needs a partially instantiated first argument. So, _H must be replaced by a term with at least the same principal functor:

```
solve(_H) :-
     functor(_H,_N,_Ar) ,
     functor(_NewH,_N,_Ar) ,
     clause(_NewH,_B) , occur(_H,_NewH) ,solve(_B) .
```

The above rules are overkill for most Prolog programs and the transformations described above should only be applied at the places where cyclic terms are to be expected and avoided.

The predicate **occurs/2** permits the writing of customized unification predicates. For example: it is possible to write a meta-interpreter which performs unification with functions.

The cost of using occur/2

The effect of the above transformations can be shown on the famous *naive reverse* benchmark where performance degrades by a factor 5. This shows why Prolog implementors choose to omit the *occurs check*. Also, the predicate **occurs/2** is rather expensive.

3.3 Built-ins and cyclic terms

Unification without *occurs check* occurs in the head of a clause, by explicit calls to **=/2** and in arguments of built-in predicates.

Unification

It was shown already how the execution of the goal:

```
_X = f(_X)
```

creates a cyclic term and - most importantly for the programmer - does so in finite time. In fact ***ProLog by BIM*** allows any unification involving cyclic terms.

```
?- _X = f(_X) , _Y = f(_Y) , g(a,_X) = g(b,_Y) .
No
?- _X = f(_X) , _Y = f(_Y) , g(_X,a) = g(_Y,b) .
No
?- _X = [1,2,3|_X] , [_Y|_Z] = _X .
_X = [1,2,3,1,...]
_Y = 1
_Z = [2,3,1,2,...]
Yes
```

I/O

In Why avoid cyclic terms at all? on page 28, it was shown already that ***ProLog by BIM*** has a way to write out cyclic terms. For this, one has to set the please option **wd** to a positive integer with **please/2**:

```
?- please(wd,8) .
```

reduces the depth to which terms are written out to eight levels:

```
?- _X = f(_X), write(_X) .
f(f(f(f(f(f(f(f(...)))))))
```

All variants of **write/1** are influenced in the same way by the setting of the write depth.

The default write depth is -1, meaning that terms are written out to their actual depth. Writing cyclic terms with write depth equal to -1 causes an infinite loop, produces lots of output and either ***ProLog by BIM*** crashes or the user has to stop ***ProLog by BIM*** by ^C or ^Z .

The setting of the maximum write depth for the debugger is independent and is achieved with **debug/2** or the D command of the debugger.

Reading an infinite term is not possible, since it cannot be syntactically represented. On the other hand, the sequence ... is a correct atom, so that a f(f(f(f(f(f(f(f(...)))))))) is a syntactically correct and finite term.

Database predicates

The reader is also referred to the chapter *The dynamic Prolog database on page 9* in this manual.

The predicates **assert/retract/clause** cannot deal with cyclic terms directly; that is, the query:

```
?- _X = f(_X) , assert(fact(_X)) .
```

will cause an overflow of the internal tables.

Still, one can mimic such an assert by the query:

```
?- assert((fact(_X) :- _X = f(_X))) .
```

The transformation from the original cyclic clause to the clause which can be asserted can be performed with canonical_clause/2 or cse_clause/2 (See "*Built-ins - In-core database manipulation*")

The **record** predicates can be used for storing and retrieving cyclic terms as shown in the following examples (the write depth is set to 5):

A cyclic term is recorded (under key cyclic) and retrieved:

```
?- _X = f(_X) , record(cyclic,_X) .
_X = f(f(f(f(f(...)))))
Yes
?- recorded(cyclic,_Z) .
_Z = f(f(f(f(f(...)))))
Yes
```

A sub-term (on nesting level 3) of the recorded cyclic term can be retrieved:

```
?- recorded_arg([1,1,1],cyclic,_Z) .
_Z = f(f(f(f(f(...)))))
Yes
```

The recorded term is changed to a non-cyclic term:

```
?- rerecord_arg([1,1,1],cyclic,gee) .
Yes
?- recorded(cyclic,_Z) .
_Z = f(gee)
Yes
```

A cyclic list is recorded:

```
?- _X = [1,2,3|_X] , rerecord(cyclic,_X) .
_X = [1,2,3,1,...]
Yes
```

An atom is pushed on the recorded cyclic list:

```
?- record_push(cyclic,haa) .
Yes
?- recorded(cyclic,_Z) .
_Z = [haa,1,2,3,...]
Yes
```

The top of the recorded cyclic list is popped off:

```
?- record_pop(cyclic,_Popped) .
_Popped = haa
Yes
?- record_pop(cyclic,_Popped) .
_Popped = 1
Yes
?- record_pop(cyclic,_Popped) .
_Popped = 2
Yes
?- recorded(cyclic,_Z) .
_Z = [3,1,2,3,...]
Yes
```

The ***ProLog by BIM ClauseDB*** can store the same terms as the in-core database and external databases cannot store cyclic terms at al

Conversion predicates

This section describes the impact of cyclic terms on built-in predicates that are used for conversion of terms.

Atomtolist/2, name/2 and asciilist/2

If the second argument of **atomtolist/2, name/2** or **asciilist/2** is a cyclic list, then an atom will be built with the maximum length MAX_ATOM. The maximum length can be found in the reference manual.

The following query illustrates this:

```
?- _X = [a,b|_X],atomtolist(_Y,_X) , atomlength(_Y,_L) ,
          write(_L) , nl , fail .
16383
```

Arg/3, functor/3 and =../2

A set of examples covering most cases is given:

```
?- _X = f(a,_X,b) , arg(2,_X,_Y) .
_X = f(a,f(a,f(a,f(a,f(...,...,...),b),b),b),b)
_Y = f(a,f(a,f(a,f(a,f(...,...,...),b),b),b),b)
Yes
?- arg(1,f(_X),f(_X)) .
_X = f(f(f(f(f(...)))))
Yes
?- _X = f(a,_X,b) , functor(_X,_N,_A) .
_X = f(a,f(a,f(a,f(a,f(...,...,...),b),b),b),b)
_N = f
_A = 3
Yes
?- _X = f(a,_X,b) , functor(_X,f,3) .
_X = f(a,f(a,f(a,f(a,f(...,...,...),b),b),b),b)
Yes
?- _X = f(a,_X,b) , _X =.. [_N|_Arglist] .
_X = f(a,f(a,f(a,f(a,f(...,...,...),b),b),b),b)
_N = f
_Arglist = [a,f(a,f(a,f(...,...,...),b),b),b]
Yes
?- _X = [a,b|_X] , _Y =.. _X .
*** BUILTIN 525 *** =../2 : The second argument must be a
[]-terminated list with at most 255 elements and first element
an atom.
?- _X = [a,b|_X] , a(1,2) =.. _X .
No
```

Other built-in predicates

The following is a list of built-in predicates which cause problems when called with a cyclic term at a specific argument. There are two kinds of problems: either the predicate will never finish, or ***ProLog by BIM*** will crash after some time. The former is called an infinite loop and can be interrupted by ^C if an external handler has been installed. Arguments that must be atomic or free are never mentioned.

- **assert predicates:** possible crash.
- **listing/1**: infinite loop: keeps on displaying predicates
- **namevars/3,4**: argument 1: possible crash
- The list notation for consult: if this list is cyclic, consulting goes on forever
- **op/3**: third argument: infinite loop

3.4 Examples of using cyclic terms

The names of the days of the year

Suppose one wants a predicate that prints out something like:

1 = monday
2 = tuesday
3 = wednesday
4 = thursday
5 = friday
6 = saturday
7 = sunday
8 = monday
...
362 = friday
363 = saturday
364 = sunday
365 = monday

for every day of the year, its number and name of the day, but it fails. One possibility is:

```
print_days ( _N ) :-
      _N < 366 ,
      _D is ( _N mod 7 ) + 1,
      dayname ( _D , _Day ),
      write ( _N = _Day ), nl,
      print_days ( ?(_N+1) ) .
```

and the predicate needs the facts:

```
dayname ( 1 , monday ) .
dayname ( 2 , tuesday ) .
dayname ( 3 , wednesday ) .
dayname ( 4 , thursday ) .
dayname ( 5 , friday ) .
dayname ( 6 , saturday ) .
dayname ( 7 , sunday ) .
```

With cyclic terms, there is a more elegant solution:

```
print_days ( _N , [_Day|_Rest_days] ) :-
      _N < 366,
      write ( _N = _Day ), nl ,
      print_days ( ?(_N+1) , _Rest_days ) .
```

to be called as follows:

```
?- _Weekdays = [monday,tuesday,wednesday,thursday,friday,
      saturday, sunday|_Weekdays],
      print_days ( 1 , _Weekdays ) .
```

Path searching in a graph

Suppose a graph is specified by a series of facts connected/2 like:

```
connected(a,b) .
connected(b,c) .
connected(c,a) .
connected(c,d) .
connected(c,e) .
connected(e,a) .
connected(a,f) .
connected(f,b) .
```

and since it is supposed to be a non-directed graph, it is also possible to define:

```
linked(_X,_Y) :- connected(_X,_Y) .
linked(_X,_Y) :- connected(_Y,_X) .
```

Then a predicate path/2, which given 2 nodes produces all paths from the first one to the second one, could be defined as:

```
path(_Start,_End,_Path) :-
     path(_Start,_End,[_Start],_Path) .
path(_Start,_Start,_Path,_Path) .
path(_Start,_End,_Pin,_Pout) :-
     linked(_Start,_Newstart) ,
     \+ member(_Newstart,_Pin) ,
     path(_Newstart,_End,[_Newstart|_Pin],_Pout) .
```

An alternative implementation represents the graph as a cyclic term and uses this term to travel through the graph. This representation has the following specifications:

- It is a list of all the nodes
- Every node is a term of the form node(_Name,_Info,_Linklist) where:
  _Name is the name of the node
  _Info is associated information - used here as 'visited bit'
  _Linklist is a list of nodes linked/2 to _Name and of course of a similar form

The predicate graph/1 builds up this cyclic representation of the graph:

```
graph(_Graph) :-
     setof(node(_N1,_,_),_N2 ^ linked(_N1,_N2),_Graph) ,
     fill_in(_Graph,_Graph) .


fill_in([],_) .
fill_in([node(_X,_,_L)|_R],_Graph) :-
     setof(_Y,linked(_X,_Y),_Lx) ,
     makel(_Lx,_Graph,_L) ,
     fill_in(_R,_Graph) .


makel([],_,[]) .
makel([_Y|_R],_Graph,[_Ny|_Nr]) :-
     _Ny = node(_Y,_,_) ,
     member(_Ny,_Graph) , ! ,
     makel(_R,_Graph,_Nr) .
```

Using this representation of the graph, a new implementation of path/3 is possible:

```
path(_Start,_End,[_Start|_Rest]) :-
     graph(_Gr) ,
     member(node(_Start,_Info,_Nl),_Gr) , ! ,
     _Info = visited ,
          {_Info was free: nothing has been visited yet}
     path(_Start,_End,_Rest,_Nl) .


path(_Start,_Start,[],_) :- ! .
path(_Start,_End,[_Next|_Rest],_Nl) :-
     member(node(_Next,_Info,_Newnl),_Nl) ,
     var(_Info) , { _info is free thus not visited yet }
     _Info = visited ,
     path(_Next,_End,_Rest,_Newnl) .
```

In a program that traverses the graph a lot, it might be advantageous to build up the cyclic term representation of the graph once at the beginning, and never use the facts connected/1 again. This is true for the path searching where the second path/3 predicate is faster than the former one for longer paths.

Warning

The above representation might not be generally applicable to all kinds of graphs: the above example graph is non-directed, cyclic and connected.

User's Guide

Chapter 4
All-Solution Predicates

All-solution predicates are predicates that combine the results of an underlying query. They gather all solutions. ***ProLog by BIM*** provides the following all-solution predicates: **findall/3, findall/4, bagof/3** and **setof/3**.

The basic functionality is provided by the built-in **findall/3**. The predicates **bagof/3** and **setof/3** can be built on top of it.

4.1 Findall/3

Introduction

findall/3 constructs a list out of all solutions of the call that is given as the second argument to **findall/3**. The list contains one element for each solution the system would report if the query were a top level query. The list of solutions is the third argument of the predicate **findall/3**. Each element in the list is the result of the instantiation of the first argument in the call to **findall/3** upon success of the query.

Example

```
a(1).
a(2).

?- findall(X,a(X),L).                  % [1]
L = [1,2]
?- findall(t(X),a(X),L).               % [2]
L = [t(1),t(2)]
?- findall(sol,a(X),L).                % [3]
L = [sol,sol]
?- M = sol, findall(M=X,a(X),L).       % [4]
L = [sol=1,sol=2]
?- findall(X,(a(X),X=3Example:),L)     % [5]
L = []
```

The queries can be read as:
[1] extend the list L with the term X each time a(X) succeeds.
[2] extend the list L with the term t(X) each time a(X) succeeds.
[3] extend the list L with the term sol each time a(X) succeeds.
[4] extend the list L with the term M=X each time a(X) succeeds.
[5] extend the list L with the term X each time a(X),X=3 succeeds.

The results can be read as:
[1] a(X) succeeded twice:
after the first time X==1 and
after the second time X==2
[2] a(X) succeeded twice:
after the first time t(X)==t(1) and
after the second time t(X)==t(2)
[3] a(X) succeeded twice: sol occurs twice in the solution list
[4] a(X) succeeded twice:
after the first time M=X == sol=1 and
after the second time M=X == sol=2
[5] a(X),X=3 never succeeded

Description

findall(_Template, _Goal, _SolutionList)

_Goal: the underlying goal of which all solutions will be sought

_Template: a description of the terms that will make up the elements of the solutionlist for each success of the underlying goal

_SolutionList: the list consisting of the resulting template values for each success of the underlying goal

Note:

- free variables in either the template or the goal before the execution of the predicate **findall/3** are still free after the execution of the predicate.
- **findall/3** always succeeds.

Examples

Example

```
a(1).
a(2).

?- findall(a(_X),a(_X),_SolutionList).
_X = _12
_SolutionList = [a(1),a(2)]
```

Example

```
a(1).
a(2).

?- _X = a(_Y), findall(_X,_X,_SolutionList).
_X = a(_14)
_Y = _14
_SolutionList = [a(1),a(2)]
```

Example

```
a(1,a).
a(2,b).

?- findall(_X,a(_X,_Y),_SolutionList).
_X = _8
_Y = _13
_SolutionList = [1,2]
```

Example

```
a(1,a).
a(2,b).
?- findall(_X,a(_X,c),_SolutionList).
_X = _8
_SolutionList = []
```

4.2 Bagof/3

Relation with findall/3

Often, **bagof/3** behaves exactly as **findall/3**.

Example

```
a(1).
a(2).

?- bagof(_X,a(_X),_SolutionList).
_SolutionList = [1,2]
```

This is the same answer as for **findall/3**. The queries for which this is true are characterized by: the _Goal does not contain any variable not contained in the _Template and _Goal has at least one answer. For such queries, the use of **findall/3** is recommended.

Behavior of bagof/3 if the goal has no solutions

Example

```
a(1,a).
a(2,b).

?- bagof(_X,a(_X,c),_SolutionList).
No
```

The call to **bagof/3** fails. This is different from **findall/3**, which returns the empty list.

Backtracking bagof/3 calls

The main difference between **bagof/3 and findall/3** becomes clear when the second argument _Goal contains free variables not contained in the _Template. In that case, **bagof/3** can return multiple solutions through backtracking. Each of these solutions corresponds to a different binding for these free variables, effectively partitioning the solution list.

Example

```
a(1,a).
a(2,b).
a(3,a).
a(4,b).

?- findall(_X,a(_X,_Y),_SolutionList).
_SolutionList = [1,2,3,4]
Yes
```

Note that findall does neither instantiate _X nor _Y.

The predicate bagof/3 partitions the list of solution [1,2,3,4] depending on the different instantiations for the initially free variable _Y. The query:

```
?- bagof(_X,a(_X,_Y),_SolutionList).
```

produces the following solutions:

```
_Y = a
_SolutionList = [1,3]
Yes ;
_Y = b
_SolutionList = [2,4]
Yes ;
No
```

This means that:

```
for _Y = a, the solutions for _X are [1,3]
for _Y = b, the solutions for _X are [2,4]
```

All solutions are calculated before the first solution list is returned. This feature is not clear at first, since **bagof/3** may return multiple solution lists through backtracking, creating the illusion that the next solution lists are not calculated until backtracking occurs.

The variable _Y becomes instantiated after the call, whereas the variable _X remains untouched.

In general, all free variables in the first argument of **bagof/3** remain free, but free variables occurring in the second argument only, are instantiated and every different instantiation thereof results in a separate solution list of **bagof/3**.

The notion of different instantiation may vary from Prolog system to Prolog system when the bindings do not make the variables ground. The next example shows how ***ProLog by BIM*** treats this:

```
a(1,_X,_X).
a(2,_Y,_Y).
a(3,_X,_Y).
a(4,f(_X),f(_X)).
a(5,f(_Y),f(_Y)).
a(6,f(_X),f(_Y)).

?- bagof(_X,a(_X,_Y,_Z),_L) .
_Y = _1
_Z = _1
_L = [1,2]
Yes ;
_Y = _1
_Z = _2
_L = [3]
Yes ;
_Y = f(_1)
_Z = f(_1)
_L = [4,5]
Yes ;
_Y = f(_1)
_Z = f(_2)
_L = [6]
Yes ;
No
```

Reading of the answers:

```
when _Y == _Z and free,
     _X is one of [1,2]
when _Y and _Z are free and not identical,
     _X is 3
when _Y == _Z == f(_),
     _X is one of [4,5]
when _Y == f(_1) and _Z == f(_2) and not identical,
     _X is 6
```

A mixed declarative/procedural reading for a goal like:

```
?- bagof(_X,a(_X,_Y),_L) .
```

is:

```
is there a _Y
     such that _L is the list of _X
     such that a(_X,_Y) succeeds ?
```

Existential quantification in bagof/3

The partitioning of the solutions can be modified by existential quantification of the variables in the goal. To indicate that solutions for _X are wanted for which there exists a value for _Y such that goal a(_X,_Y) is satisfied, no matter what that value for _Y is, the following syntax is used:

```
a(1,a).
a(2,b).
a(3,a).
a(4,b).

?- bagof(_X,_Y^a(_X,_Y),_SolutionList).
_X = _8
_Y = _12
_SolutionList = [1,2,3,4]
Yes
```

This example shows how existential quantification for the variables in the second argument is written: _Qualifier_Term ^ _Goal instead of _Goal. No backtracking over the value of free variables in the argument _Goal occurs if they occur in the _Qualifier_Term.

The free variables in _Qualifier_Term remain free after the call to **bagof/3**, just like the free variables in the _Template argument.

The query:

```
?- bagof(_X,_Y^a(_X,_Y),_SolutionList) .
```

now reads:

```
what is the list of _X such that there exist a _Y such that
a(_X,_Y) succeeds ?
```

Void variables in the argument _Goal are often overlooked:

```
a(1,a).
a(2,b).
a(3,a).
a(4,b).

?- bagof(_X,a(_X,_),_SolutionList) .
_X = _8
_SolutionList = [1,3]
Yes ;
_X = _8
_SolutionList = [2,4]
Yes ;
No
```

In the example above, the intended meaning of the void variable is very likely that it does not matter what its instantiation is. This cannot be expressed when a void variable is used. Existential quantification is needed and a named variable must be used:

```
a(1,a).
a(2,b).
a(3,a).
a(4,b).

?- bagof(_X,_Any^a(_X,_Any),_SolutionList).
_X = _8
_Any = _12
_SolutionList = [1,2,3,4]
Yes
```

4.3 Setof/3

Relation with bagof/3

The predicate **setof/3** is an extension of **bagof/3**: it sorts the solution lists and removes duplicates. The sorting is based on the **@</2** relation.

The following example shows the difference:

Example

```
a(1,a).
a(2,a).
a(1,b).
a(2,b).
a(3,a).
a(1,a).
a(2,b).
a(1,b).

?- bagof(_X,a(_X,_Y),_SolutionList).
_X = _8
_Y = a
_SolutionList = [1,2,3,1]
Yes ;
_X = _8
_Y = b
_SolutionList = [1,2,2,1]
Yes ;
No
?- setof(_X,a(_X,_Y),_SolutionList).
_X = _8
_Y = a
_SolutionList = [1,2,3]
Yes ;
_X = _8
_Y = b
_SolutionList = [1,2]
Yes ;
No
```

Existential quantification in setof/3

The effect of existential quantification in **setof/3** is similar to its effect in **bagof/3**:

Example

```
a(1,a).
a(2,a).
a(1,b).
a(2,b).
a(3,a).
a(1,a).
a(2,b).
a(1,b).

?- setof(_X,_Y^a(_X,_Y),_SolutionList) .
_X = _8
_Y = _12
_SolutionList = [1,2,3]
Yes
```

4.4 The order of solutions

The order of solutions in the list

As mentioned above, the order in _SolutionList as constructed by **setof/3** is determined by **@</2**.

The order in _SolutionList as constructed by **findall/3** and **bagof/3**, is determined by the order in which the _Goal has found its answers. Thus:

```
a(f(9)).
a(b(6)).

?- bagof(_X,a(_X),_L) .
```

is guaranteed to deliver the answer:

```
_L = [f(9),b(6)]
```

and not:

```
_L = [b(6),f(9)]
```

Programs can thus rely on the order inside the _SolutionList.

Order of solutions by backtracking

Only **bagof/3** and **setof/3** are concerned here: the order of solutions by backtracking is implementation dependent. Thus, in the following example:

```
a(1,a).
a(2,b).

?- bagof(_X,a(_X,_Y),_L) .
```

the outcome can be first _Y = a and _L = [1] and then _Y = b and _L = [2], or the other way around.

It is therefore advised not to write programs that rely upon a particular order.

4.5 Discussion of all-solution predicates

Unnecessary use

The all-solution predicates can be useful, but are easily misused. It is not uncommon to find code similar to the following:

```
fact(1).
fact(2).
fact(3).
b :- findall(_X,fact(_X),_L), member(_Y,_L), c(_Y).
```

Note:

The predicate member/2 is defined as usual:

```
member(_X,[_X|_L]).
member(_X,[_|_L]) :- member(_X,_L).
```

If side-effects do not influence the solution set of fact/1, this is better coded as:

```
fact(1).
fact(2).
fact(3).
b :- fact(_X), c(_X).
```

Findall/3 and member behave as almost inverse predicates:findall/3 makes a list of solutions obtained by backtracking, whereas member/3 backtracks over the elements of a list. Compare:

```
..., findall(X,p(X),L), member(Y,L), ...
..., p(Y), ...
```

Compare

```
..., findall(X,member(X,L),NL), /* ground(L), NL == L */
```

Late pruning

Here is another example of the misuse of the **findall/3** predicate:

```
fact(1).
fact(2).
fact(3).
b :- findall(_X,fact(_X),_L), member(_Y,_L), _Y > 1, c(_Y).
```

Note:

the predicate member/2 is defined as usual:

```
member(_X,[_X|_L]).
member(_X,[_|_L]) :- member(_X,_L).
```

This example could be rewritten as:

```
fact(1).
fact(2).
fact(3).
b :- findall(_X,(fact(_X), _X > 1),_L), member(_Y,_L), c(_Y).
```

This solution discards unwanted solutions before they are put in the list _L.

A further improvement is:

```
fact(1).
fact(2).
fact(3).
d(_X) :- fact(_X), _X > 1.
b :- findall(_X,d(_X),_L), member(_Y,_L), c(_Y).
```

The improvement in this version is the introduction of a new goal instead of the conjunction as the second argument to **bagof/3**. The goal in the **bagof/3** is interpreted. For a simple goal as in the above example, the effect of interpretation is small. For more complicated conjunctions however, it can become significant.

The best way to code the above example is of course:

```
fact(1).
fact(2).
fact(3).
b :- fact(_X), _X > 1, c(_X) .
```

Wrong data representation

Even if a list of items is really needed for some predicates, **findall/3** may not be the best way to create this list.

Example

```
fact(a).
fact(b).
fact(c).
needs_list :-
     findall(_X,fact(_X),_L), sumlist(_L,_Sum).
needs_fact :-
     fact(_X), c(_X) .
```

Note: member/2 and sumlist/2 are defined as:

```
member(_X,[_X|_L]).
member(_X,[_|_L]) :- member(_X,_L).
sumlist(_L,_Sum) :- sumlist(_L,_Sum,0).
sumlist([],_Sum,_Sum).
sumlist([_Element|_L],_Sum,_Partial) :-
     _NewPartial is _Partial + _Element,
     sumlist(_L,_Sum,_NewPartial).
```

A recoding of the above example:

```
list_facts([a,b,c]).          % note the change
needs_list :-
     list_facts(_L), sumlist(_L,_Sum).
needs_fact :-
     list_facts(_L), member(_X,_L), c(_X) .
```

We change the data representation and trade member/2 for findall/3. In most cases that change is advantageous. One should be aware that such an alternative implementation exists, based on a different representation. Which of the two is best, must be judged on a per case basis.

An interesting variation is the following:

```
fact(a,1).
fact(b,2).
fact(c,3).
list_facts(_L) :- list_facts(_L,1).
list_facts([_X|_L],_N) :-
     fact(_X,_N),
     /* !, green cut due to representation of fact/2 */
     list_facts(_L,?(_N+1)).
list_facts([],_) .
needs_list :- list_facts(_L), sumlist(_L,_Sum).
needs_fact :- fact(_X,_), c(_X).
```

The example above tries to combine the best of both worlds. It allows the selective retrieval of facts, and a fairly efficient creation of the solution list. It is important for speed to have indexing on both the first and second argument of fact/2, otherwise a search for the Nth element must be done during the creation of the solution list.

It is possible to merge list_facts/1 and sumlist/2 into a new predicate. This is not done as it would obscure the idea.

Alternatives to using findall/3

The paper 'Findall without findall/3', A. Mariën, B. Demoen, in the Proceedings of the 10th ICLP 1992, Budapest, shows a number of ways to compute all solutions without all-solution predicates.

Conclusion

All-solution predicates are used if predicates need to have access to more than one solution at the same time, and if a different data representation cannot be used to avoid the use of the all-solution predicates.

User's Guide

Chapter 5
Optimizations

This section of the manual discusses optimization techniques for use with the ***ProLog by BIM*** system. Optimization of an application program can be very important for its final acceptance. Optimization can happen at several levels. The most fundamental goal when writing a program is to produce a correct program. Only if that goal is reached should optimization be considered.

The first and most important optimization step is optimizing the algorithm. Neither the complexity of algorithms nor the selection of algorithms for use with Prolog is discussed here.

The second step in optimization is finding the hot spots of the program: a small part of a program is very often responsible for most of the execution time. In the libraries that come with ***ProLog by BIM*** you will find some tools to assist in this task.

Once the hot spots have been identified low-level optimizations can be performed. These optimizations can be either space or time optimizations. Sometimes, there is a trade-off, but often space optimization will result in time optimization. Some data indicate that for certain types of programs the runtime of a program is proportional to its heap usage. The optimizations discussed hereafter are mostly time optimizations.

A word of warning: in the world of optimization and benchmarking there is a golden rule: measure the effect of supposed optimizations on realistic problem sizes. There may be surprises.

5.1 Good :- simple, clear

A Prolog system is tuned towards execution of Prolog. This apparently obvious fact conveys a meaningful message: clean and pure Prolog code achieves the best speeds. Many built-ins have a significantly higher cost than Prolog code. This higher cost implies that programs which use built-ins extensively, benefit the least from a high speed implementation.

5.2 Determinism and indexing

Why bother?

Prolog is a language with built-in backtracking. While this feature is one of the powers of the language it should be used with care because its use is not free: it can significantly increase both the space and time requirements of a program. Many subproblems in a program are deterministic. There is no search necessary, and hence, no backtracking should occur. How to achieve execution without backtracking for these deterministic programs is the subject of the following discussion.

What is determinism?

A first informal definition of determinism was given in the problem statement: a deterministic predicate is a predicate without inherent search. Typical examples are predicates that have a number of input arguments which uniquely determine the output arguments. There is always one and exactly one solution. The example below shows a predicate that is called with a number and returns the English and French word.

Example

```
number_to_word(1,one,un) :- ...
number_to_word(2,two,deux) :- ...
number_to_word(3,three,trois) :- ...

?- number_to_word(2,E,F).
```

Note that the notion of input and output arguments was used. It should be clear that the actual use of a predicate determines the nature of the execution: deterministic execution will assume a number of calling patterns, often just one. A possible way to describe such a pattern is using a mode declaration:

```
:- mode number_to_word(+,-,-).
```

5.3 Exploiting determinism

Introduction

The previous definition of determinism is too informal to be of any assistance. Saying that a predicate is deterministic if it succeeds onceat the most for any call is equally very general but not of any practical use.

A definition is needed which captures exploitable determinism by the Prolog system. A first step in that direction is to consider the calling patterns, which could be described by mode declarations, to define when a predicate is deterministic. A predicate is deterministic with respect to a calling pattern if for that calling pattern there is one clause at the most which the head unifies with that goal.

Classical indexing

Historically, Prolog systems have only exploited an approximation of the determinism described in that last definition. They only consider the first argument of each clause of the predicate and the call, and then only the top-level functor thereof. Based on that information they will preselect a subset of possibly matching clauses. This technique is traditionally called indexing. If the selected set of clauses is a singleton, no backtracking occurs for this particular call.

Even if ***ProLog by BIM*** will recognize many more predicates as deterministic it is good to know this limitation in other systems. It can be of assistance in porting and in tuning programs developed with other implementations.

The examples below clarify the sort of determinism most Prolog systems recognize.

Example 1

```
p(1,a).
p(2,b).
p(3,c).
p(4,d).

?- p(2,L).
```

Only the second clause is tried. (***ProLog by BIM***: only one)

Example 2

```
p(1,a).
p(2,b).
p(3,c).
p(4,d).

?- p(N,b).
```

All clauses are tried (***ProLog by BIM***: only one): the actual first argument is free.

Example 3

```
p(f(1),a).
p(f(2),b).
p(f(3),c).
p(f(4),d).

?- p(f(2),L).
```

All clauses are tried (***ProLog by BIM***: all): the top-level argument is the same in all clauses.

The following example is somewhat larger and shows the different subsets that are selected depending on the call:

Example 4

```
clause head                    index

(1) a(_U) :-                   _
(2) a(1) :-                    1
(3) a([_X|_L]) :-              [_|_]
(4) a(c) :-                    c
(5) a(d) :-                    d
(6) a(s(a)) :-                 s/1
(7) a(s(b)) :-                 s/1
(8) a(t(_P,_Q)) :-             t/2
(9) a(_U) :-                   _
```

The second column shows the information that is considered for indexing.

Some example calls and the corresponding selected clause subsets:

```
query                          indexslection

(a) ?- a(1).                   1      [1,2,9]
(b) ?- a(2).                   2      [1,9]
(c) ?- a([1,2]).               [_|_]  [1,3,9]
(d) ?- a(c).                   c      [1,4,9]
(e) ?- a(s(_)).                s/1    [1,6,7,9]
(f) ?- a(s(a)).                s/1    [1,6,7,9]
(g) ?- a(s(c)).                s/1    [1,6,7,9]
(h) ?-a(t(_,_)).               t/2    [1,8,9]
(i) ?-a(_).                    _      [1,2,3,4,5,6,7,8,9]
```

The second column gives the information used for indexing, the third the selected subset.

First, note the significant reduction in the number of clauses which ***ProLog by BIM*** tries. Secondly, for both the calls (f) and (g) the list of selected clauses is the same as for the query (e), although the clause heads that actually match (f) and (g) are [1,6,9] and [1,9]. This is because only the principal functor is used in the indexing. Finally, this predicate is non-deterministic for all of the above calling patterns as each selection contains more than one clause.

The use of cuts

When optimizing a predicate that is used deterministically it is very important to know when the system finds out that the predicate is deterministic and when not. This information is used to place calls to the predicate !/0 if and only if they are needed. An unnecessary cut destroys the multi-directionality whereas forgetting a cut may have a dramatic effect on the memory and time consumption of your program and may render it difficult to find bugs.

Consider the previous example number two:

Example 2

```
p(1,a).
p(2,b).
p(3,c).
p(4,d).

?- p(N,b).
```

If the mode of the predicate is known to be

```
:- mode p(-,+).
```

it should be rewritten for most systems (not for ***ProLog by BIM***) as either:

Example 2a

```
p(1,a) :- !.
p(2,b) :- !.
p(3,c) :- !.
p(4,d).
```

or even better:

Example 2b

```
p(O,I) :- pp(I,O).

pp(a,1).
pp(b,2).
pp(c,3).
pp(d,4).
```

The cuts in example 2a are there to commit to a clause as soon as a first matching head is found using the backtracking mechanism. That mechanism is much slower than the indexing mechanism. Without the cuts there will be open choices left. The Prolog system must retain enough information to restart the computation from this point. A number of resources may be blocked to make backtracking possible, a clearly unwanted effect.

Example 2b shows what can be found in many programs: the predicates get twisted to allow the system to find the determinism. It is clear that if the predicate can be both called with either the first or the second argument known, the solution proposed in example 2b can no longer be applied.

5.4 Indexing in ProLog by BIM

Multiple argument indexing

The system indexes on up to three arguments, determined automatically, using the top-level functor of those arguments. As a result many more predicates can be found to be deterministic and less cuts are needed in programs.

Example 1

```
p(1,a).
p(2,b).
p(3,c).
p(4,d).

?- p(2,L).
?- p(N,b).
```

ProLog by BIM executes both queries with no backtracking. No cuts must be added. As a consequence p/2 can still be used to generate all pairs with the query:

```
?- p(N,L).
```

In the next example all queries execute deterministically.

Example 2

```
number_to_word(1,one,un) :- ...
number_to_word(2,two,deux) :- ...
number_to_word(3,three,trois) :- ...

?- number_to_word(2,E,F).
?- number_to_word(N,two,_).
?- number_to_word(N,_,trois).
?- number_to_word(_,three,trois).
```

Example 3

```
clause head                    index1index2

(1) a(1,a) :- ...              1     a
(2) a(2,b) :- ...              2     b
(3) a(3,a) :- ...              3     a
(4) a(_,c) :- ...              _     c
```

Call and selected clauses list:

```
query                          arg1:sel1arg2:sel2sel1 ⌒
sel2

?- a(_,_) .                    _:all_:all[1,2,3,4]
?- a(1,_) .                    1:[1,4]_:all[1,4]
?- a(_,a) .                    _:alla:[1,3][1,3]
?- a(4,_) .                    4:[4]_:all[4]
?- a(_,c) .                    _:allc:[4][4]
?- a(1,a) .                    1:[1,4]a:[1,3][1]
```

Here, two-argument indexing is required to make the execution of the last call deterministic. For the second and third call, indexing reduces the number of clauses to be tried. For the next to last call it is sufficient to index on the second argument only. ***ProLog by BIM*** starts with indexing on the first argument, and only if this does not lead to a single clause and indexing on the second argument may further reduce the number of selected clauses, it will index on the second argument.

Using mode declarations

Mode declarations are extremely useful to decide on which arguments to index. Clearly, it makes no sense to index on an output argument and it makes a lot of sense to index on an input argument.

Example

```
:-mode p(-,+).

p(1,a).
p(2,b).
p(3,c).
p(4,d).
```

For the above example ***ProLog by BIM*** will index on the second argument only.

Without the mode declaration it would provide all the code to index on the first argument too. Execution would start with checking the first argument to find it is free and only then use the second argument. The mode declaration reduces both the execution time and the space occupied by the code for the predicate.

Furthermore, the compiler can be sure the second argument is instantiated and therefor knows that only one alternative is selected for any call, allowing other optimizations like redundant cut removal.

Index declarations

The user can rely on the system to automatically select the indexing arguments it will use. In some cases the user may want to control the indexing. The index directive has the following forms:

```
:- predname/arity index onlyindex.
:- predname/arity index (firstindex,secondindex).
:- predname/arity index (firstindex,secondindex,thirdindex).
```

Example 1a

```
:- mode a(i,i,i,i) .

a(1,2,a,1) .
a(c,2,a,2) .
a(c,2,b,3) .
a(c,3,b,4) .
```

Default indexing uses all three first arguments. Still, it is better - and sufficient for determinism - to index on the fourth argument only:

Example 1b

```
:- mode a(i,i,i,i).
:- a/4 index 4.
a(1,2,a,1).
a(c,2,a,2).
a(c,2,b,3).
a(c,3,b,4).
```

In general, if there is an argument that uniquely determines the right clause it is better to indicate this with an **index declaration**. It may result in less code and faster execution, besides offering a valuable comment to the reader.

5.5 Extended indexing in ProLog by BIM

With the above indexing scheme a number of cases are detected when a call executes deterministically. An important number of cases is still not handled very well. The next section discusses an extension of the indexing to capture even more cases were predicates can be executed deterministically by ***ProLog by BIM***.

!/0

A call to !/0 as the first subgoal obviously gives useful information about the intended use of a predicate. The indexing in ***ProLog by BIM*** uses this information.

Example 1

```
db(john) :- !.
db(mary) :- !.
db(susan) :- !.
db(will) :- !.
db(X) :- write(not_in_db(X)) , fail.
```

The last clause is a 'catch all' clause.

The cuts are added, because we don't want to have the message printed out for persons in the database. Simple indexing would create a choice point whenever we call the predicate with the name of a person that is in the database.

In the extended indexing scheme of ***ProLog by BIM*** 4.0, the choice point is not created due to the presence of the cuts. The effect of the cuts is produced by the indexing.

The cut can be incorporated into the extended indexing only if the following **conditions** are fulfilled simultaneously:

- cut must be 'leftmost' in the body (a relaxation of this condition follows)
- unification in the head must not fail if the basic indexing on the arguments has chosen this clause (to assure the !/0 is reached)

Example 2

```
db(john) :- do_john , !.
db(mary) :- do_mary , !.
db(X) :- write(not_in_db(X)) , fail.
```

Example 2 violates the conditions under which the extended indexing can use the !/0 information: the call to the !/0 is not leftmost.

The cut need not be leftmost in a strict sense. The general rule is that to the left of the cut, only certain built-in predicates that do not impair the usefulness of cut for indexing, should appear.

Example 3

```
p(f(X),X) :- !.
p(g(X),X) :- !.
p(A,B) :- ...
```

Suppose p/2 is called with its first argument instantiated to f(1). The argument based indexing will select the first clause. The cut is useless for indexing, because the head unification can still fail, because of the presence of the second X.

```
?- p(f(1),2).
```

The unification of the first head fails and the choice point is needed to try the third clause later on.

So, it becomes important to be able to see when a head unification can fail after it has been selected on the basis of a set of indexing arguments. There is a rule of thumb: if the head contains a variable twice, or if a non-indexing argument is not a variable, then the unification can still fail. One should disregard arguments declared as output in a mode declaration from this rule.

This rule looks complicated, but fortunately, there is a better rule:

write exactly what you intend

If in example 3 the second argument is meant as an output argument, there should either be a mode declaration

```
:- mode(p(?,o)).
```

or the 'construction of the output' should be moved after the cut:

```
p(f(X),Y) :- ! , Y = X .
```

This expresses more clearly what is intended because it reads as:

if argument 1 is of the form f(X)
then commit to this clause and make argument 2 equal to X

The above form is sometimes referred to as 'steadfast'. We will return to this point later.

It is of course possible that the original definition of p/2 is exactly what you intended and then it is exactly what you should write!

Type checking built-ins

A number of type checking built-ins are considered during the indexing, if they appear as the first subgoals in the body. They are listed below:

atomic/1	is the argument atomic?
var/1	is the argument free?
nonvar/1	is the argument not free?
compound/1	is the argument a term or a non-empty list?
list/1	is the argument a non-empty list?

The extended indexing uses this information to reduce the set of candidate clauses.

Example 1a

```
mytype(X,Type) :- var(X), Type = var.
mytype(A,Type) :- atomic(A), Type = atomic.
mytype(C,Type) :- compound(C), Type = compound.
```

The type checks are mutually exclusive and hence ***ProLog by BIM*** will not create a choice point for any call to this predicate.

Example 1b

```
mytype(X,var) :- var(X).
mytype(A,atomic) :- atomic(A).
mytype(C,compound) :- compound(C).
```

Example 1b is an alternative definition for the predicate mytype/2. The argument based indexing will pick argument two as argument one does not make much sense. Unfortunately, if the intended mode is mytype(+,-) indexing on the second argument is useless.

Can indexing on the first argument be applied? The answer is no: we cannot guarantee the same semantics even not if we use the mode declaration

```
:- mode mytype(?,-).
```

because of a call like

```
?- mytype(X,X).
```

The above can be checked with your version of ***ProLog by BIM***. Start the system with showsolution mode active (this may be the default) and type in the above or another query from the top-level. if there is no choice point the system will reply Yes and prompt for a next query. If there is a choice point the system will wait after the prompt Yes.

Type checking built-ins and the cut

Example 1c

```
mytype(X,Type) :- var(X), !, Type = var.
mytype(A,Type) :- atomic(A), !, Type = atomic.
mytype(C,Type) :- /* compound(C), !, */ Type = compound.
```

The extended indexing will produce the same code for example 1c as for example 1a.

Example 2

```
p(X) :- var(X), !, ...
p(A) :- atomic(A), !, ...
p(L) :- list(L), !, ...
p(T) :- ...
```

Example 2 shows code typical for term manipulation predicates. No choice point is created by ***ProLog by BIM*** for this predicate p/1.

Example 3a

```
p(X) :- var(X), !, ...
p(a) :- ...
p(b) :- ...
p(c) :- ...
```

Example 3 shows code typical for catching unwanted calls with a variable argument. The execution will proceed without choice point creation for the predicate p/1.

Example 3b

```
p(X) :- var(X), !, ...
p(X) :- pi(X).

:-mode pi(+).

pi(a) :- ...
pi(b) :- ...
pi(c) :- ...
```

Example 3b shows how to catch the mode error and use a mode declaration.

Note that list/1 and compound/1 are non-exclusive tests. When the input is a list both succeed. If we extend example 1 to include a case for list and still maintain the intended meaning, a cut should be added. The predicate mytype/2 becomes

```
mytype(X,Type) :- var(X), Type = var.
mytype(A,Type) :- atomic(A), Type = atomic.
mytype(L,Type) :- list(L), !, Type = list.
                                  % note the cut
mytype(C,Type) :- compound(C), Type = compound.
```

Explicit unification

Example 1

```
append(X,Y,Z) :-
     X = [], Y = Z.
append(X,Y,Z) :-
     X = [H|T], Z = [H|TZ],
     append(T,Y,TZ).
```

In example 1 the calls to the built-in =/2 are explicit unifications.

First a short note: for many people, the above code looks cleaner or more natural or more attractive than the usual append/3 definition. Indeed, the above code has an explicit variable name for each argument and makes it easier to reason about arguments. The reading of the first definition is

> the relation append holds between X, Y and Z if
> X is the empty list and Y equals Z

The usual first definition of append/3:

```
append([],L,L).
```

reads more clumsily as

> append is true if
> its first argument is the empty list and
> its second argument is equal to its third argument

The reason for this more difficult reading is that there is no explicit name in the source code for each argument.

With the basic indexing scheme the query

```
?- append([1,2,3],[4],L).
```

would have created 4 choice points. That was incentive enough not to write code of the above form. With the extended indexing this situation is remedied: the code generated for the definition of append/3 with explicit unification is exactly the same as with unification in the head only. It means that novice Prolog users can stick to their habit of using explicit unification and reputed users can return to it: the compiler does not punish you for your programming style.

When is explicit unification useful for indexing?

- explicit unification is only of use for variables that occur as a top-level argument in the head.
- to be of any use for indexing, explicit unification should occur 'leftmost' in the body.
  (We have said a similar thing about cut and type tests, so there is a need for putting things together in the next section.)

Example 2a

```
p(X) :- X = [].
```

Example 2b

```
p([]).
```

Example 2a with explicit unification is equivalent to example 2b: there is no difference in the executed code.

Example 3

```
p(f(X)) :- X = g(Y).
```

In example 3 the explicit unification is not used in indexing, because X occurs inside a term and not as an argument of the head of the predicate.

Example 4

```
p(X) :- q(X), X = [].
```

In example 4 the explicit unification is not used in indexing, because it is not 'leftmost'.

Putting it all together.

The following can be taken into account for extended indexing:

- the arguments of the head
- the explicit unifications of variables occurring as arguments of the head
- type tests of variables occurring as arguments of the head
- the cut

These must occur in this order and should not be interrupted by calls of a different nature.

Example 1

```
:- mode(f(i,?,?)) .% see later why this is necessary
f(X,Y,Z) :- X = foo , atomic(Y) , list(Z) , ! .
```

taken into account: X = foo , atomic(Y) , list(Z) , !

```
f(X,Y,Z) :- X = foo , atomic(Y) , ! , list(Z) .
```

taken into account: X = foo , atomic(Y) , !

```
f(X,Y,Z) :- X = foo , atomic(Y) , bar(Z) , ! .
```

taken into account: X = foo , atomic(Y)

```
f(X,Y,Z) :- atomic(Y) , X < Z , ! .
```

taken into account: atomic(Y)

Moreover, one has to take into account the 'possible unification' (failure and success) rules:

- if unification can fail before a certain predicate is reached, this predicate cannot be used for extended indexing
- if unification can 'change' the value of a type tested variable, this type test cannot be used for extended indexing - this is why the mode declaration is necessary in the above examples

Example 2

```
a(X,X) :- var(X) , ! , ...
```

Before the type test var(X) is reached, X can because of the unification of both arguments, become instantiated, so the var/1 test is not used for indexing.

5.6 Explicit unification

In the previous section we already explained that explicit unification can be useful to improve the indexing. Explicit unification also makes your program more efficient:

Example 1

```
sorted([X,Y|R]) :- X < Y, sorted([Y|R]).
```

in the recursive call, the term [Y|R] is built, although it was present as a part of the input argument; so you can write:

```
sorted([X|Tail]) :- Tail = [Y|R], X < Y, sorted(Tail).
```

and have a more efficient program.

5.7 Auxiliary predicates

Some determinism remains undetected because only the top-level functor is used for indexing. This can be remedied by introducing extra predicates to have a second layer of indexing.

Example

```
a(s(a)) :- ... .
a(s(b)) :- ... .
a(t(a)) :- ... .
a(t(b)) :- ... .
```

can be rewritten as:

```
a(s(_X)) :- a1(_X) .
a(t(_X)) :- a2(_X) .
a1(a) :- ... .
a1(b) :- ... .
a2(a) :- ... .
a2(b) :- ... .
```

In many cases it is possible to choose meaningful names for these auxiliary predicates, as the similarity in data structure suggests some similarity in meaning.

5.8 Garbage collection of Prolog terms

A garbage collector is an essential component of any Prolog system, since almost every program creates terms that are no longer needed later on. Where is the garbage coming from, and how are programs optimized by controlling the creation of garbage? These two questions are treated in this section.

The origin of garbage

The garbage here consists of terms which were once constructed, but are no longer needed. In ***ProLog by BIM***, lists and terms are allocated on the heap, so this topic is about heap garbage collection. The selection of the base size of the heap and expansion information is user selectable, and described in the reference manual.

Example

```
a(_X,_Y) :- b(_X,_Z), c(_Z,_Y) .
```

In this predicate, the variable _Z is a link between b/2 and c/2. Whatever its value may be, when a/2 has finished, it is no longer needed. If _Z got instantiated to a complex term, this term is probably garbage (probably, because sharing may occur between the values of _X,_Y and _Z). As _Y is constructed after _Z, immediate deallocation of the space used by _Z is non-trivial. ***ProLog by BIM*** does not attempt to do this. An important class of predicates which create garbage are predicates used to update a Prolog term. As such an update means creating a new term with a different part, the old part becomes garbage.

Example

```
:-mode delete(i,i,o) .
delete([_X|_L],_X,_L) .
delete([_Y|_L],_X,[_Y|_Dl]) :-
     delete(_L,_X,_Dl) .
```

For some instances of the update problem, using the technique of open-ended structures avoids the need to make updates and limits garbage creation. One example is the use of a version list: the actual value is the last element in the open-ended list. This makes sense only if this version list is part of a large structure. Another example, often published, is how to avoid the use of append/3 alias concat/3 through the use of difference lists - an instance of open-ended structures. Garbage is avoided, since most often the first argument of append/3 becomes garbage.

Garbage and backtracking

The above discussion is not complete because the terms which may no longer be needed in one alternative, may be needed in another. Equally, garbage which was created is cleaned up without garbage collection by backtracking past the point of creation.

This means that whenever backtracking is still possible, many terms which are no longer needed in the current alternative, may be kept even when garbage collection is finished. Because of that, a forgotten green cut can cause unexpected behavior, as far as heap usage is concerned.

On the other hand, one can make use of the automatic cleanup of garbage through failure to avoid garbage collection, or at least to delay it.

Example

```
a(_X) :- test_1(_X), !, b(_X) .
a(_X) :- c(_X) .
```

Suppose test_1/1 is creating garbage internally. Suppose that in most cases test_1/1 succeeds and b/1 is called. The following version generates less garbage:

```
a(_X) :- neg_test1(_X), !, c(_X) .
a(_X) :- b(_X) .
```

A comparison is not easy. The first version will run faster if one assumes that both test_1/1 and neg_test1/1 have the same speed, unless garbage collection is caused by the extra amount that got created. Which is best cannot be answered in general.

5.9 Specific versus general code

In the design stage it is natural to write general code which solves more than the problem at hand, especially in Prolog. If optimization indicates that part of the code is time critical or memory consuming, it can be worthwhile to write specialized code. A well-known example of this technique from other programming languages is division by a power of two. Division is expensive compared to a shift, and will use this more specific operation. In Prolog there are several possibilities to use more specific code.

Example

```
member(_X,[_X|_L]) .
member(_X,[_|_L]) :- member(_X,_L) .
?- member(_X,[1,2,3]) .
```

becomes:

```
member_123(1) .
member_123(2) .
member_123(3) .
?- member_123(_X) .
```

Example

```
forterm(_Term,_Cnt) :-
     functor(_Term,_,_Arity),
     foreacharg(1,_Arity,_Term,0,_Cnt) .
foreacharg(_ArgNr,_Arity,_Term,_CntIn,_CntOut) :-
     _ArgNr =< _Arity, !,
     arg(_ArgNr,_Term,_Value),
     _CntTmp is _CntIn + _Value,
     _NewArgNr is _ArgNr + 1,
     foreacharg(_NewArgNr,_Arity,_Term,_CntTmp,_CntOut) .
foreacharg(_ArgNr,_Arity,_Term,_CntOut,_CntOut) .
?- forterm(u(1,2,3,4),_Cnt) .
?- forterm(t(1,2,3),_Cnt) .
```

If all calls to forterm/2 have as first argument a term u/4 or t/3, a specialization is shown in the following code:

```
forterm(_Term,_Cnt) :-
     functor(_Term,_Functor,_),
     forterm_t(_Functor,_Term,_Cnt) .
:- mode forterm_t(i,i,?) .
forterm_t(u,_Term,_Cnt) :-
     foreacharg(u,[1,2,3,4],_Term,0,_Cnt) .
forterm_t(t,_Term,_Cnt) :-
     foreacharg(t,[1,2,3],_Term,0,_Cnt) .
:- mode select(i,i,i,o) .
select(u,1,u(_Value,_,_,_),_Value) .
select(u,2,u(_,_Value,_,_),_Value) .
select(u,3,u(_,_,_Value,_),_Value) .
select(u,4,u(_,_,_,_Value),_Value) .
select(t,1,t(_Value,_,_),_Value) .
select(t,2,t(_,_Value,_),_Value) .
select(t,3,t(_,_,_Value),_Value) .
foreacharg(_Func,[_ArgNr|_ArList],_Term,_CntIn,_CntOut) :-
     select(_Func,_ArgNr,_Term,_Value),
     _CntTmp is _CntIn + _Value,
     foreacharg(_Func,_ArList,_Term,_CntTmp,_CntOut) .
foreacharg(_Func,[],_Term,_CntOut,_CntOut) .
?- forterm(u(1,2,3,4),_Cnt) .
?- forterm(t(1,2,3),_Cnt) .
```

The options used were:

- The functor determines the arity.
- The selection list avoids the choice point in "foreacharg" and turns calculation into list manipulation for stepping through the arguments.
- The **arg/3** predicate is avoided with a special select/3 predicate.

Not all of these options have the same effect. The special select predicate is questionable in this case, but is shown here because it might be advantageous in other situations.

If all calls were with a term u/4, the predicates become a lot simpler:

```
forterm(_Term,_Cnt) :-
     foreacharg([1,2,3,4],_Term,0,_Cnt) .
:- mode select(i,i,i,o) .
select(1,u(_Value,_,_,_),_Value) .
select(2,u(_,_Value,_,_),_Value) .
select(3,u(_,_,_Value,_),_Value) .
select(4,u(_,_,_,_Value),_Value) .
foreacharg([_ArgNr|_ArList],_Term,_CntIn,_CntOut) :-
     select(_ArgNr,_Term,_Value),
     _CntTmp is _CntIn + _Value,
     foreacharg(_Func,_ArList,_Term,_CntTmp,_CntOut) .
foreacharg([],_Term,_CntOut,_CntOut) .
?- forterm(u(1,2,3,4),_Cnt) .
```

Although this looks like a big improvement, the final step is, of course:

```
forterm(u(_P,_Q,_R,_S),_Cnt) :-
     _Cnt is _P + _Q + _R + _S .
forterm(t(_P,_Q,_R),_Cnt) :-
     _Cnt is _P + _Q + _R .
?- forterm(u(1,2,3,4),_Cnt) .
?- forterm(t(1,2,3),_Cnt) .
```

This is much simpler.

5.10 Partial evaluation

Partial evaluation is an optimization technique by which improved execution is obtained by executing parts of the program at compile time instead of at run-time. In a classical setup, one could consider the evaluation of arithmetic expressions not containing variables to a number, as an instance of partial evaluation. In Prolog, making some inferences at compile time is an aspect of partial evaluation.

There is a fair amount of literature on partial evaluation systems. In a nutshell, the results are that significant increases in speed are possible if you start with bad code, reasonable increases can be obtained on readable and reasonable code. Care must be taken with partial evaluation, because it must not get in the way of the Prolog compiler. The compiler and partial evaluator must communicate to achieve real improvements. Useful examples are:

- Partial evaluation to specialize the predicate to the call.
- Evaluating away initializing predicates.
- Evaluating away wrappers of arguments.

Note that the simplistic measure of reducing the number of inferences is very misleading. One may well substitute simple for expensive inferences.

Creating a predicate with the matching clauses only

Example

```
transform(s(_X),_Y) :- p(_X,_Y) .
transform(t(_X),_Y) :- q(_X,_Y) .
transform(_X,_Y) :- r(_X,_Y) .
p(_X,_Y) :- transform(t(_X),_Y) .
```

becomes:

```
transform_2(_X,_Y) :- q(_X,_Y) .
transform_2(_X,_Y) :- r(t(_X),_Y) .
p(_X,_Y) :- transform_2(_X,_Y) .
```

Example

```
transform(s(_X),_Y) :- t_s(_X,_Y) .
transform(t(_X),_Y) :- t_t(_X,_Y) .
p(_X,_Y) :- transform(t(_X),_Y) .
```

becomes:

```
p(_X,_Y) :- t_t(_X,_Y) .
```

Eliminating initialization predicates

Example

```
p(_X) :- p(_X,[]) .
p(...,...) :- ... .
q :- p(_X) .
```

becomes:

```
p(...,...) :- ... .
q :- p(_X,[]) .
```

Eliminating structures

Example

```
p(f(_X,_Y)) :- ... .
q :- p(f(a,b)) .
```

becomes:

```
p_2(_X,_Y) :- ... .
q :- p_2(a,b) .
```

5.11 Avoiding duplication of work

The usual way to write the member_check/2 predicate is:

```
member_check(_X,[_X|_L]) :- ! .
member_check(_X,[_|_L]) :- member_check(_X,_L) .
```

This predicate lends itself nicely to show how duplication of work can be avoided by writing the predicate in the following way:

```
member_check(_X,[_Y|_L]) :- member_check(_X,_Y,_L) .
member_check(_X,_X,_L) :- ! .
member_check(_X,_,[_Y|_L]) :- member_check(_X,_Y,_L) .
```

The interesting differences are:

- On advancing in the list, the splitting of the list in its head and tail is performed exactly once.
- The end of the list is detected one step earlier.
- The arity of the recursive predicate has increased by one.

The first two differences are advantageous, the last is disadvantageous. In this case, the net result is an improvement in speed.

Another instance of this optimization is related to multiple clauses versus disjunctions.

Example

```
a(f(_X,_Y), g(_U,_V)) :- _X > _U, !, b(_Y,_V) .
a(f(_X,_Y), g(_U,_V)) :- b(_V,_Y) .
```

versus

```
a(f(_X,_Y), g(_U,_V)) :-
      (
            _X > _U, !, b(_Y,_V)
      ;
            b(_V,_Y)
      ) .
```

The effectiveness of this optimization depends on many factors, such as the arity of the predicate, the complexity of the terms, and the number of subgoals in the clauses.

Common subterm grouping is yet another example:

```
a(_X,_Y,f(_X,_Y)) :- b(f(_X,_Y)) .
```

versus:

```
a(_X,_Y,_T) :- _T = f(_X,_Y), b(_T) .
```

The speed difference is determined by different factors, such as the calling mode of the argument, the effect on indexing, the number of times the common term occurs, the complexity of the term, the chance of failure of head-unification. Future implementations can be expected to capture many cases of this optimization.

5.12 Tail recursion

Example

```
sumlist([],0) .
sumlist([_E|_L],_Total) :-
      sumlist(_L,_Partial),
      _Total is _Partial + _E .
```

versus:

```
sumlist(_L,_Total) :- sumlist(_L,_Total,0) .
sumlist([],_Total,_Total) .
sumlist([_E|_L],_Total,_Partial) :-
      _NewPartial is _Partial + _E,
      sumlist(_L,_Total,_NewPartial) .
```

In this case, the effect is magnified because in the second version the second clause of sumlist/3 does not need an environment, because ***ProLog by BIM*** treats calls to the built-in **is/2** in a special way.

Counter example

Version 1:

```
delete([_X|_L],_X,_L) .
delete([_Y|_L],_X,[_Y|_DL]) :- delete(_L,_X,_DL) .
permute([],[]) .
permute(_L,[_X|_P]) :-
      delete(_L,_X,_DL),
      permute(_DL,_P) .
```

This straightforward definition is tail-recursive, but cannot make use of indexing on the first or second argument.

Version 2:

```
delete([_X|_L],_X,_L) .
delete([_Y|_L],_X,[_Y|_DL]) :- delete(_L,_X,_DL) .
permute([],[]) .
permute([_Y|_L],[_X|_P]) :-
      delete([_Y|_L],_X,_DL),
      permute(_DL,_P) .
```

ProLog by BIM indexes on the first argument of permute in this version, but not in the previous version. This is a counter example of common subterm elimination: a common subterm, [_Y|_L] is introduced.

Version 3:

```
insert(_X,_L,[_X|_L]) .
insert(_X,[_Y|_DL],[_Y|_L]) :- insert(_X,_DL,_L) .
permute([],[]) .
permute([_X|_L],_P) :-
     permute(_L,_PL),
     insert(_X,_PL,_P) .
```

Version 3 is not tail-recursive, but is nevertheless much faster than both version 1 and 2. The explanation is not simple, but worthwhile.

Some observations:

- insert/3 and delete/3 are actually the same predicates, with different names and argument order:

```
delete(_LL,_E,_SL) :- insert(_SL,_E,_LL) .
```

or:

```
insert(_SL,_E,_LL) :- delete(_LL,_E,_SL) .
```

- Both insert/3 and delete/3 consist of a fact and a clause which has one subgoal: a call to itself. These predicates are responsible for the multiple solutions; that is, they are the backtracking predicates.
- For each call to permute, the matching clause can be determined through indexing, permute/2 is deterministic.
- Version 1 and 2 interleaves calls to delete and to permute, backtracking over calls to permute and calls to delete.
- Version 3 first calls permute, then calls insert, backtracking over calls to insert.
- When calls to permute have to be re-satisfied in version 1 and 2, this among other things requires setting up an environment.

From these observations it should be clear why versions 1 and 2 are that much more expensive: more work is done after backtracking than in the third version. Actually, there is a general idea behind this, related to something which is well-known from imperative languages: move loop invariant code out of the loop. In the Prolog context, this becomes: move work which is independent of the selected alternative to a point before the call to the predicate which selects the alternatives.

From this example, one should remember: making a predicate tail-recursive is *not* unconditionally the best way of programming it. It *is* right to use tail-recursive predicates if all the predicates involved are deterministic. If some predicates are not deterministic, try to put the backtracking predicates at the end of the conjunction.

5.13 Argument order

The following optimization is particular to WAM-inspired implementations, such as ***ProLog by BIM***. For such implementations, the following code is cheap:

```
p(_A,_B,_C,_D,...,_Z) :- q(_A,_B,_C,_D,...,_Z) .
```

It costs the same as the following piece of code:

```
p :- q .
```

If an argument of the head is the same variable as an argument of the first *real* subgoal, there is no cost in WAM-inspired implementations. What a *real* subgoal is, is implementation dependent. Arithmetic operations, arithmetic comparisons, explicit unification and !/0 are the most important examples in ***ProLog by BIM*** which are *not* real subgoals.

Whenever a predicate uses accumulating parameters, it is a good idea to consider adding them at the end. This is illustrated with two versions of list_length/2 in terms of list_length/3:

Version 1:

```
list_length(_L,_N) :- list_length(_L,0,_N) .
```

Version 2:

```
list_length(_L,_N) :- list_length(_L,_N,0) .
```

This looks like a small difference, but there is a more extreme example:

```
a([_X|_L],...) :- b(_X,_L,...) .
```

versus:

```
a([_X|_L],...) :- b(_L,...,_X) .
```

In the first version, all arguments have to be moved up one position in the argument list. The second version avoids these moves entirely. Obviously, careless placement of arguments can lead to an extra factor of complexity in a program. In the worst case, this factor equals the arity of the predicates involved.

5.14 Failure driven loops

Here is an example of space optimization. The simplest example is a program which continuously reads data, performs some action on the data, and provides the result(s):

```
p :- read(_Data), process(_Data,_Result),
          output(_Result), p .
```

As all complex data are allocated on the heap, the heap will be filled quickly with a program of this structure. The alternative is to use a failure driven loop:

```
p :- read(_Data), process(_Data,_Result), output(_Result),
          fail .
p :- p .
```

It is clear that a logical reading of such a predicate is difficult.

In a similar spirit, consider the following program:

```
p([],[]) .
p([_Data|_In],_) :-
     process(_Data,_Result), record(p_result,_Result),
          fail .
p([_Data|_In],[_Result|_Out]) :-
     recorded(p_result,_Result), erase(p_result),
          p(_In,_Out) .
```

There is a time penalty for both the copying to/from the record heap, and the backtracking. There is less space used, possibly giving better paging behavior and avoidance of garbage collection. It has the disadvantage of creating garbage on the record heap, for which garbage collection is the only way to recover the garbage.

5.15 Replacing arithmetic with list processing

The demonstration program is:

```
subl_n(_L,_N,_S) : _S is a sublist of length _N of the list _L
```

The first version uses arithmetic:

```
subl_n(_L,0,[]) :- ! .
subl_n([_X|_L],_N,[_X|_S]) :-
     subl_n(_L,?(_N-1),_S) .
subl_n([_X|_L],_N,_S) :-
     subl_n(_L,_N,_S) .
```

This version first constructs a list of the correct length and then instantiates the elements:

```
subl_n(_L,_N,_S) :-
     makelist(_N,_S),
     subl_n_i(_L,_S) .
makelist(0,[]) :- ! .
makelist(_N,[_|_S]) :-
     makelist(?(_N-1),_S) .
subl_n_i(_L,[]) .
subl_n_i([_X|_L],[_X|_S]) :-
     subl_n_i(_L,_S) .
subl_n_i([_X|_L],[_E|_S]) :-
     subl_n_i(_L,[_E|_S]) .
```

For finding all solutions the second version is noticeably faster.

5.16 Explicit negation instead of not/1 or \+/1

Many programs contain the call not member(_X,_L). One could program the predicate not_member instead. This can improve execution time considerably.

```
member(_X,[_Y|_L]) :- member(_X,_Y,_L) .
member(_X,_X,_L) .
member(_X,_,[_Y|_L]) :- member(_X,_Y,_L) .
?- not member(1,[2,3,4,5]) .
```

versus:

```
not_member(_X,[_Y|_L]) :- not_member(_X,_Y,_L) .
not_member(_X,[]).
not_member(_X,_X,_L) :- !, fail .
not_member(_X,_Y,[]) .
not_member(_X,_,[_Y|_L]) :- not_member(_X,_Y,_L) .
?- not_member(1,[2,3,4,5]) .
```

This is especially useful if all calls to member/2 are actually negated calls. On the other hand, when the call and the negated call are used, an extra coding effort is required, and two versions need to be maintained.

5.17 Mode declarations

ProLog by BIM allows the user to give mode declarations. Unlike other systems, ***ProLog by BIM*** uses the declared modes both while debugging and for performance improvements. Apart from this, it makes the program more readable and better documented. It makes sense to start writing a predicate with the declaration of its modes.

The mode declaration looks like:

```
:- mode predname(mode,mode,mode,...) .
```

where mode is one of *i*(+) *o*(-) *?*(*?*). The notation between brackets is used when *compatibility* syntax is turned on.

The meaning of the modes is:

i : + : input
This argument should be sufficiently instantiated. Ground is always sufficiently instantiated. The debugger produces warning messages if the actual argument is not ground.

o : - : output
This argument must be a free variable in the actual call.

? : *?* : unknown
This argument may be anything.

5.18 Minimize the overhead of backtracking

The cost of setting up the Prolog backtracking is high. It is worth paying attention to this fact when writing backtracking predicates. The following example is taken from a simple meta-program:

Version 1:

```
meta_predicate((_A;_B)) :- !,
      (
            meta_predicate(_A)
      ;
            meta_predicate(_B)
      ) .
meta_predicate((_A,_B)) :- !,
      meta_predicate(_A),
      meta_predicate(_B) .
meta_predicate(_C) :-
      clause(_C,_B),
      meta_predicate(_B) .
```

Version 2:

```
meta_predicate((_A;_B)) :-
      meta_predicate(_A) .
meta_predicate((_A;_B)) :- !,
      meta_predicate(_B) .
meta_predicate((_A,_B)) :- !,
      meta_predicate(_A),
      meta_predicate(_B) .
meta_predicate(_C) :-
      clause(_C,_B),
      meta_predicate(_B) .
```

In version 1, ***ProLog by BIM*** creates a choice point for meta_predicate/1. If the goal which is meta-interpreted is a disjunction, ***ProLog by BIM*** cuts the choice point for meta_predicate/1 away, and immediately creates a new choice point for the disjunction in the body of the first clause. In version 2, only the choice point for meta_predicate/1 is created. On the other hand, the unification of the goal with the term (_A;_B) is repeated. Unification is a cheap operation compared to choice point creation, so the net result is positive.

The reader is warned that the above meta-interpreter is used to illustrate the technique to minimize the overhead of backtracking, but the above example is not the recommended prototype of a fast meta-interpreter.

5.19 The If-Then-Else

The if-then-else construction in Prolog is written with syntax

```
Condition -> Then ; Else
```

In general this is syntactic sugar for

```
(Condition , ! , Then ; Else)
```

(apart form the scope of the introduced cut) and at run-time, a choice point was always created, even if the Condition was 'simple'.

In ***ProLog by BIM*** from 4.0 on, simple conditions in an if-then-else, are treated in a more efficient way and lead to much improved efficiency. The simple conditions are the following:

true/0
fail/0
!/0
var/1
nonvar/1
atom/1
atomic/1
number/1
real/1
pointer/1
integer/1
compound/1
list/1
ground/1
is_inf/1
@</2
@=</2
@>/2
@>=/2
>/2
</2
>=/2
=</2
term_type/2
=/2 (restricted form)
==/2
any conjunction of the above

The optimisation of =/2 in a condition of if-then-else amounts to the restriction that general unification is not considered simple, i.e.

X = foo	simple
X = f(_)	simple (a void variable needs no general unification)
X = f(Y)	not simple
X = Y	not simple

As an example of an if-then-else with a simple condition:

```
a(X,Y,Z) :-
      ((var(X) , Y < Z) ->
            write(ok)
      ;
            write(not_ok)
      ).
```

Knowing this optimisation makes it attractive to change some existing code, e.g. code like for the split/4 predicate (used in the quicksort program):

```
split(X,[],[],[]) .
split(X,[Y|R],[Y|Small],Large) :-
      Y < X , ! ,
      split(X,R,Small,Large) .
split(X,[Y|R],Small,[Y|Large]) :-
       split(X,R,Small,Large) .
```

can be changed to

```
split(X,[],[],[]) .
split(X,[Y|R],Small,Large) :-
      (Y < X ->
            Small = [Y|S] , split(X,R,S,Large)
      ;
            Large = [Y|L] , split(X,R,Small,L)
      ).
```

and increase speed dramatically.

Note:

- Since other implementations do not optimize if-then-else, or because your programming style is different, you might be inclined to write (or have written) clauses like:

  ```
  a(X,Y,Z) :- X < Y , ! , Z = Y ; Z = X .
  ```

  There is no need to change this to the form:

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

  because the system itself does the conversion internally whenever that makes sense. Again, you need not adapt your programming style because of the new feature!

If-then-else compared to indexing

Since indexing takes into account type tests and explicit unification, and also if-then-else optimizes on them, one can wonder which is more efficient to use. The question is similar to the one in an imperative language like C: should I use a cascade of if-then-else or a case statement? The complete answer is technical and depends on execution characteristics of your program. As a rule of thumb: a cascade of 4 or more if-then-else, might better be to replaced by a case statement, not just for efficiency reasons, but also for reasons of legibility.

User's Guide

Chapter 6
Using the External Language Interface

In this chapter, different aspects of the external language interface of ***ProLog by BIM*** are illustrated. A single problem is solved in six ways. Each demonstrates one technique for hybrid programming. The source of the programs is included on the distribution tape. It can be found in the directory $BIM_PROLOG_DIR/examples/xtern/c.

6.1 Hybrid programming

The term *hybrid programming* denotes all programming in which more than one programming language is involved. For ***ProLog by BIM***, this is any combination of Prolog programs with one of the conventional languages that can be linked with ***ProLog by BIM***: C, FORTRAN and Pascal (and assembler language).

Hybrid programs can range from Prolog programs with some external code to external programs with some Prolog code. At one end of the range, there is the case of Prolog programs with some external predicates for specific tasks. These external predicates are usually meant to perform tasks for which Prolog is not too well suited. The real hybrid systems are those where the program is equally spread over Prolog and external code. In such programs, data as well as code can be situated in the Prolog and in the external part. It may also happen that the code in one part uses the data in the other part. Finally, there are the embedded systems, in which the Prolog program is only a small part of a large system.

When dealing with existing systems that must be extended, it is often clear what to write in Prolog and what to write in an external language. It may be an existing Prolog program to which a certain module must be added. If possible, this new module will be written in Prolog, but if it is easier to write it in some other, conventional language, the program will become a hybrid system. When the existing system is mainly written in an external language, the new module could be written in Prolog because it is easier to solve the problem in a logic language.

With a mixed system, or when a new system is being designed, some questions may arise :

- Where should the data be located?
- What parts of the algorithm must be written in what language?
- Is it possible to transfer data between both languages?
- How easy is it to use external program sections in Prolog, and vice versa?

This chapter provides answers to these questions.

6.2 Problem description

Consider a predicate listing the contents of a directory. It is given a file name in which the wildcard * may appear. The desired result is a list of all files that match the given pattern. A file matches the pattern if it has a name that can be formed from the given file name by replacing the wildcard with some text. For each returned file, the following information must be returned: mode, user id, group id, file size and file name. This is called *ls* after the similar Unix command.

It turns out that this predicate can have multiple solutions for a specified query, because of the wildcard in the pattern. Depending on the application that uses the predicate, it might be necessary to have all solutions available at once, or only one at a time. When the solutions are required one at a time, they can be processed immediately. And this processing may involve another, nested call of the predicate (for example to scan subdirectories). This may pose additional requirements on the implementation of the predicate. Certain implementations do not allow nesting. A possible case where the solutions are needed all at once in a collection, is when the set of solutions must be processed as a whole (for example sorted) before each solution is further processed.

Criteria for choosing between different solutions include:

- Are the solutions needed one by one, or collected?
- Should it be possible to have nested calls of the predicate?
- Is closed memory management required?

The question of memory management is due to the fact that the result of the predicate is a complex (non-trivial) data structure, which means the implementation has to provide memory for storing that data. And of course, when the result is consumed, the memory must be made available again. With closed memory management, no memory leaks will be present: everything that is not used any longer is returned to the pool of available memory. An implementation with a memory leak may cause the program to terminate because of memory faults after repeated use of the predicate.

In the following sections, different implementations of this problem are given. The criteria above are discussed for each solution. How the predicate is used for each solution, is illustrated with an example application. The example is the predicate:

ls_print/1

ls_print(_Pattern)

arg1 : ground : atom

The list of files matching pattern *arg1* is printed out on the output stream. For each file, a line is printed consisting of mode, user id, group id, file size and file name.

This query prints all ***ProLog by BIM*** source files in the current directory:

```
?- ls_print( '*.pro' ) .
-rw-r--r--  719 12                    5053  lsx.pro
-rw-r--r--  719 12                    4295  ls.pro
```

6.3 Common part of solution

The first solution presented only uses external predicates from the C library. All other solutions are implemented partly in Prolog and partly in C. The kernel of the ls routine is written in C, and is the same for all solutions (except the first of course). It is built in the form of an iterator. There are three routines:

***int* ls_initialize(*arg* , *lsi*)**
char * arg;
LS_ITER * lsi;

An iterator, pointed to by *lsi*, is initialized for the look-up of files matching pattern *arg*. The routine returns *TRUE* if the iterator is successfully initialized. It returns *FALSE* if the directory that must be listed, cannot be opened.

***int* ls_next(*lsi* , *lse*)**
LS_ITER * lsi;
LS_ENTRY * lse;

The next entry for iterator *lsi* is returned in the structure pointed to by *lse*. If there are no further entries matching the restrictions of the iterator, the routine returns *FALSE*, otherwise it returns *TRUE*.

ls_terminate(*lsi*)
LS_ITER * lsi;

The iterator *lsi* is terminated.

There are also two common Prolog predicates. They are used in the example application for printing one solution, and for checking the return of an external predicate.

ls_print_line/5

ls_print_line(_Mode , _UserId , _GroupId , _FileSize , _FileName)

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

The solution with fields *arg1* to *arg5* is printed on the output stream.

ls_check_return/1

ls_check_return(_Return)

arg1 : ground : integer

The return code *arg1* is checked on 0 value. If it is not 0, an error message is printed.

The following sections contain the implementation of these C routines and Prolog predicates. From the perspective of illustrating the usage of the external language interface, this code is not so important. It is included for completeness and can easily be skipped without affecting the understanding of the rest of the code.

Structure definitions

The following structures are defined. They are used by all solutions that use the common external part.

```
#include <sys/param.h>
#include <dirent.h>

#define FALSE 0
#define TRUE 1

typedef struct {
     DIR *   ls_dirdesc;
     char    ls_path[MAXPATHLEN];
     char    ls_suffix[MAXPATHLEN];
     char *  ls_fname;
     int     ls_prefixlen;
     int     ls_suffixlen;
} LS_ITER;

typedef struct {
     int     ls_uid;
     int     ls_gid;
     int     ls_size;
     char    ls_mode[11];
     char    ls_name[MAXPATHLEN+1];
} LS_ENTRY;
```

The iterator structure contains the open directory descriptor and information concerning the pattern. This consists of the whole path, the prefix and the suffix before and after the wildcard, and a length indication of both. This makes it possible to distinguish the special case when no wildcard is used, compared to the case where there is no suffix.

Iterator routines

This section contains the code for the three iterator routines, ls_initialize(), ls_next() and ls_terminate().

The initialization routine sets up the iterator. It parses the given argument for the path, file name, prefix wildcard and suffix. Finally, the requested directory is opened.

```
#include <sys/types.h>
#include <sys/stat.h>
#include "ls.h"

int ls_initialize( arg , lsi )
char *arg;
LS_ITER *lsi;
{
     register char c, *src, *dst;

     /* Parse argument for root file name prefix and suffix */
     src = arg;
     dst = lsi->ls_path;
     *dst++ = '.'; *dst++ = '/';
     while ( ( c = *src++ ) && c != '*' ) *dst++ = c;
     *dst = 0;
     lsi->ls_prefixlen = dst - lsi->ls_path;
     while ( *--dst != '/' ) ;
     lsi->ls_fname = dst;
     lsi->ls_prefixlen -= dst - lsi->ls_path;
     if ( c )
     {
          dst = lsi->ls_suffix;
          while ( *dst++ = *src++ ) ;
          lsi->ls_suffixlen = dst - 1 - lsi->ls_suffix;
     }
     else
     {
          lsi->ls_suffixlen = -1;
     }
     /* Open directory */
     *lsi->ls_fname = '\0';
     if ( ! ( lsi->ls_dirdesc = opendir(lsi->ls_path) ) )
          return(FALSE);
     *lsi->ls_fname = '/';
     return(TRUE);
} /* ls_initialize */
```

The routine for retrieving the next solution uses the information in the iterator to search for the next matching entry in the directory. It continuously reads a directory entry until either a matching entry is found or the end of the directory is reached.

If a matching entry is found, its file status is requested to retrieve the required information about the file. The mode bits are converted to textual form.

```
int ls_next( lsi , lse )
LS_ITER *lsi;
LS_ENTRY *lse;
{
     struct dirent *de;
     struct stat statbuf;
     register char c, *src, *dst;
     register int mode;
     int ftype;

     /* Walk through directory */
     while ( de = readdir(lsi->ls_dirdesc) )
     {
          if ( ! de->d_fileno ) continue;
          src = de->d_name;
```

```
            /* Check prefix */
            dst = lsi->ls_fname;
            while ( ( c = *++dst ) && c == *src++ ) ;
            if ( c ) continue;
            /* Check suffix */
            if ( lsi->ls_suffixlen >= 0 )
            {
                                    src = de->d_name +
de->d_namlen - lsi->ls_suffixlen;
                                    dst = lsi->ls_suffix;
                                    while ( ( c = *dst++ ) && c ==
*src++ ) ;
            }
            if ( c || *src ) continue;
            /* Check file access */
            dst = lsi->ls_fname + lsi->ls_prefixlen;
            src = de->d_name + lsi->ls_prefixlen - 1;
            while ( *dst++ = *src++ ) ;
            if ( stat(lsi->ls_path,&statbuf) )
                                    return(FALSE);
            /* Process entry */
            lse->ls_uid = statbuf.st_uid;
            lse->ls_gid = statbuf.st_gid;
            lse->ls_size = statbuf.st_size;
            mode = statbuf.st_mode;
            dst = lse->ls_mode;
            ftype = mode & S_IFMT;
            *dst++ = ( ftype == S_IFDIR ) ? 'd' : ( ftype ==
S_IFLNK ) ? 'l' : '-';
            *dst++ = ( mode & S_IREAD ) ? 'r' : '-';
            *dst++ = ( mode & S_IWRITE ) ? 'w' : '-';
            *dst++ = ( mode & S_IEXEC ) ? 'x' : '-';
            mode <<= 3;
            *dst++ = ( mode & S_IREAD ) ? 'r' : '-';
            *dst++ = ( mode & S_IWRITE ) ? 'w' : '-';
            *dst++ = ( mode & S_IEXEC ) ? 'x' : '-';
            mode <<= 3;
            *dst++ = ( mode & S_IREAD ) ? 'r' : '-';
            *dst++ = ( mode & S_IWRITE ) ? 'w' : '-';
            *dst++ = ( mode & S_IEXEC ) ? 'x' : '-';
            *dst = '\0';
            strcpy( lse->ls_name , de->d_name );
            *(lsi->ls_fname+lsi->ls_prefixlen) = '\0';
            return(TRUE);
        }
        return(FALSE);
    } /* ls_next */
```

The termination routine closes the directory, whose descriptor is stored in the iterator data structure.

```
    ls_terminate( lsi )
    LS_ITER *lsi;
    {
        closedir(lsi->ls_dirdesc);
    } /* ls_terminate */
```

This predicate is used to print a single solution:

```
ls_print_line( _Mode , _Uid , _Gid , _Size , _Name ) :-
     printf( '%s ' , _Mode ),
     printf( '%4d ' , _Uid ),
     printf( '%4d ' , _Gid ),
     printf( '%8d ' , _Size ),
     printf( '%s\n' , _Name ) .
```

This predicate checks the return code of an external predicate:

```
ls_check_return( 0 ) :- ! .
ls_check_return( _Ret ) :-
     printf( stderr , 'Error %d in external routine.\n' , _Ret
),
     fail .
```

6.4 Solution 1: all Prolog predicates

The problem can be solved **completely in Prolog, except for the system calls.** As can be seen from the code, this involves a lot of string manipulation. In addition, there is a need for a whole set of external predicates. These provide the access to the basic system calls that are needed to open a directory, read entries from it, and close it. The read entry must be investigated. For each matching entry, the file status must be retrieved in order to collect the requested information about the file. Therefore, a stat structure must be allocated. This is done only once, and the handle to that structure is recorded.

To allow for nesting, the stat structure cannot be recorded. There must be one such structure for each nested activation of the predicate. The best solution in this case is to pass the stat structure handle as a parameter.

The data handling is complex, and so is the Prolog control structure.

An additional burden is the conversion of the mode bits to a textual form. This requires another bunch of string manipulation code.

The ***ProLog by BIM UnixFileSys*** library is used for the system call external predicates.

```
:- include( '$BIM_PROLOG_DIR/include/UnixFileSys.mod' ) .
ls1_print( _Arg ) :-
     ls1_parse_target( _Arg , _Path , _Target ),
     opendir( _Dir , _Path ),
     create_stat( _Stat ),
     rerecord( ls1_print_stat , _Stat ),
     ls1_print_loop( _Dir , _Target ),
     closedir( _Dir ),
     destroy_stat( _Stat ),
     erase( ls1_print_stat ) .
ls1_print_loop( _Dir , _Target ) :-
     ls1_next_match( _Dir , _Target , _Mode , _Uid , _Gid ,
_Size , _Name ), !,
     ls_print_line( _Mode , _Uid , _Gid , _Size , _Name ),
     ls1_print_loop( _Dir , _Target ) .
ls1_print_loop( _Dir , _Target ) .
ls1_next_match( _Dir , _Target , _Mode , _Uid , _Gid , _Size
, _Name ) :-
     readdir( _DirEnt , _Dir ),
     (
          ls1_match( _DirEnt , _Target , _Mode , _Uid , _Gid
, _Size , _Name ), !
```

```
     ;
          ls1_next_match( _Dir , _Target , _Mode , _Uid , _Gid
, _Size , _Name )
     ) .

ls1_match( _DirEnt , _Target , _Mode , _Uid , _Gid , _Size ,
_Name ) :-
     get_dirent( _DirEnt , _ , _ , _ , _ , _Name ),
     ls1_match_target( _Name , _Target , _Path ),
     recorded( ls1_print_stat , _Stat ),
     stat( _Path , _Stat ),
     get_stat( _Stat , _ , _ , _ModeBits , _ , _Uid , _Gid ,
_ , _Size , _ , _ , _ , _ ),
     ls1_mode_text( _ModeBits , _Mode ) .

ls1_parse_target( _Arg , _Path ,
Target(_WildCardPos,_Path,_Prefix,_Suffix) ) :-
     atomverify( _Arg , '*' , _WildCardPos ),
     ls1_parse_pre_suf_fix( _Arg , _WildCardPos , _PathPrefix
, _Suffix ),
     atomlength( _PathPrefix , _Length ),
     atomverify( _PathPrefix , '/' , _Length , _ , _DirPos ),
     ls1_parse_path_prefix( _PathPrefix , _DirPos , _Path ,
_Prefix ) .

ls1_parse_pre_suf_fix( _Arg , 0 , _Arg , '' ) :- ! .

ls1_parse_pre_suf_fix( _Arg , _WildCardPos , _Prefix , _Suffix
) :-
     atompart( _Arg , _Prefix , 1 , ?(_WildCardPos-1) ),
     atompart( _Arg , _Suffix , ?(_WildCardPos+1) , _ ) .

ls1_parse_path_prefix( _PathPrefix , 0 , '.' , _PathPrefix )
:- ! .

ls1_parse_path_prefix( _PathPrefix , _DirPos , _Path , _Prefix
) :-
     atompart( _PathPrefix , _Path , 1 , ?(_DirPos-1) ),
     atompart( _PathPrefix , _Prefix , ?(_DirPos+1) , _ ) .

ls1_match_target( _Name , Target(0,_Path,_Name,_) , _PathFile
) :- !,
     atomconcat( [_Path,'/',_Name] , _PathFile ) .

ls1_match_target( _Name , Target(_,_Path,_Prefix,_Suffix) ,
_PathFile ) :-
     atomconcat( _Prefix , _WildSuffix , _Name ),
     atomconcat( _Wild , _Suffix , _WildSuffix ),
     atomconcat( [_Path,'/',_Name] , _PathFile ) .

ls1_mode_text( _ModeBits , _Mode ) :-
     get_stat_mode( _ModeBits , _ModeList ),
     ls1_mode_check_special_bit( _ModeList , _Bs ),
     ls1_mode_check_bit( _ModeList , S_IRUSR , r , _Bur ),
     ls1_mode_check_bit( _ModeList , S_IWUSR , w , _Buw ),
     ls1_mode_check_bit( _ModeList , S_IXUSR , x , _Bux ),
     ls1_mode_check_bit( _ModeList , S_IRGRP , r , _Bgr ),
     ls1_mode_check_bit( _ModeList , S_IWGRP , w , _Bgw ),
     ls1_mode_check_bit( _ModeList , S_IXGRP , x , _Bgx ),
     ls1_mode_check_bit( _ModeList , S_IROTH , r , _Bor ),
     ls1_mode_check_bit( _ModeList , S_IWOTH , w , _Bow ),
     ls1_mode_check_bit( _ModeList , S_IXOTH , x , _Box ),
     atomconcat(
[_Bs,_Bur,_Buw,_Bux,_Bgr,_Bgw,_Bgx,_Bor,_Bow,_Box] ,
          _Mode ) .

ls1_mode_check_special_bit( _ModeList , _Bs ) :-
     ls1_mode_check_bit( _ModeList , S_IFDIR , d , _Bd ),
     (
          _Bd == d, !, _Bs = _Bd
     ;
          ls1_mode_check_bit( _ModeList , S_IFLNK , l , _Bs )
     ) .
```

```
ls1_mode_check_bit( _ModeList , _Field , _Symbol , _Symbol )
:-
     ls1_mode_bit_in_list( _ModeList , _Field ), ! .
ls1_mode_check_bit( _ModeList , _Field , _Symbol , '-' ) .
ls1_mode_bit_in_list( [_Field|_ModeList] , _Field ) :- ! .
ls1_mode_bit_in_list( [_|_ModeList] , _Field ) :-
     ls1_mode_bit_in_list( _ModeList , _Field ) .
```

6.5 Solution 2: retrieve all at once

This first hybrid solution uses a rather simplistic approach. It provides for **a fixed, limited number of solutions**. The necessary data structures are passed from Prolog to C. They are completely filled up with the solutions in C. After all solutions are collected, control returns to Prolog.

As the solutions are returned all at once, the problem of nesting is not relevant here. Therefore, only one iterator must be provided. This can be accomplished with a local data structure inside the external routine ls2().

The rest of the memory management is equally straightforward. The arrays that must hold the solutions are created in Prolog in the form of terms. All memory management for this is completely handled by ***ProLog by BIM***.

The big disadvantage is the limited and fixed number of solutions that can be retrieved.

Specification of the external predicate that implements this solution:

ls2/9

ls2(_Ret,_Arg,_SizeIn,_SizeOut,_Mode,_Uid,_Gid,_Size,_Name)

arg1 : r : integer : return code (0=OK)
arg2 : i : atom : argument to ls
arg3 : i : integer : maximal length of result lists
arg4 : o : integer : final length of result lists
arg5 : o : list(atom) : mode list
arg6 : o : list(integer) : uid list
arg7 : o : list(integer) : gid list
arg8 : o : list(integer) : size list
arg9 : o : list(atom) : file name list

Does ls of *arg2*. The result is returned in lists *arg5* to *arg9*, *arg3* entries at the most. The exact number of returned entries is *arg4*.

The necessary declaration to use it:

```
:- extern_predicate(
     ls2( integer:r , string:i , integer:i , integer:o ,
atom:array:m , integer:array:m ,
          integer:array:m , integer:array:m , atom:array:m )
) .
```

The corresponding C code:

```
int ls2( arg , sizein , sizeout , mode , uid , gid , size ,
name )
char *arg;
int sizein;
int *sizeout;
Atom *mode;
int *uid;
int *gid;
int *size;
Atom *name;
{
     LS_ITER lsi;
     static LS_ENTRY lse;
```

```
        int idx;
        if ( ! ls_initialize(arg,&lsi) ) return( 1 );
        idx = 0;
        while ( sizein-- && ls_next(&lsi,&lse) )
        {
                *mode++ =
BIM_Prolog_string_to_atom(FALSE,lse.ls_mode);
                *uid++ = lse.ls_uid;
                *gid++ = lse.ls_gid;
                *size++ = lse.ls_size;
                *name++ =
BIM_Prolog_string_to_atom(FALSE,lse.ls_name);
                idx++;
        }
        *sizeout = idx;
        ls_terminate( &lsi );
        return( 0 );
} /* ls2 */
```

Here follows the example application, using ls2/9 to retrieve all solutions at once. It has to decompose the resulting list in order to print each solution. The maximum number of solutions is set to 64.

```
ls2max( 64 ) .

ls2_print( _Arg ) :-
        ls2max( _Max ),
        functor( _Mode , Mode , _Max ),
        functor( _Uid , Uid , _Max ),
        functor( _Gid , Gid , _Max ),
        functor( _Size , Size , _Max ),
        functor( _Name , Name , _Max ),
        ls2( _Ret , _Arg , _Max , _Nr , _Mode , _Uid , _Gid ,
_Size , _Name ),
        ls_check_return( _Ret ),
        ls2_print( 0 , _Nr , _Mode , _Uid , _Gid , _Size , _Name
) .

ls2_print( _Max , _Max , _Mode , _Uid , _Gid , _Size , _Name
) :- ! .

ls2_print( _Nr , _Max , _Mode , _Uid , _Gid , _Size , _Name ) :-
        _Nr1 is _Nr + 1,
        arg( _Nr1 , _Mode , _M ),
        arg( _Nr1 , _Uid , _U ),
        arg( _Nr1 , _Gid , _G ),
        arg( _Nr1 , _Size , _S ),
        arg( _Nr1 , _Name , _N ),
        ls_print_line( _M , _U , _G , _S , _N ),
        ls2_print( _Nr1 , _Max , _Mode , _Uid , _Gid , _Size ,
_Name ) .
```

6.6 Solution 3: retrieve one by one

In this solution, **the control over the iteration is moved from C to Prolog.** This is realized by defining an external predicate for each of the three iterator routines: one for setting up the iterator, one for retrieving its next solution, and one for terminating the iterator. The Prolog program, that looks up a directory, is then responsible for controlling the iterator.

With this approach, each solution is immediately available for processing at the moment it is retrieved. As a result, nesting of iterators might be required. This causes some additional effort for memory management: the external iterator data structure cannot be a static variable. There must be a separate data structure for each iterator that is set up. Thanks to the logical structuring of the three predicates, the management of these data structures is straightforward. The data must be allocated when the iterator is set up.

De-allocation happens when the iterator is terminated. If the application does not terminate its iterators properly, a memory leak occurs. Moreover, as the directory is only closed when the iterator is terminated, not terminating the iterators would cause an overflow of open file descriptors.

The three external predicates are specified as:

ls3init/3

ls3init(_Ret,_Arg,_Iter)

arg1 : r : integer : return code (0=OK)
arg2 : i : atom : argument to ls
arg3 : o : pointer : iterator handle

Initiates an iterator with handle *arg3* for an ls of *arg2*.

ls3next/7

ls3next(_Ret,_Iter,_Mode,_Uid,_Gid,_Size,_Name)

arg1 : r : integer : return code (0=OK)
arg2 : i : pointer : iterator handle
arg3 : o : atom : mode
arg4 : o : integer : uid
arg5 : o : integer : gid
arg6 : o : integer : size
arg7 : o : atom : file name

Returns in *arg3* to *arg7* the entry that is retrieved by doing an iteration step over iterator handle *arg2*.

ls3stop/1

ls3stop(_Iter)

arg1 : i : pointer : iterator handle

Terminates iterator with handle *arg1*.

They are made available with the following declarations:

```
:- extern_predicate(
     ls3init( integer:r , string:i , pointer:o ) ) .
:- extern_predicate(
     ls3next(
integer:r,pointer:i,string:o,integer:o,integer:o,
integer:o,string:o ) ) .
:- extern_predicate(
     ls3stop( pointer:i ) ) .
```

The implementation of the external predicates in C is nothing more than the management of the iterator data structure, the mapping of the predicate to the corresponding basic iterator routine, and the passing of the resulting data.

```
int ls3init( arg , iter )
char *arg;
LS_ITER **iter;
{
     LS_ITER *lsi;
     if ( ! ( lsi = (LS_ITER *)malloc(sizeof(LS_ITER)) ) )
return( 2 );
     if ( ! ls_initialize(arg,lsi) ) return( 1 );
     *iter = lsi;
     return( 0 );
} /* ls3init */
int ls3next( iter , mode , uid , gid , size , name )
LS_ITER *iter;
char **mode;
int *uid;
int *gid;
```

```
int *size;
char **name;
{
     static LS_ENTRY lse;
     if ( ! ls_next(iter,&lse) ) return( 1 );
     *mode = lse.ls_mode;
     *uid = lse.ls_uid;
     *gid = lse.ls_gid;
     *size = lse.ls_size;
     *name = lse.ls_name;
     return( 0 );
} /* ls3next */
ls3stop( iter )
LS_ITER *iter;
{
     ls_terminate( iter );
     free( iter );
} /* ls3stop */
```

Below follows the application that prints the matching files. As can be seen, this results in a clear control structure in Prolog. The iteration in Prolog is programmed with a tail-recursive loop. A failure driven loop is not recommended here. This is because the external predicate that retrieves the next solution is a deterministic predicate. To get all solutions with a failure driven loop would require an extra predicate around ls3next/7 that provides the non-determinism.

Checking for the end of the iteration is implicit. The return argument (argument one) is instantiated to 0 (successful retrieval). If the external predicate returns another value than 0, the call of the predicate will fail on exit.

```
ls3_print( _Arg ) :-
     ls3init( _Ret , _Arg , _Iter ),
     ls_check_return( _Ret ),
     ls3_print_loop( _Iter ),
     ls3stop( _Iter ) .
ls3_print_loop( _Iter ) :-
     ls3next( 0 , _Iter , _Mode , _Uid , _Gid , _Size , _Name
), !,
     ls_print_line( _Mode , _Uid , _Gid , _Size , _Name ),
     ls3_print_loop( _Iter ) .
ls3_print_loop( _Iter ) .
```

6.7 Solution 4: backtracking external predicate

Whereas the previous solution provides an iteration mechanism based on deterministic external predicates, this solution has **an iterator in Prolog with a non-deterministic external predicate**. There are two external predicates. One is for setting up the iterator. The other is used for making the next iteration step. If it detects that there are no more solutions, it will terminate the loop (by cutting away the choice point), and destroy the iterator.

This solution involves the necessary memory management for enabling nesting of iterators. The external predicate that sets up the iterator also allocates space for the iterator data structure. When the iterator is destroyed, at the end of the loop, the memory it occupied is made available again.

The two external predicates are specified as follows:

ls4init/4

ls4init(_Ret,_Arg,_Iter,_Mark)

arg1 : r : integer : return code (0=OK)
arg2 : i : atom : argument to ls
arg3 : o : pointer : iterator handle
arg4 : o : atom : mark name

Initiates an iterator with handle *arg3* for an ls of *arg2*.

ls4next/7

ls4next(_Ret,_Iter,_Mode,_Uid,_Gid,_Size,_Name)

arg1 : r : integer : return code (0=OK)
arg2 : i : pointer : iterator handle
arg3 : o : atom : mode
arg4 : o : integer : uid
arg5 : o : integer : gid
arg6 : o : integer : size
arg7 : o : atom : file name

Returns in *arg3* to *arg7* the entry retrieved by doing an iteration step over iterator handle *arg2*.

Their declarations in Prolog:

```
:- extern_predicate(
     ls4init( integer:r , string:i , pointer:o , string:o ) )
.
:- extern_predicate(
     ls4next(
integer:r,pointer:i,string:o,integer:o,integer:o,integer:o,s
tring:o ) ) .
```

The two corresponding C routines are defined below.

External backtracking is realized through the use of *marked choice points*. These are choice points that are set in Prolog. They differ from ordinary choice points in two ways. First, they are labeled with a name tag (the mark). Secondly, they must be set explicitly, while normal Prolog choice points are implicit. The external non-deterministic predicate is responsible for keeping track of the already retrieved solutions. On each successive call, it must return the next solution. It is also responsible for terminating the loop. Therefore it can use an external ***ProLog by BIM*** routine to cut away the marked choice point.

In this program, the mark is named by the external routine.

```
#define LS_MARK "LS_MARK"
int ls4init( arg , iter , mark )
char *arg;
LS_ITER **iter;
char **mark;
{
     LS_ITER *lsi;
     if ( ! ( lsi = (LS_ITER *)malloc(sizeof(LS_ITER)) ) )
return( 2 );
     if ( ! ls_initialize(arg,lsi) ) return( 1 );
     *iter = lsi;
     *mark = LS_MARK;
     return( 0 );
} /* ls4init */
int ls4next( iter , mode , uid , gid , size , name )
LS_ITER *iter;
char **mode;
int *uid;
int *gid;
```

```
int *size;
char **name;
{
     static LS_ENTRY lse;
     if ( ! ls_next(iter,&lse) )
     {
          BIM_Prolog_rm_mrepeat( LS_MARK );
          free( iter );
          return( 1 );
     }
     *mode = lse.ls_mode;
     *uid = lse.ls_uid;
     *gid = lse.ls_gid;
     *size = lse.ls_size;
     *name = lse.ls_name;
     return( 0 );
} /* ls4next */
```

As shown in the example application below, this leads to a very clear, and typical Prolog control structure in the Prolog program that uses this non-deterministic external predicate. The iteration loop in Prolog is programmed with a failure driven loop, which is most natural in this case.

The initialization predicate returns the iterator handle and the mark that must be used for the choice point. Following this call, the marked choice point is set with the built-in **marked_repeat/2**. The next-solution external predicate is called and its result is printed. The **fail/0** activates backtracking over the next-solution predicate.

Checking for the end of the iteration is implicit. The return argument (argument one) is instantiated to 0 (successful retrieval). If the external predicate returns a value other than 0, the call of the predicate will fail on exit. At that moment, the external routine will have cut away the marked choice point. As a result, the backtracking will exit the first clause of ls4_print/1.

```
ls4_print( _Arg ) :-
     ls4init( _Ret , _Arg , _Iter , _Mark ),
     ls_check_return( _Ret ),
     mark_repeat( _Iter , _Mark ),
     ls4next( 0 , _Iter , _Mode , _Uid , _Gid , _Size , _Name
),
     ls_print_line( _Mode , _Uid , _Gid , _Size , _Name ),
     fail .
ls4_print( _Arg ) .
```

6.8 Solution 5: callback Prolog predicate

The usage of the *callback* mechanism leads to a very elegant solution. There is only **one external predicate. It has complete control over the iteration**. This is similar to solution 2. Unlike the solution, in which all answers are collected before being returned to Prolog, here each answer is processed immediately. It is passed to Prolog via the callback predicate, whose name is passed to the external predicate.

Because the whole iteration control remains in C, there is no need for special memory management to enable nesting of iterators. The iterator data structure is local to the external routine. A recursive call of this external predicate, from the callback predicate, is translated into a recursive call of the C routine, thus using the C recursion mechanism.

As the loop is controlled in the external predicate, no control structures are needed in the Prolog part.

The specification of the external predicate is:

ls5/3

ls5(_Ret,_Arg,_CallBack)
arg1 : r : integer : return code (0=OK)
arg2 : i : atom : argument to ls
arg3 : i : atom : name of callback predicate

Does ls of *arg2*. Each entry is passed to the callback predicate *arg3/5*.

This is made available with the declaration :

```
:- extern_predicate(
     ls5( integer:r , string:i , atom:i ) ) .
```

The C definition of the external routine includes a call from C to Prolog. The callback predicate is called each time a solution is found.

```
int ls5( arg , atom )
char *arg;
BP_Atom atom;
{
     LS_ITER lsi;
     LS_ENTRY lse;
     BP_Functor pred;
     if ( ! ls_initialize(arg,&lsi) ) return( 1 );
     pred = BIM_Prolog_get_predicate( atom , 5 );
     while ( ls_next(&lsi,&lse) )
     {
          BIM_Prolog_call_predicate( pred ,

BP_M_IN|BP_S_SIMPLE|BP_T_STRING , lse.ls_mode ,

BP_M_IN|BP_S_SIMPLE|BP_T_INTEGER , lse.ls_uid ,

BP_M_IN|BP_S_SIMPLE|BP_T_INTEGER , lse.ls_gid ,

BP_M_IN|BP_S_SIMPLE|BP_T_INTEGER , lse.ls_size ,

BP_M_IN|BP_S_SIMPLE|BP_T_STRING , lse.ls_name );
     }
     ls_terminate( &lsi );
     return( 0 );
} /* ls5 */
```

The Prolog application for printing the matching files is very simple. It consists of two small predicates. The basic predicate calls the external predicate and passes it the name of the callback predicate. This callback predicate does nothing else than printing out the solution. In fact, the ls_print_line/5 predicate could have been passed directly as callback to the external predicate, instead of having an extra layer. This extra predicate is only added here for clarity.

```
ls5_print( _Arg ) :-
     ls5( _Ret , _Arg , ls5_print );
     ls_check_return( _Ret ) .
ls5_print( _Mode , _Uid , _Gid , _Size , _Name ) :-
     ls_print_line( _Mode , _Uid , _Gid , _Size , _Name ) .
```

6.9 Solution 6: external term construction

For applications where all answers are needed together, **the best solution is to collect the answers externally before returning to Prolog**. Solution 2 works this way. However, it has the serious restriction that the number of answers is limited. Solution 6 also collects the answers in C, but there is no restriction on the number of answers. This is realized by directly building a Prolog term in the external routine.

The question of nesting is not applicable to this solution, so there is no memory management necessary for this. As for the list of answers, this is completely stored on the heap of ***ProLog by BIM***. The only required precaution is for garbage collection of the heap. Garbage collection in the middle of a term construction must be avoided. ***ProLog by BIM*** provides two facilities for preventing the destruction of an externally created term by garbage collection. External terms can be protected. This means they will not be destroyed during garbage collection. In addition, the protected term handle is guaranteed to always point to the term. Non-protected term handles may become dangling pointers after heap garbage collection. The other provision is an external routine to ensure that there is enough space left on the heap before starting the creation of a term. As this may invoke the heap garbage collector, any term that already exists must be protected.

In this program, the heap space is checked each time a new answer is added to the term. More efficient approaches test for heap space once for the whole term to be created. But here, the size of the whole term is not known in advance.

This is the specification of the external predicate:

ls6/3

ls6(_Ret,_Arg,_List)
arg1 : r : integer : return code (0=OK)
arg2 : i : atom : argument to ls
arg3 : o : list(LS/5) : ls result list

Does ls of *arg2*. *Arg3* is the resulting list of entries.

The declaration to make this predicate available:

```
:- extern_predicate(
     ls6( integer:r , string:i , bpterm ) ) .
```

The C implementation of the external routine is given below. The created Prolog term is a list of terms with functor LS/5 and with the five fields of the answer as arguments. It is built up by further instantiating the term argument that is passed from Prolog.

```
int ls6( arg , term )
char *arg;
BP_Term term;
{
     LS_ITER lsi;
     LS_ENTRY lse;
     BP_Atom nil;
     BP_Functor funcls;
     BP_Term term1, term2;
     nil = BIM_Prolog_string_to_atom(TRUE, "nil" );
     funcls = BIM_Prolog_get_predicate(
          BIM_Prolog_string_to_atom(FALSE,"LS") , 5 );
     if ( ! ls_initialize(arg,&lsi) ) return( 1 );
     while ( ls_next(&lsi,&lse) )
     {
          term = BIM_Prolog_protect_term( term );
          if ( ! BIM_Prolog_term_space( 8 ) ) return( 2 );
          term = BIM_Prolog_unprotect_term( term );
```

```
            BIM_Prolog_unify_term_value( term , BP_T_LIST );
            BIM_Prolog_get_term_arg( term , 1 , &term1 );
            BIM_Prolog_unify_term_value( term1 ,
BP_T_STRUCTURE , funcls );
            BIM_Prolog_get_term_arg( term , 2 , &term );
            BIM_Prolog_get_term_arg( term1 , 1 , &term2 );
            BIM_Prolog_unify_term_value( term2 , BP_T_ATOM ,

BIM_Prolog_string_to_atom(FALSE,lse.ls_mode) );
            BIM_Prolog_get_term_arg( term1 , 2 , &term2 );
            BIM_Prolog_unify_term_value( term2 , BP_T_INTEGER
, lse.ls_uid );
            BIM_Prolog_get_term_arg( term1 , 3 , &term2 );
            BIM_Prolog_unify_term_value( term2 , BP_T_INTEGER
, lse.ls_gid );
            BIM_Prolog_get_term_arg( term1 , 4 , &term2 );
            BIM_Prolog_unify_term_value( term2 , BP_T_INTEGER
, lse.ls_size );
            BIM_Prolog_get_term_arg( term1 , 5 , &term2 );
            BIM_Prolog_unify_term_value( term2 , BP_T_ATOM ,

BIM_Prolog_string_to_atom(FALSE,lse.ls_name) );
      }
      BIM_Prolog_unify_term_value( term , BP_T_ATOM , nil );
      ls_terminate( &lsi );
      return( 0 );
} /* ls6 */
```

The application that uses this external predicate just calls it and then processes the answer list.

```
ls6_print( _Arg ) :-
      ls6( _Ret , _Arg , _List ),
      ls_check_return( _Ret ),
      ls6_print_list( _List ) .
ls6_print_list( [] ) .
ls6_print_list( [LS(_Mode,_Uid,_Gid,_Size,_Name)|_List] ) :-
      ls_print_line( _Mode , _Uid , _Gid , _Size , _Name ),
      ls6_print_list( _List ) .
```

User's Guide

Chapter 7
Debugging

This chapter starts with the presentation of the five port box model that is used in ***ProLog by BIM***.

The remainder of the chapter explains how to use the ***ProLog by BIM*** window environment for debugging Prolog programs. This is done on different levels. First, the usage of the *monitor* is shown for setting up a debug session. Then, two debug sessions are demonstrated. One uses the *source-oriented debugger* of ***ProLog by BIM***. The other is a *post-execution analysis* session. Finally, some hints are given for customizing the ***ProLog by BIM*** debuggers.

Neither the *monitor* nor the *debugger window* are described in this chapter. It is only explained how they can be used for debugging. A full and detailed description of these environment components is given in the ***ProLog by BIM*** *Reference Manual*.

7.1 The five port box model

The presentation of the five port box model used in ***ProLog by BIM*** is included for users who want to debug in the more traditional *execution model oriented* way. ***ProLog by BIM*** provides extended and flexible facilities for this debugging method. However, users are encouraged to debug with the more advanced source-oriented debugger.

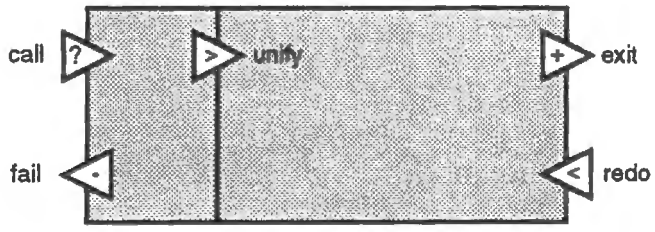
The rest of this chapter is independent of this section. The reader who prefers to use the advanced source-oriented debugger, or post-execution analyzer, can easily skip it.

In a box model, a *box* is associated to each active predicate invocation. If there are *n* active calls of a given predicate, there will be *n* boxes, all for the same predicate. A box starts its life as soon as a predicate is being called. Its existence terminates when all clauses of the predicate are completely executed. This means there are no active subgoals any more and no alternatives are left.

All boxes are coupled to each other via their ports. Boxes of successive subgoals are laid out sequentially. The boxes of the subgoals of a predicate are located inside the predicate's box.

A box has input and output ports. The figure below gives a graphical representation of a box in ***ProLog by BIM***.

The five port box of ***ProLog by BIM***.



Each port is labeled with its logical name and with the symbol that is used in ***ProLog by BIM*** to indicate the port type in a trace. The ports correspond to the following points in the execution of the predicate.

call (?)
: This port is entered when the predicate is called, before trying to unify the arguments that are used in calling it, with the arguments in the head of a definition for the predicate. It is only crossed once, when the box is created.

unify (>)
: This internal port is crossed after a successful unification of the arguments in the calling subgoal with the arguments in the head of a definition for the predicate. It can be crossed as many times as there are clauses defining the predicate. One could also think of the box having multiple *unify* ports, one for each clause.

exit (+)
The box is left via this port after the successful solving of all subgoals in one of the predicate's clauses. As for the *unify* port, it can be crossed as many times as there are clauses defining the predicate. After the box is left for the last possible time via this port, the box ceases to exist.

fail (-)
The box is left via this port after a failure. This can be a failure in the unification of the clause head. In that case, the corresponding *unify* port will not have been crossed. Another possibility is the failing of a subgoal in the predicate's clauses. This port can be crossed as many times as the predicate can fail. This may be more than the number of clauses defining the predicate. After the box is left for the last possible time via this port, the box ceases to exist.

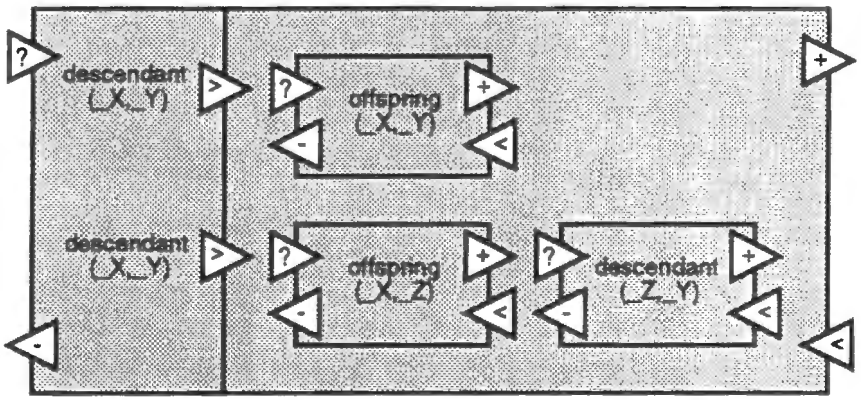
redo (<)
This entrance port is crossed during backtracking from outside the predicate. It leads execution to the most recent pending alternative inside the box. It is entered after a *fail* port has been crossed. This can be either the *fail* port of the goal following the goal represented by this box, or it can be the box's own *fail* port.

A small example is used to give a graphical representation of the lay-out of different boxes.

```
descendant( _X , _Y ) :-
     offspring( _X , _Y ) .
descendant( _X , _Y ) :-
     offspring( _X , _Z ),
     descendant( _Z , _Y ) .
```

The box for the top most invocation of descendant/2, together with the boxes for its subgoals, are shown in the next figure.

Lay-out of boxes for nested and successive subgoals.



The boxes for the subgoals in both clauses are positioned inside the box for the top invocation of descendant/2. The boxes for the two successive subgoals in the second clause are positioned one after the other. This positioning also suggests how the ports are coupled to each other. It does not show all couplings. This would be too difficult in a two-dimensional picture.

7.2 Setting up a debug session with the monitor

This section illustrates how the *debugger window* is activated. It also explains how to set up a debug session using the *monitor*.

Activating the debugger window

From the Unix command level, ***ProLog by BIM*** can be started up with the *monitor* activated:

```
BIMprolog -Pem+
```

From a running ***ProLog by BIM***, it is activated with the following call of **please/2**:

```
?- please( em , on ) .
```

The abbreviation *em* stands for *e*nvironment *m*onitor.

Similarly, the *debugger window* can be activated, either from the Unix command level with the command:

```
BIMprolog -Ped+
```

or from a ***ProLog by BIM*** session with:

```
?- please( ed , on ) .
```

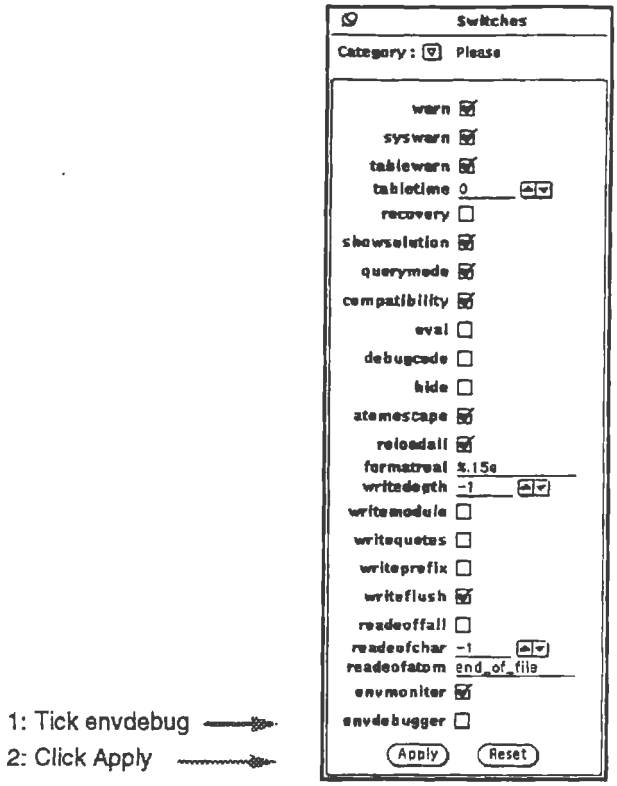
Here, *ed* stands for *e*nvironment *d*ebugger.

As both windows are activated (and deactivated) with the built-in **please/2**, they can also be controlled from the *switches* window of the *monitor*. Therefore, the *monitor* must be active, of course. Figure 3 shows how to do this:

1: Tick *envdebug*
The *em* switch (full name *envdebug*) of **please/2** is set on.

2: Click *Apply*
The switches that are set on the window only become active when they are applied with this button. The previous settings of the switches are restored with the *Reset* button.

The *monitor* with its *switches* window.



Setting up the debug session

The program to be debugged must be consulted. Using the *monitor*'s *files* window, the file can easily be searched in the directory hierarchy. The above figure illustrates this.

1: Fill in directory
The directory path can be adapted in the *Dir* field. Alternatively, if one does not know the exact path, the directories can be scanned by clicking on the desired directory in the window. A directory is displayed in boldface. To go up one level in the hierarchy, the special link. must be clicked. The *Dir* field is always updated to reflect the path of the displayed directory.

2: Click *Refresh*
After filling in the directory path, the display must be refreshed to give the contents of the indicated directory. This refresh is automatic if the directories are scanned by clicking in the display.

Note:

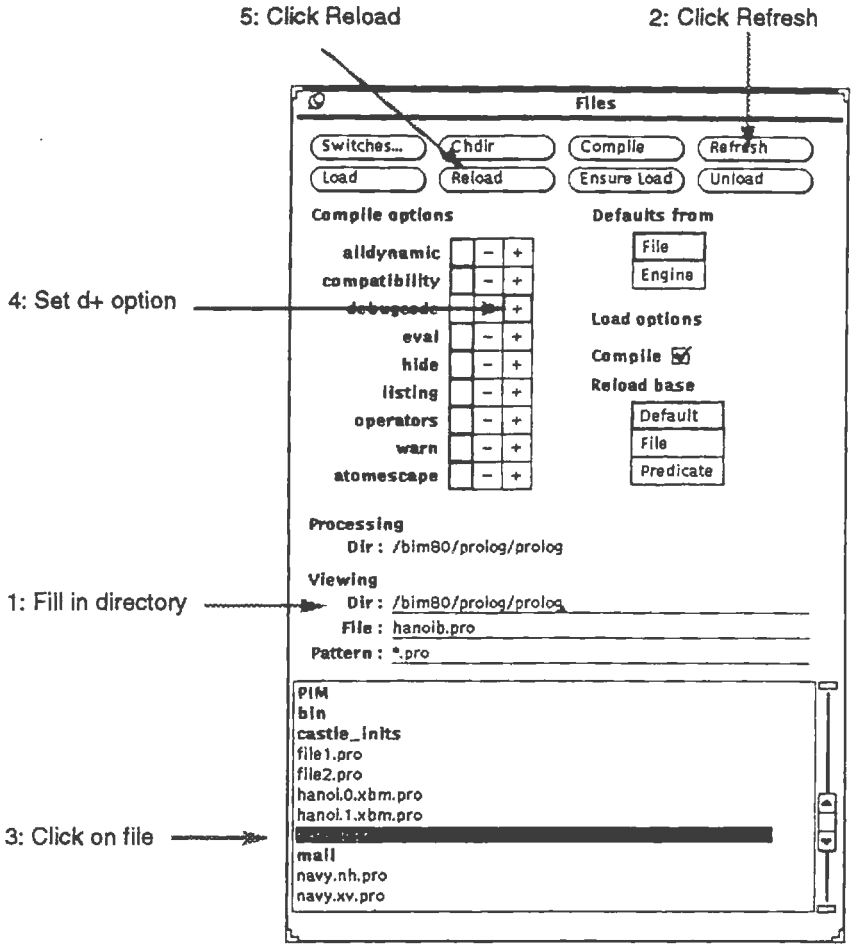
The pattern **.pro* specifies that only files ending on *.pro* are to be displayed.

3: Click on file
Clicking on the name of a file on the display, selects it. The name is copied in the *File* field.

4: Set *d+*
The compiler *debug* option is set, to compile the file for debugging.

5: Click *Reconsult*
By clicking on this button, the indicated file is reconsulted, with the specified compiler options.

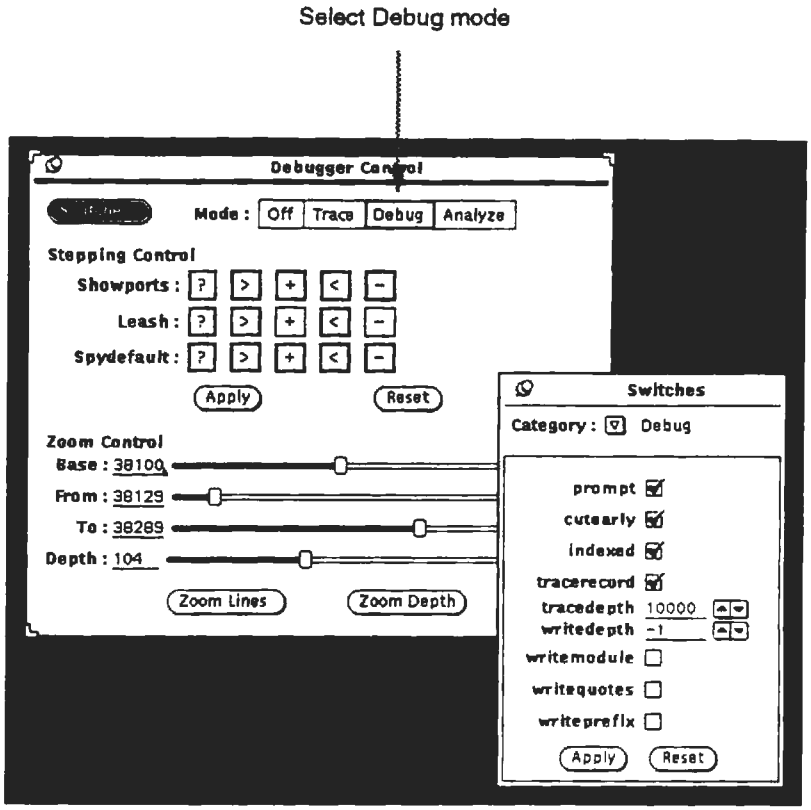
Consulting a file with the *Files* window of the *Monitor*.



This procedure can be repeated if more than one file must be consulted.

When all necessary files are consulted, the engine must be switched to execution in debug mode (It is not required, but handier that all files are consulted before switching into debug mode). On the *Debugger Control* window (), the execution mode for the engine is indicated with a choice. To switch to debug mode, the desired debugger mode can be chosen. The figure below shows how to switch the execution mode to *Debug* (which is used for instance, for source-oriented debugging). Once an execution mode is set, it cannot be changed any more until the execution is completed. This does not mean that the whole program must be terminated. A debug session can be terminated at any moment with the *quit* command.

Switching to *Debug* mode on the *Debugger Control* window.



7.3 Source-oriented debugging

In this section, some basic principles of source-oriented debugging are mentioned. The example demonstration is introduced next. This is followed by the guided debug session.

Principles

The underlying idea of source-oriented debugging is, that it situates the Prolog programmer in a familiar context to debug his program. It shows the programmer how the program executes by referring directly to his source program. With this approach, the programmer needs no knowledge about the execution mechanism of the Prolog engine. The traditional box-model debugging does require that the programmer understands the execution mechanism.

With the source-oriented debugger, the programmer can indicate the points of interest in the source program. The query is entered and ***ProLog by BIM*** starts executing it. As soon as one of the mentioned points of interest is encountered, the execution is suspended. The debugger indicates, in the source program, where the execution has arrived. It tells which line will be executed next. At that moment, the programmer can ask all desired information. Very interesting information is the values of variables in the clause that is being executed. One can also walk through the chain of ancestor predicates. This is the list of active predicate calls, starting with the query and going down one level for each selected subgoal. Of course, it is also possible to ask for the value of a variable anywhere in that ancestor chain. Once the programmer has gathered all the necessary information, the program can be resumed. There are different modes for resuming a suspended program. Roughly, the mode determines how far the execution must continue before being suspended again.

Source-oriented debugging works with an ordinary terminal and keyboard input. But the most promising environment is a workstation, with multiple windows for each type of output and with mouse input. In such an environment, the debugging process can be sped up. Most input can be given with only a few mouse clicks. No long commands or arguments have to be typed in. The commands are chosen from a button bar or from a pop-up menu. The arguments can be indicated with the mouse anywhere on the screen. A separate window for the source program makes a clear distinction between this source code and the command log.

Example problem

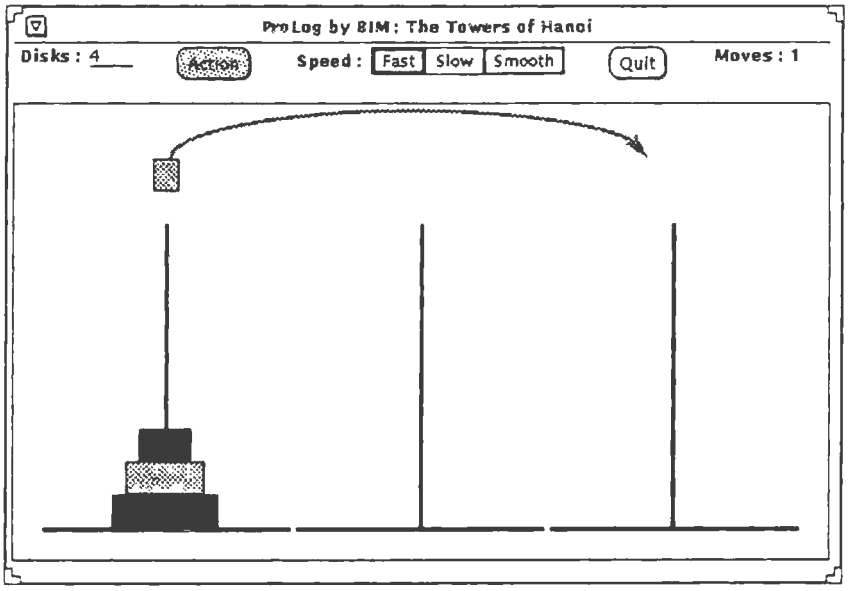
The usage of the source-oriented debugger, in a window environment, is illustrated with the well-known *Towers of Hanoi* example. The source for this graphical demonstration program is included on the distribution tape, and can be found in the directory $BIM_PROLOG_DIR/demos/windowXV (or /SunView). This demonstration shows in particular, the usefulness of this debugging method for programs that are based on XView (or SunView).

Note:

To illustrate the debugger, we introduced a bug in the program.

The picture of the program in the figure below illustrates the bug. In *Smooth* transition mode, a disk must move with small steps. First upwards until it is above the pin, then horizontally until it reaches the destination pin. From there, it must descend the pin until it stacks on top of the other disks on that pin. From a run of the program, it is noticed that the disks do not move smoothly up and down, but not horizontally. Instead of moving in small steps, the disk immediately jumps from its source pin to its destination pin.

The *Towers of Hanoi* demo with a bug.



Continued set-up of the debug session

In section *7.2 Setting up a debug session with the monitor*, it is shown how the Debugger Window is activated, and how the demonstration program can be consulted with the Monitor. Here, the demonstration is continued, setting up the debug session.

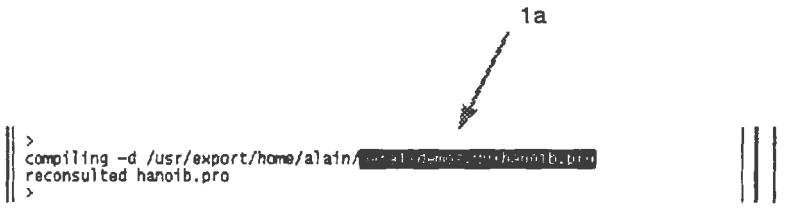
Reconsulting the source program, generates a message in the *Unix Command Window* (where ***ProLog by BIM*** is started up), telling which file is reconsulted. This information is useful to load the file in the *Debugger Source Window*. This does not happen automatically. Source files are displayed only when a source line must be indicated. To set up the debug session, we need the source file. Therefore we explicitly load it. This is shown in the next two figures.

1: Loading the source file in the *Debugger Source Window*.

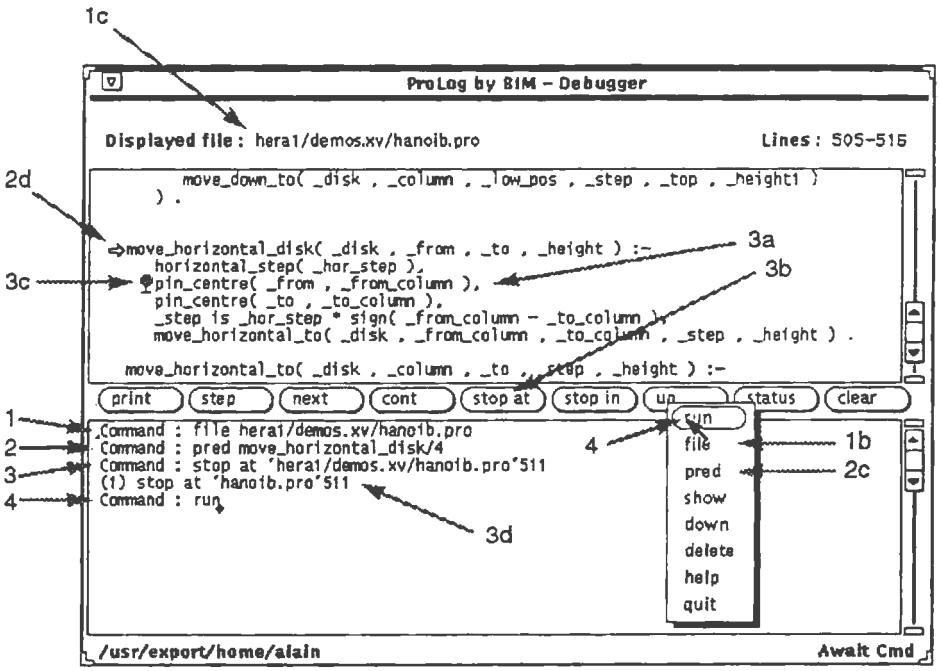
First, the name of the desired source file is selected on the screen. Here, it is selected in the *Unix Command Window* where the reconsult message is displayed (1a).

This selected text acts as argument to the *file* command, that is chosen from the pop-up menu in the *Debugger Command Bar* (1b).The result is that the specified file is displayed in the *Debugger Source Window*. The file name of that displayed file, is indicated on the *Debugger Information Panel* (1c).

Setting up the source-oriented debug session - *Unix Command Window*.

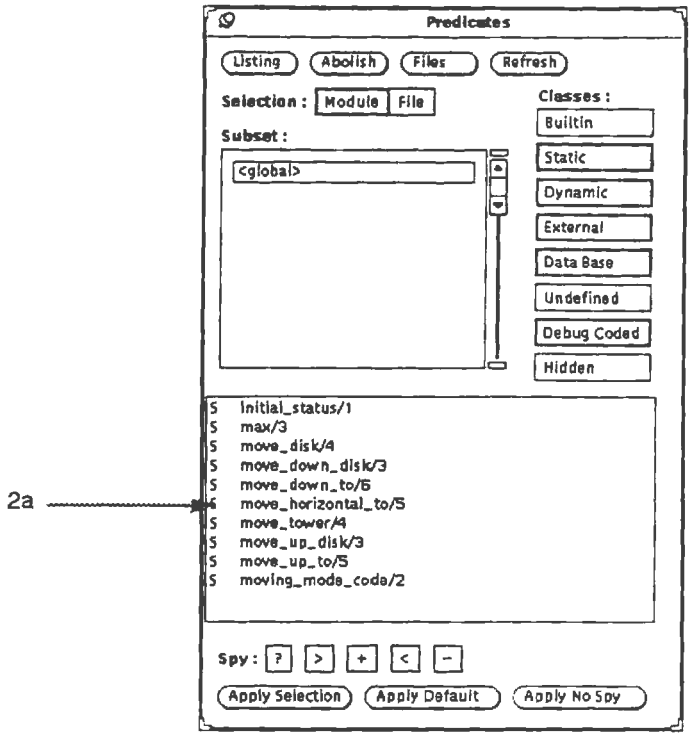


Setting up the source-oriented debug session - *Debugger Window*.



Once the program source file is loaded, it can be used to indicate the points of interest by setting *break points*. From a test run, we discovered that something goes wrong with moving a disk in horizontal direction in smooth transition mode. Our point of interest is the predicate that is responsible for this movement. The exact name of the predicate can be looked up in the *Monitor Predicates Window* (see next figure). The above figure shows how this is combined to display the definition of the predicate in the *Debugger Source Window*.

Setting up the source-oriented debug session - *Monitor Predicates Window.*



2a →

2: Looking up and displaying the suspected predicate.

The program resides in the global module. The *Predicates Window* displays the current predicates in that module. If the suspected predicate were in another module, this module's predicates should first be displayed. This could be accomplished by selecting the module from the list, and clicking the *Refresh* button.

Scrolling through the list of predicates (which is ordered alphanumerically), reveals the name of the predicate that we are searching for: move_horizontal_disk/4. This predicate is selected by clicking on its name in the display (2a).

In that field, the whole text, predicate name and arity, is selected (2b).

This selection is the argument for the call of the *pred* command. That command is executed by choosing it on the pop-up menu in the *Debugger Command Bar* (2c).

The result of this command, is that the *Debugger Source Window* is scrolled to display the definition of the predicate. This is indicated with a white arrow to the left of the first clause for that predicate (2d).

Note:

In this particular case, the first step of loading the source file could have been omitted. When using the pred command to display a predicate definition, the source file containing that definition is loaded automatically if necessary. The explicit demand of loading a source file is demonstrated for use in other cases. It might happen that the programmer wants to search for the points of interest by scrolling through the source file directly. For that, it must be loaded explicitly.

A look at the definition of move_horizontal_disk/4, ensures that this is the suspected predicate. Now, a break point can be set in two ways: either *in* the predicate, or *at* a certain line of the predicate. A break point *in* a predicate, means that the execution must be suspended whenever it enters one of its clauses (In box-model terminology: when reaching a *unify* port of one of the predicate's clauses). To set a break point *in* our predicate, we click on the *stop in* button. This button takes the current selection as argument. After the previous step of displaying the predicate definition, the selection is still the predicate name and arity.

As we know that the first subgoal (horizontal_step/1) calculates the step for a single small horizontal move of the disk, we set our point of interest after it. We will then check the value of this step. Such point of interest, at a certain point in the clause, is indicated with a break point at the line containing that point. This step is shown on the previous debugger window.

3: Set a break point at the line of interest.
The line is indicated by clicking on it in the *Debugger Source Window* (3a). The debugger translates this action internally to a file name (of the displayed file) and line number (of the indicated line in that file). This is then taken as argument for the *stop at* command. This command is executed by clicking on the *stop at* button in the *Debugger Command Bar* (3b).
As a matter of feedback, the debugger reports a successful setting of a break point in two ways. In the *Debugger Source Window*, a stop sign is set to the left of the line where the break point is set (3c). In the *Debugger Source Window* the setting of the break point is traced with a message (3d). This consists of a line containing the break point number (1 in this case), the type of break point (*stop at*) and the file name and line number in which it is set.

Note:

The line number that is reported when a break point is set, may differ from the number given in the argument of the *stop at* command. This is the case when there is no code at the indicated line (e.g. a blank line) or when a subgoal extends over several lines, and the indicated line is not its first line. In both cases, the debugger searches for the logically closest line that contains code.

When all points of interest are indicated, the program can be started. In order to switch back control from the debugger to the execution engine, the *run* command is issued:

4: Switching control from debugger to execution engine.
The *run* command is chosen on the pop-up menu in the *Debugger Command Bar* (4).

The program has to be started up in the *Unix Command Window*, where ***ProLog by BIM*** is started. The next query starts the *Towers of Hanoi* demo:

```
?- go .
```

Debugging

As the program is started, the window with the picture of the towers pops up. The number of disks is set to 4 in the *Disks* field. Transition mode is chosen as *Smooth* on the *Speed* choice field. Then the game is started by clicking on the *Action* button. The top most disk on the left pin is lifted up (smoothly). When it reaches a height where it can move horizontally, the program is suspended. This happens at the moment the break point in move_horizontal_disk/4 is encountered. Here is how the *Debugger Window* looks like at this point (see next figure):

1: Program suspension at break point.
The program is suspended just before executing the subgoal at the line that contains the break point (1a). If necessary, the debugger loads the source file containing that line and scrolls its *Source Window* to display it. The fact that the execution is suspended at that line, is indicated with a black arrow to the left of the stop sign. In the snapshot, this arrow has already moved (4), as a number of program continuation commands were issued before the snapshot was taken.
A second indication of the current suspension point, is given in the *Debugger Information Panel* (1b). The file name and line number of the suspension point is mentioned. The name and arity of the most recent active predicate at that point, is mentioned as well. This information can be useful, as the black arrow may be invisible. Due to the execution of information gathering commands, other pieces of the source may be displayed. This causes the black arrow to move out of sight. To bring it back, the *show* command can be used.
In the right lower corner of the *Debugger Window* the status mentions *Await Cmd* (1c), indicating that debugger commands can be entered now.

2: Asking the value of a variable.

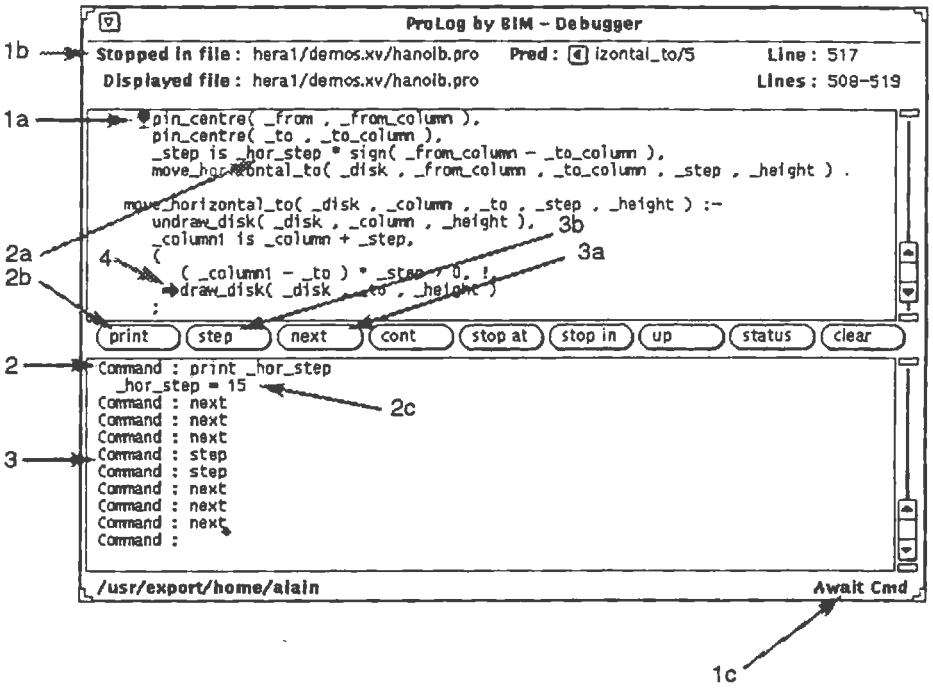
Our point of interest was just after the calculation of the horizontal step size. To get that value, we use the *print* command. The name of the variable is indicated on the screen (2a) by selecting it (only one or a few letters of it are enough). This selection is used as argument of the *print* command. Executing the command is done by clicking on the *print* button (2b).

The debugger answers this request by printing out the value of the variable in the *Debugger Command Window* (2c).

Note:

Only active variables in the focussed environment can be printed out. Active variables are those that are either undefined or that still have a meaningful value at the suspension point in the clause. The focussed environment is in the beginning the most recent active predicate. It can be changed to one of the ancestor predicates (less recent active predicates). The commands *up* and *down* can be used to move the environment focus through the active predicate chain. Trying to print out an inactive or unreachable variable results in a message telling it is undefined.

Debug session - first snapshot.



After printing out the horizontal step, which seems to have a reasonable value, the execution is continued ():

3: Continuing program execution

The most drastic way to resume execution is the *cont* command. This continues the program execution until another break point is reached. In our case this would be when another disk has been moved up.

As this is too far, we continue in smaller steps. We could also use the *cont* command after setting additional break points, for instance in the predicate move_horizontal_to/5, which is the most suspect subgoal.

With three successive *next* commands, the subgoals for pin_centre/2 and for the calculation are skipped. These commands are issued by clicking the *next* button (3a). Skipping this subgoal execution does not mean they are not executed. It only means, we are not interested in their execution. The whole subgoal is executed and the execution is suspended when the line following it, is reached.

This brings us to the call of move_horizontal_to/5. As this subgoal does interest us, we use a still smaller stepwise continuation. With *step* (3b) only a

single line of the source is executed. The subgoal on that line is entered, but not executed. Our first *step* brings us to the head (*unify* port), of the predicate move_horizontal_to/5. The second *step* enters this clause. (A *next* at this point, would have executed the predicate completely.)
The first subgoal undraws the disk. The next calculates the new position for the disk. Further, the predicate is split in two alternatives: one to draw the disk at its final position and terminate. The second alternative draws the disk at the next intermediate position and continues the smooth move of the disk by a recursive call. We continued execution with three *next* commands, until after the test that determines which alternative will be taken.
To our surprise, the first one is taken. The disk is immediately transferred to its final destination instead of moving it a little bit to the right.

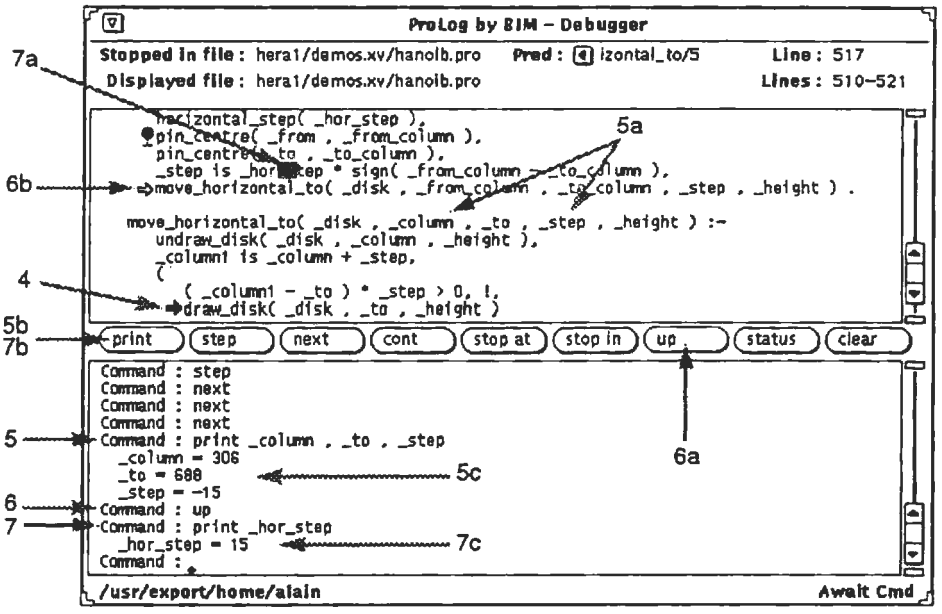
4: Point of program suspension, after a number of continuation commands.
This is the place where the first snapshot was taken. The black arrow (4) indicates the current suspension point.

Now, we have to find out why the test for taking the first alternative has succeeded. How this is done, can be followed on :

5: Printing multiple variables in one motion.
To understand what is happening, we ask the values of different variables. As the variables in which we are interested, happen to appear as three successive arguments, we can select them all together (5a). Clicking on the *print* button (5b), reports the values of all three variables (5c).

The value for the current column position and for the final destination position seem reasonable. The step however, is negative. Moving from left to right involves a positive step. The step being calculated in the ancestor predicate, we decide to take a look there.

Debug session - second snapshot.



Looking in an ancestor environment is explained below, and shown in :

6: Moving the focus to the ancestor environment.
The *up* command brings the focus environment one level higher in the active predicate chain. It is executed with a click on the *up* button (6a).
Feedback is given in the form of a white arrow at the line from where the predicate is called in the ancestor predicate (6b).

7: Printing a variable in the ancestor environment.
The step is calculated by multiplication of the fixed horizontal step with a sign factor. Checking the horizontal step is performed by selecting the variable name on the screen (7a), and clicking the *print* button (7b). The result (7c) indicates that this step is positive. This means, the calculated sign is -1.

The wrong sign is calculated as source position minus destination position. This must be switched. Moving from left to right (increasing coordinates) requires a positive step.

The *quit* command terminates the debug session.

7.4 Post-execution analysis

With the post-execution analyzer, a program is debugged in a breadth-first manner. First the program is executed. The debugger is told to keep a trace of the program execution. After termination, that trace is analyzed by the debugger, under guidance of the programmer. The trace is analyzed breadth-first. All subgoals of the query are displayed. By selecting one of these, the focus is moved one level down in the execution tree. With this method, it is possible to immediately go down in any of the subgoals. The bug location can be found much faster than when each subgoal must be investigated in depth before the next one can be investigated.

The usage of this debug tool is illustrated with the following small insertion sort program.

```
isort( [] , [] ) .
isort( [_X|_List] , _Sort ) :-
     isort( _List , _SortTail ),
     insert( _X , _SortTail , _Sort ) .
insert( _X , [_Y|_Sort1] , [_Y|_Sort2] ) :-
     X @> _Y, !,
     insert( _X , _Sort1 , _Sort2 ) .
insert( _X , _Sort , [_X|_Sort] ) .
```

The insertion sort of the list [_X|_List] is _Sort, if the insertion sort of the tail _List is _SortTail, and the insertion of the head _X in that sorted tail _SortTail, is _Sort.

For insertion there are two clauses. The first is when the element is bigger than the first element of the list. In that case, the element must be inserted in the tail of the list. The second is for the other case. There, the element can be put immediately at the head of the list, as it is smaller than all elements in that (sorted) list.

This is a sample query with its wrong result (the list is not sorted):

```
?- isort( [b,a,c] , _X ).
_X = [b,a,c]
Yes
```

This query will now be used to debug the program. The explanation below contains all debugger output that appears in the *Debugger Command Window*. At the end of the session, a snapshot of the *Debugger Window* has been taken, which is shown in the next figure.

The debug session is set up as in the previous example. The *Debugger Window* is activated, the source file is consulted and the execution mode is switched to *Analyze*. Then the query is re-entered. The debugger records a trace of the execution. After this is terminated, the following output appears in the *Debugger Command Window*:

```
0:    ?-
1:          isort( [b,a,c],_9 ) <--- failed
```

This means the subgoal of the query has failed. This is normal as the first clause of isort/2 only sorts an empty list. Now we ask the analyzer to *advance* to the next solution of the query. The *advance* command can be found on the pop-up menu in the *Debugger Command Bar*.

Command : advance

```
0:    ?-
1:          isort( [b,a,c],_9 ).
```

Now we got a solution. The analyzer has different modes for displaying. By default, the goal and subgoals are shown as they were instantiated at the moment of *call*. (In box-model terminology: at the *call* or *redo* port). If the result of a subgoal should be made visible, the display mode must be switched to *exit*. In this mode, the subgoals are shown as they were instantiated at *exit* (*exit* or *fail* port).

Command : exit

```
0:    ?-
1:          isort( [b,a,c],[b,a,c] ).
```

We see that the result is wrong. Therefore we will investigate the first (and only) subgoal. This is done with the command *invest* that takes the number of the subgoal as argument. Selecting the subgoal number and clicking on the *invest* button, does the job. When the analyzer is used on an ordinary terminal, the predefined aliases come in handy (see section *7.5 Customizing the debugger* for more information on aliases). Instead of having to type in *invest 1*, one can simply type in *1*, and so on for the other subgoals (up to 9).

Command : invest 1

```
0:    isort( [b,a,c],[b,a,c] ) :-
1:          isort( [a,c],[a,c] ),
2:          insert( b,[a,c],[b,a,c] ).
```

At this moment, a black arrow indicates the definition of the predicate that is displayed in the *Debugger Source Window*. From the output, it is clear that the problem is located in the second subgoal. The element b is not inserted correctly in the well-sorted list [a,c]. Investigating subgoal 2 reveals:

Command : invest 2

```
0:    insert( b,[a,c],[b,a,c] ) .
```

In the *Source Window*, we can see that the second clause of insert/2 was chosen here. We had expected the first one, as b is larger than a (in Prolog's standard order of terms). The analyzer not showing this first clause, means it has failed. To speed up the localization of a bug, the analyzer always skips the failing branches in the execution tree and only displays the first succeeded branch. However, as it is important in our case to know what happened in that failing first clause of insert/2, we tell the analyzer to display failing branches too, with the command *Fail*. We go two levels up again in order to be able to re-investigate the failing branch that was skipped.

Command : Fail

Failures will be investigated.

Command : back

```
0:    isort( [b,a,c],[b,a,c] ) :-
1:          isort( [a,c],[a,c] ),
2:          insert( b,[a,c],[b,a,c] ).
```

Command : back

```
0:    ?-
1:          isort( [b,a,c],[b,a,c] ).
```

Command : invest 1

```
0:    isort( [b,a,c],[b,a,c] ) :-
1:          isort( [a,c],_30 ) <--- failed
```

At this point, the analyzer shows the failed branch of the first subgoal, that was not shown the previous time we came here. We are not interested in this failure, as we know that it is caused by the first clause of isort/2 for an empty list. We *advance* to the next failure.

Command : advance

```
0:    isort( [b,a,c],[b,a,c] ) :-
1:          isort( [a,c],[a,c] ),
2:          insert( b,[a,c],_9 ) <--- failed
```

From here we can investigate the failing first clause of insert/2. This can be done with the command *invest 2*. There is also a shortcut: the command *fail* investigates the failing subgoal. This is unambiguous as there is always at most one failing subgoal.

Command : fail

```
0:    insert( b,[a,c],_9 ) :-
1:          X @> a <--- failed
```

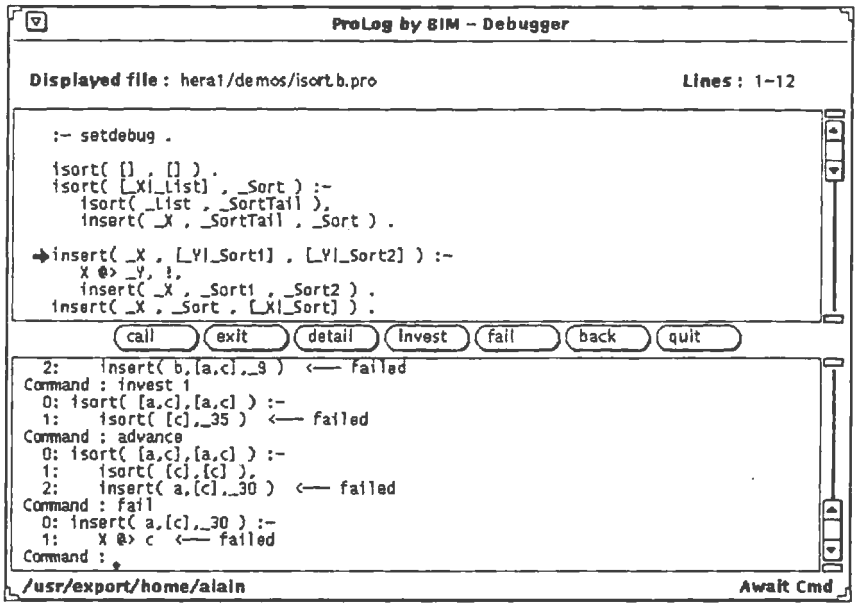
This, and a look at the source indicated in the *Source Window*, reveals the bug: the atom X is compared to the variable _Y: the leading underscore for the variable _X was forgotten.

The analyzer is terminated with the *quit* command:

Command : quit

Below is a snapshot of the *Debugger Window* at the end of the analyze session.

The analyzer - snapshot



7.5 Customizing the debugger

ProLog by BIM has several facilities to customize its debug tools. There are two main reasons why this is desirable. For ease of use, the programmer must be able to adapt the debug tools to his personal debugging style. As the available debug tools are very extended, a programmer usually selects a subset of it to work with. By providing the necessary facilities, this can be made easier. The second reason is that, for optimal performance, the debugger must be tuned towards the application to be debugged. Depending on the type of application program (and of the bug), different debugging strategies are needed for best results. For each kind of data structure that is used in an application, other enquiry procedures should be provided for retrieving this data during a debug session. Again, customizing facilities enable the definition of these strategies and procedures.

Customizing commands

The customizing facilities of ***ProLog by BIM*** are provided in the form of a set of debugger commands. These include the commands *alias*, *button*, *unbutton*, *menu*, *unmenu*, *command* and the printing mode commands. Each of these are treated in the following paragraphs.

The alias command

The *alias* command is used to give another name to a predefined command, or to give a name to a sequence of commands. It can be used for abbreviating frequently used commands or for taking together frequently used sequences of commands into a single command. Another possible usage is to mimic other debuggers, by renaming the commands.

There are two forms of the *alias* command:

alias
alias *key string*

In the first form, without arguments, a list of defined aliases is printed out. The second form defines an alias. The *key* is the name that is given to the *string*. The *key* must be a simple atom. If there is already an alias with that name, it will be removed. The *string* can be any combination of debugger commands and other text. If it consists of more than one atom, it must be enclosed within quotes (single or double). Whenever the *key* is used at the beginning of a debugger command, it is replaced by the associated *string*. This substitution is done only once for each command. As a result, an alias cannot be defined in function of other aliases. These would not be substituted any more, leading to a debugger syntax error.

The classical box commands (like *creep*, *skip*, *leap*, ...) have predefined aliases (*c*, *s*, *l*, ...).

As an example, the following aliases could be defined in case the source-oriented debugger is frequently used from an ordinary terminal. In such an environment, there are no buttons for the commands. It is desirable to have short names for the commands.

```
alias s step
alias n next
alias c cont
alias si "stop in"
alias sa "stop at"
```

Remark that these definitions override the standard predefined aliases (where for instance *s* stood for *skip*). With the above set, a break point can be set at a line as in the next example:

```
sa 'hanoib'517
```

A sequence of commands is formed by separating the commands with ' ;' (a blank and a semi-colon). Such a sequence can also be given a new name with *alias*:

```
alias sp "step ;print _X"
```

The above example defines an alias for a sequence of two commands. By issuing the command *sp*, first the command *step* is executed, immediately followed by a *print* of the variable _X.

The button and menu commands

These commands are only relevant in the *Debugger Window*. Their purpose is to define the contents and the lay-out of the command buttons and the command pop-up menu in the *Debugger Command Bar*.

The form of these commands is:

button *command*
unbutton *command*
menu *command*
unmenu *command*

With *button* a button for the indicated *command* is added to the right of the buttons in the *Debugger Command Bar*. The command *unbutton* removes the button for the indicated *command*, if it was displayed. Using a combination of these two commands, the existing default lay-out and composition of the *Debugger Command Bar* can be changed totally.

The command *menu* adds an item for the indicated *command* at the end of the pop-up command menu. With *unmenu*, the indicated *command* is removed from the menu.

Note:

There are two sets of command buttons and two menus in the *Debugger Command Bar*. One for the box- and source-oriented debugger, and one for the post-execution analyzer. The *button* and *menu* commands have only effect on the set corresponding to the debug mode that is used at the moment they are executed.

The command command

With the command *command*, it is possible to specify user-defined commands. A user-defined command consists of a command keyword and an associated predicate. Whenever the keyword is used as a debugger command, the debugger calls the predicate. There are different possibilities for passing arguments to that predicate.

The following forms can be used:

command
command *cmd*
command *cmd pred*
command *cmd pred:type*
command *cmd pred:type:modifier*

In the first, without arguments, the currently defined commands are printed out.

The second form, with one argument, erases any existing definition for *cmd*.

The last three forms, each with two arguments, specify the user-defined command *cmd*. Any existing definition is overridden. The name of the associated predicate is *pred*. The optional *type* specifies the type of the arguments that must be passed to the predicate. If this *type* is omitted, the predicate will be called without arguments. In that case, the predicate has arity 0. When a *type* is specified, a type *modifier* may be added. This determines the number of arguments to be passed. In both cases, with or without *modifier*, the predicate has arity one.

The table below gives an overview of available types.

Available types for user-defined commands.

Type Specifier	Argument Type	Argument Description
lit eral	atom	The selection, taken literally.
number	integer	The selection, expanded to a number.
atom	atom	The selection, expanded to a legal atom.
linenr	[atom,atom, integer]	Descriptor for the selected source line number. Form: [_FilePath,_FileName,_LineNumber]
filename	[atom,atom]	The selection, expanded to a file name. Form: [_FilePath,_FileName]
predname	atom/integer	The selection, expanded to a predicate description. Form: _PredName/_Arity
varname	atom	The selection, expanded to the name of a variable.
varvalue	term	The value of the selected variable.

As can be seen from the table, the user-defined commands are fully integrated with the *Debugger Window* environment. Their arguments can be selected with the mouse. Depending on the specified type, the selection will be expanded to a value of the right type, before passing it as argument to the predicate. For all cases, a single value as passed as argument. This single value is in certain cases a composed term, making up a descriptor. A source file name is described by its directory path and its file name. A source line is described by a file name descriptor, and a line number. Both of these descriptors are represented with a list. A predicate descriptor has the usual form of name/arity. A variable name is an atom that is the name of the variable without the leading underscore (in *compatibility* syntax mode, there is no leading underscore for variable names). The argument that is passed for type *varvalue* is not the name of the selected variable, but its value in the focussed environment. That can be any Prolog term.

There are two possible modifiers: *list* and *optional*. The *list* modifier indicates that more than one argument of the given type may be selected. All selected arguments are packed in a list that is passed as argument to the predicate. Even when there is only one selected argument, it is passed within a list. With the *optional* modifier, either one or no arguments are passed. If nothing appropriate is selected, the empty list [] is passed as argument.

The next example shows to use of *command* for defining a command for structured data enquiry.

```
command printstruct print_term:varvalue
button printstruct
```

The command with as name printstruct is defined. Its associated predicate is print_term/1 and the argument it will be passed, is the value of the selected variable. There must be a definition for print_term/1, with a head that looks like:

```
print_term( _Term ) :- ...
```

This predicate must then print out its argument in a structured manner.

A possible call of this command would be

```
printstruct _X
```

This can be typed in manually or using the created button.

The next example provides a user-defined command for record enquiry. It is useful in programs that use the record data base to store data.

```
command printkey print_key:atom:list
button printkey
```

The *list* modifier is used to enable the usage of double key records. By providing two definitions for the associated predicate print_key/1, both cases of single and double record key can be solved with one command. The beginning of these two clauses look like:

```
print_key( [_Key1,_Key2] ) :-
      recorded( _Key1 , _Key2 , _Value ),
      ...
print_key( [_Key1,] ) :-
      recorded( _Key1 , _Value ),
      ...
```

These are possible calls of the command:

```
printkey domain1 , key1
printkey key2
```

Again, these calls can be typed in by the user, or they can be generated by selecting the keys somewhere on the screen and clicking the printkey button.

Write mode commands

The write mode commands specify how the debugger must write out its information. This is similar to, but independent from the normal system write options (specified with the **please/2** built-in).

These are the possible forms of write mode commands:

```
Depth
Depth n
Module
Quote
Prefix
```

The *Depth* command, without arguments, prints out the current setting of the write depth. With an argument, the write depth is set to *n* levels. Terms that have more nesting levels than *n* are printed with ellipsis for its subterms at that depth. Setting the depth to -1, means that it must be unlimited.

The *Module* command toggles the printing of module qualifications for atoms.

The *Quote* command toggles the printing of quotes around atoms with special characters or around operators.

The *Prefix* command toggles the printing of operators in normal functor form.

Saving customizing

The composition and lay-out of the buttons and menus in the *Debugger Command Bar* can be predefined on a per-user basis using the defaults data base (defaultsedit for SunView and .Xdefaults for XView). This is explained in detail in the ***ProLog by BIM*** *Reference Manual*. Other customizable attributes in this data base are the lay-out and dimension of the environment windows.

The other debugger commands can be saved in a *.pro* initialization file. This file is silently consulted by ***ProLog by BIM*** when the engine is started up. The way to store these commands in such a file, is by giving them as arguments of calls of the built-in **debug/1**. The argument of this built-in can be a single atom or a list of atoms. Each of these atoms is assumed to be a debugger command. These commands are executed by the debugger when the calls of **debug/1** are executed.

The following calls could appear in a *.pro* file:

```
?- debug( [ 'alias s step' , 'alias n next' , 'alias c cont'
] ) .
?- debug( 'alias sa "stop at"' ), debug( 'alias si "stop in"'
) .
```

Note:

There are two possible locations for a *.pro* file. ***ProLog by BIM*** looks first in the current directory. If it does not find a *.pro* file, it looks in the user's home directory for it. The file in the home directory is meant for global user customizing. The file in the current directory is useful to customize the system on a per-project basis. If both the local and the global files must be used, the global file should be consulted from the local file. This is simply accomplished by adding the following query to the local *.pro* file:

```
?- consult( '~/.pro' ) .
```

The ProLog by BIM Graphical User Interface Generator : CARMEN

Carmen 1.3

Chapter 0
Introduction

0.1 Introduction

The problem

Where the use of a high-level programming language like Prolog usually accelerates the overall process of software development due to its compact nature and its powerful concepts, this is unfortunately not true for developing the user interface part of a complex application. However sophisticated the interfacing toolkit may be, developing a user interface requires a great deal of clerical and low-level manipulations that are time consuming in the development phase, and especially so in the testing and debugging phase.

The solution

Written in ***ProLog by BIM*** and running under the available **Sun™** Window System (X11/NeWS), **Carmen** (Computer Aided Realization and Management of Environment Networks) is built up around a graphical editor, that not only allows the designer to easily sketch the different screens, including all of their attributes like text fields, buttons, pop-up menus, toggles, etc., but also to link these screens and their associated interaction modes together in a smooth and logical dialogue structure. On the basis of this definition, Carmen generates an 'empty' application, i.e. all the code needed to display the different screens with their associated buttons and pop-up menus and to link them together is generated. It suffices then to fill in the blanks, i.e. to write the Prolog code that executes the action to be associated with the buttons and pop-up menus. Even this is facilitated by Carmen thanks to the generation of the appropriate headings of the action code, as was specified by the developer during the creation of the screens. A few minutes of window sketching and dialogue definition thus eliminates the laborious production of hundreds of lines of interfacing code which would otherwise require days of low-level programming.

An immense gain in productivity!

0.2 Installation of Carmen

Please refer to the ***ProLog*** installation guide in order to install Carmen. This will create a prelinked version of Carmen.

0.3 Carmen v1.3 release notes

Changes to the Carmen software

- Support for loading multiple projects at run-time.
  Each of the projects can be initialized and activated in any order.
  A project can be terminated and re-initialized for a new activation.
- Color support.
  All objects can get a (named) color. Standard X11 color naming is used (including XCMS conventions). Drawing predicates provide color as well.
- Enhanced and extended drawing predicates.
  A new set of drawing predicates is defined, starting with 'ui_drawable_'.
  All existing X11 graphics functions are provided.
  Graphical attributes can be specified with each call. All X11 graphics context parameters can be set via these attributes.
  These predicates replace the 'ui_canvas_' predicates.
- Creation of drawable objects is possible from the application.
- Files with image descriptions can now also be X bitmap files.
- Miscellaneous inquiry functions for defined projects and objects.
- The notifier predicates will write out the values of its arguments.

Changes to the Carmen installation

The Carmen generator will now looks into $BIM_PROLOG_DIR/lib/carmen to find its run-time libraries. Existing Carmen application will still when one copies the libraries to $BIM_PROLOG_DIR/lib .

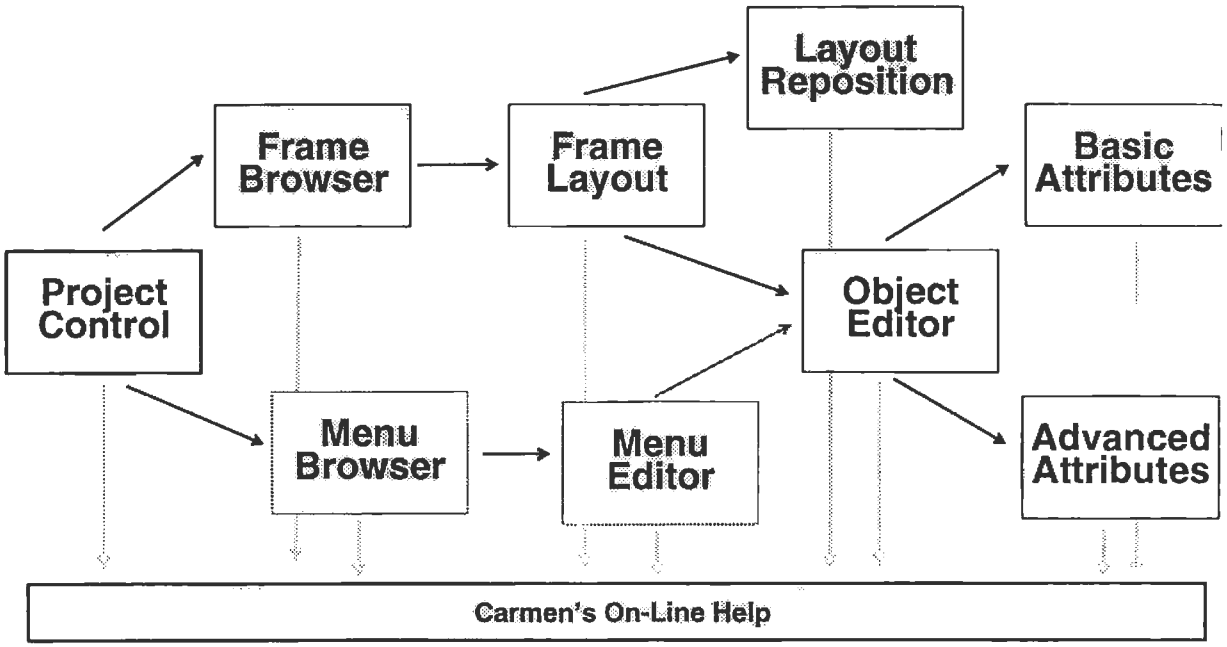
Carmen 1.3

Chapter 1
Reference Guide

1.1 Structure of the Carmen GUI generator

The structure of the Carmen interface reflects the structure of the generated window interfaces. It has essentially two branches: one for frames and one for menus. Each branch is divided into a *browser* part and an *editor* part. There is also a connection between the two branches.

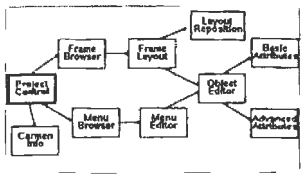
The following figure represents the different frames of the Carmen interface with their logical structure.



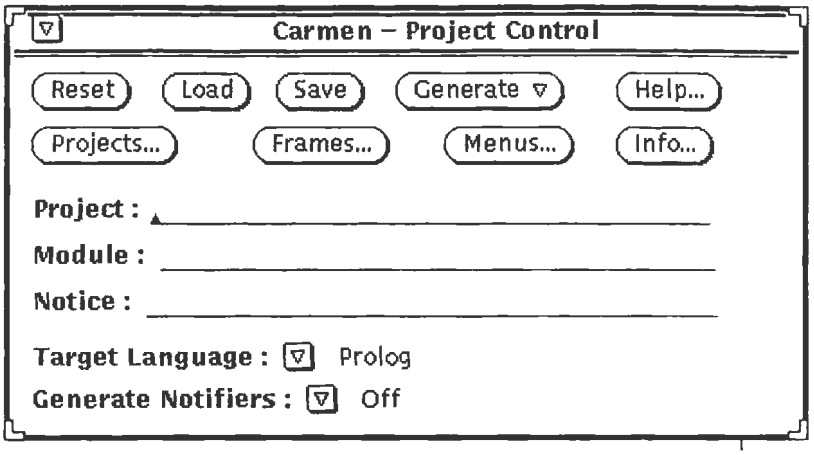
How to use Carmen

The usage of Carmen is explained window by window. Starting from the **Project Control** frame, going over the Frame manipulation frames to the Object frames, and then treating the Menu manipulation frames.

Project control



The main frame of Carmen provides for general control over the project. A project covers the entire window interface design, management and generation. To this purpose, the frame contains a number of buttons and a number of information fields, all collected in the control panel.



Control panel - fields

Project
This TextField must be set to the project name. This name will be used as base name for the files (see below under Files). The project name is automatically set when applying a project in the Project browser (see below "*Project Browser*").

Module
The module name for the user interface. Default is the global module.

Notice
A notice that is included in the generated user interface code. This is useful to include a copyright notice in the code.

Target Language
This SelectButton must be set to the desired language for the generated code. The only current choice is Prolog (see below under "*Files*").

Generate Notifiers
Enables or disables the generation of notify handler stubs. These are headers for all notify handler routines that appear in the interface description. They are generated for the same target language.

Control panel - buttons

Reset
Clears the data base of Carmen. The project that is being edited, is thrown away.

Load
The project, indicated in the Project field, is loaded into Carmen. Any other project that was being edited, is first thrown away.

Save
The project that is edited, is saved in an internal description form, named as indicated in the Project field. No code is generated with this operation. The project remains loaded.

Generate
Code is generated for the loaded project. The language is taken from the Target Language field, and the window system from the menu that pops up from this MenuButton. The only current choice is XView.

Projects
WindowButton that activates the Project Browser frame.

Frames
WindowButton that activates the **Frame Browser** frame.

Menus
WindowButton that activates the **Menu Browser** frame.

Info
This WindowButton activates a frame containing some information on Carmen.

Files

***project*.wid**
This file contains the window interface description for *project*. It is created (or changed) by the Save operation, and read in by the Load operation.

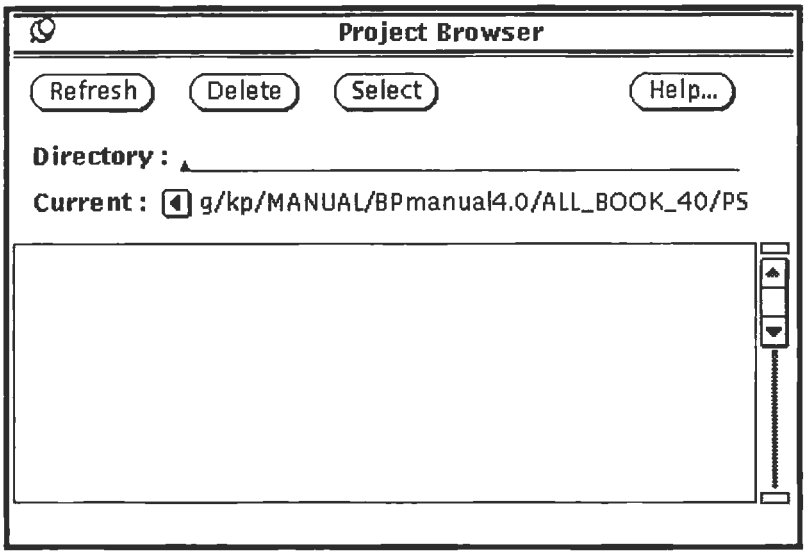
***project*.nh.pro - *project*.nh.c - *project*.nh.a**
File with notify handler routine stubs for *project*. This is generated by the Generate command, if the Generate Notifiers field is set to On.

***project*.xv.pro - *project*.xv.c - *project*.xv.a**
XView user interface code for *project*. It is generated when the XView item is chosen on the Generate MenuButton.

If a file whose name already exists must be created, the existing file is first backed up into a file with as name the original name extended with a '%'.

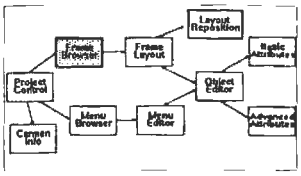
Project browser

The **Project Browser** frame consists of a control panel and a display list.

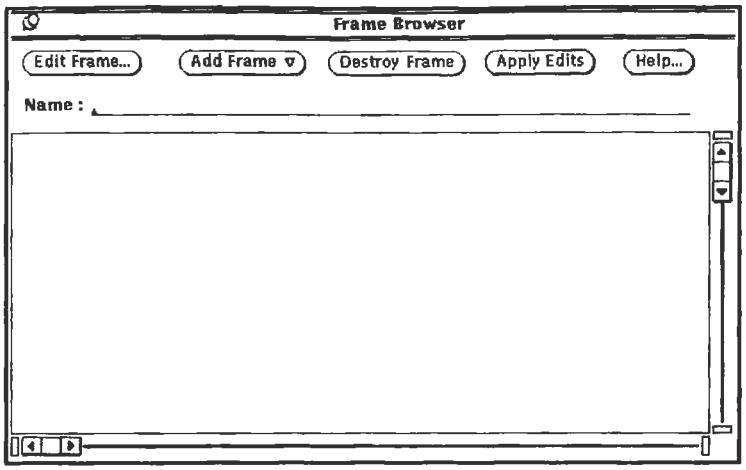


The Project Browser display list gives project names (normal font) and subdirectories (bold font). Clicking on a directory opens it and replaces the contents of the display list with the contents of that directory. A project is selected by clicking on its name, followed by the **Select** button. To delete a whole project, the **Delete** button must be clicked after clicking on the project name. The **Directory** field indicates the path of the displayed directory. It can be edited to open another directory. This is accomplished by clicking the **Refresh** button.

Frame browser



The **Frame Browser** frame has a control panel and a display list. The control panel contains a set of buttons and an information field. The display list has several columns.



Control panel

Name
This field holds the name of the *selected frame*. It is filled in when a frame is selected. Before creating a new frame, it must be changed to the name of the new frame.

Edit Frame
If a frame is *selected*, it now becomes *focussed*, and is loaded in the Frame Layout frame to be edited.

Add Frame
To create a new frame, after filling in the name, chose from this MenuButton between creating a new sub frame or a new base frame. A new sub frame is added to the previously *selected frame*'s list of sub frames.

Destroy Frame
The currently *selected frame* is destroyed, together with all its sub objects. If it is a base frame, then its sub frames are also destroyed.

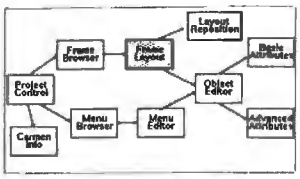
Apply Edits
After changing the name in the Name field, this new name can be given to the *selected frame*, by pressing this button.

Display list

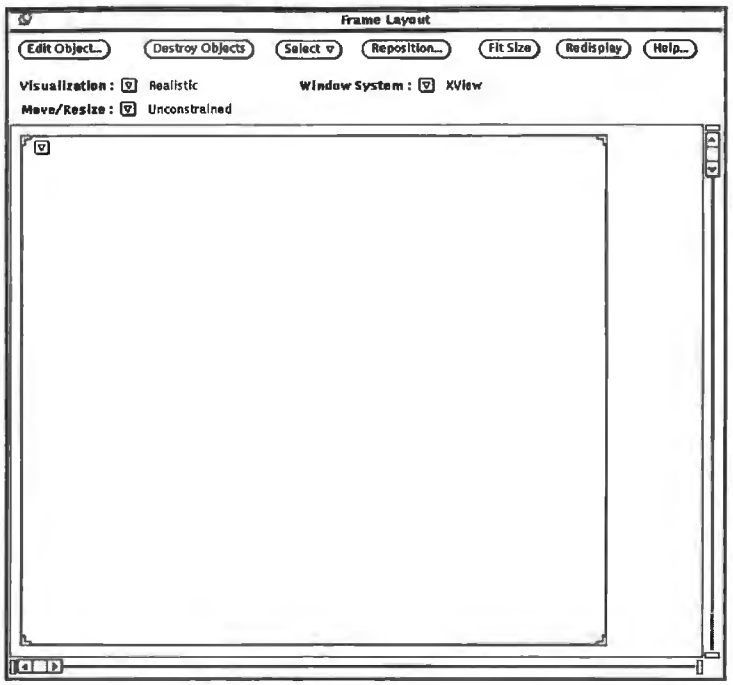
Data
Each column on the display list represents a base frame with its sub frames. The base frame is shown at the top in bold. The first column's base frame is the main base frame of the interface.

Actions
Clicking SELECT on a frame name on the display, makes that frame *selected*.

Frame layout



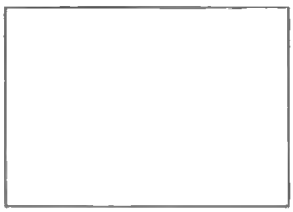
This frame has a control panel with command buttons, state switch fields, and a graphical window. It displays the *focussed frame* and is used to change the layout of the frame and to edit its objects.



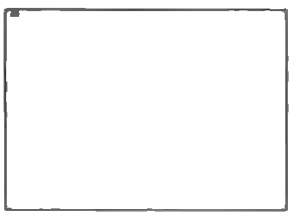
The basic layout operations on a frame are done by direct manipulation on the graphical window. This means the objects are selected and handled directly with the mouse.

There is a notion of *selected object set*, referring to the set of selected objects. All selected objects have small black *resize handles* in their corners, on the graphical window.

In Outlines mode



In Realistic mode



Control panel - fields

Visualization
This switch controls the visualization mode on the graphical window. It can be set to **Outlines** or to **Realistic**. In Outlines mode, the objects are represented by their bounding box rectangles. The Realistic visualization shows the objects in a very close approximation of their real life form.

Window System
As the layout of objects is window system dependent, it must be tuned towards the desired window system. Currently only XView is supported.

Move/Resize
This switch can be set to Unconstrained, Horizontal Only or Vertical Only, and determines how move and resize operations are to be constrained. If set to an Only value, the objects remain unchanged in the other direction. I.e., when an object is moved in Horizontal Only restriction, it will remain at the same vertical position.

Control panel - buttons

Edit Object
If there is one single object in the *selected object set*, that object is made *focussed object*, and the **Object Editor** frame is popped up for edition of this object.

Destroy Objects
All objects in the *selected object set* are destroyed, together with all their sub objects.

Select
This MenuButton provides an alternative way for changing the *selected object set*. By choosing the select Parents from its menu, the *selected object set* is changed to hold all parent objects of all objects that were previously in the *selected object set*.
The select Sub Objects command, changes the *selected object set* to the set of all sub objects of all previously *selected* objects.

Reposition
A frame is activated with facilities for automatic repositioning of the *selected object set*.

Fit Size
The sizes of the objects in the selected object set, are adapted to fit the values of the object's attributes. Normally this is done automatically whenever an attribute is changed.

Redisplay
The graphical window is entirely redrawn.

Graphical window - actions

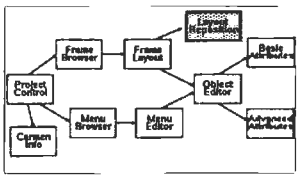
Object selection
The *selected object set* is manipulated by performing selection operations with the SELECT or ADJUST mouse buttons.

The set is cleared (made empty), by clicking SELECT on the background (aside from any object).
The set is filled with a single object by clicking SELECT above that object. In case there is more than one object under the mouse cursor, the object of the highest level is selected.
An unselected object is added to the set by clicking ADJUST on it.
A selected object is removed from the set by clicking ADJUST on it.
An accelerator for selecting multiple objects at once, is to drag SELECT over a number of objects. All objects that are completely within the bounds of the dragged rectangle, become the *selected set*.
Dragging ADJUST over a number of objects has as effect that unselected objects, falling in the dragged rectangle, are added to the set, and the selected objects in the rectangle, are removed from the set.

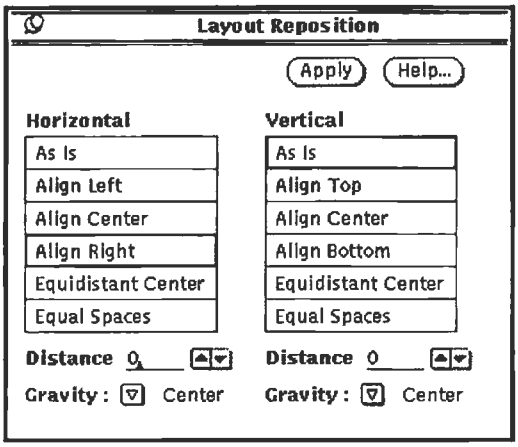
Moving objects
All objects in the *selected object set* are moved as a whole, retaining their relative positions, unless this would bring them out of their parent's border.
The *selected object set* is moved by dragging SELECT from within a selected object. During this dragging, a bounding rectangle reflects the new position, and the coordinates of the upper left corner are shown on the Control Panel.
In case an object would cross its parent's borders after a move, it will be moved back as much as necessary in horizontal and/or vertical direction to get it completely inside its parent.

Resizing objects
A *selected object* can be resized by grabbing a resize handle and dragging SELECT. The corresponding corner of the object is moved away from or towards the others. During the dragging, a rectangle reflects the new size of the object, and the width and height values are displayed on the Control Panel.
Expanding an object outside of its parent is impossible: the expansion is automatically clipped by the parent's borders.The opposite is also impossible: an object cannot be shrunk that much that its sub objects would cross its borders. Again the shrinking is clipped by the bounding box of all sub objects.

Layout reposition



The **Layout reposition** frame provides a number of facilities for automatic repositioning of the *selected object set*. The objects can be repositioned in horizontal and vertical direction, in a single operation. How they have to be moved, is indicated for both directions independently. This can be either by aligning or distribution.



For distribution operations, the *selected object set* must contain objects that have the same parent. For an alignment, the parent itself may also be a member of the *selected object set*. Repositions that would cause the objects to exceed their parent borders, are refused.

Moving modes

As Is
The objects remain at their position in that direction.

Align Left/Top
The objects are moved until their left/top edges are at the same position.

Align Center
The objects are moved until their centers are at the same position.

Align Right/Bottom
The objects are moved until their right/bottom edges are at the same position.

Equidistant Center
The objects are distributed until their centers are at the same distance from each other, indicated in the Distance field. The relative ordering of the object centers remains invariant.

Equal Spaces
The objects are distributed such that the space between two adjacent objects is the same, as indicated in the Distance field. The relative ordering of the object centers remains invariant.

Moving conditions

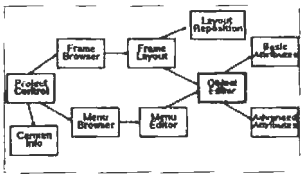
Distance
This field indicates the distance between two objects or their centers, for distribution operations.

Gravity
The gravity tells which side (Left/Top, Center or Right/Bottom) of the bounding box of the *selected objects* should remain at its location in a repositioning.

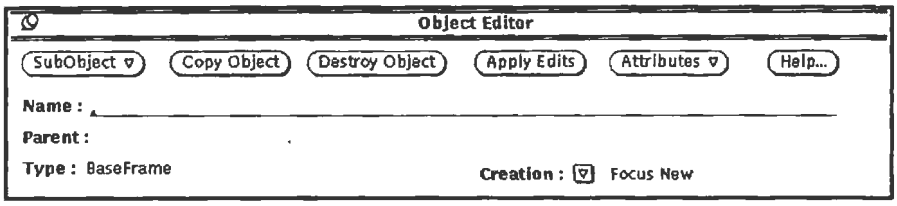
Buttons

Apply
Applies the specified repositioning to the *selected object set*, unless this would bring some object outside of its parent's borders.

Object editor



This frame holds the *focussed object*, and is used to perform non-geometrical operations to that object. It has only a control panel with a set of buttons and information fields.



Control panel - fields

Name
Here, the name of the *focussed object* is displayed. Before creating a new (sub) object, it has to be set to the name of the new object.

Parent
This field displays the name of the *focussed object*'s parent object.

Type
The type of the *focussed object.*

Creation
Determines how the focus must be changed when a new object is created. This can either be Focus New, in which case the newly created object becomes focussed, or Freeze Focus, which means the originally (parent) *focussed object* remains focussed.
The latter mode is very convenient when a whole series of sub objects of a same parent must be created sequentially. The former mode is more convenient when the new object will be adapted immediately to its final state.

Control panel - buttons

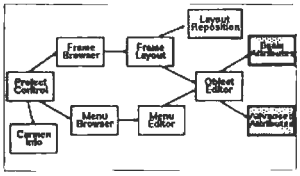
SubObject
If the type of the *focussed object* permits it, a menu of possible sub object types is provided with this MenuButton. Choosing an item from it, creates a new object with the previously *focussed object* as parent. The new object gets default attribute values and a default position, which is the upper left corner in its parent.

Destroy Object
The *focussed object*, and all of its sub objects, is destroyed. A Frame object cannot be destroyed with this operator. To destroy a Frame, the Destroy Frame operator on the **Frame Browser** should be used instead.

Apply Edits
If the Name field is edited, the *focussed object* will get this new name.

Attributes
This MenuButton holds two WindowItems, to pop up the **Basic** and **Advanced Attributes** frames with the attributes of the *focussed object.*

Attributes

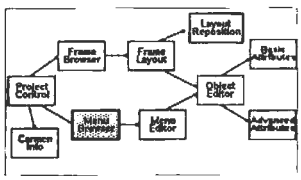


The **Basic** and **Advanced Attributes** frames hold a control panel with fields that are dependent on the type of the *focussed object.* The field labels are the names of the attributes and the type of each field depends on the type of the attribute. Besides this variable information, the control panel also holds two buttons.

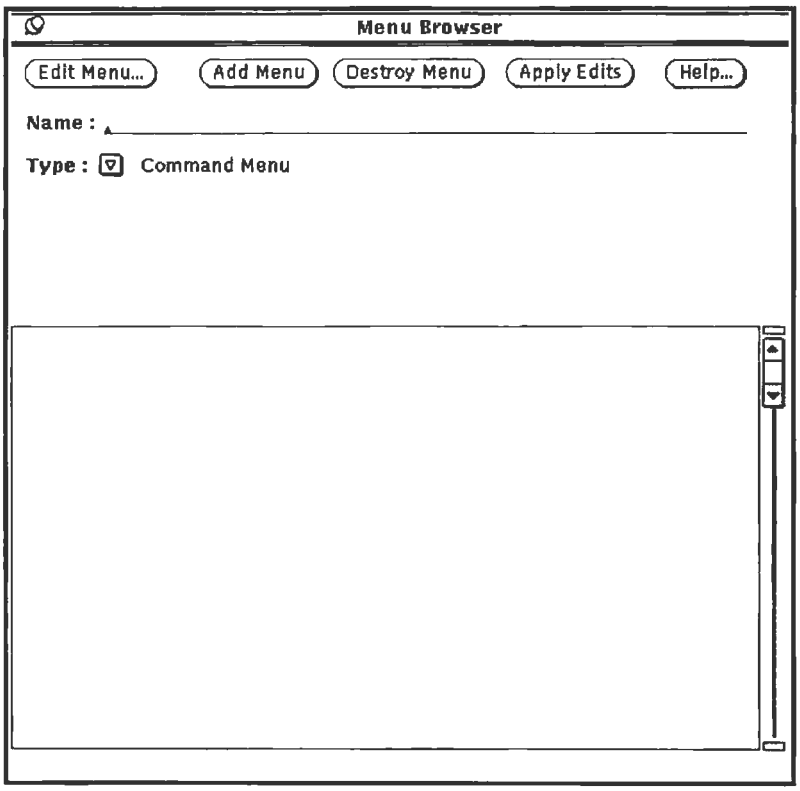
Apply
The new values of the attribute fields are applied to the *focussed object.*

Cancel
The edits to the attribute fields are undone. The fields are updated to reflect the current attribute values of the *focussed object.*

Menu browser



The **Menu Browser** frame has three parts: a control panel, an attribute panel and a display list. It gives an overview of the defined menus, and is used to manipulate menus.



Control Panel - fields

Name
This TextField displays the name of the *selected menu*. It is filled in whenever a menu is selected. Before creating a new menu, it must be set to the name of the new menu.

Type
The type of the *selected menu* is shown in this SelectButton. Each time a menu is selected, it is switched to reflect the new menu type. It must be set to the desired type before creating a new menu.

Control panel - buttons

Edit Menu
The *selected menu* is made *focussed*. It is loaded in the **Menu Editor** that is popped-up, for editing the menu.

Add Menu
A new menu, with name and type as given on the control panel information fields, is created. It is added at the end of the list of defined menus.

Destroy Menu
The *selected menu* and all its items, are destroyed. If the menu has MenuItems, the associated menus are not destroyed.

Apply Edits
The edited Name field value is taken as new name for the *selected menu*. Any change to the Type field is refused: it is not possible to change the type of an existing menu. If the menu has attributes, the values of the attributes are taken from the attribute panel.

Attribute panel

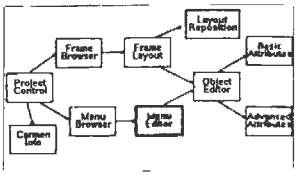
The attribute panel displays the attributes of the *selected menu*.

Display list

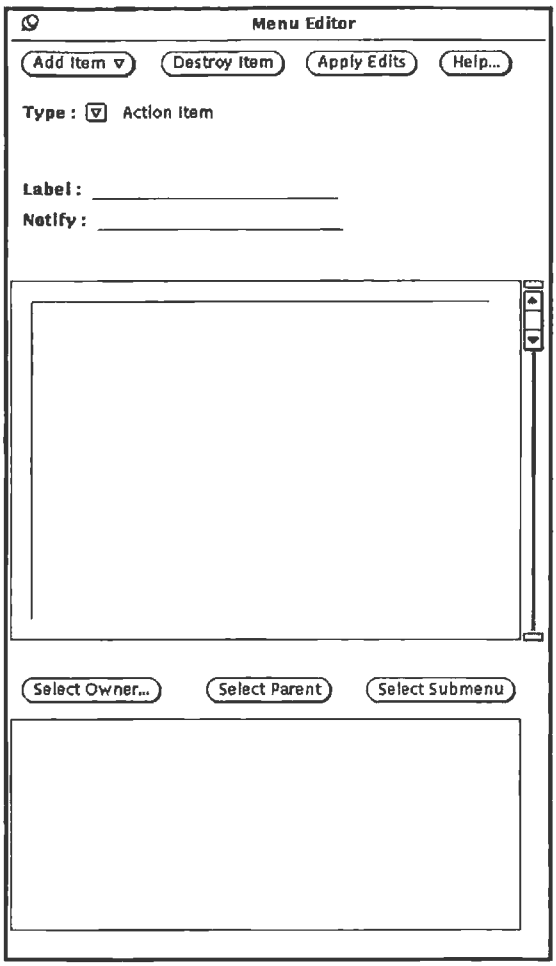
Data
The display list has one column, containing a list of all defined menus.

Actions
Clicking SELECT on a menu name on the display, makes that menu *selected*, and brings its information in the control and attribute panels.

Menu editor



This frame shows the *focussed menu* and provides editing facilities for that menu. It consists of a control panel, an attribute panel, an item display list, another control panel and an object display list, in that order from top to bottom.



Attribute panel

The attribute panel's fields are menu item specific. They depend on the type of the *focussed menu item.*

Control panel - fields

Type
This SelectButton shows the type of the *focussed menu item.* It is switched each time an item is selected. Before creating a new item, it must be set to the desired item type.

Control panel - buttons

Add Item
A new item of the type indicated on the Type field, is created. It is added either to the end of the menu, or after or before the *focussed menu item*, depending on the choice on the menu that is associated to this MenuButton.
The attribute values for the item are taken from the attribute panel.

Destroy Item
The *focussed menu item* is destroyed. Items below it, are moved up one place.

Apply Edits
The *focussed menu item*'s attributes are set to the values in the attribute panel. Changes of item type are refused. It is impossible to change the type of a menu item.

Item display list

Data
This display shows the list of menu items on the *focussed menu*, in the same order as they will appear on the menu.

Actions
Clicking SELECT on an item label on the display, makes that item *focussed*. Its type is indicated on the control panel, and its attribute values are filled in on the attribute panel.

Control panel2 - buttons

Select Owner
The *selected owner object* is made *focussed*. The **Object Editor** is popped up with that object.

Select Parent
If the *focussed menu* is associated to a MenuItem, and it was made *focussed* via the Select Submenu operation, the menu that holds the MenuItem from which it was *focussed*, is loaded back into the **Menu Editor**.

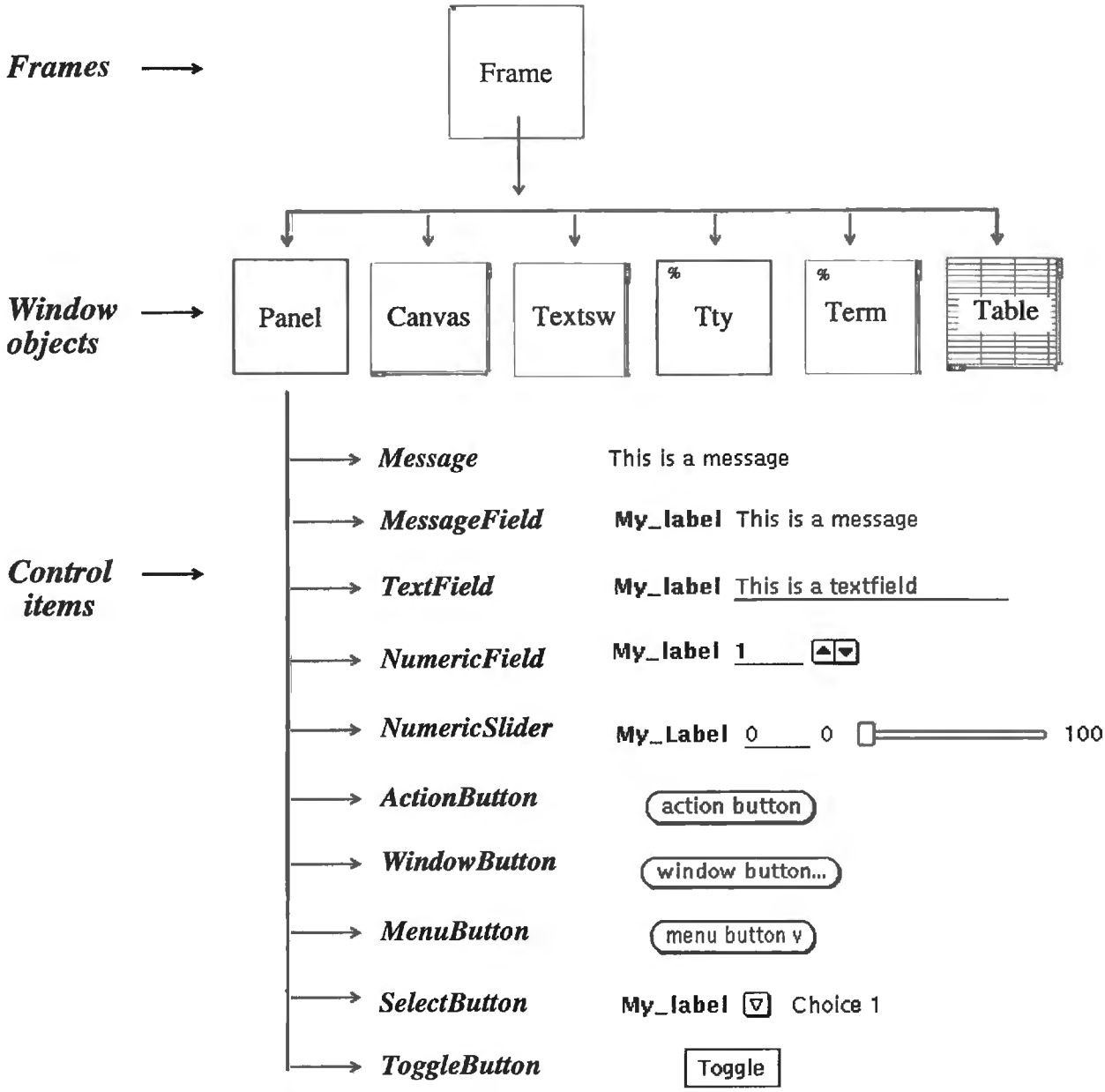
Select Submenu
If the *focussed menu item* is a MenuItem, its associated menu is made *focussed* and loaded into the **Menu Editor**.

Object display list

Data
This display holds a list of all objects referring to the *focussed menu*.

Actions
Clicking SELECT on an object name on the display, makes that object *selected owner*.

1.2 Carmen's objects



Callbacks

Certain objects have an attribute **Notify** that can be used to specify a **callback** for the object.

A **callback** is a predicate that is called when a specific event occurs in the object. Usually this event is caused by the user clicking the mouse on the object.

The number of parameters for a callback varies from object to object. As a general rule, all callbacks have the same first two parameters.

The **first** one is the **object handle** on which the event occurred.
Remark that this is not the name of the object. To convert an object handle to an object name, the library predicate ui_handle_to_name/2 can be used (see section 5.1).

The **second** parameter of a callback is an **event handle**. This is a pointer to a structure describing the event that occurred. This event can be investigated with more advanced window system predicates.

Label & images

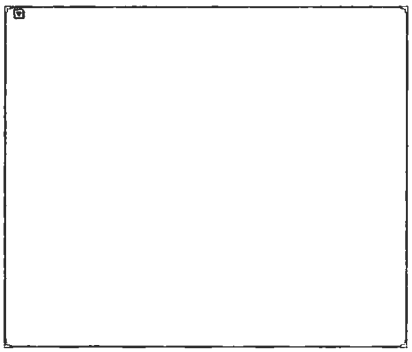
For each object having a label field, it is possible to have an image appear instead of a simple text. The notation for this attribute value is the following:

S:<my_label>	for a string label with text <my_label> The 'S:' may be omitted.
I:<my_file>	for a picture label that resides in file <my_file>

The picture file must be in standard XView or X image format. These are the formats that are used in image files produced by *iconedit* and *bitmap* respectively.

Frame

A Frame object is an object that represents a frame. All variants of frames (main frame, base frame and sub frame) are represented by the same Frame object.



A Frame has the form of a rectangle with a border all around and with an additional name stripe at the top.

The behavior of the Frame is determined by the window manager. All events that fall on the Frame's background (its border and name stripe), are interpreted by the window manager.

Basic attributes

Label
The text that is displayed in the Frame's name stripe.

Icon
The name of the file containing a description of the Frame's icon. This is only meaningful for base frames.

Advanced attributes

Menu
The name of the pop-up menu that appears when the right mouse button is pressed.

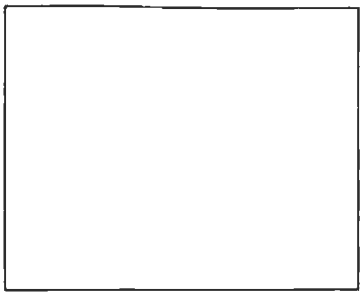
ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

Background
Name of a color to be used as background.

Panel

A Panel is an object that reserves an area to hold a number of control items.



It is a rectangle, that can have a border, depending on the window system that is used.

Basic attributes

None.

Advanced attributes

Visible
Whether the object is visible or not.

Menu
The name of the pop-up menu that appears when the right mouse button is pressed.

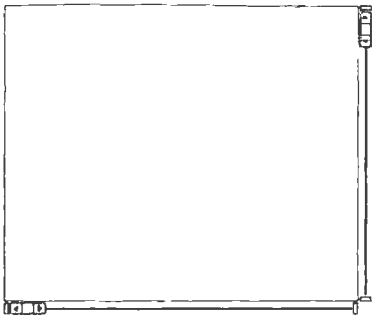
ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

Background
Name of a color to be used as background.

Canvas

A Canvas is mainly a drawing area. It can also accept input events from the user. It consists of an inner area (the *drawable*), on which the drawings are done, and of a viewport that displays the visible section of the drawing area.



It is represented by a rectangular area. Depending on the window system, it can have a border. If the drawable is bigger than the viewport, a scrollbar is attached to the border of the Canvas, in horizontal and/or vertical direction.

The scrollbars can be used to scroll the viewport over the drawable. Mouse events, like clicking and dragging, of each of the three mouse buttons (SELECT, ADJUST and MENU), are reported to the application, if desired.

Basic attributes

Drawable
Name of the Canvas's drawable.

Width
Width of the drawable in pixels.

Height
Height of the drawable in pixels.

ClickLeft
Callback routine for a click of the left mouse button.

arg1: Canvas handle.
arg2: Event handle.
arg3: Event position (x, y coordinate tuple).

DragLeft
Callback routines for a drag of the left mouse button. The first callback is called when the drag starts, the second is called for every move and the third is called when the drag finishes.

Drag start callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Event position (x, y coordinate tuple).

Drag to callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Previous event position (x, y coordinate tuple).
arg5: Event position (x, y coordinate tuple).

Drag stop callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Event position (x, y coordinate tuple).

ClickMiddle
Callback routine for a click of the middle mouse button.

arg1: Canvas handle.
arg2: Event handle.
arg3: Event position (x, y coordinate tuple).

DragMiddle
Callback routines for a drag of the middle mouse button. The first callback is called when the drag starts, the second is called for every move and the third is called when the drag finishes.

Drag start callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Event position (x, y coordinate tuple).

Drag to callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Previous event position (x, y coordinate tuple).
arg5: Event position (x, y coordinate tuple).

Drag stop callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Event position (x, y coordinate tuple).

ClickRight
Callback routine for a click of the right mouse button.

arg1: Canvas handle.
arg2: Event handle.
arg3: Event position (x, y coordinate tuple).

DragRight
Callback routines for a drag of the right mouse button. The first callback is called when the drag starts, the second is called for every move and the third is called when the drag finishes.

Drag start callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Event position (x, y coordinate tuple).

Drag to callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Previous event position (x, y coordinate tuple).
arg5: Event position (x, y coordinate tuple).

Drag stop callback:

arg1: Canvas handle.
arg2: Event handle.
arg3: Drag origin position (x, y coordinate tuple).
arg4: Event position (x, y coordinate tuple).

Advanced attributes

Visible
Whether the object is visible or not.

Repaint
Name of the callback routine that repaints the Canvas. If omitted, the Canvas will be self-repainting, as far as the window system supports this.

arg1: Canvas handle
arg2: Drawable handle
arg3: List of damaged rectangles of the drawable

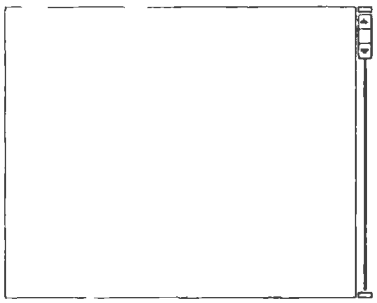
ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

Background
Name of a color to be used as background.

Textsw

A Textsw object is a text editor.



It is represented by a rectangle with a vertical scrollbar.

Keyboard actions are handled by the Textsw object, and the scrollbar is used to scroll through the text.

Basic attributes

None

Advanced attributes

Visible
Whether the object is visible or not.

Font
The font that must be used in the Textsw.

Menu
The pop-up menu of the Textsw.

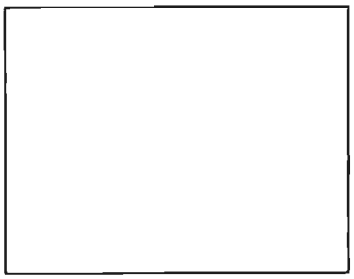
ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

Background
Name of a color to be used as background.

Tty

A Tty object implements a terminal emulator.



Its representation is a rectangle.

It handles keyboard input like any terminal.

Basic attributes

Forkchild
Whether a child process must be forked in the Tty, or not.

ChildCommand
A list of arguments that are used in forking a child process. The first argument is the name of the program that must be invoked. Default is to fork a *shell*.

Advanced attributes

Visible
Whether the object is visible or not.

Font
The font that must be used in the Tty.

Menu
The pop-up menu of the Tty.

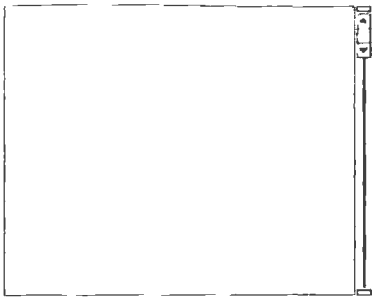
ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

Background
Name of a color to be used as background.

Term

A Term object is a variant of a terminal emulator. It is extended with a history log. Normal keyboard input is handled like in a terminal, and additionally the scrollbar enables scrolling through the history log.



The representation is a rectangle with a vertical scrollbar.

Basic attributes

Forkchild
Whether a child process must be forked in the Term, or not.

ChildCommand
A list of arguments that are used in forking a child process. The first argument is the name of the program that must be invoked. Default is to fork a *shell*.

Advanced attributes

Visible
Whether the object is visible or not.

Font
The font that must be used in the Term.

Menu
The pop-up menu of the Term.

ClientData
Application specific data, associated with this object.

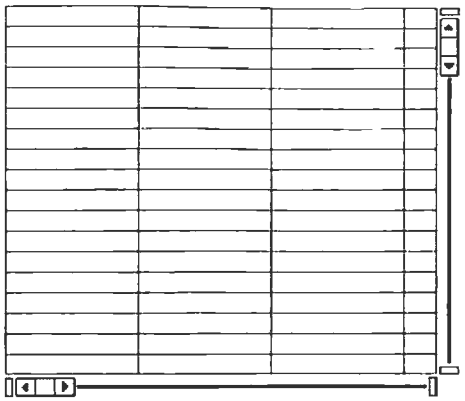
Foreground
Name of a color to be used as foreground.

Background
Name of a color to be used as background.

Table

A **Table** is a kind of simplified spreadsheet. It collects and displays data in a raster. The raster is made up of columns and rows, that can have different sizes. At the top of the raster, there can be a number of fixed rows. The rows below these, are movable: they can be scrolled up and down. Similarly, at the left side of the raster, there can be a set of fixed columns. Data can be rendered in the raster in different formats. It can be adjusted to the left, to the right, or centered in the cell. Another attribute is the font mode: the text is drawn in the plain font or in any combination of bold, dimmed and underlined variants.

These are several possibilities of selecting one or more cells in the Table. An editable Table also offers the possibility to edit the contents of a cell.
A Table is shown in a viewport.



A Table object is represented by a rectangular area, which acts as a viewport. It has scrollbars if the whole Table does not fit in the viewport. Clicking SELECT in a cell causes this cell to become the new selection. Any previously selected cells are deselected (without notification of this fact). The user program is notified of the new single selection. Clicking ADJUST toggles the selection state of the cell. Other cells are not affected. The program is notified of the selection state change. Dragging SELECT over an area of cells makes this area the new selection. the previous selection is deselected (without notification). The multiple selection is notified to the program. Dragging ADJUST over an area of cells toggles the selection state of these cells. The program is notified of the selection state change. Clicking SHIFT SELECT in a cell, enters edit mode for that cell. A window is popped up in which the text that is displayed in the cell can be edited. There are also facilities for changing the Table mode (adjustment and fill style) of the cell. If the changes are accepted, the user program is notified of the new values and attributes.

Basic attributes

Boxed
Whether the cells in the raster are drawn as boxes with visible lines or not.

Columns
The total number of columns in the Table.

FixedColumns
The number of fixed columns at the left side of the Table.

ColumnWidth
Number of characters in columns of default width.

ColumnWidths
List of column widths, in number of characters per column. This list specifies the widths of the first columns. All following columns have default width.

Rows
The total number of rows on the Table.

FixedRows
The number of fixed rows at the top of the Table.

NotifySingle
The callback that will be invoked when a single selection is made.

arg1: Table object handle.
arg2: Event handle.
arg3: Column index.
arg4: Row index
arg5: Text stored in the selected cell
arg6: Selection state of the cell (TRUE if selected, FALSE if unselected)

NotifyMultiple
The callback that will be invoked when a multiple selection is made.

arg1: Table object handle
arg2: Event handle
arg3: First column index
Last column index
arg5: First row index
arg6: Last row index
arg7: List of texts stored in the selected cells (in column first order)
arg8: Selection states of the cells (TRUE if selected, FALSE if unselected)

NotifyEdit
The callback that will be invoked when a cell has been edited.

arg1: Table object handle.
arg2: Event handle.
arg3: Column index.
arg4: Row index
arg5: New text value of the edited cell
arg6: Fill style of the cell
arg7: Adjust mode of the cell

Advanced attributes

Visible
Whether the object is visible or not.

Font
The font to be used for displaying the data.

ColumnGap
The gap, in pixels, between two adjacent columns.

RowGap
The gap, in pixels, between two adjacent rows.

Margin
Width, in pixels, of the margin around the viewport.

Select Feedback
Toggle that determines wether the selected box must be inverted or not.

Editable
Whether the Table cells can be edited or not.

SelectExclusive
Whether it is possible to select more than one cell or only a single cell.

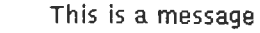
ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

Background
Name of a color to be used as background.

Message

A Message object simply displays a short message.
It consists only of the text, or the image of the message.



No input is accepted.

Basic attributes

Value
Text or image of the message.

ValueBold
Whether the text must be displayed in bold font or not.

ValueWidth
Width of the text value field. Larger values are truncated.

Advanced attributes

Visible
Whether the object is visible or not.

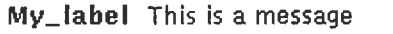
ValueFont
Font for a text value.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

MessageField

This object is a variant of the Message object. It displays as well a label as a message.
Typically, the label is set once for all, while the message value can change frequently.



The MessageField is visualized by the label, followed by the message text.
No input events are accepted.

Basic attributes

Label
Text or image of the label of the field.

LabelBold
Whether the label must be displayed in bold font or not.

LabelWidth
Width of the text label field.

Value
Text of the message.

ValueBold
Whether the text must be displayed in bold font or not.

ValueWidth
Width of the value text. Larger values are truncated.

Advanced attributes

Visible
Whether the object is visible or not.

LabelFont
Font for a text label.

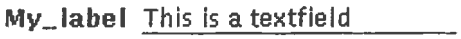
ValueFont
Font for the value text.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

TextField

A TextField is a labeled field that can be used to display and enter a single line of textual data. Its main purpose is to act as an input field for texts.



It is represented by a label, followed by an editable field. This field is underlined if the window system provides this feature.

Keyboard input is accepted and translated to text or editing operations on the text in the field. Pressing RETURN, changes the associated key and invokes a callback.

Basic attributes

Label
Text or image of the label of the field.

LabelBold
Whether the label must be displayed in bold font or not.

LabelWidth
Width of the text label field.

Value
The text data.

ValueBold
Whether the text must be displayed in bold font or not.

ValueWidth
Number of characters that must be displayable in the TextField. On overflow, the text is scrolled horizontally.

Key
Identifier for the associated key. This key, if specified, will hold the value of the TextField, and be updated when this changes.

Notify
The callback that must be called when the TextField value is changed.

arg1: Button item handle.
arg2: Event handle.
arg3: The new text value.

Advanced attributes

Visible
Whether the object is visible or not.

LabelFont
Font for a text label.

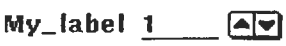
ValueFont
Font for the value text.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

NumericField

A NumericField is a labeled field with a button that can be used to display and enter (integer) numeric data.



It consists of a label, a value field displaying the numeric value, and a button to change the value. The relative layout of these three elements can be changed.

No keyboard input is accepted. The button has two reaction zones: the lower part to decrement and the upper part to increment the value. Clicking SELECT on the button changes the value according to the zone in which the mouse is positioned, with a minimal amount. Clicking SHIFT SELECT changes the value with a higher amount. Leaving the NumericField, changes the associated key and invokes a callback.

Basic attributes

Label
Text of the field's label.

LabelBold
Whether the label must be displayed in bold font or not.

LabelWidth
Width of the label field.

Value
The numerical value.

ValueBold
Whether the value must be displayed in bold font or not.

ValueWidth
Number of digits to represent the value.

ValueMin
Lower bound for the value.

ValueMax
Upper bound for the value.

IncrSlow
Amount with which to change the value in a slow motion.

IncrFast
Amount with which to change the value in a fast motion.

Key
Identifier for the associated key. This key, if specified, will hold the value of the NumericField, and be updated when this changes.

Notify
The callback that must be called when the NumericField value is changed.

arg1: Item handle.
arg2: Event handle.
arg3: Field value.

Advanced attributes

Visible
Whether the object is visible or not.

LabelFont
Font for a text label.

ValueFont
Font for the value text.

Layout
How the three composing elements must be laid out.

NotifyImmediate
Switch for the notification behavior. In Immediate mode, the callback routine is activated on each change of the field, while otherwise, it is only called when leaving the field.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

NumericSlider

A NumericSlider represents a numeric value more visually than a NumericField. The value is shown proportional to its allowed range. This representation is only usable for smaller value ranges. As the accuracy to set the value is limited by the resolution of the screen.

Clicking SELECT somewhere on the slider bar, sets the value according to the relative position within the slider. Dragging SELECT in the slider bar, changes the value and reflects the new value during the drag. When the value is changed, the callback routine is called with the new value.

Basic attributes

Label
Text or image of the label.

LabelBold
Whether a text label must be displayed in bold font or not.

LabelWidth
The field width of a text label.

Value
The numerical value.

ValueBold
Whether the textual form of the value must be displayed in bold font or not.

ValueWidth
The field width of the value.

ValueMin
The minimal value.

ValueMax
The maximal value.

SliderWidth
Width of the slider bar in pixels.

Key
Identifier for the associated key. This key, if specified, will hold the value of the NumericSlider, and be updated when this changes.

Notify
The callback that must be called when the NumericSlider value is changed.

arg1: Item handle.
arg2: Event handle.
arg3: Field value.

Advanced attributes

Visible
Whether the object is visible or not.

VisibleRange
Visibility of the allowed range of the value.

VisibleValue
Visibility of the textual form of the value.

LabelFont
Font for a text label.

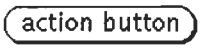
ValueFont
Font for the textual form of the value.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

ActionButton

An ActionButton is a control element through which some action can be triggered.



A button picture, holding the name of the action, serves as representation form.

Upon clicking SELECT in the button, the associated callback is activated.

Basic attributes

Label
Name of the action, or image representing it.

LabelBold
Whether the label must be displayed in bold font or not.

LabelWidth
Width of the button, in characters.

Notify
The callback that must be called when the button is pressed.

arg1: Button item handle.
arg2: Event handle.

Advanced attributes

Visible
Whether the object is visible or not.

LabelFont
Font for a text label.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

WindowButton

A WindowButton is a special ActionButton. It is meant to control the display of a window (or frame).

The button that represents it, holds a label, extended with an ellipsis (...).

When SELECT is clicked on the button, the associated Frame is made visible, and a callback is called (giving the possibility of initializing the Frame).

Basic attributes

Label
Text or image for the button.

LabelBold
Whether the label must be displayed in bold font or not.

LabelWidth
Width of the button, in characters.

Frame
Name of the Frame to be displayed.

Notify
The callback that must be called when the button is pressed.

arg1: Button item handle.
arg2: Event handle.
arg3: Frame name.

Advanced attributes

Visible
Whether the object is visible or not.

LabelFont
Font for a text label.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

MenuButton

A MenuButton is a variant of an ActionButton. It provides a means to pop up a Menu.

This object is represented with a button, holding a label that is extended with a right arrow (=>).

To pop up the Menu, the MENU mouse button must be held down over the button. At pressing the MENU button, a callback is called. This way, the Menu can be initialized if necessary. Then a selection from the Menu can be made. When SELECT is clicked on the MenuButton, the default selection is taken from the Menu, if the window system supports this.

Basic attributes

Label
Text or image for the button.

LabelBold
Whether the label must be displayed in bold font or not.

LabelWidth
Width of the button, in characters.

Menu
Name of the Menu to be popped-up.

Notify
The callback that must be called when the button is pressed.

arg1: Button item handle.
arg2: Event handle.
arg3: Menu name.

Advanced attributes

Visible
Whether the object is visible or not.

LabelFont
Font for a text label.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

SelectButton

With a SelectButton, it is possible to choose a value from a given set, and to pass this to an action or associate it with a key.

This type of object is represented by a label, and one of, or all the possible choices. Depending on the window system, an additional marker can identify the type of object.

Clicking SELECT on a choice, selects that choice. In case only a single choice is displayed, the behavior is window system dependent. Pressing the MENU button over a SelectButton, pops-up a ChoiceMenu holding all possible choices, from which one can be selected. When a choice is selected, the associated key is set to its corresponding value. This value is also passed to a callback.

Basic attributes

Label
Text or image for the button label.

LabelBold
Whether the label must be displayed in bold font or not.

LabelWidth
Width of the label, in characters.

Choices
List of choice texts and corresponding values.

Value
The selected choice value.

Key
Identifier for the associated key. This key, if specified, will hold the value corresponding to the selected choice.

Layout
Switch that determines the layout of the SelectButton. Abbreviated results in the display of only a single choice, Horizontal means a horizontal list with all choices, and Vertical a vertical list of all choices.

Notify
The callback that must be called when a choice is selected.

arg1: Button item handle.
arg2: Event handle.
arg3: Value corresponding to the last choice.

Advanced attributes

Visible
Whether the object is visible or not.

LabelFont
Font for a text label.

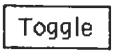
ChoicesFont
Font for the choices

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

ToggleButton

A ToggleButton implements a binary switch. It is meant to toggle a switch in the application.



It is represented by a label, that is displayed in inverse, or with a ticked box if the switch is set, and in plain font or with an empty check box when it is not set.

Clicking SELECT on the ToggleButton switches its state. The associated key is set to TRUE or FALSE to reflect the toggle's state. A callback is called with the new value if the toggle is changed.

Basic attributes

Label
Text or image for the button label.

LabelBold
Whether the label must be displayed in bold font or not.

LabelWidth
Width of the button, in characters.

Value
The state of the toggle.

Key
Identifier for the associated key. This key, if specified, will hold the value corresponding to the toggle's state.

Layout
Display mode of the button. Inverted mode displays the label in plain or inverted font. Marked mode provides a check box with a tick for setting the toggle.

Notify
The callback that must be called when the toggle is switched.

arg1: Button item handle.
arg2: Event handle.
arg3: Value corresponding to the last choice.

Advanced attributes

Visible
Whether the object is visible or not.

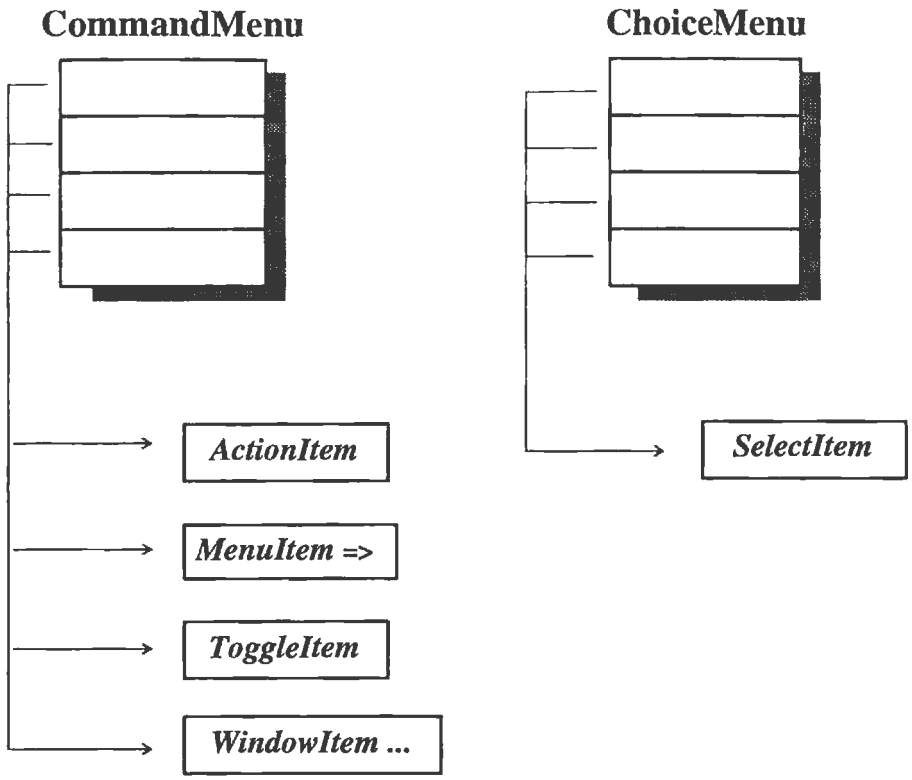
LabelFont
Font for a text label.

ClientData
Application specific data, associated with this object.

Foreground
Name of a color to be used as foreground.

1.3 Menus

There are two types of Menus: ***CommandMenu*** and ***ChoiceMenu***.
CommandMenus can have items of four different types, while ChoiceMenus only consist of Selectitems.



CommandMenu

A CommandMenu is a panel to hold some items that execute commands when selected.

The representation consists of a rectangular area, with a shadow behind it.

An item is selected by releasing the MENU button over it, or by clicking SELECT on it.

Attributes

None

ChoiceMenu

On a ChoiceMenu, a number of choices is displayed, from which one can be selected.

The representation of a ChoiceMenu is a rectangle with a shadow behind it.

Making a choice from the Menu is done by either releasing the MENU button or clicking the SELECT button on the choice. When a selection is made, a key, associated to that particular Menu, is set to the choice's value. And a callback is called with that value as parameter.

Attributes

Key
The key that must be set to the selected choice value.

Notify
The callback that must be called when a choice is selected.

arg1: Menu handle.
arg2: Menu item handle.
arg3: Value corresponding to the last choice.

1.4 Menu items

ActionItem

An ActionItem is a means to invoke an action by selection from a Menu. It is similar to an ActionButton.

This item type is represented by a label on the Menu.

Selecting the item, will activate a callback.

Attributes

Label
The item's label.

Notify
The callback that must be called when the item is selected.

arg1: Menu handle.
arg2: Menu item handle.

WindowItem

A WindowItem is the Menu equivalent of a WindowButton, and enables the displaying of a Frame from a Menu.

It is represented by a label, followed by an ellipsis (...).

Selecting the item, displays the associated Frame and activates a callback, that can initialize that Frame as necessary.

Attributes

Label
The item's label.

Frame
The associated Frame's name.

Notify
The callback that must be called when the item is selected.

arg1: Menu handle.
arg2: Menu item handle.
arg3: Frame name.

MenuItem

A MenuItem is similar to a MenuButton. On a Menu, it serves as a pull-right item.

It is visualized by a label and a right arrow (=>).

When the MENU button is moved to its right, the associated Menu is popped-up. At that moment, a callback is called that can adapt the Menu if necessary. A selection can then be made from that Menu. Clicking SELECT on a MenuItem, results in the selection of the default item from the associated Menu.

Attributes

Label
The item's label.

Menu
The associated Menu's name.

Notify
The callback that must be called when the Menu is popped up.

arg1: Menu handle.
arg2: Menu item handle.
arg3: Menu name.

ToggleItem

A ToggleItem is the Menu version of a ToggleButton. It implements a binary switch on a Menu.

Such item is displayed by a label that is in normal mode if the toggle is not set, and in inverse if it is set.

Selecting the item switches the toggle state and updates the associated key to reflect the new state. A callback is called with this new state as parameter.

Attributes

Label
The item's label

Key
The associated key.

Notify
The callback that must be called when the toggle is switched.

arg1: Menu handle.
arg2: Menu item handle.
arg3: Value corresponding to the last choice.

SelectItem

A SelectItem corresponds in some sense to a SelectButton. It is one of the choices that appear on a ChoiceMenu.

This item is simply represented by a label.

Selecting the item has an effect on the ChoiceMenu in which it occurs, but not on the item as such.

Attributes

Label
The item's label.

Value
The value corresponding to this choice.

1.5 Programmatic interface

The programmatic interface from the application program to the user interface, makes use of the run-time library of Carmen. That library provides routines for conversion of object handles, for the manipulation of objects and for object specific operators. This last category includes frame manipulation, drawing in a canvas, tty operations, writing on a Table and scrolling.

Multiple Projects

Several projects can be handled by the run-time Carmen library. Therefore projects must be identified with a unique identifier. It is determined from the 'Project' field in the Project Control window. The name entered there is used as identifier after replacing all non-alphanumerical characters with '_'. Notice that if a path is given in that field, it is used as part of the project identifier. To avoid this, the project should be loaded via the Project Browser window.

Since multiple projects may be loaded at the same time, one object name may represent several objects (one in each project). Predicates that are passed an object name will be performed on each of the objects with the given name, upon backtracking. To avoid this behavior and to force the operation to be performed on one object in a given project, use different names in your projects or use ui_name_to_handle/3 to retrieve the single object handle and pass that as argument to the predicate. Dynamically created objects (using ui_create or ui_drawable_create) belong to the same project as the parent.

Activation

The normal execution sequence of a project is first to initialize it with ui_initialize. Then it is activated either 'in background' with ui_activate or 'in foreground' with ui_main. When the project is not needed anymore it is terminated with ui_terminate. For a 'foreground' project, this implies that the ui_main predicate exits.

ui_initialize/0

ui_initialize

Window configurations of all loaded projects are created and the projects are initialized. Initializing an already initialized project has no effect.

ui_initialize/1

ui_initialize(_Project)

arg1 : ground: atom : Project name

Window configuration of loaded project *arg1* is created and the project is initialized. Initializing an already initialized project has no effect. Window configurations of all loaded projects are created and the projects are initialized. Initializing an already initialized project has no effect.

ui_initialize/1

ui_initialize(_Project)

arg1 : ground : atom : Project name

Window configuration of loaded project *arg1* is created and the project is initialized. Initializing an already initialized project has no effect.

ui_terminate/0

ui_terminate

All loaded projects are terminated and their window configurations are destroyed. Terminating a non-initialized project has no effect.

ui_terminate/1

ui_terminate(_Project)

arg1 : ground : atom : Project name

Project *arg1* is terminated and its window configuration is destroyed. Terminating a non-initialized project has no effect.

ui_activate/0

ui_activate

All initialized projects are activated in background. Activating an already activated project has no effect.

ui_activate/1

ui_activate(_Project)

arg1 : ground : atom : Project name

Project *arg1* is activated in background if it is initialized. Activating an already activated project has no effect. Fails if the project is not initialized.

ui_main/0

ui_main

All initialized projects are activated, one in foreground and the rest in background.

ui_main/1

ui_main(_Project)

arg1 : ground : atom : Project name

Project *arg1* is activated in foreground if it is initialized. Fails if the project is not initialized.

Object handle conversion

Objects are represented by a name in the user interface description, and also when they are referred to in notify handlers. Internally, during run-time, they are treated through a handle. Each notify handler has as first argument a handle representing the object or menu from which it is activated.

Inquiry predicates

ui_defined_project/1

ui_defined_project(_Project)

arg1 : any : atom : Project name

Succeeds once for each defined project, with *arg1* the project name. A project is defined once the .xv.pro file is loaded.

ui_defined_object/3

ui_defined_object(_Project,_Name,_Type)

arg1 : any : atom : Project name
arg2 : any : atom : Object name
arg3 : any : atom : Object type

Succeeds once for each defined object, with *arg1* the project, *arg2* the object name and *arg3* its type. An object is defined once the project has been initialized with ui_initialize.

ui_defined_object/2

ui_defined_object(_Name,_Type)

arg1 : any : atom : Object name
arg2 : any : atom : Object type

Same as ui_defined_object/3 with *arg1* discarded.

ui_server_object/1

ui_server_object(_Server)

arg1 : any : pointer : Server object handle

Returns in *arg1* the server object handle. Fails if no server connection has been set up. A server connection is set up when the first project is initialized.

ui_base_frame_object/2

ui_base_frame_object(_Project,_BaseFrame)

arg1 : ground : atom : Project name
arg2 : any : pointer : BaseFrame object handle

Returns in *arg2* the base frame object handle for project *arg1*. Fails if the project has not been initialized.

ui_base_frame_object/1

ui_base_frame_object(_BaseFrame)

arg1 : any : pointer : BaseFrame object handle

Same as ui_base_frame_object/2, succeeding once for each initialised project upon backtracking.

ui_name_to_handle/3

ui_name_to_handle(_Project,_Name,_Handle)

arg1 : any : atom : Project name
arg2 : ground : atom : Object name
arg3 : free : pointer : Object handle

Object name *arg2* in project *arg1* is converted to object handle *arg3*. If *arg1* is free, this predicate succeeds for all objects with given name in each initialized project and it is instantiated to the project name.

ui_name_to_handle/2

ui_name_to_handle(_Name,_Handle)

arg1 : ground : atom : Object name
arg2 : free : pointer : Object handle

Same as ui_name_to_handle/3 with *arg1* free.

ui_handle_to_name/3

ui_handle_to_name(_Handle,_Project,_Name)

arg1 : ground : pointer : Object handle
arg2 : any : atom : Project name
arg3 : free : atom : Object name

Object handle *arg1* is converted to project name *arg2* and object name *arg3*.

ui_handle_to_name/2

ui_handle_to_name(_Handle,_Name)

arg1 : ground : pointer : Object handle
arg2 : free : atom : Object name

Same as ui_handle_to_name/3 with *arg2* discarded.

Object manipulation

There is a set of routines for dynamically creating new objects, for changing their attributes and for retrieving their attribute values. The predicates for creating and manipulating a drawable are described in "*Drawable operations - Drawables'* on page 55

Object manipulation routines

ui_create/4

arg1: Object name or handle.
arg2: Parent name or handle.
arg3: Object type.
arg4: Object dimension.

A new object of type *arg3* is created, as sub object of *arg2*. If *arg1* is defined, its value is taken as the name of the object. If it is undefined, it is set to the object handle. And the object handle is further used as name for the object. The dimension of the object is *arg4*. This is an *ObjectRect/4* structure with as arguments the left and top edge position relative to the parent and the width and height.

ui_create/5

arg1: Object name or handle.
arg2: Parent name or handle.
arg3: Object type.
arg4: Object dimension.
arg5: Attribute name, value list.

A new object of type *arg3* is created, as sub object of *arg2*. If *arg1* is defined, its value is taken as the name of the object. If it is undefined, it is set to the object handle. And the object handle is further used as name for the object. The dimension of the object is *arg4*. This is an *ObjectRect/4* structure with as arguments the left and top edge position relative to the parent and the width and height. The attributes mentioned in *arg5* are initialized to the corresponding values.

ui_get/3

arg1: Object name or handle.
arg2: Attribute name.
arg3: Corresponding value.

The value of attribute *arg2* of the object with name or handle *arg1* is retrieved in *arg3*.

ui_set/3

arg1: Object name or handle.
arg2: Attribute name.
arg3: Attribute value.

The value of attribute *arg2* of the object with name or handle *arg1* is set to *arg3*.

ui_set/2

arg1: Object name or handle.
arg2: Attribute name, value list.

Arg2 is a list of attribute name, value pairs. The value of the attributes listed in *arg2* are set for the object with name or handle *arg1*.

Frame manipulation

Two routines are provided for opening and closing Frame objects.

ui_frame_open/1

arg1: Frame handle or name

The Frame arg1 is opened from its iconic or invisible state.

ui_frame_close/1

arg1: Frame handle or name

The Frame arg1 is closed to its iconic or invisible state.

Tty operations

It is possible to send input and output to a Tty from the application program.

Tty input

ui_tty_input/3

arg1: Tty handle or name.
arg2: Text string.
arg3: Text length.

Arg3 characters from text *arg2* are sent as input to Tty *arg1*.

Tty output

ui_tty_output/3

arg1: Tty handle or name.
arg2: Text string.
arg3: Text length.

Arg3 characters from text *arg2* are sent as output to Tty *arg1*.

Table operations

The library includes a number of routines for manipulating Display objects from the program.

Table clearing

ui_table_clear/1

arg1: Table handle or name

The Table *arg*1 is cleared.

ui_table_clear_column/2

arg1: Table handle or name
arg2: Column number

Column number *arg2* of Table *arg1* is cleared.

ui_table_clear_row/2

arg1: Table handle or name
arg2: Row number

Row number *arg2* of Table *arg1* is cleared.

ui_table_clear_area/5

arg1: Table handle or name
arg2: First column index
arg3: Last column index
arg4: First row index
arg5: Last row index

All cells in an area of Table *arg1* are cleared. The area extends from column *arg2* to column *arg3* and from *arg4* to row *arg5*.

Table filling

Fill Styles
An entry on a Table is filled in a certain style. This style is indicated with a value of TABLE_FILL_PLAIN, TABLE_FILL_BOLD, TABLE_FILL_DIMMED, TABLE_FILL_UNDERLINED or any logical expression using these values.

Adjust Modes
The Table entry is adjusted in one of the modes TABLE_ADJUST_LEFT, TABLE_ADJUST_CENTER or TABLE_ADJUST_RIGHT.

ui_table_fill/6

arg1: Table handle or name
arg2: Column number
arg3: Row number
arg4: Text
arg5: Fill style
arg6: Adjust mode

Text *arg4* is filled in on Table *arg*1 at column *arg2*, row *arg3*, in fill style *arg5* and adjusted in mode *arg6*.

ui_table_fill_column/7

arg1: Table handle or name
arg2: Column number
arg3: Row index
arg4: Number of rows
arg5: Fill style
arg6: Adjust mode

The list of text *arg5* is filled in, in Table *arg1*. It is filled in, in column *arg2*, starting from row *arg3*. *Arg4* rows are filled. The texts are displayed with fill style *arg6* and adjusted in mode *arg7*.
There must be at least *arg4* texts in the list *arg5*.

ui_table_fill_row/7

arg1: Table handle or name
arg2: Column index
arg3: Row index
arg4: Number of columns
arg5: Fill style
arg6: Adjust mode

The list of text *arg5* is filled in, in Table *arg1*. It is filled in, in row *arg3*, starting from column *arg2*. *Arg4* columns are filled. The texts are displayed with fill style *arg6* and adjusted in mode *arg7*.
There must be at least arg4 texts in the list *arg5*.

ui_table_fill_area/8

arg1: Table handle or name
arg2: First column index
arg3: Last column index
arg4: First row index.
arg5: Last row index
arg6: list of texts
arg7: Fill style.
arg8: Adjust mode

The list of texts *arg5* is filled in, in an area of table *arg1*. The area extends from column *arg2* to column *arg3*, and from row *arg4* to row *arg5*. The texts are displayed with fill style *arg6* and adjusted in mode *arg7*.
There must be enough texts in the list *arg5*. These must be ordered in column first order: all rows of the first column are filled with the first texts in the list. The following texts are used for the next columns.

Table retrieval

ui_table_get_cell/7

arg1: Table handle or name.
arg2: Column index
arg3: Row index
arg4: Text
arg5: Fill style
arg6: Adjust mode
arg7: Selection state (TRUE if selected, FALSE if not selected)

The cell at column position *arg2* and row position *arg3* in table *arg1* is retrieved. Arg4 is instantiated to its text value, *arg5* to its fill style, *arg6* to its adjust mode, and *arg7* to its selection state.

Selection manipulation

ui_table_deselect_all/1

arg1: Table handle or name

All cells of table *arg1* are deselected (without notification).

ui_table_set_selection/4

arg1: Table handle or name.
arg2: Column index.
arg3: Row index
arg4: Selection state (TRUE if selected, FALSE if not selected)

The selection state of the cell at column position *arg2* and row position *arg3* in table *arg1* is changed. It is set to the value *arg4*. No notification is performed for this selection change.

If the table has exclusive selection behavior, all selected cells are first deselected (without notification). In the other case, when multiple selection are allowed, the already selected cells are not affected.

Drawable operations

The next section provides a set of routines to draw on a drawable. The drawable coordinates are integers and range from 0,0 in the upper left corner, to the right and down. A drawable can in general be the drawable of a Canvas or an Image drawable.

Drawing mode

Each draw operation can be performed in different modes. This mode is indicated using an attribute list, which is a list consisting of attribute/value definitions, chosen from the table below. They are represented as Prolog terms with the attribute name as functor and the value as argument. The order in which the definitions appear in the list is irrelevant, except if the same attribute is set several times, in which case the last definition is considered. Not all attributes apply to each of the drawing operations. The specifications of the drawing predicates indicate which of the attributes are relevant.

Attribute	Semantics
function	Logical drawing function
plane_mask	Mask of planes
foreground	Foreground pixel color
background	Background pixel color
line_width	Line width (0 for thin line)
line_style	Style of line
cap_style	Style for ends of lines
join_style	Style for joining lines
fill_style	Style for filling areas
fill_rule	Rule for determining in/out of polygons
arc_mode	Mode for drawing arcs
tile	Image for tiling operations
stipple	Image for stippling operations
ts_x_origin	Offset for tile or stipple
ts_y_origin	Offset for tile or stipple
font	Font for text drawing
graphics_exposures	Whether expose events must be generated
clip_x_origin	Origin for clipping
clip_y_origin	Origin for clipping
dash_offset	Offset into dash to start line with
dashes	Length of dashes and gaps.

Possible values of these attributes are as follows.

function
Takes one of the enumerated values in the list below. The semantics of each operator is explained as a logical combination result of the source image (that is drawn) and the destination (on which it is drawn).
Default : Copy

Operator	Semantics
Clear	0
Set	1
Src	src
Copy	src
Dst	dst
NoOp	dst
Reverse	not src
CopyInverted	not src
Invert	not dst
And	src and dst
AndReverse	src and (not dst)
Nand	(not src) or (not dst)
Or	src or dst
OrReverse	src or (not dst)
OrInverted	(not src) or dst
Nor	(not src) and (not dst)
Xor	src xor dst
Equiv	(not src) xor dst
Overlay	(not src) and dst
AndInverted	(not src) and dst

plane_mask
Integer.
Default : All 1's

foreground
Pixel color indication.
Default : Black

background
Pixel color indication.
Default : White

line_width
Positive integer.
Default : 0

line_style
One of LineSolid, LineOnOffDash, LineDoubleDash.
Default : LineSolid

cap_style
One of CapNotLast, CapButt, CapRound, CapProjecting.
Default : CapButt

join_style
One of JoinMiter, JoinRound, JoinBevel.
Default : JoinMiter

fill_style
One of FillSolid, FillTiled, FillStippled, FileOpaqueStippled.
Default : FillSolid

fill_rule
One of EvenOddRule, WindingRule.
Default : EvenOddRule

arc_mode
One of ArcChord, ArcPieSlice.
Default : ArcPieSlice

tile
Drawable of same depth as drawable.
Default : Image filled with foreground

stipple
Drawable of depth 1.
Default : Image filled with 1's

ts_x_origin
Integer.
Default : 0

ts_y_origin
Integer.
Default : 0

font
Font specification.
Default : Default server font

graphics_exposures
Boolean, one of True, False.
Default : True

clip_x_origin
Integer.
Default : 0

clip_y_origin
Integer.
Default : 0

dash_offset
Integer.
Default : 0

dashes
Positive integer.
Default : 4

Arguments

Arguments are specified using the following forms:

Point
Represented as a tuple of X and Y coordinate (X,Y). Coordinates may be integer or real but will always be converted to integer by rounding.

Point list
Represented as a list of points.

Size
Sizes are represented like points as a tuple (Width,Height). Dimensions may be integer or real but will always be converted to integer by rounding.

Drawable
Represented by its name or handle.

Color
Foreground and background must be specified as named colors. Both server defined names and device-independent Xcms specifications are allowed.

Drawables

ui_drawable_create/5

ui_drawable_create(_Draw,_Parent,_Width,_Height,_Depth)

arg1 : any : atom or pointer : Resulting drawable
arg2 : ground : atom or pointer : Parent drawable
arg3 : ground : integer : Drawable width
arg4 : ground : integer : Drawable height
arg5 : any : integer : Drawable depth
A new off-screen drawable is created.

If *arg1* is defined, its value is taken as name of the drawable. Otherwise, *arg1* is instantiated to the handle of the created drawable and this is further used as its name. The parent drawable *arg2* is used to determine which screen the drawable is meant for. Dimensions of the drawable are indicated with *arg3*, *arg4* and *arg5*. If the depth *arg5* is undefined, the same depth is taken as for the parent drawable and *arg5* is instantiated to it.

ui_drawable_create/4

ui_drawable_create(_Draw,_Width,_Height,_Depth)

arg1 : any : atom or pointer : Resulting drawable
arg2 : ground : integer : Drawable width
arg3 : ground : integer : Drawable height
arg4 : any : integer : Drawable depth

Same as ui_drawable_create/5 with the default root window as parent drawable.

ui_drawable_destroy/1

ui_drawable_destroy(_Draw)

arg1 : ground : atom or pointer : Drawable

Drawable *arg1* is destroyed.

ui_drawable_set_defaults/2

ui_drawable_set_defaults(_Draw,_Attrs)

arg1 : ground : atom or pointer : Destination drawable
arg2 : ground : list : Attribute list

The default attributes for drawable *arg1* are set to *arg2*.

Areas

ui_drawable_clear/1

ui_drawable_clear(_Draw)

arg1 : ground : atom or pointer : Drawable

The whole drawable *arg1* is cleared.

ui_drawable_clear_area/3

ui_drawable_clear_area(_Draw,_Origin,_Size)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : Upper left corner point
arg3 : ground : Size : Area size

The area on drawable *arg1* starting at point *arg2*, extending over size *arg3* is cleared. If the width or height in the size indication is 0, it extends to the right or bottom border of the drawable.

ui_drawable_copy_area/6

ui_drawable_copy_area(_Draw,_OrgDst,_Size,_Src,_OrgSrc,_Attrs)

arg1 : ground : atom or pointer : Destination drawable
arg2 : ground : Point : Upper left corner point in destination drawable
arg3 : ground : Size : Area size
arg4 : ground : atom or pointer : Source drawable
arg5 : ground : Point : Upper left corner point in source drawable
arg6 : ground : list : Attribute list

The area from source *arg4* starting at position *arg5* is copied onto destination drawable *arg1* starting at position *arg2* and extending over size *arg3*.
Attributes as specified in *arg6*. Attributes considered :

function
plane_mask, graphics_exposures
clip_x_origin, clip_y_origin

Geometric figures

ui_drawable_draw_point/3

ui_drawable_draw_point(_Draw,_Point,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : Point
arg3 : ground : list : Attribute list

A point is drawn in drawable *arg1* at position *arg2*.
Attributes as specified in *arg3*. Attributes considered:

function, foreground
plane_mask
clip_x_origin, clip_y_origin

ui_drawable_draw_points/4

ui_drawable_draw_points(_Draw,_PointList,_CoordMode,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : list of Point : Point list
arg3 : ground : atom : Coordinate mode
('CoordModeOrigin', 'CoordModePrevious')
arg4 : ground : list : Attribute list

A list of points is drawn in drawable *arg1* at positions arg2. Coordinates in *arg2* are interpreted as indicated in *arg3*: relative to the drawable origin, or relative to the previous point in the list.
Attributes as specified in *arg4*. Attributes considered:

function, foreground
plane_mask
clip_x_origin, clip_y_origin

ui_drawable_draw_line/4

ui_drawable_draw_line(_Draw,_Point1,_Point2,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : First point
arg3 : ground : Point : Second point
arg4 : ground : list : Attribute list

A line is drawn in drawable *arg1* from point *arg2* to point *arg3*. Attributes as specified in *arg4*. Attributes considered:

function, foreground, background
plane_mask
line_width, line_style, cap_style, fill_style
tile, stipple, ts_x_origin, ts_y_origin
dash_offset, dashes
clip_x_origin, clip_y_origin

ui_drawable_draw_lines/4

ui_drawable_draw_lines(_Draw,_PointList,_CoordMode,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : list of Point : List of points
arg3 : ground : atom : Coordinate mode
('CoordModeOrigin','CoordModePrevious')
arg4 : ground : list : Attribute list

A sequence of lines is drawn in drawable *arg1* between points *arg2*. If the first and last point of *arg2* coincide, they will be joined correctly. Coordinates in *arg2* are interpreted as indicated in *arg3*: relative to the drawable origin, or relative to the previous point in the list.
Attributes as specified in *arg4*. Attributes considered:

function, foreground, background
plane_mask
line_width, line_style, cap_style, join_style, fill_style
tile, stipple, ts_x_origin, ts_y_origin
dash_offset, dashes
clip_x_origin, clip_y_origin

ui_drawable_fill_polygon/4

ui_drawable_fill_polygon(_Draw,_PointList,_CoordMode,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : list of Point : List of points
arg3 : ground : atom : Coordinate mode
('CoordModeOrigin', 'CoordModePrevious')
arg4 : ground : list : Attribute list

A polygon is filled in drawable *arg1*, bounded by the sequence of lines between points *arg2*. If the last point in *arg2* does not coincide with the first one, the polygon is closed automatically. Coordinates in *arg2* are interpreted as indicated in *arg3*: relative to the drawable origin, or relative to the previous point in the list.
Attributes as specified in *arg4*. Attributes considered:

function, foreground, background
plane_mask
fill_style, fill_rule
tile, stipple, ts_x_origin, ts_y_origin
clip_x_origin, clip_y_origin

ui_drawable_draw_rectangle/4

ui_drawable_draw_rectangle(_Draw,_Origin,_Size,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : Upper left corner point
arg3 : ground : Size : Rectangle size
arg4 : ground : list : Attribute list

A rectangle is drawn in drawable *arg1*, with upper left corner at point *arg2* and a size of *arg3*. Attributes as specified in *arg4*. Attributes considered:

function, foreground, background
plane_mask
line_width, line_style, join_style, fill_style
tile, stipple, ts_x_origin, ts_y_origin
dash_offset, dashes
clip_x_origin, clip_y_origin

ui_drawable_fill_rectangle/4

ui_drawable_fill_rectangle(_Draw,_Origin,_Size,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : Upper left corner point
arg3 : ground : Size : Rectangle size
arg4 : ground : list : Attribute list

A rectangle area is filled in drawable *arg1*, with upper left corner at point *arg2* and a size of *arg3*. Attributes as specified in *arg4*. Attributes considered:

function, foreground, background
plane_mask
fill_style
tile, stipple, ts_x_origin, ts_y_origin
clip_x_origin, clip_y_origin

ui_drawable_draw_arc/6

ui_drawable_draw_arc(_Draw,_Origin,_Size,_OrgAngle,_ExtAngle,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : Upper left corner point
arg3 : ground : Size : Bounding box size
arg4 : ground : Angle : Origin angle
arg5 : ground : Angle : Extent angle
arg6 : ground : list : Attribute list

An elliptical arc is drawn in drawable arg1, in a bounding box with upper left corner at point *arg2* and a size of *arg3*. The arc starts at angle *arg4*, counted in degrees from the 3 o'clock position counter clockwise. It extends over *arg5* degrees. Attributes as specified in *arg6*. Attributes considered:

function, foreground, background
plane_mask
line_width, line_style, cap_style, join_style, fill_style
tile, stipple, ts_x_origin, ts_y_origin
dash_offset, dashes
clip_x_origin, clip_y_origin

ui_drawable_fill_arc/6

ui_drawable_fill_arc(_Draw,_Origin,_Size,_OrgAngle,_ExtAngle,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : Upper left corner point
arg3 : ground : Size : Bounding box size
arg4 : ground : Angle : Origin angle
arg5 : ground : Angle : Extent angle
arg6 : ground : list : Attribute list

An elliptical arc is filled in drawable *arg1*, in a bounding box with upper left corner at point *arg2* and a size of *arg3*. The arc starts at angle *arg4*, counted in degrees from the 3 o'clock position counter clockwise. It extends over *arg5* degrees. Attributes as specified in *arg6*. Attributes considered:

function, foreground, background
plane_mask
fill_style, arc_mode
tile, stipple, ts_x_origin, ts_y_origin
clip_x_origin, clip_y_origin

Text

ui_drawable_text_extent/4

ui_drawable_text_extent(_Draw,_Text,_Extent,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : atom : Text string
arg3 : free : Size : Text extent
arg4 : ground : list : Attribute list

Calculates the extent of a text in drawable *arg1*, with text string *arg2*. The resulting extent is unified with *arg3*, being a size type structure.
Attributes as specified in *arg4*. Attributes considered:

font

ui_drawable_draw_text/4

ui_drawable_draw_text(_Draw,_Origin,_Text,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : Upper left corner point
arg3 : ground : atom : Text string
arg4 : ground : list : Attribute list

A text is drawn in drawable *arg1* with its baseline origin at point *arg2* and with text string *arg3*. Attributes as specified in *arg4*. Attributes considered:

function, foreground, background
plane_mask
fill_style, font
tile, stipple, ts_x_origin, ts_y_origin
clip_x_origin, clip_y_origin

ui_drawable_fill_text/4

ui_drawable_fill_text(_Draw,_Origin,_Text,_Attrs)

arg1 : ground : atom or pointer : Drawable
arg2 : ground : Point : Upper left corner point
arg3 : ground : atom : Text string
arg4 : ground : list : Attribute list

A text is drawn in drawable *arg1* with its baseline origin at point *arg2* and with text string *arg3*. Its bounding box is filled as well. Attributes as specified in *arg4*. Attributes considered:

foreground, background
plane_mask
font
clip_x_origin, clip_y_origin

The function attribute used is Copy and the fill_style is FillSolid.

Canvas drawing

Important note:

The following section is only provided for compatibility with the earlier Carmen versions. This functionality will no longer be supported in the next release. Use the predicates of *'Drawable operations'* on page 52.

The library contains a set of routines to draw on a Canvas drawable. The drawable coordinates are integers and range from 0,0 in the upper left corner, to the right and down. A drawable can in general be the drawable of a Canvas or an Image drawable.

Combination operators

Certain drawing predicates make a combination of two images: a source and a destination. The destination is then replaced with the combination of the old destination and a source image. This combination is made, using an operator.
The following operators are defined:

Operator	Semantics
Src	src
Invert	not(dst)
Or	src or dst
And	src and dst
Xor	src xor dst

Areas

ui_canvas_clear_area/5

arg1: Drawable handle or name
arg2: X coordinate of left side
arg3: Y coordinate of top side
arg4: Width of area
arg5: Height of area

An area on drawable *arg1* is cleared. It extends from position *arg2*, *arg3* over a width *arg4* and a height *arg5*. If the width or height are 0, the area extends to the right, respectively the bottom border of the drawable.

ui_canvas_copy_area/9

arg1: Destination drawable handle or name
arg2: X coordinate in destination
arg3: Y coordinate in destination
arg4: Width of area
arg5: Height of area
arg6: Source drawable handle or name
arg7: X coordinate in source
arg8: Y coordinate in source
arg9: Operator for combining source and destination

An area of width *arg4* and height *arg5* is copied from source drawable arg6 or destination drawable *arg1*. It is taken from position *arg7*, *arg8* in the source. The position in the destination where the area is copied, starts at *arg2*, *arg3*. The source and destination are combined to form the new destination value. Operator *arg9* is used for this combination.

ui_canvas_clear/1

arg1: Drawable handle or name

The drawable *arg1* is cleared.

Lines and figures

ui_canvas_draw_line/7

arg1: Drawable handle or name
arg2: X coordinate of first point
arg3: Y coordinate of first point
arg4: X coordinate of second point
arg5: Y coordinate of second point.
arg6: Line width
arg7: Operator

A line is drawn on the drawable arg1. The line is drawn between points *arg2*, *arg3* and *arg4*, *arg5*. *Arg6* is the line width. If this 0, a thin line is drawn. The operator used to draw the line is *arg7*. It is used to combine the destination with a foreground color line.

ui_canvas_draw_polyline/6

arg1: Drawable handle or name
arg2: X coordinates of points
arg3: Y coordinates of points
arg4: Line width
arg5: Operator
arg6: Coordinate mode

A polyline is drawn on drawable *arg1*. *Arg2* and *arg3* are the X and Y coordinates of the points of the polyline. These can be absolute coordinates or relative coordinates. Relative coordinates are interpreted as a displacement from the previous point in the list. *Arg4* is the line width. If this is 0, a thin line is drawn. The operator used to draw the line is *arg5*. It is used to combine the destination with a foreground color line. The coordinate mode is given *arg6*. This must be one CoordModeAbsolute, CoordModePrevious.

ui_canvas_fill_polygon/6

arg1: Drawable handle or name
arg2: X coordinates of points
arg3: Y coordinates of points
arg4: Source image drawable handle or name
arg5: Operator
arg6: Coordinate mode

A polygon is filled on drawable *arg1*. *Arg2* and *arg3* are the X and Y coordinates if the points of the polygon. These can be absolute coordinates or relative coordinates. Relative coordinates are interpreted as a displacement from the previous point in the list. *Arg4* is the source image that is used to fill the polygon. The operator used to combine the destination with the source image, is *arg5*. The coordinate mode is given as *arg6*. This must be one of CoordModeOrigin, CoordModePevious.

ui_canvas_draw_rectangle/7

arg1: Drawable handle or name
arg2: X coordinate of upper left corner
arg3: Y coordinate of upper left corner
arg4: Width of rectangle
arg5: Height of rectangle
arg6: Line width
arg7: Operator

A rectangle is drawn on drawable *arg1*. *Arg2* and *arg3* are the X and Y coordinates of the upper left corner of the rectangle. *Arg4* and *arg5* are its width and height. The line width is *arg6*. It is 0, a thin line is drawn. The operator used to draw the rectangle lines is *arg7*. It is used to combine the destination with a foreground color line.

ui_canvas_fill_rectangle/7

arg1: Drawable handle or name
arg2: X coordinate of upper left corner
arg3: Y coordinate of upper left corner
arg4: Width of rectangle
arg5: Height of rectangle
arg6: Source image drawable handle or name
arg7: Operator

A rectangle is filled on drawable *arg1*. *Arg2* and *arg3* are the X and Y coordinates of the upper left corner of the rectangle. *Arg4* and *arg5* are its width and height. *Arg6* is the source image that is used for filling. The operator used to combine the destination with the source is *arg7*.

ui_canvas_text_width/5

arg1: Drawable handle or name
arg2: Text string
arg3: Font handle
arg4: Width of bounding box
arg5: Height of bounding box

The bounding box of a text is calculated. *Arg1* is the drawable on which the text would be drawn. The text is given in *arg2*. *Arg3* is the font for drawing the text. The width and height of the bounding box are returned in *arg4* and *arg5*.

ui_canvas_draw_text/7

arg1: Drawable handle or name
arg2: X coordinate of first character base line
arg3: Y coordinate of first character base line
arg4: Text string
arg5: Font handle
arg6: Source image drawable handle or name
arg7: Operator

A text is drawn on drawable *arg1*. The baseline position of the first character is at *arg2*, *arg3*. *Arg4* is the text string. *Arg5* is the font for drawing the text. The text is drawn by combining the destination with the source image *arg6*, with *arg7* as operator.

Carmen

Chapter 2
Tutorial

2.1 Carmen's principles

A window interface consists of **Frames** and **Menus**.

Frames

A **Frame** is a window that is either displayed or invisible.
In a window interface, Frames are structured in hierachies. Each hierarchy is composed of a **base frame** and possibly some **subframes**. The difference between them lies in their invisibility behavior. A base frame is represented by an **icon** in closed state, while a closed subframe is completely invisible.

A Frame can include some **window objects**:
- ***Canvas*** - A subwindow into which programs can draw.
- ***Textsw*** - A subwindow containing text.
- ***Tty*** - A terminal emulation in which commands can be run.
- ***Term*** - A Tty with scrolling facility.
- ***Table*** - A spreadsheet skeleton.
- ***Panel*** - A subwindow containing **control items**.

Control items are components that facilitate some particular types of interaction between the user and the application. Control items include: ***Message***, ***MessageField***, ***TextField***, ***NumericField***, ***NumericSlider***, ***ActionButton***, ***WindowButton***, ***MenuButton***, ***ToggleButton*** and ***SelectButton***.

Menus

A **Menu** is a specialized window through which the user makes choices and issues commands. Menus appear when the user presses the right mouse button and disappear when they have served their purpose.
Menus are composed of **menu items**: ***ActionItem***, ***WindowItem***, ***MenuItem***, ***ToggleItem*** and ***SelectItem***. They can invoke a specific procedure, display a Frame or pop-up another Menu.

Frames - menus relations

As stated before, ***Frames*** can include some ***Panel*** subwindows containing control items and, in particular, ***MenuButtons***. MenuButtons are buttons that activate a specific Menu, establishing a link between Frames and Menus.

On the other hand, Frames can be displayed by choosing a ***WindowItem*** in a menu.
WindowItems are connected to a particular frame which is displayed as soon as the item is chosen in the menu.

MenuButton and ***WindowItem*** are the two key-objects in the relation between Frames and Menus.

Frames - frames relations

Hierarchical dependencies between ***Frames*** provided by ***WindowButtons***. Defined in a ***Panel***, a WindowButton is connected to a definite Frame which is displayed as soon as the button is selected.

Callbacks and keys

The connection between the interface and the underlying application is established through the use of **callbacks** and **keys**.

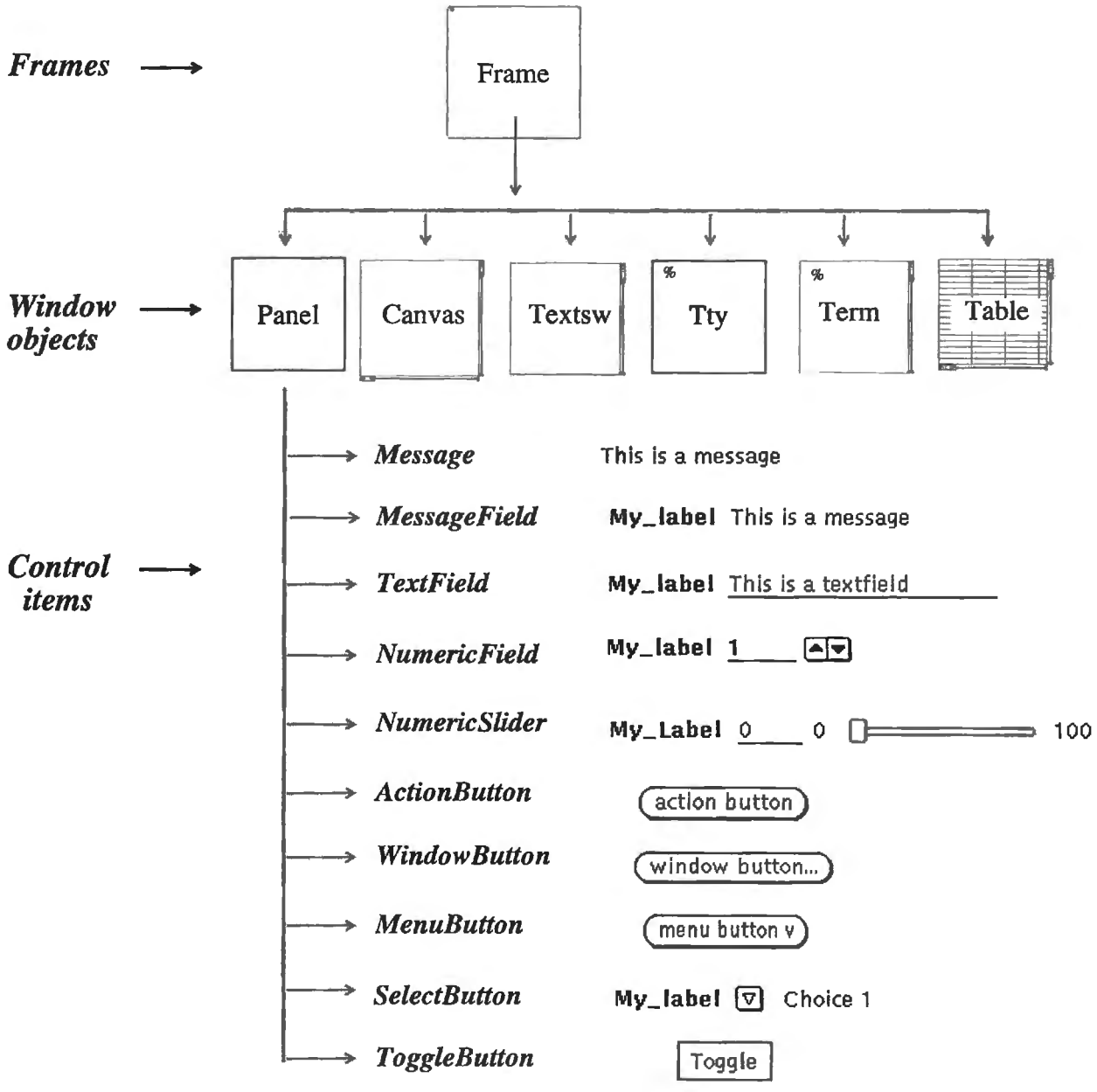
A **callback** is a routine from the application that is called when a special event, triggered by the user, occurs in the interface (e.g. clicking a button).

A **key** is a data structure in the application that is set by the interface, whenever some event occurs (e.g. making a choice or toggling a switch).

Callbacks and ***keys*** are described in the Reference Guide.

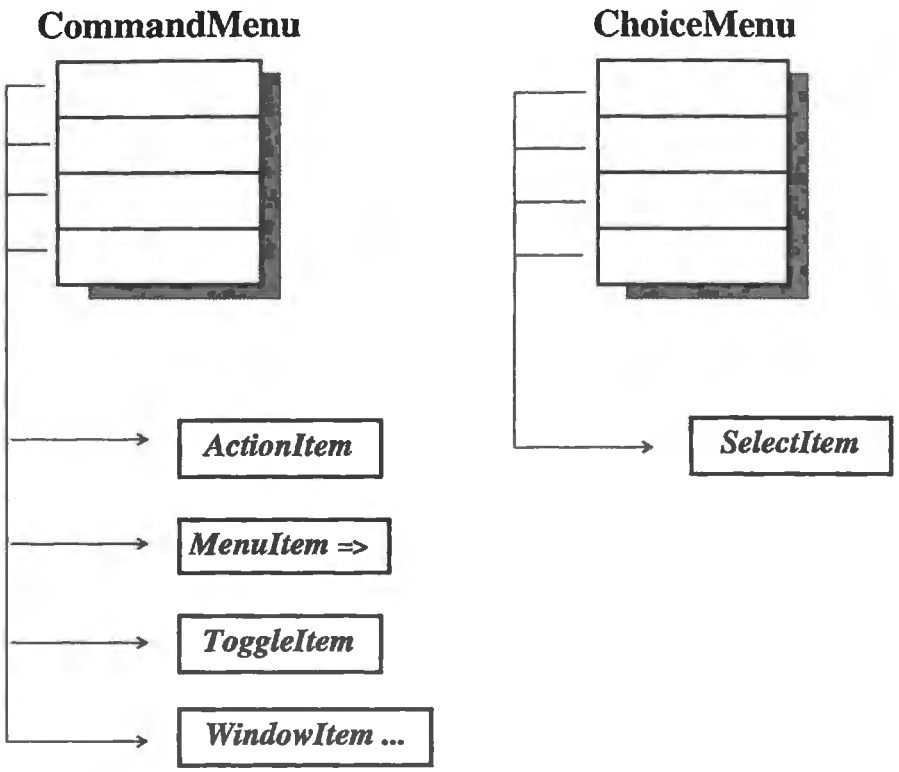
Frames, window objects and control items

One can distinguish three classes of objects: ***Frames***, ***Window Objects*** and ***Control Items***. Frames (base frame and subframe) can contain any of the window objects. Control items are restricted to Panel objects.



Menus and Menu Items

There are two types of Menus: **CommandMenu** and **ChoiceMenu**.
CommandMenus can have items of four different types, while ChoiceMenus only consist of SelectItems.



Menus can be connected to MenuButtons. The pop-up menu appears when the MenuButton is clicked.
Menus can also be connected to Panel, Textsw, Tty, Term and MenuItems.
In this case, the pop-up menu appears anywhere in the object to which the menu is linked, as soon as the right mouse button is clicked.

2.2 Running Carmen

Please refer to the ***ProLog*** installation guide in order to install Carmen. This will create a prelinked version of Carmen.

After installation Carmen can be run with:

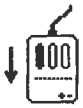
```
prompt % Carmen
```

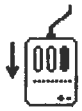
On SunOS 5 systems Carmen can be run without creating the prelinked version. But since this means that linking is going to be performed at each startup we advise to install the prelinked version of Carmen.

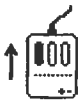
Conventions

Click the left button of the mouse: 

Click the right button of the mouse: 

Press the left button of the mouse and hold it down: 

Press the right button of the mouse and hold it down: 

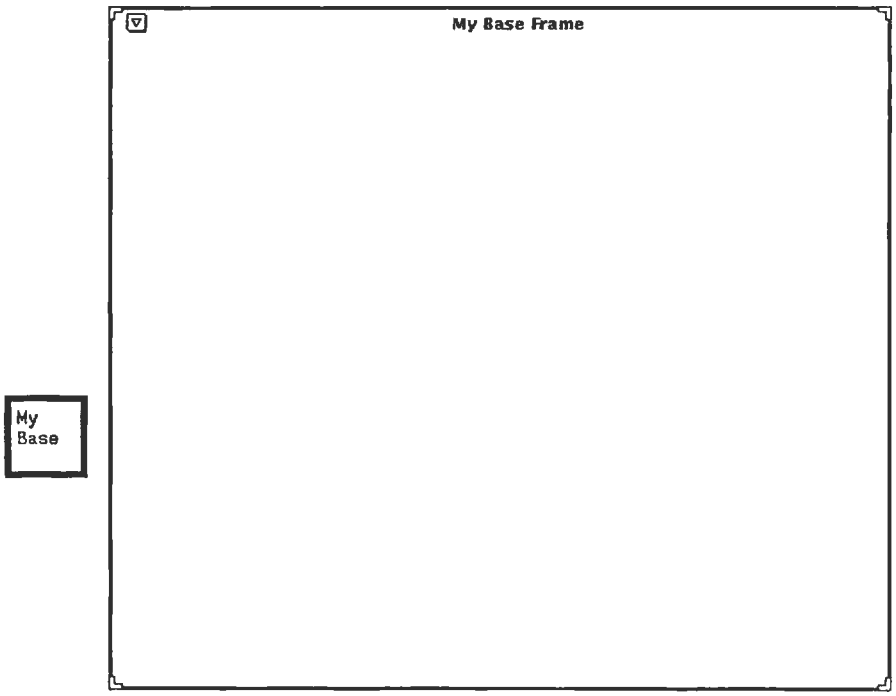
Release the left button of the mouse: 

Release the right button of the mouse: 

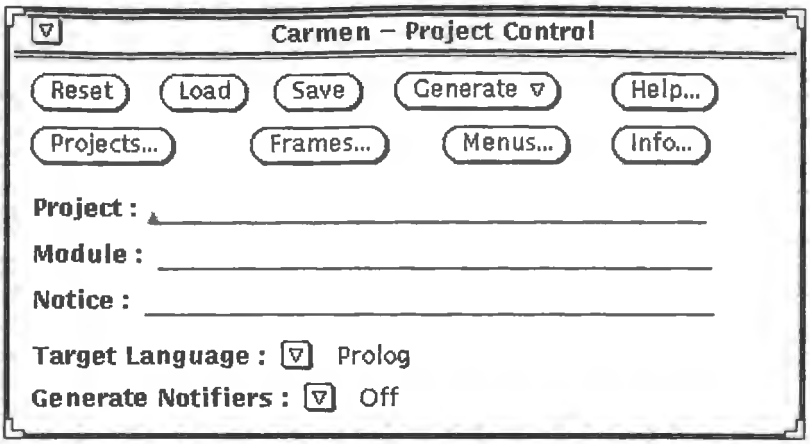
2.3 Carmen by example

Example one

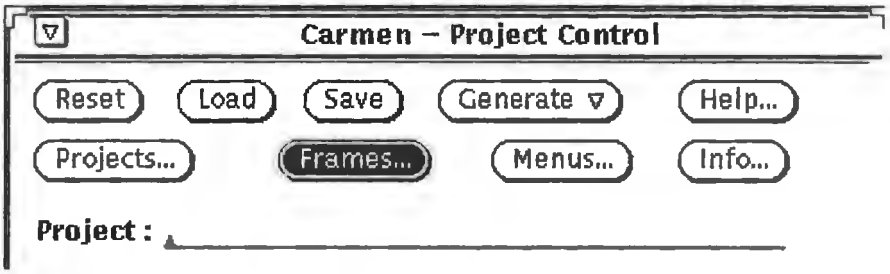
We are going to develop a simple user interface as shown below:



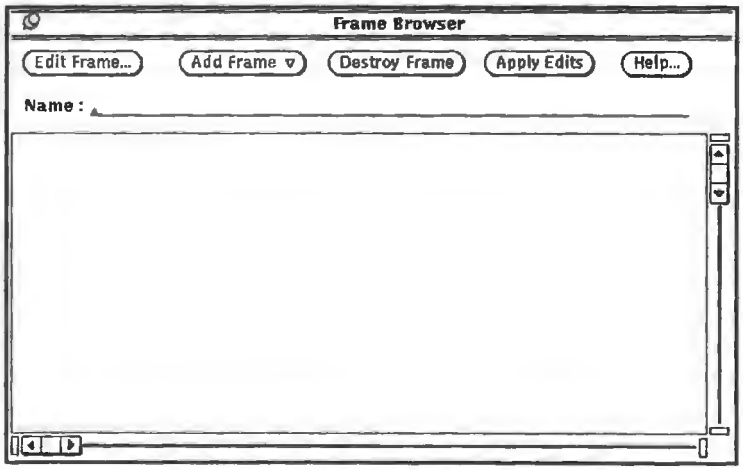
Designing and running this little example will help you to discover Carmen's mechanical details.



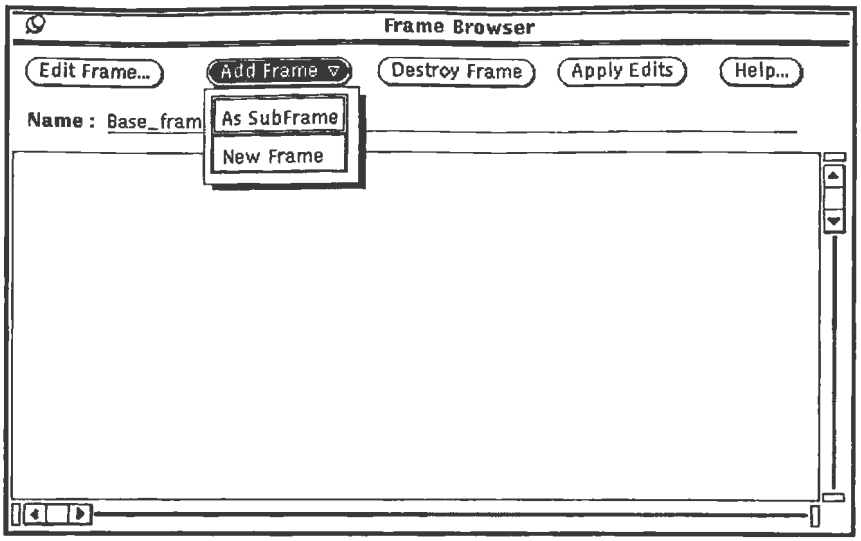
This frame is used to reset, load and save a user window interface (also called project) and to generate the equivalent Prolog code.
It is also from this window that it is possible to open the Frame and Menu Browsers.
Open the Frame Browser.

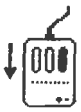


The **Frame Browser** displays the list of defined Frames for the current project.
In this case, we are designing a new project, thus there are no defined frames yet.

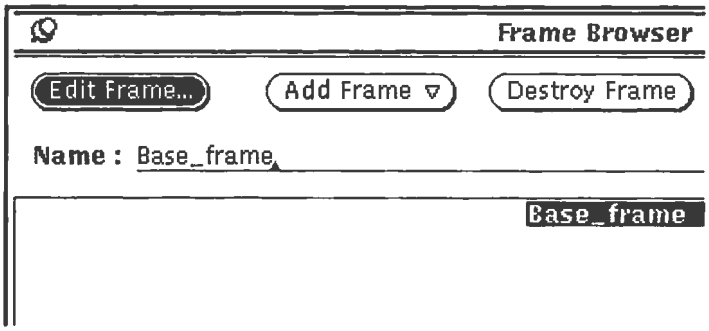


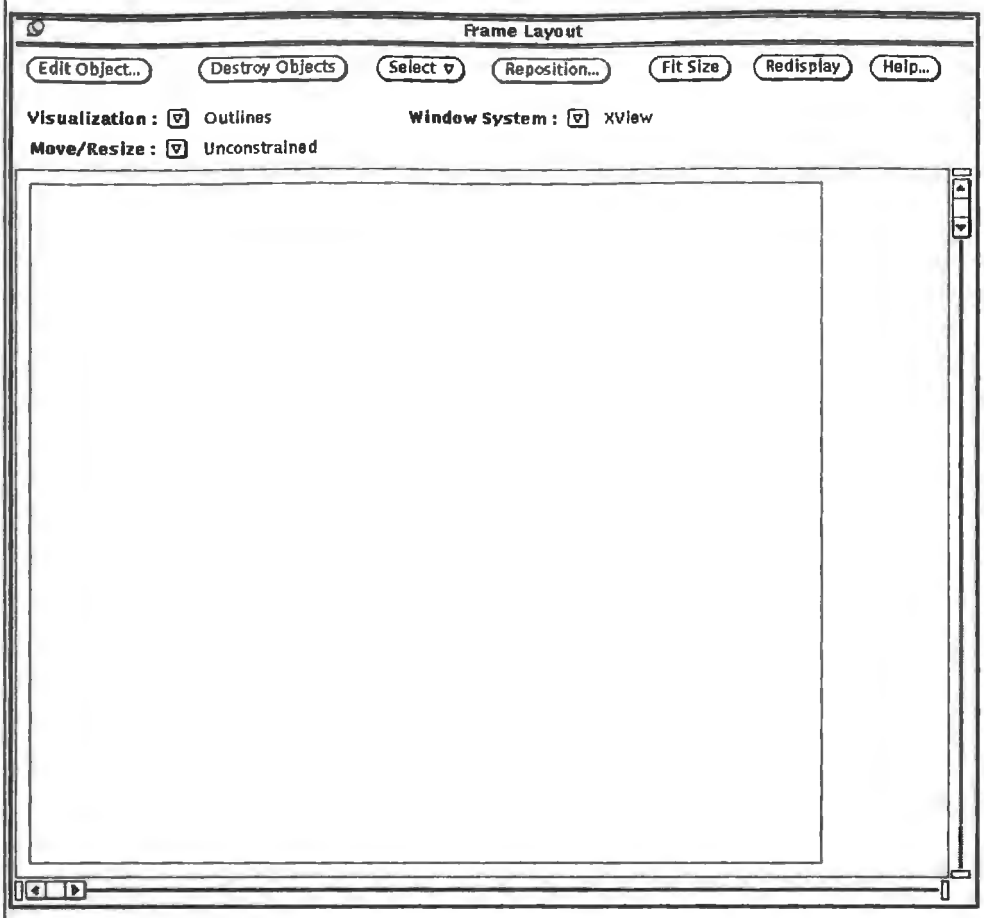
Type in *Base_frame*, the name of the example frame and add it as a new frame:



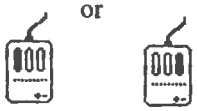
 ... the **Add Frame** MenuButton and choose the **New Frame** item... 

Now, the name of the new frame appears in reverse, which means that it is selected. You can now edit this new frame. Open the Frame Editor...





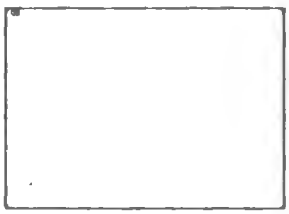
Depending on the status of the **Visualization** SelectButton, objects are displayed either in a raw form **(Outlines)** or in a wysiwyg way **(Realistic)**. This visualization mode can simply be changed by or and selecting one of the modes.



... an object selects it. When selected, little squares appear in reverse, delimiting the borders of the object. Here is a selected frame:

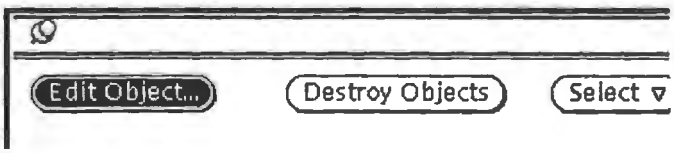
In Outlines mode

In Realistic mode

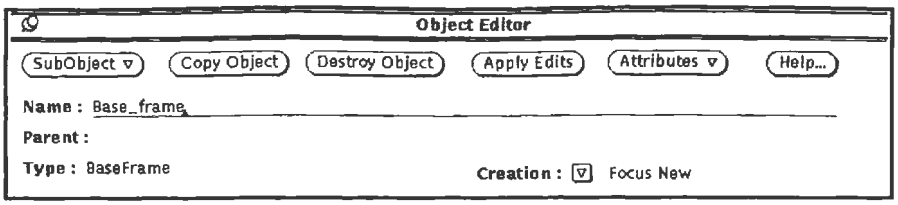


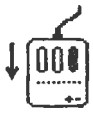
Once selected, an object can be edited.

Select the Frame and open the Object Editor:

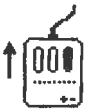


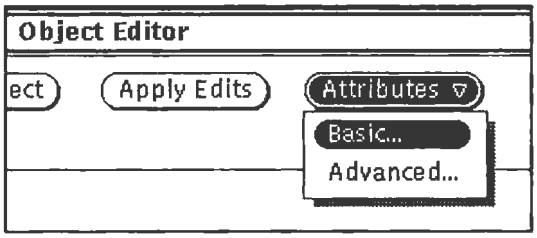
The **Object Editor** pops up. In this frame, it is possible to rename or destroy an object, to extend it with another subobject, and to specify Basic and Advanced attributes for this object.



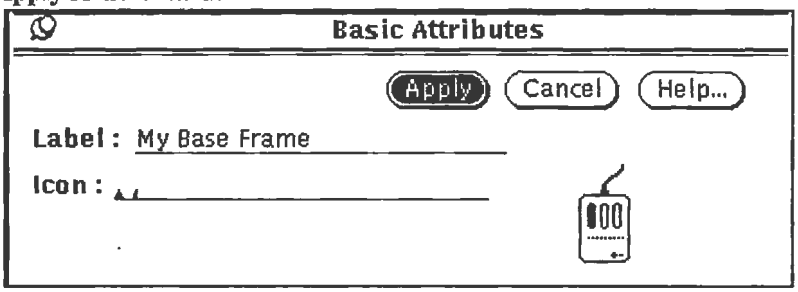


... on the **Attributes** MenuButton and select the **Basic** WindowItem.

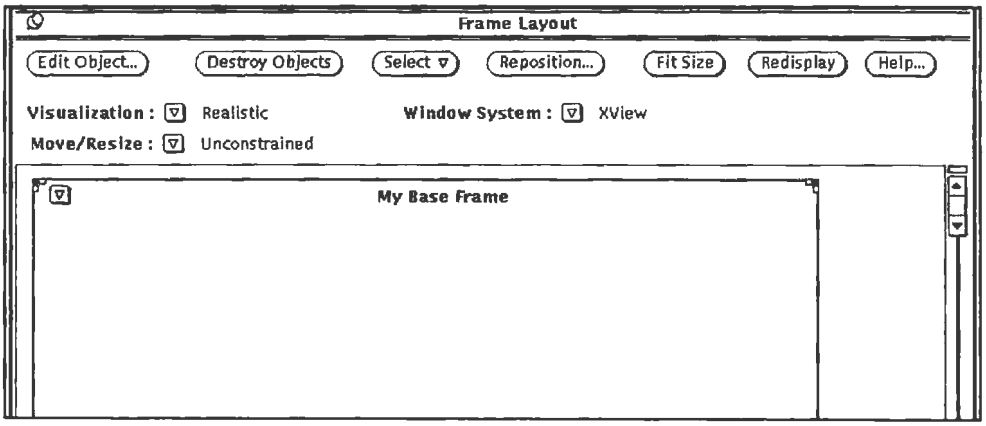




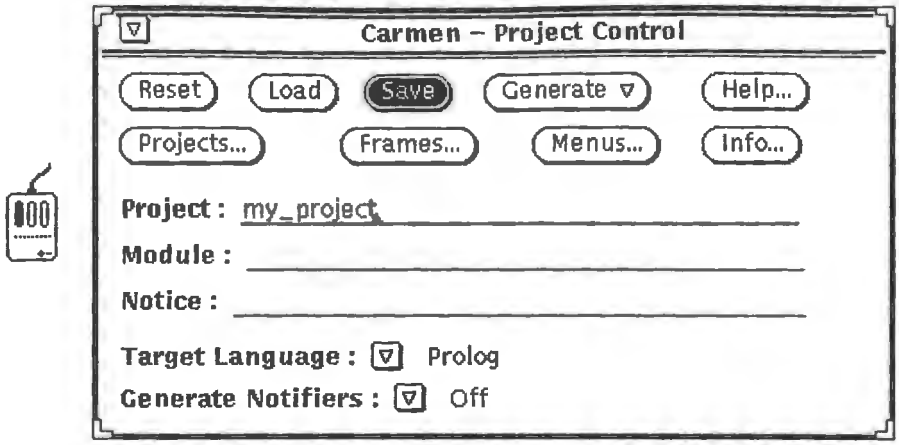
Type in 'My Base Frame', a label which will appear in the upper border of the frame and then click the **Apply** ActionButton.



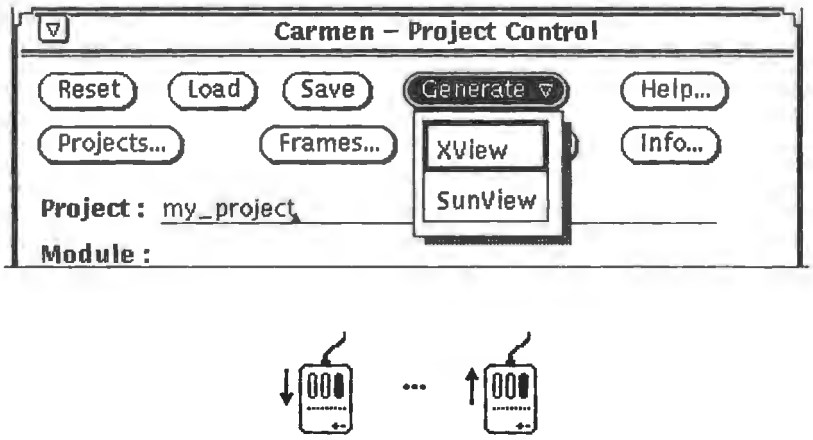
Our Base_frame is now correctly labelled:



The design of our first example is nearly finished. One of the last things we have to take care of, is to give a name to our project and to save it. Fill in the **Project** Textfield with the name of our project followed by <return>. Then, click the **Save** ActionButton.



The specification of our project is now saved in the file *my_project.wid.*
The code generation is activated by using the Generate MenuButton.



The generated file *(my_project.xv.pro)* is placed in the current directory. At this point, Carmen is no longer needed and can be quitted.

The *my_project* application can be executed by invoking the following command:

```
% BIMprologXV my_project.xv
```

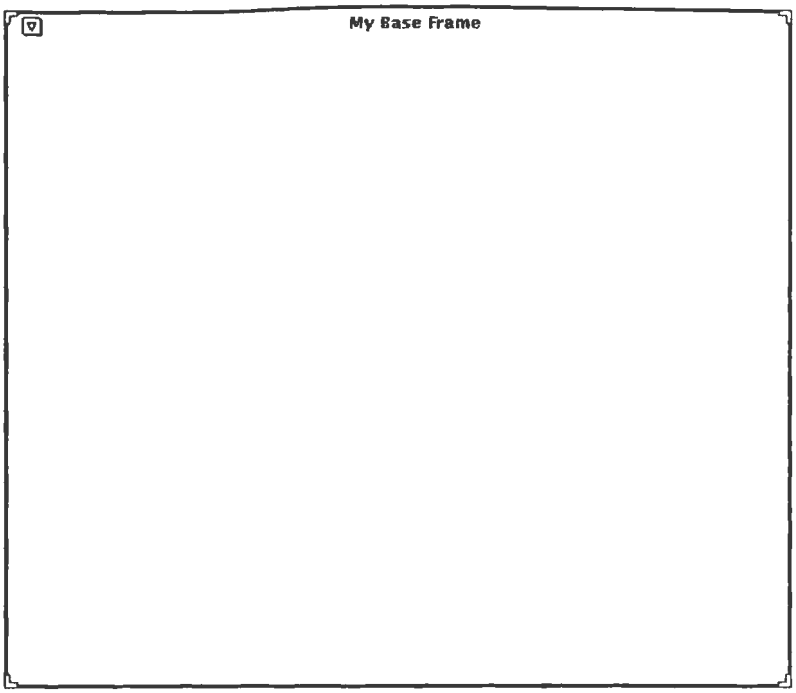
... and the Prolog goal

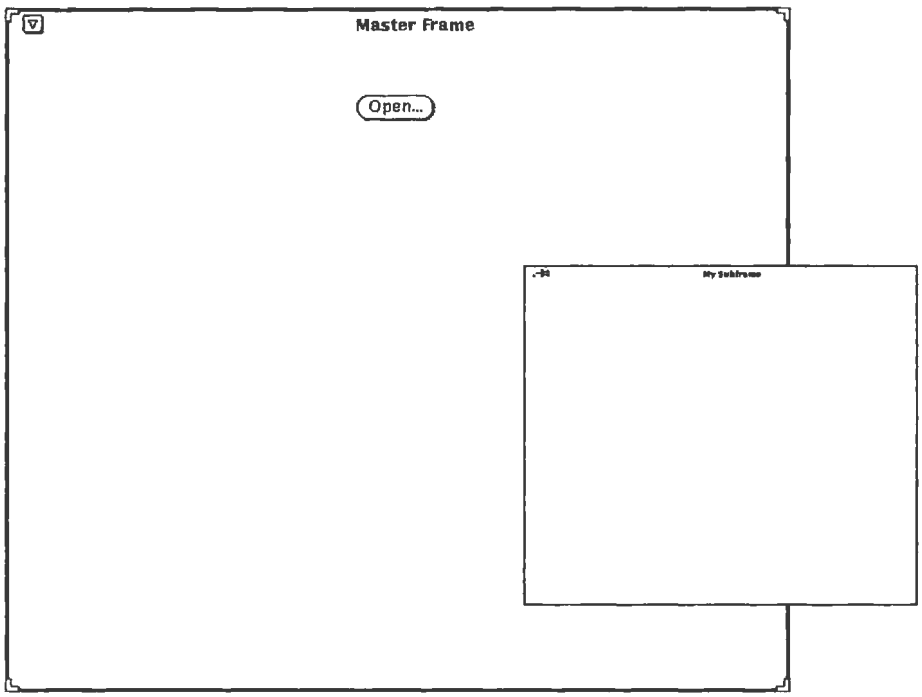
```
?- ui_initialize, ui_main.
```

Your first application designed with Carmen is running:

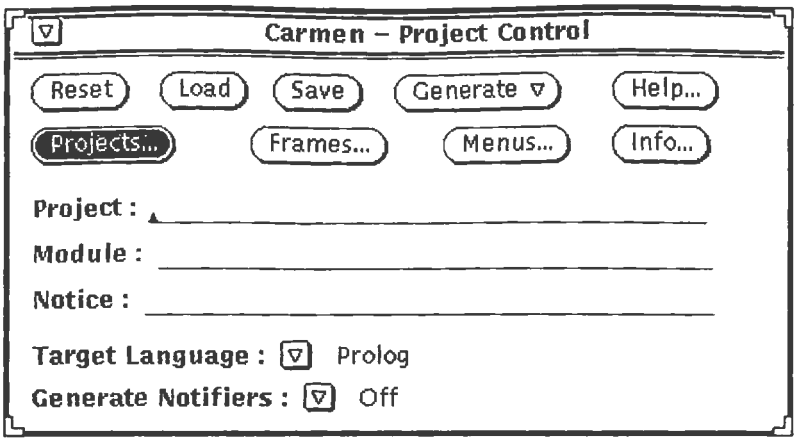
Example two

Let's enhance example one with one panel subwindow containing a windowbutton that opens a frame subwindow:

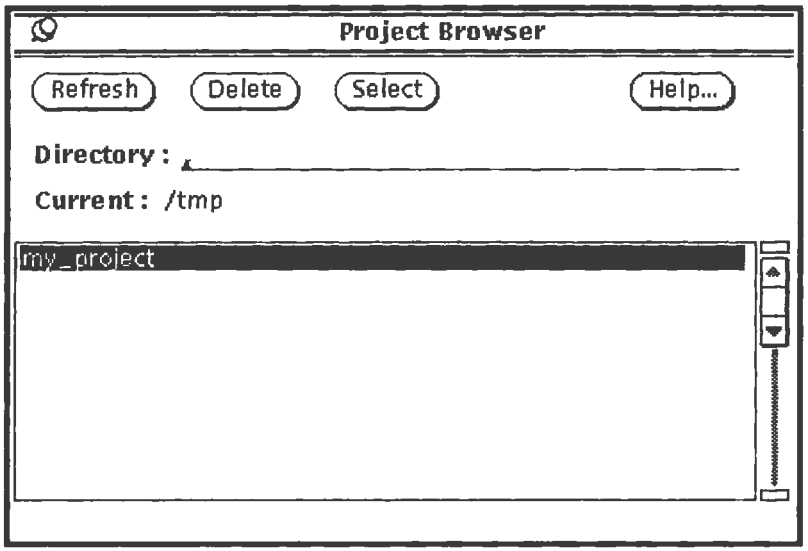


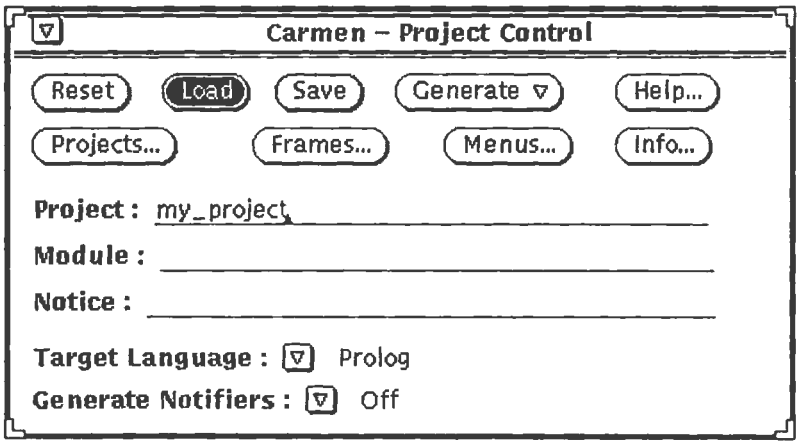


Run Carmen, open the **Project Control** frame, and  the **Projects** WindowButton.

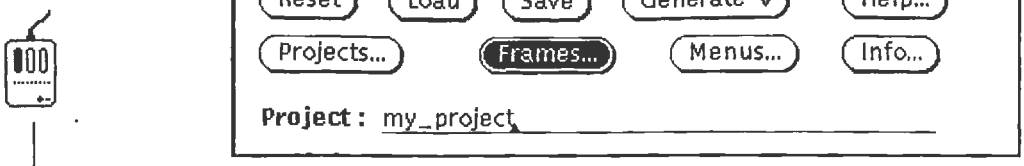


Select *my-project* and click the **Select** Button...

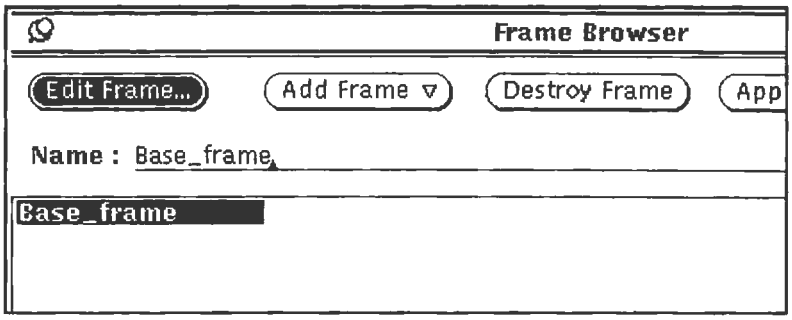




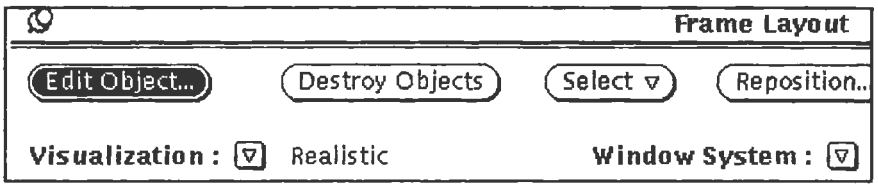
... open the Frame Browser...



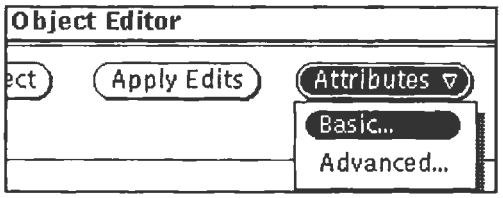
Edit *Base_frame*...



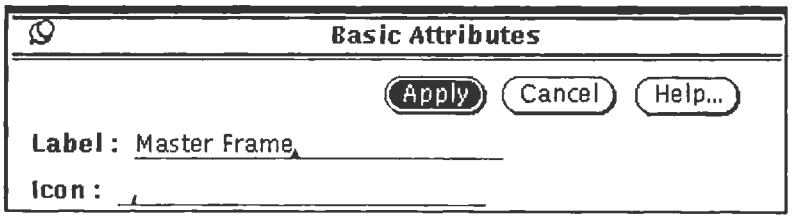
We are going to change the frame label from 'My Base Frame' to 'Master Frame'. Select the frame (little squares appear in reverse) and edit it.



In the Object Editor window, Select the **Basic attributes** option in the menu of the **Attributes** MenuButton.

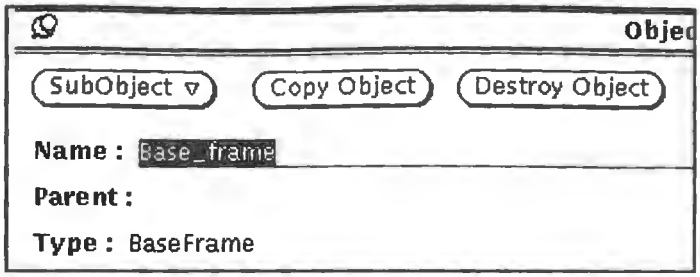


Change the Label to '*Master Frame*' and select the Apply ActionButton.



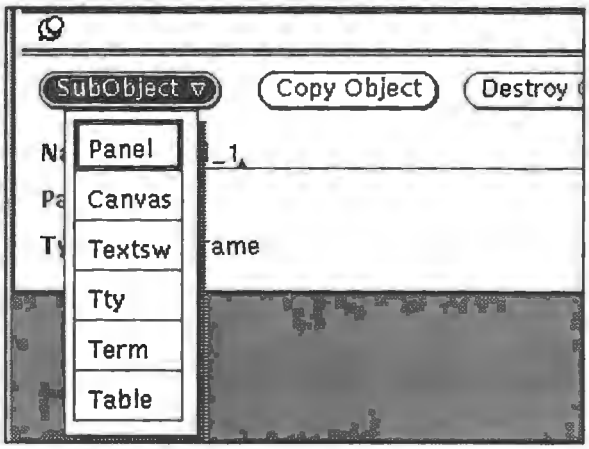
We will now define a panel (Panel_1) inside the frame (Base_frame). Select the Base_frame (if it is de-selected) and edit it. Delete the *Base_frame* name by...

...L10 (cut)

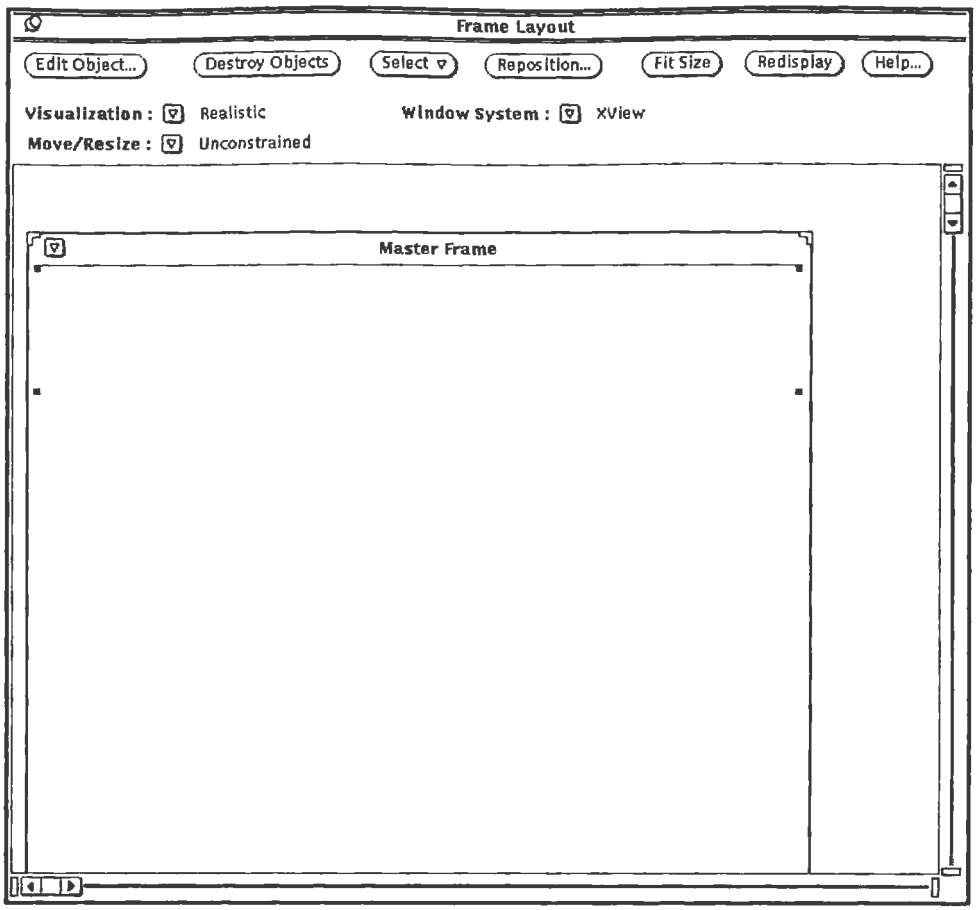


... and replace it by the name of the object you are going to create in *Base_frame: Panel_1*.

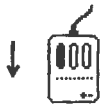
Select the type of the new object you are creating: **Panel**.



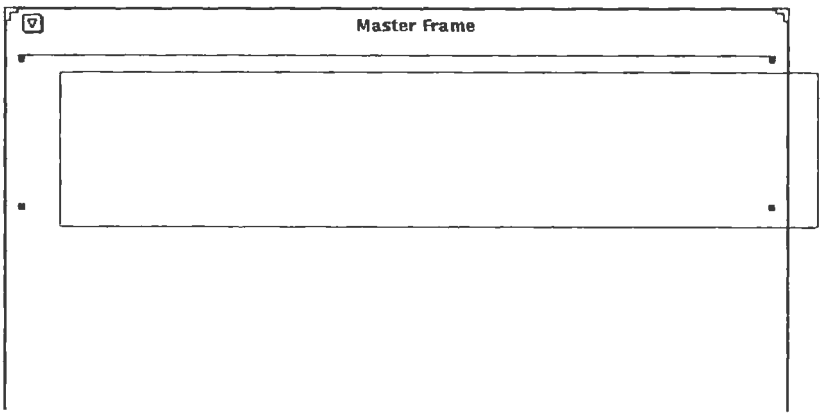
The panel *Panel_1* appears in *Base_frame* with a default size and position.

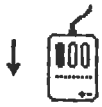
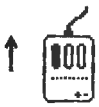


Objects in the Frame Editor can be moved and resized **Horizontally**, **Vertically** or without movement constraint. These options are set with the **Move/Resize** SelectButton.

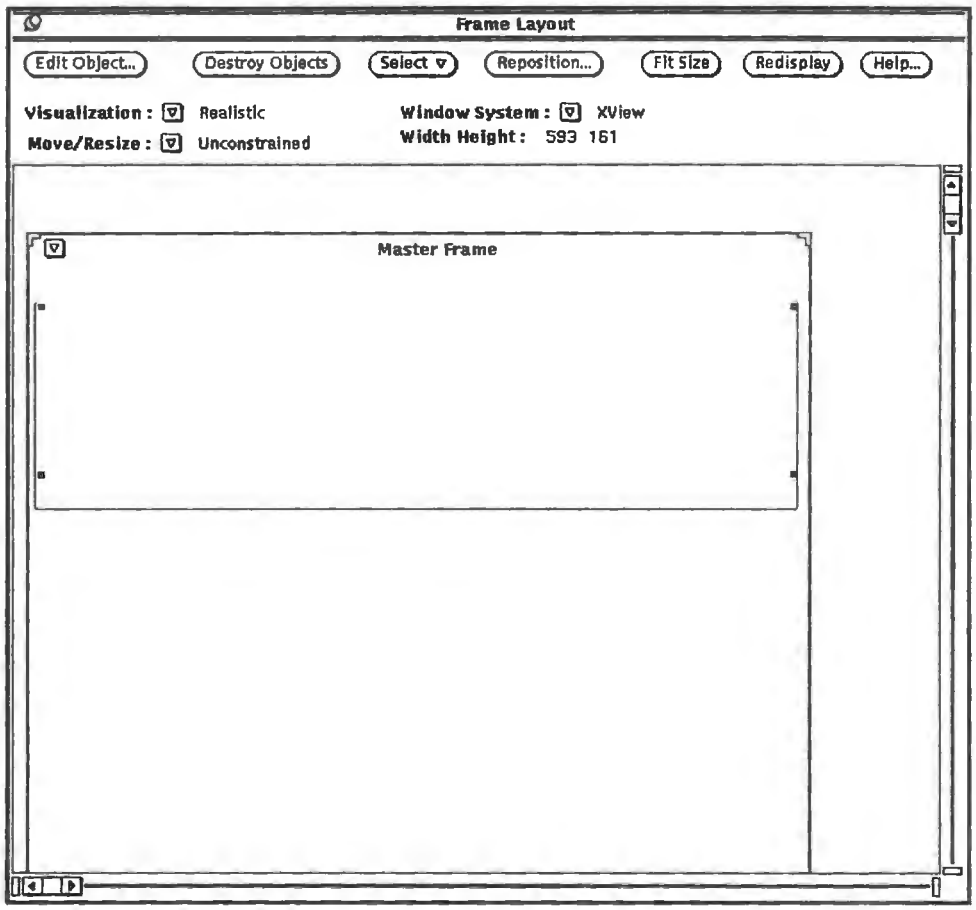
To **move** an object,  once to select it,  then move

it to the desired position and 

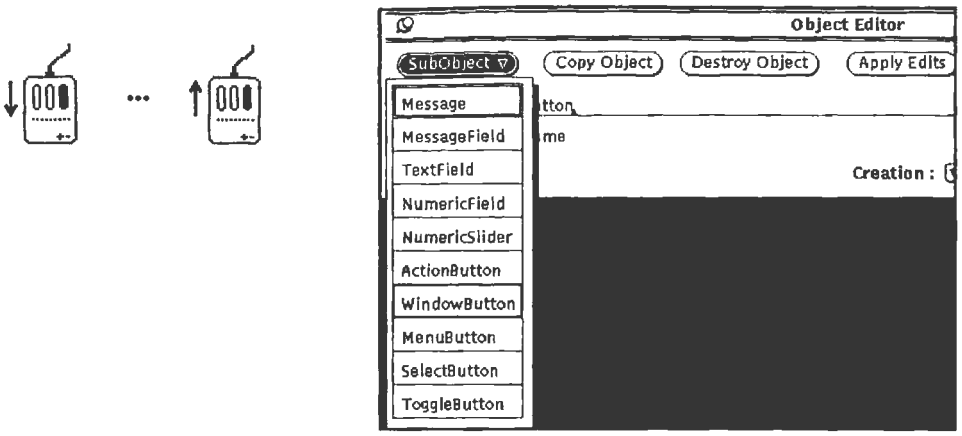


To **resize** an object,  once to select it and  on one of the four squares delimiting the object, move the mouse to resize it and when the correct size is reached. 

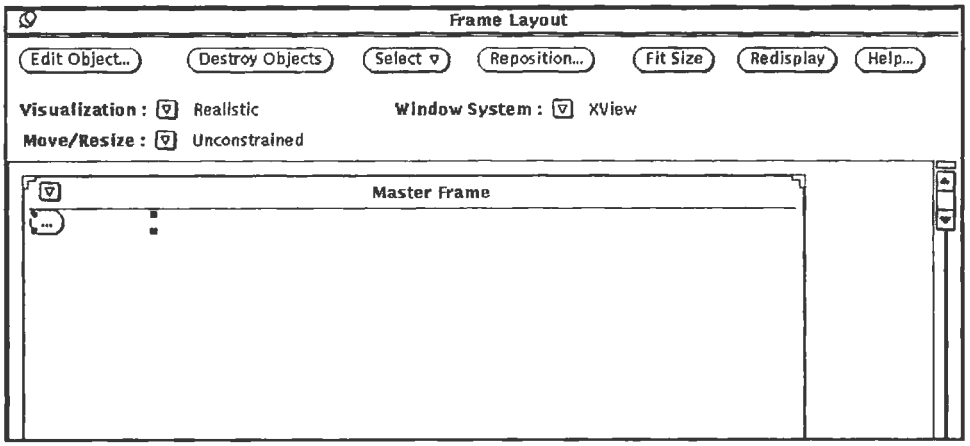
We now resize the *Panel_1* object:



In the panel, we include a WindowButton, *Open_button*, that will open a SubFrame, *Sub_frame*. Select the Panel (if it is de-selected) and edit it. Delete the *Panel_1* name and replace it by the name of the new object: *Open_button*. Select the type of the new object you are creating: WindowButton

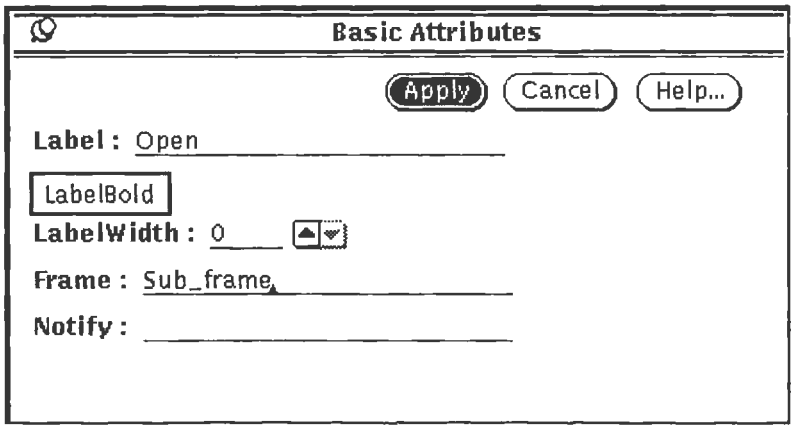


The new object appears in its parent, *Panel_1*.

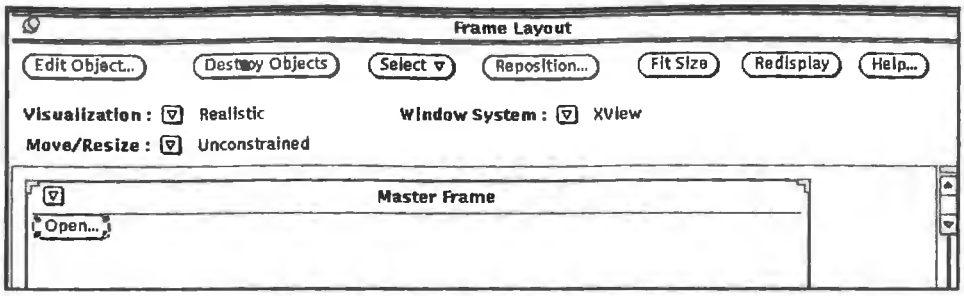


Select and edit the WindowButton, and open the Basic Attributes window. Give the label *Open* to the button.

the **LabelBold** toggle, and fill in the **Frame** field with the name of the SubFrame: *Sub_frame*. Then, click the **Apply** button.

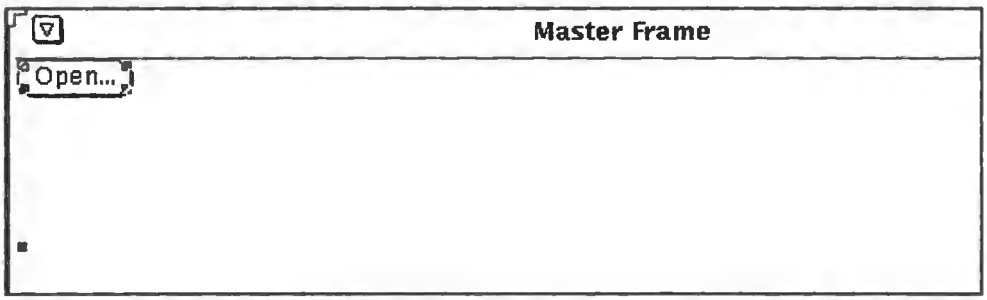


The OpenButton is now correctly labelled:

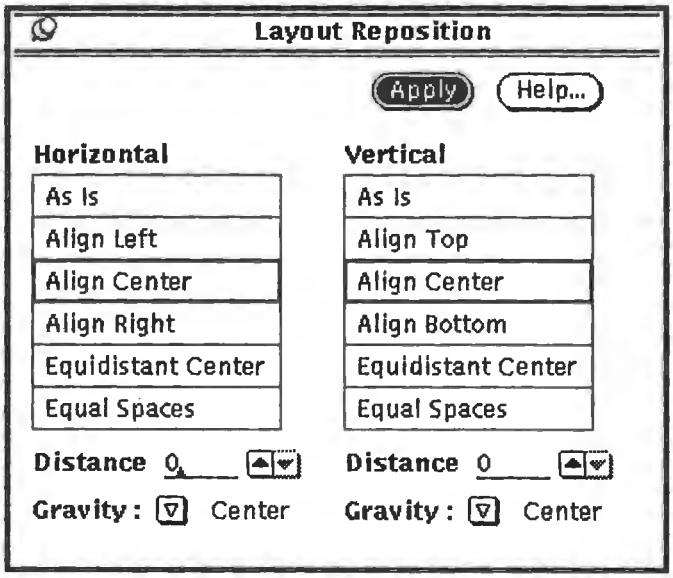


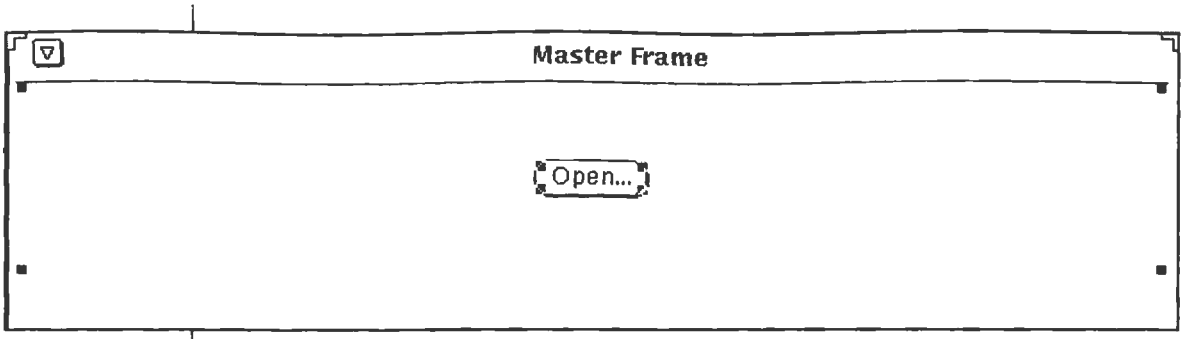
We now place the button just in the middle of the panel.

 . on the button,  . the panel.



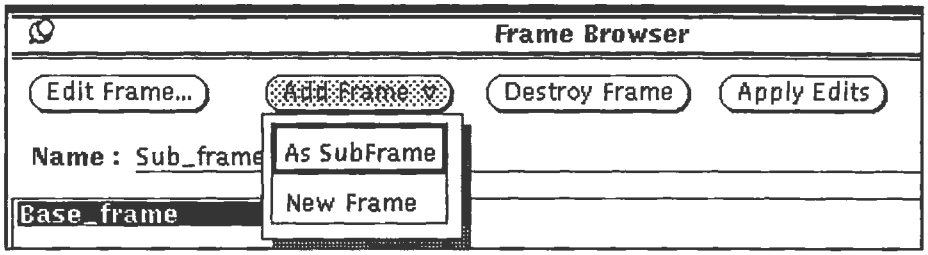
 ... the **Reposition** WindowButton. Select the correct alignment and apply.



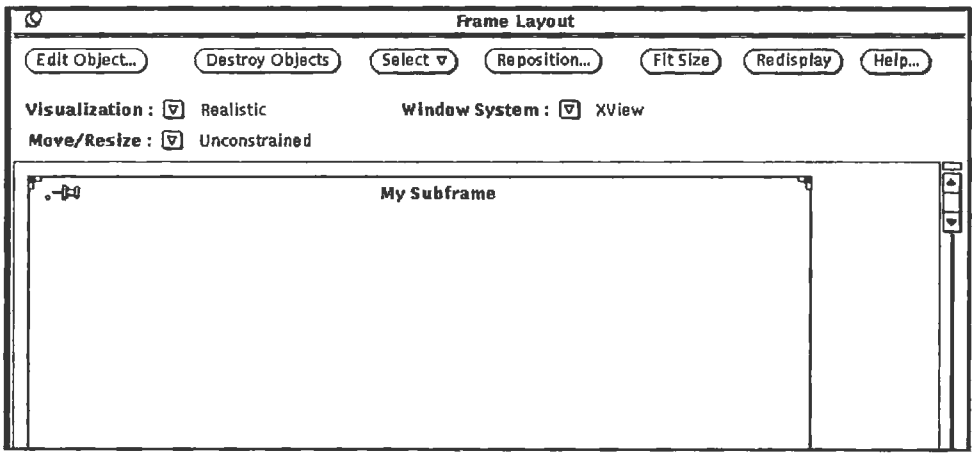


The design of our *Base_frame* is finished.

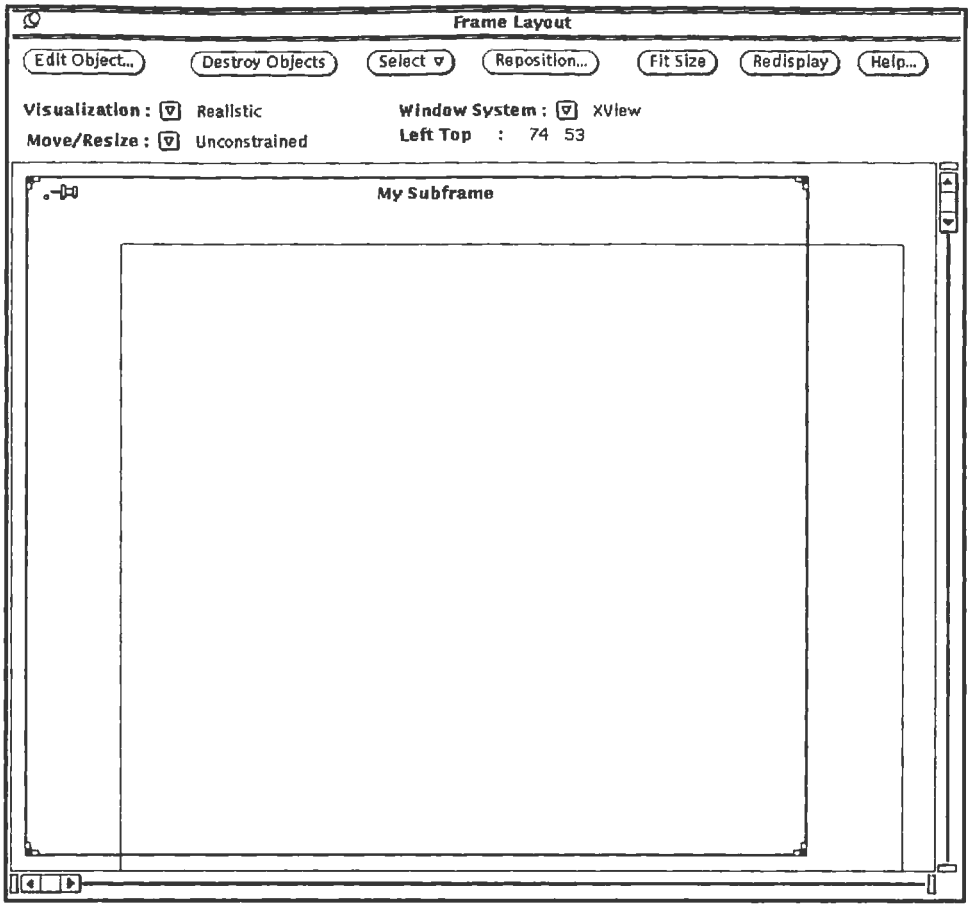
We now create the subframe in the Frame Browser. We replace the **Name** field of the frame browser by *Sub_frame*, the name of the frame that will appear when clicking the *Open_button*.



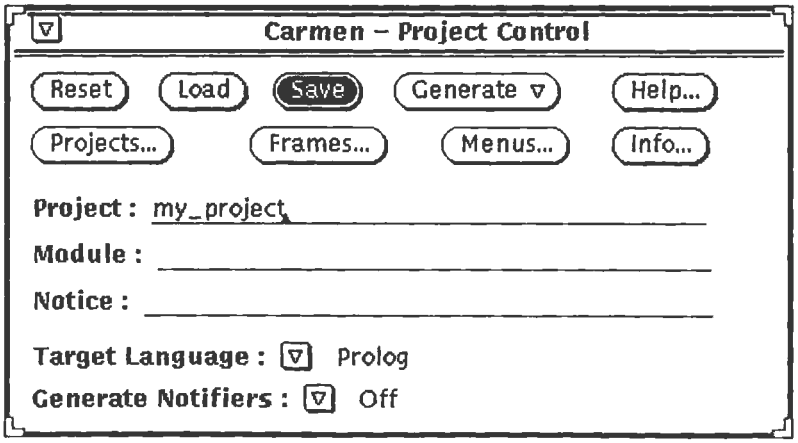
Edit *Sub_frame* and label it as *My Subframe*, then resize it like this:



The position of the subframe in the Frame Layout reflects the position of the subframe relative to the upper-left corner of its frame parent. Move the subframe down and to the right:

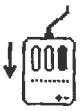


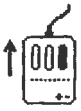
In the Project control window, the **Module** field is optional. When filled in with a name, the generated code contains a " **:- module**" directive with the specified module name. The **Notice** field is used to insert in the header of the generated files a copyright or a comment that you specify. The **Generate Notifiers** SelectButton is used to generate a separate file containing the heads of the prolog predicates that are called by the user interface, this will be explained in example 3.
Save the project.

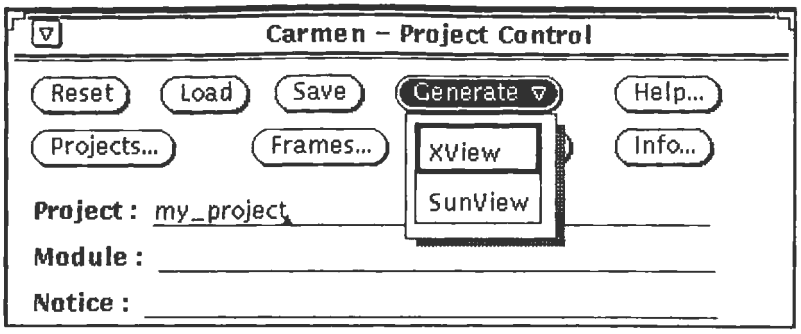


A pop-up message appears asking for confirmation to overwrite the code of the project, saved under the same name (example one).

Generate the Prolog code...







The generated file (*my_project.xv.pro*) is placed in the current directory. At this point, Carmen is no more needed and can be quitted.

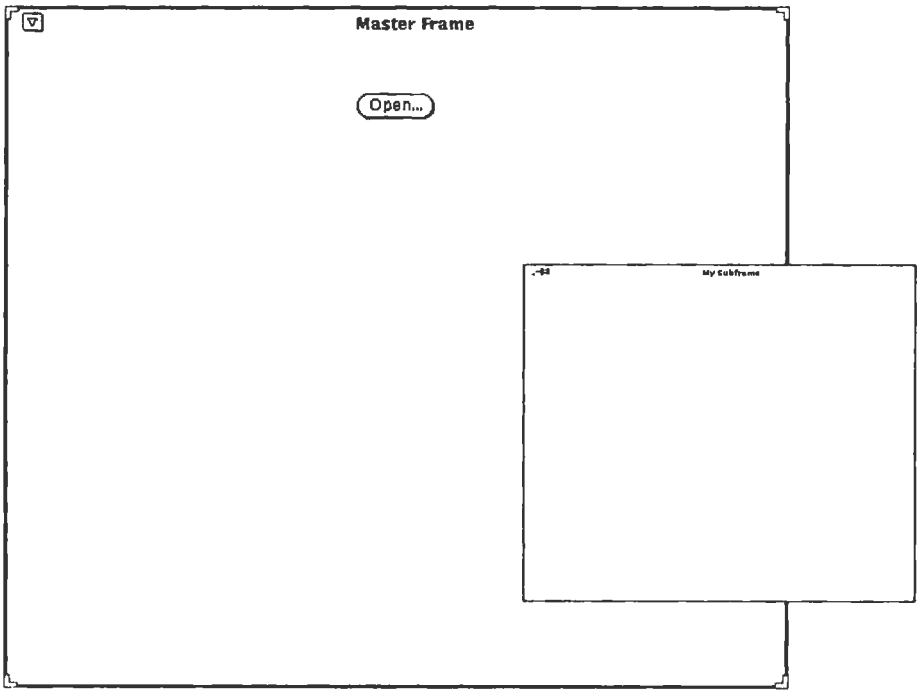
The *my_project* application can be executed by invoking the following command:

```
% BIMprologXV my_project.xv
```

... and the Prolog goal

```
?- ui_initialize, ui_main.
```

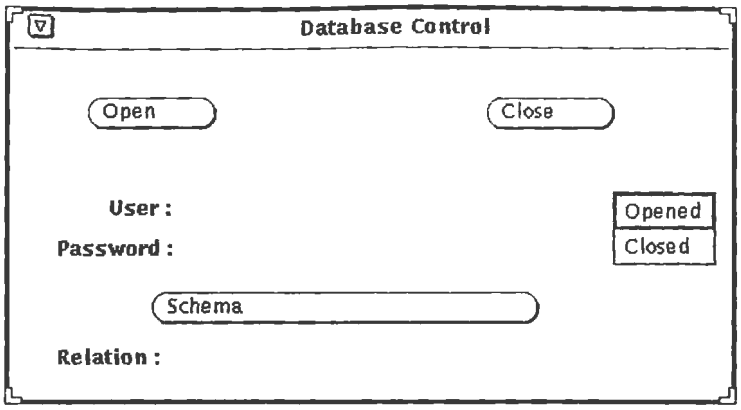
Your second application designed with Carmen is running:



Example three

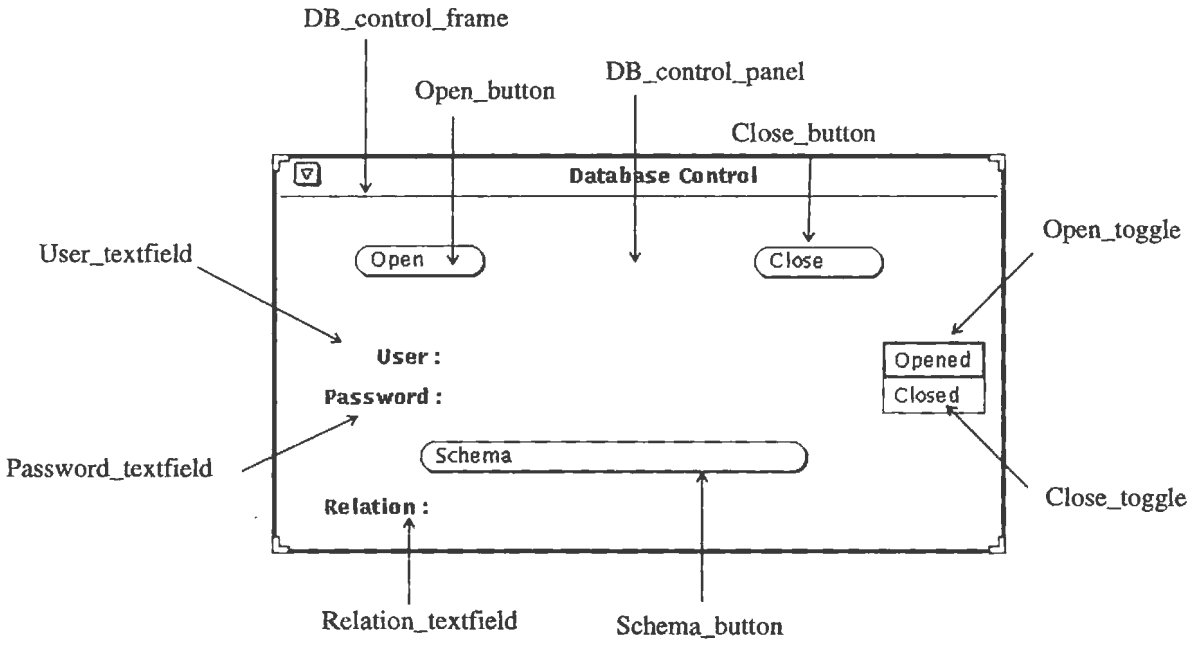
Database Control

As third example, we will design a small Database Control interface looking like this:



We skip the basic steps leading to the creation of the layout of this user interface. Nevertheless, an overview of the interface components is given hereafter.

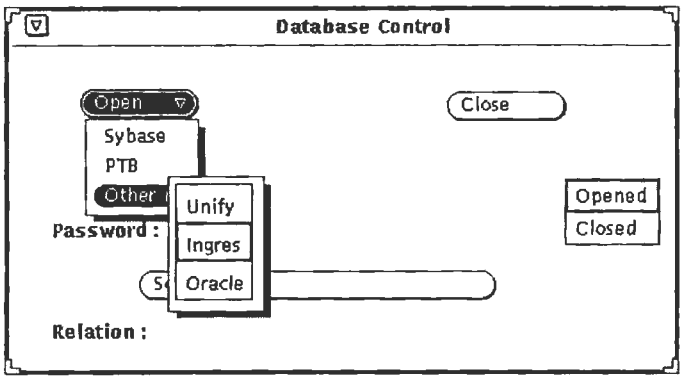
Basic components



Object Type | Name | Attributes to be set:

DB_control_frame | Frame | Label=Database Control
DB_control_panel | Panel | none
Open_button | WindowButton | Label=Open, LabelBold, LabelWidth=13, Menu=Open_menu
Close_button | ActionButton | Label=Close, LabelBold, LabelWidth=13, Notify=Close_db
Schema_button | ActionButton | Label=Schema, LabelBold, LabelWidth=25, Notify=Get_schema
User_textfield | Textfield | Label=User: , LabelBold, ValueWidth=20
Password_textfield | Textfield | Label=Password: , LabelBold, ValueWidth=20
Open_toggle | ToggleButton | Label=Opened, LabelBold
Close_toggle | ToggleButon | Label=Closed, LabelBold, Value

Clicking the Open button will pop-up a Menu like:

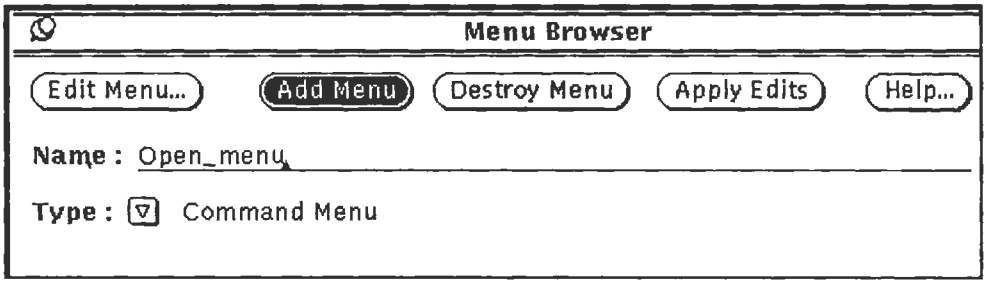


The invoked Menu, *Open_menu*, is a CommandMenu containing two ActionItems and One MenuItem calling another Menu, *Other_menu*.
Other_menu is a ChoiceMenu containing three SelectItems.

Here is how to create those Menus:

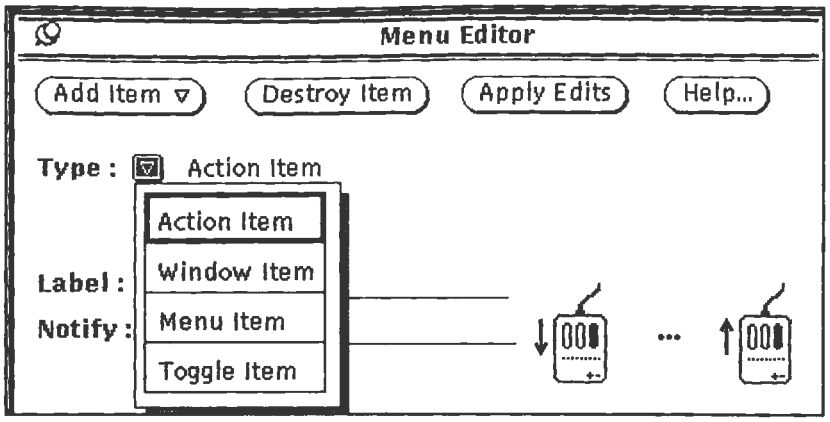
In the Project Control Frame,  the Menus WindowButton.

In the Menu Browser, fill in the **Name** field with the name of the CommandMenu: *Open_menu*. Set the Menu type to **Command Menu** and add this Menu.

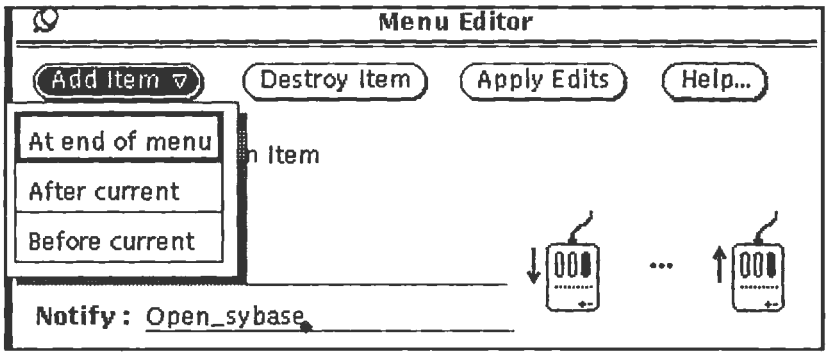


Then, edit it...

In the Menu Editor, define the items contained in the CommandMenu. Select the Action Item choice of the **Type** SelectButton:

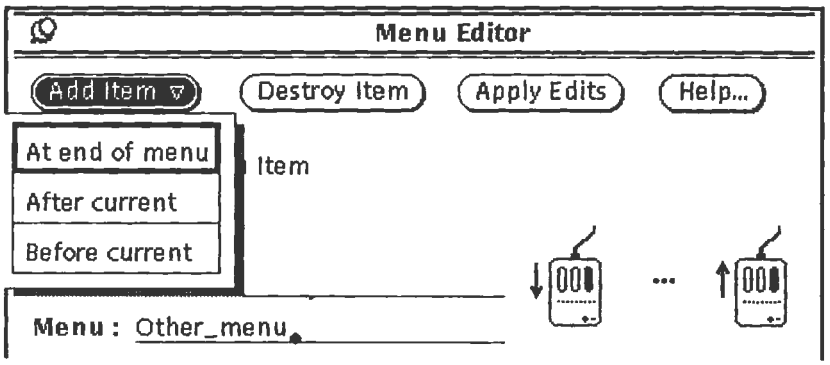


The label of the first Item is *Sybase* and the **Notify** to be called is *Open_Sybase*. Add this new item, at the end of the Menu.



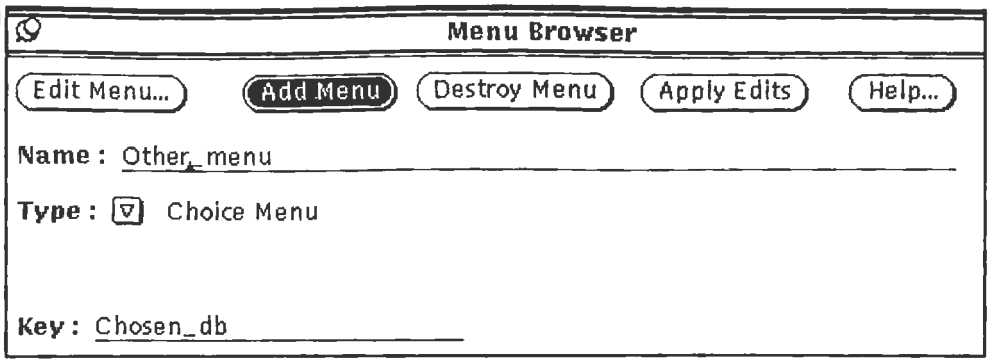
Repeat this operation for the second Item *PTB* with as notify *Open_PTB*.

The third Item is a MenuItem. Select the Menu Item choice of the **Type** SelectButton and fill in the **Label**, and **Menu** fields. Then add the Item at the end of the Menu:



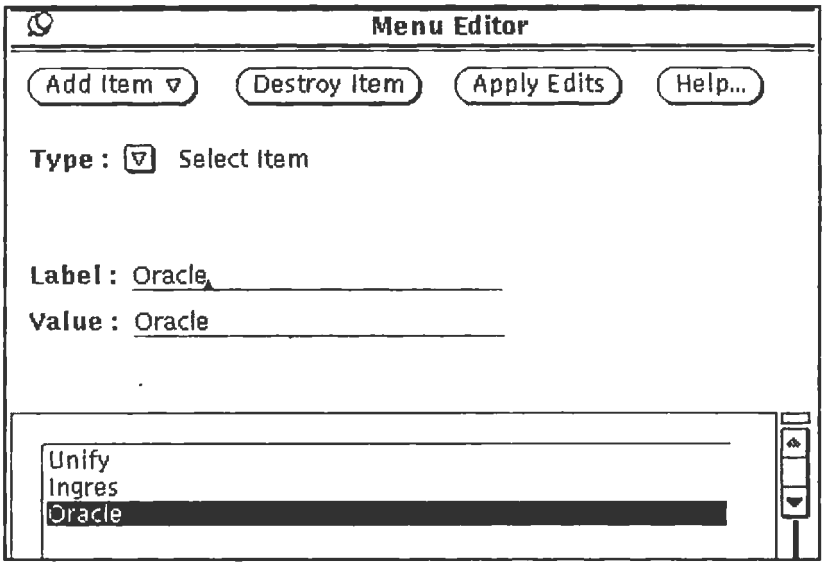
In the Menu Browser, set the Type Menu to **Choice Menu**.
Two fields appear: **Key** and **Notify**.
When specified, the name in the first field will serve as a key for a Prolog global value. (record(key_name,_value)) This global value will contain the value of the latest selected item in the Menu.
The **Notify** procedure specified in the second field will be called whenever an item will be selected in the Menu.

Create *Other_menu*:



Edit *Other_menu* and add the Select items:

Labels: Unify, Ingres, Oracle.
Values: Unify, Ingres, Oracle.



The design of this user interface is finished. You may close all windows but the Project Control one. Type in a project name theres (e.g. dbcontrol) followed by <return>, specify your own copyright, and save the project. Switch on the **Generate Notifiers** and generate the Prolog code. Now you can quit Carmen.

Two files have been generated: dbcontrol.xv.pro and dbcontrol.nh.pro

The latter contains the definitions of the notify procedures you referred to, in the user interface.

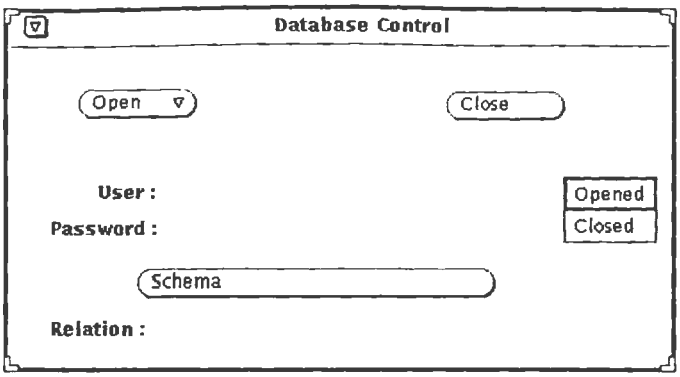
Run the application:

```
% BIMprologXV dbcontrol/xv dbcontrol.nh
```

... and the Prolog goal

```
?- ui_initialize, ui_main.
```

Your third application designed with Carmen is running:



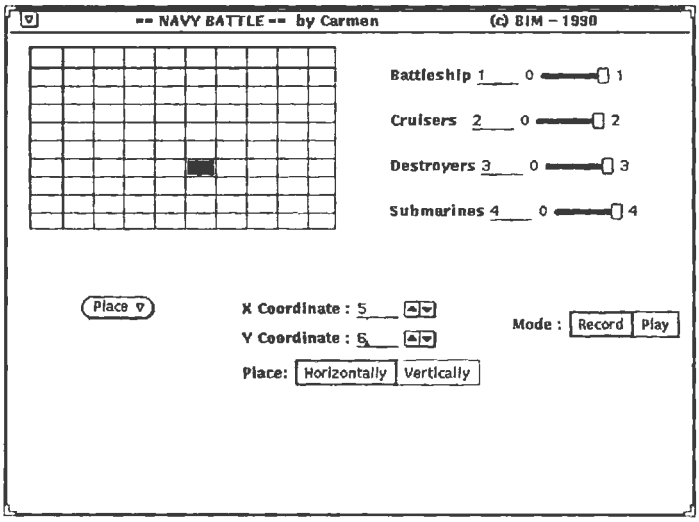
You can now complete the definitions in dbcontrol.nh.pro, in order to have your Database control interface 'really' running.

Example four

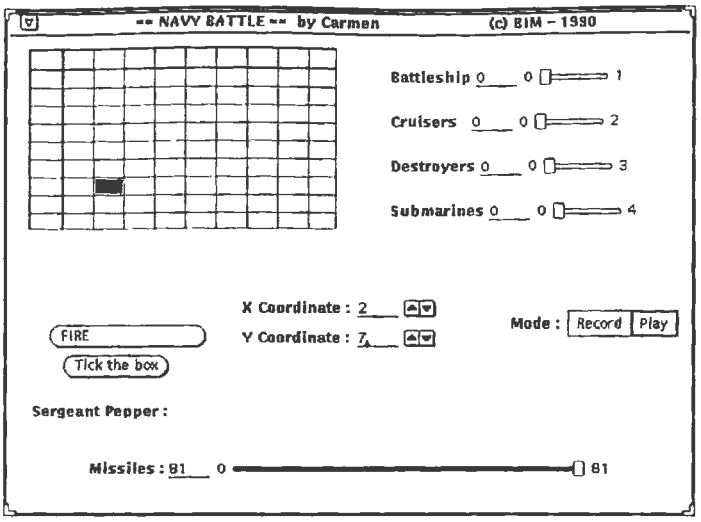
Navy Battle

This application is in fact a game that can be easily designed with Carmen.
One can distinguish between two modes:

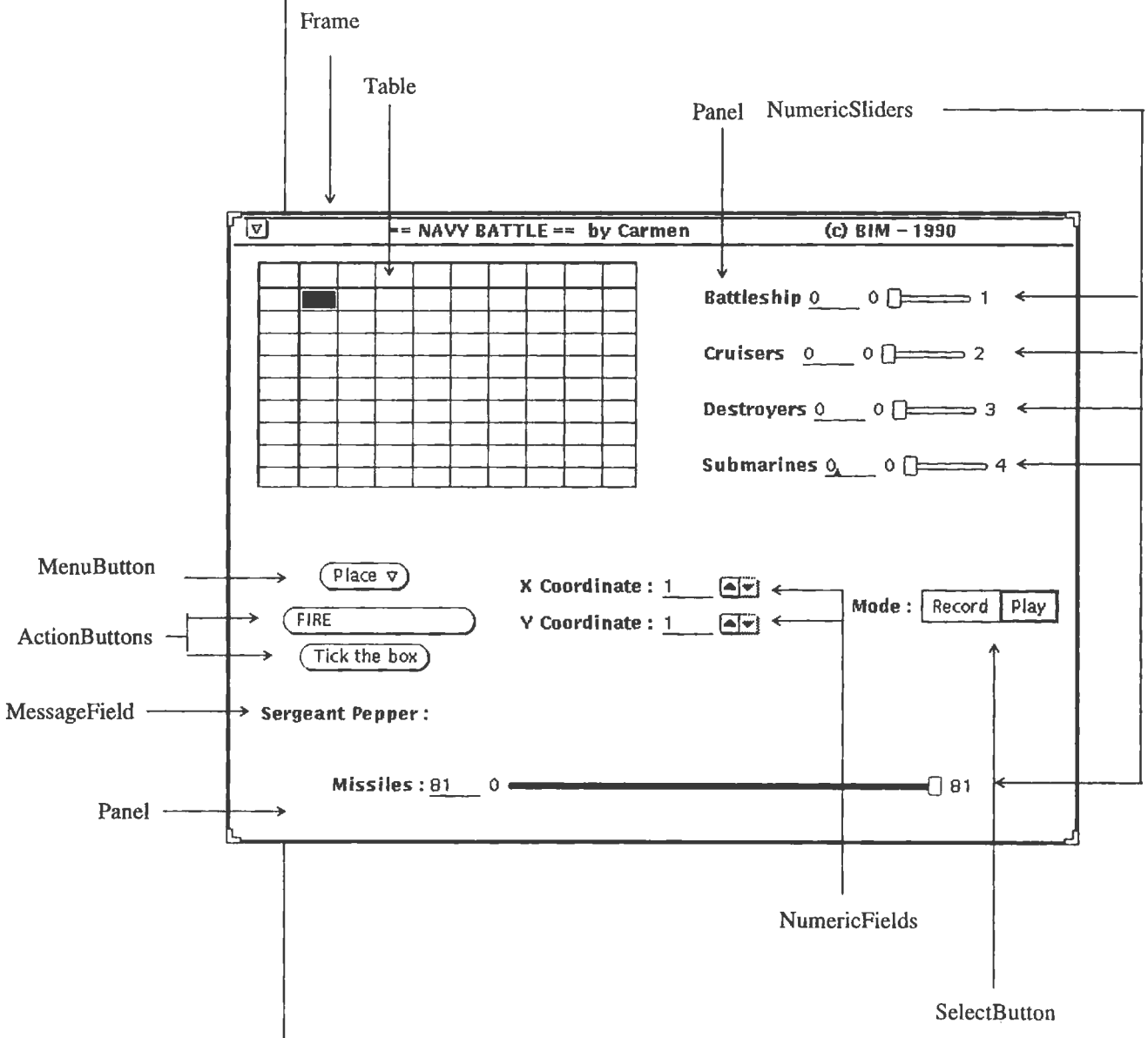
Record Mode: Placing of the ships on the grid.



2. Play mode.



Basic Components



As an exercise, the reader is invited to try to design the described interface and implement his/her own rules. Nevertheless, a ready-to-run application is available in the examples of Carmen ($BIM_PROLOG_DIR/demos/Carmen). The concerned files are navy.wid, navy.xv.pro and navy_engine.pro .

navy_engine.pro is documented and shows the possibility of using the high-level predicates of Carmen to access the interface objects, but also to access them with XView predicates for particular operations that would not have been defined in Carmen.

The high-level predicates of Carmen are described in the Carmen Reference Guide.