University of Amsterdam

Dept. of Social Science Informatics (SWI)
Roeterstraat 15, 1018 WB Amsterdam
The Netherlands
Tel. (+31) 20 5256121

SWI-Prolog 4.0

Reference Manual

Updated for version 4.0.2, March 2001

Jan Wielemaker

jan@swi.psy.uva.nl http://www.swi.psy.uva.nl/projects/SWI-Prolog/

SWI-Prolog is a Prolog implementation based on a subset of the WAM (Warren Abstract Machine). SWI-Prolog was developed as an *open* Prolog environment, providing a powerful and bi-directional interface to C in an era this was unknown to other Prolog implementations. This environment is required to deal with XPCE, an object-oriented GUI system developed at SWI. XPCE is used at SWI for the development of knowledge-intensive graphical applications.

As SWI-Prolog became more popular, a large user-community provided requirements that guided its development. Compatibility, portability, scalability, stability and providing a powerful development environment have been the most important requirements. Edinburgh, Quintus, SICStus and the ISO-standard guide the development of the SWI-Prolog primitives.

This document gives an overview of the features, system limits and built-in predicates.

Contents

1 Introduction

1.1 SWI-Prolog

SWI-Prolog has been designed and implemented to get a Prolog implementation which can be used for experiments with logic programming and the relation to other programming paradigms. The intention was to build a Prolog environment which offers enough power and flexibility to write substantial applications, but is straightforward enough to be modified for experiments with debugging, optimisation or the introduction of non-standard data types. Performance optimisation is limited due to the main objectives: portability (SWI-Prolog is entirely written in C and Prolog) and modifiability.

SWI-Prolog is based on a very restricted form of the WAM (Warren Abstract Machine) described in [Bowen & Byrd, 1983] which defines only 7 instructions. Prolog can easily be compiled into this language and the abstract machine code is easily decompiled back into Prolog. As it is also possible to wire a standard 4-port debugger in the WAM interpreter there is no need for a distinction between compiled and interpreted code. Besides simplifying the design of the Prolog system itself this approach has advantages for program development: the compiler is simple and fast, the user does not have to decide in advance whether debugging is required and the system only runs slightly slower when in debug mode. The price we have to pay is some performance degradation (taking out the debugger from the WAM interpreter improves performance by about 20%) and somewhat additional memory usage to help the decompiler and debugger.

SWI-Prolog extends the minimal set of instructions described in [Bowen & Byrd, 1983] to improve performance. While extending this set care has been taken to maintain the advantages of decompilation and tracing of compiled code. The extensions include specialised instructions for unification, predicate invocation, some frequently used built-in predicates, arithmetic, and control (`;/2`, `|/2`), if-then (`->/2`) and negation-by-failure (`\+/1`).

1.1.1 Other books about Prolog

This manual does not describe the full syntax and semantics of Prolog, nor how one should write a program in Prolog. These subjects have been described extensively in the literature. See [Bratko, 1986], [Sterling & Shapiro, 1986], and [Clocksin & Melish, 1987]. For more advanced Prolog material see [OKeefe, 1990]. Syntax and standard operator declarations confirm to the 'Edinburgh standard'. Most built in predicates are compatible with those described in [Clocksin & Melish, 1987]. SWI-Prolog also offers a number of primitive predicates compatible with Quintus Prolog[1] [Qui, 1997] and BIM_Prolog[2] [BIM, 1989].

ISO compliant predicates are based on "Prolog: The Standard", [Deransart *et al.*, 1996], validated using [**?**].

[1]Quintus is a trademark of Quintus Computer Systems Inc., USA
[2]BIM is a trademark of BIM sa/nv., Belgium

1.2 Status

This manual describes version 4.0 of SWI-Prolog. SWI-Prolog has been used now for many years. The application range includes Prolog course material, meta-interpreters, simulation of parallel Prolog, learning systems, natural language processing and two large workbenches for knowledge engineering. Although we experienced rather obvious and critical bugs can remain unnoticed for a remarkable long period, we assume the basic Prolog system is fairly stable. Bugs can be expected in infrequently used built-in predicates.

Some bugs are known to the author. They are described as footnotes in this manual.

1.3 Compliance to the ISO standard

SWI-Prolog 3.3.0 implements all predicates described in "Prolog: The Standard" [Deransart *et al.*, 1996].

Exceptions and warning are still weak. Some SWI-Prolog predicates silently fail on conditions where the ISO specification requires an exception (`functor/3` for example). Many predicates print warnings rather than raising an exception. All predicates where exceptions may be caused due to a correct program operating in an imperfect world (I/O, arithmetic, resource overflows) should behave according to the ISO standard. In other words: SWI-Prolog should be able to execute any program conforming to [Deransart *et al.*, 1996] that does not rely on exceptions generated by errors in the program.

1.4 Should you be using SWI-Prolog?

There are a number of reasons why you better choose a commercial Prolog system, or another academic product:

- *SWI-Prolog is not supported*
  Although I usually fix bugs shortly after a bug report arrives, I cannot promise anything. Now that the sources are provided, you can always dig into them yourself.

- *Memory requirements and performance are your first concerns*
  A number of commercial compilers are more keen on memory and performance than SWI-Prolog. I do not wish to sacrifice some of the nice features of the system, nor its portability to compete on raw performance.

- *You need features not offered by SWI-Prolog*
  In this case you may wish to give me suggestions for extensions. If you have great plans, please contact me (you might have to implement them yourself however).

On the other hand, SWI-Prolog offers some nice facilities:

- *Nice environment*
  This includes 'Do What I Mean', automatic completion of atom names, history mechanism and a tracer that operates on single key-strokes. Interfaces to some standard editors are provided (and can be extended), as well as a facility to maintain programs (see `make/0`).

- *Very fast compiler*
  Even very large applications can be loaded in seconds on most machines. If this is not enough, there is a Quick Load Format that is slightly more compact and loading is almost always I/O bound.

- *Transparent compiled code*
  SWI-Prolog compiled code can be treated just as interpreted code: you can list it, trace it, etc. This implies you do not have to decide beforehand whether a module should be loaded for debugging or not. Also, performance is much better than the performance of most interpreters.

- *Profiling*
  SWI-Prolog offers tools for performance analysis, which can be very useful to optimise programs. Unless you are very familiar with Prolog and Prolog performance considerations this might be more helpful than a better compiler without these facilities.

- *Flexibility*
  SWI-Prolog can easily be integrated with C, supporting non-determinism in Prolog calling C as well as C calling Prolog (see section 5. It can also be *embedded* embedded in external programs (see section 5.7). System predicates can be redefined locally to provide compatibility with other Prolog systems.

- *Integration with XPCE*
  SWI-Prolog offers a tight integration to the Object Oriented Package for User Interface Development, called XPCE [Anjewierden & Wielemaker, 1989]. XPCE allows you to implement graphical user interfaces that are source-code compatible over Unix/X11 and Win32 (Windows 95 and NT).

1.5 The XPCE GUI system for Prolog

The XPCE GUI system for dynamically typed languages has been with SWI-Prolog for a long time. It is developed by Anjo Anjewierden and Jan Wielemaker from the department of SWI, University of Amsterdam. It aims at a high-productive development environment for graphical applications based on Prolog.

Object oriented technology has proven to be a suitable model for implementing GUIs, which typically deal with things Prolog is not very good at: event-driven control and global state. With XPCE, we designed a system that has similar characteristics that make Prolog such a powerful tool: dynamic typing, meta-programming and dynamic modification of the running system.

XPCE is an object-system written in the C-language. It provides for the implementation of methods in multiple languages. New XPCE classes may be defined from Prolog using a simple, natural syntax. The body of the method is executed by Prolog itself, providing a natural interface between the two systems. Below is a very simple class definition.

```
:- pce_begin_class(prolog_lister, frame,
                   "List Prolog predicates").

initialise(Self) :->
        "As the C++ constructor"::
        send(Self, send_super, initialise, 'Prolog Lister'),
        send(Self, append, new(D, dialog)),
        send(D, append,
             text_item(predicate, message(Self, list, @arg1))),
        send(new(view), below, D).

list(Self, From:name) :->
        "List predicates from specification"::
        (   catch(term_to_atom(Term, From), _, fail)
        ->  get(Self, member, view, V),
            pce_open(V, write, Fd),
            set_output(Fd),
            listing(Term),
            close(Fd)
        ;   send(Self, report, error, 'Syntax error')
        ).

:- pce_end_class.

test :- send(new(prolog_lister), open).
```

Its 165 built-in classes deal with the meta-environment, data-representation and—of course—graphics. The graphics classes concentrate on direct-manipulation of diagrammatic representations.

Availability. XPCE runs on most Unix^{tm} platforms, Windows 95, 98 and Windows NT. It has been connected to SWI-Prolog, SICStus^{tm} and Quintus^{tm} Prolog as well as some Lisp dialects and C++. The Quintus version is commercially distributed and supported as ProWindows-3^{tm}.

Info. further information is available from `http://www.swi.psy.uva.nl/projects/xpce/` or by E-mail to `xpce-request@swi.psy.uva.nl`. There are demo versions for Windows 95, 98, NT and i386/Linux available from the XPCE download page.

1.6 Release Notes

Collected release-notes. This section only contains some highlights. Smaller changes to especially older releases have been removed. For a complete log, see the file `ChangeLog` from the distribution.

1.6.1 Version 1.8 Release Notes

Version 1.8 offers a stack-shifter to provide dynamically expanding stacks on machines that do not offer operating-system support for implementing dynamic stacks.

1.6.2 Version 1.9 Release Notes

Version 1.9 offers better portability including an MS-Windows 3.1 version. Changes to the Prolog system include:

- *Redefinition of system predicates*
  Redefinition of system predicates was allowed silently in older versions. Version 1.9 only allows it if the new definition is headed by a :- `redefine_system_predicate/1` directive.

- *'Answer' reuse*
  The toplevel maintains a table of bindings returned by toplevel goals and allows for reuse of these bindings by prefixing the variables with the $ sign. See section 2.8.

- *Better source code administration*
  Allows for proper updating of multifile predicates and finding the sources of individual clauses.

1.6.3 Version 2.0 Release Notes

New features offered:

- *32-bit Virtual Machine*
  Removes various limits and improves performance.

- *Inline foreign functions*
  'Simple' foreign predicates no longer build a Prolog stack-frame, but are directly called from the VM. Notably provides a speedup for the test predicates such as `var/1`, etc.

- *Various compatibility improvements*

- *Stream based I/O library*
  All SWI-Prolog's I/O is now handled by the stream-package defined in the foreign include file `SWI-Stream.h`. Physical I/O of Prolog streams may be redefined through the foreign language interface, facilitating much simpler integration in window environments.

1.6.4 Version 2.5 Release Notes

Version 2.5 is an intermediate release on the path from 2.1 to 3.0. All changes are to the foreign-language interface, both to user- and system-predicates implemented in the C-language. The aim is twofold. First of all to make garbage-collection and stack-expansion (stack-shifts) possible while foreign code is active without the C-programmer having to worry about locking and unlocking C-variables pointing to Prolog terms. The new approach is closely compatible to the Quintus and SICStus Prolog foreign interface using the `+term` argument specification (see their respective manuals). This allows for writing foreign interfaces that are easily portable over these three Prolog platforms.

Apart from various bug fixes listed in the Changelog file, these are the main changes since 2.1.0:

- *ISO compatibility*
  Many ISO compatibility features have been added: `open/4`, arithmetic functions, syntax, etc.

- *Win32*
  Many fixes for the Win32 (NT, '95 and win32s) platforms. Notably many problems related to pathnames and a problem in the garbage collector.

- *Performance*
  Many changes to the clause indexing system: added hash-tables, lazy computation of the index information, etc.

- *Portable saved-states*
  The predicate `qsave_program/[1,2]` allows for the creating of machine independent saved-states that load very quickly.

1.6.5 Version 2.6 Release Notes

Version 2.6 provides a stable implementation of the features added in the 2.5.x releases, but at the same time implements a number of new features that may have impact on the system stability.

- *32-bit integer and double float arithmetic*
  The biggest change is the support for full 32-bit signed integers and raw machine-format double precision floats. The internal data representation as well as the arithmetic instruction set and interface to the arithmetic functions has been changed for this.

- *Embedding for Win32 applications*
  The Win32 version has been reorganised. The Prolog kernel is now implemented as Win32 DLL that may be embedded in C-applications. Two front ends are provided, one for window-based operation and one to run as a Win32 console application.

- *Creating stand-alone executables*
  Version 2.6.0 can create stand-alone executables by attaching the saved-state to the emulator. See `qsave_program/2`.

1.6.6 Version 2.7 Release Notes

Version 2.7 reorganises the entire data-representation of the Prolog data itself. The aim is to remove most of the assumption on the machine's memory layout to improve portability in general and enable embedding on systems where the memory layout may depend on invocation or on how the executable is linked. The latter is notably a problem on the Win32 platforms. Porting to 64-bit architectures is feasible now.

Furthermore, 2.7 lifts the limits on arity of predicates and number of variables in a clause considerably and allow for further expansion at minimal cost.

1.6.7 Version 2.8 Release Notes

With version 2.8, we declare the data-representation changes of 2.7.x stable. Version 2.8 exploits the changes of 2.7 to support 64-bit processors like the DEC Alpha. As of version 2.8.5, the representation of recorded terms has changed, and terms on the heap are now represented in a compiled format. SWI-Prolog no longer limits the use of `malloc()` or uses assumptions on the addresses returned by this function.

1.6.8 Version 2.9 Release Notes

Version 2.9 is the next step towards version 3.0, improving ISO compliance and introducing ISO compliant exception handling. New are `catch/3`, `throw/1`, `abolish/1`, `write_term/[2,3]`, `write_canonical/[1,2]` and the C-functions `PL_exception()` and `PL_throw()`. The predicates `display/[1,2]` and `displayq/[1,2]` have been moved to library(`backcomp`), so old code referring to them will autoload them.

The interface to `PL_open_query()` has changed. The *debug* argument is replaced by a bitwise or'ed *flags* argument. The values `FALSE` and `TRUE` have their familiar meaning, making old code using these constants compatible. Non-zero values other than `TRUE` (1) will be interpreted different.

1.6.9 Version 3.0 Release Notes

Complete redesign of the saved-state mechanism, providing the possibility of 'program resources'. See `resource/3`, `open_resource/3`, and `qsave_program/[1,2]`.

1.6.10 Version 3.1 Release Notes

Improvements on exception-handling. Allows relating software interrupts (signals) to exceptions, handling signals in Prolog and C (see `on_signal/3` and `PL_signal()`). Prolog stack overflows now raise the `resource_error` exception and thus can be handled in Prolog using `catch/3`.

1.6.11 Version 3.3 Release Notes

Version 3.3 is a major release, changing many things internally and externally. The highlights are a complete redesign of the high-level I/O system, which is now based on explicit streams rather then current input/output. The old Edinburgh predicates (`see/1`, `tell/1`, etc.) are now defined on top of this layer instead of the other way around. This fixes various internal problems and removes Prolog limits on the number of streams.

Much progress has been made to improve ISO compliance: handling strings as lists of one-character atoms is now supported (next to character codes as integers). Many more exceptions have been added and printing of exceptions and messages is rationalised using Quintus and SICStus Prolog compatible `print_message/2`, `message_hook/3` and `print_message_lines/3`. All predicates descriped in [Deransart *et al.*, 1996] are now implemented.

As of version 3.3, SWI-Prolog adheres the ISO *logical update view* for dynamic predicates. See section 3.13.1 for details.

SWI-Prolog 3.3 includes garbage collection on atoms, removing the last serious memory leak especially in text-manipulation applications. See section 5.6.2. In addition, both the user-level and foreign interface supports atoms holding *0-bytes*.

Finally, an alpha version of a multi-threaded SWI-Prolog for Linux is added. This version is still much slower than the single-threaded version due to frequent access to 'thread-local-data' as well as some too detailed mutex locks. The basic thread API is ready for serious use and testing however. See section 3.39.

Incompatible changes

A number of incompatible changes result from this upgrade. They are all easily fixed however.

- `!/0`, `call/1`
  The cut now behaves according to the ISO standard. This implies it works in compound goals passed to `call/1` and is local to the *condition* part of if-then-else as well as the argument of `\+/1`.

- `atom_chars/2`
  This predicate is now ISO compliant and thus generates a list of one-character atoms. The

behaviour of the old predicate is available in the —also ISO compliant— `atom_codes/2` predicate. Safest repair is a replacement of all `atom_chars` into `atom_codes`. If you do not want to change any souce-code, you might want to use

```
user:goal_expansion(atom_chars(A,B), atom_codes(A,B)).
```

- `number_chars/2`
  Same applies for `number_chars/2` and `number_codes/2`.
- `feature/2`, `set_feature/2`
  These are replaced by the ISO compliant `current_prolog_flag/2` and `set_prolog_flag/2`. The library library(`backcomp`) provides definitions for `feature/2` and `set_feature/2`, so no source **has** to be updated.
- *Accessing command-line arguments*
  This used to be provided by the undocumented '$argv'/1 and Quintus compatible library `unix/1`. Now there is also documented `current_prolog_flag`(*argv*, *Argv*).
- `dup_stream/2`
  Has been deleted. New stream-aliases can deal with most of the problems for which `dup_stream/2` was designed and `dup/2` from the *clib* package can with most others.
- `op/3`
  Operators are now **local to modules**. This implies any modification of the operator-table does not influence other modules. This is consistent with the proposed ISO behaviour and a necessity to have any usable handling of operators in a multi-threaded environment.
- *set_prolog_flag(character_escapes, Bool)*
  This prolog flag is now an interface to changing attributes on the current source-module, effectively making this flag module-local as well. This is required for consistent handling of sources written with ISO (obligatory) character-escape sequences together with old Edinburgh code.
- `current_stream/3` *and stream_position*
  These predicates have been moved to library(`quintus`).

1.6.12 Version 3.4 Release Notes

The 3.4 release is a consolidation release. It consolidates the improvements and standard conformance of the 3.3 releases. This version is closely compatible with the 3.3 version except for one important change:

- *Argument order in* `select/3`
  The list-processing predicate `select/3` somehow got into a very early version of SWI-Prolog with the wrong argument order. This has been fixed in 3.4.0. The correct order is select(?Elem, ?List, ?Rest).

  As `select/3` has no error conditions, runtime checking cannot be done. To simplify debugging, the library module library(`checkselect`) will print references to `select/3` in your source code and install a version of select that enters the debugger if select is called and the second argument is not a list.

  This library can be loaded explicitely or by calling `check_old_select/0`.

1.6.13 Version 4.0 Release Notes

As of version 4.0 the standard distribution of SWI-Prolog is bundled with a number of its popular extension packages, among which the now open source XPCE GUI toolkit (see section 1.5). No significant changes have been made to the basic SWI-Prolog engine.

Some useful tricks in the integrated environment:

- *Register the GUI tracer*
  Using a call to `guitracer/0`, hooks are installed that replace the normal command-line driven tracer with a graphical forntend.
- *Register PceEmacs for editing files*
  From your initialisation file. you can load library(`emacs/swi_prolog`) that cause `edit/1` to use the built-in PceEmacs editor.

1.7 Acknowledgements

Some small parts of the Prolog code of SWI-Prolog are modified versions of the corresponding Edinburgh C-Prolog code: grammar rule compilation and `writef/2`. Also some of the C-code originates from C-Prolog: finding the path of the currently running executable and the code underlying `absolute_file_name/2`. Ideas on programming style and techniques originate from C-Prolog and Richard O'Keefe's *thief* editor. An important source of inspiration are the programming techniques introduced by Anjo Anjewierden in PCE version 1 and 2.

I also would like to thank those who had the fade of using the early versions of this system, suggested extensions or reported bugs. Among them are Anjo Anjewierden, Huub Knops, Bob Wielinga, Wouter Jansweijer, Luc Peerdeman, Eric Nombden, Frank van Harmelen, Bert Rengel.

Martin Jansche (`jansche@novell1.gs.uni-heidelberg.de`) has been so kind to reorganise the sources for version 2.1.3 of this manual.

Horst von Brand has been so kind to fix many typos in the 2.7.14 manual. Thanks!

Overview 2

2.1 Getting started quickly

2.1.1 Starting SWI-Prolog

Starting SWI-Prolog on Unix

By default, SWI-Prolog is installed as ‘pl’, though some administrators call it ‘swipl’ or ‘swi-prolog’. The command-line arguments of SWI-Prolog itself and its utility programs are documented using standard Unix `man` pages. SWI-Prolog is normally operated as an interactive application simply by starting the program:

```
machine% pl
% /staff/jan/.plrc compiled 0.00 sec, 1,260 bytes
Welcome to SWI-Prolog (Version 3.4.0)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
```

After starting Prolog, one normally loads a program into it using `consult/1`, which—for historical reasons—may be abbreviated by putting the name of the program file between square brackets. The following goal loads the file `likes.pl` containing clauses for the predicates `likes/2`:

```
?- [likes].
% likes compiled, 0.00 sec, 596 bytes.

Yes
?-
```

After this point, Unix and Windows users are united again.

Starting SWI-Prolog on Windows

After SWI-Prolog has been installed on a Windows system, the following important new things are available to the user:

- A folder (called *directory* in the remainder of this document) called `pl` containing the executables, libraries, etc. of the system. No files are installed outside this directory.

- A program `plwin.exe`, providing a window for interaction with Prolog. The program `plcon.exe` is a version of SWI-Prolog that runs in a DOS-box.
- The file-extension `.pl` is associated with the program `plwin.exe`. Opening a `.pl` file will cause `plwin.exe` to start, change directory to the directory in which the file-to-open resides and load this file.

The normal way to start with the `likes.pl` file mentioned in section 2.1.1 is by simply double-clicking this file in the Windows explorer.

2.1.2 Executing a query

After loading a program, one can ask Prolog queries about the program. The query below asks Prolog to prove whether 'john' likes someone and who is liked by 'john'. The system responds with `X =` ⟨*value*⟩ if it can prove the goal for a certain *X*. The user can type the semi-colon (;) if (s)he wants another solution, or RETURN if (s)he is satisfied, after which Prolog will say **Yes**. If Prolog answers **No**, it indicates it cannot find any more answers to the query. Finally, Prolog can answer using an error message to indicate the query or program contains an error.

```
?- likes(john, X).

X = mary
```

2.2 The user's initialisation file

After the necessary system initialisation the system consults (see `consult/1`) the user's startup file. The base-name of this file follows conventions of the operating system. On MS-Windows, it is the file `pl.ini` and on Unix systems `.plrc`. The file is searched using the `file_search_path/2` clauses for `user_profile`. The table below shows the default value for this search-path.

	Unix	**Windows**
local	`.`	`.`
home	`~`	`%HOME%` or `%HOMEDRIVE%\%HOMEPATH%`
global		SWI-Home directory or `%WINDIR%` or `%SYSTEMROOT%`

After the first startup file is found it is loaded and Prolog stops looking for further startup files. The name of the startup file can be changed with the '`-f` *`file`*' option. If *File* denotes an absolute path, this file is loaded, otherwise the file is searched for using the same conventions as for the default startup file. Finally, if *file* is `none`, no file is loaded.

2.3 Initialisation files and goals

Using commandline arguments (see section 2.4), SWI-Prolog can be forced to load files and execute queries for initialisation purposes or non-interactive operation. The most commonly used options are `-f` *`file`* to make Prolog load an initialisation file, `-g` *`goal`* to define an initialisation goal and `-t` *`goal`* to define the *toplevel goal*. The following is a typical example for starting an application directly from the commandline.

```
machine% pl -f load.pl -g go -t halt
```

It tells SWI-Prolog to load `load.pl`, start the application using the *entry-point* `go/0` and —instead of entering the interactive toplevel— exit after completing `go/0`.

In MS-Windows, the same can be achieved using a short-cut with appropriately defined commandline arguments. A typically seen alternative is to write a file `run.pl` with content as illustrated below. Double-clicking `run.pl` will start the application.

```
:- [load].                      % load program
:- go.                          % run it
:- halt.                        % and exit
```

Chapter 6 discusses the generation of runtime executables. Runtime executables are a mean to deliver executables that do not require the Prolog system.

2.4 Command line options

The full set of command line options is given below:

-help
When given as the only option, it summarises the most important options.

-v
When given as the only option, it summarises the version and the architecture identifier.

-arch
When given as the only option, it prints the architecture identifier (see current_prolog_flag(arch, Arch)) and exits. See also `-dump-runtime-variables`.

-dump-runtime-variables
When given as the only option, it prints a sequence of variable settings that can be used in shell-scripts to deal with Prolog parameters. This feature is also used by `plld` (see section 5.7). Below is a typical example of using this feature.

```
eval ‘pl -dump-runtime-variables‘
cc -I$PLBASE/include -L$PLBASE/runtime/$PLARCH ...
```

-L*size[km]*
Give local stack limit (2 Mbytes default). Note that there is no space between the size option and its argument. By default, the argument is interpreted in Kbytes. Postfixing the argument with `m` causes the argument to be interpreted in Mbytes. The following example specifies 32 Mbytes local stack.

```
% pl -L32m
```

A maximum is useful to stop buggy programs from claiming all memory resources. `-L0` sets the limit to the highest possible value. See section 2.16.

-G*size[km]*
Give global stack limit (4 Mbytes default). See `-L` for more details.

-T*size[km]*
Give trail stack limit (4 Mbytes default). This limit is relatively high because trail-stack overflows are not often caused by program bugs. See `-L` for more details.

-A*size[km]*
Give argument stack limit (1 Mbytes default). The argument stack limits the maximum nesting of terms that can be compiled and executed. SWI-Prolog does 'last-argument optimisation' to avoid many deeply nested structure using this stack. Enlarging this limit is only necessary in extreme cases. See `-L` for more details.

-c *file . . .*
Compile files into an 'intermediate code file'. See section 2.10.

-o *output*
Used in combination with `-c` or `-b` to determine output file for compilation.

-O
Optimised compilation. See `current_prolog_flag/2`.

-f *file*
Use *file* as startup file instead of the default. '`-f` *`none`*' stops SWI-Prolog from searching for a startup file. See section 2.2.

-F *script*
Selects a startup-script from the SWI-Prolog home directory. The script-file is named ⟨*script*⟩`.rc`. The default *script* name is deduced from the executable, taking the leading alphanumerical characters (letters, digits and underscore) from the program-name. `-F` *`none`* stops looking for a script. Intended for simple management of slightly different versions. One could for example write a script `iso.rc` and then select ISO compatibility mode using `pl -F iso` or make a link from `iso-pl` to `pl`.

-g *goal*
Goal is executed just before entering the top level. Default is a predicate which prints the welcome message. The welcome message can thus be suppressed by giving `-g` *`true`*. *goal* can be a complex term. In this case quotes are normally needed to protect it from being expanded by the Unix shell.

-t *goal*
Use *goal* as interactive toplevel instead of the default goal `prolog/0`. *goal* can be a complex term. If the toplevel goal succeeds SWI-Prolog exits with status 0. If it fails the exit status is 1. This flag also determines the goal started by `break/0` and `abort/0`. If you want to stop the user from entering interactive mode start the application with '`-g` *`goal`*' and give 'halt' as toplevel.

-tty
Unix only. Switches controlling the terminal for allowing single-character commands to the tracer and `get_single_char/1`. By default manipulating the terminal is enabled unless the

system detects it is not connected to a terminal or it is running as a GNU-Emacs inferior process. This flag is sometimes required for smooth interaction with other applications.

-x *bootfile*
: Boot from *bootfile* instead of the system's default boot file. A bootfile is a file resulting from a Prolog compilation using the `-b` or `-c` option or a program saved using `qsave_program/[1,2]`.

-p *alias=path1[:path2 ...]*
: Define a path alias for file_search_path. *alias* is the name of the alias, *path1* ... is a `:` separated list of values for the alias. A value is either a term of the form alias(value) or pathname. The computed aliases are added to `file_search_path/2` using `asserta/1`, so they precede predefined values for the alias. See `file_search_path/2` for details on using this file-location mechanism.

`--`
: Stops scanning for more arguments, so you can pass arguments for your application after this one. See `current_prolog_flag/2` using the flag `argv` for obtaining the commandline arguments.

The following options are for system maintenance. They are given for reference only.

-b *initfile* ... `-c` *file* ...
: Boot compilation. *initfile* ... are compiled by the C-written bootstrap compiler, *file* ... by the normal Prolog compiler. System maintenance only.

-d *level*
: Set debug level to *level*. Only has effect if the system is compiled with the `-DO_DEBUG` flag. System maintenance only.

2.5 GNU Emacs Interface

The default Prolog mode for GNU-Emacs can be activated by adding the following rules to your Emacs initialisation file:

```
(setq auto-mode-alist
      (append
       '(("\\.pl" . prolog-mode))
       auto-mode-alist))
(setq prolog-program-name "pl")
(setq prolog-consult-string "[user].\n")
;If you want this.  Indentation is either poor or I don't use
;it as intended.
;(setq prolog-indent-width 8)
```

Unfortunately the default Prolog mode of GNU-Emacs is not very good. An alternative `prolog.el` file for GNU-Emacs 20 is available from `http://www.freesoft.cz/ pdm/software/emacs/prolog-mode/` and for GNU-Emacs 19 from `http://w1.858.telia.com/ u85810764/Prolog-mode/index.html`

2.6 Online Help

Online help provides a fast lookup and browsing facility to this manual. The online manual can show predicate definitions as well as entire sections of the manual.

The online help is displayed from the file library(`'MANUAL'`). The file library(`helpidx`) provides an index into this file. library(`'MANUAL'`) is created from the LaTeX sources with a modified version of `dvitty`, using overstrike for printing bold text and underlining for rendering italic text. XPCE is shipped with library(`swi_help`), presenting the information from the online help in a hypertext window. The prolog-flag `write_help_with_overstrike` controls whether or not `help/1` writes its output using overstrike to realise bold and underlined output or not. If this prolog-flag is not set it is initialised by the help library to `true` if the `TERM` variable equals `xterm` and `false` otherwise. If this default does not satisfy you, add the following line to your personal startup file (see section 2.2):

```
:- set_prolog_flag(write_help_with_overstrike, true).
```

help

Equivalent to `help(help/1)`.

help(+*What***)**

Show specified part of the manual. *What* is one of:

⟨*Name*⟩/⟨*Arity*⟩	Give help on specified predicate
⟨*Name*⟩	Give help on named predicate with any arity or C interface function with that name
⟨*Section*⟩	Display specified section. Section numbers are dash-separated numbers: `2-3` refers to section 2.3 of the manual. Section numbers are obtained using `apropos/1`.

Examples:

`?- help(assert).`	Give help on predicate assert
`?- help(3-4).`	Display section 3.4 of the manual
`?- help('PL_retry').`	Give help on interface function `PL_retry()`

See also `apropos/1`, and the SWI-Prolog home page at `http://www.swi.psy.uva.nl/projects/SWI-Prolog/`, which provides a FAQ, an HTML version of manual for online browsing and HTML and PDF versions for downloading.

apropos(+*Pattern***)**

Display all predicates, functions and sections that have *Pattern* in their name or summary description. Lowercase letters in *Pattern* also match a corresponding uppercase letter. Example:

`?- apropos(file).`	Display predicates, functions and sections that have ‘file’ (or ‘File’, etc.) in their summary description.

`!!.`	Repeat last query
`!nr.`	Repeat query numbered ⟨*nr*⟩
`!str.`	Repeat last query starting with ⟨*str*⟩
`!?str.`	Repeat last query holding ⟨*str*⟩
`^old^new.`	Substitute ⟨*old*⟩ into ⟨*new*⟩ in last query
`!nr^old^new.`	Substitute in query numbered ⟨*nr*⟩
`!str^old^new.`	Substitute in query starting with ⟨*str*⟩
`!?str^old^new.`	Substitute in query holding ⟨*str*⟩
`h.`	Show history list
`!h.`	Show this list

Table 2.1: History commands

explain(+*ToExplain***)**

Give an explanation on the given 'object'. The argument may be any Prolog data object. If the argument is an atom, a term of the form *Name/Arity* or a term of the form *Module:Name/Arity*, explain will try to explain the predicate as well as possible references to it.

explain(+*ToExplain*, -*Explanation***)**

Unify *Explanation* with an explanation for *ToExplain*. Backtracking yields further explanations.

2.7 Query Substitutions

SWI-Prolog offers a query substitution mechanism similar to that of Unix csh (csh(1)), called 'history'. The availability of this feature is controlled by `set_prolog_flag/2`, using the `history` prolog-flag. By default, history is available if the prolog-flag `readline` is `false`. To enable this feature, remembering the last 50 commands, put the following into your startup file (see section 2.2:

```
:- set_prolog_flag(history, 50).
```

The history system allows the user to compose new queries from those typed before and remembered by the system. It also allows to correct queries and syntax errors. SWI-Prolog does not offer the Unix csh capabilities to include arguments. This is omitted as it is unclear how the first, second, etc. argument should be defined.[1]

The available history commands are shown in table 2.1.

2.7.1 Limitations of the History System

History expansion is executed after *raw-reading*. This is the first stage of `read_term/2` and friends, reading the term into a string while deleting comment and canonising blank. This makes it hard to use it for correcting syntax errors. Command-line editing as provided using the GNU-readline library is more suitable for this. History expansion is first of all useful for executing or combining commands from long ago.

[1]One could choose words, defining words as a sequence of alpha-numeric characters and the word separators as anything else, but one could also choose Prolog arguments

```
1 ?- maplist(plus(1), "hello", X).

X = [105,102,109,109,112]

Yes
2 ?- format('~s~n', [$X]).
ifmmp

Yes
3 ?-
```

Figure 2.1: Reusing toplevel bindings

2.8 Reuse of toplevel bindings

Bindings resulting from the successful execution of a toplevel goal are asserted in a database. These values may be reused in further toplevel queries as $Var. Only the latest binding is available. Example:

Note that variables may be set by executing =/2:

```
6 ?- X = statistics.

X = statistics

Yes
7 ?- $X.
28.00 seconds cpu time for 183,128 inferences
4,016 atoms, 1,904 functors, 2,042 predicates, 52 modules
55,915 byte codes; 11,239 external references

                      Limit    Allocated       In use
Heap         :                              624,820 Bytes
Local  stack :    2,048,000        8,192        404 Bytes
Global stack :    4,096,000       16,384        968 Bytes
Trail  stack :    4,096,000        8,192        432 Bytes

Yes
8 ?-
```

2.9 Overview of the Debugger

SWI-Prolog has a 6-port tracer, extending the standard 4-port tracer [Clocksin & Melish, 1987] with two additional ports. The optional *unify* port allows the user to inspect the result after unification of the head. The *exception* port shows exceptions raised by throw/1 or one of the built-in predicates. See section 3.9.

```
1 ?- visible(+all), leash(-exit).

Yes
2 ?- trace, min([3, 2], X).
  Call:  ( 3) min([3, 2], G235) ? creep
  Unify: ( 3) min([3, 2], G235)
  Call:  ( 4) min([2], G244) ? creep
  Unify: ( 4) min([2], 2)
  Exit:  ( 4) min([2], 2)
  Call:  ( 4) min(3, 2, G235) ? creep
  Unify: ( 4) min(3, 2, G235)
  Call:  ( 5) 3 < 2 ? creep
  Fail:  ( 5) 3 < 2 ? creep
  Redo:  ( 4) min(3, 2, G235) ? creep
  Exit:  ( 4) min(3, 2, 2)
  Exit:  ( 3) min([3, 2], 2)

Yes
[trace] 3 ?-
```

Figure 2.2: Example trace

The standard ports are called `call`, `exit`, `redo`, `fail` and `unify`. The tracer is started by the `trace/0` command, when a spy point is reached and the system is in debugging mode (see `spy/1` and `debug/0`) or when an exception is raised.

The interactive toplevel goal `trace/0` means "trace the next query". The tracer shows the port, displaying the port name, the current depth of the recursion and the goal. The goal is printed using the Prolog predicate `write_term/2`. The style is defined by the prolog-flag `debugger_print_options` and can be modified using this flag or using the `w`, `p` and `d` commands of the tracer.

On *leashed ports* (set with the predicate `leash/1`, default are `call`, `exit`, `redo` and `fail`) the user is prompted for an action. All actions are single character commands which are executed **without** waiting for a return, unless the command line option `-tty` is active. Tracer options:

+ (Spy)
Set a spy point (see `spy/1`) on the current predicate.

- (No spy)
Remove the spy point (see `nospy/1`) from the current predicate.

/ (Find)
Search for a port. After the '/', the user can enter a line to specify the port to search for. This line consists of a set of letters indicating the port type, followed by an optional term, that should unify with the goal run by the port. If no term is specified it is taken as a variable, searching for any port of the specified type. If an atom is given, any goal whose functor has a name equal to that atom matches. Examples:

`/f`	Search for any fail port
`/fe solve`	Search for a fail or exit port of any goal with name `solve`
`/c solve(a, _)`	Search for a call to solve/2 whose first argument is a variable or the atom `a`
`/a member(_, _)`	Search for any port on `member/2`. This is equivalent to setting a spy point on `member/2`.

`.` **(Repeat find)**
Repeat the last find command (see '/').

`A` **(Alternatives)**
Show all goals that have alternatives.

`C` **(Context)**
Toggle 'Show Context'. If `on` the context module of the goal is displayed between square brackets (see section 4). Default is `off`.

`L` **(Listing)**
List the current predicate with `listing/1`.

`a` **(Abort)**
Abort Prolog execution (see `abort/0`).

`b` **(Break)**
Enter a Prolog break environment (see `break/0`).

`c` **(Creep)**
Continue execution, stop at next port. (Also return, space).

`d` **(Display)**
Set the `max_depth`(*Depth*) option of `debugger_print_options`, limiting the depth to which terms are printed. See also the `w` and `p` options.

`e` **(Exit)**
Terminate Prolog (see `halt/0`).

`f` **(Fail)**
Force failure of the current goal.

`g` **(Goals)**
Show the list of parent goals (the execution stack). Note that due to tail recursion optimization a number of parent goals might not exist any more.

`h` **(Help)**
Show available options (also '?').

`i` **(Ignore)**
Ignore the current goal, pretending it succeeded.

`l` **(Leap)**
Continue execution, stop at next spy point.

`n` **(No debug)**
: Continue execution in ‘no debug’ mode.

`p` **(Print)**
: Set the prolog-flag `debugger_print_options` to `[quoted(true), portray(true), max_depth(1`
This is the default.

`r` **(Retry)**
: Undo all actions (except for database and i/o actions) back to the call port of the current goal and resume execution at the call port.

`s` **(Skip)**
: Continue execution, stop at the next port of **this** goal (thus skipping all calls to children of this goal).

`u` **(Up)**
: Continue execution, stop at the next port of **the parent** goal (thus skipping this goal and all calls to children of this goal). This option is useful to stop tracing a failure driven loop.

`w` **(Write)**
: Set the prolog-flag `debugger_print_options` to `[quoted(true)]`, bypassing `portray/1`, etc.

The ideal 4 port model as described in many Prolog books [Clocksin & Melish, 1987] is not visible in many Prolog implementations because code optimisation removes part of the choice- and exit-points. Backtrack points are not shown if either the goal succeeded deterministically or its alternatives were removed using the cut. When running in debug mode (`debug/0`) choice points are only destroyed when removed by the cut. In debug mode, tail recursion optimisation is switched off.[2]

Reference information to all predicates available for manipulating the debugger is in section 3.42.

2.10 Compilation

2.10.1 During program development

During program development, programs are normally loaded using `consult/1`, or the list abbreviation. It is common practice to organise a project as a collection of source-files and a *load-file*, a Prolog file containing only `use_module/[1,2]` or `ensure_loaded/1` directives, possibly with a definition of the *entry-point* of the program, the predicate that is normally used to start the program. This file is often called `load.pl`. If the entry-point is called *go*, a typical session starts as:

```
% pl
<banner>

1 ?- [load].
<compilation messages>
```

[2]This implies the system can run out of local stack in debug mode, while no problems arise when running in non-debug mode.

```
Yes
2 ?- go.
<program interaction>
```

When using Windows, the user may open `load.pl` from the Windows explorer, which will cause `plwin.exe` to be started in the directory holding `load.pl`. Prolog loads `load.pl` before entering the toplevel.

2.10.2 For running the result

There are various options if you want to make your program ready for real usage. The best choice depends on whether the program is to be used only on machines holding the SWI-Prolog development system, the size of the program and the operating system (Unix vs. Windows).

Creating a shell-script

Especially on Unix systems and not-too-large applications, writing a shell-script that simply loads your application and calls the entry-point is often a good choice. A skeleton for the script is given below, followed by the Prolog code to obtain the program arguments.

```
#!/bin/sh

base=<absolute-path-to-source>
PL=pl

exec $PL -f none -g "load_files(['$base/load'],[silent(true)])" \
         -t go -- $*
```

```
go :-
        current_prolog_flag(argv, Arguments),
        append(_SytemArgs, [--|Args], Arguments), !,
        go(Args).

go(Args) :-
        ...
```

On Windows systems, similar behaviour can be achieved by creating a shortcut to Prolog, passing the proper options or writing a `.bat` file.

Creating a saved-state

For larger programs, as well as for programs that are required run on systems that do not have the SWI-Prolog development system installed, creating a saved state is the best solution. A saved state is created using `qsave_program/[1,2]` or using the linker plld(1). A saved state is a file containing machine-independent intermediate code in a format dedicated for fast loading. Optionally, the emulator may be integrated in the saved state, creating a single-file, but machine-dependent, executable. This process is described in chapter 6.

Compilation using the -c commandline option

This mechanism loads a series of Prolog source files and then creates a saved-state as `qsave_program/2` does. The command syntax is:

```
% pl [option ...] [-o output] -c file ...
```

The *options* argument are options to `qsave_program/2` written in the format below. The option-names and their values are described with `qsave_program/2`.

`--`*option-name*=*option-value*

For example, to create a stan-alone executable that starts by executing `main/0` and for which the source is loaded through `load.pl`, use the command

```
% pl --goal=main --stand_alone=true -o myprog -c load.pl
```

This performs exactly the same as executing

```
% pl
<banner>
?- [load].
?- qsave_program(myprog,
                 [ goal(main),
                   stand_alone(true)
                 ]).
?- halt.
```

2.11 Environment Control (Prolog flags)

The predicates `current_prolog_flag/2` and `set_prolog_flag/2` allow the user to examine and modify the execution environment. It provides access to whether optional features are available on this version, operating system, foreign-code environment, command-line arguments, version, as well as runtime flags to control the runtime behaviour of certain predicates to achieve compatibility with other Prolog environments.

current_prolog_flag(*?Key*, *-Value***)**

The predicate `current_prolog_flag/2` defines an interface to installation features: options compiled in, version, home, etc. With both arguments unbound, it will generate all defined prolog-flags. With the 'Key' instantiated it unify the value of the prolog-flag. Features come in three types: boolean prolog-flags, prolog-flags with an atom value and prolog-flags with an integer value. A boolean prolog-flag is true iff the prolog-flag is present **and** the *Value* is the atom `true`. Currently defined keys:

arch *(atom)*

Identifier for the hardware and operating system SWI-Prolog is running on. Used to determine the startup file as well as to select foreign files for the right architecture. See also section 5.4.

version *(integer)*
The version identifier is an integer with value:

$$10000 \times \mathit{Major} + 100 \times \mathit{Minor} + \mathit{Patch}$$

Note that in releases upto 2.7.10 this prolog-flag yielded an atom holding the three numbers separated by dots. The current representation is much easier for implementing version-conditional statements.

home *(atom)*
SWI-Prolog's notion of the home-directory. SWI-Prolog uses it's home directory to find its startup file as ⟨*home*⟩`/startup/startup.`⟨*arch*⟩ and to find its library as ⟨*home*⟩`/library`.

executable *(atom)*
Path-name of the running executable. Used by `qsave_program/2` as default emulator.

argv *(list)*
List is a list of atoms representing the command-line arguments used to invoke SWI-Prolog. Please note that **all** arguments are included in the list returned.

pipe *(bool, changeable)*
If true, `open(pipe(command), mode, Stream)`, etc. are supported. Can be changed to disable the use of pipes in applications testing this feature. Not recommended.

open_shared_object *(bool)*
If true, `open_shared_object/2` and friends are implemented, providing access to shared libraries (`.so` files) or dynamic link libraries (`.DLL` files).

shared_object_extension *(atom)*
Extension used by the operating system for shared objects. `so` for most Unix systems and `dll` for Windows. Used for locating files using the `file_type executable`. See also `absolute_file_name/3`.

dynamic_stacks *(bool)*
If `true`, the system uses some form of 'sparse-memory management' to realise the stacks. If false, malloc()/realloc() are used for the stacks. In earlier days this had consequenses for foreign code. As of version 2.5, this is no longer the case.

Systems using 'sparse-memory management' are a bit faster as there is no stack-shifter, and checking the stack-boundary is often realised by the hardware using a 'guard-page'. Also, memory is actually returned to the system after a garbage collection or call to `trim_stacks/0` (called by `prolog/0` after finishing a user-query).

c_libs *(atom)*
Libraries passed to the C-linker when SWI-Prolog was linked. May be used to determine the libraries needed to create statically linked extensions for SWI-Prolog. See section 5.7.

c_cc *(atom)*
Name of the C-compiler used to compile SWI-Prolog. Normally either gcc or cc. See section 5.7.

c_ldflags *(atom)*
Special linker flags passed to link SWI-Prolog. See section 5.7.

readline *(bool)*
: If true, SWI-Prolog is linked with the readline library. This is done by default if you have this library installed on your system. It is also true for the Win32 plwin.exe version of SWI-Prolog, which realises a subset of the readline functionality.

saved_program *(bool)*
: If true, Prolog is started from a state saved with `qsave_program/[1,2]`.

runtime *(bool)*
: If true, SWI-Prolog is compiled with -DO_RUNTIME, disabling various useful development features (currently the tracer and profiler).

max_integer *(integer)*
: Maximum integer value. Most arithmetic operations will automatically convert to floats if integer values above this are returned.

min_integer *(integer)*
: Minimum integer value.

max_tagged_integer *(integer)*
: Maximum integer value represented as a 'tagged' value. Tagged integers require 4-bytes storage and are used for indexing. Larger integers are represented as 'indirect data' and require 16-bytes on the stacks (though a copy requires only 4 additional bytes).

min_tagged_integer *(integer)*
: Start of the tagged-integer value range.

float_format *(atom, changeable)*
: C `printf()` format specification used by `write/1` and friends to determine how floating point numbers are printed. The default is `%g`. The specified value is passed to printf() without further checking. For example, if you want more digits printed, `%.12g` will print all floats using 12 digits instead of the default 6. See also `format/[1,2]`, `write/1`, `print/1` and `portray/1`.

toplevel_print_options *(term, changeable)*
: This argument is given as option-list to `write_term/2` for printing results of queries. Default is `[quoted(true), portray(true), max_depth(10)]`.

debugger_print_options *(term, changeable)*
: This argument is given as option-list to `write_term/2` for printing goals by the debugger. Modified by the 'w', 'p' and '⟨*N*⟩ d' commands of the debugger. Default is `[quoted(true), portray(true), max_depth(10)]`.

debugger_show_context *(bool, changeable)*
: If `true`, show the context module while printing a stack-frame in the tracer. Normally controlled using the 'C' option of the tracer.

compiled_at *(atom)*
: Describes when the system has been compiled. Only available if the C-compiler used to compile SWI-Prolog provides the __DATE__ and __TIME__ macros.

character_escapes *(bool, changeable)*
: If `true` (default), `read/1` interprets \ escape sequences in quoted atoms and strings. May be changed. This flag is local to the module in which it is changed.

double_quotes *(codes,chars,atom,string, changeable)*
: This flag determines how double-quotes strings are read by Prolog and is —like character_escapes— maintained for each module. If `codes` (default), a list of character-codes is returned, if `chars` a list of one-character atoms, if `atom` double quotes are the same as single-quotes and finally, `string` reads the text into a Prolog string (see section 3.23). See also `atom_chars/2` and `atom_codes/2`.

allow_variable_name_as_functor *(bool, changeable)*
: If true (default is false), `Functor(arg)` is read as if it was written `'Functor'(arg)`. Some applications use the Prolog `read/1` predicate for reading an application defined script language. In these cases, it is often difficult to explain none-Prolog users of the application that constants and functions can only start with a lowercase letter. Variables can be turned into atoms starting with an uppercase atom by calling `read_term/2` using the option `variable_names` and binding the variables to their name. Using this feature, F(x) can be turned into valid syntax for such script languages. Suggested by Robert van Engelen. SWI-Prolog specific.

history *(integer, changeable)*
: If *integer* > 0, support Unix `csh(1)` like history as described in section 2.7. Otherwise, only support reusing commands through the commandline editor. The default is to set this prolog-flag to 0 if a commandline editor is provided (see prolog-flag `readline`) and 15 otherwise.

gc *(bool, changeable)*
: If true (default), the garbage collector is active. If false, neither garbage-collection, nor stack-shifts will take place, even not on explicit request. May be changed.

agc_margin *(integer, changeable)*
: If this amount of atoms has been created since the last atom-garbage collection, perform atom garbage collection at the first opportunity. Initial value is 10,000. May be changed. A value of 0 (zero) disables atom garbage collection. See also `PL_register_atom()`.

iso *(bool, changeable)*
: Include some weird ISO compatibility that is incompatible to normal SWI-Prolog behaviour. Currently it has the following effect:

  - `is/2` and evaluation under `flag/3` do not automatically convert floats to integers if the float represents an integer.
  - The `//2` (float division) *always* return a float, even if applied to integers that can be divided.
  - In the standard order of terms (see section 3.6.1), all floats are before all integers.
  - `atom_length/2` yields an instantiation error if the first argument is a number.
  - `clause/[2,3]` raises a permission error when accessing static predicates.
  - `abolish/[1,2]` raises a permission error when accessing static predicates.

optimise *(bool, changeable)*
: If `true`, compile in optimised mode. The initial value is `true` if Prolog was started with the `-O` commandline option.

  Currently optimise compilation implies compilation of arithmetic, and deletion of redundant `true/0` that may result from `expand_goal/2`.

  Later versions might imply various other optimisations such as integrating small predicates into their callers, eliminating constant expressions and other predictable constructs.

Source code optimisation is never applied to predicates that are declared dynamic (see `dynamic/1`).

char_conversion *(bool, changeable)*
Determines whether character-conversion takes place while reading terms. See also `char_conversion/2`.

autoload *(bool, changeable)*
If `true` (default) autoloading of library functions is enabled. See section 2.13.

verbose_autoload *(bool, changeable)*
If `true` the normal consult message will be printed if a library is autoloaded. By default this message is suppressed. Intended to be used for debugging purposes.

trace_gc *(bool, changeable)*
If true (false is the default), garbage collections and stack-shifts will be reported on the terminal. May be changed.

max_arity *(unbounded)*
ISO prolog-flag describing there is no maximum arity to compound terms.

integer_rounding_function *(down,toward_zero)*
SO prolog-flag describing rounding by `//` and `rem` arithmetic functions. Value depends on the C-compiler used.

bounded *(true)*
ISO prolog-flag describing integer representation is bound by `min_integer` and `min_integer`.

tty_control *(bool)*
Determines whether the terminal is switched to raw mode for `get_single_char/1`, which also reads the user-actions for the trace. May be set. See also the `+/-tty` command-line option.

unknown *(fail,warning,error, changeable)*
Determines the behaviour if an undefined procedure is encountered. If `fail`, the predicates fails silently. If `warn`, a warning is printed, and execution continues as if the predicate was not defined and if `error` (default), an `existence_error` exception is raised. This flag is local to each module.

debug *(bool, changeable)*
Switch on/off debugging mode. If debug mode is activated the system traps encountered spy-points (see `spy/1`) and trace-points (see `trace/1`). In addition, tail-recursion optimisation is disabled and the system is more conservative in destroying choice-points to simplify debugging.

Disabling these optimisations can cause the system to run out of memory on programs that behave correctly if debug mode is off.

tail_recursion_optimisation *(bool, changeable)*
Determines whether or not tail-recursion optimisation is enabled. Normally the value of this flag is equal to the `debug` flag. As programs may run out of stack if tail-recursion optimisation is omitted, it is sometimes necessary to enable it during debugging.

abort_with_exception *(bool, changeable)*
Determines how `abort/0` is realised. See the description of `abort/0` for details.

debug_on_error *(bool, changeable)*
If `true`, start the tracer after an error is detected. Otherwise just continue execution. The goal that raised the error will normally fail. See also `fileerrors/2` and the prolog-flag `report_error`. May be changed. Default is `true`, except for the runtime version.

report_error *(bool, changeable)*
If `true`, print error messages, otherwise suppress them. May be changed. See also the `debug_on_error` prolog-flag. Default is `true`, except for the runtime version.

file_name_variables *(bool, changeable)*
If `true` (default `false`), expand `$varname` and ˜ in arguments of builtin-predicates that accept a file name (`open/3`, `exists_file/1`, `access_file/2`, etc.). The predicate `expand_file_name/2` should be used to expand environment variables and wildcard patterns. This prolog-flag is intended for backward compatibility with older versions of SWI-Prolog.

unix *(bool)*
If `true`, the operating system is some version of Unix. Defined if the C-compiler used to compile this version of SWI-Prolog either defines `__unix__` or `unix`.

windows *(bool)*
If `true`, the operating system is an implementation of Microsoft Windows (3.1, 95, NT, etc.).

set_prolog_flag(*+Key, +Value***)**
Define a new prolog-flag or change its value. *Key* is an atom. If the flag is a system-defined flag that is not marked *changeable* above, an attempt to modify the flag yields a `permission_error`. If the provided *Value* does not match the type of the flag, a `type_error` is raised.

In addition to ISO, SWI-Prolog allows for user-defined prolog flags. The type of the flag is determined from the initial value and cannot be changed afterwards.

2.12 An overview of hook predicates

SWI-Prolog provides a large number of hooks, mainly to control handling messages, debugging, startup, shut-down, macro-expansion, etc. Below is a summary of all defined hooks with an indication of their portability.

- `portray/1`
  Hook into `write_term/3` to alter the way terms are printed (ISO).

- `message_hook/3`
  Hook into `print_message/2` to alter the way system messages are printed (Quintus/SICStus).

- `library_directory/1`
  Hook into `absolute_file_name/3` to define new library directories. (most Prolog system).

- `file_search_path/2`
  Hook into `absolute_file_name/3` to define new search-paths (Quintus/SICStus).

- `term_expansion/2`
  Hook into `load_files/1` to modify read terms before they are compiled (macro-processing) (most Prolog system).
- `goal_expansion/2`
  Same as `term_expansion/2` for individual goals (SICStus).
- `prolog_edit:locate/3`
  Hook into `edit/1` to locate objects (SWI).
- `prolog_edit:edit_source/1`
  Hook into `edit/1` to call some internal editor (SWI).
- `prolog_edit:edit_command/2`
  Hook into `edit/1` to define the external editor to use (SWI).
- `prolog_list_goal/1`
  Hook into the tracer to list the code associated to a particular goal (SWI).
- `prolog_trace_interception/4`
  Hook into the tracer to handle trace-events (SWI).
- `resource/3`
  Defines a new resource (not really a hook, but similar) (SWI).
- `exception/3`
  Old attempt to a generic hook mechanism. Handles undefined predicates (SWI).

2.13 Automatic loading of libraries

If —at runtime— an undefined predicate is trapped the system will first try to import the predicate from the module's default module. If this fails the *auto loader* is activated. On first activation an index to all library files in all library directories is loaded in core (see `library_directory/1`). If the undefined predicate can be located in the one of the libraries that library file is automatically loaded and the call to the (previously undefined) predicate is resumed. By default this mechanism loads the file silently. The `current_prolog_flag/2` `verbose_autoload` is provided to get verbose loading. The prolog-flag `autoload` can be used to enable/disable the entire auto load system.

The auto-loader only works if the unknown flag (see `unknown/2`) is set to `trace` (default). A more appropriate interaction with this flag will be considered.

Autoloading only handles (library) source files that use the module mechanism described in chapter 4. The files are loaded with `use_module/2` and only the trapped undefined predicate will be imported to the module where the undefined predicate was called. Each library directory must hold a file `INDEX.pl` that contains an index to all library files in the directory. This file consists of lines of the following format:

```
index(Name, Arity, Module, File).
```

The predicate `make/0` scans the autoload libraries and updates the index if it exists, is writable and out-of-date. It is advised to create an empty file called `INDEX.pl` in a library directory meant for auto loading before doing anything else. This index file can then be updated by running the prolog `make_library_index/1` ('%' is the Unix prompt):

```
% mkdir ~/lib/prolog
% cd !$
% pl -g true -t 'make_library_index(.)'
```

If there are more than one library files containing the desired predicate the following search schema is followed:

1. If there is a library file that defines the module in which the undefined predicate is trapped, this file is used.
2. Otherwise library files are considered in the order they appear in the `library_directory/1` predicate and within the directory alphabetically.

make_library_index(+*Directory***)**
Create an index for this directory. The index is written to the file 'INDEX.pl' in the specified directory. Fails with a warning if the directory does not exist or is write protected.

2.14 Garbage Collection

SWI-Prolog version 1.4 was the first release to support garbage collection. Together with last-call optimisation this guarantees forward chaining programs do not waste infinite amounts of memory.

2.15 Syntax Notes

SWI-Prolog uses standard 'Edinburgh' syntax. A description of this syntax can be found in the Prolog books referenced in the introduction. Below are some non-standard or non-common constructs that are accepted by SWI-Prolog:

- `0'`⟨*char*⟩
  This construct is not accepted by all Prolog systems that claim to have Edinburgh compatible syntax. It describes the ASCII value of ⟨*char*⟩. To test whether `C` is a lower case character one can use `between(0'a, 0'z, C)`.
- `/* .../* ...*/ ...*/`
  The `/* ...*/` comment statement can be nested. This is useful if some code with `/* ...*/` comment statements in it should be commented out.

2.15.1 ISO Syntax Support

SWI-Prolog offers ISO compatible extensions to the Edinburgh syntax.

Character Escape Syntax

Within quoted atoms (using single quotes: `'`⟨*atom*⟩`'` special characters are represented using escape-sequences. An escape sequence is lead in by the backslash (\) character. The list of escape sequences is compatible with the ISO standard, but contains one extension and the interpretation of numerically specified characters is slightly more flexible to improve compatibility.

`\a`
: Alert character. Normally the ASCII character 7 (beep).

`\b`
: Backspace character.

`\c`
: No output. All input characters upto but not including the first non-layout character are skipped. This allows for the specification of pretty-looking long lines. For compatibility with Quintus Prolog. Nor supported by ISO. Example:

```
format('This is a long line that would look better if it was \c
       split across multiple physical lines in the input')
```

\⟨RETURN⟩
: No output. Skips input till the next non-layout character or to the end of the next line. Same intention as `\c` but ISO compatible.

`\f`
: Form-feed character.

`\n`
: Next-line character.

`\r`
: Carriage-return only (i.e. go back to the start of the line).

`\t`
: Horizontal tab-character.

`\v`
: Vertical tab-character (ASCII 11).

`\x23`
: Hexadecimal specification of a character. `23` is just an example. The 'x' may be followed by a maximum of 2 hexadecimal digits. The closing \ is optional. The code `\xa\3` emits the character 10 (hexadecimal 'a') followed by '3'. The code `\x201` emits 32 (hexadecimal '20') followed by '1'. According to ISO, the closing \ is obligatory and the number of digits is unlimited. The SWI-Prolog definition allows for ISO compatible specification, but is compatible with other implementations.

`\40`
: Octal character specification. The rules and remarks for hexadecimal specifications apply to octal specifications too, but the maximum allowed number of octal digits is 3.

\⟨*character*⟩
: Any character immediately preceded by a \ and not covered by the above escape sequences is copied verbatim. Thus, `'\\'` is an atom consisting of a single \ and `'\''` and `''''` both describe the atom with a single `'`.

Character escaping is only available if the `current_prolog_flag(character_escapes, true)` is active (default). See `current_prolog_flag/2`. Character escapes conflict with `writef/2` in two ways: `\40` is interpreted as decimal 40 by `writef/2`, but character escapes handling by read has already interpreted as 32 (40 octal). Also, `\l` is translated to a single 'l'. It is adviced to use the more widely supported `format/[2,3]` predicate instead. If you insist using writef, either switch `character_escapes` to `false`, or use double `\\`, as in `writef('\\l')`.

Syntax for non-decimal numbers

SWI-Prolog implements both Edinburgh and ISO representations for non-decimal numbers. According to Edinburgh syntax, such numbers are written as ⟨*radix*⟩`'`⟨*number*⟩, where ⟨*radix*⟩ is a number between 2 and 36. ISO defines binary, octal and hexadecimal numbers using `0[bxo]`⟨*number*⟩. For example: `A is 0b100 \/ 0xf00` is a valid expression. Such numbers are always unsigned.

2.16 System limits

2.16.1 Limits on memory areas

SWI-Prolog has a number of memory areas which are only enlarged to a certain limit. The default sizes for these areas should suffice for most applications, but big applications may require larger ones. They are modified by command line options. The table below shows these areas. The first column gives the option name to modify the size of the area. The option character is immediately followed by a number and optionally by a `k` or `m`. With `k` or no unit indicator, the value is interpreted in Kbytes (1024 bytes), with `m`, the value is interpreted in Mbytes (1024×1024 bytes).

The local-, global- and trail-stack are limited to 128 Mbytes on 32 bit processors, or more in general to $2^{\text{bits-per-long}-5}$ bytes.

The heap

With the heap, we refer to the memory area used by `malloc()` and friends. SWI-Prolog uses the area to store atoms, functors, predicates and their clauses, records and other dynamic data. As of SWI-Prolog 2.8.5, no limits are imposed on the addresses returned by `malloc()` and friends.

On some machines, the runtime stacks described above are allocated using 'sparse allocation'. Virtual space upto the limit is claimed at startup and committed and released while the area grows and shrinks. On Win32 platform this is realised using `VirtualAlloc()` and friends. On Unix systems this is realised using `mmap()`.

2.16.2 Other Limits

Clauses Currently the following limitations apply to clauses. The arity may not be more than 1024 and the number of variables should be less than 65536.

Atoms and Strings SWI-Prolog has no limits on the sizes of atoms and strings. `read/1` and its derivatives however normally limit the number of newlines in an atom or string to 5 to improve error detection and recovery. This can be switched off with `style_check/1`.

Option	Default	Area name	Description
`-L`	2M	**local stack**	The local stack is used to store the execution environments of procedure invocations. The space for an environment is reclaimed when it fails, exits without leaving choice points, the alternatives are cut of with the !/0 predicate or no choice points have been created since the invocation and the last subclause is started (tail recursion optimisation).
`-G`	4M	**global stack**	The global stack is used to store terms created during Prolog's execution. Terms on this stack will be reclaimed by backtracking to a point before the term was created or by garbage collection (provided the term is no longer referenced).
`-T`	4M	**trail stack**	The trail stack is used to store assignments during execution. Entries on this stack remain alive until backtracking before the point of creation or the garbage collector determines they are nor needed any longer.
`-A`	1M	**argument stack**	The argument stack is used to store one of the intermediate code interpreter's registers. The amount of space needed on this stack is determined entirely by the depth in which terms are nested in the clauses that constitute the program. Overflow is most likely when using long strings in a clause.

Table 2.2: Memory areas

Address space SWI-Prolog data is packed in a 32-bit word, which contains both type and value information. The size of the various memory areas is limited to 128 Mb for each of the areas, except for the program heap, which is not limited.

Integers Integers are 32-bit to the user, but integers upto the value of the `max_tagged_integer` prolog-flag are represented more efficiently.

Floats Floating point numbers are represented as native double precision floats, 64 bit IEEE on most machines.

2.16.3 Reserved Names

The boot compiler (see `-b` option) does not support the module system. As large parts of the system are written in Prolog itself we need some way to avoid name clashes with the user's predicates, database keys, etc. Like Edinburgh C-Prolog [Pereira, 1986] all predicates, database keys, etc. that should be hidden from the user start with a dollar (`$`) sign (see `style_check/1`).

3 Built-in predicates

3.1 Notation of Predicate Descriptions

We have tried to keep the predicate descriptions clear and concise. First the predicate name is printed in bold face, followed by the arguments in italics. Arguments are preceded by a '+', '-' or '?' sign. '+' indicates the argument is input to the predicate, '-' denotes output and '?' denotes 'either input or output'.[1] Constructs like '`op/3`' refer to the predicate 'op' with arity '3'.

3.2 Character representation

In traditional (Edinburgh-) Prolog, characters are represented using *character-codes*. Character codes are integer indices into a specific character set. Traditionally the character set was 7-bits US-ASCII. Since a long while 8-bit character sets are allowed, providing support for national character sets, of which iso-latin-1 (ISO 8859-1) is applicable to many western languages. Text-files are supposed to represent a sequence of character-codes.

ISO Prolog introduces three types, two of which are used for characters and one for accessing binary streams (see `open/4`). These types are:

- *code*
  A *character-code* is an integer representing a single character. As files may use multi-byte encoding for supporting different character sets (utf-8 encoding for example), reading a code from a text-file is in general not the same as reading a byte.

- *char*
  Alternatively, characters may be represented as *one-character-atoms*. This is a very natural representation, hiding encoding problems from the programmer as well as providing much easier debugging.

- *byte*
  Bytes are used for accessing binary-streams.

The current version of SWI-Prolog does not provide support for multi-byte character encoding. This implies for example that it is not capable of breaking a multi-byte encoded atom into characters. For SWI-Prolog, bytes and codes are the same and one-character-atoms are simple atoms containing one byte.

To ease the pain of these multiple representations, SWI-Prolog's built-in predicates dealing with character-data work as flexible as possible: they accept data in any of these formats as long as the interpretation is unambiguous. In addition, for output arguments that are instantiated, the character

[1] These marks do **not** suggest instantiation (e.g. var(+Var)).

is extracted before unification. This implies that the following two calls are identical, both testing whether the next input characters is an `a`.

```
peek_code(Stream, a).
peek_code(Stream, 97).
```

These multiple-representations are handled by a large number of built-in predicates, all of which are ISO-compatible. For converting betweem code and character there is `char_code/2`. For breaking atoms and numbers into characters are are `atom_chars/2`, `atom_codes/2`, `number_codes/2` and `number_chars/2`. For character I/O on streams there is `get_char/[1,2]`, `get_code/[1,2]`, `get_byte/[1,2]`, `peek_char/[1,2]`, `peek_code/[1,2]`, `peek_byte/[1,2]`, `put_code/[1,2]`, `put_char/[1,2]` and `put_byte/[1,2]`. The prolog-flag `double_quotes` (see `current_prolog_flag/2`) controls how text between double-quotes is interpreted.

3.3 Loading Prolog source files

This section deals with loading Prolog source-files. A Prolog source file is a text-file (often referred to as *ASCII-file*) containing a Prolog program or part thereof. Prolog source files come in three flavours:

A traditional Prolog source file contains a Prolog clauses and directives, but no *module-declaration*. They are normally loaded using `consult/1` or `ensure_loaded/1`.

A module Prolog source file starts with a module declaration. The subsequent Prolog code is loaded into the specified module and only the *public* predicates are made available to the context loading the module. Module files are normally loaded using `use_module/[1,2]`. See chapter 4 for details.

A include Prolog source file is loaded using the `include/1` directive and normally contains only directives.

Prolog source-files are located using `absolute_file_name/3` with the following options:

```
locate_prolog_file(Spec, Path) :-
        absolute_file_name(Spec,
                           [ file_type(prolog),
                             access(read)
                           ],
                           Path).
```

The `file_type`(*prolog*) option is used to determine the extension of the file using `prolog_file_type/2`. The default extension is `.pl`. *Spec* allows for the *path-alias* construct defined by `absolute_file_name/3`. The most commonly used path-alias is `library`(*LibraryFile*). The example below loads the library file `oset.pl` (containing predicates for manipulating ordered sets).

```
:- use_module(library(oset)).
```

SWI-Prolog recognises grammar rules (DCG) as defined in [Clocksin & Melish, 1987]. The user may define additional compilation of the source file by defining the dynamic predicate `term_expansion/2`. Transformations by this predicate overrule the systems grammar rule transformations. It is not allowed to use `assert/1`, `retract/1` or any other database predicate in `term_expansion/2` other than for local computational purposes.[2]

Directives may be placed anywhere in a source file, invoking any predicate. They are executed when encountered. If the directive fails, a warning is printed. Directives are specified by :-/1 or ?-/1. There is no difference between the two.

SWI-Prolog does not have a separate `reconsult/1` predicate. Reconsulting is implied automatically by the fact that a file is consulted which is already loaded.

load_files(+*Files*, +*Options***)**

The predicate `load_files/2` is the parent of all the other loading predicates. It currently supports a subset of the options of Quintus `load_files/2`. *Files* is either specifies a single, or a list of source-files. The specification for a source-file is handled `absolute_file_name/2`. See this predicate for the supported expansions. *Options* is a list of options using the format

OptionName(*OptionValue*)

The following options are currently supported:

if(*Condition***)**

Load the file only if the specified condition is satisfied. The value `true` loads the file unconditionally, `changed` loads the file if it was not loaded before, or has been modified since it was loaded the last time, `not_loaded` loads the file if it was not loaded before.

must_be_module(*Bool***)**

If `true`, raise an error if the file is not a module file. Used by `use_module/[1,2]`.

imports(*ListOrAll***)**

If `all` and the file is a module file, import all public predicates. Otherwise import only the named predicates. Each predicate is refered to as ⟨*name*⟩/⟨*arity*⟩. This option has no effect if the file is not a module file.

silent(*Bool***)**

If `true`, load the file without printing a message. The specified value is the default for all files loaded as a result of loading the specified files.

consult(+*File***)**

Read *File* as a Prolog source file. *File* may be a list of files, in which case all members are consulted in turn. *File* may start with the csh(1) special sequences ~, ⟨*user*⟩ and `$`⟨*var*⟩. *File* may also be `library(Name)`, in which case the libraries are searched for a file with the specified name. See also `library_directory/1` and `file_search_path/2`. `consult/1` may be abbreviated by just typing a number of file names in a list. Examples:

```
?- consult(load).          % consult load or load.pl
?- [library(quintus)].     % load Quintus compatibility library
```

Equivalent to load_files(Files, []).

[2]It does work for normal loading, but not for `qcompile/1`.

ensure_loaded(*+File***)**

If the file is not already loaded, this is equivalent to `consult/1`. Otherwise, if the file defines a module, import all public predicates. Finally, if the file is already loaded, is not a module file and the context module is not the global user module, `ensure_loaded/1` will call `consult/1`.

With the semantics, we hope to get as closely possible to the clear semantics without the presence of a module system. Applications using modules should consider using `use_module/[1,2]`.

Equivalent to load_files(Files, [if(changed)]).

include(*+File***)**

Pretend the terms in *File* are in the source-file in which `:- include(File)` appears. The include construct is only honnoured if it appears as a directive in a source-file. Normally *File* contains a sequence of directives.

require(*+ListOfNameAndArity***)**

Declare that this file/module requires the specified predicates to be defined "with their commonly accepted definition". This predicate originates from the Prolog portability layer for XPCE. It is intended to provide a portable mechanism for specifying that this module requires the specified predicates.

The implementation normally first verifies whether the predicate is already defined. If not, it will search the libraries and load the required library.

SWI-Prolog, having autoloading, does **not** load the library. Instead it creates a procedure header for the predicate if this does not exist. This will flag the predicate as 'undefined'. See also `check/0` and `autoload/0`.

make

Consult all source files that have been changed since they were consulted. It checks *all* loaded source files: files loaded into a compiled state using `pl -c ...` and files loaded using consult or one of its derivatives. The predicate `make/0` is called after `edit/1`, automatically reloading all modified files. It the user uses an external editor (in a separate window), `make/0` is normally used to update the program after editing.

library_directory(*?Atom***)**

Dynamic predicate used to specify library directories. Default `./lib`, `~/lib/prolog` and the system's library (in this order) are defined. The user may add library directories using `assert/1`, `asserta/1` or remove system defaults using `retract/1`.

file_search_path(*+Alias, ?Path***)**

Dynamic predicate used to specify 'path-aliases'. This feature is best described using an example. Given the definition

```
file_search_path(demo, '/usr/lib/prolog/demo').
```

the file specification `demo(myfile)` will be expanded to `/usr/lib/prolog/demo/myfile`. The second argument of `file_search_path/2` may be another alias.

Below is the initial definition of the file search path. This path implies `swi(`⟨*Path*⟩`)` refers to a file in the SWI-Prolog home directory. The alias `foreign(`⟨*Path*⟩`)` is intended for storing shared libraries (`.so` or `.DLL` files). See also `load_foreign_library/[1,2]`.

```
user:file_search_path(library, X) :-
        library_directory(X).
user:file_search_path(swi, Home) :-
        current_prolog_flag(home, Home).
user:file_search_path(foreign, swi(ArchLib)) :-
        current_prolog_flag(arch, Arch),
        atom_concat('lib/', Arch, ArchLib).
user:file_search_path(foreign, swi(lib)).
```

The `file_search_path/2` expansion is used by all loading predicates as well as by `absolute_file_name/[2,3]`.

expand_file_search_path(*+Spec*, **-***Path***)**
Unifies *Path* will all possible expansions of the file name specification *Spec*. See also `absolute_file_name/3`.

prolog_file_type(*?Extension*, *?Type***)**
This dynamic multifile predicate defined in module `user` determines the extensions considered by `file_search_path/2`. *Extension* is the filename extension without the leading dot, *Type* denotes the type as used by the `file_type`(*Type*) option of `file_search_path/2`. Here is the initial definition of `prolog_file_type/2`:

```
user:prolog_file_type(pl,       prolog).
user:prolog_file_type(Ext,      prolog) :-
        current_prolog_flag(associate, Ext),
        Ext \== pl.
user:prolog_file_type(qlf,      qlf).
user:prolog_file_type(Ext,      executable) :-
        current_prolog_flag(shared_object_extension, Ext).
```

Users may wish to change the extension used for Prolog source files to avoid conflicts (for example with `perl`) as well as to be compatible with some specific implementation. The preferred alternative extension is `.pro`.

source_file(*?File***)**
Succeeds if *File* is a loaded Prolog source file. *File* is the absolute and canonical path to the source-file.

source_file(*?Pred*, *?File***)**
Is true if the predicate specified by *Pred* was loaded from file *File*, where *File* is an absolute path name (see `absolute_file_name/2`). Can be used with any instantiation pattern, but the database only maintains the source file for each predicate. See also `clause_property/2`.

prolog_load_context(*?Key*, *?Value***)**
Determine loading context. The following keys are defined:

Key	Description
`module`	Module into which file is loaded
`file`	File loaded
`stream`	Stream identifier (see `current_input/1`)
`directory`	Directory in which *File* lives.
`term_position`	Position of last term read. Term of the form `'$stream_position'(0,`⟨*Line*⟩`,0,0,0)`

Quintus compatibility predicate. See also `source_location/2`.

source_location(-*File*, -*Line***)**
If the last term has been read from a physical file (i.e. not from the file `user` or a string), unify *File* with an absolute path to the file and *Line* with the line-number in the file. New code should use `prolog_load_context/2`.

term_expansion(+*Term1*, -*Term2***)**
Dynamic predicate, normally not defined. When defined by the user all terms read during consulting that are given to this predicate. If the predicate succeeds Prolog will assert *Term2* in the database rather then the read term (*Term1*). *Term2* may be a term of a the form '?- *Goal*' or ':- *Goal*'. *Goal* is then treated as a directive. If *Term2* is a list all terms of the list are stored in the database or called (for directives). If *Term2* is of the form below, the system will assert *Clause* and record the indicated source-location with it.

`'$source_location'(`⟨*File*⟩`, `⟨*Line*⟩`):`⟨*Clause*⟩

When compiling a module (see chapter 4 and the directive `module/2`), `expand_term/2` will first try `term_expansion/2` in the module being compiled to allow for term-expansion rules that are local to a module. If there is no local definition, or the local definition fails to translate the term, `expand_term/2` will try `term_expansion/2` in module `user`. For compatibility with SICStus and Quintus Prolog, this feature should not be used. See also `expand_term/2`, `goal_expansion/2` and `expand_goal/2`.

expand_term(+*Term1*, -*Term2***)**
This predicate is normally called by the compiler to perform preprocessing. First it calls `term_expansion/2`. If this predicate fails it performs a grammar-rule translation. If this fails it returns the first argument.

goal_expansion(+*Goal1*, -*Goal2***)**
Like `term_expansion/2`, `goal_expansion/2` provides for macro-expansion of Prolog source-code. Between `expand_term/2` and the actual compilation, the body of clauses analysed and the goals are handed to `expand_goal/2`, which uses the `goal_expansion/2` hook to do user-defined expansion.

The predicate `goal_expansion/2` is first called in the module that is being compiled, and then on the `user` module.

Only goals apearing in the body of clauses when reading a source-file are expanded using mechanism, and only if they appear literally in the clause, or as an argument to the meta-predicates `not/1`, `call/1` or `forall/2`. A real predicate definition is required to deal with dynamically constructed calls.

expand_goal(+*Goal1*, -*Goal2***)**
: This predicate is normally called by the compiler to perform preprocessing. First it calls `goal_expansion/2`. If this fails it returns the first argument.

at_initialization(+*Goal***)**
: Register *Goal* to be ran when the system initialises. Initialisation takes place after reloading a .qlf (formerly .wic) file as well as after reloading a saved-state. The hooks are run in the order they were registered. A warning message is issued if *Goal* fails, but execution continues. See also `at_halt/1`

at_halt(+*Goal***)**
: Register *Goal* to be ran when the system halts. The hooks are run in the order they were registered. Success or failure executing a hook is ignored. These hooks may not call `halt/[0,1]`.

initialization(+*Goal***)**
: Call *Goal* and register it using `at_initialization/1`. Directives that do other things that creating clauses, records, flags or setting predicate attributes should normally be written using this tag to ensure the initialisation is executed when a saved system starts. See also `qsave_program/[1,2]`.

compiling
: Succeeds if the system is compiling source files with the `-c` option into an intermediate code file. Can be used to perform code optimisations in `expand_term/2` under this condition.

preprocessor(-*Old*, +*New***)**
: Read the input file via a Unix process that acts as preprocessor. A preprocessor is specified as an atom. The first occurrence of the string '`%f`' is replaced by the name of the file to be loaded. The resulting atom is called as a Unix command and the standard output of this command is loaded. To use the Unix C preprocessor one should define:

```
?- preprocessor(Old, '/lib/cpp -C -P %f'), consult(...).

Old = none
```

3.3.1 Quick load files

The features described in this section should be regarded **alpha**.

As of version 2.0.0, SWI-Prolog supports compilation of individual or multiple Prolog sourcefiles into 'Quick Load Files'. A 'Quick Load Files' (`.qlf` file) stores the contents of the file in a precompiled format.

These files load considerably faster than sourcefiles and are normally more compact. They are machine independent and may thus be loaded on any implementation of SWI-Prolog. Note however that clauses are stored as virtual machine instructions. Changes to the compiler will generally make old compiled files unusable.

Quick Load Files are created using `qcompile/1`. They are loaded using `consult/1` or one of the other file-loading predicates described in section 3.3. If consult is given the explicit `.pl` file, it will load the Prolog source. When given the `.qlf` file, it will load the file. When no extension is specified, it will load the `.qlf` file when present and the fileextpl file otherwise.

qcompile(+*File***)**

Takes a single file specification like `consult/1` (i.e. accepts constructs like `library(LibFile)` and creates a Quick Load File from *File*. The file-extension of this file is `.qlf`. The base name of the Quick Load File is the same as the input file.

If the file contains '`:- consult(+File)`' or '`:- [+File]`' statements, the referred files are compiled into the same `.qlf` file. Other directives will be stored in the `.qlf` file and executed in the same fashion as when loading the `.pl` file.

For `term_expansion/2`, the same rules as described in section 2.10 apply.

Source references (`source_file/2`) in the Quick Load File refer to the Prolog source file from which the compiled code originates.

3.4 Listing and Editor Interface

SWI-Prolog offers an extensible interface which allows the user to edit objects of the program: predicates, modules, files, etc. The editor interface is implemented by `edit/1` and consists of three parts: *locating*, *selecting* and *starting the editor*.

Any of these parts may be extended or redefined by adding clauses to various multi-file (see `multifile/1`) predicates defined in the module `prolog_edit`.

The built-in edit specifications for `edit/1` (see `prolog_edit:locate/3`) are described below.

Fully specified objects	
⟨*Module*⟩:⟨*Name*⟩/⟨*Arity*⟩	Refers a predicate
module(⟨*Module*⟩)	Refers to a module
file(⟨*Path*⟩)	Refers to a file
source_file(⟨*Path*⟩)	Refers to a loaded source-file
Ambiguous specifications	
⟨*Name*⟩/⟨*Arity*⟩	Refers this predicate in any module
⟨*Name*⟩	Refers to (1) named predicate in any module with any arity, (2) a (source) file or (3) a module.

edit(+*Specification***)**

First exploits `prolog_edit:locate/3` to translate *Specification* into a list of *Locations*. If there is more than one 'hit', the user is allows to select from the found locations. Finally, `prolog_edit:edit_source/1` is used to invoke the user's preferred editor.

prolog_edit:locate(+*Spec*, -*FullSpec*, -*Location***)**

Where *Spec* is the specification provided through `edit/1`. This multifile predicate is used to enumerate locations at with an object satisfying the given *Spec* can be found. *FullSpec* is unified with the complete specification for the object. This distinction is used to allow for ambiguous specifications. For example, if *Spec* is an atom, which appears as the base-name of a loaded file and as the name of a predicate, *FullSpec* will be bound to `file`(*Path*) or *Name*/*Arity*.

Location is a list of attributes of the location. Normally, this list will contain the term `file`(*File*) and —if available— the term `line`(*Line*).

prolog_edit:locate(+*Spec*, -*Location***)**

Same as `prolog_edit:locate/3`, but only deals with fully-sepecified objects.

prolog_edit:edit_source(+*Location***)**

Start editor on *Location*. See `prolog_edit:locate/3` for the format of a location term. This multi-file predicate is normally not defined. If it succeeds, `edit/1` assumes the editor is started.

If it fails, `edit/1` will invoke an external editor. The editor to be invoked is determined from the evironment variable `EDITOR`, which may be set from the operating system or from the Prolog initialisation file using `setenv/2`. If no editor is defined, `vi` is the default in Unix systems, and `notepad` on Windows.

The predicate `prolog_edit:edit_command/2` defines how the editor will be invoked.

prolog_edit:edit_command(+*Editor*, -*Command***)**

Determines how *Editor* is to be invoked using `shell/1`. *Editor* is the determined editor (see `edit_source/1`), without the full path specification, and without possible (exe) extension. *Command* is an atom describing the command. The pattern `%f` is replaced by the full file-name of the location, and `%d` by the line number. If the editor can deal with starting at a specified line, two clauses should be provided, one holding only the `%f` pattern, and one holding both patterns.

The default contains definitions for `vi`, `emacs`, `emacsclient`, `vim` and `notepad` (latter without line-number version).

Please contribute your specifications to `jan@swi.psy.uva.nl`.

prolog_edit:load

Normally not-defined multifile predicate. This predicate may be defined to provide loading hooks for user-extensions to the edit module. For example, XPCE provides the code below to load library(`swi_edit`), containing definitions to locate classes and methods as well as to bind this package to the PceEmacs built-in editor.

```
:- multifile prolog_edit:load/0.

prolog_edit:load :-
        ensure_loaded(library(swi_edit)).
```

listing(+*Pred***)**

List specified predicates (when an atom is given all predicates with this name will be listed). The listing is produced on the basis of the internal representation, thus loosing user's layout and variable name information. See also `portray_clause/1`.

listing

List all predicates of the database using `listing/1`.

portray_clause(+*Clause***)**

Pretty print a clause. A clause should be specified as a term '⟨*Head*⟩ `:-` ⟨*Body*⟩'. Facts are represented as '⟨*Head*⟩ `:- true`'.

3.5 Verify Type of a Term

var(+*Term***)**
Succeeds if *Term* currently is a free variable.

nonvar(+*Term***)**
Succeeds if *Term* currently is not a free variable.

integer(+*Term***)**
Succeeds if *Term* is bound to an integer.

float(+*Term***)**
Succeeds if *Term* is bound to a floating point number.

number(+*Term***)**
Succeeds if *Term* is bound to an integer or a floating point number.

atom(+*Term***)**
Succeeds if *Term* is bound to an atom.

string(+*Term***)**
Succeeds if *Term* is bound to a string.

atomic(+*Term***)**
Succeeds if *Term* is bound to an atom, string, integer or floating point number.

compound(+*Term***)**
Succeeds if *Term* is bound to a compound term. See also `functor/3` and =../2.

callable(+*Term***)**
Succeeds if *Term* is bound to an atom or a compound term, so it can be handed without type-error to `call/1`, `functor/3` and =../2.

ground(+*Term***)**
Succeeds if *Term* holds no free variables.

3.6 Comparison and Unification or Terms

3.6.1 Standard Order of Terms

Comparison and unification of arbitrary terms. Terms are ordered in the so called “standard order”. This order is defined as follows:

1. *Variables* < *Atoms* < *Strings* < *Numbers* < *Terms*[3]

2. *Old Variable* < *New Variable*[4]

3. *Atoms* are compared alphabetically.

[3]Strings might be considered atoms in future versions. See also section 3.23

[4]In fact the variables are compared on their (dereferenced) addresses. Variables living on the global stack are always < than variables on the local stack. Programs should not rely on the order in which variables are sorted.

4. *Strings* are compared alphabetically.
5. *Numbers* are compared by value. Integers and floats are treated identically.
6. *Compound* terms are first checked on their arity, then on their functor-name (alphabetically) and finally recursively on their arguments, leftmost argument first.

If the prolog_flag (see `current_prolog_flag/2`) `iso` is defined, all floating point numbers precede all integers.

+Term1 **`==`** *+Term2*
: Succeeds if *Term1* is equivalent to *Term2*. A variable is only identical to a sharing variable.

+Term1 **`\==`** *+Term2*
: Equivalent to `\+Term1 == Term2`.

+Term1 **`=`** *+Term2*
: Unify *Term1* with *Term2*. Succeeds if the unification succeeds.

unify_with_occurs_check(*+Term1, +Term2***)**
: As `=/2`, but using *sound-unification*. That is, a variable only unifies to a term if this term does not contain the variable itself. To illustrate this, consider the two goals below:

```
1 ?- A = f(A).

A = f(f(f(f(f(f(f(f(f(f(...))))))))))
2 ?- unify_with_occurs_check(A, f(A)).

No
```

I.e. the first creates a *cyclic-term*, which is printed as an infinitly nested `f/1` term (see the `max_depth` option of `write_term/2`). The second executes logically sound unification and thus fails.

+Term1 **`\=`** *+Term2*
: Equivalent to `\+Term1 = Term2`.

+Term1 **`=@=`** *+Term2*
: Succeeds if *Term1* is 'structurally equal' to *Term2*. Structural equivalence is weaker than equivalence (`==/2`), but stronger than unification (`=/2`). Two terms are structurally equal if their tree representation is identical and they have the same 'pattern' of variables. Examples:

`a`	`=@=`	`A`	false
`A`	`=@=`	`B`	true
`x(A,A)`	`=@=`	`x(B,C)`	false
`x(A,A)`	`=@=`	`x(B,B)`	true
`x(A,B)`	`=@=`	`x(C,D)`	true

+Term1 **`\=@=`** *+Term2*
: Equivalent to `'\+Term1 =@= Term2'`.

+Term1 **`@<`** *+Term2*
: Succeeds if *Term1* is before *Term2* in the standard order of terms.

+Term1 **`@=<`** *+Term2*
: Succeeds if both terms are equal (`==/2`) or *Term1* is before *Term2* in the standard order of terms.

+Term1 **`@>`** *+Term2*
: Succeeds if *Term1* is after *Term2* in the standard order of terms.

+Term1 **`@>=`** *+Term2*
: Succeeds if both terms are equal (`==/2`) or *Term1* is after *Term2* in the standard order of terms.

compare(*?Order, +Term1, +Term2***)**
: Determine or test the *Order* between two terms in the standard order of terms. *Order* is one of `<`, `>` or `=`, with the obvious meaning.

3.7 Control Predicates

The predicates of this section implement control structures. Normally these constructs are translated into virtual machine instructions by the compiler. It is still necessary to implement these constructs as true predicates to support meta-calls, as demonstrated in the example below. The predicate finds all currently defined atoms of 1 character long. Note that the cut has no effect when called via one of these predicates (see !/0).

```
one_character_atoms(As) :-
        findall(A, (current_atom(A), atom_length(A, 1)), As).
```

fail
: Always fail. The predicate `fail/0` is translated into a single virtual machine instruction.

true
: Always succeed. The predicate `true/0` is translated into a single virtual machine instruction.

repeat
: Always succeed, provide an infinite number of choice points.

!
: Cut. Discard choice points of parent frame and frames created after the parent frame. As of SWI-Prolog 3.3, the semantics of the cut are compliant with the ISO standard. This implies that the cut is transparent to `;/2`, `->/2` and `*->/2`. Cuts appearing in the *condition* part of `->/2` and `*->/2` as well as in `\+/1` are local to the condition.

  As an extension, a variable goal that is bound the the `!` is handled as a true cut. This is the only difference between `call/1` and a variable appearing as subgoal.

```
t1 :- (a, !, fail ; b).         % cuts a/0 and t1/0
t2 :- (a -> b, !  ; c).         % cuts b/0 and t2/0
t3(G) :- a, G, fail.            % if ‘G = !’ cuts a/0 and t1/1
t4(G) :- a, call(G), fail.      % if ‘G = !’ cut has no effect
t5 :- call((a, !, fail ; b)).   % cuts a/0
t6 :- \+(a, !, fail ; b).       % cuts a/0
```

+Goal1 **`,`** *+Goal2*

Conjunction. Succeeds if both 'Goal1' and 'Goal2' can be proved. It is defined as (this definition does not lead to a loop as the second comma is handled by the compiler):

```
Goal1, Goal2 :- Goal1, Goal2.
```

+Goal1 **`;`** *+Goal2*

The 'or' predicate is defined as:

```
Goal1 ; _Goal2 :- Goal1.
_Goal1 ; Goal2 :- Goal2.
```

+Goal1 **`|`** *+Goal2*

Equivalent to `;/2`. Retained for compatibility only. New code should use `;/2`. Still nice though for grammar rules.

+Condition **`->`** *+Action*

If-then and If-Then-Else. The `->/2` construct commits to the choices made at its left-hand side, destroying choice-points created inside the clause (by `;/2`), or by goals called by this clause. Unlike `!/0`, the choicepoint of the predicate as a whole (due to multiple clauses) is **not** destroyed. The combination `;/2` and `->/2` is defines as:

```
If -> Then; _Else :- If, !, Then.
If -> _Then; Else :- !, Else.
If -> Then :- If, !, Then.
```

Note that the operator precedence relation between `;` and `->` ensure `If -> Then ; Else` is actually a term of the form `;(->(If, Then), Else)`. The first two clauses belong to the definition of `;/2`), while only the last defines `->/2`.

+Condition **`*->`** *+Action ; +Else*

This construct implements the so-called 'soft-cut'. The control is defined as follows: If *Condition* succeeds at least once, the semantics is the same as (*Condition*, *Action*). If *Condition* does not succeed, the semantics is that of (*Condition*, *Else*). In other words, If *Condition* succeeds at least once, simply behave as the conjunction of *Condition* and *Action*, otherwise execute *Else*.

`\+` *+Goal*

Succeeds if 'Goal' cannot be proven (mnemonic: `+` refers to *provable* and the backslash (`\`) is normally used to indicate negation).

3.8 Meta-Call Predicates

Meta call predicates are used to call terms constructed at run time. The basic meta-call mechanism offered by SWI-Prolog is to use variables as a subclause (which should of course be bound to a valid goal at runtime). A meta-call is slower than a normal call as it involves actually searching the database at runtime for the predicate, while for normal calls this search is done at compile time.

call(+*Goal***)**
: Invoke *Goal* as a goal. Note that clauses may have variables as subclauses, which is identical to `call/1`, except when the argument is bound to the cut. See `!/0`.

call(+*Goal*, +*ExtraArg1*, . . . **)**
: Append *ExtraArg1*, *ExtraArg2*, . . . to the argument list of *Goal* and call the result. For example, `call(plus(1), 2, X)` will call `plus/3`, binding *X* to 3.

The call/[2..] construct is handled by the compiler, which implies that redefinition as a predicate has no effect. The predicates `call/[2-6]` are defined as true predicates, so they can be handled by interpreted code.

apply(+*Term*, +*List***)**
: Append the members of *List* to the arguments of *Term* and call the resulting term. For example: `apply(plus(1), [2, X])` will call `plus(1, 2, X)`. `apply/2` is incorporated in the virtual machine of SWI-Prolog. This implies that the overhead can be compared to the overhead of `call/1`. New code should use call/[2..] if the length of *List* is fixed, which is more widely supported and faster because there is no need to build and examine the argument list.

not(+*Goal***)**
: Succeeds when *Goal* cannot be proven. Retained for compatibility only. New code should use `\+/1`.

once(+*Goal***)**
: Defined as:

```
once(Goal) :-
        Goal, !.
```

`once/1` can in many cases be replaced with `->/2`. The only difference is how the cut behaves (see !/0). The following two clauses are identical:

```
1) a :- once((b, c)), d.
2) a :- b, c -> d.
```

ignore(+*Goal***)**
: Calls *Goal* as `once/1`, but succeeds, regardless of whether *Goal* succeeded or not. Defined as:

```
ignore(Goal) :-
        Goal, !.
ignore(_).
```

call_with_depth_limit(+*Goal*, +*Limit*, -*Result***)**
: If *Goal* can be proven without recursion deeper than *Limit* levels, `call_with_depth_limit/3` succeeds, binding *Result* to the deepest recursion level used during the proof. Otherwise, *Result* is unified with `depth_limit_exceeded` if the limit was exceeded during the proof, or the entire predicate fails if *Goal* fails without exceeding *Limit*.

The depth-limit is guarded by the internal machinery. This differ from the depth computed based on a theoretical model. For example, `true/0` is translated into an inlined virtual machine instruction. Also, `repeat/0` is not implemented as below, but as a non-deterministic foreign predicate.

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

As a result, `call_with_depth_limit/3` may still loop inifitly on programs that should theoretically finish in finite time. This problem can be cured by using Prolog equivalents to such built-in predicates.

This predicate may be used for theorem-provers to realise techniques like *iterrative deepening*. It was implemented after discussion with Steve Moyle `smoyle@ermine.ox.ac.uk`.

3.9 ISO compliant Exception handling

SWI-Prolog defines the predicates `catch/3` and `throw/1` for ISO compliant raising and catching of exceptions. In the current implementation (2.9.0), only part of the built-in predicates generate exceptions. In general, exceptions are implemented for I/O and arithmetic.

catch(*:Goal, +Catcher, :Recover***)**
Behaves as `call/1` if no exception is raised when executing *Goal*. If a exception is raised using `throw/1` while *Goal* executes, and the *Goal* is the innermost goal for which *Catcher* unifies with the argument of `throw/1`, all choicepoints generated by *Goal* are cut, and *Recover* is called as in `call/1`.

The overhead of calling a goal through `catch/3` is very comparable to `call/1`. Recovery from an exception has a similar overhead.

throw(*+Exception***)**
Raise an exception. The system will look for the innermost `catch/3` ancestor for which *Exception* unifies with the *Catcher* argument of the `catch/3` call. See `catch/3` for details.

If there is no `catch/3` willing to catch the error in the current Prolog context, the toplevel (`prolog/0`) catches the error and prints a warning message. If an exception was raised in a callback from C (see chapter 5), `PL_next_solution()` will fail and the exception context can be retrieved using `PL_exception()`.

3.9.1 Debugging and exceptions

Before the introduction of exceptions in SWI-Prolog a runtime error was handled by printing an error message, after which the predicate failed. If the prolog_flag (see `current_prolog_flag/2`) `debug_on_error` was in effect (default), the tracer was switched on. The combination of the error message and trace information is generally sufficient to locate the error.

With exception handling, things are different. A programmer may wish to trap an exception using `catch/3` to avoid it reaching the user. If the exception is not handled by user-code, the interactive toplevel will trap it to prevent termination.

If we do not take special precautions, the context information associated with an unexpected exception (i.e. a programming error) is lost. Therefore, if an exception is raised, which is not caught using `catch/3` and the toplevel is running, the error will be printed, and the system will enter trace mode.

If the system is in an non-interactive callback from foreign code and there is no `catch/3` active in the current context, it cannot determine whether or not the exception will be caught by the external routine calling Prolog. It will then base its behaviour on the prolog_flag debug_on_error:

- *current_prolog_flag(debug_on_error, false)*
  The exception does not trap the debugger and is returned to the foreign routine calling Prolog, where it can be accessed using `PL_exception()`. This is the default.

- *current_prolog_flag(debug_on_error, true)*
  If the exception is not caught by Prolog in the current context, it will trap the tracer to help analysing the context of the error.

While looking for the context in which an exception takes place, it is adviced to switch on debug mode using the predicate `debug/0`.

3.9.2 The exception term

Builtin predicates generates exceptions using a term `error`(*Formal*, *Context*). The first argument is the 'formal' description of the error, specifying the class and generic defined context information. When applicable, the ISO error-term definition is used. The second part describes some additional context to help the programmer while debugging. In its most generic form this is a term of the form `context`(*Name/Arity*, *Message*), where *Name*/*Arity* describes the built-in predicate that raised the error, and *Message* provides an additional description of the error. Any part of this structure may be a variable if no information was present.

3.9.3 Printing messages

The predicate `print_message/2` may be used to print a message term in a human readable format. The other predicates from this section allow the user to refine and extend the message system. The most common usage of `print_message/2` is to print error messages from exceptions. The code below prints errors encountered during the execution of *Goal*, without further propagating the exception and without starting the debugger.

```
        ...,
        catch(Goal, E,
              ( print_message(error, E),
                fail
              )),
        ...
```

Another common use is to defined `message_hook/3` for printing messages that are normally *silent*, suppressing messages, redirecting messages or make something happen in addition to printing the message.

print_message(+*Kind*, +*Term***)**

The predicate `print_message/2` is used to print messages, notably from exceptions in a human-readable format. *Kind* is one of `informational`, `warning`, `error`, `help` or `silent`. A human-readable message is printed to the stream `user_error`.

This predicate first translates the *Term* into a list of 'message lines' (see `print_message_lines/3` for details). Next it will call the hook `message_hook/3` to allow the user intercepting the message. If `message_hook/3` fails it will print the message unless *Kind* is silent.

The `print_message/2` predicate and its rules are in the file ⟨*plhome*⟩`/boot/messages.pl`, which may be inspected for more information on the error messages and related error terms.

See also `message_to_string/2`.

print_message_lines(+*Stream*, +*Prefix*, +*Lines***)**

Print a message (see `print_message/2`) that has been translated to a list of message elements. The elements of this list are:

⟨*Format*⟩-⟨*Args*⟩
: Where *Format* is an atom and *Args* is a list of format argument. Handed to `format/3`.

`flush`
: If this appears as the last element, *Stream* is flushed (see `flush_output/1`) and no final newline is generated.

`at_same_line`
: If this appears as first element, no prefix is printed for the first line and the line-position is not forced to 0 (see `format/1`, `~N`).

⟨*Format*⟩
: Handed to `format/3` as `format(Stream, Format, [])`.

nl
: A new line is started and if the message is not complete the *Prefix* is printed too.

See also `print_message/2` and `message_hook/3`.

message_hook(+*Term*, +*Kind*, +*Lines***)**

Hook predicate that may be define in the module `user` to intercept messages from `print_message/2`. *Term* and *Kind* are the same as passed to `print_message/2`. *Lines* is a list of format statements as described with `print_message_lines/3`. See also `message_to_string/2`.

This predicate should be defined dynamic and multifile to allow other modules defining clauses for it too.

message_to_string(+*Term*, -*String***)**

Translates a message-term into a string object (see section **??**. Primarily intended to write messages to Windows in XPCE (see section 1.5) or other GUI environments.

3.10 Handling signals

As of version 3.1.0, SWI-Prolog is capable to handle software interrupts (signals) in Prolog as well as in foreign (C) code (see section 5.6.12).

Signals are used to handle internal errors (execution of a non-existing CPU intruction, arithmetic domain errors, illegal memory access, resource overflow, etc.), as well as for dealing asynchronous inter-process communication.

Signals are defined by the Posix standard and part of all Unix machines. The MS-Windows Win32 provides a subset of the signal handling routines, lacking the vital funtionality to raise a signal in another thread for achieving asynchronous inter-process (or inter-thread) communication (Unix kill() function).

on_signal(+*Signal*, -*Old*, :*New***)**

Determines the reaction on *Signal*. *Old* is unified with the old behaviour, while the behaviour is switched to *New*. As with similar environment-control predicates, the current value is retrieved using `on_signal(Signal, Current, Current)`.

The action description is an atom denoting the name of the predicate that will be called if *Signal* arrives. `on_signal/3` is a meta predicate, which implies that ⟨*Module*⟩:⟨*Name*⟩ refers the ⟨*Name*⟩/1 in the module ⟨*Module*⟩.

Two predicate-names have special meaning. `throw` implies Prolog will map the signal onto a Prolog exception as described in section 3.9. `default` resets the handler to the settings active before SWI-Prolog manipulated the handler.

Signals bound to a foreign function through `PL_signal()` are reported using the term `$foreign_function(`*Address*`)`.

After receiving a signal mapped to `throw`, the exception raised has the structure

`error(signal(`⟨*SigName*⟩`, `⟨*SigNum*⟩`), `⟨*Context*⟩`)`

One possible usage of this is, for example, to limit the time spent on proving a goal. This requires a little C-code for setting the alarm timer (see chapter 5):

```
#include <SWI-Prolog.h>
#include <unistd.h>

foreign_t
pl_alarm(term_t time)
{ double t;

  if ( PL_get_float(time, &t) )
  { alarm((long)(t+0.5));

    PL_succeed;
  }

  PL_fail;
}
```

```
install_t
install()
{ PL_register_foreign("alarm", 1, pl_alarm, 0);
}
```

Next, we can define the following Prolog code:

```
:- load_foreign_library(alarm).

:- on_signal(alrm, throw).

:- module_transparent
        call_with_time_limit/2.

call_with_time_limit(Goal, MaxTime) :-
        alarm(MaxTime),
        catch(Goal, error(signal(alrm, _), _), fail), !,
        alarm(0).
call_with_time_limit(_, _) :-
        alarm(0),
        fail.
```

The signal names are defined by the C-Posix standards as symbols of the form `SIG_`⟨*SIGNAME*⟩. The Prolog name for a signal is the lowercase version of ⟨*SIGNAME*⟩. The predicate `current_signal/3` may be used to map between names and signals.

Initially, some signals are mapped to `throw`, while all other signals are `default`. The following signals throw an exception: `ill`, `fpe`, `segv`, `pipe`, `alrm`, `bus`, `xcpu`, `xfsz` and `vtalrm`.

current_signal(*?Name, ?Id, ?Handler***)**
Enumerate the currently defined signal handling. *Name* is the signal name, *Id* is the numerical identifier and *Handler* is the currently defined handler (see `on_signal/3`).

3.10.1 Notes on signal handling

Before deciding to deal with signals in your application, please consider the following:

- *Portibility*
  On MS-Windows, the signal interface is severely limited. Different Unix brands support different sets of signals, and the relation between signal name and number may vary.

- *Safety*
  Signal handling is not completely safe in the current implementation, especially if `throw` is used in combination with external foreign code. The system will use the C longjmp() construct to direct control to the innermost `PL_next_solution()`, thus forcing an external procedure to be abandoned at an arbitrary moment. Most likely not all SWI-Prologs own foreign code is (yet) safe too.

- *Garbage Collection*
  The garbage collector will block all signals that are handled by Prolog. While handling a signal, the garbage-collector is disabled.

- *Time of delivery*
  Normally delivery is immediate (or as defined by the operating system used). Signals are blocked when the garbage collector is active, and internally delayed if they occur within in a ‘critical section’. The critical sections are generally very short.

3.11 The ‘block’ control-structure

The `block/3` predicate and friends have been introduced before ISO compatible `catch/3` exception handling for compatibility with some Prolog implementation. The only feature not covered by `catch/3` and `throw/1` is the posibility to execute global cuts. New code should use `catch/3` and `throw/1` to deal with exceptions.

block(*+Label, +Goal, -ExitValue***)**
Execute *Goal* in a *block*. *Label* is the name of the block. *Label* is normally an atom, but the system imposes no type constraints and may even be a variable. *ExitValue* is normally unified to the second argument of an `exit/2` call invoked by *Goal*.

exit(*+Label, +Value***)**
Calling `exit/2` makes the innermost *block* which *Label* unifies exit. The block’s *ExitValue* is unified with *Value*. If this unification fails the block fails.

fail(*+Label***)**
Calling `fail/1` makes the innermost *block* which *Label* unifies fail immediately. Implemented as

```
fail(Label) :- !(Label), fail.
```

!(*+Label***)**
Cut all choice-points created since the entry of the innermost *block* which *Label* unifies.

3.12 DCG Grammar rules

Grammar rules form a comfortable interface to *difference-lists*. They are designed both to support writing parsers that build a parse-tree from a list as for generating a flat list from a term. Unfortunately, Definite Clause Grammar (DCG) handling is not part of the Prolog standard. Most Prolog engines implement DCG, but the details differ slightly.

Grammar rules look like ordinary clauses using `-->/2` for separating the head and body rather then `:-/2`. Expanding grammar rules is done by `expand_term/2`, which adds two additional argument to each term for representing the difference list. We will illustrate the behaviour by defining a rule-set for parsing an integer.

```
integer(I) -->
        digit(D0),
        digits(D),
        { number_chars(I, [D0|D])
        }.

digits([D|T]) -->
        digit(D), !,
        digits(T).
digits([]) -->
        [].

digit(D) -->
        [D],
        { code_type(D, digit)
        }.
```

The body of a grammar rule can contain three types of terms. A compound term interpreted as a reference to a grammar-rule. Code between `{...}` is interpreted as a reference to ordinary Prolog code and finally, a list is interpreted as a sequence of literals. The Prolog control-constructs (`\+/1`, `->/2`, `;//2`, `,/2` and `!/0`) can be used in grammar rules.

Grammar rule-sets are called using the builtin predicates `phrase/2` and `phrase/3`:

phrase(+*RuleSet*, +*InputList***)**
Equivalent to `phrase(`*`RuleSet`*`, `*`InputList`*`, [])`.

phrase(+*RuleSet*, +*InputList*, -*Rest***)**
Activate the rule-set with given name. 'InputList' is the list of tokens to parse, 'Rest' is unified with the remaining tokens if the sentence is parsed correctly. The example below calls the rule-set 'integer' defined above.

```
?- phrase(integer(X), "42 times", Rest).

X = 42
Rest = [32, 116, 105, 109, 101, 115]
```

3.13 Database

SWI-Prolog offers three different database mechanisms. The first one is the common assert/retract mechanism for manipulating the clause database. As facts and clauses asserted using `assert/1` or one of its derivatives become part of the program these predicates compile the term given to them. `retract/1` and `retractall/1` have to unify a term and therefore have to decompile the program. For these reasons the assert/retract mechanism is expensive. On the other hand, once compiled, queries to the database are faster than querying the recorded database discussed below. See also `dynamic/1`.

The second way of storing arbitrary terms in the database is using the "recorded database". In this database terms are associated with a *key*. A key can be an atom, integer or term. In the last case only the functor and arity determine the key. Each key has a chain of terms associated with it. New terms can be added either at the head or at the tail of this chain. This mechanism is considerably faster than the assert/retract mechanism as terms are not compiled, but just copied into the heap.

The third mechanism is a special purpose one. It associates an integer or atom with a key, which is an atom, integer or term. Each key can only have one atom or integer associated with it. It is faster than the mechanisms described above, but can only be used to store simple status information like counters, etc.

abolish(*:PredicateIndicator***)**

Removes all clauses of a predicate with functor *Functor* and arity *Arity* from the database. All predicate attributes (dynamic, multifile, index, etc.) are reset to their defaults. Abolishing an imported predicate only removes the import link; the predicate will keep its old definition in its definition module.

According to the ISO standard, `abolish/1` can only be applied to dynamic procedures. This is odd, as for dealing with dynamic procedures there is already `retract/1` and `retractall/1`. The `abolish/1` predicate has been introduced in DEC-10 Prolog precisely for dealing with static procedures. In SWI-Prolog, `abolish/1` works on static procedures, unless the prolog flag `iso` is set to `true`.

It is adviced to use `retractall/1` for erasing all clauses of a dynamic predicate.

abolish(*+Name, +Arity***)**

Same as `abolish(Name/Arity)`. The predicate `abolish/2` conforms to the Edinburgh standard, while `abolish/1` is ISO compliant.

redefine_system_predicate(*+Head***)**

This directive may be used both in module `user` and in normal modules to redefine any system predicate. If the system definition is redefined in module `user`, the new definition is the default definition for all sub-modules. Otherwise the redefinition is local to the module. The system definition remains in the module `system`.

Redefining system predicate facilitates the definition of compatibility packages. Use in other context is discouraged.

retract(*+Term***)**

When *Term* is an atom or a term it is unified with the first unifying fact or clause in the database. The fact or clause is removed from the database.

retractall(*+Head***)**

All facts or clauses in the database for which the *head* unifies with *Head* are removed.

assert(*+Term***)**

Assert a fact or clause in the database. *Term* is asserted as the last fact or clause of the corresponding predicate.

asserta(*+Term***)**

Equivalent to `assert/1`, but *Term* is asserted as first clause or fact of the predicate.

assertz(*+Term***)**

Equivalent to `assert/1`.

assert(*+Term, -Reference***)**
: Equivalent to `assert/1`, but *Reference* is unified with a unique reference to the asserted clause. This key can later be used with `clause/3` or `erase/1`.

asserta(*+Term, -Reference***)**
: Equivalent to `assert/2`, but *Term* is asserted as first clause or fact of the predicate.

assertz(*+Term, -Reference***)**
: Equivalent to `assert/2`.

recorda(*+Key, +Term, -Reference***)**
: Assert *Term* in the recorded database under key *Key*. *Key* is an integer, atom or term. *Reference* is unified with a unique reference to the record (see `erase/1`).

recorda(*+Key, +Term***)**
: Equivalent to `recorda(`*`Key, Value,`*` _)`.

recordz(*+Key, +Term, -Reference***)**
: Equivalent to `recorda/3`, but puts the *Term* at the tail of the terms recorded under *Key*.

recordz(*+Key, +Term***)**
: Equivalent to `recordz(`*`Key, Value,`*` _)`.

recorded(*+Key, -Value, -Reference***)**
: Unify *Value* with the first term recorded under *Key* which does unify. *Reference* is unified with the memory location of the record.

recorded(*+Key, -Value***)**
: Equivalent to `recorded(`*`Key, Value,`*` _)`.

erase(*+Reference***)**
: Erase a record or clause from the database. *Reference* is an integer returned by `recorda/3` or `recorded/3`, `clause/3`, `assert/2`, `asserta/2` or `assertz/2`. Other integers might conflict with the internal consistency of the system. Erase can only be called once on a record or clause. A second call also might conflict with the internal consistency of the system.[5]

flag(*+Key, -Old, +New***)**
: *Key* is an atom, integer or term. Unify *Old* with the old value associated with *Key*. If the key is used for the first time *Old* is unified with the integer 0. Then store the value of *New*, which should be an integer, float, atom or arithmetic expression, under *Key*. `flag/3` is a very fast mechanism for storing simple facts in the database. Example:

```
:- module_transparent succeeds_n_times/2.

succeeds_n_times(Goal, Times) :-
        (   flag(succeeds_n_times, Old, 0),
            Goal,
               flag(succeeds_n_times, N, N+1),
```

[5]BUG: The system should have a special type for pointers, thus avoiding the Prolog user having to worry about consistency matters. Currently some simple heuristics are used to determine whether a reference is valid.

```
                fail
            ;   flag(succeeds_n_times, Times, Old)
            ).
```

3.13.1 Update view

Traditionally, Prolog systems used the *immediate update view*: new clauses became visible to predicates backtracking over dynamic predicates immediately and retracted clauses became invisible immediately.

Starting with SWI-Prolog 3.3.0 we adhere the *logical update view*, where backtrackable predicates that enter the definition of a predicate will not see any changes (either caused by `assert/1` or `retract/1`) to the predicate. This view is the ISO standard, the most commonly used and the most 'safe'.[6] Logical updates are realised by keeping reference-counts on predicates and *generation* information on clauses. Each change to the database causes an increment of the generation of the database. Each goal is tagged with the generation in which it was started. Each clause is flagged with the generation it was created as well as the generation it was erased. Only clauses with 'created' ... 'erased' interval that encloses the generation of the current goal are considered visible.

3.13.2 Indexing databases

By default, SWI-Prolog, as most other implementations, indexes predicates on their first argument. SWI-Prolog allows indexing on other and multiple arguments using the declaration `index/1`.

For advanced database indexing, it defines `hash_term/2`:

hash_term(+*Term*, -*HashKey***)**

If *Term* is a ground term (see `ground/1`), *HashKey* is unified with a positive integer value that may be used as a hash-key to the value. If *Term* is not ground, the predicate succeeds immediately, leaving *HashKey* an unbound variable.

This predicate may be used to build hash-tables as well as to exploit argument-indexing to find complex terms more quickly.

The hash-key does not rely on temporary information like addresses of atoms and may be assumed constant over different invocations of SWI-Prolog.

3.14 Declaring predicates properties

This section describes directives which manipulate attributes of predicate definitions. The functors `dynamic/1`, `multifile/1` and `discontiguous/1` are operators of priority 1150 (see `op/3`), which implies the list of predicates they involve can just be a comma separated list:

```
:- dynamic
        foo/0,
        baz/2.
```

On SWI-Prolog all these directives are just predicates. This implies they can also be called by a program. Do not rely on this feature if you want to maintain portability to other Prolog implementations.

[6]For example, using the immediate update view, no call to a dynamic predicate is deterministic.

dynamic *+Functor/+Arity, ...*

Informs the interpreter that the definition of the predicate(s) may change during execution (using `assert/1` and/or `retract/1`). Currently `dynamic/1` only stops the interpreter from complaining about undefined predicates (see `unknown/2`). Future releases might prohibit `assert/1` and `retract/1` for not-dynamic declared procedures.

multifile *+Functor/+Arity, ...*

Informs the system that the specified predicate(s) may be defined over more than one file. This stops `consult/1` from redefining a predicate when a new definition is found.

discontiguous *+Functor/+Arity, ...*

Informs the system that the clauses of the specified predicate(s) might not be together in the source file. See also `style_check/1`.

index(*+Head***)**

Index the clauses of the predicate with the same name and arity as *Head* on the specified arguments. *Head* is a term of which all arguments are either '1' (denoting 'index this argument') or '0' (denoting 'do not index this argument'). Indexing has no implications for the semantics of a predicate, only on its performance. If indexing is enabled on a predicate a special purpose algorithm is used to select candidate clauses based on the actual arguments of the goal. This algorithm checks whether indexed arguments might unify in the clause head. Only atoms, integers and functors (e.g. name and arity of a term) are considered. Indexing is very useful for predicates with many clauses representing facts.

Due to the representation technique used at most 4 arguments can be indexed. All indexed arguments should be in the first 32 arguments of the predicate. If more than 4 arguments are specified for indexing only the first 4 will be accepted. Arguments above 32 are ignored for indexing.

By default all predicates with $\langle arity \rangle \geq 1$ are indexed on their first argument. It is possible to redefine indexing on predicates that already have clauses attached to them. This will initiate a scan through the predicates clause list to update the index summary information stored with each clause.

If—for example—one wants to represents sub-types using a fact list 'sub_type(Sub, Super)' that should be used both to determine sub- and super types one should declare sub_type/2 as follows:

```
:- index(sub_type(1, 1)).

sub_type(horse, animal).
...
...
```

3.15 Examining the program

current_atom(*-Atom***)**

Successively unifies *Atom* with all atoms known to the system. Note that `current_atom/1` always succeeds if *Atom* is instantiated to an atom.

current_functor(_?Name, ?Arity_**)**
Successively unifies _Name_ with the name and _Arity_ with the arity of functors known to the system.

current_flag(-_FlagKey_**)**
Successively unifies _FlagKey_ with all keys used for flags (see `flag/3`).

current_key(-_Key_**)**
Successively unifies _Key_ with all keys used for records (see `recorda/3`, etc.).

current_predicate(_?Name, ?Head_**)**
Successively unifies _Name_ with the name of predicates currently defined and _Head_ with the most general term built from _Name_ and the arity of the predicate. This predicate succeeds for all predicates defined in the specified module, imported to it, or in one of the modules from which the predicate will be imported if it is called.

predicate_property(_?Head_, _?Property_**)**
Succeeds if _Head_ refers to a predicate that has property _Property_. Can be used to test whether a predicate has a certain property, obtain all properties known for _Head_, find all predicates having _property_ or even obtaining all information available about the current program. _Property_ is one of:

interpreted
Is true if the predicate is defined in Prolog. We return true on this because, although the code is actually compiled, it is completely transparent, just like interpreted code.

built_in
Is true if the predicate is locked as a built-in predicate. This implies it cannot be redefined in its definition module and it can normally not be seen in the tracer.

foreign
Is true if the predicate is defined in the C language.

dynamic
Is true if the predicate is declared dynamic using the `dynamic/1` declaration.

multifile
Is true if the predicate is declared multifile using the `multifile/1` declaration.

undefined
Is true if a procedure definition block for the predicate exists, but there are no clauses in it and it is not declared dynamic. This is true if the predicate occurs in the body of a loaded predicate, an attempt to call it has been made via one of the meta-call predicates or the predicate had a definition in the past. See the library package _check_ for example usage.

transparent
Is true if the predicate is declared transparent using the `module_transparent/1` declaration.

exported
Is true if the predicate is in the public list of the context module.

imported_from(_Module_**)**
Is true if the predicate is imported into the context module from module _Module_.

indexed(*Head***)**
Predicate is indexed (see `index/1`) according to *Head*. *Head* is a term whose name and arity are identical to the predicate. The arguments are unified with '1' for indexed arguments, '0' otherwise.

file(*FileName***)**
Unify *FileName* with the name of the sourcefile in which the predicate is defined. See also `source_file/2`.

line_count(*LineNumber***)**
Unify *LineNumber* with the line number of the first clause of the predicate. Fails if the predicate is not associated with a file. See also `source_file/2`.

number_of_clauses(*ClauseCount***)**
Unify *ClauseCount* to the number of clauses associated with the predicate. Fails for foreign predicates.

dwim_predicate(*+Term, -Dwim***)**
'Do What I Mean' ('dwim') support predicate. *Term* is a term, which name and arity are used as a predicate specification. *Dwim* is instantiated with the most general term built from *Name* and the arity of a defined predicate that matches the predicate specified by *Term* in the 'Do What I Mean' sense. See `dwim_match/2` for 'Do What I Mean' string matching. Internal system predicates are not generated, unless `style_check(+dollar)` is active. Backtracking provides all alternative matches.

clause(*?Head, ?Body***)**
Succeeds when *Head* can be unified with a clause head and *Body* with the corresponding clause body. Gives alternative clauses on backtracking. For facts *Body* is unified with the atom *true*. Normally `clause/2` is used to find clause definitions for a predicate, but it can also be used to find clause heads for some body template.

clause(*?Head, ?Body, ?Reference***)**
Equivalent to `clause/2`, but unifies *Reference* with a unique reference to the clause (see also `assert/2`, `erase/1`). If *Reference* is instantiated to a reference the clause's head and body will be unified with *Head* and *Body*.

nth_clause(*?Pred, ?Index, ?Reference***)**
Provides access to the clauses of a predicate using their index number. Counting starts at 1. If *Reference* is specified it unifies *Pred* with the most general term with the same name/arity as the predicate and *Index* with the index-number of the clause. Otherwise the name and arity of *Pred* are used to determine the predicate. If *Index* is provided *Reference* will be unified with the clause reference. If *Index* is unbound, backtracking will yield both the indices and the references of all clauses of the predicate. The following example finds the 2nd clause of `member/2`:

```
?- nth_clause(member(_,_), 2, Ref), clause(Head, Body, Ref).

Ref = 160088
Head = system : member(G575, [G578|G579])
Body = member(G575, G579)
```

clause_property(+*ClauseRef*, -*Property***)**
Queries properties of a clause. *ClauseRef* is a reference to a clause as produced by `clause/3`, `nth_clause/3` or `prolog_frame_attribute/3`. *Property* is one of the following:

file(*FileName***)**
Unify *FileName* with the name of the sourcefile in which the clause is defined. Fails if the clause is not associated to a file.

line_count(*LineNumber***)**
Unify *LineNumber* with the line number of the clause. Fails if the clause is not associated to a file.

fact
True if the clause has no body.

erased
True if the clause has been erased, but not yet reclaimed because it is referenced.

3.16 Input and output

SWI-Prolog provides two different packages for input and output. One confirms to the Edinburgh standard. This package has a notion of 'current-input' and 'current-output'. The reading and writing predicates implicitly refer to these streams. In the second package, streams are opened explicitly and the resulting handle is used as an argument to the reading and writing predicate to specify the source or destination. Both packages are fully integrated; the user may switch freely between them.

3.16.1 Input and output using implicit source and destination

The package for implicit input and output destination is upwards compatible to DEC-10 and C-Prolog. The reading and writing predicates refer to resp. the current input- and output stream. Initially these streams are connected to the terminal. The current output stream is changed using `tell/1` or `append/1`. The current input stream is changed using `see/1`. The streams current value can be obtained using `telling/1` for output- and `seeing/1` for input streams. The table below shows the valid stream specifications. The reserved names `user_input`, `user_output` and `user_error` are for neat integration with the explicit streams.

`user`	This reserved name refers to the terminal
`user_input`	Input from the terminal
`user_output`	Output to the terminal
`user_error`	Unix error stream (output only)
⟨*Atom*⟩	Name of a Unix file
`pipe(`⟨*Atom*⟩`)`	Name of a Unix command

Source and destination are either a file, one of the reserved words above, or a term 'pipe(*Command*)'. In the predicate descriptions below we will call the source/destination argument '*SrcDest*'. Below are some examples of source/destination specifications.

```
?- see(data).          % Start reading from file 'data'.
?- tell(user_error).   % Start writing on the error stream.
?- tell(pipe(lpr)).    % Start writing to the printer.
```

Another example of using the `pipe/1` construct is shown below. Note that the `pipe/1` construct is not part of Prolog's standard I/O repertoire.

```
getwd(Wd) :-
        seeing(Old), see(pipe(pwd)),
        collect_wd(String),
        seen, see(Old),
        atom_codes(Wd, String).

collect_wd([C|R]) :-
        get0(C), C \== -1, !,
        collect_wd(R).
collect_wd([]).
```

see(+*SrcDest***)**
: Make *SrcDest* the current input stream. If *SrcDest* was already opened for reading with `see/1` and has not been closed since, reading will be resumed. Otherwise *SrcDest* will be opened and the file pointer is positioned at the start of the file.

tell(+*SrcDest***)**
: Make *SrcDest* the current output stream. If *SrcDest* was already opened for writing with `tell/1` or `append/1` and has not been closed since, writing will be resumed. Otherwise the file is created or—when existing—truncated. See also `append/1`.

append(+*File***)**
: Similar to `tell/1`, but positions the file pointer at the end of *File* rather than truncating an existing file. The pipe construct is not accepted by this predicate.

seeing(?*SrcDest***)**
: Unify the name of the current input stream with *SrcDest*.

telling(?*SrcDest***)**
: Unify the name of the current output stream with *SrcDest*.

seen
: Close the current input stream. The new input stream becomes *user*.

told
: Close the current output stream. The new output stream becomes *user*.

3.16.2 Explicit Input and Output Streams

The predicates below are part of the Quintus compatible stream-based I/O package. In this package streams are explicitly created using the predicate `open/3`. The resulting stream identifier is then passed as a parameter to the reading and writing predicates to specify the source or destination of the data.

open(+*SrcDest*, +*Mode*, -*Stream*, +*Options***)**
: ISO compliant predicate to open a stream. *SrcDes* is either an atom, specifying a Unix file, or

a term ‘`pipe(`*`Command`*`)`’, just like `see/1` and `tell/1`. *Mode* is one of `read`, `write`, `append` or `update`. Mode `append` opens the file for writing, positioning the file-pointer at the end. Mode `update` opens the file for writing, positioning the file-pointer at the beginning of the file without truncating the file. See also `stream_position/3`. *Stream* is either a variable, in which case it is bound to an integer identifying the stream, or an atom, in which case this atom will be the stream identifier. The *Options* list can contain the following options:

type(*Type***)**
Using type `text` (default), Prolog will write a text-file in an operating-system compatible way. Using type `binary` the bytes will be read or written without any translation. Note there is no difference between the two on Unix systems.

alias(*Atom***)**
Gives the stream a name. Below is an example. Be careful with this option as stream-names are global. See also `set_stream/2`.

```
?- open(data, read, Fd, [alias(input)]).

        ...,
        read(input, Term),
        ...
```

eof_action(*Action***)**
Defines what happens if the end of the input stream is reached. Action `eof_code` makes `get0/1` and friends return -1 and `read/1` and friends return the atom `end_of_file`. Repetitive reading keeps yielding the same result. Action `error` is like `eof_code`, but repetitive reading will raise an error. With action `reset`, Prolog will examine the file again and return more data if the file has grown.

buffer(*Buffering***)**
Defines output buffering. The atom `fullf` (default) defines full buffering, `line` buffering by line, and `false` implies the stream is fully unbuffered. Smaller buffering is useful if another process or the user is waiting for the output as it is being produced. See also `flush_output/[0,1]`. This option is not an ISO option.

close_on_abort(*Bool***)**
If `true` (default), the stream is closed on an abort (see `abort/0`). If `false`, the stream is not closed. If it is an output stream, it will be flushed however. Useful for logfiles and if the stream is associated to a process (using the `pipe/1` construct).

The option `reposition` is not supported in SWI-Prolog. All streams connected to a file may be repositioned.

open(*+SrcDest, +Mode, ?Stream***)**
Equivalent to `open/4` with an empty option-list.

open_null_stream(*?Stream***)**
Open a stream that produces no output. All counting functions are enabled on such a stream. An attempt to read from a null-stream will immediately signal end-of-file. Similar to Unix `/dev/null`. *Stream* can be an atom, giving the null-stream an alias name.

close(+*Stream***)**
: Close the specified stream. If *Stream* is not open an error message is displayed. If the closed stream is the current input or output stream the terminal is made the current input or output.

close(+*Stream*, +*Options***)**
: Provides `close`(*Stream, [force(true)]*) as the only option. Called this way, any resource error (such as write-errors while flushing the output buffer) are ignored.

stream_property(?*Stream*, ?*StreamProperty***)**
: ISO compatible predicate for querying status of open I/O streams. *StreamProperty* is one of:

file_name(*Atom***)**
: If *Stream* is associated to a file, unify *Atom* to the name of this file.

mode(*IOMode***)**
: Unify *IOMode* to the mode given to `open/4` for opening the stream. Values are: `read`, `write`, `append` and the SWI-Prolog extension `update`.

input
: True if *Stream* has mode `read`.

output
: True if *Stream* has mode `write`, `append` or `update`.

alias(*Atom***)**
: If *Atom* is bound, test of the stream has the specified alias. Otherwise unify *Atom* with the first alias of the stream.[7]

position(*Term***)**
: Unify *Term* with the current stream-position. A stream-position is a term of format `$stream_position`(*CharIndex, LineNo, LinePos*). See also `term_position/3`.

end_of_stream(*E***)**
: If *Stream* is an input stream, unify *E* with one of the atoms `not`, `at` or `past`. See also `at_end_of_stream/[0,1]`.

eof_action(*A***)**
: Unify *A* with one of `eof_code`, `reset` or `error`. See `open/4` for details.

reposition(*Bool***)**
: Unify *Bool* with *true* if the position of the stream can be set (see `seek/4`). It is assumed the position can be set if the stream has a *seek-function* and is not based on a POSIX file-descriptor that is not associated to a regular file.

type(*T***)**
: Unify *Bool* with `text` or `binary`.

file_no(*Integer***)**
: If the stream is associated with a POSIX file-descriptor, unify *Integer* with the descriptor number. SWI-Prolog extension used primarily for integration with foreign code. See also Sfileno() from `SWI-Stream.h`.

current_stream(?*Object*, ?*Mode*, ?*Stream***)**
: The predicate `current_stream/3` is used to access the status of a stream as well as to

[7]BUG: Backtracking does not give other aliases.

generate all open streams. *Object* is the name of the file opened if the stream refers to an open file, an integer file-descriptor if the stream encapsulates an operating-system stream or the atom `[]` if the stream refers to some other object. *Mode* is one of `read` or `write`.

set_stream_position(+*Stream*, +*Pos***)**
Set the current position of *Stream* to *Pos*. *Pos* is a term as returned by `stream_property/2` using the `position`(*Pos*) property. See also `seek/4`.

seek(+*Stream*, +*Offset*, +*Method*, -*NewLocation***)**
Reposition the current point of the given *Stream*. *Method* is one of `bof`, *current* or *eof*, indicating positioning relative to the start, current point or end of the underlying object. *NewLocation* is unified with the new offset, relative to the start of the stream.

If the seek modifies the current location, the line number and character position in the line are set to 0.

If the stream cannot be repostioned, a `reposition` error is raised. The predicate `seek/4` is compatible to Quintus Prolog, though the error conditions and signalling is ISO compliant. See also `stream_position/3`.

set_stream(+*Stream*, +*Attribute***)**
Modify an attribute of an existing stream. *Attribute* is in the current implemention only `alias`(*AliasName*) to set the alias of an already created stream. If *AliasName* is the name of one of the standard streams is used, this stream is rebound. Thus, `set_stream(S, current_input)` is the same as `set_input/1` and by setting the alias of a stream to `user_input`, etc. all user terminal input is read from this stream. See also `interactor/0`.

3.16.3 Switching Between Implicit and Explicit I/O

The predicates below can be used for switching between the implicit- and the explicit stream based I/O predicates.

set_input(+*Stream***)**
Set the current input stream to become *Stream*. Thus, open(file, read, Stream), set_input(Stream) is equivalent to see(file).

set_output(+*Stream***)**
Set the current output stream to become *Stream*.

current_input(-*Stream***)**
Get the current input stream. Useful to get access to the status predicates associated with streams.

current_output(-*Stream***)**
Get the current output stream.

3.17 Status of streams

wait_for_input(+*ListOfStreams*, -*ReadyList*, +*TimeOut***)**
Wait for input on one of the streams in *ListOfStreams* and return a list of streams on which input

is available in *ReadyList*. `wait_for_input/3` waits for at most *TimeOut* seconds. *Timeout* may be specified as a floating point number to specify fractions of a second. If *Timeout* equals 0, `wait_for_input/3` waits indefinitely. This predicate can be used to implement timeout while reading and to handle input from multiple sources. The following example will wait for input from the user and an explicitly opened second terminal. On return, *Inputs* may hold `user` or *P4* or both.

```
?- open('/dev/ttyp4', read, P4),
   wait_for_input([user, P4], Inputs, 0).
```

character_count(+*Stream*, -*Count***)**
Unify *Count* with the current character index. For input streams this is the number of characters read since the open, for output streams this is the number of characters written. Counting starts at 0.

line_count(+*Stream*, -*Count***)**
Unify *Count* with the number of lines read or written. Counting starts at 1.

line_position(+*Stream*, -*Count***)**
Unify *Count* with the position on the current line. Note that this assumes the position is 0 after the open. Tabs are assumed to be defined on each 8-th character and backspaces are assumed to reduce the count by one, provided it is positive.

fileerrors(-*Old*, +*New***)**
Define error behaviour on errors when opening a file for reading or writing. Valid values are the atoms `on` (default) and `off`. First *Old* is unified with the current value. Then the new value is set to *New*.[8]

3.18 Primitive character I/O

See section 3.2 for an overview of supported character representations.

nl
Write a newline character to the current output stream. On Unix systems `nl/0` is equivalent to `put(10)`.

nl(+*Stream***)**
Write a newline to *Stream*.

put(+*Char***)**
Write *Char* to the current output stream, *Char* is either an integer-expression evaluating to an ASCII value ($0 \leq Char \leq 255$) or an atom of one character.

put(+*Stream*, +*Char***)**
Write *Char* to *Stream*.

[8]Note that Edinburgh Prolog defines `fileerrors/0` and `nofileerrors/0`. As this does not allow you to switch back to the old mode I think this definition is better.

put_byte(+*Byte***)**
: Alias for `put/1`.

put_byte(+*Stream*, +*Byte***)**
: Alias for `put/2`

put_char(+*Char***)**
: Alias for `put_char/1`.

put(+*Stream*, +*Char***)**
: Alias for `put/2`

put_code(+*Code***)**
: Alias for `put/1`.

put_code(+*Stream*, +*Code***)**
: Alias for `put/2`

tab(+*Amount***)**
: Writes *Amount* spaces on the current output stream. *Amount* should be an expression that evaluates to a positive integer (see section 3.26).

tab(+*Stream*, +*Amount***)**
: Writes *Amount* spaces to *Stream*.

flush_output
: Flush pending output on current output stream. `flush_output/0` is automatically generated by `read/1` and derivatives if the current input stream is `user` and the cursor is not at the left margin.

flush_output(+*Stream***)**
: Flush output on the specified stream. The stream must be open for writing.

ttyflush
: Flush pending output on stream *user*. See also `flush_output/[0,1]`.

get_byte(-*Byte***)**
: Read the current input stream and unify the next byte with *Byte* (an integer between 0 and 255. *Byte* is unified with -1 on end of file.

get_byte(+*Stream*, -*Byte***)**
: Read the next byte from *Stream*.

get_code(-*Code***)**
: Read the current input stream and unify *Code* with the character code of the next character. *Char* is unified with -1 on end of file. See also `get_char/1`.

get_code(+*Stream*, -*Code***)**
: Read the next character-code from *Stream*.

get_char(-*Char***)**
: Read the current input stream and unify *Char* with the next character as a one-character-atom. See also `atom_chars/2`. On end-of-file, *Char* is unified to the atom `end_of_file`.

get_char(+*Stream*, -*Char***)**
Unify *Char* with the next character from *Stream* as a one-character-atom. See also `get_char/2`, `get_byte/2` and `get_code/2`.

get0(-*Char***)**
Edinburgh version of the ISO `get_byte/1` predicate.

get0(+*Stream*, -*Char***)**
Edinburgh version of the ISO `get_byte/2` predicate.

get(-*Char***)**
Read the current input stream and unify the next non-blank character with *Char*. *Char* is unified with -1 on end of file.

get(+*Stream*, -*Char***)**
Read the next non-blank character from *Stream*.

peek_byte(-*Byte***)**
Reads the next input byte like `get_byte/1`, but does not remove it from the input stream.

peek_byte(+*Stream*, -*Byte***)**
Reads the next input byte like `get_byte/2`, but does not remove it from the stream.

peek_code(-*Code***)**
Reads the next input code like `get_code/1`, but does not remove it from the input stream.

peek_code(+*Stream*, -*Code***)**
Reads the next input code like `get_code/2`, but does not remove it from the stream.

peek_char(-*Char***)**
Reads the next input character like `get_char/1`, but does not remove it from the input stream.

peek_char(+*Stream*, -*Char***)**
Reads the next input character like `get_char/2`, but does not remove it from the stream.

skip(+*Char***)**
Read the input until *Char* or the end of the file is encountered. A subsequent call to `get0/1` will read the first character after *Char*.

skip(+*Stream*, +*Char***)**
Skip input (as `skip/1`) on *Stream*.

get_single_char(-*Char***)**
Get a single character from input stream 'user' (regardless of the current input stream). Unlike `get0/1` this predicate does not wait for a return. The character is not echoed to the user's terminal. This predicate is meant for keyboard menu selection etc. If SWI-Prolog was started with the `-tty` option this predicate reads an entire line of input and returns the first non-blank character on this line, or the ASCII code of the newline (10) if the entire line consisted of blank characters.

at_end_of_stream
Succeeds after the last character of the current input stream has been read. Also succeeds if there is no valid current input stream.

at_end_of_stream(+*Stream***)**
Succeeds after the last character of the named stream is read, or *Stream* is not a valid input stream. The end-of-stream test is only available on buffered input stream (unbuffered input streams are rarely used, see `open/4`).

copy_stream_data(+*StreamIn*, +*StreamOut*, +*Len***)**
Copy *Len* bytes from stream *StreamIn* to *StreamOut*.

copy_stream_data(+*StreamIn*, +*StreamOut***)**
Copy data all (remaining) data from stream *StreamIn* to *StreamOut*.

3.19 Term reading and writing

This section describes the basic term reading and writing predicates. The predicates `term_to_atom/2`, `atom_to_term/3` and `sformat/3` provide means for translating atoms and strings to terms. The predicates `format/[1,2]` and `writef/2` provide formatted output.

There are two ways to manipulate the output format. The predicate `print/[1,2]` may be programmed using `portray/1`. The format of floating point numbers may be manipulated using the prolog_flag (see `current_prolog_flag/2`) `float_format`.

Reading is sensitive to the prolog_flag `character_escapes`, which controls the interpretation of the `\` character in quoted atoms and strings.

write_term(+*Term*, +*Options***)**
The predicate `write_term/2` is the generic form of all Prolog term-write predicates. Valid options are:

quoted(`true` *or* `false`**)**
If `true`, atoms and functors that needs quotes will be quoted. The default is `false`.

character_escapes(`true` *or* `false`**)**
If `true`, and `quoted`(*true*) is active, special characters in quoted atoms and strings are emitted as ISO escape-sequences. Default is taken from the reference module (see below).

ignore_ops(`true` *or* `false`**)**
If `true`, the generic term-representation (⟨*functor*⟩(⟨*args*⟩ ...)) will be used for all terms, Otherwise (default), operators, list-notation and `{}`/1 will be written using their special syntax.

module(*Module***)**
Define the reference module (default `user`). This defines the default value for the `character_escapes` option as well as the operator definitions to use. See also `op/3`.

numbervars(`true` *or* `false`**)**
If `true`, terms of the format `$VAR(N)`, where ⟨*N*⟩ is a positive integer, will be written as a variable name. The default is `false`.

portray(`true` *or* `false`**)**
If `true`, the hook `portray/1` is called before printing a term that is not a variable. If `portray/1` succeeds, the term is considered printed. See also `print/1`. The default is `false`. This option is an extension to the ISO write_term options.

max_depth(*Integer*)
If the term is nested deaper than *Integer*, print the remainder as eclipse (...). A 0 (zero) value (default) imposes no depth limit. This option also delimits the number of printed for a list. Example:

```
?- write_term(a(s(s(s(s(0)))), [a,b,c,d,e,f]), [max_depth(3)]).
a(s(s(...)), [a, b|...])

Yes
```

Used by the toplevel and debugger to limit screen output. See also the prolog-flags `toplevel_print_options` and `debugger_print_options`.

write_term(*+Stream, +Term, +Options*)
As `write_term/2`, but output is sent to *Stream* rather than the current output.

write_canonical(*+Term*)
Write *Term* on the current output stream using standard parenthesised prefix notation (i.e. ignoring operator declarations). Atoms that need quotes are quoted. Terms written with this predicate can always be read back, regardless of current operator declarations. Equivalent to `write_term/2` using the options `ignore_ops` and `quoted`.

write_canonical(*+Stream, +Term*)
Write *Term* in canonical form on *Stream*.

write(*+Term*)
Write *Term* to the current output, using brackets and operators where appropriate. See `current_prolog_flag/2` for controlling floating point output format.

write(*+Stream, +Term*)
Write *Term* to *Stream*.

writeq(*+Term*)
Write *Term* to the current output, using brackets and operators where appropriate. Atoms that need quotes are quoted. Terms written with this predicate can be read back with `read/1` provided the currently active operator declarations are identical.

writeq(*+Stream, +Term*)
Write *Term* to *Stream*, inserting quotes.

print(*+Term*)
Prints *Term* on the current output stream similar to `write/1`, but for each (sub)term of *Term* first the dynamic predicate `portray/1` is called. If this predicate succeeds *print* assumes the (sub)term has been written. This allows for user defined term writing.

print(*+Stream, +Term*)
Print *Term* to *Stream*.

portray(*+Term*)
A dynamic predicate, which can be defined by the user to change the behaviour of `print/1` on (sub)terms. For each subterm encountered that is not a variable `print/1` first calls

`portray/1` using the term as argument. For lists only the list as a whole is given to `portray/1`. If portray succeeds `print/1` assumes the term has been written.

read(-*Term***)**
Read the next Prolog term from the current input stream and unify it with *Term*. On a syntax error `read/1` displays an error message, attempts to skip the erroneous term and fails. On reaching end-of-file *Term* is unified with the atom `end_of_file`.

read(+*Stream*, -*Term***)**
Read *Term* from *Stream*.

read_clause(-*Term***)**
Equivalent to `read/1`, but warns the user for variables only occurring once in a term (singleton variables) which do not start with an underscore if `style_check(singleton)` is active (default). Used to read Prolog source files (see `consult/1`). New code should use `read_term/2` with the option `singletons(warning)`.

read_clause(+*Stream*, -*Term***)**
Read a clause from *Stream*. See `read_clause/1`.

read_term(-*Term*, +*Options***)**
Read a term from the current input stream and unify the term with *Term*. The reading is controlled by options from the list of *Options*. If this list is empty, the behaviour is the same as for `read/1`. The options are upward compatible to Quintus Prolog. The argument order is according to the ISO standard. Syntax-errors are always reported using exception-handling (see `catch/3`). Options:

variables(*Vars***)**
Unify *Vars* with a list of variables in the term. The variables appear in the order they have been read. See also `free_variables/2`. (ISO).

variable_names(*Vars***)**
Unify *Vars* with a list of '*Name* = *Var*', where *Name* is an atom describing the variable name and *Var* is a variable that shares with the corresponding variable in *Term*. (ISO).

singletons(*Vars***)**
As `variable_names`, but only reports the variables occurring only once in the *Term* read. Variables starting with an underscore ('_') are not included in this list. (ISO).

syntex_errors(*Atom***)**
If `error` (default), throw and exception on a syntax error. Other values are `fail`, which causes a message to be printed using `print_message/2`, after which the predicate fails, `quiet` which causes the predicate to fail silently and `dec10` which causes syntax errors to be printed, after which `read_term/[2,3]` continues reading the next term. Using `dec10`, `read_term/[2,3]` never fails. (Quintus, SICStus).

module(*Module***)**
Specify *Module* for operators, `character_escapes` flag and `double_quotes` flag. The value of the latter two is overruled if the corresponding `read_term/3` option is provided. If no module is specified, the current 'source-module' is used. (SWI-Prolog).

character_escapes(*Bool***)**
Defines how to read \ escape-sequences in quoted atoms. See the prolog-flags `character_escapes`, `current_prolog_flag/2`. (SWI-Prolog).

double_quotes(*Bool***)**
Defines how to read "..." strings. See the prolog-flags `double_quotes`, `current_prolog_flag/2`. (SWI-Prolog).

term_position(*Pos***)**
Unifies *Pos* with the starting position of the term read. *Pos* if of the same format as use by `stream_position/3`.

subterm_positions(*TermPos***)**
Describes the detailed layout of the term. The formats for the various types of terms if given below. All positions are character positions. If the input is related to a normal stream, these positions are relative to the start of the input, when reading from the terminal, they are relative to the start of the term.

From-To
Used for primitive types (atoms, numbers, variables).

string_position(From, To**)**
Used to indicate the position of a string enclosed in double quotes (`"`).

brace_term_position(From, To, Arg**)**
Term of the form `{...}`, as used in DCG rules. *Arg* describes the argument.

list_position(From, To, Elms, Tail**)**
A list. *Elms* describes the positions of the elements. If the list specifies the tail as `|`⟨*TailTerm*⟩, *Tail* is unified with the term-position of the tail, otherwise with the atom `none`.

term_position(From, To, FFrom, FTo, SubPos**)**
Used for a compound term not matching one of the above. *FFrom* and *FTo* describe the position of the functor. *SubPos* is a list, each element of which describes the term-position of the corresponding subterm.

read_term(*+Stream, -Term, +Options***)**
Read term with options from *Stream*. See `read_term/2`.

read_history(*+Show, +Help, +Special, +Prompt, -Term, -Bindings***)**
Similar to `read_term/2` using the option `variable_names`, but allows for history substitutions. `read_history/6` is used by the top level to read the user's actions. *Show* is the command the user should type to show the saved events. *Help* is the command to get an overview of the capabilities. *Special* is a list of commands that are not saved in the history. *Prompt* is the first prompt given. Continuation prompts for more lines are determined by `prompt/2`. A `%w` in the prompt is substituted by the event number. See section 2.7 for available substitutions.

SWI-Prolog calls `read_history/6` as follows:

```
read_history(h, '!h', [trace], '%w ?- ', Goal, Bindings)
```

prompt(*-Old, +New***)**
Set prompt associated with `read/1` and its derivatives. *Old* is first unified with the current

prompt. On success the prompt will be set to *New* if this is an atom. Otherwise an error message is displayed. A prompt is printed if one of the read predicates is called and the cursor is at the left margin. It is also printed whenever a newline is given and the term has not been terminated. Prompts are only printed when the current input stream is *user*.

prompt1(+*Prompt***)**
Sets the prompt for the next line to be read. Continuation lines will be read using the prompt defined by `prompt/2`.

3.20 Analysing and Constructing Terms

functor(*?Term, ?Functor, ?Arity***)**
Succeeds if *Term* is a term with functor *Functor* and arity *Arity*. If *Term* is a variable it is unified with a new term holding only variables. `functor/3` silently fails on instantiation faults[9] If *Term* is an atom or number, *Functor* will be unified with *Term* and arity will be unified with the integer 0 (zero).

arg(*?Arg, ?Term, ?Value***)**
Term should be instantiated to a term, *Arg* to an integer between 1 and the arity of *Term*. *Value* is unified with the *Arg*-th argument of *Term*. *Arg* may also be unbound. In this case *Value* will be unified with the successive arguments of the term. On successful unification, *Arg* is unified with the argument number. Backtracking yields alternative solutions.[10] The predicate `arg/3` fails silently if $Arg = 0$ or $Arg > arity$ and raises the exception `domain_error(not_less_then_zero, Arg)` if $Arg < 0$.

setarg(+*Arg*, +*Term*, +*Value***)**
Extra-logical predicate. Assigns the *Arg*-th argument of the compound term *Term* with the given *Value*. The assignment is undone if backtracking brings the state back into a position before the `setarg/3` call.

This predicate may be used for destructive assignment to terms, using them as and extra-logical storage bin.

?Term **=..** *?List*
List is a list which head is the functor of *Term* and the remaining arguments are the arguments of the term. Each of the arguments may be a variable, but not both. This predicate is called 'Univ'. Examples:

```
?- foo(hello, X) =.. List.

List = [foo, hello, X]

?- Term =.. [baz, foo(1)]

Term = baz(foo(1))
```

[9] In version 1.2 instantiation faults led to error messages. The new version can be used to do type testing without the need to catch illegal instantiations first.

[10] The instantiation pattern (-, +, ?) is an extension to 'standard' Prolog.

numbervars(*+Term, +Functor, +Start, -End***)**

Unify the free variables of *Term* with a term constructed from the atom *Functor* with one argument. The argument is the number of the variable. Counting starts at *Start*. *End* is unified with the number that should be given to the next variable. Example:

```
?- numbervars(foo(A, B, A), this_is_a_variable, 0, End).

A = this_is_a_variable(0)
B = this_is_a_variable(1)
End = 2
```

In Edinburgh Prolog the second argument is missing. It is fixed to be `$VAR`.

free_variables(*+Term, -List***)**

Unify *List* with a list of variables, each sharing with a unique variable of *Term*. For example:

```
?- free_variables(a(X, b(Y, X), Z), L).

L = [G367, G366, G371]
X = G367
Y = G366
Z = G371
```

copy_term(*+In, -Out***)**

Make a copy of term *In* and unify the result with *Out*. Ground parts of *In* are shared by *Out*. Provided *In* and *Out* have no sharing variables before this call they will have no sharing variables afterwards. `copy_term/2` is semantically equivalent to:

```
copy_term(In, Out) :-
        recorda(copy_key, In, Ref),
        recorded(copy_key, Out, Ref),
        erase(Ref).
```

3.21 Analysing and constructing atoms

These predicates convert between Prolog constants and lists of ASCII values. The predicates `atom_codes/2`, `number_codes/2` and `name/2` behave the same when converting from a constant to a list of ASCII values. When converting the other way around, `atom_codes/2` will generate an atom, `number_codes/2` will generate a number or exception and `name/2` will return a number if possible and an atom otherwise.

The ISO standard defines `atom_chars/2` to describe the 'broken-up' atom as a list of one-character atoms instead of a list of codes. Upto version 3.2.x, SWI-Prolog's `atom_chars/2` behaved, compatible to Quintus and SICStus Prolog, like atom_codes. As of 3.3.x SWI-Prolog `atom_codes/2` and `atom_chars/2` are compliant to the ISO standard.

To ease the pain of all variations in the Prolog community, all SWI-Prolog predicates behave as flexible as possible. This implies the 'list-side' accepts either a code-list or a char-list and the 'atom-side' accept all atomic types (atom, number and string).

atom_codes(?Atom, ?String**)**

Convert between an atom and a list of ASCII values. If *Atom* is instantiated, if will be translated into a list of ASCII values and the result is unified with *String*. If *Atom* is unbound and *String* is a list of ASCII values, it will *Atom* will be unified with an atom constructed from this list.

atom_chars(?Atom, ?CharList**)**

As `atom_codes/2`, but *CharList* is a list of one-character atoms rather than a list of ASCII values[11].

```
?- atom_chars(hello, X).

X = [h, e, l, l, o]
```

char_code(?Atom, ?ASCII**)**

Convert between character and ASCII value for a single character.[12]

number_chars(?Number, ?CharList**)**

Similar to `atom_chars/2`, but converts between a number and its representation as a list of one-character atoms. Fails with a `representation_error` if *Number* is unbound and *CharList* does not describe a number.

number_codes(?Number, ?CodeList**)**

As `number_chars/2`, but converts to a list of character codes (normally ASCII values) rather than one-character atoms. In the mode -, +, both predicates behave identically to improve handling of non-ISO source.

name(?AtomOrInt, ?String**)**

String is a list of ASCII values describing *Atom*. Each of the arguments may be a variable, but not both. When *String* is bound to an ASCII value list describing an integer and *Atom* is a variable *Atom* will be unified with the integer value described by *String* (e.g. '`name(N, "300"), 400 is N + 100`' succeeds).

int_to_atom(+Int, +Base, -Atom**)**

Convert *Int* to an ASCII representation using base *Base* and unify the result with *Atom*. If *Base* $\neq 10$ the base will be prepended to *Atom*. *Base* $= 0$ will try to interpret *Int* as an ASCII value and return `0'`$\langle c \rangle$. Otherwise $2 \leq$ *Base* ≤ 36. Some examples are given below.

int_to_atom(45, 2, A)	$\longrightarrow$	$A = 2'101101$
int_to_atom(97, 0, A)	$\longrightarrow$	$A = 0'a$
int_to_atom(56, 10, A)	$\longrightarrow$	$A = 56$

int_to_atom(+Int, -Atom**)**

Equivalent to `int_to_atom(Int, 10, Atom)`.

[11] Upto version 3.2.x, `atom_chars/2` behaved as the current `atom_codes/2`. The current definition is compliant with the ISO standard

[12] This is also called `atom_char/2` in older versions of SWI-Prolog as well as some other Prolog implementations. `atom_char/2` is available from the library `backcomp.pl`

term_to_atom(*?Term, ?Atom***)**
Succeeds if *Atom* describes a term that unifies with *Term*. When *Atom* is instantiated *Atom* is converted and then unified with *Term*. If *Atom* has no valid syntax, a `syntax_error` exception is raised. Otherwise *Term* is "written" on *Atom* using `write/1`.

atom_to_term(*+Atom, -Term, -Bindings***)**
Use *Atom* as input to `read_term/2` using the option `variable_names` and return the read term in *Term* and the variable bindings in *Bindings*. *Bindings* is a list of *Name* = *Var* couples, thus providing access to the actual variable names. See also `read_term/2`. If *Atom* has no valid syntax, a `syntax_error` exception is raised.

atom_concat(*?Atom1, ?Atom2, ?Atom3***)**
Atom3 forms the concatenation of *Atom1* and *Atom2*. At least two of the arguments must be instantiated to atoms, integers or floating point numbers. For ISO compliance, the instantiation-pattern -, -, + is allowed too, non-deterministically splitting the 3-th argument into two parts (as `append/3` does for lists). See also `string_concat/3`.

concat_atom(*+List, -Atom***)**
List is a list of atoms, integers or floating point numbers. Succeeds if *Atom* can be unified with the concatenated elements of *List*. If *List* has exactly 2 elements it is equivalent to `atom_concat/3`, allowing for variables in the list.

concat_atom(*?List, +Separator, ?Atom***)**
Creates an atom just like `concat_atom/2`, but inserts *Separator* between each pair of atoms. For example:

```
?- concat_atom([gnu, gnat], ', ', A).

A = 'gnu, gnat'
```

This predicate can also be used to split atoms by instantiating *Separator* and *Atom*:

```
?- concat_atom(L, -, 'gnu-gnat').

L = [gnu, gnat]
```

atom_length(*+Atom, -Length***)**
Succeeds if *Atom* is an atom of *Length* characters long. This predicate also works for integers and floats, expressing the number of characters output when given to `write/1`.

atom_prefix(*+Atom, +Prefix***)**
Succeeds if *Atom* starts with the characters from *Prefix*. Its behaviour is equivalent to `?- concat(`*`Prefix`*`, _, `*`Atom`*`)`, but avoids the construction of an atom for the 'remainder'.

sub_atom(*+Atom, ?Before, ?Len, ?After, ?Sub***)**
ISO predicate for breaking atoms. It maintains the following relation: *Sub* is a sub-atom of *Atom* that starts at *Before*, has *Len* characters and *Atom* contains *After* characters after the match.

```
?- sub_atom(abc, 1, 1, A, S).

A = 1, S = b
```

The implementation minimalises non-determinism and creation of atoms. This is a very flexible predicate that can do search, prefix- and suffix-matching, etc.

3.22 Classifying characters

SWI-Prolog offers two comprehensive predicates for classifying characters and character-codes. These predicates are defined as built-in predicates to exploit the C-character classification's handling of *locale* (handling of local character-sets). These predicates are fast, logical and deterministic if applicable.

In addition, there is the library library(`ctype`) providing compatibility to some other Prolog systems. The predicates of this library are defined in terms of `code_type/2`.

char_type(*?Char, ?Type***)**
Tests or generates alternative *Type*s or *Char*s. The character-types are inspired by the standard C `<ctype.h>` primitives.

alnum
Char is a letter (upper- or lowercase) or digit.

alpha
Char is a letter (upper- or lowercase).

csym
Char is a letter (upper- or lowercase), digit or the underscore (_). These are valid C- and Prolog symbol characters.

csymf
Char is a letter (upper- or lowercase) or the underscore (_). These are valid first characters for C- and Prolog symbols

ascii
Char is a 7-bits ASCII character (0..127).

white
Char is a space or tab. E.i. white space inside a line.

cntrl
Char is an ASCII control-character (0..31).

digit
Char is a digit.

digit(*Weigth***)**
Char is a digit with value *Weigth*. I.e. `char_type(X, digit(6)` yields *X* = `'6'`. Useful for parsing numbers.

xdigit(*Weigth***)**
Char is a haxe-decimal digit with value *Weigth*. I.e. `char_type(a, xdigit(X)` yields *X* = `'10'`. Useful for parsing numbers.

graph
: *Char* produces a visible mark on a page when printed. Note that the space is not included!

lower
: *Char* is a lower-case letter.

lower(*Upper***)**
: *Char* is a lower-case version of *Upper*. Only true if *Char* is lowercase and *Upper* uppercase.

to_lower(*Upper***)**
: *Char* is a lower-case version of *Upper*. For non-letters, or letter without case, *Char* and *Lower* are the same.

upper
: *Char* is an upper-case letter.

upper(*Lower***)**
: *Char* is an upper-case version of *Lower*. Only true if *Char* is uppercase and *Lower* lowercase.

to_upper(*Lower***)**
: *Char* is an upper-case version of *Lower*. For non-letters, or letter without case, *Char* and *Lower* are the same.

punct
: *Char* is a punctuation character. This is a `graph` character that is not a letter or digit.

space
: *Char* is some form of layout character (tab, vertical-tab, newline, etc.).

end_of_file
: *Char* is -1.

end_of_line
: *Char* ends a line (ASCII: 10..13).

newline
: *Char* is a the newline character (10).

period
: *Char* counts as the end of a sentence (.,!,?).

quote
: *Char* is a quote-character (`"`, `'`, `` ` ``).

paren(*Close***)**
: *Char* is an open-parenthesis and *Close* is the corresponding close-parenthesis.

code_type(*?Code, ?Type***)**
: As `char_type/2`, but uses character-codes rather than one-character atoms. Please note that both predicates are as flexible as possible. They handle either representation if the argument is instantiated and only will instantiate with an integer code or one-character atom depending of the version used. See also the prolog-flag `double_quotes`, `atom_chars/2` and `atom_codes/2`.

3.23 Representing text in strings

SWI-Prolog supports the data type *string*. Strings are a time and space efficient mechanism to handle text text in Prolog. Strings are stores as a byte array on the global (term) stack and thus destroyed on backtracking and reclaimed by the garbage collector.

Strings were added to SWI-Prolog based on an early draft of the ISO standard, offerring a mechanism to represent temporary character data efficiently. As SWI-Prolog strings can handle 0-bytes, they are frequently used through the foreign language interface (section 5) for storing arbitrary byte-sequences.

Starting with version 3.3, SWI-Prolog offers garbage collection on the atom-space as well as representing 0-bytes in atoms. Although strings and atoms still have different features, new code should consider using atoms to avoid too many representations for text as well as for compatibility to other Prolog systems. Below are some of the differences:

- *creation*
  Creating strings is fast, as the data is simply copied to the global stack. Atoms are unique and therefore more expensive in terms of memory and time to create. On the other hand, if the same text has to be represented multiple times, atoms are more efficient.

- *destruction*
  Backtracking destroys strings at no cost. They are cheap to handle by the garbage collector, but it should be noted that extensive use of strings will cause many garbage collections. Atom garbage collection is generally faster.

See also the prolog-flag `double_quotes`.

string_to_atom(*?String, ?Atom***)**
Logical conversion between a string and an atom. At least one of the two arguments must be instantiated. *Atom* can also be an integer or floating point number.

string_to_list(*?String, ?List***)**
Logical conversion between a string and a list of ASCII characters. At least one of the two arguments must be instantiated.

string_length(*+String, -Length***)**
Unify *Length* with the number of characters in *String*. This predicate is functionally equivalent to `atom_length/2` and also accepts atoms, integers and floats as its first argument.

string_concat(*?String1, ?String2, ?String3***)**
Similar to `atom_concat/3`, but the unbound argument will be unified with a string object rather than an atom. Also, if both *String1* and *String2* are unbound and *String3* is bound to text, it breaks *String3*, unifying the start with *String1* and the end with *String2* as append does with lists. Note that this is not particularly fast on long strings as for each redo the system has to create two entirely new strings, while the list equivalent only creates a single new list-cell and moves some pointers around.

sub_string(*+String, ?Start, ?Length, ?After, ?Sub***)**
Sub is a substring of *String* starting at *Start*, with length *Length* and *String* has *After* characters left after the match. See also `sub_atom/5`.

3.24 Operators

Operators are defined to improve the readibility of source-code. For example, without operators, to write `2*3+4*5` one would have to write `+(*(2,3),*(4,5))`. In Prolog, a number of operators have been predefined. All operators, except for the comma (,) can be redefined by the user.

Some care has to be taken before defining new operators. Defining too many operators might make your source 'natural' looking, but at the same time lead to hard to understand the limits of your syntax. To ease the pain, as of SWI-Prolog 3.3.0, operators are local to the module in which they are defined. The module-table of the module `user` acts as default table for all modules. This global table can be modified explictly from inside a module:

```
:- module(prove,
          [ prove/1
          ]).

:- op(900, xfx, user:(=>)).
```

Unlike what many users think, operators and quoted atoms have no relation: defining a atom as an operator does **not** influence parsing characters into atoms and quoting an atom does **not** stop it from acting as an operator. To stop an atom acting as an operator, enclose it in braces like this: (myop).

op(+*Precedence*, +*Type*, :*Name***)**

Declare *Name* to be an operator of type *Type* with precedence *Precedence*. *Name* can also be a list of names, in which case all elements of the list are declared to be identical operators. *Precedence* is an integer between 0 and 1200. Precedence 0 removes the declaration. *Type* is one of: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`, `fy` or `fx`. The '`f`' indicates the position of the functor, while `x` and `y` indicate the position of the arguments. '`y`' should be interpreted as "on this position a term with precedence lower or equal to the precedence of the functor should occur". For '`x`' the precedence of the argument must be strictly lower. The precedence of a term is 0, unless its principal functor is an operator, in which case the precedence is the precedence of this operator. A term enclosed in brackets `(...)` has precedence 0.

The predefined operators are shown in table 3.1. Note that all operators can be redefined by the user.

current_op(?*Precedence*, ?*Type*, ?:*Name***)**

Succeeds when *Name* is currently defined as an operator of type *Type* with precedence *Precedence*. See also `op/3`.

3.25 Character Conversion

Although I wouldn't really know for what you would like to use these features, they are provided for ISO complicancy.

char_conversion(+*CharIn*, +*CharOut***)**

Define that term-input (see `read_term/3`) maps each character read as *CharIn* to the character *CharOut*. Character conversion is only executed if the prolog-flag `char_conversion` is set to `true` and not inside quoted atoms or strings. The initial table maps each character onto itself. See also `current_char_conversion/2`.

1200	xfx	`-->`, `:-`
1200	fx	`:-`, `?-`
1150	fx	`dynamic`, `multifile`, `module_transparent`, `discontiguous`, `volatile`, `initialization`
1100	xfy	`;`, `|`
1050	xfy	`->`
1000	xfy	`,`
954	xfy	`\`
900	fy	`\+`
900	fx	`~`
700	xfx	`<`, `=`, `=..`, `=@=`, `=:=`, `=<`, `==`, `=\=`, `>`, `>=`, `@<`, `@=<`, `@>`, `@>=`, `\=`, `\==`, `is`
600	xfy	`:`
500	yfx	`+`, `-`, `/\`, `\/`, `xor`
500	fx	`+`, `-`, `?`, `\`
400	yfx	`*`, `/`, `//`, `<<`, `>>`, `mod`, `rem`
200	xfx	`**`
200	xfy	`^`

Table 3.1: System operators

current_char_conversion(?*CharIn*, ?*CharOut***)**
Queries the current character conversion-table. See `char_conversion/2` for details.

3.26 Arithmetic

Arithmetic can be divided into some special purpose integer predicates and a series of general predicates for floating point and integer arithmetic as appropriate. The integer predicates are as "logical" as possible. Their usage is recommended whenever applicable, resulting in faster and more "logical" programs.

The general arithmetic predicates are optionally compiled now (see `set_prolog_flag/2` and the `-O` command line option). Compiled arithmetic reduces global stack requirements and improves performance. Unfortunately compiled arithmetic cannot be traced, which is why it is optional.

The general arithmetic predicates all handle *expressions*. An expression is either a simple number or a *function*. The arguments of a function are expressions. The functions are described in section 3.27.

between(+*Low*, +*High*, ?*Value***)**
Low and *High* are integers, *High* $\geq$ *Low*. If *Value* is an integer, *Low* $\leq$ *Value* $\leq$ *High*. When *Value* is a variable it is successively bound to all integers between *Low* and *High*.

succ(?*Int1*, ?*Int2***)**
Succeeds if *Int2* $=$ *Int1* $+ 1$. At least one of the arguments must be instantiated to an integer.

plus(?*Int1*, ?*Int2*, ?*Int3***)**
Succeeds if *Int3* $=$ *Int1* $+$ *Int2*. At least two of the three arguments must be instantiated to integers.

+Expr1 **>** *+Expr2*
: Succeeds when expression *Expr1* evaluates to a larger number than *Expr2*.

+Expr1 **<** *+Expr2*
: Succeeds when expression *Expr1* evaluates to a smaller number than *Expr2*.

+Expr1 **=<** *+Expr2*
: Succeeds when expression *Expr1* evaluates to a smaller or equal number to *Expr2*.

+Expr1 **>=** *+Expr2*
: Succeeds when expression *Expr1* evaluates to a larger or equal number to *Expr2*.

+Expr1 **=\=** *+Expr2*
: Succeeds when expression *Expr1* evaluates to a number non-equal to *Expr2*.

+Expr1 **=:=** *+Expr2*
: Succeeds when expression *Expr1* evaluates to a number equal to *Expr2*.

-Number **is** *+Expr*
: Succeeds when *Number* has successfully been unified with the number *Expr* evaluates to. If *Expr* evaluates to a float that can be represented using an integer (i.e. the value is integer and within the range that can be described by Prolog's integer representation), *Expr* is unified with the integer value.

Note that normally, `is/2` will be used with unbound left operand. If equality is to be tested, =:=/2 should be used. For example:

`?- 1.0 is sin(pi/2).`	Fails!. sin(pi/2) evaluates to 1.0, but `is/2` will represent this as the integer 1, after which unify will fail.
`?- 1.0 is float(sin(pi/2)).`	Succeeds, as the `float/1` function forces the result to be float.
`?- 1.0 =:= sin(pi/2).`	Succeeds as expected.

3.27 Arithmetic Functions

Arithmetic functions are terms which are evaluated by the arithmetic predicates described above. SWI-Prolog tries to hide the difference between integer arithmetic and floating point arithmetic from the Prolog user. Arithmetic is done as integer arithmetic as long as possible and converted to floating point arithmetic whenever one of the arguments or the combination of them requires it. If a function returns a floating point value which is whole it is automatically transformed into an integer. There are three types of arguments to functions:

Expr	Arbitrary expression, returning either a floating point value or an integer.
IntExpr	Arbitrary expression that should evaluate into an integer.
Int	An integer.

In case integer addition, subtraction and multiplication would lead to an integer overflow the operands are automatically converted to floating point numbers. The floating point functions (`sin/1`, `exp/1`, etc.) form a direct interface to the corresponding C library functions used to compile SWI-Prolog. Please refer to the C library documentation for details on precision, error handling, etc.

- *+Expr*
: $Result = -Expr$

+Expr1 **+** *+Expr2*
: $Result = Expr1 + Expr2$

+Expr1 **-** *+Expr2*
: $Result = Expr1 - Expr2$

+Expr1 ***** *+Expr2*
: $Result = Expr1 \times Expr2$

+Expr1 **/** *+Expr2*
: $Result = \frac{Expr1}{Expr2}$

+IntExpr1 **mod** *+IntExpr2*
: Modulo: *Result* = *IntExpr1* - (*IntExpr1* // *IntExpr2*) $\times$ *IntExpr2* The function `mod/2` is implemented using the C `%` operator. It's behaviour with negtive values is illustrated in the table below.

2	=	17	mod	5
2	=	17	mod	-5
-2	=	-17	mod	5
-2	=	-17	mod	5

+IntExpr1 **rem** *+IntExpr2*
: Remainder of division: *Result* = float_fractional_part(*IntExpr1*/*IntExpr2*)

+IntExpr1 **//** *+IntExpr2*
: Integer division: *Result* = truncate(*Expr1*/*Expr2*)

abs(*+Expr***)**
: Evaluate *Expr* and return the absolute value of it.

sign(*+Expr***)**
: Evaluate to -1 if $Expr < 0$, 1 if $Expr > 0$ and 0 if $Expr = 0$.

max(*+Expr1, +Expr2***)**
: Evaluates to the largest of both *Expr1* and *Expr2*.

min(*+Expr1, +Expr2***)**
: Evaluates to the smallest of both *Expr1* and *Expr2*.

`.`(+*Int*, *[]***)**
A list of one element evaluates to the element. This implies `"a"` evaluates to the ASCII value of the letter 'a' (97). This option is available for compatibility only. It will not work if '`style_check(+string)`' is active as `"a"` will then be transformed into a string object. The recommended way to specify the ASCII value of the letter 'a' is `0'a`.

random(+*Int***)**
Evaluates to a random integer *i* for which $0 \leq i < Int$. The seed of this random generator is determined by the system clock when SWI-Prolog was started.

round(+*Expr***)**
Evaluates *Expr* and rounds the result to the nearest integer.

integer(+*Expr***)**
Same as `round/1` (backward compatibility).

float(+*Expr***)**
Translate the result to a floating point number. Normally, Prolog will use integers whenever possible. When used around the 2nd argument of `is/2`, the result will be returned as a floating point number. In other contexts, the operation has no effect.

float_fractional_part(+*Expr***)**
Fractional part of a floating-point number. Negative if *Expr* is negative, 0 if *Expr* is integer.

float_integer_part(+*Expr***)**
Integer part of floating-point number. Negative if *Expr* is negative, *Expr* if *Expr* is integer.

truncate(+*Expr***)**
Truncate *Expr* to an integer. Same as `float_integer_part/1`.

floor(+*Expr***)**
Evaluates *Expr* and returns the largest integer smaller or equal to the result of the evaluation.

ceiling(+*Expr***)**
Evaluates *Expr* and returns the smallest integer larger or equal to the result of the evaluation.

ceil(+*Expr***)**
Same as `ceiling/1` (backward compatibility).

+*IntExpr* **`>>`** +*IntExpr*
Bitwise shift *IntExpr1* by *IntExpr2* bits to the right.

+*IntExpr* **`<<`** +*IntExpr*
Bitwise shift *IntExpr1* by *IntExpr2* bits to the left.

+*IntExpr* **`\/`** +*IntExpr*
Bitwise 'or' *IntExpr1* and *IntExpr2*.

+*IntExpr* **`/\`** +*IntExpr*
Bitwise 'and' *IntExpr1* and *IntExpr2*.

+*IntExpr* **xor** +*IntExpr*
Bitwise 'exclusive or' *IntExpr1* and *IntExpr2*.

\ *+IntExpr*
Bitwise negation.

sqrt(*+Expr***)**
$\mathit{Result} = \sqrt{\mathit{Expr}}$

sin(*+Expr***)**
$\mathit{Result} = \sin \mathit{Expr}$. *Expr* is the angle in radians.

cos(*+Expr***)**
$\mathit{Result} = \cos \mathit{Expr}$. *Expr* is the angle in radians.

tan(*+Expr***)**
$\mathit{Result} = \tan \mathit{Expr}$. *Expr* is the angle in radians.

asin(*+Expr***)**
$\mathit{Result} = \arcsin \mathit{Expr}$. *Result* is the angle in radians.

acos(*+Expr***)**
$\mathit{Result} = \arccos \mathit{Expr}$. *Result* is the angle in radians.

atan(*+Expr***)**
$\mathit{Result} = \arctan \mathit{Expr}$. *Result* is the angle in radians.

atan(*+YExpr, +XExpr***)**
$\mathit{Result} = \arctan \frac{\mathit{YExpr}}{\mathit{XExpr}}$. *Result* is the angle in radians. The return value is in the range $[-\pi \ldots \pi]$. Used to convert between rectangular and polar coordinate system.

log(*+Expr***)**
$\mathit{Result} = \ln \mathit{Expr}$

log10(*+Expr***)**
$\mathit{Result} = \lg \mathit{Expr}$

exp(*+Expr***)**
$\mathit{Result} = e^{\mathit{Expr}}$

+Expr1 ****** *+Expr2*
$\mathit{Result} = \mathit{Expr1}^{\mathit{Expr2}}$

+Expr1 **^** *+Expr2*
Same as **/2. (backward compatibility).

pi
Evaluates to the mathematical constant π (3.141593).

e
Evaluates to the mathematical constant e (2.718282).

cputime
Evaluates to a floating point number expressing the CPU time (in seconds) used by Prolog up till now. See also `statistics/2` and `time/1`.

3.28 Adding Arithmetic Functions

Prolog predicates can be given the role of arithmetic function. The last argument is used to return the result, the arguments before the last are the inputs. Arithmetic functions are added using the predicate `arithmetic_function/1`, which takes the head as its argument. Arithmetic functions are module sensitive, that is they are only visible from the module in which the function is defined and declared. Global arithmetic functions should be defined and registered from module `user`. Global definitions can be overruled locally in modules. The builtin functions described above can be redefined as well.

arithmetic_function(+*Head***)**

Register a Prolog predicate as an arithmetic function (see `is/2`, `>/2`, etc.). The Prolog predicate should have one more argument than specified by *Head*, which it either a term *Name/Arity*, an atom or a complex term. This last argument is an unbound variable at call time and should be instantiated to an integer or floating point number. The other arguments are the parameters. This predicate is module sensitive and will declare the arithmetic function only for the context module, unless declared from module `user`. Example:

```
1 ?- [user].
:- arithmetic_function(mean/2).

mean(A, B, C) :-
        C is (A+B)/2.
user compiled, 0.07 sec, 440 bytes.

Yes
2 ?- A is mean(4, 5).

A = 4.500000
```

current_arithmetic_function(?*Head***)**

Successively unifies all arithmetic functions that are visible from the context module with *Head*.

3.29 List Manipulation

is_list(+*Term***)**

Succeeds if *Term* is bound to the empty list (`[]`) or a term with functor ‘`.`’ and arity 2.

proper_list(+*Term***)**

Equivalent to `is_list/1`, but also requires the tail of the list to be a list (recursively). Examples:

```
is_list([x|A])          % true
proper_list([x|A])      % false
```

append(_?List1, ?List2, ?List3_**)**
Succeeds when _List3_ unifies with the concatenation of _List1_ and _List2_. The predicate can be used with any instantiation pattern (even three variables).

member(_?Elem, ?List_**)**
Succeeds when _Elem_ can be unified with one of the members of _List_. The predicate can be used with any instantiation pattern.

memberchk(_?Elem, +List_**)**
Equivalent to `member/2`, but leaves no choice point.

delete(_+List1, ?Elem, ?List2_**)**
Delete all members of _List1_ that simultaneously unify with _Elem_ and unify the result with _List2_.

select(_?Elem, ?List, ?Rest_**)**
Select _Elem_ from _List_ leaving _Rest_. It behaves as `member/2`, returning the remaining elements in _Rest_. Note that besides selecting elements from a list, it can also be used to insert elements.[13]

nth0(_?Index, ?List, ?Elem_**)**
Succeeds when the _Index_-th element of _List_ unifies with _Elem_. Counting starts at 0.

nth1(_?Index, ?List, ?Elem_**)**
Succeeds when the _Index_-th element of _List_ unifies with _Elem_. Counting starts at 1.

last(_?Elem, ?List_**)**
Succeeds if _Elem_ unifies with the last element of _List_. If _List_ is a proper list `last/2` is deterministic. If _List_ has an unbound tail, backtracking will cause _List_ to grow.

reverse(_+List1, -List2_**)**
Reverse the order of the elements in _List1_ and unify the result with the elements of _List2_.

flatten(_+List1, -List2_**)**
Transform _List1_, possibly holding lists as elements into a 'flat' list by replacing each list with its elements (recursively). Unify the resulting flat list with _List2_. Example:

```
?- flatten([a, [b, [c, d], e]], X).

X = [a, b, c, d, e]
```

length(_?List, ?Int_**)**
Succeeds if _Int_ represents the number of elements of list _List_. Can be used to create a list holding only variables.

merge(_+List1, +List2, -List3_**)**
List1 and _List2_ are lists, sorted to the standard order of terms (see section 3.6). _List3_ will be unified with an ordered list holding both the elements of _List1_ and _List2_. Duplicates are **not** removed.

[13] BUG: Upto SWI-Prolog 3.3.10, the definition of this predicate was not according to the de-facto standard. The first two arguments were in the wrong order.

3.30 Set Manipulation

is_set(+*Set***)**
Succeeds if *Set* is a proper list (see `proper_list/1`) without duplicates.

list_to_set(+*List*, -*Set***)**
Unifies *Set* with a list holding the same elements as *List* in the same order. If *list* contains duplicates, only the first is retained. See also `sort/2`. Example:

```
?- list_to_set([a,b,a], X)

X = [a,b]
```

intersection(+*Set1*, +*Set2*, -*Set3***)**
Succeeds if *Set3* unifies with the intersection of *Set1* and *Set2*. *Set1* and *Set2* are lists without duplicates. They need not be ordered.

subtract(+*Set*, +*Delete*, -*Result***)**
Delete all elements of set 'Delete' from 'Set' and unify the resulting set with 'Result'.

union(+*Set1*, +*Set2*, -*Set3***)**
Succeeds if *Set3* unifies with the union of *Set1* and *Set2*. *Set1* and *Set2* are lists without duplicates. They need not be ordered.

subset(+*Subset*, +*Set***)**
Succeeds if all elements of *Subset* are elements of *Set* as well.

merge_set(+*Set1*, +*Set2*, -*Set3***)**
Set1 and *Set2* are lists without duplicates, sorted to the standard order of terms. *Set3* is unified with an ordered list without duplicates holding the union of the elements of *Set1* and *Set2*.

3.31 Sorting Lists

sort(+*List*, -*Sorted***)**
Succeeds if *Sorted* can be unified with a list holding the elements of *List*, sorted to the standard order of terms (see section 3.6). Duplicates are removed. Implemented by translating the input list into a temporary array, calling the C-library function `qsort(3)` using `PL_compare()` for comparing the elements, after which the result is translated into the result list.

msort(+*List*, -*Sorted***)**
Equivalent to `sort/2`, but does not remove duplicates.

keysort(+*List*, -*Sorted***)**
List is a proper list whose elements are `Key-Value`, that is, terms whose principal functor is (-)/2, whose first argument is the sorting key, and whose second argument is the satellite data to be carried along with the key. `keysort/2` sorts *List* like `msort/2`, but only compares the keys. Can be used to sort terms not on standard order, but on any criterion that can be expressed on a multi-dimensional scale. Sorting on more than one criterion can be done using terms as keys, putting the first criterion as argument 1, the second as argument 2, etc. The order of multiple elements that have the same *Key* is not changed.

predsort(+*Pred*, +*List*, -*Sorted***)**

Sorts similar to `sort/2`, but determines the order of two terms by calling *Pred*(-*Delta*, +*E1*, +*E2*). This call must unify *Delta* with one of `<`, const`>` or `=`. If built-in predicate `compare/3` is used, the result is the same as `sort/2`. See also `keysort/2`.[14]

3.32 Finding all Solutions to a Goal

findall(+*Var*, +*Goal*, -*Bag***)**

Creates a list of the instantiations *Var* gets successively on backtracking over *Goal* and unifies the result with *Bag*. Succeeds with an empty list if *Goal* has no solutions. `findall/3` is equivalent to `bagof/3` with all free variables bound with the existence operator (`^`), except that `bagof/3` fails when goal has no solutions.

bagof(+*Var*, +*Goal*, -*Bag***)**

Unify *Bag* with the alternatives of *Var*, if *Goal* has free variables besides the one sharing with *Var* bagof will backtrack over the alternatives of these free variables, unifying *Bag* with the corresponding alternatives of *Var*. The construct `+Var^Goal` tells bagof not to bind *Var* in *Goal*. `bagof/3` fails if *Goal* has no solutions.

The example below illustrates `bagof/3` and the `^` operator. The variable bindings are printed together on one line to save paper.

```
2 ?- listing(foo).

foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).

Yes
3 ?- bagof(C, foo(A, B, C), Cs).

A = a, B = b, C = G308, Cs = [c, d] ;
A = b, B = c, C = G308, Cs = [e, f] ;
A = c, B = c, C = G308, Cs = [g] ;

No
4 ?- bagof(C, A^foo(A, B, C), Cs).

A = G324, B = b, C = G326, Cs = [c, d] ;
A = G324, B = c, C = G326, Cs = [e, f, g] ;

No
5 ?-
```

[14]Please note that the semantics have changed between 3.1.1 and 3.1.2

setof(+*Var*, +*Goal*, -*Set***)**
Equivalent to `bagof/3`, but sorts the result using `sort/2` to get a sorted list of alternatives without duplicates.

3.33 Invoking Predicates on all Members of a List

All the predicates in this section call a predicate on all members of a list or until the predicate called fails. The predicate is called via call/[2..], which implies common arguments can be put in front of the arguments obtained from the list(s). For example:

```
?- maplist(plus(1), [0, 1, 2], X).

X = [1, 2, 3]
```

we will phrase this as "*Predicate* is applied on …"

checklist(+*Pred*, +*List***)**
Pred is applied successively on each element of *List* until the end of the list or *Pred* fails. In the latter case the `checklist/2` fails.

maplist(+*Pred*, ?*List1*, ?*List2***)**
Apply *Pred* on all successive pairs of elements from *List1* and *List2*. Fails if *Pred* can not be applied to a pair. See the example above.

sublist(+*Pred*, +*List1*, ?*List2***)**
Unify *List2* with a list of all elements of *List1* to which *Pred* applies.

3.34 Forall

forall(+*Cond*, +*Action***)**
For all alternative bindings of *Cond Action* can be proven. The example verifies that all arithmetic statements in the list *L* are correct. It does not say which is wrong if one proves wrong.

```
?- forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 * 2]),
                 Result =:= Formula).
```

3.35 Formatted Write

The current version of SWI-Prolog provides two formatted write predicates. The first is `writef/[1,2]`, which is compatible with Edinburgh C-Prolog. The second is `format/[1,2]`, which is compatible with Quintus Prolog. We hope the Prolog community will once define a standard formatted write predicate. If you want performance use `format/[1,2]` as this predicate is defined in C. Otherwise compatibility reasons might tell you which predicate to use.

3.35.1 Writef

write_ln(+*Term***)**
Equivalent to `write(Term), nl.`

writef(+*Atom***)**
Equivalent to `writef(Atom, []).`

writef(+*Format*, +*Arguments***)**
Formatted write. *Format* is an atom whose characters will be printed. *Format* may contain certain special character sequences which specify certain formatting and substitution actions. *Arguments* then provides all the terms required to be output.

Escape sequences to generate a single special character:

`\n`	Output a nemline character (see also `nl/[0,1]`)
`\l`	Output a line separator (same as `\n`)
`\r`	Output a carriage-return character (ASCII 13)
`\t`	Output the ASCII character TAB (9)
`\\`	The character `\` is output
`\%`	The character `%` is output
`\nnn`	where ⟨*nnn*⟩ is an integer (1-3 digits) the character with ASCII code ⟨*nnn*⟩ is output (NB : ⟨*nnn*⟩ is read as **decimal**)

Note that `\l`, `\nnn` and `\\` are interpreted differently when character-escapes are in effect. See section 2.15.1.

Escape sequences to include arguments from *Arguments*. Each time a % escape sequence is found in *Format* the next argument from *Arguments* is formatted according to the specification.

`%t`	`print/1` the next item (mnemonic: term)
`%w`	`write/1` the next item
`%q`	`writeq/1` the next item
`%d`	Write the term, ignoring operators. See also `write_term/2`. Mnemonic: old Edinburgh `display/1`.
`%p`	`print/1` the next item (identical to `%t`)
`%n`	Put the next item as a character (i.e. it is an ASCII value)
`%r`	Write the next item N times where N is the second item (an integer)
`%s`	Write the next item as a String (so it must be a list of characters)
`%f`	Perform a `ttyflush/0` (no items used)
`%Nc`	Write the next item Centered in N columns.
`%Nl`	Write the next item Left justified in N columns.
`%Nr`	Write the next item Right justified in N columns. N is a decimal number with at least one digit. The item must be an atom, integer, float or string.

swritef(-*String*, +*Format*, +*Arguments***)**
Equivalent to `writef/2`, but “writes” the result on *String* instead of the current output stream. Example:

```
?- swritef(S, '%15L%w', ['Hello', 'World']).

S = "Hello          World"
```

swritef(-*String*, +*Format***)**
Equivalent to `swritef(String, Format, [])`.

3.35.2 Format

format(+*Format***)**
Defined as ‘`format(Format) :- format(Format, []).`’

format(+*Format*, +*Arguments***)**
Format is an atom, list of ASCII values, or a Prolog string. *Arguments* provides the arguments required by the format specification. If only one argument is required and this is not a list of ASCII values the argument need not be put in a list. Otherwise the arguments are put in a list.

Special sequences start with the tilde (`~`), followed by an optional numeric argument, followed by a character describing the action to be undertaken. A numeric argument is either a sequence of digits, representing a positive decimal number, a sequence `'`⟨*character*⟩, representing the ASCII value of the character (only useful for `~t`) or a asterisk (`*`), in when the numeric argument is taken from the next argument of the argument list, which should be a positive integer. Actions are:

`~` Output the tilde itself.

`a` Output the next argument, which should be an atom. This option is equivalent to **w**. Compatibility reasons only.

`c` Output the next argument as an ASCII value. This argument should be an integer in the range [0, ..., 255] (including 0 and 255).

`d` Output next argument as a decimal number. It should be an integer. If a numeric argument is specified a dot is inserted *argument* positions from the right (useful for doing fixed point arithmetic with integers, such as handling amounts of money).

`D` Same as **d**, but makes large values easier to read by inserting a comma every three digits left to the dot or right.

`e` Output next argument as a floating point number in exponential notation. The numeric argument specifies the precision. Default is 6 digits. Exact representation depends on the C library function printf(). This function is invoked with the format `%.`⟨*precision*⟩`e`.

`E` Equivalent to **e**, but outputs a capital E to indicate the exponent.

`f` Floating point in non-exponential notation. See C library function printf().

`g` Floating point in **e** or **f** notation, whichever is shorter.

`G` Floating point in **E** or **f** notation, whichever is shorter.

`i` Ignore next argument of the argument list. Produces no output.

`k` Give the next argument to `displayq/1` (canonical write).

`n` Output a newline character.

`N` Only output a newline if the last character output on this stream was not a newline. Not properly implemented yet.

`p` Give the next argument to `print/1`.

`q` Give the next argument to `writeq/1`.

`r` Print integer in radix the numeric argument notation. Thus `~16r` prints its argument hexadecimal. The argument should be in the range $[2, \ldots, 36]$. Lower case letters are used for digits above 9.

`R` Same as **r**, but uses upper case letters for digits above 9.

`s` Output a string of ASCII characters or a string (see `string/1` and section 3.23) from the next argument.

`t` All remaining space between 2 tabs tops is distributed equally over `~t` statements between the tabs tops. This space is padded with spaces by default. If an argument is supplied this is taken to be the ASCII value of the character used for padding. This can be used to do left or right alignment, centering, distributing, etc. See also `~|` and `~+` to set tab stops. A tabs top is assumed at the start of each line.

`|` Set a tabs top on the current position. If an argument is supplied set a tabs top on the position of that argument. This will cause all `~t`'s to be distributed between the previous and this tabs top.

`+` Set a tabs top relative to the current position. Further the same as `~|`.

`w` Give the next argument to `write/1`.

`W` Give the next two argument to `write_term/2`. This option is SWI-Prolog specific.

Example:

```
simple_statistics :-
    <obtain statistics>         % left to the user
    format('~tStatistics~t~72|~n~n'),
    format('Runtime: ~'.t ~2f~34|  Inferences: ~'.t ~D~72|~n',
                                            [RunT, Inf]),
    ....
```

Will output

```
                             Statistics

Runtime: .................. 3.45  Inferences: .......... 60,345
```

format(+*Stream*, +*Format*, +*Arguments***)**
As `format/2`, but write the output on the given *Stream*.

sformat(-*String*, +*Format*, +*Arguments***)**
Equivalent to `format/2`, but "writes" the result on *String* instead of the current output stream. Example:

```
?- sformat(S, '~w~t~15|~w', ['Hello', 'World']).

S = "Hello          World"
```

sformat(-*String*, +*Format***)**
Equivalent to '`sformat(String, Format, []).`'

3.35.3 Programming Format

format_predicate(+*Char*, +*Head***)**
If a sequence `~c` (tilde, followed by some character) is found, the format derivatives will first check whether the user has defined a predicate to handle the format. If not, the built in formatting rules described above are used. *Char* is either an ASCII value, or a one character atom, specifying the letter to be (re)defined. *Head* is a term, whose name and arity are used to determine the predicate to call for the redefined formatting character. The first argument to the predicate is the numeric argument of the format command, or the atom `default` if no argument is specified. The remaining arguments are filled from the argument list. The example below redefines `~n` to produce *Arg* times return followed by linefeed (so a (Grr.) DOS machine is happy with the output).

```
:- format_predicate(n, dos_newline(_Arg)).

dos_newline(Arg) :-
        between(1, Ar, _), put(13), put(10), fail ; true.
```

current_format_predicate(*?Code, ?:Head***)**
Enumerates all user-defined format predicates. *Code* is the character code of the format character. *Head* is unified with a term with the same name and arity as the predicate. If the predicate does not reside in module `user`, *Head* is qualified with the definition module of the predicate.

3.36 Terminal Control

The following predicates form a simple access mechanism to the Unix termcap library to provide terminal independent I/O for screen terminals. These predicates are only available on Unix machines. The SWI-Prolog Windows consoles accepts the ANSI escape sequences.

tty_get_capability(*+Name, +Type, -Result***)**
Get the capability named *Name* from the termcap library. See termcap(5) for the capability names. *Type* specifies the type of the expected result, and is one of `string`, `number` or `bool`. String results are returned as an atom, number result as an integer and bool results as the atom `on` or `off`. If an option cannot be found this predicate fails silently. The results are only computed once. Successive queries on the same capability are fast.

tty_goto(*+X, +Y***)**
Goto position (*X*, *Y*) on the screen. Note that the predicates `line_count/2` and `line_position/2` will not have a well defined behaviour while using this predicate.

tty_put(*+Atom, +Lines***)**
Put an atom via the termcap library function tputs(). This function decodes padding information in the strings returned by `tty_get_capability/3` and should be used to output these strings. *Lines* is the number of lines affected by the operation, or 1 if not applicable (as in almost all cases).

set_tty(*-OldStream, +NewStream***)**
Set the output stream, used by `tty_put/2` and `tty_goto/2` to a specific stream. Default is user_output.

3.37 Operating System Interaction

shell(*+Command, -Status***)**
Execute *Command* on the operating system. *Command* is given to the Bourne shell (/bin/sh). *Status* is unified with the exit status of the command.

On *Win32* systems, `shell/[1,2]` executes the command using the CreateProcess() API and waits for the command to terminate. If the command ends with a & sign, the command is handed to the WinExec() API, which does not wait for the new task to terminate. See also `win_exec/2` and `win_shell/2`. Please note that the CreateProcess() API does **not** imply the Windows command interpreter (`command.exe` on Windows 95/98 and `cmd.exe` on Windows-NT) and therefore commands built-in to the command-interpreter can only be activated using the command interpreter. For example: `'command.exe /C copy file1.txt file2.txt'`

shell(*+Command***)**
Equivalent to '`shell(Command, 0)`'.

shell

Start an interactive Unix shell. Default is `/bin/sh`, the environment variable `SHELL` overrides this default. Not available for Win32 platforms.

win_exec(*+Command, +Show***)**

Win32 systems only. Spawns a Windows task without waiting for its completion. *Show* is either `iconic` or `normal` and dictates the initial status of the window. The `iconic` option is notably handy to start (DDE) servers.

win_shell(*+Operation, +File***)**

Win32 systems only. Opens the document *File* using the windows shell-rules for doing so. *Operation* is one of `open`, `print` or `explore` or another operation registered with the shell for the given document-type. On modern systems it is also possible to pass a URL as *File*, opening the URL in Windows default browser. This call interfaces to the Win32 API ShellExecute().

win_registry_get_value(*+Key, +Name, -Value***)**

Win32 systems only. Fetches the value of a Win32 registry key. *Key* is an atom formed as a path-name describing the desired registry key. *Name* is the desired attribute name of the key. *Value* is unified with the value. If the value is of type `DWORD`, the value is returned as an integer. If the value is a string it is returned as a Prolog atom. Other types are currently not supported. The default 'root' is `HKEY_CURRENT_USER`. Other roots can be specified explicitely as `HKEY_CLASSES_ROOT`, `HKEY_CURRENT_USER`, `HKEY_LOCAL_MACHINE` or `HKEY_USERS`. The example below fetches the extension to use for Prolog files (see `README.TXT` on the Windows version):

```
?- win_registry_get_value('HKEY_LOCAL_MACHINE/Software/SWI/Prolog',
                          fileExtension,
                          Ext).

Ext = pl
```

getenv(*+Name, -Value***)**

Get environment variable. Fails silently if the variable does not exist. Please note that environment variable names are case-sensitive on Unix systems and case-insensitive on Windows.

setenv(*+Name, +Value***)**

Set environment variable. *Name* and *Value* should be instantiated to atoms or integers. The environment variable will be passed to `shell/[0-2]` and can be requested using `getenv/2`. They also influence `expand_file_name/2`.

unsetenv(*+Name***)**

Remove environment variable from the environment.

unix(*+Command***)**

This predicate comes from the Quintus compatibility library and provides a partial implementation thereof. It provides access to some operating system features and unlike the name suggests, is not operating system specific. Currently it is the only way to fetch the Prolog command-line arguments. Defined *Command*'s are below.

system(+*Command***)**
Equivalent to calling `shell/1`. Use for compatibility only.

shell(+*Command***)**
Equivalent to calling `shell/1`. Use for compatibility only.

shell
Equivalent to calling `shell/0`. Use for compatibility only.

cd
Equivalent to calling `chdir/1` as `chdir(~)`. Use for compatibility only.

cd(+*Directory***)**
Equivalent to calling `chdir/1`. Use for compatibility only.

argv(-*Argv***)**
Unify *Argv* with the list of commandline arguments provides to this Prolog run. Please note that Prolog system-arguments and application arguments are separated by `--`. Integer arguments are passed as Prolog integers, float arguments and Prolog floating point numbers and all other arguments as Prolog atoms. New applications should use the prolog-flag `argv`.

A stand-alone program could use the following skeleton to handle command-line arguments. See also section 2.10.2.

```
main :-
        unix(argv(Argv)),
        append(_PrologArgs, [--|AppArgs], Argv), !,
        main(AppArgs).
```

get_time(-*Time***)**
Return the number of seconds that elapsed since the epoch of the POSIX, tim representation: January 1970, 0 hours. *Time* is a floating point number. The granularity is system dependent.

convert_time(+*Time*, -*Year*, -*Month*, -*Day*, -*Hour*, -*Minute*, -*Second*, -*MilliSeconds***)**
Convert a time stamp, provided by `get_time/1`, `time_file/2`, etc. *Year* is unified with the year, *Month* with the month number (January is 1), *Day* with the day of the month (starting with 1), *Hour* with the hour of the day (0–23), *Minute* with the minute (0–59). *Second* with the second (0–59) and *MilliSecond* with the milliseconds (0–999). Note that the latter might not be accurate or might always be 0, depending on the timing capabilities of the system. See also `convert_time/2`.

convert_time(+*Time*, -*String***)**
Convert a time-stamp as obtained though `get_time/1` into a textual representation using the C-library function `ctime()`. The value is returned as a SWI-Prolog string object (see section 3.23). See also `convert_time/8`.

3.38 File System Interaction

access_file(+*File*, +*Mode***)**
Succeeds if *File* exists and can be accessed by this prolog process under mode *Mode*. *Mode*

is one of the atoms `read`, `write`, `append`, `exist`, `none` or `execute`. *File* may also be the name of a directory. Fails silently otherwise. `access_file(File, none)` simply succeeds without testing anything.

If 'Mode' is `write` or `append`, this predicate also succeeds if the file does not exist and the user has write-access to the directory of the specified location.

exists_file(+*File***)**
Succeeds when *File* exists. This does not imply the user has read and/or write permission for the file.

file_directory_name(+*File*, -*Directory***)**
Extracts the directory-part of *File*. The resulting *Directory* name ends with the directory separator character `/`. If *File* is an atom that does not contain any directory separator characters, the empty atom `''` is returned. See also `file_base_name/2`.

file_base_name(+*File*, -*BaseName***)**
Extracts the filename part from a path specification. If *File* does not contain any directory separators, *File* is returned.

same_file(+*File1*, +*File2***)**
Succeeds if both filenames refer to the same physical file. That is, if *File1* and *File2* are the same string or both names exist and point to the same file (due to hard or symbolic links and/or relative vs. absolute paths).

exists_directory(+*Directory***)**
Succeeds if *Directory* exists. This does not imply the user has read, search and or write permission for the directory.

delete_file(+*File***)**
Remove *File* from the file system.

rename_file(+*File1*, +*File2***)**
Rename *File1* into *File2*. Currently files cannot be moved across devices.

size_file(+*File*, -*Size***)**
Unify *Size* with the size of *File* in characters.

time_file(+*File*, -*Time***)**
Unify the last modification time of *File* with *Time*. *Time* is a floating point number expressing the seconds elapsed since Jan 1, 1970. See also `convert_time/[2,8]` and `get_time/1`.

absolute_file_name(+*File*, -*Absolute***)**
Expand Unix file specification into an absolute path. User home directory expansion (˜ and ˜⟨*user*⟩) and variable expansion is done. The absolute path is canonised: references to `.` and `..` are deleted. SWI-Prolog uses absolute file names to register source files independent of the current working directory. See also `absolute_file_name/3`.

absolute_file_name(+*Spec*, +*Options*, -*Absolute***)**
Converts the given file specification into an absolute path. *Option* is a list of options to guide the conversion:

extensions(*ListOfExtensions***)**
List of file-extensions to try. Default is `''`. For each extension, `absolute_file_name/3` will first add the extension and then verify the conditions imposed by the other options. If the condition fails, the next extension of the list is tried. Extensions may be specified both as `..ext` or plain `ext`.

access(*Mode***)**
Imposes the condition access_file(*File*, *Mode*). *Mode* is on of `read`, `write`, `append`, `exist` or `none`. See also `access_file/2`.

file_type(*Type***)**
Defines extensions. Current mapping: `txt` implies `['']`, `prolog` implies `['.pl', '']`, `executable` implies `['.so', '']`, `qlf` implies `['.qlf', '']` and `directory` implies `['']`.

file_errors(*fail/error***)**
If `error` (default), throw and `existence_error` exception if the file cannot be found. If `fail`, stay silent.[15]

solutions(*first/all***)**
If `first` (default), the predicates leaves no choice-point. Otherwise a choice-point will be left and backtracking may yield more solutions.

is_absolute_file_name(*+File***)**
True if *File* specifies and absolute path-name. On Unix systems, this implies the path starts with a '/'. For Microsoft based systems this implies the path starts with ⟨*letter*⟩`:`. This predicate is intended to provide platform-independent checking for absolute paths. See also `absolute_file_name/2` and `prolog_to_os_filename/2`.

file_name_extension(*?Base, ?Extension, ?Name***)**
This predicate is used to add, remove or test filename extensions. The main reason for its introduction is to deal with different filename properties in a portable manner. If the file system is case-insensitive, testing for an extension will be done case-insensitive too. *Extension* may be specified with or without a leading dot (`.`). If an *Extension* is generated, it will not have a leading dot.

expand_file_name(*+WildCard, -List***)**
Unify *List* with a sorted list of files or directories matching *WildCard*. The normal Unix wildcard constructs '`?`', '`*`', '`[...]`' and '`{...}`' are recognised. The interpretation of '`{...}`' is interpreted slightly different from the C shell (csh(1)). The comma separated argument can be arbitrary patterns, including '`{...}`' patterns. The empty pattern is legal as well: '`\{.pl,\}`' matches either '`.pl`' or the empty string.

If the pattern does contains wildcard characters, only existing files and directories are returned. Expanding a 'pattern' without wildcard characters returns the argument, regardless on whether or not it exists.

Before expanding wildchards, the construct `$var` is expanded to the value of the environment variable *var* and a possible leading ~ character is expanded to the user's home directory.[16].

[15]Silent operation was the default upto version 3.2.6.

[16]On Windows, the home directory is determined as follows: if the environment variable `HOME` exists, this is used. If

prolog_to_os_filename(*?PrologPath, ?OsPath***)**
Converts between the internal Prolog pathname conventions and the operating-system pathname conventions. The internal conventions are Unix and this predicates is equivalent to =/2 (unify) on Unix systems. On DOS systems it will change the directory-separator, limit the filename length map dots, except for the last one, onto underscores.

read_link(*+File, -Link, -Target***)**
If *File* points to a symbolic link, unify *Link* with the value of the link and *Target* to the file the link is pointing to. *Target* points to a file, directory or non-existing entry in the file system, but never to a link. Fails if *File* is not a link. Fails always on systems that do not support symbolic links.

tmp_file(*+Base, -TmpName***)**
Create a name for a temporary file. *Base* is an identifier for the category of file. The *TmpName* is guaranteed to be unique. If the system halts, it will automatically remove all created temporary files.

make_directory(*+Directory***)**
Create a new directory (folder) on the filesystem. Raises an exception on failure. On Unix systems, the directory is created with default permissions (defined by the process *umask* setting).

delete_directory(*+Directory***)**
Delete directory (folder) from the filesystem. Raises an exception on failure. Please note that in general it will not be possible to delete a non-empty directory.

chdir(*+Path***)**
Change working directory to *Path*.[17]

3.39 Multi-threading (alpha code)

The features described in this section are only enabled on Unix systems providing POSIX threads and if the system is configured using the `--enable-mt` option. SWI-Prolog multi-theading support is experimental and in some areas not safe.

SWI-Prolog multithreading is based on standard C-language multithreading support. It is not like *ParLog* or other paralel implementations of the Prolog language. Prolog threads have their own stacks and only share the Prolog *heap*: predicates, records, flags and other global non-backtrackable data. SWI-Prolog thread support is designed with the following goals in mind.

- *Multi-threaded server applications*
  Todays computing services often focus on (internet) server applications. Such applications often have need for communication between services and/or fast non-blocking service to multiple concurrent clients. The shared heap provides fast communication and thread creation is relatively cheap (A Pentium-II/450 can create and join approx. 10,000 threads per second on Linux 2.2).

the variables `HOMEDRIVE` and `HOMEPATH` exist (Windows-NT), these are used. At initialisation, the system will set the environment variable `HOME` to point to the SWI-Prolog home directory if neither `HOME` nor `HOMEPATH` and `HOMEDRIVE` are defined

[17]BUG: Some of the file-I/O predicates use local filenames. Using `chdir/1` while file-bound streams are open causes wrong results on `telling/1`, `seeing/1` and `current_stream/3`

- *Interactive applications*
  Interactive applications often need to perform extensive computation. If such computations are executed in a new thread, the main thread can process events and allow the user to cancel the ongoing computation. User interfaces can also use multiple threads, each thread dealing with input from a distinct group of windows.

- *Natural integration with foreign code*
  Each Prolog thread runs in a C-thread, automatically making them cooperate with *MT-safe* foreign-code. In addition, any foreign thread can create its own Prolog engine for dealing with calling Prolog from C-code.

thread_create(*:Goal, -Id, +Options***)**
Create a new Prolog thread (and underlying C-thread) and start it by executing *Goal*. If the thread is created succesfully, the thread-identifier of the created thread is unified to *Id*. *Options* is a list of options. Currently defined options are:

local(*K-Bytes***)**
Set the limit to which the local stack of this thread may grow. If omited, the limit of the calling thread is used. See also the `-L` commandline option.

global(*K-Bytes***)**
Set the limit to which the global stack of this thread may grow. If omited, the limit of the calling thread is used. See also the `-G` commandline option.

trail(*K-Bytes***)**
Set the limit to which the trail stack of this thread may grow. If omited, the limit of the calling thread is used. See also the `-T` commandline option.

argument(*K-Bytes***)**
Set the limit to which the argument stack of this thread may grow. If omited, the limit of the calling thread is used. See also the `-A` commandline option.

alias(*AliasName***)**
Associate an ‘alias-name’ with the thread. This named may be used to refer to the thread and remains valid until the thread is joined (see `thread_join/2`).

detached(*Bool***)**
If `false` (default), the thread can be waited for using `thread_join/2`. `thread_join/2` must be called on this thread to reclaim the all resources associated to the thread. If `true`, the system will reclaim all associated resources automatically after the thread finishes. Please not that thread identifiers are freed for reuse after a detached thread finishes or a normal thread has been joined.

The *Goal* argument is *copied* to the new Prolog engine. This implies further instantiation of this term in either thread does not have consequences for the other thread: Prolog threads do not share data from their stacks.

thread_self(*-Id***)**
Get the Prolog thread identifier of the running thread. If the thread has an alias, the alias-name is returned.

current_thread(*?Id, ?Status***)**
Enumerates identifiers and status of all currently known threads. Calling `current_thread/2` does not influence any thread. See also `thread_join/2`. For threads that have an alias-name, this name is returned in *Id* instead of the numerical thread identifier. *Status* is one of:

running
The thread is running. This is the initial status of a thread. Please note that threats waiting for something are considered running too.

false
The *Goal* of the thread has been completed and failed.

true
The *Goal* of the thread has been completed and succeeded.

exited(*Term***)**
The *Goal* of the thread has been terminated using `thread_exit/1` with *Term* as argument.

exception(*Term***)**
The *Goal* of the thread has been terminated due to an uncaught exception (see `throw/1` and `catch/3`).

thread_join(*+Id, -Status***)**
Wait for the termination of thread with given *Id*. Then unify the result-status (see `thread_exit/1`) of the thread with *Status*. After this call, *Id* becomes invalid and all resources associated with the thread are reclaimed. See also `current_thread/2`.

A thread that has been completed without `thread_join/2` being called on it is partly reclaimed: the Prolog stacks are released and the C-thread is destroyed. A small data-structure represening the exit-status of the thread is retained until `thread_join/2` is called on the thread.

thread_exit(*+Term***)**
Terminates the thread immediately, leaving `exited`(*Term*) as result-state. The Prolog stacks and C-thread are reclaimed.

thread_at_exit(*:Goal***)**
Run *Goal* after the execution of this thread has terminated. This is to be compared to `at_halt/1`, but only for the current thread. These hooks are ran regardless of why the execution of the thread has been completed. As these hooks are run, the return-code is already available through `current_thread/2`.

3.39.1 Thread communication

Prolog threads can exchange data using dynamic predicates, database records, and other globally shared data. In addition, they can send messages to each other. If a threads needs to wait for another thread until that thread has produced some data, using only the database forces the waiting thread to poll the database continuously. Waiting for a message suspends the thread execution until the message has arrived in its message queue.

thread_send_message(+*ThreadId, +Term***)**

Place *Term* in the message queue of the indicated thread (which can even be the message queue of itself (see `thread_self/1`). Any term can be placed in a message queue, but note that the term is copied to to receiving thread and variable-bindings are thus lost. This call returns immediately.

thread_get_message(*?Term***)**

Examines the thread message-queue and if necessary blocks execution until a term that unifies to *Term* arrives in the queue. After a term from the queue has been unified unified to *Term*, this term is deleted from the queue and this predicate returns.

Please note that not-unifying messages remain in the queue. After the following has been executed, thread 1 has the term `b`(*gnu*) in its queue and continues execution using *A* is `gnat`.

```
<thread 1>
thread_get_message(a(A)),

<thread 2>
thread_send_message(b(gnu)),
thread_send_message(a(gnat)),
```

See also `thread_peek_message/1`.

thread_peek_message(*?Term***)**

Examines the thread message-queue and compares the queued terms with *Term* until one unifies or the end of the queue has been reached. In the first case the call succeeds (possibly instantiating *Term*. If no term from the queue unifies this call fails.

thread_signal(+*ThreadId, :Goal***)**

Make thread *ThreadId* execute *Goal* at the first opportunity. In the current implementation, this implies at the first pass through the *Call-port*. The predicate `thread_signal/2` itself places *Goal* into the signalled-thread's signal queue and returns immediately.

Signals (interrupts) do not cooperate well with the world of multi-threading, mainly because the status of mutexes cannot be guaranteed easily. At the call-port, the Prolog virtual machine holds no locks and therefore the asynchronous execution is safe.

Goal can be any valid Prolog goal, including `throw/1` to make the receiving thread generate an exception and `trace/0` to start tracing the receiving thread.

3.39.2 Thread synchronisation

All internal Prolog operations are thread-safe. This implies two Prolog threads can operate on the same dynamic predicate without corrupting the consistency of the predicate. This section deals with user-level *mutexes* (called *monitors* in ADA or *critical-sections* by Microsoft). A mutex is a **MUT**ual **EX**clusive device, which implies at most one thread can *hold* a mutex.

Mutexes are used to realise related updates to the Prolog database. With 'related', we refer to the situation where a 'transaction' implies two or more changes to the Prolog database. For example, we have a predicate `address/2`, representing the address of a person and we want to change the address by retracting the old and asserting the new address. Between these two operations the database is invalid: this person has either no address or two addresses (depending on the assert/retract order).

Here is how to realise a correct update:

```
:- initialization
        mutex_create(addressbook).

change_address(Id, Address) :-
        mutex_lock(addressbook),
        retractall(address(Id, _)),
        asserta(address(Id, Address)),
        mutex_unlock(addressbook).
```

mutex_create(*?MutexId*)
: Create a mutex. if *MutexId* is an atom, a *named* mutex is created. If it is a variable, an anonymous mutex reference is returned. There is no limit to the number of mutexes that can be created.

mutex_destroy(*+MutexId*)
: Destroy a mutex. After this call, *MutexId* becomes invalid and further references yield an `existence_error` exception.

mutex_lock(*+MutexId*)
: Lock the mutex. Prolog mutexes are *recursive* mutexes: they can be locked multiple times by the same thread. Only after unlocking it as many times as it is locked, the mutex becomes available for locking by other threads. If another thread has locked the mutex the calling thread is suspended until to mutex is unlocked.

  If *MutexId* is an atom, and there is no current mutex with that name, the mutex is created automatically using `mutex_create/1`. This implies named mutexes need not be declared explicitly.

  Please note that locking and unlocking mutexes should be paired carefully. Especially make sure to unlock mutexes even if the protected code fails or raises an exception. For most common cases use `with_mutex/2`, wich provides a safer way for handling prolog-level mutexes.

mutex_trylock(*+MutexId*)
: As `mutex_lock/1`, but if the mutex is held by another thread, this predicates fails immediately.

mutex_unlock(*+MutexId*)
: Unlock the mutex. This can only be called if the mutex is held by the calling thread. If this is not the case, a `permission_error` exception is raised.

mutex_unlock_all
: Unlock all mutexes held by the current thread. This call is especially useful to handle thread-termination using `abort/0` or exceptions. See also `thread_signal/2`.

current_mutex(*?MutexId, ?ThreadId, ?Count*)
: Enumerates all existing mutexes. If the mutex is held by some thread, *ThreadId* is unified with

the identifier of te holding thread and *Count* with the recursive count of the mutex. Otherwise, *ThreadId* is `[]` and *Count* is 0.

with_mutex(+*MutexId*, :*Goal***)**

Execute *Goal* while holding *MutexId*. If *Goal* leaves choicepointes, these are destroyed (as in `once/1`). The mutex is unlocked regardless of whether *Goal* succeeds, fails or raises an exception. An exception thrown by *Goal* is re-thrown after the mutex has been successfully unlocked. See also `mutex_create/2`.

Although described in the thread-section, this predicate is also available in the single-threaded version, where it behaves simply as `once/1`.

3.39.3 Thread-support library(threadutil)

This library defines a couple of useful predicates for demonstrating and debugging multi-threaded applications. This library is certainly not complete.

threads

Lists all current threads and their status. In addition, all 'zombie' threads (finished threads that are not detached, nor waited for) are joined to reclaim their resources.

interactor

Create a new console and run the Prolog toplevel in this new console. See also `attach_console/0`.

attach_console

If the current thread has no console attached yet, attach one and redirect the user streams (input, output, and error) to the new console window. The console is an `xterm` application. For this to work, you should be running X-windows and your xterm should know the `-Sccn`.

This predicate has a couple of useful applications. One is to separate (debugging) I/O of different threads. Another is to start debugging a thread that is running in the background. If thread 10 is running, the following sequence starts the tracer on this thread:

```
?- thread_signal(10, (attach_console, trace)).
```

3.39.4 Status of the thread implementation

It is assumed that the basic Prolog execution is thread-safe. Various problems are to be expected though, both dead-locks as well as not-thread-safe code in builtin-predicates.

3.40 User Toplevel Manipulation

break

Recursively start a new Prolog top level. This Prolog top level has its own stacks, but shares the heap with all break environments and the top level. Debugging is switched off on entering a break and restored on leaving one. The break environment is terminated by typing the system's end-of-file character (control-D). If the `-t` *`toplevel`* command line option is given this goal is started instead of entering the default interactive top level (`prolog/0`).

abort

Abort the Prolog execution and restart the top level. If the `-t` *`toplevel`* command line options is given this goal is started instead of entering the default interactive top level.

There are two implementations of `abort/0`. The default one uses the exception mechanism (see `throw/1`), throwing the exception `$aborted`. The other one uses the C-construct longjmp() to discard the entire environment and rebuild a new one. Using exceptions allows for proper recovery of predicates exploiting exceptions. Rebuilding the environment is safer if the Prolog stacks are corrupt. Therefore the system will use the rebuild-strategy if the abort was generated by an internal consistency check and the exception mechanism otherwise. Prolog can be forced to use the rebuild-strategy setting the prolog flag `abort_with_exception` to `false`.

halt

Terminate Prolog execution. Open files are closed and if the command line option `-tty` is not active the terminal status (see Unix stty(1)) is restored. Hooks may be registered both in Prolog and in foreign code. Prolog hooks are registered using `at_halt/1`. `halt/0` is equivalent to `halt(0)`.

halt(+*Status***)**

Terminate Prolog execution with given status. Status is an integer. See also `halt/0`.

prolog

This goal starts the default interactive top level. Queries are read from the stream `user_input`. See also the `history` prolog_flag (`current_prolog_flag/2`). The `prolog/0` predicate is terminated (succeeds) by typing the end-of-file character (On most systems control-D).

The following two hooks allow for expanding queries and handling the result of a query. These hooks are used by the toplevel variable expansion mechanism described in section 2.8.

expand_query(+*Query*, -*Expanded*, +*Bindings*, -*ExpandedBindings***)**

Hook in module `user`, normally not defined. *Query* and *Bindings* represents the query read from the user and the names of the free variables as obtained using `read_term/3`. If this predicate succeeds, it should bind *Expanded* and *ExpandedBindings* to the query and bindings to be executed by the toplevel. This predicate is used by the toplevel (`prolog/0`). See also `expand_answer/2` and `term_expansion/2`.

expand_answer(+*Bindings*, -*ExpandedBindings***)**

Hook in module `user`, normally not defined. Expand the result of a successfully executed toplevel query. *Bindings* is the query $\langle Name \rangle = \langle Value \rangle$ binding list from the query. *ExpandedBindings* must be unified with the bindings the toplevel should print.

3.41 Creating a Protocol of the User Interaction

SWI-Prolog offers the possibility to log the interaction with the user on a file.[18] All Prolog interaction, including warnings and tracer output, are written on the protocol file.

[18] A similar facility was added to Edinburgh C-Prolog by Wouter Jansweijer.

protocol(+*File***)**
Start protocolling on file *File*. If there is already a protocol file open then close it first. If *File* exists it is truncated.

protocola(+*File***)**
Equivalent to `protocol/1`, but does not truncate the *File* if it exists.

noprotocol
Stop making a protocol of the user interaction. Pending output is flushed on the file.

protocolling(-*File***)**
Succeeds if a protocol was started with `protocol/1` or `protocola/1` and unifies *File* with the current protocol output file.

3.42 Debugging and Tracing Programs

This section is a reference to the debugger interaction predicates. A more use-oriented overview of the debugger is in section 2.9.

If you have installed XPCE, you can use the graphical frontend of the tracer. This frontend is installed using the predicate `guitracer/0`.

trace
Start the tracer. `trace/0` itself cannot be seen in the tracer. Note that the Prolog toplevel treats `trace/0` special; it means 'trace the next goal'.

tracing
Succeeds when the tracer is currently switched on. `tracing/0` itself can not be seen in the tracer.

notrace
Stop the tracer. `notrace/0` itself cannot be seen in the tracer.

guitracer
Installs hooks (see `prolog_trace_interception/4`) into the system that redirects tracing information to a GUI frontend providing structured access to variable-bindings, graphical overview of the stack and highlighting of relevant source-code.

noguitracer
Reverts back to the textual tracer.

trace(+*Pred***)**
Equivalent to `trace(`*`Pred`*`, +all)`.

trace(+*Pred*, +*Ports***)**
Put a trace-point on all predicates satisfying the predicate specification *Pred*. *Ports* is a list of portnames (`call`, `redo`, `exit`, `fail`). The atom `all` refers to all ports. If the port is preceded by a `-` sign the trace-point is cleared for the port. If it is preceded by a `+` the trace-point is set.

The predicate `trace/2` activates debug mode (see `debug/0`). Each time a port (of the 4-port model) is passed that has a trace-point set the goal is printed as with `trace/0`. Unlike `trace/0` however, the execution is continued without asking for further information. Examples:

`?- trace(hello).`	Trace all ports of hello with any arity in any module.
`?- trace(foo/2, +fail).`	Trace failures of foo/2 in any module.
`?- trace(bar/1, -all).`	Stop tracing bar/1.

The predicate `debugging/0` shows all currently defined trace-points.

notrace(+*Goal***)**
Call *Goal*, but suspend the debugger while *Goal* is executing. The current implementation cuts the choicepoints of *Goal* after successful completion. See `once/1`. Later implementations may have the same semantics as `call/1`.

debug
Start debugger. In debug mode, Prolog stops at spy- and trace-points, disables tail-recursion optimisation and aggressive destruction of choice-points to make debugging information accessible. Implemented by the Prolog flag `debug`.

nodebug
Stop debugger. Implementated by the prolog flag `debug`. See also `debug/0`.

debugging
Print debug status and spy points on current output stream. See also the prolog flag `debug`.

spy(+*Pred***)**
Put a spy point on all predicates meeting the predicate specification *Pred*. See section 3.4.

nospy(+*Pred***)**
Remove spy point from all predicates meeting the predicate specification *Pred*.

nospyall
Remove all spy points from the entire program.

leash(?*Ports***)**
Set/query leashing (ports which allow for user interaction). *Ports* is one of +*Name*, -*Name*, ?*Name* or a list of these. +*Name* enables leashing on that port, -*Name* disables it and ?*Name* succeeds or fails according to the current setting. Recognised ports are: `call`, `redo`, `exit`, `fail` and `unify`. The special shorthand `all` refers to all ports, `full` refers to all ports except for the unify port (default). `half` refers to the `call`, `redo` and `fail` port.

visible(+*Ports***)**
Set the ports shown by the debugger. See `leash/1` for a description of the port specification. Default is `full`.

unknown(-*Old*, +*New***)**
Edinburgh-prolog compatibility predicate, interfacing to the ISO prolog flag `unknown`. Values are `trace` (meaning `error`) and `fail`. If the `unknown` flag is set to `warning`, `unknown/2` reports the value as `trace`.

style_check(+*Spec***)**

Set style checking options. *Spec* is either `+`⟨*option*⟩, `-`⟨*option*⟩, `?`⟨*option*⟩ or a list of such options. `+`⟨*option*⟩ sets a style checking option, `-`⟨*option*⟩ clears it and `?`⟨*option*⟩ succeeds or fails according to the current setting. `consult/1` and derivatives resets the style checking options to their value before loading the file. If—for example—a file containing long atoms should be loaded the user can start the file with:

```
:- style_check(-atom).
```

Currently available options are:

Name	Default	Description
`singleton`	on	`read_clause/1` (used by `consult/1`) warns on variables only appearing once in a term (clause) which have a name not starting with an underscore.
`atom`	on	`read/1` and derivatives will produce an error message on quoted atoms or strings longer than 5 lines.
`dollar`	off	Accept dollar as a lower case character, thus avoiding the need for quoting atoms with dollar signs. System maintenance use only.
`discontiguous`	on	Warn if the clauses for a predicate are not together in the same source file.
`string`	off	Backward compatibility. See the prolog-flag `double_quotes` (`current_prolog_flag/2`).

3.43 Obtaining Runtime Statistics

statistics(+*Key*, -*Value***)**

Unify system statistics determined by *Key* with *Value*. The possible keys are given in the table 3.2.

statistics

Display a table of system statistics on the current output stream.

time(+*Goal***)**

Execute *Goal* just like `once/1` (i.e. leaving no choice points), but print used time, number of logical inferences and the average number of *lips* (logical inferences per second). Note that SWI-Prolog counts the actual executed number of inferences rather than the number of passes through the call- and redo ports of the theoretical 4-port model.

3.44 Finding Performance Bottlenecks

SWI-Prolog offers a statistical program profiler similar to Unix prof(1) for C and some other languages. A profiler is used as an aid to find performance pigs in programs. It provides information on the time spent in the various Prolog predicates.

agc	Number of atom garbage-collections performed
agc_gained	Number of atoms removed
agc_time	Time spent in atom garbage-collections
cputime	(User) CPU time since Prolog was started in seconds
inferences	Total number of passes via the call and redo ports since Prolog was started.
heap	Estimated total size of the heap (see section 2.16.1)
heapused	Bytes heap in use by Prolog.
heaplimit	Maximum size of the heap (see section 2.16.1)
local	Allocated size of the local stack in bytes.
localused	Number of bytes in use on the local stack.
locallimit	Size to which the local stack is allowed to grow
global	Allocated size of the global stack in bytes.
globalused	Number of bytes in use on the global stack.
globallimit	Size to which the global stack is allowed to grow
trail	Allocated size of the trail stack in bytes.
trailused	Number of bytes in use on the trail stack.
traillimit	Size to which the trail stack is allowed to grow
atoms	Total number of defined atoms.
functors	Total number of defined name/arity pairs.
predicates	Total number of predicate definitions.
modules	Total number of module definitions.
codes	Total amount of byte codes in all clauses.
threads	MT-version: number of active threads
threads_created	MT-version: number of created threads
threads_cputime	MT-version: seconds CPU time used by finished threads

Table 3.2: Keys for `statistics/2`

The profiler is based on the assumption that if we monitor the functions on the execution stack on time intervals not correlated to the program's execution the number of times we find a procedure on the environment stack is a measure of the time spent in this procedure. It is implemented by calling a procedure each time slice Prolog is active. This procedure scans the local stack and either just counts the procedure on top of this stack (`plain` profiling) or all procedures on the stack (`cumulative` profiling). To get accurate results each procedure one is interested in should have a reasonable number of counts. Typically a minute runtime will suffice to get a rough overview of the most expensive procedures.

profile(+*Goal*, +*Style*, +*Number***)**

Execute *Goal* just like `time/1`. Collect profiling statistics according to style (see `profiler/2`) and show the top *Number* procedures on the current output stream (see `show_profile/1`). The results are kept in the database until `reset_profiler/0` or `profile/3` is called and can be displayed again with `show_profile/1`. `profile/3` is the normal way to invoke the profiler. The predicates below are low-level predicates that can be used for special cases.

show_profile(+*Number***)**

Show the collected results of the profiler. Stops the profiler first to avoid interference from `show_profile/1`. It shows the top *Number* predicates according the percentage CPU-time used.[19]

profiler(-*Old*, +*New***)**

Query or change the status of the profiler. The status is one of `off`, `plain` or `cumulative`. `plain` implies the time used by children of a predicate is not added to the time of the predicate. For status `cumulative` the time of children is added (except for recursive calls). Cumulative profiling implies the stack is scanned up to the top on each time slice to find all active predicates. This implies the overhead grows with the number of active frames on the stack. Cumulative profiling starts debugging mode to disable tail recursion optimisation, which would otherwise remove the necessary parent environments. Switching status from `plain` to `cumulative` resets the profiler. Switching to and from status `off` does not reset the collected statistics, thus allowing to suspend profiling for certain parts of the program.

reset_profiler

Switches the profiler to `off` and clears all collected statistics.

profile_count(+*Head*, -*Calls*, -*Promilage***)**

Obtain profile statistics of the predicate specified by *Head*. *Head* is an atom for predicates with arity 0 or a term with the same name and arity as the predicate required (see `current_predicate/2`). *Calls* is unified with the number of 'calls' and 'redos' while the profiler was active. *Promilage* is unified with the relative number of counts the predicate was active (`cumulative`) or on top of the stack (`plain`). *Promilage* is an integer between 0 and 1000.

[19] `show_profile/1` is defined in Prolog and takes a considerable amount of memory.

3.45 Memory Management

Note: `limit_stack/2` and `trim_stacks/0` have no effect on machines that do not offer dynamic stack expansion. On these machines these predicates simply succeed to improve portability.

garbage_collect

Invoke the global- and trail stack garbage collector. Normally the garbage collector is invoked automatically if necessary. Explicit invocation might be useful to reduce the need for garbage collections in time critical segments of the code. After the garbage collection `trim_stacks/0` is invoked to release the collected memory resources.

garbage_collect_atoms

Reclaim unused atoms. Normally invoked after `agc_margin` (a prolog flag) atoms have been created.

limit_stack(*+Key, +Kbytes***)**

Limit one of the stack areas to the specified value. *Key* is one of `local`, `global` or `trail`. The limit is an integer, expressing the desired stack limit in K bytes. If the desired limit is smaller than the currently used value, the limit is set to the nearest legal value above the currently used value. If the desired value is larger than the maximum, the maximum is taken. Finally, if the desired value is either 0 or the atom `unlimited` the limit is set to its maximum. The maximum and initial limit is determined by the command line options `-L`, `-G` and `-T`.

trim_stacks

Release stack memory resources that are not in use at this moment, returning them to the operating system. Trim stack is a relatively cheap call. It can be used to release memory resources in a backtracking loop, where the iterations require typically seconds of execution time and very different, potentially large, amounts of stack space. Such a loop should be written as follows:

```
loop :-
        generator,
            trim_stacks,
            potentially_expensive_operation,
        stop_condition, !.
```

The prolog top level loop is written this way, reclaiming memory resources after every user query.

stack_parameter(*+Stack, +Key, -Old, +New***)**

Query/set a parameter for the runtime stacks. *Stack* is one of `local`, `global`, `trail` or `argument`. The table below describes the *Key*/*Value* pairs. Old is first unified with the current value.

`limit`	Maximum size of the stack in bytes
`min_free`	Minimum free space at entry of foreign predicate

This predicate is currently only available on versions that use the stack-shifter to enlarge the runtime stacks when necessary. It's definition is subject to change.

3.46 Windows DDE interface

The predicates in this section deal with MS-Windows 'Dynamic Data Exchange' or DDE protocol.[20] A Windows DDE conversation is a form of interprocess communication based on sending reserved window-events between the communicating processes.

See also section 5.4 for loading Windows DLL's into SWI-Prolog.

3.46.1 DDE client interface

The DDE client interface allows Prolog to talk to DDE server programs. We will demonstrate the use of the DDE interface using the Windows PROGMAN (Program Manager) application:

```
1 ?- open_dde_conversation(progman, progman, C).

C = 0
2 ?- dde_request(0, groups, X)

--> Unifies X with description of groups

3 ?- dde_execute(0, '[CreateGroup("DDE Demo")]').

Yes

4 ?- close_dde_conversation(0).

Yes
```

For details on interacting with `progman`, use the SDK online manual section on the Shell DDE interface. See also the Prolog `library(progman)`, which may be used to write simple Windows setup scripts in Prolog.

open_dde_conversation(+*Service*, +*Topic*, -*Handle***)**
Open a conversation with a server supporting the given service name and topic (atoms). If successful, *Handle* may be used to send transactions to the server. If no willing server is found this predicate fails silently.

close_dde_conversation(+*Handle***)**
Close the conversation associated with *Handle*. All opened conversations should be closed when they're no longer needed, although the system will close any that remain open on process termination.

dde_request(+*Handle*, +*Item*, -*Value***)**
Request a value from the server. *Item* is an atom that identifies the requested data, and *Value* will be a string (`CF_TEXT` data in DDE parlance) representing that data, if the request is successful. If unsuccessful, *Value* will be unified with a term of form `error(`⟨*Reason*⟩`)`, identifying the problem. This call uses SWI-Prolog string objects to return the value rather then atoms to reduce the load on the atom-space. See section 3.23 for a discussion on this data type.

[20] This interface is contributed by Don Dwiggins.

dde_execute(+*Handle*, +*Command***)**
Request the DDE server to execute the given command-string. Succeeds if the command could be executed and fails with error message otherwise.

dde_poke(+*Handle*, +*Item*, +*Command***)**
Issue a `POKE` command to the server on the specified *Item*. Command is passed as data of type `CF_TEXT`.

3.46.2 DDE server mode

The (autoload) `library(dde)` defines primitives to realise simple DDE server applications in SWI-Prolog. These features are provided as of version 2.0.6 and should be regarded prototypes. The C-part of the DDE server can handle some more primitives, so if you need features not provided by this interface, please study `library(dde)`.

dde_register_service(+*Template*, +*Goal***)**
Register a server to handle DDE request or DDE execute requests from other applications. To register a service for a DDE request, *Template* is of the form:

+Service(+Topic, +Item, +Value)

Service is the name of the DDE service provided (like `progman` in the client example above). *Topic* is either an atom, indicating *Goal* only handles requests on this topic or a variable that also appears in *Goal*. *Item* and *Value* are variables that also appear in *Goal*. *Item* represents the request data as a Prolog atom.[21]

The example below registers the Prolog `current_prolog_flag/2` predicate to be accessible from other applications. The request may be given from the same Prolog as well as from another application.

```
?- dde_register_service(prolog(current_prolog_flag, F, V),
                        current_prolog_flag(F, V)).

?- open_dde_conversation(prolog, current_prolog_flag, Handle),
   dde_request(Handle, home, Home),
   close_dde_conversation(Handle).

Home = '/usr/local/lib/pl-2.0.6/'
```

Handling DDE `execute` requests is very similar. In this case the template is of the form:

+Service(+Topic, +Item)

Passing a *Value* argument is not needed as execute requests either succeed or fail. If *Goal* fails, a 'not processed' is passed back to the caller of the DDE request.

[21]Upto version 3.4.5 this was a list of character codes. As recent versions have atom garbage collection there is no need for this anymore.

dde_unregister_service(+*Service***)**
Stop responding to *Service*. If Prolog is halted, it will automatically call this on all open services.

dde_current_service(-*Service*, -*Topic***)**
Find currently registered services and the topics served on them.

dde_current_connection(-*Service*, -*Topic***)**
Find currently open conversations.

3.47 Miscellaneous

dwim_match(+*Atom1*, +*Atom2***)**
Succeeds if *Atom1* matches *Atom2* in 'Do What I Mean' sense. Both *Atom1* and *Atom2* may also be integers or floats. The two atoms match if:

- They are identical
- They differ by one character (spy ≡ spu)
- One character is inserted/deleted (debug ≡ deug)
- Two characters are transposed (trace ≡ tarce)
- 'Sub-words' are glued differently (existsfile ≡ existsFile ≡ exists_file)
- Two adjacent sub words are transposed (existsFile ≡ fileExists)

dwim_match(+*Atom1*, +*Atom2*, -*Difference***)**
Equivalent to `dwim_match/2`, but unifies *Difference* with an atom identifying the the difference between *Atom1* and *Atom2*. The return values are (in the same order as above): `equal`, `mismatched_char`, `inserted_char`, `transposed_char`, `separated` and `transposed_word`.

wildcard_match(+*Pattern*, +*String***)**
Succeeds if *String* matches the wildcard pattern *Pattern*. *Pattern* is very similar the the Unix csh pattern matcher. The patterns are given below:

`?`	Matches one arbitrary character.
`*`	Matches any number of arbitrary characters.
`[...]`	Matches one of the characters specified between the brackets. ⟨*char1*⟩`-`⟨*char2*⟩ indicates a range.
`{...}`	Matches any of the patterns of the comma separated list between the braces.

Example:

```
?- wildcard_match('[a-z]*.{pro,pl}[%~]', 'a_hello.pl%').

Yes
```

gensym(+*Base*, -*Unique***)**

Generate a unique atom from base *Base* and unify it with *Unique*. *Base* should be an atom. The first call will return ⟨*base*⟩1, the next ⟨*base*⟩2, etc. Note that this is no warrant that the atom is unique in the system.[22]

sleep(+*Time***)**

Suspend execution *Time* seconds. *Time* is either a floating point number or an integer. Granularity is dependent on the system's timer granularity. A negative time causes the timer to return immediately. On most non-realtime operating systems we can only ensure execution is suspended for **at least** *Time* seconds.

[22]BUG: I plan to supply a real `gensym/2` which does give this warrant for future versions.

Using Modules 4

4.1 Why Using Modules?

In traditional Prolog systems the predicate space was flat. This approach is not very suitable for the development of large applications, certainly not if these applications are developed by more than one programmer. In many cases, the definition of a Prolog predicate requires sub-predicates that are intended only to complete the definition of the main predicate. With a flat and global predicate space these support predicates will be visible from the entire program.

For this reason, it is desirable that each source module has it's own predicate space. A module consists of a declaration for it's name, it's *public predicates* and the predicates themselves. This approach allow the programmer to use short (local) names for support predicates without worrying about name conflicts with the support predicates of other modules. The module declaration also makes explicit which predicates are meant for public usage and which for private purposes. Finally, using the module information, cross reference programs can indicate possible problems much better.

4.2 Name-based versus Predicate-based Modules

Two approaches to realize a module system are commonly used in Prolog and other languages. The first one is the *name based* module system. In these systems, each atom read is tagged (normally prefixed) with the module name, with the exception of those atoms that are defined *public*. In the second approach, each module actually implements its own predicate space.

A critical problem with using modules in Prolog is introduced by the meta-predicates that transform between Prolog data and Prolog predicates. Consider the case where we write:

```
:- module(extend, [add_extension/3]).

add_extension(Extension, Plain, Extended) :-
        maplist(extend_atom(Extension), Plain, Extended).

extend_atom(Extension, Plain, Extended) :-
        concat(Plain, Extension, Extended).
```

In this case we would like maplist to call extend_atom/3 in the module `extend`. A name based module system will do this correctly. It will tag the atom `extend_atom` with the module and maplist will use this to construct the tagged term extend_atom/3. A name based module however, will not only tag the atoms that will eventually be used to refer to a predicate, but **all** atoms that are not declared public. So, with a name based module system also data is local to the module. This introduces another serious problem:

```
:- module(action, [action/3]).

action(Object, sleep, Arg) :- ....
action(Object, awake, Arg) :- ....

:- module(process, [awake_process/2]).

awake_process(Process, Arg) :-
        action(Process, awake, Arg).
```

This code uses a simple object-oriented implementation technique were atoms are used as method selectors. Using a name based module system, this code will not work, unless we declare the selectors public atoms in all modules that use them. Predicate based module systems do not require particular precautions for handling this case.

It appears we have to choose either to have local data, or to have trouble with meta-predicates. Probably it is best to choose for the predicate based approach as novice users will not often write generic meta-predicates that have to be used across multiple modules, but are likely to write programs that pass data around across modules. Experienced Prolog programmers should be able to deal with the complexities of meta-predicates in a predicate based module system.

4.3 Defining a Module

Modules normally are created by loading a *module file*. A module file is a file holding a `module/2` directive as its first term. The `module/2` directive declares the name and the public (i.e. externally visible) predicates of the module. The rest of the file is loaded into the module. Below is an example of a module file, defining `reverse/2`.

```
:- module(reverse, [reverse/2]).

reverse(List1, List2) :-
        rev(List1, [], List2).

rev([], List, List).
rev([Head|List1], List2, List3) :-
        rev(List1, [Head|List2], List3).
```

4.4 Importing Predicates into a Module

As explained before, in the predicate based approach adapted by SWI-Prolog, each module has it's own predicate space. In SWI-Prolog, a module initially is completely empty. Predicates can be added to a module by loading a module file as demonstrated in the previous section, using assert or by *importing* them from another module.

Two mechanisms for importing predicates explicitly from another module exist. The `use_module/[1,2]` predicates load a module file and import (part of the) public predicates of the file. The `import/1` predicate imports any predicate from any module.

use_module(+*File***)**
Load the file(s) specified with *File* just like `ensure_loaded/1`. The files should all be module files. All exported predicates from the loaded files are imported into the context module. The difference between this predicate and `ensure_loaded/1` becomes apparent if the file is already loaded into another module. In this case `ensure_loaded/1` does nothing; use_module will import all public predicates of the module into the current context module.

use_module(+*File*, +*ImportList***)**
Load the file specified with *File* (only one file is accepted). *File* should be a module file. *ImportList* is a list of name/arity pairs specifying the predicates that should be imported from the loaded module. If a predicate is specified that is not exported from the loaded module a warning will be printed. The predicate will nevertheless be imported to simplify debugging.

import(+*Head***)**
Import predicate *Head* into the current context module. *Head* should specify the source module using the ⟨*module*⟩:⟨*term*⟩ construct. Note that predicates are normally imported using one of the directives `use_module/[1,2]`. `import/1` is meant for handling imports into dynamically created modules.

It would be rather inconvenient to have to import each predicate referred to by the module, including the system predicates. For this reason each module is assigned a *default module*. All predicates in the default module are available without extra declarations. Their definition however can be overruled in the local module. This schedule is implemented by the exception handling mechanism of SWI-Prolog: if an undefined predicate exception is raised for a predicate in some module, the exception handler first tries to import the predicate from the module's default module. On success, normal execution is resumed.

4.4.1 Reserved Modules

SWI-Prolog contains two special modules. The first one is the module `system`. This module contains all built-in predicates described in this manual. Module `system` has no default module assigned to it. The second special module is the module `user`. This module forms the initial working space of the user. Initially it is empty. The default module of module `user` is `system`, making all built-in predicate definitions available as defaults. Built-in predicates thus can be overruled by defining them in module `user` before they are used.

All other modules default to module `user`. This implies they can use all predicates imported into `user` without explicitly importing them.

4.5 Using the Module System

The current structure of the module system has been designed with some specific organisations for large programs in mind. Many large programs define a basic library layer on top of which the actual program itself is defined. The module `user`, acting as the default module for all other modules of the program can be used to distribute these definitions over all program module without introducing the need to import this common layer each time explicitly. It can also be used to redefine built-in predicates if this is required to maintain compatibility to some other Prolog implementation. Typically, the loadfile of a large application looks like this:

```
:- use_module(compatibility).   % load XYZ prolog compatibility

:- use_module(                  % load generic parts
        [ error                 % errors and warnings
        , goodies               % general goodies (li-
brary extensions)
        , debug                 % application specific debugging
        , virtual_machine       % virtual machine of application
        , ...                   % more generic stuff
        ]).

:- ensure_loaded(
        [ ...                   % the application itself
        ]).
```

The 'use_module' declarations will import the public predicates from the generic modules into the `user` module. The 'ensure_loaded' directive loads the modules that constitute the actual application. It is assumed these modules import predicates from each other using `use_module/[1,2]` as far as necessary.

In combination with the object-oriented schema described below it is possible to define a neat modular architecture. The generic code defines general utilities and the message passing predicates (invoke/3 in the example below). The application modules define classes that communicate using the message passing predicates.

4.5.1 Object Oriented Programming

Another typical way to use the module system is for defining classes within an object oriented paradigm. The class structure and the methods of a class can be defined in a module and the explicit module-boundary overruling describes in section 4.6.2 can by used by the message passing code to invoke the behaviour. An outline of this mechanism is given below.

```
%       Define class point

:- module(point, []).           % class point, no exports

%        name            type,           default access
%                                        value

variable(x,             integer,        0,      both).
variable(y,             integer,        0,      both).

%         method name   predicate name  arguments

behaviour(mirror,       mirror,         []).

mirror(P) :-
        fetch(P, x, X),
        fetch(P, y, Y),
        store(P, y, X),
        store(P, x, Y).
```

The predicates fetch/3 and store/3 are predicates that change instance variables of instances. The figure below indicates how message passing can easily be implemented:

```
%       invoke(+Instance, +Selector, ?ArgumentList)
%       send a message to an instance

invoke(I, S, Args) :-
        class_of_instance(I, Class),
        Class:behaviour(S, P, ArgCheck), !,
        convert_arguments(ArgCheck, Args, ConvArgs),
        Goal =.. [P|ConvArgs],
        Class:Goal.
```

The construct ⟨*Module*⟩:⟨*Goal*⟩ explicitly calls *Goal* in module *Module*. It is discussed in more detail in section 3.8.

4.6 Meta-Predicates in Modules

As indicated in the introduction, the problem with a predicate based module system lies in the difficulty to find the correct predicate from a Prolog term. The predicate 'solution(Solution)' can exist in more than one module, but 'assert(solution(4))' in some module is supposed to refer to the correct version of solution/1.

Various approaches are possible to solve this problem. One is to add an extra argument to all predicates (e.g. 'assert(Module, Term)'). Another is to tag the term somehow to indicate which module is desired (e.g. 'assert(Module:Term)'). Both approaches are not very attractive as they make the user responsible for choosing the correct module, inviting unclear programming by asserting in other modules. The predicate `assert/1` is supposed to assert in the module it is called from and should do so without being told explicitly. For this reason, the notion *context module* has been introduced.

4.6.1 Definition and Context Module

Each predicate of the program is assigned a module, called it's *definition module*. The definition module of a predicate is always the module in which the predicate was originally defined. Each active goal in the Prolog system has a *context module* assigned to it.

The context module is used to find predicates from a Prolog term. By default, this module is the definition module of the predicate running the goal. For meta-predicates however, this is the context module of the goal that invoked them. We call this *module_transparent* in SWI-Prolog. In the 'using maplist' example above, the predicate `maplist/3` is declared module_transparent. This implies the context module remains `extend`, the context module of add_extension/3. This way `maplist/3` can decide to call extend_atom in module `extend` rather than in it's own definition module.

All built-in predicates that refer to predicates via a Prolog term are declared module_transparent. Below is the code defining maplist.

```
:- module(maplist, maplist/3).

:- module_transparent maplist/3.

%       maplist(+Goal, +List1, ?List2)
%       True if Goal can successfully be applied to all succes-
sive pairs
%       of elements of List1 and List2.

maplist(_, [], []).
maplist(Goal, [Elem1|Tail1], [Elem2|Tail2]) :-
        apply(Goal, [Elem1, Elem2]),
        maplist(Goal, Tail1, Tail2).
```

4.6.2 Overruling Module Boundaries

The mechanism above is sufficient to create an acceptable module system. There are however cases in which we would like to be able to overrule this schema and explicitly call a predicate in some module or assert explicitly in some module. The first is useful to invoke goals in some module from the user's toplevel or to implement a object-oriented system (see above). The latter is useful to create and modify *dynamic modules* (see section 4.7).

For this purpose, the reserved term `:/2` has been introduced. All built-in predicates that transform a term into a predicate reference will check whether this term is of the form '⟨*Module*⟩:⟨*Term*⟩'. If so, the predicate is searched for in *Module* instead of the goal's context module. The `:` operator may be nested, in which case the inner-most module is used.

The special calling construct ⟨*Module*⟩:⟨*Goal*⟩ pretends *Goal* is called from *Module* instead of the context module. Examples:

```
?- assert(world:done).  % asserts done/0 into module world
?- world:assert(done).  % the same
?- world:done.          % calls done/0 in module world
```

4.7 Dynamic Modules

So far, we discussed modules that were created by loading a module-file. These modules have been introduced on facilitate the development of large applications. The modules are fully defined at load-time of the application and normally will not change during execution. Having the notion of a set of predicates as a self-contained world can be attractive for other purposes as well. For example, assume an application that can reason about multiple worlds. It is attractive to store the data of a particular world in a module, so we extract information from a world simply by invoking goals in this world.

Dynamic modules can easily be created. Any built-in predicate that tries to locate a predicate in a specific module will create this module as a side-effect if it did not yet exist. Example:

```
?- assert(world_a:consistent),
   world_a:unknown(_, fail).
```

These calls create a module called ‘world_a’ and make the call ‘world_a:consistent’ succeed. Undefined predicates will not start the tracer or autoloader for this module (see `unknown/2`).

Import and export from dynamically created world is arranged via the predicates `import/1` and `export/1`:

```
?- world_b:export(solve(_,_)).          % exports solve/2 from world_b
?- world_c:import(world_b:solve(_,_)).  % and import it to world_c
```

4.8 Module Handling Predicates

This section gives the predicate definitions for the remaining built-in predicates that handle modules.

:- module(+*Module*, +*PublicList***)**
: This directive can only be used as the first term of a source file. It declares the file to be a *module file*, defining *Module* and exporting the predicates of *PublicList*. *PublicList* is a list of name/arity pairs.

module_transparent +*Preds*
: *Preds* is a comma separated list of name/arity pairs (like `dynamic/1`). Each goal associated with a transparent declared predicate will inherit the *context module* from its parent goal.

meta_predicate +*Heads*
: This predicate is defined in library(`quintus`) and provides a partial emulation of the Quintus predicate. See section 4.9.1 for details.

current_module(-*Module***)**
: Generates all currently known modules.

current_module(?*Module*, ?*File***)**
: Is true if *File* is the file from which *Module* was loaded. *File* is the internal canonical filename. See also `source_file/[1,2]`.

context_module(-*Module***)**
: Unify *Module* with the context module of the current goal. `context_module/1` itself is transparent.

export(+*Head***)**
: Add a predicate to the public list of the context module. This implies the predicate will be imported into another module if this module is imported with `use_module/[1,2]`. Note that predicates are normally exported using the directive `module/2`. `export/1` is meant to handle export from dynamically created modules.

export_list(+*Module*, ?*Exports***)**
: Unifies *Exports* with a list of terms. Each term has the name and arity of a public predicate of *Module*. The order of the terms in *Exports* is not defined. See also `predicate_property/2`.

default_module(+*Module*, -*Default***)**

Succesively unifies *Default* with the module names from which a call in *Module* attempts to use the definition. For the module `user`, this will generate `user` and `system`. For any other module, this will generate the module itself, followed by `user` and `system`.

module(+*Module***)**

The call `module(Module)` may be used to switch the default working module for the interactive toplevel (see `prolog/0`). This may be used to when debugging a module. The example below lists the clauses of file_of_label/2 in the module `tex`.

```
1 ?- module(tex).

Yes
tex: 2 ?- listing(file_of_label/2).
...
```

4.9 Compatibility of the Module System

The principles behind the module system of SWI-Prolog differ in a number of aspects from the Quintus Prolog module system.

- The SWI-Prolog module system allows the user to redefine system predicates.
- All predicates that are available in the `system` and `user` modules are visible in all other modules as well.
- Quintus has the '`meta_predicate/1`' declaration were SWI-Prolog has the `module_transparent/1` declaration.

The `meta_predicate/1` declaration causes the compiler to tag arguments that pass module sensitive information with the module using the `:/2` operator. This approach has some disadvantages:

- Changing a meta_predicate declaration implies all predicates **calling** the predicate need to be reloaded. This can cause serious consistency problems.
- It does not help for dynamically defined predicates calling module sensitive predicates.
- It slows down the compiler (at least in the SWI-Prolog architecture).
- At least within the SWI-Prolog architecture the run-time overhead is larger than the overhead introduced by the transparent mechanism.

Unfortunately the transparent predicate approach also has some disadvantages. If a predicate A passes module sensitive information to a predicate B, passing the same information to a module sensitive system predicate both A and B should be declared transparent. Using the Quintus approach only A needs to be treated special (i.e. declared with `meta_predicate/1`)[1]. A second problem arises if the body of a transparent predicate uses module sensitive predicates for which it wants to refer to its own module. Suppose we want to define `findall/3` using `assert/1` and `retract/1`[2]. The example in figure 4.1 gives the solution.

[1]Although this would make it impossible to call B directly.

[2]The system version uses `recordz/2` and `recorded/3`.

```
:- module(findall, [findall/3]).

:- dynamic
        solution/1.

:- module_transparent
        findall/3,
        store/2.

findall(Var, Goal, Bag) :-
        assert(findall:solution('$mark')),
        store(Var, Goal),
        collect(Bag).

store(Var, Goal) :-
        Goal,                   % refers to context module of
                                % caller of findall/3
        assert(findall:solution(Var)),
        fail.
store(_, _).

collect(Bag) :-
        ...,
```

Figure 4.1: `findall/3` using modules

4.9.1 Emulating meta_predicate/1

The Quintus meta_predicate/1 directive can in many cases be replaced by the transparent declaration. Below is the definition of meta_predicate/1 as available from library(quintus).

```
:- op(1150, fx, (meta_predicate)).

meta_predicate((Head, More)) :- !,
        meta_predicate1(Head),
        meta_predicate(More).
meta_predicate(Head) :-
        meta_predicate1(Head).

meta_predicate1(Head) :-
        Head =.. [Name|Arguments],
        member(Arg, Arguments),
        module_expansion_argument(Arg), !,
        functor(Head, Name, Arity),
        module_transparent(Name/Arity).
meta_predicate1(_).             % just a mode declaration

module_expansion_argument(:).
module_expansion_argument(N) :- integer(N).
```

The discussion above about the problems with the transparent mechanism show the two cases in which this simple transformation does not work.

Foreign Language Interface 5

SWI-Prolog offers a powerful interface to C [Kernighan & Ritchie, 1978]. The main design objectives of the foreign language interface are flexibility and performance. A foreign predicate is a C-function that has the same number of arguments as the predicate represented. C-functions are provided to analyse the passed terms, convert them to basic C-types as well as to instantiate arguments using unification. Non-deterministic foreign predicates are supported, providing the foreign function with a handle to control backtracking.

C can call Prolog predicates, providing both an query interface and an interface to extract multiple solutions from an non-deterministic Prolog predicate. There is no limit to the nesting of Prolog calling C, calling Prolog, etc. It is also possible to write the 'main' in C and use Prolog as an embedded logical engine.

5.1 Overview of the Interface

A special include file called `SWI-Prolog.h` should be included with each C-source file that is to be loaded via the foreign interface. The installation process installs this file in the directory `include` in the SWI-Prolog home directory (`?- current_prolog_flag(home, Home).`). This C-header file defines various data types, macros and functions that can be used to communicate with SWI-Prolog. Functions and macros can be divided into the following categories:

- Analysing Prolog terms
- Constructing new terms
- Unifying terms
- Returning control information to Prolog
- Registering foreign predicates with Prolog
- Calling Prolog from C
- Recorded database interactions
- Global actions on Prolog (halt, break, abort, etc.)

5.2 Linking Foreign Modules

Foreign modules may be linked to Prolog in three ways. Using *static linking*, the extensions, a small description file and the basic SWI-Prolog object file are linked together to form a new executable. Using *dynamic linking*, the extensions are linked to a shared library (`.so` file on most Unix systems) or dynamic-link library (`.DLL` file on Microsoft platforms) and loaded into the the running Prolog process.[1].

5.2.1 What linking is provided?

The *static linking* schema can be used on all versions of SWI-Prolog. Whether or not dynamic linking is supported can be deduced from the prolog-flag `open_shared_object` (see `current_prolog_flag/2`). If this prolog-flag yields true, `open_shared_object/2` and related predicates are defined. See section 5.4 for a suitable high-level interface to these predicates.

5.2.2 What kind of loading should I be using?

All described approaches have their advantages and disadvantages. Static linking is portable and allows for debugging on all platforms. It is relatively cumbersome and the libraries you need to pass to the linker may vary from system to system, though the utility program `plld` described in section 5.7 often hides these problems from the user.

Loading shared objects (DLL files on Windows) provides sharing and protection and is generally the best choice. If a saved-state is created using `qsave_program/[1,2]`, an `initialization/1` directive may be used to load the appropriate library at startup.

Note that the definition of the foreign predicates is the same, regardless of the linking type used.

5.3 Dynamic Linking of shared libraries

The interface defined in this section allows the user to load shared libraries (`.so` files on most Unix systems, `.dll` files on Windows). This interface is portable to Windows as well as to Unix machines providing `dlopen(2)` (Solaris, Linux, FreeBSD, Irix and many more) or `shl_open(2)` (HP/UX). It is advised to use the predicates from section 5.4 in your application.

open_shared_object(+*File*, -*Handle***)**

File is the name of a `.so` file (see your C programmers documentation on how to create a `.so` file). This file is attached to the current process and *Handle* is unified with a handle to the shared object. Equivalent to `open_shared_object(File, [global], Handle)`. See also `load_foreign_library/[1,2]`.

On errors, an exception `shared_object`(*Action*, *Message*) is raised. *Message* is the return value from `dlerror()`.

open_shared_object(+*File*, +*Options*, -*Handle***)**

As `open_shared_object/2`, but allows for additional flags to be passed. *Options* is a list of atoms. `now` implies the symbols are resolved immediately rather than lazy (default). `global` implies symbols of the loaded object are visible while loading other shared objects (by default they are local). Note that these flags may not be supported by your operating system. Check the documentation of dlopen() or equivalent on your operating system. Unsupported flags are silently ignored.

[1]The system also contains code to load `.o` files directly for some operating systems, notably Unix systems using the BSD `a.out` executable format. As the number of Unix platforms supporting this gets quickly smaller and this interface is difficult to port and slow, it is no longer described in this manual. The best alternatively would be to use the dld package on machines do not have shared libraries

close_shared_object(+*Handle***)**
: Detach the shared object identified by *Handle*.

call_shared_object_function(+*Handle*, +*Function***)**
: Call the named function in the loaded shared library. The function is called without arguments and the return-value is ignored. Normally this function installs foreign language predicates using calls to `PL_register_foreign()`.

5.4 Using the library shlib for `.DLL` and `.so` files

This section discusses the functionality of the (autoload) library `shlib.pl`, providing an interface to shared libraries. This library can only be used if the prolog-flag `open_shared_object` is enabled.

load_foreign_library(+*Lib*, +*Entry***)**
: Search for the given foreign library and link it to the current SWI-Prolog instance. The library may be specified with or without the extension. First, `absolute_file_name/3` is used to locate the file. If this succeeds, the full path is passed to the low-level function to open the library. Otherwise, the plain library name is passed, exploiting the operating-system defined search mechanism for the shared library. The `file_search_path/2` alias mechanism defines the alias `foreign`, which refers to the directories ⟨*plhome*⟩`/lib/`⟨*arch*⟩ and ⟨*plhome*⟩`/lib`, in this order.

  If the library can be loaded, the function called *Entry* will be called without arguments. The return value of the function is ignored.

  The *Entry* function will normally call `PL_register_foreign()` to declare functions in the library as foreign predicates.

load_foreign_library(+*Lib***)**
: Equivalent to `load_foreign_library/2`. For the entry-point, this function first identifies the ‘base-name’ of the library, which is defined to be the file-name with path nor extension. It will then try the entry-point `install-`⟨*base*⟩. On failure it will try to function install(). Otherwise no install function will be called.

unload_foreign_library(+*Lib***)**
: If the foreign library defines the function uninstall_⟨*base*⟩() or uninstall(), this function will be called without arguments and its return value is ignored. Next, `abolish/2` is used to remove all known foreign predicates defined in the library. Finally the library itself is detached from the process.

current_foreign_library(-*Lib*, -*Predicates***)**
: Query the currently loaded foreign libraries and their predicates. *Predicates* is a list with elements of the form *Module:Head*, indicating the predicates installed with `PL_register_foreign()` when the entry-point of the library was called.

Figure 5.1 connects a Windows message-box using a foreign function. This example was tested using Windows NT and Microsoft Visual C++ 2.0.

```
#include <windows.h>
#include <SWI-Prolog.h>

static foreign_t
pl_say_hello(term_t to)
{ char *a;

  if ( PL_get_atom_chars(to, &a) )
  { MessageBox(NULL, a, "DLL test", MB_OK|MB_TASKMODAL);

    PL_succeed;
  }

  PL_fail;
}

install_t
install()
{ PL_register_foreign("say_hello", 1, pl_say_hello, 0);
}
```

Figure 5.1: MessageBox() example in Windows NT

5.4.1 Static Linking

Below is an outline of the files structure required for statically linking SWI-Prolog with foreign extensions. `\ldots/pl` refers to the SWI-Prolog home directory (see `current_prolog_flag/2`). ⟨*arch*⟩ refers to the architecture identifier that may be obtained using `current_prolog_flag/2`.

`.../pl/runtime/`⟨*arch*⟩`/libpl.a`	SWI-Library
`\ldots/pl/include/SWI-Prolog.h`	Include file
`\ldots/pl/include/SWI-Stream.h`	Stream I/O include file
`\ldots/pl/include/SWI-Exports`	Export declarations (AIX only)
`\ldots/pl/include/stub.c`	Extension stub

The definition of the foreign predicates is the same as for dynamic linking. Unlike with dynamic linking however, there is no initialisation function. Instead, the file `\ldots/pl/include/stub.c` may be copied to your project and modified to define the foreign extensions. Below is stub.c, modified to link the lowercase example described later in this chapter:

```
/*  Copyright (c) 1991 Jan Wielemaker. All rights reserved.
    jan@swi.psy.uva.nl

    Purpose: Skeleton for extensions
*/

#include <stdio.h>
```

```
#include <SWI-Prolog.h>

extern foreign_t pl_lowercase(term, term);

PL_extension predicates[] =
{
/*{ "name",      arity,  function,      PL_FA_<flags> },*/

  { "lowercase", 2       pl_lowercase,  0 },
  { NULL,        0,      NULL,          0 }     /* terminat-
ing line */
};


int
main(int argc, char **argv)
{ PL_register_extensions(predicates);

  if ( !PL_initialise(argc, argv) )
    PL_halt(1);

  PL_install_readline();                /* delete if not re-
quired */

  PL_halt(PL_toplevel() ? 0 : 1);
}
```

Now, a new executable may be created by compiling this file and linking it to libpl.a from the runtime directory and the libraries required by both the extensions and the SWI-Prolog kernel. This may be done by hand, or using the `plld` utility described in secrefplld.

5.5 Interface Data types

5.5.1 Type `term_t`: a reference to a Prolog term

The principal data-type is `term_t`. Type `term_t` is what Quintus calls `QP_term_ref`. This name indicates better what the type represents: it is a *handle* for a term rather than the term itself. Terms can only be represented and manipulated using this type, as this is the only safe way to ensure the Prolog kernel is aware of all terms referenced by foreign code and thus allows the kernel to perform garbage-collection and/or stack-shifts while foreign code is active, for example during a callback from C.

A term reference is a C unsigned long, representing the offset of a variable on the Prolog environment-stack. A foreign function is passed term references for the predicate-arguments, one for each argument. If references for intermediate results are needed, such references may be created using `PL_new_term_ref()` or `PL_new_term_refs()`. These references normally live till the foreign function returns control back to Prolog. Their scope can be explicitly limited using `PL_open_foreign_frame()` and

`PL_close_foreign_frame()`/`PL_discard_foreign_frame()`.

A term_t always refers to a valid Prolog term (variable, atom, integer, float or compound term). A term lives either until backtracking takes us back to a point before the term was created, the garbage collector has collected the term or the term was created after a `PL_open_foreign_frame()` and `PL_discard_foreign_frame()` has been called.

The foreign-interface functions can either *read*, *unify* or *write* to term-references. In the this document we use the following notation for arguments of type term_t:

`term_t +t`	Accessed in read-mode. The '+' indicates the argument is 'input'.
`term_t -t`	Accessed in write-mode.
`term_t ?t`	Accessed in unify-mode.

Term references are obtained in any of the following ways.

- *Passed as argument*
  The C-functions implementing foreign predicates are passed their arguments as term-references. These references may be read or unified. Writing to these variables causes undefined behaviour.
- *Created by* `PL_new_term_ref()`
  A term created by `PL_new_term_ref()` is normally used to build temporary terms or be written by one of the interface functions. For example, `PL_get_arg()` writes a reference to the term-argument in its last argument.
- *Created by* `PL_new_term_refs(`*int n*`)`
  This function returns a set of term refs with the same characteristics as `PL_new_term_ref()`. See `PL_open_query()`.
- *Created by* `PL_copy_term_ref(`*term_t t*`)`
  Creates a new term-reference to the same term as the argument. The term may be written to. See figure 5.3.

Term-references can safely be copied to other C-variables of type term_t, but all copies will always refer to the same term.

`term_t` **PL_new_term_ref**()

Return a fresh reference to a term. The reference is allocated on the *local* stack. Allocating a term-reference may trigger a stack-shift on machines that cannot use sparse-memory management for allocation the Prolog stacks. The returned reference describes a variable.

`term_t` **PL_new_term_refs**(*int n*)

Return *n* new term references. The first term-reference is returned. The others are $t+1$, $t+2$, etc. There are two reasons for using this function. `PL_open_query()` expects the arguments as a set of consecutive term references and *very* time-critical code requiring a number of term-references can be written as:

```
pl_mypredicate(term_t a0, term_t a1)
{ term_t t0 = PL_new_term_refs(2);
  term_t t1 = t0+1;
```

```
    ...
  }
```

`term_t` **PL_copy_term_ref**(*term_t from*)
Create a new term reference and make it point initially to the same term as *from*. This function is commonly used to copy a predicate argument to a term reference that may be written.

`void` **PL_reset_term_refs**(*term_t after*)
Destroy all term references that have been created after *after*, including *after* itself. Any reference to the invalidated term references after this call results in undefined behaviour.

Note that returning from the foreign context to Prolog will reclaim all references used in the foreign context. This call is only necessary if references are created inside a loop that never exits back to Prolog. See also `PL_open_foreign_frame()`, `PL_close_foreign_frame()` and `PL_discard_foreign_frame()`.

Interaction with the garbage collector and stack-shifter

Prolog implements two mechanisms for avoiding stack overflow: garbage collection and stack expansion. On machines that allow for it, Prolog will use virtual memory management to detect stack overflow and expand the runtime stacks. On other machines Prolog will reallocate the stacks and update all pointers to them. To do so, Prolog needs to know which data is referenced by C-code. As all Prolog data known by C is referenced through term references (`term_t`), Prolog has all information necessary to perform its memory management without special precautions from the C-programmer.

5.5.2 Other foreign interface types

atom_t An atom in Prologs internal representation. Atoms are pointers to an opaque structure. They are a unique representation for represented text, which implies that atom A represents the same text as atom B if-and-only-if A and B are the same pointer.

Atoms are the central representation for textual constants in Prolog The transformation of C a character string to an atom implies a hash-table lookup. If the same atom is needed often, it is advised to store its reference in a global variable to avoid repeated lookup.

functor_t A functor is the internal representation of a name/arity pair. They are used to find the name and arity of a compound term as well as to construct new compound terms. Like atoms they live for the whole Prolog session and are unique.

predicate_t Handle to a Prolog predicate. Predicate handles live forever (although they can loose their definition).

qid_t Query Identifier. Used by `PL_open_query()`/`PL_next_solution()`/`PL_close_query()` to handle backtracking from C.

fid_t Frame Identifier. Used by `PL_open_foreign_frame()`/`PL_close_foreign_frame()`.

module_t A module is a unique handle to a Prolog module. Modules are used only to call predicates in a specific module.

foreign_t Return type for a C-function implementing a Prolog predicate.

control_t Passed as additional argument to non-deterministic foreign functions. See PL_retry*() and PL_foreign_context*().

install_t Type for the install() and uninstall() functions of shared or dynamic link libraries. See secrefshlib.

5.6 The Foreign Include File

5.6.1 Argument Passing and Control

If Prolog encounters a foreign predicate at run time it will call a function specified in the predicate definition of the foreign predicate. The arguments $1, \ldots, \langle arity \rangle$ pass the Prolog arguments to the goal as Prolog terms. Foreign functions should be declared of type `foreign_t`. Deterministic foreign functions have two alternatives to return control back to Prolog:

void **PL_succeed()**
Succeed deterministically. PL_succeed is defined as `return TRUE`.

void **PL_fail()**
Fail and start Prolog backtracking. PL_fail is defined as `return FALSE`.

Non-deterministic Foreign Predicates

By default foreign predicates are deterministic. Using the `PL_FA_NONDETERMINISTIC` attribute (see `PL_register_foreign()`) it is possible to register a predicate as a non-deterministic predicate. Writing non-deterministic foreign predicates is slightly more complicated as the foreign function needs context information for generating the next solution. Note that the same foreign function should be prepared to be simultaneously active in more than one goal. Suppose the natural_number_below_n/2 is a non-deterministic foreign predicate, backtracking over all natural numbers lower than the first argument. Now consider the following predicate:

```
quotient_below_n(Q, N) :-
        natural_number_below_n(N, N1),
        natural_number_below_n(N, N2),
        Q =:= N1 / N2, !.
```

In this predicate the function natural_number_below_n/2 simultaneously generates solutions for both its invocations.

Non-deterministic foreign functions should be prepared to handle three different calls from Prolog:

- *Initial call (*`PL_FIRST_CALL`*)*
  Prolog has just created a frame for the foreign function and asks it to produce the first answer.

- *Redo call (*`PL_REDO`*)*
  The previous invocation of the foreign function associated with the current goal indicated it was possible to backtrack. The foreign function should produce the next solution.

- *Terminate call (*`PL_CUTTED`*)*
  The choice point left by the foreign function has been destroyed by a cut. The foreign function is given the opportunity to clean the environment.

Both the context information and the type of call is provided by an argument of type `control_t` appended to the argument list for deterministic foreign functions. The macro `PL_foreign_control()` extracts the type of call from the control argument. The foreign function can pass a context handle using the `PL_retry*()` macros and extract the handle from the extra argument using the `PL_foreign_context*()` macro.

void **PL_retry(***long***)**
: The foreign function succeeds while leaving a choice point. On backtracking over this goal the foreign function will be called again, but the control argument now indicates it is a 'Redo' call and the macro `PL_foreign_context()` will return the handle passed via `PL_retry()`. This handle is a 30 bits signed value (two bits are used for status indication).

void **PL_retry_address(***void ****)**
: As `PL_retry()`, but ensures an address as returned by malloc() is correctly recovered by `PL_foreign_context_address()`.

int **PL_foreign_control(***control_t***)**
: Extracts the type of call from the control argument. The return values are described above. Note that the function should be prepared to handle the `PL_CUTTED` case and should be aware that the other arguments are not valid in this case.

long **PL_foreign_context(***control_t***)**
: Extracts the context from the context argument. In the call type is `PL_FIRST_CALL` the context value is 0L. Otherwise it is the value returned by the last `PL_retry()` associated with this goal (both if the call type is `PL_REDO` as `PL_CUTTED`).

*void ** **PL_foreign_context_address(***control_t***)**
: Extracts an address as passed in by `PL_retry_address()`.

Note: If a non-deterministic foreign function returns using PL_succeed or PL_fail, Prolog assumes the foreign function has cleaned its environment. **No** call with control argument `PL_CUTTED` will follow.

The code of figure 5.2 shows a skeleton for a non-deterministic foreign predicate definition.

5.6.2 Atoms and functors

The following functions provide for communication using atoms and functors.

`atom_t` **PL_new_atom**(*const char **)
: Return an atom handle for the given C-string. This function always succeeds. The returned handle is valid as long as the atom is referenced (see section 5.6.2).

`const char*` **PL_atom_chars**(*atom_t atom*)
: Return a C-string for the text represented by the given atom. The returned text will not be changed by Prolog. It is not allowed to modify the contents, not even 'temporary' as the string may reside in read-only memory.

```
typedef struct                  /* define a context structure */
{ ...
} context;

foreign_t
my_function(term_t a0, term_t a1, foreign_t handle)
{ struct context * ctxt;

  switch( PL_foreign_control(handle) )
  { case PL_FIRST_CALL:
        ctxt = malloc(sizeof(struct context));
        ...
        PL_retry_address(ctxt);
    case PL_REDO:
        ctxt = PL_foreign_context_address(handle);
        ...
        PL_retry_address(ctxt);
    case PL_CUTTED:
        free(ctxt);
        PL_succeed;
  }
}
```

Figure 5.2: Skeleton for non-deterministic foreign functions

`functor_t` **PL_new_functor**(*atom_t name, int arity*)
Returns a *functor identifier*, a handle for the name/arity pair. The returned handle is valid for the entire Prolog session.

`atom_t` **PL_functor_name**(*functor_t f*)
Return an atom representing the name of the given functor.

`int` **PL_functor_arity**(*functor_t f*)
Return the arity of the given functor.

Atoms and atom-garbage collection

With the introduction of atom-garbage collection in version 3.3.0, atoms no longer have live as long as the process. Instead, their lifetime is guaranteed only as long as they are referenced. In the single-threaded version, atom garbage collections are only invoked at the *call-port*. In the multi-threaded version (see section 3.39, they appear asynchronously, except for the invoking thread.

For dealing with atom garbage collection, two additional functions are provided:

`void` **PL_register_atom**(*atom_t atom*)
Increment the reference count of the atom by one. `PL_new_atom()` performs this automatically, returning an atom with a reference count of at least one.[2]

`void` **PL_unregister_atom**(*atom_t atom*)
Decrement the reference count of the atom. If the reference-count drops below zero, an assertion error is raised.

Please note that the following two calls are different with respect to atom garbage collection:

```
PL_unify_atom_chars(t, "text");
PL_unify_atom(t, PL_new_atom("text"));
```

The latter increments the reference count of the atom `text`, which effectively ensures the atom will never be collected. It is adviced to use the *_chars() or *_nchars() functions whenever applicable.

5.6.3 Analysing Terms via the Foreign Interface

Each argument of a foreign function (except for the control argument) is of type `term_t`, an opaque handle to a Prolog term. Three groups of functions are available for the analysis of terms. The first just validates the type, like the Prolog predicates `var/1`, `atom/1`, etc and are called `PL_is_*()`. The second group attempts to translate the argument into a C primitive type. These predicates take a `term_t` and a pointer to the appropriate C-type and return `TRUE` or `FALSE` depending on successful or unsuccessful translation. If the translation fails, the pointed-to data is never modified.

[2]Otherwise asynchronous atom garbage collection might detroy the atom before it is used.

Testing the type of a term

`int` **PL_term_type**(*term_t*)

Obtain the type of a term, which should be a term returned by one of the other interface predicates or passed as an argument. The function returns the type of the Prolog term. The type identifiers are listed below. Note that the extraction functions `PL_get_t*()` also validate the type and thus the two sections below are equivalent.

```
        if ( PL_is_atom(t) )
        { char *s;

          PL_get_atom_chars(t, &s);
          ...;
        }
```

or

```
        char *s;
        if ( PL_get_atom_chars(t, &s) )
        { ...;
        }
```

`PL_VARIABLE`	An unbound variable. The value of term as such is a unique identifier for the variable.
`PL_ATOM`	A Prolog atom.
`PL_STRING`	A Prolog string.
`PL_INTEGER`	A Prolog integer.
`PL_FLOAT`	A Prolog floating point number.
`PL_TERM`	A compound term. Note that a list is a compound term `./2`.

The functions PL_is_⟨*type*⟩ are an alternative to `PL_term_type()`. The test `PL_is_variable(`*term*`)` is equivalent to `PL_term_type(`*term*`) == PL_VARIABLE`, but the first is considerably faster. On the other hand, using a switch over `PL_term_type()` is faster and more readable then using an if-then-else using the functions below. All these functions return either `TRUE` or `FALSE`.

`int` **PL_is_variable**(*term_t*)

Returns non-zero if *term* is a variable.

`int` **PL_is_atom**(*term_t*)

Returns non-zero if *term* is an atom.

`int` **PL_is_string**(*term_t*)

Returns non-zero if *term* is a string.

`int` **PL_is_integer**(*term_t*)

Returns non-zero if *term* is an integer.

`int` **PL_is_float**(*term_t*)
Returns non-zero if *term* is a float.

`int` **PL_is_compound**(*term_t*)
Returns non-zero if *term* is a compound term.

`int` **PL_is_functor**(*term_t, functor_t*)
Returns non-zero if *term* is compound and its functor is *functor*. This test is equivalent to `PL_get_functor()`, followed by testing the functor, but easier to write and faster.

`int` **PL_is_list**(*term_t*)
Returns non-zero if *term* is a compound term with functor ./2 or the atom `[]`.

`int` **PL_is_atomic**(*term_t*)
Returns non-zero if *term* is atomic (not variable or compound).

`int` **PL_is_number**(*term_t*)
Returns non-zero if *term* is an integer or float.

Reading data from a term

The functions `PL_get_*()` read information from a Prolog term. Most of them take two arguments. The first is the input term and the second is a pointer to the output value or a term-reference.

`int` **PL_get_atom**(*term_t +t, atom_t *a*)
If *t* is an atom, store the unique atom identifier over *a*. See also `PL_atom_chars()` and `PL_new_atom()`. If there is no need to access the data (characters) of an atom, it is advised to manipulate atoms using their handle. As the atom is referenced by *t*, it will live at least as long as *t* does. If longer live-time is required, the atom should be locked using `PL_register_atom()`.

`int` **PL_get_atom_chars**(*term_t +t, char **s*)
If *t* is an atom, store a pointer to a 0-terminated C-string in *s*. It is explicitly **not** allowed to modify the contents of this string. Some built-in atoms may have the string allocated in read-only memory, so 'temporary manipulation' can cause an error.

`int` **PL_get_string_chars**(*term_t +t, char **s, int *len*)
If *t* is a string object, store a pointer to a 0-terminated C-string in *s* and the length of the string in *len*. Note that this pointer is invalidated by backtracking, garbage-collection and stack-shifts, so generally the only save operations are to pass it immediately to a C-function that doesn't involve Prolog.

`int` **PL_get_chars**(*term_t +t, char **s, unsigned flags*)
Convert the argument term *t* to a 0-terminated C-string. *flags* is a bitwise disjunction from two groups of constants. The first specifies which term-types should converted and the second how the argument is stored. Below is a specification of these constants. `BUF_RING` implies, if the data is not static (as from an atom), the data is copied to the next buffer from a ring of 16 buffers. This is a convenient way of converting multiple arguments passed to a foreign predicate to C-strings. If BUF_MALLOC is used, the data must be freed using free() when not needed any longer.

`CVT_ATOM`	Convert if term is an atom
`CVT_STRING`	Convert if term is a string
`CVT_LIST`	Convert if term is a list of integers between 1 and 255
`CVT_INTEGER`	Convert if term is an integer (using `%d`)
`CVT_FLOAT`	Convert if term is a float (using `%f`)
`CVT_NUMBER`	Convert if term is a integer or float
`CVT_ATOMIC`	Convert if term is atomic
`CVT_VARIABLE`	Convert variable to print-name
`CVT_WRITE`	Convert any term that is not converted by any of the other flags using `write/1`. If no `BUF_*` is provided, `BUF_RING` is implied.
`CVT_ALL`	Convert if term is any of the above, except for `CVT_VARIABLE` and `CVT_WRITE`
`BUF_DISCARDABLE`	Data must copied immediately
`BUF_RING`	Data is stored in a ring of buffers
`BUF_MALLOC`	Data is copied to a new buffer returned by `malloc(3)`

`int` **PL_get_list_chars**(*+term_t l, char **s, unsigned flags*)
: Same as `PL_get_chars(`l, s, *CVT_LIST*—flags`)`, provided *flags* contains no of the `CVT_*` flags.

`int` **PL_get_integer**(*+term_t t, int *i*)
: If *t* is a Prolog integer, assign its value over *i*. On 32-bit machines, this is the same as `PL_get_long()`, but avoids a warning from the compiler. See also `PL_get_long()`.

`int` **PL_get_long**(*term_t +t, long *i*)
: If *t* is a Prolog integer, assign its value over *i*. Note that Prolog integers have limited value-range. If *t* is a floating point number that can be represented as a long, this function succeeds as well.

`int` **PL_get_pointer**(*term_t +t, void **ptr*)
: In the current system, pointers are represented by Prolog integers, but need some manipulation to make sure they do not get truncated due to the limited Prolog integer range. `PL_put_pointer()`/`PL_get_pointer()` guarantees pointers in the range of malloc() are handled without truncating.

`int` **PL_get_float**(*term_t +t, double *f*)
: If *t* is a float or integer, its value is assigned over *f*.

`int` **PL_get_functor**(*term_t +t, functor_t *f*)
: If *t* is compound or an atom, the Prolog representation of the name-arity pair will be assigned over *f*. See also `PL_get_name_arity()` and `PL_is_functor()`.

`int` **PL_get_name_arity**(*term_t +t, atom_t *name, int *arity*)
: If *t* is compound or an atom, the functor-name will be assigned over *name* and the arity over *arity*. See also `PL_get_functor()` and `PL_is_functor()`.

`int` **PL_get_module**(*term_t +t, module_t *module*)
: If *t* is an atom, the system will lookup or create the corresponding module and assign an opaque pointer to it over *module*,.

`int` **PL_get_arg**(*int index, term_t +t, term_t -a*)
If *t* is compound and index is between 1 and arity (including), assign *a* with a term-reference to the argument.

`int` **_PL_get_arg**(*int index, term_t +t, term_t -a*)
Same as `PL_get_arg()`, but no checking is performed, nor whether *t* is actually a term, nor whether *index* is a valid argument-index.

Exchanging text using length and string

All internal text-representation of SWI-Prolog is represented using `char *` plus length and allow for *0-bytes* in them. The foreign library supports this by implementing a *_nchars() function for each applicable *_chars() function. Below we briefly present the signatures of these functions. For full documentation consult the *_chars() function.

`int` **PL_get_atom_nchars**(*term_t t, unsigned int len, char **s*)

`int` **PL_get_list_nchars**(*term_t t, unsigned int len, char **s*)

`int` **PL_get_nchars**(*term_t t, unsigned int len, char **s, unsigned int flags*)

`int` **PL_put_atom_nchars**(*term_t t, unsigned int len, const char *s*)

`int` **PL_put_string_nchars**(*term_t t, unsigned int len, const char *s*)

`int` **PL_put_list_ncodes**(*term_t t, unsigned int len, const char *s*)

`int` **PL_put_list_nchars**(*term_t t, unsigned int len, const char *s*)

`int` **PL_unify_atom_nchars**(*term_t t, unsigned int len, const char *s*)

`int` **PL_unify_string_nchars**(*term_t t, unsigned int len, const char *s*)

`int` **PL_unify_list_ncodes**(*term_t t, unsigned int len, const char *s*)

`int` **PL_unify_list_nchars**(*term_t t, unsigned int len, const char *s*)

In addition, the following functions are available for creating and inspecting atoms:

`atom_t` **PL_new_atom_nchars**(*unsigned int len, const char *s*)
Create a new atom as `PL_new_atom()`, but from length and characters.

`const char *` **PL_atom_nchars**(*atom_t a, unsigned int *len*)
Extract text and length of an atom.

Reading a list

The functions from this section are intended to read a Prolog list from C. Suppose we expect a list of atoms, the following code will print the atoms, each on a line:

```
foreign_t
pl_write_atoms(term_t l)
{ term_t head = PL_new_term_ref();      /* variable for the ele-
ments */
  term_t list = PL_copy_term_ref(l);    /* copy as we need to write */

  while( PL_get_list(list, head, list) )
  { char *s;

    if ( PL_get_atom_chars(head, &s) )
      Sprintf("%s\n", s);
    else
      PL_fail;
  }

  return PL_get_nil(list);              /* test end for [] */
}
```

`int` **PL_get_list**(*term_t +l, term_t -h, term_t -t*)
If *l* is a list and not `[]` assign a term-reference to the head to *h* and to the tail to *t*.

`int` **PL_get_head**(*term_t +l, term_t -h*)
If *l* is a list and not `[]` assign a term-reference to the head to *h*.

`int` **PL_get_tail**(*term_t +l, term_t -t*)
If *l* is a list and not `[]` assign a term-reference to the tail to *t*.

`int` **PL_get_nil**(*term_t +l*)
Succeeds if represents the atom `[]`.

An example: defining `write/1` in C

Figure 5.3 shows a simplified definition of `write/1` to illustrate the described functions. This simplified version does not deal with operators. It is called `display/1`, because it mimics closely the behaviour of this Edinburgh predicate.

5.6.4 Constructing Terms

Terms can be constructed using functions from the `PL_put_*()` and `PL_cons_*()` families. This approach builds the term 'inside-out', starting at the leaves and subsequently creating compound

```
foreign_t
pl_display(term_t t)
{ functor_t functor;
  int arity, len, n;
  char *s;

  switch( PL_term_type(t) )
  { case PL_VARIABLE:
    case PL_ATOM:
    case PL_INTEGER:
    case PL_FLOAT:
      PL_get_chars(t, &s, CVT_ALL);
      Sprintf("%s", s);
      break;
    case PL_STRING:
      PL_get_string_chars(t, &s, &len);
      Sprintf("\"%s\"", s);
      break;
    case PL_TERM:
    { term_t a = PL_new_term_ref();

      PL_get_name_arity(t, &name, &arity);
      Sprintf("%s(", PL_atom_chars(name));
      for(n=1; n<=arity; n++)
      { PL_get_arg(n, t, a);
        if ( n > 1 )
          Sprintf(", ");
        pl_display(a);
      }
      Sprintf(")");
      break;
    default:
      PL_fail;                          /* should not happen */
  }

  PL_succeed;
}
```

Figure 5.3: A Foreign definition of `display/1`

terms. Alternatively, terms may be created 'top-down', first creating a compound holding only variables and subsequently unifying the arguments. This section discusses functions for the first approach. This approach is generally used for creating arguments for `PL_call()` and PL_open_query.

`void` **PL_put_variable**(*term_t -t*)
: Put a fresh variable in the term. The new variable lives on the global stack. Note that the initial variable lives on the local stack and is lost after a write to the term-references. After using this function, the variable will continue to live.

`void` **PL_put_atom**(*term_t -t, atom_t a*)
: Put an atom in the term reference from a handle. See also `PL_new_atom()` and `PL_atom_chars()`.

`void` **PL_put_atom_chars**(*term_t -t, const char *chars*)
: Put an atom in the term-reference constructed from the 0-terminated string. The string itself will never be references by Prolog after this function.

`void` **PL_put_string_chars**(*term_t -t, const char *chars*)
: Put a zero-terminated string in the term-reference. The data will be copied. See also `PL_put_string_nchars()`.

`void` **PL_put_string_nchars**(*term_t -t, unsigned int len, const char *chars*)

: Put a string, represented by a length/start pointer pair in the term-reference. The data will be copied. This interface can deal with 0-bytes in the string. See also section 5.6.18.

`void` **PL_put_list_chars**(*term_t -t, const char *chars*)
: Put a list of ASCII values in the term-reference.

`void` **PL_put_integer**(*term_t -t, long i*)
: Put a Prolog integer in the term reference.

`void` **PL_put_pointer**(*term_t -t, void *ptr*)
: Put a Prolog integer in the term-reference. Provided ptr is in the 'malloc()-area', `PL_get_pointer()` will get the pointer back.

`void` **PL_put_float**(*term_t -t, double f*)
: Put a floating-point value in the term-reference.

`void` **PL_put_functor**(*term_t -t, functor_t functor*)
: Create a new compound term from *functor* and bind *t* to this term. All arguments of the term will be variables. To create a term with instantiated arguments, either instantiate the arguments using the `PL_unify_*()` functions or use `PL_cons_functor()`.

`void` **PL_put_list**(*term_t -l*)
: Same as `PL_put_functor(`*l, PL_new_functor(PL_new_atom(".")*`, 2))`.

`void` **PL_put_nil**(*term_t -l*)
: Same as `PL_put_atom_chars(`*"[]"*`)`.

`void` **PL_put_term**(*term_t -t1, term_t +t2*)
Make *t1* point to the same term as *t2*.

`void` **PL_cons_functor**(*term_t -h, functor_t f, . . .*)
Create a term, whose arguments are filled from variable argument list holding the same number of term_t objects as the arity of the functor. To create the term `animal(gnu, 50)`, use:

```
        term_t a1 = PL_new_term_ref();
        term_t a2 = PL_new_term_ref();
        term_t t  = PL_new_term_ref();

        PL_put_atom_chars(a1, "gnu");
        PL_put_integer(a2, 50);
        PL_cons_functor(t, PL_new_functor(PL_new_atom("animal"), 2),
                        a1, a2);
```

After this sequence, the term-references *a1* and *a2* may be used for other purposes.

`void` **PL_cons_functor_v**(*term_t -h, functor_t f, term_t a0*)
Creates a compound term like `PL_cons_functor()`, but *a0* is an array of term references as returned by `PL_new_term_refs()`. The length of this array should match the number of arguments required by the functor.

`void` **PL_cons_list**(*term_t -l, term_t +h, term_t +t*)
Create a list (cons-) cell in *l* from the head and tail. The code below creates a list of atoms from a `char **`. The list is built tail-to-head. The `PL_unify_*()` functions can be used to build a list head-to-tail.

```
void
put_list(term_t l, int n, char **words)
{ term_t a = PL_new_term_ref();

  PL_put_nil(l);
  while( --n >= 0 )
  { PL_put_atom_chars(a, words[n]);
    PL_cons_list(l, a, l);
  }
}
```

Note that *l* can be redefined within a `PL_cons_list` call as shown here because operationally its old value is consumed before its new value is set.

5.6.5 Unifying data

The functions of this sections *unify* terms with other terms or translated C-data structures. Except for `PL_unify()`, the functions of this section are specific to SWI-Prolog. They have been introduced to make translation of old code easier, but also because they provide for a faster mechanism for returning data to Prolog that requires less term-references. Consider the case where we want a foreign function to return the host name of the machine Prolog is running on. Using the `PL_get_*()` and `PL_put_*()` functions, the code becomes:

```
foreign_t
pl_hostname(term_t name)
{ char buf[100];

  if ( gethostname(buf, sizeof(buf)) )
  { term_t tmp = PL_new_term_ref();

    PL_put_atom_chars(tmp, buf);
    return PL_unify(name, buf);
  }

  PL_fail;
}
```

Using `PL_unify_atom_chars()`, this becomes:

```
foreign_t
pl_hostname(term_t name)
{ char buf[100];

  if ( gethostname(buf, sizeof(buf)) )
    return PL_unify_atom_chars(name, buf);

  PL_fail;
}
```

`int` **PL_unify**(*term_t ?t1, term_t ?t2*)
: Unify two Prolog terms and return non-zero on success.

`int` **PL_unify_atom**(*term_t ?t, atom_t a*)
: Unify *t* with the atom *a* and return non-zero on success.

`int` **PL_unify_atom_chars**(*term_t ?t, const char *chars*)
: Unify *t* with an atom created from *chars* and return non-zero on success.

`int` **PL_unify_list_chars**(*term_t ?t, const char *chars*)
: Unify *t* with a list of ASCII characters constructed from *chars*.

`void` **PL_unify_string_chars**(*term_t ?t, const char *chars*)
: Unify *t* with a Prolog string object created from the zero-terminated string *chars*. The data will be copied. See also `PL_unify_string_nchars()`.

`void` **PL_unify_string_nchars**(*term_t ?t, unsigned int len, const char *chars*)
: Unify *t* with a Prolog string object created from the string created from the *len*/*chars* pair. The data will be copied. This interface can deal with 0-bytes in the string. See also section 5.6.18.

`int` **PL_unify_integer**(*term_t ?t, long n*)
Unify *t* with a Prolog integer from *n*.

`int` **PL_unify_float**(*term_t ?t, double f*)
Unify *t* with a Prolog float from *f*.

`int` **PL_unify_pointer**(*term_t ?t, void *ptr*)
Unify *t* with a Prolog integer describing the pointer. See also `PL_put_pointer()` and `PL_get_pointer()`.

`int` **PL_unify_functor**(*term_t ?t, functor_t f*)
If *t* is a compound term with the given functor, just succeed. If it is unbound, create a term and bind the variable, else fails. Not that this function does not create a term if the argument is already instantiated.

`int` **PL_unify_list**(*term_t ?l, term_t -h, term_t -t*)
Unify *l* with a list-cell (`./2`). If successful, write a reference to the head of the list to *h* and a reference to the tail of the list in *t*. This reference may be used for subsequent calls to this function. Suppose we want to return a list of atoms from a `char **`. We could use the example described by `PL_put_list()`, followed by a call to `PL_unify()`, or we can use the code below. If the predicate argument is unbound, the difference is minimal (the code based on `PL_put_list()` is probably slightly faster). If the argument is bound, the code below may fail before reaching the end of the word-list, but even if the unification succeeds, this code avoids a duplicate (garbage) list and a deep unification.

```
foreign_t
pl_get_environ(term_t env)
{ term_t l = PL_copy_term_ref(env);
  term_t a = PL_new_term_ref();
  extern char **environ;
  char **e;

  for(e = environ; *e; e++)
  { if ( !PL_unify_list(l, a, l) ||
         !PL_unify_atom_chars(a, *e) )
      PL_fail;
  }

  return PL_unify_nil(l);
}
```

`int` **PL_unify_nil**(*term_t ?l*)
Unify *l* with the atom `[]`.

`int` **PL_unify_arg**(*int index, term_t ?t, term_t ?a*)
Unifies the *index-th* argument (1-based) of *t* with *a*.

`int` **PL_unify_term**(*term_t ?t, ...*)

Unify *t* with a (normally) compound term. The remaining arguments is a sequence of a type identifier, followed by the required arguments. This predicate is an extension to the Quintus and SICStus foreign interface from which the SWI-Prolog foreign interface has been derived, but has proved to be a powerful and comfortable way to create compound terms from C. Due to the vararg packing/unpacking and the required type-switching this interface is slightly slower than using the primitives. Please note that some bad C-compilers have fairly low limits on the number of arguments that may be passed to a function.

The type identifiers are:

`PL_VARIABLE` ***none***
: No op. Used in arguments of `PL_FUNCTOR`.

`PL_ATOM` ***atom_t***
: Unify the argument with an atom, as in `PL_unify_atom()`.

`PL_INTEGER` ***long***
: Unify the argument with an integer, as in `PL_unify_integer()`.

`PL_FLOAT` ***double***
: Unify the argument with a float, as in `PL_unify_float()`. Note that, as the argument is passed using the C vararg conventions, a float must be casted to a double explicitly.

`PL_STRING` ***const char ****
: Unify the argument with a string object, as in `PL_unify_string_chars()`.

`PL_TERM` ***term_t***
: Unify a subterm. Note this may the return value of a `PL_new_term_ref()` call to get access to a variable.

`PL_CHARS` ***const char ****
: Unify the argument with an atom, constructed from the C `char *`, as in `PL_unify_atom_chars()`.

`PL_FUNCTOR` ***functor_t, ...***
: Unify the argument with a compound term. This specification should be followed by exactly as many specifications as the number of arguments of the compound term.

`PL_LIST` ***int length, ...***
: Create a list of the indicated length. The following arguments contain the elements of the list.

For example, to unify an argument with the term `language(dutch)`, the following skeleton may be used:

```
static functor_t FUNCTOR_language1;

static void
init_constants()
{ FUNCTOR_language1 = PL_new_functor(PL_new_atom("language"), 1);
}
```

```
foreign_t
pl_get_lang(term_t r)
{ return PL_unify_term(r,
                       PL_FUNCTOR, FUNCTOR_language1,
                           PL_CHARS, "dutch");
}

install_t
install()
{ PL_register_foreign("get_lang", 1, pl_get_lang, 0);
  init_constants();
}
```

`int` **PL_chars_to_term**(*const char *chars, term_t -t*)

Parse the string *chars* and put the resulting Prolog term into *t*. *chars* may or may not be closed using a Prolog full-stop (i.e. a dot followed by a blank). Returns `FALSE` if a syntax error was encountered and `TRUE` after successful completion. In addition to returning `FALSE`, the exception-term is returned in *t* on a syntax error. See also `term_to_atom/2`.

The following example build a goal-term from a string and calls it.

```
int
call_chars(const char *goal)
{ fid_t fid = PL_open_foreign_frame();
  term_t g = PL_new_term_ref();
  BOOL rval;

  if ( PL_string_to_term(goal, g) )
    rval = PL_call(goal, NULL);
  else
    rval = FALSE;

  PL_discard_foreign_frame(fid);
  return rval;
}

  ...
  call_chars("consult(load)");
  ...
```

`char *` **PL_quote**(*int chr, const char *string*)

Return a quoted version of *string*. If *chr* is `'\''`, the result is a quoted atom. If *chr* is `'"'`, the result is a string. The result string is stored in the same ring of buffers as described with the `BUF_RING` argument of `PL_get_chars()`;

In the current implementation, the string is surrounded by *chr* and any occurence of *chr* is doubled. In the future the behaviour will depend on the `character_escape` prolog-flag. See `current_prolog_flag/2`.

5.6.6 Calling Prolog from C

The Prolog engine can be called from C. There are two interfaces for this. For the first, a term is created that could be used as an argument to `call/1` and next `PL_call()` is used to call Prolog. This system is simple, but does not allow to inspect the different answers to a non-deterministic goal and is relatively slow as the runtime system needs to find the predicate. The other interface is based on `PL_open_query()`, `PL_next_solution()` and `PL_cut_query()` or `PL_close_query()`. This mechanism is more powerful, but also more complicated to use.

Predicate references

This section discusses the functions used to communicate about predicates. Though a Prolog predicate may defined or not, redefined, etc., a Prolog predicate has a handle that is not destroyed, nor moved. This handle is known by the type `predicate_t`.

`predicate_t` **PL_pred**(*functor_t f, module_t m*)
Return a handle to a predicate for the specified name/arity in the given module. This function always succeeds, creating a handle for an undefined predicate if no handle was available.

`predicate_t` **PL_predicate**(*const char *name, int arity, const char* module*)
Same a `PL_pred()`, but provides a more convenient interface to the C-programmer.

`void` **PL_predicate_info**(*predicate_t p, atom_t *n, int *a, module_t *m*)
Return information on the predicate *p*. The name is stored over *n*, the arity over *a*, while *m* receives the definition module. Note that the latter need not be the same as specified with `PL_predicate()`. If the predicate was imported into the module given to `PL_predicate()`, this function will return the module where the predicate was defined.

Initiating a query from C

This section discusses the functions for creating and manipulating queries from C. Note that a foreign context can have at most one active query. This implies it is allowed to make strictly nested calls between C and Prolog (Prolog calls C, calls Prolog, calls C, etc., but it is **not** allowed to open multiple queries and start generating solutions for each of them by calling `PL_next_solution()`. Be sure to call `PL_cut_query()` or `PL_close_query()` on any query you opened before opening the next or returning control back to Prolog.

`qid_t` **PL_open_query**(*module_t ctx, int flags, predicate_t p, term_t +t0*)

Opens a query and returns an identifier for it. This function always succeeds, regardless whether the predicate is defined or not. *ctx* is the *context module* of the goal. When `NULL`, the context module of the calling context will be used, or `user` if there is no calling context (as may happen in embedded systems). Note that the context module only matters for *module_transparent* predicates. See `context_module/1` and `module_transparent/1`. The *p* argument specifies the predicate, and should be the result of a call to `PL_pred()` or `PL_predicate()`. Note that it is allowed to store this handle as global data and reuse it for future queries. The term-reference *t0* is the first of a vector of term-references as returned by `PL_new_term_refs(`*n*`)`.

The *flags* arguments provides some additional options concerning debugging and exception handling. It is a bitwise or of the following values:

`PL_Q_NORMAL`
: Normal operation. The debugger inherits its settings from the environment. If an exception occurs that is not handled in Prolog, a message is printed and the tracer is started to debug the error.[3]

`PL_Q_NODEBUG`
: Switch off the debugger while executing the goal. This option is used by many calls to hook-predicates to avoid tracing the hooks. An example is `print/1` calling `portray/1` from foreign code.

`PL_Q_CATCH_EXCEPTION`
: If an exception is raised while executing the goal, do not report it, but make it available for `PL_exception()`.

`PL_Q_PASS_EXCEPTION`
: As `PL_Q_CATCH_EXCEPTION`, but do not invalidate the exception-term while calling `PL_close_query()`. This option is experimental.

The example below opens a query to the predicate is_a/2 to find the ancestor of for some name.

```
char *
ancestor(const char *me)
{ term_t a0 = PL_new_term_refs(2);
  static predicate_t p;

  if ( !p )
    p = PL_predicate("is_a", 2, "database");

  PL_put_atom_chars(a0, me);
  PL_open_query(NULL, PL_Q_NORMAL, p, a0);
  ...
}
```

`int` **PL_next_solution**(*qid_t qid*)
: Generate the first (next) solution for the given query. The return value is `TRUE` if a solution was found, or `FALSE` to indicate the query could not be proven. This function may be called repeatedly until it fails to generate all solutions to the query.

`void` **PL_cut_query**(*qid*)
: Discards the query, but does not delete any of the data created by the query. It just invalidate *qid*, allowing for a new call to `PL_open_query()` in this context.

`void` **PL_close_query**(*qid*)
: As `PL_cut_query()`, but all data and bindings created by the query are destroyed.

`int` **PL_call_predicate**(*module_t m*, *int flags*, *predicate_t pred*, *term_t +t0*)
: Shorthand for `PL_open_query()`, `PL_next_solution()`, `PL_cut_query()`, generating a single solution. The arguments are the same as for `PL_open_query()`, the return value is the same as `PL_next_solution()`.

[3]Do not pass the integer 0 for normal operation, as this is interpreted as `PL_Q_NODEBUG` for backward compatibility reasons.

`int` **PL_call**(*term_t, module_t*)

Call term just like the Prolog predicate `once/1`. *Term* is called in the specified module, or in the context module if module_t = NULL. Returns `TRUE` if the call succeeds, `FALSE` otherwise. Figure 5.4 shows an example to obtain the number of defined atoms. All checks are omitted to improve readability.

5.6.7 Discarding Data

The Prolog data created and term-references needed to setup the call and/or analyse the result can in most cases be discarded right after the call. `PL_close_query()` allows for destructing the data, while leaving the term-references. The calls below may be used to destroy term-references and data. See figure 5.4 for an example.

`fid_t` **PL_open_foreign_frame**()

Created a foreign frame, holding a mark that allows the system to undo bindings and destroy data created after it as well as providing the environment for creating term-references. This function is called by the kernel before calling a foreign predicate.

`void` **PL_close_foreign_frame**(*fid_t id*)

Discard all term-references created after the frame was opened. All other Prolog data is retained. This function is called by the kernel whenever a foreign function returns control back to Prolog.

`void` **PL_discard_foreign_frame**(*fid_t id*)

Same as `PL_close_foreign_frame()`, but also undo all bindings made since the open and destroy all Prolog data.

`void` **PL_rewind_foreign_frame**(*fid_t id*)

Undo all bindings and discard all term-references created since the frame was created, but does not pop the frame. I.e. the same frame can be rewinded multiple times, and must eventually be closed or discarded.

It is obligatory to call either of the two closing functions to discard a foreign frame. Foreign frames may be nested.

5.6.8 Foreign Code and Modules

Modules are identified via a unique handle. The following functions are available to query and manipulate modules.

`module_t` **PL_context**()

Return the module identifier of the context module of the currently active foreign predicate.

`int` **PL_strip_module**(*term_t +raw, module_t *m, term_t -plain*)

Utility function. If *raw* is a term, possibly holding the module construct ⟨*module*⟩`:`⟨*rest*⟩ this function will make *plain* a reference to ⟨*rest*⟩ and fill *module* * with ⟨*module*⟩. For further nested module constructs the inner most module is returned via *module* *. If *raw* is not a module construct *arg* will simply be put in *plain*. If *module* * is `NULL` it will be set to the context module. Otherwise it will be left untouched. The following example shows how to obtain the plain term and module if the default module is the user module:

```
int
count_atoms()
{ fid_t fid = PL_open_foreign_frame();
  term_t goal  = PL_new_term_ref();
  term_t a1    = PL_new_term_ref();
  term_t a2    = PL_new_term_ref();
  functor_t s2 = PL_new_functor(PL_new_atom("statistics"), 2);
  int atoms;

  PL_put_atom_chars(a1, "atoms");
  PL_cons_functor(goal, s2, a1, a2);
  PL_call(goal, NULL);         /* call it in current module */

  PL_get_integer(a2, &atoms);
  PL_discard_foreign_frame(fid);

  return atoms;
}
```

Figure 5.4: Calling Prolog

```
    { module m = PL_new_module(PL_new_atom("user"));
      term_t plain = PL_new_term_ref();

      PL_strip_module(term, &m, plain);
      ...
```

`atom_t` **PL_module_name**(*module_t*)
  Return the name of *module* as an atom.

`module_t` **PL_new_module**(*atom_t name*)
  Find an existing or create a new module with name specified by the atom *name*.

5.6.9 Prolog exceptions in foreign code

This section discusses `PL_exception()`, `PL_throw()` and `PL_raise_exception()`, the interface functions to detect and generate Prolog exceptions from C-code. `PL_throw()` and `PL_raise_exception()` from the C-interface to raise an exception from foreign code. `PL_throw()` exploits the C-function longjmp() to return immediately to the innermost `PL_next_solution()`. `PL_raise_exception()` registers the exception term and returns `FALSE`. If a foreign predicate returns FALSE, while and exception-term is registered a Prolog exception will be raised by the virtual machine.

Calling these functions outside the context of a function implementing a foreign predicate results in undefined behaviour.

`PL_exception()` may be used after a call to `PL_next_solution()` fails, and returns a term reference to an exception term if an exception was raised, and 0 otherwise.

If a C-function, implementing a predicate calls Prolog and detects an exception using `PL_exception()`, it can handle this exception, or return with the exception. Some caution is required though. It is **not** allowed to call `PL_close_query()` or `PL_discard_foreign_frame()` afterwards, as this will invalidate the exception term. Below is the code that calls a Prolog defined arithmetic function (see `arithmethic_function/1`).

If `PL_next_solution()` succeeds, the result is analysed and translated to a number, after which the query is closed and all Prolog data created after `PL_open_foreign_frame()` is destroyed. On the other hand, if `PL_next_solution()` fails and if an exception was raised, just pass it. Otherwise generate an exception (`PL_error()` is an internal call for building the standard error terms and calling `PL_raise_exception()`). After this, the Prolog environment should be discarded using `PL_cut_query()` and `PL_close_foreign_frame()` to avoid invalidating the exception term.

```
static int
prologFunction(ArithFunction f, term_t av, Number r)
{ int arity = f->proc->definition->functor->arity;
  fid_t fid = PL_open_foreign_frame();
  qid_t qid;
  int rval;

  qid = PL_open_query(NULL, PL_Q_NORMAL, f->proc, av);

  if ( PL_next_solution(qid) )
  { rval = valueExpression(av+arity-1, r);
    PL_close_query(qid);
    PL_discard_foreign_frame(fid);
  } else
  { term_t except;

    if ( (except = PL_exception(qid)) )
    { rval = PL_throw(except);          /* pass exception */
    } else
    { char *name = stringAtom(f->proc->definition->functor->name);

                                        /* generate exception */
      rval = PL_error(name, arity-1, NULL, ERR_FAILED, f->proc);
    }

    PL_cut_query(qid);                  /* donot destroy data */
    PL_close_foreign_frame(fid);        /* same */
  }

  return rval;
}
```

`int` **PL_raise_exception**(*term_t exception*)

Generate an exception (as `throw/1`) and return `FALSE`. Below is an example returning an exception from foreign predicate:

```
foreign_t
pl_hello(term_t to)
{ char *s;

  if ( PL_get_atom_chars(to, &s) )
  { Sprintf("Hello \"%s\"\n", s);

    PL_succeed;
  } else
  { term_t except = PL_new_term_ref();

    PL_unify_term(except,
                  PL_FUNCTOR, PL_new_functor(PL_new_atom("type_error"), 2),
                    PL_CHARS, "atom",
                    PL_TERM,  to);

    return PL_raise_exception(except);
  }
}
```

`int` **PL_throw**(*term_t exception*)
: Similar to `PL_raise_exception()`, but returns using the C longjmp() function to the innermost `PL_next_solution()`.

`term_t` **PL_exception**(*qid_t qid*)
: If `PL_next_solution()` fails, this can be due to normal failure of the Prolog call, or because an exception was raised using `throw/1`. This function returns a handle to the exception term if an exception was raised, or 0 if the Prolog goal simply failed.[4].

5.6.10 Foreign code and Prolog threads

If SWI-Prolog has been build to support multi-threading (see section 3.39), all foreign-code linked to Prolog should be thread-safe (*reentrant*) or guarded in Prolog using `with_mutex/2` from simultaneous access from multiple Prolog threads. On Unix systems, this generally implies the code should be compiled with the `-D_REENTRANT` flag passed to the compiler. Please note that on many Unix systems not all systemcalls and library-functions are thread-safe. Consult your manual for details.

If you are using SWI-Prolog as an embedded engine in a multi-threaded application you can access the Prolog engine from multiple threads by creating an *engine* in each thread from which you call Prolog. Without creating an engine, a thread can only use functions that do not use the `term_t` type (for example `PL_new_atom()`).

Please note that the interface below will only work if threading in your application is based on the same thread-library as used to compile SWI-Prolog.

[4]This interface differs in two ways from Quintus. The calling predicates simp,y signal failure if an exception was raised, and a term referenced is returned, rather passed and filled with the error term. Exceptions can only be handled using the `PL_next_solution()` interface, as a handle to the query is required

`int` **PL_thread_self**()

Returns the integer Prolog identifier of the engine or -1 if the calling thread has no Prolog engine. This function is also provided in the single-threaded version of SWI-Prolog, where it returns -2.

`int` **PL_thread_attach_engine**(*PL_thread_attr_t *attr*)

Creates a new Prolog engine in the calling thread. If the calling thread already has an engine the reference count of the engine is incremented. The *attr* argument can be `NULL` to create a thread with default attributes. Otherwise it is a pointer to a structure with the definition below. For any field with value '0', the default is used.

```
typedef struct
{ unsigned long     local_size;    /* Stack sizes (K-bytes) */
  unsigned long     global_size;
  unsigned long     trail_size;
  unsigned long     argument_size;
  char *            alias;         /* alias name */
} PL_thread_attr_t;
```

The structure may be destroyed after `PL_thread_attach_engine()` has returned. If an error occurs, -1 is returned. If this Prolog is not compiled for multi-threading, -2 is returned.

`int` **PL_thread_destroy_engine**()

Destroy the Prolog engine in the calling thread. Only takes effect if `PL_thread_destroy_engine()` is called as many times as `PL_thread_attach_engine()` in this thread. Returns `TRUE` on success and `FALSE` if the calling thread has no engine or this Prolog does not support threads.

Please note that construction and destruction of engines are relatively expensive operations. Only destroy an engine if performance is not critical and memory is a critical resource. The engine is automatically destroyed if the thread finishes, regardless how many times `PL_thread_attach_engine()` has been called.

5.6.11 Miscellaneous

Term Comparison

`int` **PL_compare**(*term_t t1, term_t t2*)

Compares two terms using the standard order of terms and returns -1, 0 or 1. See also `compare/3`.

`int` **PL_same_compound**(*term_t t1, term_t t2*)

Yields `TRUE` if *t1* and *t2* refer to physically the same compound term and `FALSE` otherwise.

Recorded database

In some applications it is useful to store and retreive Prolog terms from C-code. For example, the XPCE graphical environment does this for storing arbitrary Prolog data as slot-data of XPCE objects.

Please note that the returned handles have no meaning at the Prolog level and the recorded terms are not visible from Prolog. The functions `PL_recorded()` and `PL_erase()` are the only functions that can operate on the stored term.

Two groups of functions are provided.The first group (`PL_record()` and friends) store Prolog terms on the Prolog heap for retrieval during the same session. These functions are also used by `recorda/3` and friends. The recorded database may be used to communicate Prolog terms between threads.

`record_t` **PL_record**(*term_t +t*)

Record the term *t* into the Prolog database as `recorda/3` and return an opaque handle to the term. The returned handle remains valid until `PL_erase()` is called on it. `PL_recorded()` is used to copy recorded terms back to the Prolog stack.

`void` **PL_recorded**(*record_t record, term_t -t*)

Copy a recorded term back to the Prolog stack. The same record may be used to copy multiple instances at any time to the Prolog stack. See also `PL_record()` and `PL_erase()`.

`void` **PL_erase**(*record_t record*)

Remove the recorded term from the Prolog database, reclaiming all associated memory resources.

The second group (headed by `PL_record_external()`) provides the same functionality, but the returned data has properties that enable storing the data on an external device. It has been designed to make it possible to store Prolog terms fast an compact in an external database. Here are the main features:

- *Independent of session*
  Records can be communicated to another Prolog session and made visible using `PL_recorded_external()`.

- *Binary*
  The representation is binary for maximum performance. The returned data may contain 0-bytes.

- *Byte-order independent*
  The representation can be transferred between machines with different byte-order.

- *No alignment restrictions*
  There are no memory alignment restrictions and copies of the record can thus be moved freely. For example, it is possible to use this representation to exchange terms using shared memory between different Prolog processes.

- *Compact*
  It is assumed that a smaller memory footprint will eventually outperform slightly faster representations.

- *Stable*
  The format is designed for future enhancements without breaking compatibility with older records.

`char *` **PL_record_external**(*term_t +t, unsigned int *len*)

Record the term *t* into the Prolog database as `recorda/3` and return an opaque handle to the term. The returned handle remains valid until `PL_erase()` is called on it.

It is allowed to copy the data and use `PL_recorded_external()` on the copy. The user is responsible for the memory management of the copy. After copying, the original may be discarded using `PL_erase_external()`.

`PL_recorded_external()` is used to copy such recorded terms back to the Prolog stack.

`int` **PL_recorded_external**(*const char *record, term_t -t*)

Copy a recorded term back to the Prolog stack. The same record may be used to copy multiple instances at any time to the Prolog stack. See also `PL_record_external()` and `PL_erase_external()`.

`int` **PL_erase_external**(*char *record*)

Remove the recorded term from the Prolog database, reclaiming all associated memory resources.

5.6.12 Catching Signals (Software Interrupts)

SWI-Prolog offers both a C and Prolog interface to deal with software interrupts (signals). The Prolog mapping is defined in section 3.10. This subsection deals with handling signals from C.

If a signal is not used by Prolog and the handler does not call Prolog in any way, the native signal interface routines may be used.

Some versions of SWI-Prolog, notably running on popular Unix platforms, handle `SIG_SEGV` for guarding the Prolog stacks. If the application whishes to handle this signal too, it should use `PL_signal()` to install its handler after initialisating Prolog. SWI-Prolog will pass `SIG_SEGV` to the user code if it detected the signal is not related to a Prolog stack overflow.

Any handler that wishes to call one of the Prolog interface functions should call `PL_signal()` for its installation.

`void (*)()` **PL_signal**(*sig, func*)

This function is equivalent to the BSD-Unix signal() function, regardless of the platform used. The signal handler is blocked while the signal routine is active, and automatically reactivated after the handler returns.

After a signal handler is registered using this function, the native signal interface redirects the signal to a generic signal handler inside SWI-Prolog. This generic handler validates the environment, creates a suitable environment for calling the interface functions described in this chapter and finally calls the registered user-handler.

5.6.13 Errors and warnings

`PL_warning()` prints a standard Prolog warning message to the standard error (`user_error`) stream. Please note that new code should consider using `PL_raise_exception()` to raise a Prolog exception. See also section 3.9.

`int` **PL_warning**(*format, a1, ...*)

Print an error message starting with '`[WARNING: `', followed by the output from *format*,

`PL_ACTION_TRACE`	Start Prolog tracer (`trace/0`). Requires no arguments.
`PL_ACTION_DEBUG`	Switch on Prolog debug mode (`debug/0`). Requires no arguments.
`PL_ACTION_BACKTRACE`	Print backtrace on current output stream. The argument (an int) is the number of frames printed.
`PL_ACTION_HALT`	Halt Prolog execution. This action should be called rather than Unix exit() to give Prolog the opportunity to clean up. This call does not return. The argument (an int) is the exit code. See `halt/1`.
`PL_ACTION_ABORT`	Generate a Prolog abort (`abort/0`). This call does not return. Requires no arguments.
`PL_ACTION_BREAK`	Create a standard Prolog break environment (`break/0`). Returns after the user types the end-of-file character. Requires no arguments.
`PL_ACTION_GUIAPP`	Win32: Used to indicate the kernel that the application is a GUI application if the argument is not 0 and a console application if the argument is 0. If a fatal error occurs, the system uses a windows messagebox to report this on a GUI application and simply prints the error and exits otherwise.
`PL_ACTION_WRITE`	Write the argument, a `char *` to the current output stream.
`PL_ACTION_FLUSH`	Flush the current output stream. Requires no arguments.

Table 5.1: `PL_action()` options

followed by a '`]`' and a newline. Then start the tracer. *format* and the arguments are the same as for `printf(2)`. Always returns `FALSE`.

5.6.14 Environment Control from Foreign Code

`int` **PL_action**(*int, ...*)

Perform some action on the Prolog system. *int* describes the action, Remaining arguments depend on the requested action. The actions are listed in table 5.1.

5.6.15 Querying Prolog

`C_type` **PL_query**(*int*)

Obtain status information on the Prolog system. The actual argument type depends on the information required. *int* describes what information is wanted. The options are given in table 5.2.

5.6.16 Registering Foreign Predicates

`int` **PL_register_foreign**(*const char *name, int arity, foreign_t (*function)(), int flags*)

Register a C-function to implement a Prolog predicate. After this call returns successfully a predicate with name *name* (a char *) and arity *arity* (a C int) is created. As a special case, *name* may consist of a sequence of alpha-numerical characters followed by the colon (`:`). In

`PL_QUERY_ARGC`	Return an integer holding the number of arguments given to Prolog from Unix.
`PL_QUERY_ARGV`	Return a char ** holding the argument vector given to Prolog from Unix.
`PL_QUERY_SYMBOLFILE`	Return a char * holding the current symbol file of the running process.
`PL_MAX_INTEGER`	Return a long, representing the maximal integer value represented by a Prolog integer.
`PL_MIN_INTEGER`	Return a long, representing the minimal integer value.
`PL_QUERY_VERSION`	Return a long, representing the version as $10,000 \times M + 100 \times m + p$, where M is the major, m the minor version number and p the patch-level. For example, `20717` means `2.7.17`.

Table 5.2: `PL_query()` options

this case the name uptil the colon is taken to be the destination module and the rest of the name the predicate name.

When called in Prolog, Prolog will call *function*. *flags* forms bitwise or'ed list of options for the installation. These are:

`PL_FA_NOTRACE`	Predicate cannot be seen in the tracer
`PL_FA_TRANSPARENT`	Predicate is module transparent
`PL_FA_NONDETERMINISTIC`	Predicate is non-deterministic. See also `PL_retry()`.
`PL_FA_VARARGS`	Use alternative calling convention.

`void` **PL_load_extensions**(*PL_extension *e*)

Register foreign predicates from a table of structures. This is an alternative to multiple calls to `PL_register_foreign()` and simplifies code that wishes to use `PL_register_extensions()` as an alternative. The type `PL_extension` is defined as:

```
typedef struct _PL_extension
{ char          *predicate_name; /* Name of the predicate */
  short         arity;           /* Arity of the predicate */
  pl_function_t function;        /* Implementing functions */
  short         flags;           /* Or of PL_FA_... */
} PL_extension;
```

`void` **PL_register_extensions**(*PL_extension *e*)

The function `PL_register_extensions()` behaves as `PL_load_extensions()`, but is the only PL_* function that may be called **before** `PL_initialise()`. The predicates are registered **into the module** `user` after registration of the SWI-Prolog builtin foreign predicates and before loading the initial saved state. This implies that `initialization/1` directives can refer to them.

Here is an example of its usage:

```
static PL_extension predicates[] = {
{ "foo",        1,      pl_foo, 0 },
{ "bar",        2,      pl_bar, PL_FA_NONDETERMINISTIC },
{ NULL,         0,      NULL,   0 }
};

main(int argc, char **argv)
{ PL_register_extensions(predicates);

  if ( !PL_initialise(argc, argv) )
    PL_halt(1);

  ...
}
```

5.6.17 Foreign Code Hooks

For various specific applications some hooks re provided.

`PL_dispatch_hook_t` **PL_dispatch_hook**(*PL_dispatch_hook_t*)

If this hook is not NULL, this function is called when reading from the terminal. It is supposed to dispatch events when SWI-Prolog is connected to a window environment. It can return two values: `PL_DISPATCH_INPUT` indicates Prolog input is available on file descriptor 0 or `PL_DISPATCH_TIMEOUT` to indicate a timeout. The old hook is returned. The type `PL_dispatch_hook_t` is defined as:

```
typedef int  (*PL_dispatch_hook_t)(void);
```

`void` **PL_abort_hook**(*PL_abort_hook_t*)

Install a hook when `abort/0` is executed. SWI-Prolog `abort/0` is implemented using C setjmp()/longjmp() construct. The hooks are executed in the reverse order of their registration after the longjmp() took place and before the Prolog toplevel is reinvoked. The type `PL_abort_hook_t` is defined as:

```
typedef void (*PL_abort_hook_t)(void);
```

`int` **PL_abort_unhook**(*PL_abort_hook_t*)

Remove a hook installed with `PL_abort_hook()`. Returns `FALSE` if no such hook is found, `TRUE` otherwise.

`void` **PL_on_halt**(*void (*f)(int, void *), void *closure*)

Register the function *f* to be called if SWI-Prolog is halted. The function is called with two arguments: the exit code of the process (0 if this cannot be determined on your operating system) and the *closure* argument passed to the `PL_on_halt()` call. See also `at_halt/1`.

`PL_agc_hook_t` **PL_agc_hook**(*PL_agc_hook_t new*)

Register a hook with the atom-garbage collector (see `garbage_collect_atoms/0` that is called on any atom that is reclaimed. The old hook is returned. If no hook is currently defined, `NULL` is returned. The argument of the called hook is the atom that is to be garbage collected. The return value is an `int`. If the return value is zero, the atom is **not** reclaimed. The hook may invoke any Prolog predicate.

The example below defines a foreign library for printing the garbage collected atoms for debugging purposes.

```
#include <SWI-Stream.h>
#include <SWI-Prolog.h>

static int
atom_hook(atom_t a)
{ Sdprintf("AGC: deleting %s\n", PL_atom_chars(a));

  return TRUE;
}

static PL_agc_hook_t old;

install_t
install()
{ old = PL_agc_hook(atom_hook);
}

install_t
uninstall()
{ PL_agc_hook(old);
}
```

5.6.18 Storing foreign data

This section provides some hints for handling foreign data in Prolog. With foreign data, we refer to data that is used by foreign language predicates and needs to be passed around in Prolog. Excluding combinations, there are three principal options for storing such data

- *Natural Prolog data*
  E.i. using the representation one would choose if there was no foreign interface required.
- *Opaque packed Prolog data*
  Data can also be represetented in a foreign structure and stored on the Prolog stacks using `PL_put_string_nchars()` and retrieved using `PL_get_string_chars()`. It is generally good practice to wrap the string in a compound term with arity 1, so Prolog can identify the type. `portray/1` rules may be used to streamline printing such terms during development.

- *Natural foreign data, passing a pointer*
  An alternative is to pass a pointer to the foreign data. Again, this functor may be wrapped in a compound term.

The choice may be guided using the following distinctions

- *Is the data opaque to Prolog*
  With 'opaque' data, we refer to data handled in foreign functions, passed around in Prolog, but of which Prolog never examines the contents of the data itself. If the data is opaque to Prolog, the choosen representation does not depend on simple analysis by Prolog, and the selection will be driven solely by simplicity of the interface and performance (both in time and space).

- *How big is the data*
  Is effient encoding required? For examine, a boolean aray may be expressed as a compound term, holding integers each of which contains a number of bits, or as a list of `true` and `false`.

- *What is the nature of the data*
  For examples in C, constants are often expressed using 'enum' or #define'd integer values. If prolog needs to handle this data, atoms are a more logical choice. Whether or not this mapping is used depends on whether Prolog needs to interpret the data, how important debugging is and how important performance is.

- *What is the lifetime of the data*
  We can distinguish three cases.

  1. The lifetime is dictated by the accesibility of the data on the Prolog stacks. Their is no way by which the foreign code when the data becomes 'garbage', and the data thus needs to be represented on the Prolog stacks using Prolog data-types. (2),
  2. The data lives on the 'heap' and is explicitly allocated and deallocated. In this case, representing the data using native foreign representation and passing a pointer to it is a sensible choice.
  3. The data lives as during the lifetime of a foreign predicate. If the predicate is deterministic, foreign automatic variables are suitable. if the predicate is non-deterministic, the data may be allocated using malloc() and a pointer may be passed. See section 5.6.1.

Examples for storing foreign data

In this section, we wull outline some examples, covering typical cases. In the first example, we will deal with extending Prolog's data representation with integer-sets, represented as bit-vectors. In the second example, we look at handling a 'netmask'. Finally, we discuss the outline of the DDE interface.

Integer sets with not-to-far-apart upper- and lower-bounds can be represented using bit-vectors. Common set operations, such as union, intersection, etc. are reduced to simple and'ing and or'ing the bitvectors. This can be done in Prolog, using a compound term holding integer arguments. Especially if the integers are kept below the maximum tagged integer value (see `current_prolog_flag/2`), this representation is fairly space-efficient (wasting 1 word for the functor and and 7 bits per integer for the tags). Arithmetic can all be performed in Prolog too.

For really demanding applications, foreign representation will perform better, especially time-wise. Bit-vectors are natrually expressed using string objects. If the string is wrapped in `bitvector/1`, lower-bound of the vector is 0, and the upperbound is not defined, an implementation for getting and putting the setes as well as the union predicate for it is below.

```
#include <SWI-Prolog.h>

#define max(a, b) ((a) > (b) ? (a) : (b))
#define min(a, b) ((a) < (b) ? (a) : (b))

static functor_t FUNCTOR_bitvector1;

static int
get_bitvector(term_t in, int *len, unsigned char **data)
{ if ( PL_is_functor(in, FUNCTOR_bitvector1) )
  { term_t a = PL_new_term_ref();

    PL_get_arg(1, in, a);
    return PL_get_string(a, (char **)data, len);
  }

  PL_fail;
}

static int
unify_bitvector(term_t out, int len, const unsigned char *data)
{ if ( PL_unify_functor(out, FUNCTOR_bitvector1) )
  { term_t a = PL_new_term_ref();

    PL_get_arg(1, out, a);

    return PL_unify_string_nchars(a, len, (const char *)data);
  }

  PL_fail;
}

static foreign_t
pl_bitvector_union(term_t t1, term_t t2, term_t u)
{ unsigned char *s1, *s2;
  int l1, l2;

  if ( get_bitvector(t1, &l1, &s1) &&
       get_bitvector(t2, &l2, &s2) )
  { int l = max(l1, l2);
    unsigned char *s3 = alloca(l);
    if ( s3 )
    { int n;
      int ml = min(l1, l2);

      for(n=0; n<ml; n++)
        s3[n] = s1[n] | s2[n];
      for( ; n < l1; n++)
        s3[n] = s1[n];
      for( ; n < l2; n++)
        s3[n] = s2[n];

      return unify_bitvector(u, l, s3);
    }

    return PL_warning("Not enough memory");
  }

  PL_fail;
}


install_t
install()
{ PL_register_foreign("bitvector_union", 3, pl_bitvector_union, 0);

  FUNCTOR_bitvector1 = PL_new_functor(PL_new_atom("bitvector"), 1);
}
```

Netmask's are used with TCP/IP configuration. Suppose we have an application dealing with reasoning about a network configuration. Such an application requires communicating netmask structures from the operating system, reasoning about them and possibly communicate them to the user. A netmask consists of 4 bitmasks between 0 and 255. C-application normally see them as an 4-byte wide unsigned integer. SWI-Prolog cannot do that, as integers are always signed.

We could use the string approach outlined above, but this makes it hard to handle these terms in Prolog. A better choice is a compound term `netmask/4`, holding the 4 submasks as integer arguments.

As the implementation is trivial, we will omit this here.

The DDE interface (see section 3.46) represents another common usage of the foreign interface: providing communication to new operating system features. The DDE interface requires knowledge about active DDE server and client channels. These channels contains various foreign data-types. Such an interface is normally achieved using an open/close protocol that creates and destroys a *handle*. The handle is a reference to a foreign data-structure containing the relevant information.

There are a couple of possibilities for representing the handle. The choice depends on responsibilities and debugging facilities. The simplest aproach is to using `PL_unify_pointer()` and

`PL_get_pointer()`. This approach is fast and easy, but has the drawbacks of (untyped) pointers: there is no reliable way to detect the validity of the pointer, not to verify it is pointing to a structure of the desired type. The pointer may be wrapped into a compound term with arity 1 (i.e. `dde_channel(`⟨*Pointer*⟩`)`), making the type-problem less serious.

Alternatively (used in the DDE interface), the interface code can maintain a (preferably variable length) array of pointers and return the index in this array. This provides better protection. Especially for debugging purposes, wrapping the handle in a compound is a good suggestion.

5.6.19 Embedding SWI-Prolog in a C-program

As of version 2.1.0, SWI-Prolog may be embedded in a C-program. To reach at a compiled C-program with SWI-Prolog as an embedded application is very similar to creating a statically linked SWI-Prolog executable as described in section 5.4.1.

The file `\ldots/pl/include/stub.c` defines SWI-Prologs default main program:

```
int
main(int argc, char **argv)
{ if ( !PL_initialise(argc, argv) )
    PL_halt(1);

  PL_install_readline();        /* delete if you don't want read-
line */

  PL_halt(PL_toplevel() ? 0 : 1);
}
```

This may be replaced with your own main C-program. The interface function `PL_initialise()` **must** be called before any of the other SWI-Prolog foreign language functions described in this chapter. `PL_initialise()` interprets all the command-line arguments, except for the `-t` *`toplevel`* flag that is interpreted by `PL_toplevel()`.

`int` **PL_initialise**(*int argc, char **argv*)

Initialises the SWI-Prolog heap and stacks, restores the boot QLF file, loads the system and personal initialisation files, runs the `at_initialization/1` hooks and finally runs the `-g` *`goal`* hook.

Special consideration is required for `argv[0]`. On **Unix**, this argument passes the part of the commandline that is used to locate the executable. Prolog uses this to find the file holding the running executable. The **Windows** version uses this to find a *module* of the running executable. If the specified module cannot be found, it tries the module `libpl.dll`, containing the Prolog runtime kernel. In all these cases, the resulting file is used for two purposes

- See whether a Prolog saved-state is appended to the file. If this is the case, this state will be loaded instead of the default `boot.prc` file from the SWI-Prolog home directory. See also `qsave_program/[1,2]` and section 5.7.
- Find the Prolog home directory. This process is described in detail in section 5.8.

`PL_initialise()` returns 1 if all initialisation succeeded and 0 otherwise.[5]

In most cases, *argc* and *argv* will be passed from the main program. It is allowed to create your own argument vector, provided `argv[0]` is constructed according to the rules above. For example:

```
int
main(int argc, char **argv)
{ char *av[10];
  int ac = 0;

  av[ac++] = argv[0];
  av[ac++] = "-x";
  av[ac++] = "mystate";
  av[ac]   = NULL;

  if ( !PL_initialise(ac, av) )
    PL_halt(1);
  ...
}
```

Please note that the passed argument vector may be referred from Prolog at any time and should therefore be valid as long as the Prolog engine is used.

A good setup in Windows is to add SWI-Prolog's `bin` directory to your `PATH` and either pass a module holding a saved-state, or `"libpl.dll"` as `argv[0]`.

`int` **PL_is_initialised**(*int *argc, char ***argv*)
: Test whether the Prolog engine is already initialised. Returns `FALSE` if Prolog is not initialised and `TRUE` otherwise. If the engine is initialised and *argc* is not `NULL`, the argument count used with `PL_initialise()` is stored in *argc*. Same for the argument vector *argv*.

`void` **PL_install_readline**()
: Installs the GNU-readline line-editor. Embedded applications that do not use the Prolog toplevel should normally delete this line, shrinking the Prolog kernel significantly.

`int` **PL_toplevel**()
: Runs the goal of the `-t` *`toplevel`* switch (default `prolog/0`) and returns 1 if successful, 0 otherwise.

`void` **PL_cleanup**(*int status*)
: This function performs the reverse of `PL_initialise()`. It runs the `PL_on_halt()` and `at_halt/1` handlers, closes all streams (except for the 'standard I/O' streams which are flushed only), deallocates all memory and restores all signal handlers. The *status* argument is passed to the various termination hooks and indicates the *exit-status*.

  This function allows deleting and restarting the Prolog system in the same process. Use it with care, as `PL_initialise()` is a costly function. Unix users should consider using exec() (available as part of the clib package,).

[5]BUG: Various fatal errors may cause PL_initialise to call `PL_halt(`*1*`)`, preventing it from returning at all.

`void` **PL_halt**(*int status*)

Cleanup the Prolog environment using `PL_cleanup()` and calls exit() with the status argument.

5.7 Linking embedded applications using plld

The utility program `plld` (Win32: plld.exe) may be used to link a combination of C-files and Prolog files into a stand-alone executable. `plld` automates most of what is described in the previous sections.

In the normal usage, a copy is made of the default embedding template `\ldots/pl/include/stub.c`. The main() routine is modified to suit your application. `PL_initialise()` **must** be passed the program-name (*argv[0]*) (Win32: the executing program can be obtained using `GetModuleFileName()`). The other elements of the command-line may be modified. Next, `plld` is typically invoked as:

```
plld -o output stubfile.c [other-c-or-o-files] [plfiles]
```

`plld` will first split the options into various groups for both the C-compiler and the Prolog compiler. Next, it will add various default options to the C-compiler and call it to create an executable holding the user's C-code and the Prolog kernel. Then, it will call the SWI-Prolog compiler to create a saved state from the provided Prolog files and finally, it will attach this saved state to the created emulator to create the requested executable.

Below, it is described how the options are split and which additional options are passed.

-help

Print brief synopsis.

-pl *prolog*

Select the prolog to use. This prolog is used for two purposes: get the home-directory as well as the compiler/linker options and create a saved state of the Prolog code.

-ld *linker*

Linker used to link the raw executable. Default is to use the C-compiler (Win32: link.exe).

-cc *C-compiler*

Compiler for `.c` files found on the commandline. Default is the compiler used to build SWI-Prolog (see `current_prolog_flag/2`) (Win32: cl.exe).

-c++ *C++-compiler*

Compiler for C++ sources (extensions `.cpp`, `.cxx`, `.cc` or `.C`) files found on the commandline. Default is `c++` or `g++` if the C-compiler is `gcc`) (Win32: cl.exe).

-nostate

Just relink the kernel, do not add any Prolog code to the new kernel. This is used to create a new kernel holding additional foreign predicates on machines that do not support the shared-library (DLL) interface, or if building the state cannot be handled by the default procedure used by `plld`. In the latter case the state is created seperately and appended to the kernel using `cat` ⟨*kernel*⟩ ⟨*state*⟩ `>` ⟨*out*⟩ (Win32: `copy /b` ⟨*kernel*⟩+⟨*state*⟩ ⟨*out*⟩)

-pl-options ,. . .
: Additional options passed to Prolog when creating the saved state. The first character immediately following `pl-options` is used as separator and translated to spaces when the argument is built. Example: `-pl-options,-F,xpce` passed `-F xpce` as additional flags to Prolog.

-ld-options ,. . .
: Passes options to the linker, similar to `-pl-options`.

-cc-options ,. . .
: Passes options to the C/C++ compiler, similar to `-pl-options`.

-v
: Select verbose operation, showing the various programs and their options.

-o *outfile*
: Reserved to specify the final output file.

-l*library*
: Specifies a library for the C-compiler. By default, `-lpl` (Win32: libpl.lib) and the libraries needed by the Prolog kernel are given.

-L*library-directory*
: Specifies a library directory for the C-compiler. By default the directory containing the Prolog C-library for the current architecture is passed.

`-g` | `-I`*`include-directory`* | `-D`*`definition`*
: These options are passed to the C-compiler. By default, the include directory containing `SWI-Prolog.h` is passed. `plld` adds two additional `* -D`def flags:

  -D`__SWI_PROLOG__`
  : Indicates the code is to be connected to SWI-Prolog.

  -D`__SWI_EMBEDDED__`
  : Indicates the creation of an embedded program.

**.o* | **.c* | **.C* | **.cxx* | **.cpp*
: Passed as input files to the C-compiler

.pl* |.qlf*
: Passed as input files to the Prolog compiler to create the saved-state.

`*`
: I.e. all other options. These are passed as linker options to the C-compiler.

5.7.1 A simple example

The following is a very simple example going through all the steps outlined above. It provides an arithmetic expression evaluator. We will call the application `calc` and define it in the files `calc.c` and `calc.pl`. The Prolog file is simple:

```
calc(Atom) :-
        term_to_atom(Expr, Atom),
        A is Expr,
        write(A),
        nl.
```

The C-part of the application parses the command-line options, initialises the Prolog engine, locates the calc/1 predicate and calls it. The coder is in figure 5.5.
The application is now created using the following command-line:

```
% plld -o calc calc.c calc.pl
```

The following indicates the usage of the application:

```
% calc pi/2
1.5708
```

5.8 The Prolog 'home' directory

Executables embedding SWI-Prolog should be able to find the 'home' directory of the development environment unless a self-contained saved-state has been added to the executable (see `qsave_program/[1,2]` and section 5.7).

If Prolog starts up, it will try to locate the development environment. To do so, it will try the following steps until one succeeds.

1. If the environment variable `SWI_HOME_DIR` is defined and points to an existing directory, use this.
2. If the environment variable `SWIPL` is defined and points to an existing directory, use this.
3. Locate the primary executable or (Windows only) a component (*module*) thereof and check whether the parent directory of the directory holding this file contains the file `swipl`. If so, this file contains the (relative) path to the home directory. If this directory exists, use this. This is the normal mechanism used by the binary distribution.
4. If the precompiled path exists, use it. This is only useful for a source installation.

If all fails and there is no state attached to the executable or provided Windows module (see `PL_initialise()`), SWI-Prolog gives up. If a state is attached, the current working directory is used.

The `file_search_path/2` alias `swi` is set to point to the home directory located.

5.9 Example of Using the Foreign Interface

Below is an example showing all stages of the declaration of a foreign predicate that transforms atoms possibly holding uppercase letters into an atom only holding lower case letters. Figure 5.6 shows the C-source file, figure 5.7 illustrates compiling and loading of foreign code.

```
#include <stdio.h>
#include <SWI-Prolog.h>

#define MAXLINE 1024

int
main(int argc, char **argv)
{ char expression[MAXLINE];
  char *e = expression;
  char *program = argv[0];
  char *plav[2];
  int n;

  /* combine all the arguments in a single string */

  for(n=1; n<argc; n++)
  { if ( n != 1 )
      *e++ = ' ';
    strcpy(e, argv[n]);
    e += strlen(e);
  }

  /* make the argument vector for Prolog */

  plav[0] = program;
  plav[1] = NULL;

  /* initialise Prolog */

  if ( !PL_initialise(1, plav) )
    PL_halt(1);

  /* Lookup calc/1 and make the arguments and call */

  { predicate_t pred = PL_predicate("calc", 1, "user");
    term_t h0 = PL_new_term_refs(1);
    int rval;

    PL_put_atom_chars(h0, expression);
    rval = PL_call_predicate(NULL, PL_Q_NORMAL, pred, h0);

    PL_halt(rval ? 0 : 1);
  }

  return 0;
}
```

Figure 5.5: C-source for the calc application

```
/*  Include file depends on local installation */
#include <SWI-Prolog.h>
#include <stdlib.h>
#include <ctype.h>

foreign_t
pl_lowercase(term_t u, term_t l)
{ char *copy;
  char *s, *q;
  int rval;

  if ( !PL_get_atom_chars(u, &s) )
    return PL_warning("lowercase/2: instantiation fault");
  copy = malloc(strlen(s)+1);

  for( q=copy; *s; q++, s++)
    *q = (isupper(*s) ? tolower(*s) : *s);
  *q = '\0';

  rval = PL_unify_atom_chars(l, copy);
  free(copy);

  return rval;
}

install_t
install()
{ PL_register_foreign("lowercase", 2, pl_lowercase, 0);
}
```

Figure 5.6: Lowercase source file

```
% gcc -I/usr/local/lib/pl-\plversion/include -fpic -c lowercase.c
% gcc -shared -o lowercase.so lowercase.o
% pl
Welcome to SWI-Prolog (Version \plversion)
Copyright (c) 1993-1996 University of Amsterdam.  All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- load_foreign_library(lowercase).

Yes
2 ?- lowercase('Hello World!', L).

L = 'hello world!'

Yes
```

Figure 5.7: Compiling the C-source and loading the object file

5.10 Notes on Using Foreign Code

5.10.1 Memory Allocation

SWI-Prolog's memory allocation is based on the `malloc(3)` library routines. Foreign applications can safely use `malloc(3)`, `realloc(3)` and `free(3)`. Memory allocation using `brk(2)` or `sbrk(2)` is not allowed as these calls conflict with `malloc(3)`.

5.10.2 Debugging Foreign Code

Statically linked foreign code or embedded systems can be debugged normally. Most modern environments provide debugging tools for dynamically loaded shared objects or dynamic load libraries. The following example traces the code of lowercase using `gdb(1)` in a Unix environment.

```
% gcc -I/usr/local/lib/pl-2.2.0/include -fpic -c -g lowercase.c
% gcc -shared -o lowercase.so lowercase.o
% gdb pl
(gdb) r
Welcome to SWI-Prolog (Version \plversion)
Copyright (c) 1993-1996 University of Amsterdam.  All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

?- load_foreign_library(lowercase).
<type Control-C>
(gdb) shared                   % loads symbols for shared objects
(gdb) break pl_lowercase
(gdb) continue
?- lowercase('HELLO', X).
```

5.10.3 Name Conflicts in C modules

In the current version of the system all public C functions of SWI-Prolog are in the symbol table. This can lead to name clashes with foreign code. Someday I should write a program to strip all these symbols from the symbol table (why does Unix not have that?). For now I can only suggest to give your function another name. You can do this using the C preprocessor. If—for example—your foreign package uses a function warning(), which happens to exist in SWI-Prolog as well, the following macro should fix the problem.

```
#define warning warning_
```

Note that shared libraries do not have this problem as the shared library loader will only look for symbols in the main executable for symbols that are not defined in the library itself.

5.10.4 Compatibility of the Foreign Interface

The term-reference mechanism was first used by Quintus Prolog version 3. SICStus Prolog version 3 is strongly based on the Quintus interface. The described SWI-Prolog interface is similar to using the

Quintus or SICStus interfaces, defining all foreign-predicate arguments of type `+term`. SWI-Prolog explicitly uses type `functor_t`, while Quintus and SICStus uses ⟨*name*⟩ and ⟨*arity*⟩. As the names of the functions differ from Prolog to Prolog, a simple macro layer dealing with the names can also deal with this detail. For example:

```
#define QP_put_functor(t, n, a) PL_put_functor(t, PL_new_functor(n, a))
```

The `PL_unify_*()` functions are lacking from the Quintus and SICStus interface. They can easily be emulated or the put/unify approach should be used to write compatible code.

The `PL_open_foreign_frame()`/`PL_close_foreign_frame()` combination is lacking from both other Prologs. SICStus has `PL_new_term_refs(`*0*`)`, followed by `PL_reset_term_refs()` that allows for discarding term references.

The Prolog interface for the graphical user interface package XPCE shares about 90% of the code using a simple macro layer to deal with different naming and calling conventions of the interfaces.

6 Generating Runtime Applications

This chapter describes the features of SWI-Prolog for delivering applications that can run without the development version of the system installed.

A SWI-Prolog runtime executable is a file consisting of two parts. The first part is the *emulator*, which is machine dependent. The second part is the *resource archive*, which contains the compiled program in a machine-independent format, startup options and possibly user-defined *resources*, see `resource/3` and `open_resource/3`.

These two parts can be connected in various different ways. The most common way for distributed runtime applications is to *concatenate* the two parts. This can be achieved using external commands (Unix: `cat`, Windows: `copy`), or using the `stand_alone` option to `qsave_program/2`. The second option is to attach a startup script in front of the resource that starts the emulator with the proper options. This is the default under Unix. Finally, an emulator can be told to use a specified resource file using the `-x` commandline switch.

qsave_program(+*File*, +*ListOfOptions***)**

Saves the current state of the program to the file *File*. The result is a resource archive containing a saved-state that expresses all Prolog data from the running program and all user-defined resources. Depending on the `stand_alone` option, the resource is headed by the emulator, a Unix shell-script or nothing.

ListOfOptions is a list of ⟨*Key*⟩ = ⟨*Value*⟩ or ⟨*Key*⟩(⟨*Value*⟩) pairs. The available keys are described in table 6.1.

Before writing the data to file, `qsave_program/2` will run `autoload/0` to all required autoloading the system can discover. See `autoload/0`.

Provided the application does not require any of the Prolog libraries to be loaded at runtime, the only file from the SWI-Prolog development environment required is the emulator itself. The emulator may be built in two flavours. The default is the *development emulator*. The *runtime emulator* is similar, but lacks the tracer.

If the option `stand_alone(on)` is present, the emulator is the first part of the state. If the emulator is started it will test whether a boot-file (state) is attached to the emulator itself and load this state. Provided the application has all libraries loaded, the resulting executable is completely independent of the runtime environment or location where it was build.

See also section 2.10.2.

qsave_program(+*File***)**

Equivalent to `qsave_program(File, [])`.

autoload

Check the current Prolog program for predicates that are referred to, are undefined and have a definition in the Prolog library. Load the appropriate libraries.

Key	Option	Type	Description
local	**-L**	K-bytes	Size (Limit) of local stack
global	**-G**	K-bytes	Size (Limit) of global stack
trail	**-T**	K-bytes	Size (Limit) of trail stack
argument	**-A**	K-bytes	Size (Limit) of argument stack
goal	**-g**	atom	Initialisation goal
toplevel	**-t**	atom	Prolog toplevel goal
init_file	**-f**	atom	Personal initialisation file
class		atom	If `runtime`, only read resources from the state (default). If `kernel`, lock all predicates as system predicates If `development`, save the predicates in their current state and keep reading resources from their source (if present). See also `resource/3`.
autoload		bool	If true, run `autoload/0` first
map		file	File to write info on dump
op		`save/standard`	Save operator declarations?
stand_alone		bool	Include the emulator in the state
emulator		file	Emulator attached to the (stand-alone) executable. Default is the running emulator.

Table 6.1: ⟨*Key*⟩ = ⟨*Value*⟩ pairs for `qsave_program/2`

This predicate is used by `qsave_program/[1,2]` to ensure the saved state will not depend on one of the libraries. The predicate `autoload/0` will find all **direct** references to predicates. It does not find predicates referenced via meta-predicates. The predicate log/2 is defined in the library(quintus) to provide a quintus compatible means to compute the natural logarithm of a number. The following program will behave correctly if its state is executed in an environment where the library(quintus) is not available:

```
logtable(From, To) :-
        From > To, !.
logtable(From, To) :-
        log(From, Value),
        format('~d~t~8|~2f~n', [From, Value]),
        F is From + 1,
        logtable(F, To).
```

However, the following implementation refers to log/2 through the meta-predicate `maplist/3`. Autoload will not be able to find the reference. This problem may be fixed either by loading the module libtary(quintus) explicitly or use `require/1` to tell the system that the predicate log/2 is required by this module.

```
logtable(From, To) :-
        findall(X, between(From, To, X), Xlist),
        maplist(log, Xlist, SineList),
        write_table(Xlist, SineList).

write_table([], []).
write_table([I|IT], [V|VT]) :-
        format('˜d˜t˜8|˜2f˜n', [I, V]),
        write_table(IT, VT).
```

volatile *+Name/Arity, . . .*
Declare that the clauses of specified predicates should **not** be saved to the program. The volatile declaration is normally used to avoid that the clauses of dynamic predicates that represent data for the current session is saved in the state file.

6.1 Limitations of qsave_program

There are three areas that require special attention when using `qsave_program/[1,2]`.

- If the program is an embedded Prolog application or uses the foreign language interface, care has to be taken to restore the appropriate foreign context. See section 6.2 for details.
- If the program uses directives (`:- goal.` lines) that perform other actions then setting predicate attributes (dynamic, volatile, etc.) or loading files (consult, etc.), the directive may need to be prefixed with `initialization/1`.
- Database references as returned by `clause/3`, `recorded/3`, etc. are not preserved and may thus not be part of the database when saved.

6.2 Runtimes and Foreign Code

Some applications may need to use the foreign language interface. Object code is by definition machine-dependent and thus cannot be part of the saved program file.

To complicate the matter even further there are various ways of loading foreign code:

- *Using the library(shlib) predicates*
  This is the preferred way of dealing with foreign code. It loads quickly and ensures an acceptable level of independence between the versions of the emulator and the foreign code loaded. It works on Unix machines supporting shared libraries and library functions to load them. Most modern Unixes, as well as Win32 (Windows 95/NT) satisfy this constraint.
- *Static linking*
  This mechanism works on all machines, but generally requires the same C-compiler and linker to be used for the external code as is used to build SWI-Prolog itself.

To make a runtime executable that can run on multiple platforms one must make runtime checks to find the correct way of linking. Suppose we have a source-file `myextension` defining the installation function `install()`.

If this file is compiled into a shared library, `load_foreign_library/1` will load this library and call the installation function to initialise the foreign code. If it is loaded as a static extension, define `install()` as the predicate `install/0`:

```
static foreign_t
pl_install()
{ install();

  PL_succeed;
}

PL_extension PL_extensions [] =
{
/*{ "name",     arity,  function,       PL_FA_<flags> },*/

  { "install",  0,      pl_install,     0 },
  { NULL,       0,      NULL,           0 }     /* terminat-
ing line */
};
```

Now, use the following Prolog code to load the foreign library:

```
load_foreign_extensions :-
        current_predicate(install, install), !, % static loaded
        install.
load_foreign_extensions :-                      % shared library
        load_foreign_library(foreign(myextension)).

:- initialization load_foreign_extensions.
```

The path alias `foreign` is defined by `file_search_path/2`. By default it searches the directories ⟨*home*⟩`/lib/`⟨*arch*⟩ and ⟨*home*⟩`/lib`. The application can specify additional rules for `file_search_path/2`.

6.3 Using program resources

A *resource* is very similar to a file. Resources however can be represented in two different formats: on files, as well as part of the resource *archive* of a saved-state (see `qsave_program/2`).

A resource has a *name* and a *class*. The *source* data of the resource is a file. Resources are declared by declaring the predicate `resource/3`. They are accessed using the predicate `open_resource/3`.

Before going into details, let us start with an example. Short texts can easily be expressed in Prolog sourcecode, but long texts are cumbersome. Assume our application defines a command 'help' that prints a helptext to the screen. We put the content of the helptext into a file called `help.txt`. The following code implements our help command such that help.txt is incorperated into the runtime executable.

```
resource(help, text, 'help.txt').

help :-
        open_resource(help, text, In),
        copy_stream(In, user_output),
        close(In).

copy_stream(In, Out) :-
        get0(In, C),
        copy_stream(C, In, Out).

copy_stream(-1, _, _) :- !.
copy_stream(C, In, Out) :-
        put(Out, C),
        get0(In, C2),
        copy_stream(C2, In, Out).
```

The predicate `help/0` opens the resource as a Prolog stream. If we are executing this from the development environment, this will actually return a stream to the `gelp.txt` itself. When executed from the saved-state, the stream will actually be a stream opened on the program resource file, taking care of the offset and length of the resource.

6.3.1 Predicates Definitions

resource(*+Name, +Class, +FileSpec***)**

This predicate is defined as a dynamic predicate in the module `user`. Clauses for it may be defined in any module, including the user module. *Name* is the name of the resource (an atom). A resource name may contain any character, except for $ and :, which are reserved for internal usage by the resource library. *Class* describes the what kind of object is stored in the resource. In the current implementation, it is just an atom. *FileSpec* is a file specification that may exploit `file_search_path/2` (see `absolute_file_name/2`).

Normally, resources are defined as unit clauses (facts), but the definition of this predicate also allows for rules. For proper generation of the saved state, it must be possible to enumerate the available resources by calling this predicate with all its arguments unbound.

Dynamic rules are useful to turn all files in a certain directory into resources, without specifying a resources for each file. For example, assume the `file_search_path/2` `icons` refers to the resource directory containing icon-files. The following definition makes all these images available as resources:

```
resource(Name, image, icons(XpmName)) :-
        atom(Name), !,
        file_name_extension(Name, xpm, XpmName).
resource(Name, image, XpmFile) :-
        var(Name),
        absolute_file_name(icons(.), [type(directory)], Dir)
        concat(Dir, '/*.xpm', Pattern),
        expand_file_name(Pattern, XpmFiles),
        member(XpmFile, XpmFiles).
```

open_resource(+*Name, ?Class,* -*Stream***)**

Opens the resource specified by *Name* and *Class*. If the latter is a variable, it will be unified to the class of the first resource found that has the specified *Name*. If successful, *Stream* becomes a handle to a binary input stream, providing access to the content of the resource.

The predicate `open_resource/3` first checks `resource/3`. When succesful it will open the returned resource source-file. Otherwise it will look in the programs resource database. When creating a saved-state, the system normally saves the resource contents into the resource archive, but does not save the resource clauses.

This way, the development environment uses the files (and modifications to the `resource/3` declarations and/or files containing resource info thus immediately affect the running environment, while the runtime system quickly accesses the system resources.

6.3.2 The `plrc` program

The utility program `plrc` can be used to examine and manipulate the contents of a SWI-Prolog resource file. The options are inspired by the Unix `ar` program. The basic command is:

```
% plrc option resource-file member ...
```

The options are described below.

l

List contents of the archive.

x

Extract named (or all) members of the archive into the current directory.

a

Add files to the archive. If the archive already contains a member with the same name, the contents is replaced. Anywhere in the sequence of members, the options `--class=`*class* and `--encoding=`*encoding* may appear. They affect the class and encoding of subsequent files. The initial class is `data` and encoding `none`.

d

Delete named members from the archive.

This command is also described in the `pl(1)` Unix manual page.

6.4 Finding Application files

If your application uses files that are not part of the saved program such as database files, configuration files, etc., the runtime version has to be able to locate these files. The `file_search_path/2` mechanism in combination with the `-p`alias command-line argument is the preferred way to locate runtime files. The first step is to define an alias for the toplevel directory of your application. We will call this directory `gnatdir` in our examples.

A good place for storing data associated with SWI-Prolog runtime systems is below the emulator's home-directory. `swi` is a predefined alias for this directory. The following is a useful default definition for the search path.

```
user:file_search_path(gnatdir, swi(gnat)).
```

The application should locate all files using absolute_file_name. Suppose gnatdir contains a file `config.pl` to define local configuration. Then use the code below to load this file:

```
configure_gnat :-
        (   absolute_file_name(gnatdir('config.pl'), ConfigFile)
            ->  consult(ConfigFile)
            ;   format(user_error, 'gnat: Cannot lo-
cate config.pl~n'),
            halt(1)
            ).
```

6.4.1 Passing a path to the application

Suppose the system administrator has installed the SWI-Prolog runtime environment in `/usr/local/lib/rt-pl-3.2.0`. A user wants to install `gnat`, but gnat will look for its configuration in `/usr/local/lib/rt-pl-3.2.0/gnat` where the user cannot write.

The user decides to install the gnat runtime files in `/users/bob/lib/gnat`. For one-time usage, the user may decide to start gnat using the command:

```
% gnat -p gnatdir=/users/bob/lib/gnat
```

6.5 The Runtime Environment

6.5.1 The Runtime Emulator

The sources may be used to built two versions of the emulator. By default, the *development emulator* is built. This emulator contains all features for interactive development of Prolog applications. If the system is configured using `--enable-runtime`, `make(1)` will create a *runtime version* of the emulator. This emulator is equivalent to the development version, except for the following features:

- *No input editing*
  The GNU library `-lreadline` that provides EMACS compatible editing of input lines will not be linked to the system.

- *No tracer*
  The tracer and all its options are removed, making the system a little faster too.

- *No profiler*
  `profile/3` and friends are not supported. This saves some space and provides better performance.

- *No interrupt*
  Keyboard interrupt (Control-C normally) is not rebound and will normally terminate the application.

- *current_prolog_flag(runtime, true) succeeds*
  This may be used to verify your application is running in the runtime environment rather than the development environment.

- `clause/[2,3]` *do not work on static predicates*
  This prolog-flag inhibits listing your program. It is only a very limited protection however.

The following fragment is an example for building the runtime environment in `\env{HOME}/lib/rt-pl-3.2.0`. If possible, the shared-library interface should be configured to ensure it can serve a large number of applications.

```
% cd pl-3.2.0
% mkdir runtime
% cd runtime
% ../src/configure --enable-runtime --prefix=$HOME
% make
% make rt-install
```

The runtime directory contains the components listed below. This directory may be tar'ed and shipped with your application.

`README.RT`	Info on the runtime environment
`bin/`⟨*arch*⟩`/pl`	The emulator itself
`man/pl.1`	Manual page for pl
`swipl`	pointer to the home directory (.)

`lib/`
`lib/`⟨*arch*⟩`/`
	directory for shared libraries
	machine-specific shared libraries

Hackers corner A

This appendix describes a number of predicates which enable the Prolog user to inspect the Prolog environment and manipulate (or even redefine) the debugger. They can be used as entry points for experiments with debugging tools for Prolog. The predicates described here should be handled with some care as it is easy to corrupt the consistency of the Prolog system by misusing them.

A.1 Examining the Environment Stack

prolog_current_frame(-*Frame***)**

Unify *Frame* with an integer providing a reference to the parent of the current local stack frame. A pointer to the current local frame cannot be provided as the predicate succeeds deterministically and therefore its frame is destroyed immediately after succeeding.

prolog_frame_attribute(+*Frame*, +*Key*, -*Value***)**

Obtain information about the local stack frame *Frame*. *Frame* is a frame reference as obtained through `prolog_current_frame/1`, `prolog_trace_interception/4` or this predicate. The key values are described below.

alternative

Value is unified with an integer reference to the local stack frame in which execution is resumed if the goal associated with *Frame* fails. Fails if the frame has no alternative frame.

has_alternatives

Value is unified with `true` if *Frame* still is a candidate for backtracking. `false` otherwise.

goal

Value is unified with the goal associated with *Frame*. If the definition module of the active predicate is not `user` the goal is represented as ⟨*module*⟩`:`⟨*goal*⟩. Do not instantiate variables in this goal unless you **know** what you are doing!

clause

Value is unified with a reference to the currently running clause. Fails if the current goal is associated with a foreign (C) defined predicate. See also `nth_clause/3` and `clause_property/2`.

level

Value is unified with the recursion level of *Frame*. The top level frame is at level '0'.

parent

Value is unified with an integer reference to the parent local stack frame of *Frame*. Fails if *Frame* is the top frame.

context_module
: *Value* is unified with the name of the context module of the environment.

top
: *Value* is unified with `true` if *Frame* is the top Prolog goal from a recursive call back from the foreign language. `false` otherwise.

hidden
: *Value* is unified with `true` if the frame is hidden from the user, either because a parent has the hide-childs attribute (all system predicates), or the system has no trace-me attribute.

pc
: *Value* is unified with the program-pointer saved on behalve of the parent-goal if the parent-goal is not owned by a foreign predicate.

argument(*N*)
: *Value* is unified with the *N*-th slot of the frame. Argument 1 is the first argument of the goal. Arguments above the arity refer to local variables. Fails silently if *N* is out of range.

A.2 Intercepting the Tracer

prolog_trace_interception(*+Port, +Frame, +PC, -Action***)**

Dynamic predicate, normally not defined. This predicate is called from the SWI-Prolog debugger just before it would show a port. If this predicate succeeds the debugger assumes the trace action has been taken care of and continues execution as described by *Action*. Otherwise the normal Prolog debugger actions are performed.

Port is one of `call`, `redo`, `exit`, `fail` or `unify`. *Frame* is an integer reference to the current local stack frame. *PC* is the current value of the program-counter, relative to the start of the current clause, or 0 if it is invalid, for example because the current frame runs a foreign predicate, or no clause has been selected yet. *Action* should be unified with one of the atoms `continue` (just continue execution), `retry` (retry the current goal) or `fail` (force the current goal to fail). Leaving it a variable is identical to `continue`.

Together with the predicates described in section 3.42 and the other predicates of this chapter this predicate enables the Prolog user to define a complete new debugger in Prolog. Besides this it enables the Prolog programmer monitor the execution of a program. The example below records all goals trapped by the tracer in the database.

```
prolog_trace_interception(Port, Frame, _PC, continue) :-
        prolog_frame_attribute(Frame, goal, Goal),
        prolog_frame_attribute(Frame, level, Level),
        recordz(trace, trace(Port, Level, Goal)).
```

To trace the execution of ‘go’ this way the following query should be given:

```
?- trace, go, notrace.
```

prolog_skip_level(-*Old*, +*New***)**
Unify *Old* with the old value of 'skip level' and than set this level according to *New*. New is an integer, or the special atom `very_deep` (meaning don't skip). The 'skip level' is a global variable of the Prolog system that disables the debugger on all recursion levels deeper than the level of the variable. Used to implement the trace options 'skip' (sets skip level to the level of the frame) and 'up' (sets skip level to the level of the parent frame (i.e. the level of this frame minus 1).

prolog_list_goal(*:Goal***)**
Hook, normally not defined. This hook is called by the 'L' command of the tracer in the module `user` to list the currently called predicate. This hook may be defined to list only relevant clauses of the indicated *Goal* and/or show the actual source-code in an editor. See also `portray/1` and `multifile/1`.

A.3 Hooks using the `exception/3` predicate

This section describes the predicate `exception/3`, which may be defined by the user in the module `user` as a multifile predicate. Unlike the name suggests, this is actually a *hook* predicate. Exceptions are handled by the ISO predicates `catch/3` and `throw/1`. They all frames created after the matching `catch/3` to be discarded immediately.

The predicate `exception/3` is called by the kernel on a couple of events, allowing the user to alter the behaviour on some predefined events.

exception(+*Exception*, +*Context*, -*Action***)**
Dynamic predicate, normally not defined. Called by the Prolog system on run-time exceptions. Currently `exception/3` is only used for trapping undefined predicates. Future versions might handle signal handling, floating exceptions and other runtime errors via this mechanism. The values for *Exception* are described below.

undefined_predicate
If *Exception* is `undefined_predicate` *Context* is instantiated to a term *Name*/*Arity*. *Name* refers to the name and *Arity* to the arity of the undefined predicate. If the definition module of the predicate is not *user*, *Context* will be of the form ⟨*Module*⟩`:`⟨*Name*⟩`/`⟨*Arity*⟩. If the predicate fails Prolog will generate an `esistence_error` exception. If the predicate succeeds it should instantiate the last argument either to the atom `fail` to tell Prolog to fail the predicate, the atom `retry` to tell Prolog to retry the predicate or `error` to make the system generate an exception. The action `retry` only makes sense if the exception handler has defined the predicate.

A.4 Readline Interaction

The following predicates are available if `current_prolog_flag(readline, true)` succeeds. They allow for direct interaction with the GNU readline library. See also `readline(3)`

rl_read_init_file(+*File***)**
Read a readline initialisation file. Readline by default reads `~/.inputrc`. This predicate may be used to read alternative readline initialisation files.

rl_add_history(+*Line***)**

Add a line to the Control-P/Control-N history system of the readline library.

Glossary of Terms B

anonymous [variable]
: The variable _ is called the *anonymous* variable. Multiple occurrences of _ in a single *term* are not *shared*.

arguments
: Arguments are *terms* that appear in a *compound term*. *A1* and *a2* are the first and second argument of the term `myterm`(*A1*, *a2*).

arity
: Argument count (is number of arguments) of a *compound term*.

assert
: Add a *clause* to a *predicate*. Clauses can be added at either end of the clause-list of a *predicate*. See `assert/1` and `assertz/1`.

atom
: Textual constant. Used as name for *compound* terms, to represent constants or text.

backtracking
: Searching process used by Prolog. If a predicate offers multiple *clauses* to solve a *goal*, they are tried one-by-one until one *succeeds*. If a subsequent part of the prove is not satisfied with the resulting *variable binding*, it may ask for an alternative *solution* (= *binding* of the *variables*), causing Prolog to reject the previously chosen *clause* and try the next one.

binding [of a variable]
: Current value of the *variable*. See also *backtracking* and *query*.

built-in [predicate]
: Predicate that is part of the Prolog system. Built in predicates cannot be redefined by the user, unless this is overruled using `redefine_system_predicate/1`.

body
: Part of a *clause* behind the *neck* operator (`:-`).

clause
: 'Sentence' of a Prolog program. A *clause* consists of a *head* and *body* separated by the *neck* operator (`:-`) or it is a *fact*. For example:

```
parent(X) :-
        father(X, _).
```

Expressed “X is a parent if X is a father of someone”. See also *variable* and *predicate*.

compile
: Process where a Prolog *program* is translated to a sequence of instructions. See also *interpreted*. SWI-Prolog always compiles your program before executing it.

compound [term]
: Also called *structure*. It consists of a name followed by *N arguments*, each of which are *terms*. *N* is called the *arity* of the term.

context module
: If a *term* is referring to a *predicate* in a *module*, the *context module* is used to find the target module. The context module of a *goal* is the module in which the *predicate* is defined, unless this *predicate* is *module transparent*, in which case the *context module* is inherited from the parent *goal*. See also `module_transparent/1`.

dynamic [predicate]
: A *dynamic* predicate is a predicate to which *clauses* may be *assert*ed and from which *clauses* may be *retract*ed while the program is running. See also *update view*.

exported [predicate]
: A *predicate* is said to be *exported* from a *module* if it appears in the *public list*. This implies that the predicate can be *imported* into another module to make it visible there. See also `use_module/[1,2]`.

fact
: *Clause* without a *body*. This is called a fact because interpreted as logic, there is no condition to be satisfied. The example below states `john` is a person.

```
person(john).
```

fail
: A *goal* is said to haved failed if it could not be *proven*.

float
: Computers cripled representation of a real number. Represented as ‘IEEE double’.

foreign
: Computer code expressed in other languages than Prolog. SWI-Prolog can only cooperate directly with the C and C++ computer languages.

functor
: Combination of name and *arity* of a *compound* term. The term `foo`(*a*, *b*, *c*) is said to be a term belonging to the functor `foo/3`. `foo/0` is used to refer to the *atom* `foo`.

goal
: Question stated to the Prolog engine. A *goal* is either an *atom* or a *compound* term. A *goal* succeeds, in which case the *variables* in the *compound* terms have a *binding* or *fails* if Prolog fails to prove the *goal*.

hashing
Indexing technique used for quick lookup.

head
Part of a *clause* before the *neck* instruction. This is an atom or *compound* term.

imported [predicate]
A *predicate* is said to be *imported* into a *module* if it is defined in another *module* and made available in this *module*. See also chapter 4.

indexing
Indexing is a technique used to quickly select candidate *clauses* of a *predicate* for a specific *goal*. In most Prolog systems, including SWI-Prolog, indexing is done on the first *argument* of the *head*. If this argument is instantiated to an *atom*, *integer*, *float* or *compound* term with *functor*, *hashing* is used quickly select all *clauses* of which the first argument may *unify* with the first argument of the *goal*.

integer
Whole number. On most current machines, SWI-Prolog integers are represented as '32-bit signed values', ranging from -2147483648 to 2147483647. See also `current_prolog_flag/2`.

interpreted
As opposed to *compiled*, interpreted means the Prolog system attempts to prove a *goal* by directly reading the *clauses* rather than executing instructions from an (abstract) instruction set that is not or only indirectly related to Prolog.

meta predicate
A *predicate* that reasons about other *predicates*, either by calling them, (re)defining them or querying *properties*.

module
Collection of predicates. Each module defines a name-space for predicates. *built-in* predicates are accessible from all modules. Predicates can be published (*exported*) and *imported* to make their definition available to other modules.

module transparent [predicate]
A *predicate* that does not change the *context module*. Sometimes also called a *meta predicate*.

multifile [predicate]
Predicate for which the definition is distributed over multiple source-files. See `multi_file/1`.

neck
Operator (`:-`) separating *head* from *body* in a *clause*.

operator
Symbol (*atom*) that may be placed before its *operant* (prefix), after its *operant* (postfix) or between its two *operants* (infix).

In Prolog, the expression `a+b` is exactly the same as the canonical term `+(a,b)`.

operant
: *Argument* of an *operator*.

precedence
: The *priority* of an *operator*. Operator precedence is used to interpret `a+b*c` as `+(a, *(b,c))`.

predicate
: Collection of *clauses* with the same *functor* (name/*arity*). If a *goal* is proved, the system looks for a *predicate* with the same functor, then used *indexing* to select candidate *clauses* and then tries these *clauses* one-by-one. See also *backtracking*.

priority
: In the context of *operators* a synonym for *precedence*.

program
: Collection of *predicates*.

property
: Attribute of an object. SWI-Prolog defines various **_property* predicates to query the status of predicates, clauses. etc.

prove
: Process where Prolog attempts to prove a *query* using the available *predicates*.

public list
: List of *predicates* exported from a *module*.

query
: See *goal*.

retract
: Remove a *clause* from a *predicate*. See also *dynamic*, *update view* and *assert*.

shared
: Two *variables* are called *shared* after they are *unified*. This implies if either of them is *bound*, the other is bound to the same value:

```
?- A = B, A = a.

A = a,
B = a
```

singleton [variable]
: *Variable* appearing only one time in a *clause*. SWI-Prolog normally warns for this to avoid you making spelling mistakes. If a variable appears on purpose only once in a clause, write it as _ (see *anonymous*) or make sure the first character is a _. See also the `style_check/1` option `singletons`.

solution
: *Bindings* resulting from a successfully *prove*n *goal*.

structure
: Synonym for *compound* term.

string
: Used for the following representations of text: a packed array (see section 3.23, SWI-Prolog specific), a list of character codes or a list of one-character *atoms*.

succeed
: A *goal* is said to have *succeeded* if it has been *proven*.

term
: Value in Prolog. A *term* is either a *variable*, *atom*, integer, float or *compound* term. In addition, SWI-Prolog also defines the type *string*

transparent
: See *module transparent*.

unify
: Prolog process to make two terms equal by assigning variables in one term to values at the corresponding location of the other term. For example:

```
?- foo(a, B) = foo(A, b).

A = a,
B = b
```

Unlike assignment (which does not exist in Prolog), unification is not directed.

update view
: How Prolog behaves when a *dynamic predicate* is changed while it is running. There are two models. In most older Prolog systems the change becomes immediately visible to the *goal*, in modern systems including SWI-Prolog, the running *goal* is not affected. Only new *goals* 'see' the new definition.

variable
: A Prolog variable is a value that 'is not yet bound'. After *binding* a variable, it cannot be modified. *Backtracking* to a point in the execution before the variable was bound will turn it back into a variable:

```
?- A = b, A = c.
No
?- (A = b; true; A = c).
A = b ;
A = _G283 ;
A = c ;
No
```

See also *unify*.

Summary C

C.1 Predicates

The predicate summary is used by the Prolog predicate `apropos/1` to suggest predicates from a keyword.

!/0	Cut (discard choicepoints)
!/1	Cut block. See `block/3`
,/2	Conjunction of goals
->/2	If-then-else
*->/2	Soft-cut
./2	Consult. Also list constructor
;/2	Disjunction of goals. Same as `|/2`
</2	Arithmetic smaller
=/2	Unification
=../2	“Univ.” Term to list conversion
=:=/2	Arithmetic equal
=</2	Arithmetic smaller or equal
==/2	Identical
=@=/2	Structural identical
=\=/2	Arithmetic not equal
>/2	Arithmetic larger
>=/2	Arithmetic larger or equal
@</2	Standard order smaller
@=</2	Standard order smaller or equal
@>/2	Standard order larger
@>=/2	Standard order larger or equal
\+/1	Negation by failure. Same as `not/1`
\=/2	Not unifyable
\==/2	Not identical
\=@=/2	Not structural identical
^/2	Existential quantification (`bagof/3`, `setof/3`)
|/2	Disjunction of goals. Same as `;/2`
abolish/1	Remove predicate definition from the database
abolish/2	Remove predicate definition from the database
abort/0	Abort execution, return to top level
absolute_file_name/2	Get absolute path name
absolute_file_name/3	Get absolute path name with options
access_file/2	Check access permissions of a file

append/1	Append to a file
append/3	Concatenate lists
apply/2	Call goal with additional arguments
apropos/1	library(`online_help`) Show related predicates and manual sections
arg/3	Access argument of a term
arithmetic_function/1	Register an evaluable function
assert/1	Add a clause to the database
assert/2	Add a clause to the database, give reference
asserta/1	Add a clause to the database (first)
asserta/2	Add a clause to the database (first)
assertz/1	Add a clause to the database (last)
assertz/2	Add a clause to the database (last)
attach_console/0	Attach I/O console to thread
at_end_of_stream/0	Test for end of file on input
at_end_of_stream/1	Test for end of file on stream
at_halt/1	Register goal to run at `halt/1`
at_initialization/1	Register goal to run at start-up
atom/1	Type check for an atom
atom_chars/2	Convert between atom and list of characters
atom_codes/2	Convert between atom and list of ASCII values
atom_length/2	Determine length of an atom
atom_prefix/2	Test for start of atom
atom_to_term/3	Convert between atom and term
atomic/1	Type check for primitive
autoload/0	Autoload all predicates now
bagof/3	Find all solutions to a goal
between/3	Integer range checking/generating
block/3	Start a block (‘catch’/‘throw’)
break/0	Start interactive toplevel
call/1	Call a goal
call/[2..]	Call with additional arguments
call_shared_object_function/2	UNIX: Call C-function in shared (.so) file
call_with_depth_limit/3	Prove goal with bounded depth
callable/1	Test for atom or compound term
catch/3	Call goal, watching for exceptions
char_code/2	Convert between atom and ASCII value
char_conversion/2	Provide mapping of input characters
char_type/2	Classify characters
character_count/2	Get character index on a stream
chdir/1	Change working directory
checklist/2	Invoke goal on all members of a list
clause/2	Get clauses of a predicate
clause/3	Get clauses of a predicate
clause_property/2	Get properties of a clause
close/1	Close stream
close/2	Close stream (forced)
close_dde_conversation/1	Win32: Close DDE channel

close_shared_object/1	UNIX: Close shared library (.so file)
compare/3	Compare, using a predicate to determine the order
compiling/0	Is this a compilation run?
compound/1	Test for compound term
atom_concat/3	Append two atoms
code_type/2	Classify a character-code
concat_atom/2	Append a list of atoms
concat_atom/3	Append a list of atoms with separator
consult/1	Read (compile) a Prolog source file
context_module/1	Get context module of current goal
convert_time/8	Break time stamp into fields
convert_time/2	Convert time stamp to string
copy_stream_data/2	Copy all data from stream to stream
copy_stream_data/3	Copy n bytes from stream to stream
copy_term/2	Make a copy of a term
current_arithmetic_function/1	Examine evaluable functions
current_atom/1	Examine existing atoms
current_char_conversion/2	Query input character mapping
current_flag/1	Examine existing flags
current_foreign_library/2	library(`shlib`) Examine loaded shared libraries (.so files)
current_format_predicate/2	Enumerate user-defined format codes
current_functor/2	Examine existing name/arity pairs
current_input/1	Get current input stream
current_key/1	Examine existing database keys
current_module/1	Examine existing modules
current_module/2	Examine existing modules
current_mutex/3	Examine existing mutexes
current_op/3	Examine current operator declarations
current_output/1	Get the current output stream
current_predicate/2	Examine existing predicates
current_signal/3	Current software signal mapping
current_stream/3	Examine open streams
current_thread/2	Examine Prolog threads
dde_current_connection/2	Win32: Examine open DDE connections
dde_current_service/2	Win32: Examine DDE services provided
dde_execute/2	Win32: Execute command on DDE server
dde_register_service/2	Win32: Become a DDE server
dde_request/3	Win32: Make a DDE request
dde_poke/3	Win32: POKE operation on DDE server
dde_unregister_service/1	Win32: Terminate a DDE service
debug/0	Test for debugging mode
debugging/0	Show debugger status
default_module/2	Get the default modules of a module
delete/3	Delete all matching members from a list
delete_directory/1	Remove a folder from the file system
delete_file/1	Remove a file from the file system
discontiguous/1	Indicate distributed definition of a predicate

dwim_match/2	Atoms match in "Do What I Mean" sense
dwim_match/3	Atoms match in "Do What I Mean" sense
dwim_predicate/2	Find predicate in "Do What I Mean" sense
dynamic/1	Indicate predicate definition may change
edit/1	Edit a file
ensure_loaded/1	Consult a file if that has not yet been done
erase/1	Erase a database record or clause
exception/3	(hook) Handle runtime exceptions
exists_directory/1	Check existence of directory
exists_file/1	Check existence of file
exit/2	Exit from named block. See `block/3`
expand_answer/2	Expand answer of query
expand_file_name/2	Wildcard expansion of file names
expand_file_search_path/2	Wildcard expansion of file paths
expand_goal/2	Compiler: expand goal in clause-body
expand_query/4	Expanded entered query
expand_term/2	Compiler: expand read term into clause(s)
explain/1	library(`explain`) Explain argument
explain/2	library(`explain`) 2nd argument is explanation of first
export/1	Export a predicate from a module
export_list/2	List of public predicates of a module
fail/0	Always false
fail/1	Immediately fail named block. See `block/3`
current_prolog_flag/2	Get system configuration parameters
file_base_name/2	Get file part of path
file_directory_name/2	Get directory part of path
file_name_extension/3	Add, remove or test file extensions
file_search_path/2	Define path-aliases for locating files
fileerrors/2	Do/Don't warn on file errors
findall/3	Find all solutions to a goal
flag/3	Simple global variable system
flatten/2	Transform nested list into flat list
float/1	Type check for a floating point number
flush_output/0	Output pending characters on current stream
flush_output/1	Output pending characters on specified stream
forall/2	Prove goal for all solutions of another goal
format/1	Formatted output
format/2	Formatted output with arguments
format/3	Formatted output on a stream
format_predicate/2	Program `format/[1,2]`
free_variables/2	Find unbound variables in a term
functor/3	Get name and arity of a term or construct a term
garbage_collect/0	Invoke the garbage collector
garbage_collect_atoms/0	Invoke the atom garbage collector
gensym/2	Generate unique atoms from a base
get/1	Read first non-blank character
get/2	Read first non-blank character from a stream

get0/1	Read next character
get0/2	Read next character from a stream
get_byte/1	Read next byte (ISO)
get_byte/2	Read next byte from a stream (ISO)
get_char/1	Read next character as an atom (ISO)
get_char/2	Read next character from a stream (ISO)
get_code/1	Read next character (ISO)
get_code/2	Read next character from a stream (ISO)
get_single_char/1	Read next character from the terminal
get_time/1	Get current time
getenv/2	Get shell environment variable
goal_expansion/2	Hook for macro-expanding goals
ground/1	Verify term holds no unbound variables
guitracer/0	Install hooks for the graphical debugger
halt/0	Exit from Prolog
halt/1	Exit from Prolog with status
hash_term/2	Hash-value of ground term
help/0	Give help on help
help/1	Give help on predicates and show parts of manual
ignore/1	Call the argument, but always succeed
import/1	Import a predicate from a module
include/1	Include a file with declarations
index/1	Change clause indexing
initialization/1	Initialization directive
int_to_atom/2	Convert from integer to atom
int_to_atom/3	Convert from integer to atom (non-decimal)
integer/1	Type check for integer
interactor/0	Start new thread with console and toplevel
intersection/3	Set intersection
is/2	Evaluate arithmetic expression
is_absolute_file_name/1	True if arg defines an absolute path
is_list/1	Type check for a list
is_set/1	Type check for a set
keysort/2	Sort, using a key
last/2	Last element of a list
leash/1	Change ports visited by the tracer
length/2	Length of a list
library_directory/1	(hook) Directories holding Prolog libraries
limit_stack/2	Limit stack expansion
line_count/2	Line number on stream
line_position/2	Character position in line on stream
list_to_set/2	Remove duplicates
listing/0	List program in current module
listing/1	List predicate
load_files/2	Load source files with options
load_foreign_library/1	library(`shlib`) Load shared library (.so file)
load_foreign_library/2	library(`shlib`) Load shared library (.so file)

make/0	Reconsult all changed source files
make_directory/1	Create a folder on the file system
make_fat_filemap/1	Win32: Create file containing non-FAT filenames
make_library_index/1	Create autoload file INDEX.pl
maplist/3	Transform all elements of a list
member/2	Element is member of a list
memberchk/2	Deterministic `member/2`
merge/3	Merge two sorted lists
merge_set/3	Merge two sorted sets
message_hook/3	Intercept `print_message/2`
message_to_string/2	Translate message-term to string
meta_predicate/1	Quintus compatibility
module/1	Query/set current type-in module
module/2	Declare a module
module_transparent/1	Indicate module based meta predicate
msort/2	Sort, do not remove duplicates
multifile/1	Indicate distributed definition of predicate
mutex_create/1	Create a thread-synchronisation device
mutex_destroy/1	Destroy a mutex
mutex_lock/1	Become owner of a mutex
mutex_trylock/1	Become owner of a mutex (non-blocking)
mutex_unlock/1	Release ownership of mutex
mutex_unlock_all/0	Release ownership of all mutexes
name/2	Convert between atom and list of ASCII characters
nl/0	Generate a newline
nl/1	Generate a newline on a stream
nodebug/0	Disable debugging
noguitracer/0	Disable the graphical debugger
nonvar/1	Type check for bound term
noprotocol/0	Disable logging of user interaction
nospy/1	Remove spy point
nospyall/0	Remove all spy points
not/1	Negation by failure (argument not provable). Same as `\+/1`
notrace/0	Stop tracing
notrace/1	Do not debug argument goal
nth0/3	N-th element of a list (0-based)
nth1/3	N-th element of a list (1-based)
nth_clause/3	N-th clause of a predicate
number/1	Type check for integer or float
number_chars/2	Convert between number and one-char atoms
number_codes/2	Convert between number and ASCII values
numbervars/4	Enumerate unbound variables of a term using a given base
on_signal/3	Handle a software signal
once/1	Call a goal deterministically
op/3	Declare an operator
open/3	Open a file (creating a stream)
open/4	Open a file (creating a stream)

open_dde_conversation/3	Win32: Open DDE channel
open_null_stream/1	Open a stream to discard output
open_resource/3	Open a program resource as a stream
open_shared_object/2	UNIX: Open shared library (.so file)
open_shared_object/3	UNIX: Open shared library (.so file)
peek_byte/1	Read byte without removing
peek_byte/2	Read byte without removing
peek_char/1	Read character without removing
peek_char/2	Read character without removing
peek_code/1	Read character-code without removing
peek_code/2	Read character-code without removing
phrase/2	Activate grammar-rule set
phrase/3	Activate grammar-rule set (returning rest)
please/3	Query/change environment parameters
plus/3	Logical integer addition
portray/1	(hook) Modify behaviour of `print/1`
portray_clause/1	Pretty print a clause
predicate_property/2	Query predicate attributes
predsort/3	Sort, using a predicate to determine the order
preprocessor/2	Install a preprocessor before the compiler
print/1	Print a term
print/2	Print a term on a stream
print_message/2	Print message from (exception) term
print_message_lines/3	Print message to stream
profile/3	Obtain execution statistics
profile_count/3	Obtain profile results on a predicate
profiler/2	Obtain/change status of the profiler
prolog/0	Run interactive toplevel
prolog_current_frame/1	Reference to goal's environment stack
prolog_edit:locate/2	Locate targets for `edit/1`
prolog_edit:locate/3	Locate targets for `edit/1`
prolog_edit:edit_source/1	Call editor for `edit/1`
prolog_edit:edit_command/2	Specify editor activation
prolog_edit:load/0	Load `edit/1` extensions
prolog_file_type/2	Define meaning of file extension
prolog_frame_attribute/3	Obtain information on a goal environment
prolog_list_goal/1	Hook. Intercept tracer 'L' command
prolog_load_context/2	Context information for directives
prolog_skip_level/2	Indicate deepest recursion to trace
prolog_to_os_filename/2	Convert between Prolog and OS filenames
prolog_trace_interception/4	library(`user`) Intercept the Prolog tracer
prompt1/1	Change prompt for 1 line
prompt/2	Change the prompt used by `read/1`
proper_list/1	Type check for list
protocol/1	Make a log of the user interaction
protocola/1	Append log of the user interaction to file
protocolling/1	On what file is user interaction logged

put/1	Write a character
put/2	Write a character on a stream
put_byte/1	Write a byte
put_byte/2	Write a byte on a stream
put_char/1	Write a character
put_char/2	Write a character on a stream
put_code/1	Write a character-code
put_code/2	Write a character-code on a stream
qcompile/1	Compile source to Quick Load File
qsave_program/1	Create runtime application
qsave_program/2	Create runtime application
read/1	Read Prolog term
read/2	Read Prolog term from stream
read_clause/1	Read clause
read_clause/2	Read clause from stream
read_history/6	Read using history substitution
read_link/3	Read a symbolic link
read_term/2	Read term with options
read_term/3	Read term with options from stream
recorda/2	Record term in the database (first)
recorda/3	Record term in the database (first)
recorded/2	Obtain term from the database
recorded/3	Obtain term from the database
recordz/2	Record term in the database (last)
recordz/3	Record term in the database (last)
redefine_system_predicate/1	Abolish system definition
rename_file/2	Change name of file
repeat/0	Succeed, leaving infinite backtrack points
require/1	This file requires these predicates
reset_profiler/0	Clear statistics obtained by the profiler
resource/3	Declare a program resource
retract/1	Remove clause from the database
retractall/1	Remove unifying clauses from the database
reverse/2	Inverse the order of the elements in a list
same_file/2	Succeeds if arguments refer to same file
see/1	Change the current input stream
seeing/1	Query the current input stream
seek/4	Modify the current position in a stream
seen/0	Close the current input stream
select/3	Select element of a list
set_input/1	Set current input stream from a stream
set_output/1	Set current output stream from a stream
set_prolog_flag/2	Define a system feature
set_stream/2	Set stream attribute
set_stream_position/2	Seek stream to position
set_tty/2	Set ‘tty’ stream
setarg/3	Destructive assignment on term

setenv/2	Set shell environment variable
setof/3	Find all unique solutions to a goal
sformat/2	Format on a string
sformat/3	Format on a string
shell/0	Execute interactive subshell
shell/1	Execute OS command
shell/2	Execute OS command
show_profile/1	Show results of the profiler
size_file/2	Get size of a file in characters
skip/1	Skip to character in current input
skip/2	Skip to character on stream
rl_add_history/1	Add line to readline(3) history
rl_read_init_file/1	Read readline(3) init file
sleep/1	Suspend execution for specified time
sort/2	Sort elements in a list
source_file/1	Examine currently loaded source files
source_file/2	Obtain source file of predicate
source_location/2	Location of last read term
spy/1	Force tracer on specified predicate
stack_parameter/4	Some systems: Query/Set runtime stack parameter
statistics/0	Show execution statistics
statistics/2	Obtain collected statistics
stream_property/2	Get stream properties
string/1	Type check for string
string_concat/3	`atom_concat/3` for strings
string_length/2	Determine length of a string
string_to_atom/2	Conversion between string and atom
string_to_list/2	Conversion between string and list of ASCII
style_check/1	Change level of warnings
sub_atom/5	Take a substring from an atom
sublist/3	Determine elements that meet condition
subset/2	Check subset relation for unordered sets
sub_string/5	Take a substring from a string
subtract/3	Delete elements that do not meet condition
succ/2	Logical integer successor relation
swritef/2	Formatted write on a string
swritef/3	Formatted write on a string
tab/1	Output number of spaces
tab/2	Output number of spaces on a stream
tell/1	Change current output stream
telling/1	Query current output stream
term_expansion/2	(hook) Convert term before compilation
term_to_atom/2	Convert between term and atom
thread_at_exit/1	Register goal to be called at exit
thread_create/3	Create a new Prolog task
thread_exit/1	Terminate Prolog task with value
thread_get_message/1	Wait for message

thread_join/2	Wait for Prolog task-completion
thread_peek_message/1	Test for message in queue
thread_self/1	Get identifier of current thread
thread_send_message/2	Send message to another thread
thread_signal/2	Execute goal in another thread
threads/0	List running threads
throw/1	Raise an exception (see `catch/3`)
time/1	Determine time needed to execute goal
time_file/2	Get last modification time of file
tmp_file/2	Create a temporary filename
told/0	Close current output
trace/0	Start the tracer
trace/1	Set trace-point on predicate
trace/2	Set/Clear trace-point on ports
tracing/0	Query status of the tracer
trim_stacks/0	Release unused memory resources
true/0	Succeed
tty_get_capability/3	Get terminal parameter
tty_goto/2	Goto position on screen
tty_put/2	Write control string to terminal
ttyflush/0	Flush output on terminal
union/3	Union of two sets
unify_with_occurs_check/2	Logically sound unification
unix/1	OS interaction
unknown/2	Trap undefined predicates
unload_foreign_library/1	library(`shlib`) Detach shared library (.so file)
unsetenv/1	Delete shell environment variable
use_module/1	Import a module
use_module/2	Import predicates from a module
var/1	Type check for unbound variable
visible/1	Ports that are visible in the tracer
volatile/1	Predicates that are not saved
wait_for_input/3	Wait for input with optional timeout
wildcard_match/2	Csh(1) style wildcard match
win_exec/2	Win32: spawn Windows task
win_shell/2	Win32: open document through Shell
win_registry_get_value/3	Win32: get registry value
with_mutex/2	Run goal while holding mutex
write/1	Write term
write/2	Write term to stream
write_ln/1	Write term, followed by a newline
write_canonical/1	Write a term with quotes, ignore operators
write_canonical/2	Write a term with quotes, ignore operators on a stream
write_term/2	Write term with options
write_term/3	Write term with options to stream
writef/1	Formatted write
writef/2	Formatted write on stream

writeq/1	Write term, insert quotes
writeq/2	Write term, insert quotes on stream

C.2 Arithmetic Functions

*/2	Multiplication
**/2	Power function
+/2	Addition
-/1	Unary minus
-/2	Subtraction
//2	Division
///2	Integer division
/\/2	Bitwise and
<</2	Bitwise left shift
>>/2	Bitwise right shift
./2	List of one character: character code
\/1	Bitwise negation
\//2	Bitwise or
ˆ/2	Power function
abs/1	Absolute value
acos/1	Inverse (arc) cosine
asin/1	Inverse (arc) sine
atan/1	Inverse (arc) tangent
atan/2	Rectangular to polar conversion
ceil/1	Smallest integer larger than arg
ceiling/1	Smallest integer larger than arg
cos/1	Cosine
cputime/0	Get CPU time
e/0	Mathematical constant
exp/1	Exponent (base e)
float/1	Explicitly convert to float
float_fractional_part/1	Fractional part of a float
float_integer_part/1	Integer part of a float
floor/1	Largest integer below argument
integer/1	Round to nearest integer
log/1	Natural logarithm
log10/1	10 base logarithm
max/2	Maximum of two numbers
min/2	Minimum of two numbers
mod/2	Remainder of division
random/1	Generate random number
rem/2	Remainder of division
round/1	Round to nearest integer
truncate/1	Truncate float to integer
pi/0	Mathematical constant
sign/1	Extract sign of value
sin/1	Sine
sqrt/1	Square root
tan/1	Tangent

xor/2	Bitwise exclusive or

C.3 Operators

`$`	1	fx	Bind toplevel variable
`^`	200	xfy	Predicate
`^`	200	xfy	Arithmetic function
mod	300	xfx	Arithmetic function
`*`	400	yfx	Arithmetic function
`/`	400	yfx	Arithmetic function
`//`	400	yfx	Arithmetic function
`<<`	400	yfx	Arithmetic function
`>>`	400	yfx	Arithmetic function
xor	400	yfx	Arithmetic function
`+`	500	fx	Arithmetic function
`-`	500	fx	Arithmetic function
`?`	500	fx	XPCE: obtainer
`\`	500	fx	Arithmetic function
`+`	500	yfx	Arithmetic function
`-`	500	yfx	Arithmetic function
`/\`	500	yfx	Arithmetic function
`\/`	500	yfx	Arithmetic function
`:`	600	xfy	module:term separator
`<`	700	xfx	Predicate
`=`	700	xfx	Predicate
`=..`	700	xfx	Predicate
`=:=`	700	xfx	Predicate
`<`	700	xfx	Predicate
`==`	700	xfx	Predicate
`=@=`	700	xfx	Predicate
`=\=`	700	xfx	Predicate
`>`	700	xfx	Predicate
`>=`	700	xfx	Predicate
`@<`	700	xfx	Predicate
`@=<`	700	xfx	Predicate
`@>`	700	xfx	Predicate
`@>=`	700	xfx	Predicate
is	700	xfx	Predicate
`\=`	700	xfx	Predicate
`\==`	700	xfx	Predicate
`=@=`	700	xfx	Predicate
not	900	fy	Predicate
`\+`	900	fy	Predicate
`,`	1000	xfy	Predicate
`->`	1050	xfy	Predicate
`*->`	1050	xfy	Predicate
`;`	1100	xfy	Predicate
`|`	1100	xfy	Predicate

discontiguous	1150	fx	Predicate
dynamic	1150	fx	Predicate
module_transparent	1150	fx	Predicate
multifile	1150	fx	Predicate
volatile	1150	fx	Predicate
initialization	1150	fx	Predicate
`:-`	1200	fx	Introduces a directive
`?-`	1200	fx	Introduces a directive
`-->`	1200	xfx	DCGrammar: rewrite
`:-`	1200	xfx	head `:-` body. separator

Bibliography

[Anjewierden & Wielemaker, 1989] A. Anjewierden and J. Wielemaker. Extensible objects. ESPRIT Project 1098 Technical Report UvA-C1-TR-006a, University of Amsterdam, March 1989.

[BIM, 1989] *BIM Prolog release 2.4*. Everberg, Belgium, 1989.

[Bowen & Byrd, 1983] D. L. Bowen and L. M. Byrd. A portable Prolog compiler. In L. M. Pereira, editor, *Proceedings of the Login Programming Workshop 1983*, Lisabon, Portugal, 1983. Universidade nova de Lisboa.

[Bratko, 1986] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1986.

[Clocksin & Melish, 1987] W. F. Clocksin and C. S. Melish. *Programming in Prolog*. Springer-Verlag, New York, Third, Revised and Extended edition, 1987.

[Deransart *et al.*, 1996] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.

[Kernighan & Ritchie, 1978] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[OKeefe, 1990] R. A. OKeefe. *The Craft of Prolog*. MIT Press, Massachussetts, 1990.

[Pereira, 1986] F. Pereira. *C-Prolog User's Manual*, 1986.

[Qui, 1997] *Quintus Prolog, User Guide and Reference Manual*. Berkhamsted, UK, 1997.

[Sterling & Shapiro, 1986] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 1986.

[Warren, 1983] D. H. D. Warren. The runtime environment for a prolog compiler using a copy algorithm. Technical Report 83/052, SUNY and Stone Brook, New York, 1983. Major revision March 1984.

Index