University of Amsterdam

Human-Computer Studies (HCS, formerly SWI)
Kruislaan 419, 1098 VA Amsterdam
The Netherlands
Tel. (+31) 20 8884671

SWI-Prolog 5.6

Reference Manual

Updated for version 5.6.19, September 2006

Jan Wielemaker
wielemak@science.uva.nl
http://www.swi-prolog.org

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 started back in 1986 with the requirement for a Prolog that could handle recursive interaction with the C-language: Prolog calling C and C calling Prolog recursively. Those days Prolog systems were very aware of its environment and we needed such a system to support interactive applications. Since then, SWI-Prolog's development has been guided by requests from the user community, especially focussing on (in arbitrary order) interaction with the environment, scalability, (I/O) performance, standard compliance, teaching and the program development environment.

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 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 [O'Keefe, 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 [Hodgson, 1998].

[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 5.6 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, complex interactive systems, web-server and web-server components. 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). Some 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 9). It can also be *embedded* embedded in external programs (see section 9.7). 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, Win32 (Windows 95/98/ME and NT/2000/XP) and MacOS X (darwin).

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_super(Self, 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),
            current_output(Old),
            pce_open(V, write, Fd),
            set_output(Fd),
            listing(Term),
            close(Fd),
            set_output(Old)
        ;   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/ME, Windows NT/2000/XP and MacOS X (using X11). In the past, versions for Quintus- and SICStus Prolog as well as some Lisp dialects have existed. After discontinuing active Lisp development at SWI the Lisp versions have died. Active development on the Quintus and SICStus versions has been stopped due to lack of standardisation in the the Prolog community. If adequate standards emerge we are happy to actively support other Prolog implementations.

Info. further information is available from `http://www.swi-prolog.org/packages/xpce/` or by E-mail to `info@www.swi-prolog.org`.

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.top-level

- *'Answer' reuse*
  The top-level maintains a table of bindings returned by top-level 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 `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 described 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 4.13.1 for details.

SWI-Prolog 3.3 includes garbage collection on atoms, removing the last serious memory leak especially in text-manipulation applications. See section 9.6.2. 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 8.

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 source-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 `backcomp` provides definitions for these predicates, so no source **must** 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 `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 `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 explicitly 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 front-end.

- *Register PceEmacs for editing files*
  From your initialisation file. you can load `emacs/swi_prolog` that cause `edit/1` to use the built-in PceEmacs editor.

1.6.14 Version 5.0 Release Notes

Version 5.0 marks a breakpoint in the philosophy, where SWI-Prolog moves from a dual GPL/proprietary to a uniform LGPL (Lesser GNU Public Licence) schema, providing a widely usable Free Source Prolog implementation.

On the technical site the development environment, consisting of source-level debugger, integrated editor and various analysis and navigation tools progress steadily towards a mature set of tools.

Many portability issues have been improved, including a port to MacOS X (Darwin).

For details, please visit the new website at `http://www.swi-prolog.org`

1.6.15 Version 5.1 Release Notes

Version 5.1 is a beta-serie introducing portable multi-threading. See chapter 8. In addition it introduces many new facilities to support server applications, such as the new `rlimit` library to limit system resources and the possibility to set timeouts on input streams.

1.6.16 Version 5.2 Release Notes

Version 5.2 consolidates the 5.1.x beta series that introduced threading and many related modifications to the kernel.

1.6.17 Version 5.3 Release Notes

Version 5.3.x is a development series for adding coroutining, constraints, global variables, cyclic terms (infinite trees) and other goodies to the kernel. The package JPL, providing a bidirectional Java/Prolog interface is added to the common source-tree and common binary packages.

1.6.18 Version 5.4 Release Notes

Version 5.4 consolidates the 5.3.x beta series.

1.6.19 Version 5.5 Release Notes

Version 5.5.x provides support for *wide characters* with UTF-8 and UNICODE I/O (section 2.17.1). On both 32 and 64-bit hardware Prolog integers are now at minimum 64-bit integers. If available, SWI-Prolog arithmetic uses the GNU GMP library to provided *unbounded* integer arithmetic as well as rational arithmetic. Adding GMP support is sponsored by Scientific Software and Systems Limited, `www.sss.co.nz`. This version also incorporates clp(r) by Christian Holzbaur, brought to SWI-Prolog by Tom Schrijvers and Leslie De Koninck (section A.14).

1.6.20 Version 5.6 Release Notes

Version 5.6 consolidates the 5.5.x beta series.

1.7 Donate to the SWI-Prolog project

If you are happy with SWI-Prolog, you care it to be around for much longer while it becomes faster, more stable and with more features you should consider to donate to the SWI-Prolog foundation. Please visit the page below.

```
http://www.swi-prolog.org/donate.html
```

1.8 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 some of 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!

Bart Demoen and Tom Schrijvers have helped me adding coroutining, constraints, global variables and support for cyclic terms to the kernel. Tom has provided the integer interval constraint solver, the CHR compiler and some of the coroutining predicates.

Paul Singleton has integrated Fred Dushin's Java-calls-Prolog side with his Prolog-calls-Java side into the current bidirectional JPL interface package.

Richard O'Keefe is gratefully acknowledged for his efforts to educate beginners as well as valuable comments on proposed new developments.

Scientific Software and Systems Limited, `www.sss.co.nz` has sponsored the development if the SSL library as well as unbounded integer and rational number arithmetic.

Leslie de Koninck has made clp(QR) available to SWI-Prolog.

Markus Triska has contributed to various libraries.

Paulo Moura's great experience in maintaining Logtalk for many Prolog systems including SWI-Prolog has helped in many places fixing compatibility issues. He also worked on the MacOS port and fixed many typos in the 5.6.9 release of the documentation.

2 Overview

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
Welcome to SWI-Prolog (Version 5.6.0)
Copyright (c) 1990-2005 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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 unite, so if you are using Unix please continue at section 2.1.2.

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 what food 'sam' likes. The system responds with `X =` ⟨*value*⟩ if it can prove the goal for a certain *X*. The user can type the semi-colon (;)[1] 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(sam, X).

X = dahl ;

X = tandoori ;

...

X = chips ;

No
?-
```

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%

[1] On most installations, single-character commands are executed without waiting for the RETURN key.

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 command-line 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` or `-s file` to make Prolog load a file, `-g goal` to define an initialisation goal and `-t goal` to define the *top-level goal*. The following is a typical example for starting an application directly from the command-line.

```
machine% pl -s 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 top-level— exit after completing `go/0`. The `-q` may be used to suppress all informational messages.

In MS-Windows, the same can be achieved using a short-cut with appropriately defined command-line 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
```

Section 2.10.2 discusses further scripting options and chapter 10 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 9.7). Below is a typical example of using this feature.

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

-q
Set the prolog-flag `verbose` to `silent`, suppressing informational and banner messages.

-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 2.18.

-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` flag `optimise` for details.

-nodebug
Disable debugging. See the `current_prolog_flag/2` flag `generate_debug_info` for details.

-s *file*
Use *file* as a script-file. The script file is loaded after the initialisation file specified with the `-f file` option. Unlike `-f file`, using `-s` does not stop Prolog from loading the personal initialisation file.

-f *file*

Use *file* as initialisation file instead of the default `.plrc` (Unix) or `pl.ini` (Windows). '`-f none`' stops SWI-Prolog from searching for a startup file. This option can be used as an alternative to `-s file` that stops Prolog from loading the personal initialisation file. See also 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 top-level instead of the default goal `prolog/0`. *goal* can be a complex term. If the top-level 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 top-level.

-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.

-nosignals

Inhibit any signal handling by Prolog, a property that is sometimes desirable for embedded applications. This option sets the flag `signals` to `false`. See section 9.6.20 for details.

-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 list of values for the alias. On Windows the list-separator is `;`. On other systems it is `:`. 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 command-line 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 `'MANUAL'`. The file `helpidx` provides an index into this file. `'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 `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-prolog.org`, 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.

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 Command-line history

SWI-Prolog offers a query substitution mechanism 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. The available history commands are shown in table 2.1. History expansion is not done if these sequences appear in quoted atoms or strings.

`!!.`	Repeat last query
`!nr.`	Repeat query numbered ⟨*nr*⟩
`!str.`	Repeat last query starting with ⟨*str*⟩
`h.`	Show history of commands
`!h.`	Show this list

Table 2.1: History commands

```
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 top-level bindings

2.8 Reuse of top-level bindings

Bindings resulting from the successful execution of a top-level goal are asserted in a database. These values may be reused in further top-level 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 ?-
```

```
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

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 4.9.

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 top-level 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 5). 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(10)]`. 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 4.38.

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:

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

```
% pl
<banner>

1 ?- [load].
<compilation messages>

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 top-level.

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).

Using PrologScript

New in version 4.0.5 is the possibility to use a Prolog source file directly as a Unix script-file. the same mechanism is useful to specify additional parameters for running a Prolog file on Windows.

If the first letter of a Prolog file is `#`, the first line is treated as comment.[3] To create a Prolog script, make the first line start like this:

`#!/path/to/pl` ⟨*options*⟩ `-s`

Prolog recognises this starting sequence and causes the interpreter to receive the following argument-list:

`/path/to/pl` ⟨*options*⟩ `-s` ⟨*script*⟩ `--` ⟨*ScriptArguments*⟩

Instead of `-s`, the user may use `-f` to stop Prolog from looking for a personal initialisation file.

Here is a simple script doing expression evaluation:

```
#!/usr/bin/pl -q -t main -f

eval :-
        current_prolog_flag(argv, Argv),
        append(_, [--|Args], Argv),
        concat_atom(Args, ' ', SingleArg),
        term_to_atom(Term, SingleArg),
        Val is Term,
        format('~w~n', [Val]).
```

[3] The #-sign can be the legal start of a normal Prolog clause. In the unlikely case this is required, leave the first line blank or add a header-comment.

```
main :-
        catch(eval, E, (print_message(error, E), fail)),
        halt.
main :-
        halt(1).
```

And here are two example runs:

```
% eval 1+2
3
% eval foo
ERROR: Arithmetic: 'foo/0' is not a function
%
```

The Windows version supports the `#!` construct too, but here it serves a rather different role. The Windows shell already allows the user to start Prolog source files directly through the Windows file-type association. Windows however makes it rather complicated to provide additional parameters, such as the required stack-size for an individual Prolog file. The `#!` line provides for this, providing a more flexible approach then changing the global defaults. The following starts Prolog with unlimited stack-size on the given source file:

```
#!/usr/bin/pl -L0 -T0 -G0 -s

....
```

Note the use of `/usr/bin/pl`, which specifies the interpreter. This argument is ignored in the Windows version, but required to ensure best cross-platform compatibility.

Creating a shell-script

With the introduction of *PrologScript* (see section 2.10.2), using shell-scripts as explained in this section has become redundant for most applications.

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 to 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 10.

Compilation using the -c command-line 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 stand-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. Flag values are typed. Flags marked as `bool` can have the values `true` and `false`. Some prolog flags are not defined in all versions, which is normally indicated in the documentation below as *"if present and true"*. A boolean prolog-flag is true iff the prolog-flag is present **and** the *Value* is the atom `true`. Tests for such flags should be written as below.

```
(   current_prolog_flag(windows, true)
->  <Do MS-Windows things>
;   <Do normal things>
)
```

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

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().

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 to non-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.

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.

arch *(atom)*
Identifier for the hardware and operating system SWI-Prolog is running on. Used to select foreign files for the right architecture. See also section 9.4 and `file_search_path/2`.

associate *(atom, changeable)*
On Windows systems, this is set to the filename-extension (e.g. `pl` or `pro` associated with `plwin.exe`.

autoload *(bool, changeable)*
If `true` (default) autoloading of library functions is enabled. Note that autoloading only works if the flag `unknown` is *not* set to `fail`. See section 2.13.

backquoted_string *(bool, changeable)*
If `true` (default `false`), read translates text between backquotes into a string object (see section 4.23). This flag is mainly for compatibility to LPA Prolog.

bounded *(bool)*
ISO prolog-flag. If `true`, integer representation is bound by `min_integer` and `max_integer`. If `false` integers can be arbitrary large and the `min_integer` and `max_integer` are not present. See section 4.26.2.

c_cc *(atom)*
Name of the C-compiler used to compile SWI-Prolog. Normally either gcc or cc. See section 9.7.

c_ldflags *(atom)*
Special linker flags passed to link SWI-Prolog. See section 9.7.

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 9.7.

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

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.

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.

console_menu *(bool)*
Set to `true` in `plwin.exe` to indicate the console supports menus. See also section 4.34.2.

dde *(bool)*
Set to `true` if this instance of Prolog supports DDE as described in section 4.42.

debug *(bool, changeable)*
Switch debugging mode on/off. 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.

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.

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), attributes(portray)]`.

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.

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 4.23). See also `atom_chars/2` and `atom_codes/2`.

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 consequences 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. On most systems using sparse-memory management 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).

editor *(atom, changeable)*
Determines the editor used by `edit/1`. See section 4.4 for details on selecting the editor used.

emacs_inferior_process *(bool)*
If true, SWI-Prolog is running as an *inferior process* of (GNU/X-)Emacs. SWI-Prolog assumes this is the case if the environment variable `EMACS` is `t` and `INFERIOR` is `yes`.

encoding *(atom, changeable)*
Default encoding used for opening files in `text` mode. The initial value is deduced from the environment. See section 2.17.1 for details.

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

file_name_variables *(bool, changeable)*
If `true` (default `false`), expand `$`*`varname`* and `~` in arguments of built-in 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.

float_format *(atom, changeable)*
C-library 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.

When using quoted-write, the output is guaranteed to contain a decimal dot or exponent, so `read/1` reads a floating point number. See also `format/[1,2]`, `write_term/[2,3]`.

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.

generate_debug_info *(bool, changeable)*
If `true` (default) generate code that can be debugged using `trace/0`, `spy/1`, etc. Can be set to `false` using the `-nodebug`. The predicate `load_files/2` restores the value of this flag after loading a file, causing modifications to be local to a source file. Many of the libraries have `:- set_prolog_flag(generate_debug_info, false)` to hide their details from a normal trace.[4]

gmp_version *(integer)*
If Prolog is linked with GMP, this flag gives the major version of the GMP library used. See also section 9.6.7.

gnu_libpthread_version *(atom)*
Linux systems only. Reports the version of the Linux thread library used. See section 8.2.1 for how it may affect you.

gui *(bool)*
Set to `true` if XPCE is around and can be used for graphics.

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 command-line editor. The default is to set this prolog-flag to 0 if a command-line editor is provided (see prolog-flag `readline`) and 15 otherwise.

home *(atom)*
SWI-Prolog's notion of the home-directory. SWI-Prolog uses its home directory to find its startup file as ⟨*home*⟩`/boot32.prc` (32-bit machines) or ⟨*home*⟩`/boot64.prc` (64-bit machines) and to find its library as ⟨*home*⟩`/library`.

hwnd *(integer)*
In `plwin.exe`, this refers to the MS-Windows window-handle of the console window.

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

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

- The `//2` (float division) *always* return a float, even if applied to integers that can be divided.

[4] In the current implementation this only causes a flag to be set on the predicate that causes children to be hidden from the debugger. The name anticipates on anticipated changes to the compiler.

- In the standard order of terms (see section 4.6.1), 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.

large_files *(bool)*
If present and `true`, SWI-Prolog has been compiled with *large file support* (LFS) and is capable to access files larger than 2GB on 32-bit hardware. Large file-support is default on installations built using `configure` that support it and may be switched off using the configure option `--disable-largefile`.

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

max_integer *(integer)*
Maximum integer value if integers are *bounded*. See also the flag `bounded` and section 4.26.2.

max_tagged_integer *(integer)*
Maximum integer value represented as a 'tagged' value. Tagged integers require 1 word storage. Larger integers are represented as 'indirect data' and require significantly more space.

min_integer *(integer)*
Minimum integer value if integers are *bounded*. See also the flag `bounded` and section 4.26.2.

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

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).

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

pid *(int)*
Process identifier of the running Prolog process. Existence of this flag is implementation dependent.

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.

prompt_alternatives_no_bindings *(bool, changeable)*
If present and true, the top-level prints for alternatives if the query succeeded with pending choice-point, regardless of whether or not the query has variables. As there are no variables to bind it prints the answer *More?*. Here is an example

```
?- set_prolog_flag(prompt_alternatives_no_bindings, true).
?- (write(hello);write(world)).
hello
More? ;
world
```

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.

resource_database *(atom)*
Set to the absolute-filename of the attached state. Typically this is the file `boot32.prc`, the file specified with `-x` or the running executable. See also `resource/3`.

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.

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

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

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`.

signals *(bool)*
Determine whether Prolog is handling signals (software interrupts). This flag is `false` if the hosting OS does not support signal handling or the command-line option `-nosignals` is active. See section 9.6.20 for details.

system_thread_id *(int)*
On MT systems (section 8, refers to the thread-identifier used by the system for the calling thread. See also `thread_self/1`.

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.

timezone *(integer)*
Offset in seconds west of GMT of the current time-zone. Set at initialization time from the `timezone` variable associated with the POSIX tzset() function. See also `convert_time/2`.

toplevel_print_anon *(bool, changeable)*
If `true`, top-level variables starting with an underscore (_) are printed normally. If `false` they are hidden. This may be used to hide bindings in complex queries from the top-level.

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), attributes(portray)]`.

toplevel_var_size *(int, changeable)*
Maximum size counted in literals of a term returned as a binding for a variable in a top-level query that is saved for re-use using the `$` variable reference. See section 2.8.

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

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.

unix *(bool)*
If present and `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`. On other systems this flag is not available.

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. Switching this flag to `fail` disables autoloading and thus forces complete and consistent use of `use_module/[1,2]` to load the required libraries.

verbose *(Atom, changeable)*
This flags is used by `print_message/2`. If its value is `silent`, messages of type `informational` and `banner` are suppressed. The `-q` switches the value from the initial `normal` to `silent`.

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.

verbose_load *(bool, changeable)*
If `false` normal consult messages will be suppressed. Default is `true`. The value of this flag is normally controlled by the option `silent(`*Bool*`)` privided by `load_files/2`.

verbose_file_search *(bool, changeable)*
If `true` (default `false`), print messages indicating the progress of `absolute_file_name/[2,3]` in locating files. Intended for debugging complicated file-search paths. See also `file_search_path/2`.

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

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

Note that in releases up to 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.

windows *(bool)*
If present and `true`, the operating system is an implementation of Microsoft Windows (NT/2000/XP, etc.). This flag is only available on MS-Windows based versions.

write_attributes *(atom, changeable)*
Defines how `write/1` an friends write attributed variables. The option values are described with the `attributes` option of `write_term/3`. Default is `ignore`.

write_help_with_overstrike *(bool)*
Internal flag used by `help/1` when writing to a terminal. If present and `true` it prints bold and underlined text using *overstrike*.

xpce *(bool)*
Available and set to `true` if the XPCE graphics system is loaded.

xpce_version *(atom)*
Available and set to the version of the loaded XPCE system.

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/2` 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_load_file/2*
  Hook into `load_files/2` to load other data-formats for Prolog sources from 'non-file' resources. The `load_files/2` predicate is the ancestor of `consult/1`, `use_module/1`, etc.

- *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).

- *prolog:debug_control_hook/1*
  Hook in `spy/1`, `nospy/1`, `nospyall/0` and `debugging/0` to extend these control-predicates to higher-level libraries.

- *prolog:help_hook/1*
  Hook in `help/0`, `help/1` and `apropos/1` to extend the help-system.

- *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).

- *attr_unify_hook/2*
  Unification hook for attributed variables. Can be defined in any module. See section 6.1 for details.

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` and `file_search_path/2`). 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 restarted. 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 should be considered.

Autoloading only handles (library) source files that use the module mechanism described in chapter 5. 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` updates the autoload index. It searches for all library directories (see `library_directory/1` and `file_search_path/2`) holding the file `MKINDEX.pl` or `INDEX.pl`. If the current user can write or create the file `INDEX.pl` and it does not exist or is older than the directory or one of its files, the index for this directory is updated. If the file `MKINDEX.pl` exists updating is achieved by loading this file, normally containing a directive calling `make_library_index/2`. Otherwise `make_library_index/1` is called, creating an index for all `*.pl` files containing a module.

Below is an example creating a completely indexed library directory.

```
% 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.

make_library_index(+*Directory, +ListOfPatterns*)
: Normally used in `MKINDEX.pl`, this predicate creates `INDEX.pl` for *Directory*, indexing all files that match one of the file-patterns in *ListOfPatterns*.

  Sometimes library packages consist of one public load file and a number of files used by this load-file, exporting predicates that should not be used directly by the end-user. Such a library can be placed in a sub-directory of the library and the files containing public functionality can be added to the index of the library. As an example we give the XPCE library's `MKINDEX.pl`, including the public functionality of `trace/browse.pl` to the autoloadable predicates for the XPCE package.

```
:- make_library_index('.',
                      [ '*.pl',
                        'trace/browse.pl'
                      ]).
```

reload_library_index

Force reloading the index after modifying the set of library directories by changing the rules for `library_directory/1`, `file_search_path/2`, adding or deleting `INDEX.pl` files. This predicate does *not* update the `INDEX.pl` files. Check `make_library_index/[1,2]` and `make/0` for updating the index files.

Normally, the index is reloaded automatically if a predicate cannot be found in the index and the set of library directories has changed. Using `reload_library_index/0` is necessary if directories are removed or the order of the library directories is changed.

2.14 Garbage Collection

SWI-Prolog provides garbage-collection, last-call optimization and atom garbage collection. These features are controlled using prolog flags (see `current_prolog_flag/2`).

2.15 Syntax Notes

SWI-Prolog uses ISO-Prolog standard syntax, which is closely compatible to Edinburgh Prolog 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 character code of ⟨*char*⟩. To test whether `C` is a lower case character one can use `between(0'a, 0'z, C)`. If character codes are enabled (default) ⟨*char*⟩ can use `\` escape sequences. The sequence `0'\t` represents the TAB character using symbolic notation.
- `/* .../* .../* .../*`
  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.

Processor Character Set

The processor character set specifies the class of each character used for parsing Prolog source text. Character classification is fixed to use UCS/Unicode as provided by the C-library `wchar_t` based primitives. See also section 2.17.

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 up to 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. Not 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).

\xXX..
Hexadecimal specification of a character. The closing \ is obligatory according to the ISO standard, but optional in SWI-Prolog to enhance compatibility to the older Edinburgh standard. The code \xa\3 emits the character 10 (hexadecimal 'a') followed by '3'. Characters specified this way are interpreted as Unicode characters. See also \u.

\uXXXX
Unicode character specification where the character is specified using *exactly* 4 hexadecimal digits. This is an extension to the ISO standard fixing two problems. First of all, where \x defines a numeric character code, it doesn't specify the character set in which the character should be interpreted. Second, it is not needed to use the idiosyncratic closing \ ISO Prolog syntax.

\UXXXXXXXX
Same as \uXXXX, but using 8 digits to cover the whole Unicode set.

`\40`
: Octal character specification. The rules and remarks for hexadecimal specifications apply to octal specifications as well.

`\`⟨*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 advised to use the more widely supported `format/[2,3]` predicate instead. If you insist upon using `writef/2`, 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.

Unicode Prolog source

The ISO standard specifies the Prolog syntax in ASCII characters. As SWI-Prolog supports Unicode in source files we must extend the syntax. This section describes the implication for the source files, while writing international source files is described in section 3.1.3.

The SWI-Prolog Unicode character classification is based on version 4.1.0 of the Unicode standard. Please that `char_type/2` and friends, intended to be used with all text except Prolog source code is based on the C-library locale-based classification routines.

- *Quoted atoms and strings*
  Any character of any script can be used in quoted atoms and strings. The escape sequences `\uXXXX` and `\UXXXXXXXX` (see section 2.15.1) were introduced to specify Unicode code points in ASCII files.

- *Atoms and Variables*
  We handle them in one item as they are closely related. The Unicode standard defines a syntax for identifiers in computer languages.[5] In this syntax identifiers start with `ID_Start` followed by a sequence of `ID_Continue` codes. Such sequences are handled as a single token in SWI-Prolog. The token is a *variable* iff it starts with an uppercase character or an underscore (_). Otherwise it is an atom. Note that many languages do not not have the notion of character-case. In such languages variables *must* be written as `_name`.

- *White space*
  All characters marked as separators in the Unicode tables are handled as layout characters.

[5] `http://www.unicode.org/reports/tr31/`

- *Other characters*
  The first 128 characters follow the ISO Prolog standard. All other characters not covered by the rules above are considered 'solo' characters: they form single-character atoms. We would like to have a more appropriate distinction between what is known to Prolog as 'solo' characters and 'singleton' characters.

Singleton variable checking

A singleton variable is a variable that appears only one time in a clause. It can always be replaced by _, the *anonymous* variable. In some cases however people prefer to give the variable a name. As mistyping a variable is a common mistake, Prolog systems generally give a warning (controlled by `style_check/1`) if a variable is used only once. The system can be informed a variable is known to appear once by *starting* it with an underscore. E.g. `_Name`. Please note that any variable, except plain _ shares with variables of the same name. The term `t(_X, _X)` is equivalent to `t(X, X)`, which is *different* from `t(_, _)`.

As Unicode requires variables to start with an underscore in many languages this schema needs to be extended.[6] First we define the two classes of named variables.

- *Named singleton variables*
  Named singletons start with a double underscore (__) or a single underscore followed by an uppercase letter. E.g. `__var` or `_Var`.

- *Normal variables*
  All other variables are 'normal' variables. Note this makes `_var` a normal variable.[7]

Any normal variable appearing exactly ones in the clause *and* any named singleton variables appearing more than once are reported. Below are some examples with warnings in the right column.

test(_).	
test(_a).	Singleton variables: [_a]
test(A).	Singleton variables: [A]
test(_A).	
test(__a).	
test(_, _).	
test(_a, _a).	
test(__a, __a).	Singleton-marked variables appearing more than once: [__a]
test(_A, _A).	Singleton-marked variables appearing more than once: [_A]
test(A, A).	

2.16 Infinite trees (cyclic terms)

SWI-Prolog has limited support for infinite trees, also known as cyclic terms. Full support requires special code in all built-in predicates that require recursive exploration of a term. The current version supports cycles terms in the pure Prolog kernel including the garbage collector and in the following predicates: `=../2`, `==/2`, `=@=/2`, `=/2`, `@</2`, `@=</2`, `@>=/2`, `@>/2`, `\==/2`, `\=@=/2`,

[6] After a proposal by Richard O'Keefe.
[7] Some Prolog dialects write variables this way.

`\=/2`, `acyclic_term/1`, `bagof/3`, `compare/3`, `copy_term/2`, `cyclic_term/1`, `dif/2`, `duplicate_term/2`, `findall/3`, `ground/1`, `hash_term/2`, `numbervars/[3,4]`, `recorda/3`, `recordz/3`, `setof/3`, `term_variables/2`, `throw/1`, `when/2`, `write/1` (incomplete) .

2.17 Wide character support

SWI-Prolog supports *wide characters*, characters with character codes above 255 that cannot be represented in a single *byte*. *Universal Character Set* (UCS) is the ISO/IEC 10646 standard that specifies a unique 31-bits unsigned integer for any character in any language. It is a superset of 16-bit Unicode, which in turn is a superset of ISO 8859-1 (ISO Latin-1), a superset of US-ASCII. UCS can handle strings holding characters from multiple languages and character classification (uppercase, lowercase, digit, etc.) and operations such as case-conversion are unambiguously defined.

For this reason SWI-Prolog has two representations for atoms and string objects (see section 4.23. If the text fits in ISO Latin-1, it is represented as an array of 8-bit characters. Otherwise the text is represented as an array of 32-bit numbers. This representational issue is completely transparent to the Prolog user. Users of the foreign language interface as described in section 9 sometimes need to be aware of these issues though.

Character coding comes into view when characters of strings need to be read from or written to file or when they have to be communicated to other software components using the foreign language interface. In this section we only deal with I/O through streams, which includes file I/O as well as I/O through network sockets.

2.17.1 Wide character encodings on streams

Although characters are uniquely coded using the UCS standard internally, streams and files are byte (8-bit) oriented and there are a variety of ways to represent the larger UCS codes in an 8-bit octet stream. The most popular one, especially in the context of the web, is UTF-8. Bytes 0 ... 127 represent simply the corresponding US-ASCII character, while bytes 128 ... 255 are used for multi-byte encoding of characters placed higher in the UCS space. Especially on MS-Windows the 16-bit Unicode standard, represented by pairs of bytes is also popular.

Prolog I/O streams have a property called *encoding* which specifies the used encoding that influence `get_code/2` and `put_code/2` as well as all the other text I/O predicates.

The default encoding for files is derived from the Prolog flag `encoding`, which is initialised from the environment. If the environment variable `LANG` ends in "UTF-8", this encoding is assumed. Otherwise the default is `text` and the translation is left to the wide-character functions of the C-library. [8] The encoding can be specified explicitly in `load_files/2` for loading Prolog source with an alternative encoding, `open/4` when opening files or using `set_stream/2` on any open stream. For Prolog source files we also provide the `encoding/1` directive that can be used to switch between encodings that are compatible to US-ASCII (`ascii`, `iso_latin_1`, `utf8` and many locales). See also section 3.1.3 for writing Prolog files with non-US-ASCII characters and section 2.15.1 for syntax issues. For additional information and Unicode resources, please visit `http://www.unicode.org/`.

SWI-Prolog currently defines and supports the following encodings:

[8] The Prolog native UTF-8 mode is considerably faster than the generic mbrtowc() one.

octet
Default encoding for `binary` streams. This causes the stream to be read and written fully untranslated.

ascii
7-bit encoding in 8-bit bytes. Equivalent to `iso_latin_1`, but generates errors and warnings on encountering values above 127.

iso_latin_1
8-bit encoding supporting many western languages. This causes the stream to be read and written fully untranslated.

text
C-library default locale encoding for text files. Files are read and written using the C-library functions mbrtowc() and wcrtomb(). This may be the same as one of the other locales, notably it may be the same as `iso_latin_1` for western languages and `utf8` in a UTF-8 context.

utf8
Multi-byte encoding of full UCS, compatible to `ascii`. See above.

unicode_be
Unicode *Big Endian*. Reads input in pairs of bytes, most significant byte first. Can only represent 16-bit characters.

unicode_le
Unicode *Little Endian*. Reads input in pairs of bytes, least significant byte first. Can only represent 16-bit characters.

Note that not all encodings can represent all characters. This implies that writing text to a stream may cause errors because the stream cannot represent these characters. The behaviour of a stream on these errors can be controlled using `set_stream/2`. Initially the terminal stream write the characters using Prolog escape sequences while other streams generate an I/O exception.

BOM: Byte Order Mark

From section 2.17.1, you may have got the impression text-files are complicated. This section deals with a related topic, making live often easier for the user, but providing another worry to the programmer. **BOM** or *Byte Order Marker* is a technique for identifying Unicode text-files as well as the encoding they use. Such files start with the Unicode character 0xFEFF, a non-breaking, zero-width space character. This is a pretty unique sequence that is not likely to be the start of a non-Unicode file and uniquely distinguishes the various Unicode file formats. As it is a zero-width blank, it even doesn't produce any output. This solves all problems, or ...

Some formats start of as US-ASCII and may contain some encoding mark to switch to UTF-8, such as the `encoding="UTF-8"` in an XML header. Such formats often explicitly forbid the the use of a UTF-8 BOM. In other cases there is additional information telling the encoding making the use of a BOM redundant or even illegal.

The BOM is handled by SWI-Prolog `open/4` predicate. By default, text-files are probed for the BOM when opened for reading. If a BOM is found, the encoding is set accordingly and the property `bom`(*true*) is available through `stream_property/2`. When opening a file for writing, writing a BOM can be requested using the option `bom`(*true*) with `open/4`.

2.18 System limits

2.18.1 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 generally to $2^{\text{bits-per-long}-5}$ bytes.

The PrologScript facility described in section 2.10.2 provides a mechanism for specifying options with the load-file. On Windows the default stack-sizes are controlled using the Windows registry on the key `HKEY_CURRENT_USER\Software\SWI\Prolog` using the names `localSize`, `globalSize` and `trailSize`. The value is a `DWORD` expressing the default stack size in Kbytes. A GUI for modifying these values is provided using the XPCE package. To use this, start the XPCE manual tools using `manpce/0`, after which you find *Preferences* in the *File* menu.

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 up to 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().

2.18.2 Other Limits

Clauses The only limit on clauses is their arity (the number of arguments to the head), which is limited to 1024. Raising this limit is easy and relatively cheap, removing it is harder.

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`.

The number of atoms is limited to 16777216 (16M) on 32-bit machines. On 64-bit machines this is virtually unlimited. See also section 9.6.2.

Memory areas On 32-bit hardware, 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. On 64-bit hardware there are no meaningful limits.

Integers Integers are 64-bit on 32 as well as 64-bit machines. Integers up to the value of the `max_tagged_integer` prolog-flag are represented more efficiently on the stack. For clauses and records the difference is much smaller.

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. In addition, this stack is used by some built-in predicates to handle cyclic terms. Its default size limit is proportional to the global stack limit such that it will never overflow.

Table 2.2: Memory areas

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

2.18.3 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`).

Initialising and Managing a Prolog Project

3

Prolog text-books give you an overview of the Prolog language. The manual tells you what predicates are provided in the system and what they do. This chapter wants to explain how to run a project. There is no ultimate 'right' way to do this. Over the years we developed some practice in this area and SWI-Prolog's commands are there to support this practice. This chapter describes the conventions and supporting commands.

The first two sections (section 3.1 and section 3.2 only require plain Prolog. The remainder discusses the use of the built-in graphical tools that require the XPCE graphical library installed on your system.

3.1 The project source-files

Organisation of source-files depends largely on the size of your project. If you are doing exercises for a Prolog course you'll normally use one file for each exercise. If you have a small project you'll work work with one directory holding a couple of files and some files to link it all together. Even bigger projects will be organised in sub-projects each using their own directory.

3.1.1 File Names and Locations

File Name Extensions

The first consideration is what extension to use for the source-files. Tradition calls for `.pl`, but conflicts with Perl force the use of another extension on systems where extensions have global meaning, such as MS-Windows. On such systems `.pro` is the common alternative.[1]

All versions of SWI-Prolog load files with the extension `.pl` as well as with the registered alternative extension without explicitly specifying the extension. For portability reasons we propose the following convention:

If there is no conflict because you do not use a conflicting application or the system does not force a unique relation between extension and application, use `.pl`.

With a conflict choose `.pro` and use this extension for the files you want to load through your file-manager. Use `.pl` for all other files for maximal portability.

Project Directories

Large projects are generally composed of sub-projects, each using their own directory or directory-structure. If nobody else will ever touch your files and you use only one computer there is little to

[1] On MS-Windows, the alternative extension is stored in the registry-key `HKEY_CURRENT_USER/Software/SWI/Prolog/fileExtension` or `HKEY_LOCAL_MACHINE/Software/SWI/Prolog/fileExte`

worry about, but this is rarely the case with a large project.

To improve portability, SWI-Prolog uses the POSIX notation for filenames, which uses the forward slash (`/`) to separate directories. Just before hitting the file-system it uses `prolog_to_os_filename/2` to convert the filename to the conventions used by the hosting operating system. It is *strongly* advised to write paths using the `/`, especially on systems using the `\` for this purpose (MS-Windows). Using `\` violates the portability rules and requires you to *double* the `\` due to the Prolog quoted-atom escape rules.

Portable code should use `prolog_to_os_filename/2` to convert computed paths into system-paths when constructing commands for `shell/1` and friends.

Sub-projects using search-paths

Thanks to Quintus, Prolog adapted an extensible mechanism for searching files using `file_search_path/2`. This mechanism allows for comfortable and readable specifications.

Suppose you have extensive library packages on graph-algorithms, set-operations and GUI-primitives. These sub-projects are likely candidates for re-use in future projects. A good choice is to create a directory with sub-directories for each of these sub-projects.

Next, there are three options. One is to add the sub-projects to the directory-hierarchy of the current project. Another is to use a completely dislocated directory and finally the sub-project can be added to the SWI-Prolog hierarchy. Using local installation, a typical `file_search_path/2` is:

```
:- prolog_load_context(directory, Dir),
   asserta(user:file_search_path(myapp, Dir)).

user:file_search_path(graph, myapp(graph)).
user:file_search_path(ui,    myapp(ui)).
```

For using sub-projects in the SWI-Prolog hierarchy one should use the path-alias `swi` as basis. For a system-wide installation use an absolute-path.

Extensive sub-projects with a small well-defined API should define a load-file using `use_module/1` calls to import the various library-components and export the API.

3.1.2 Project Special Files

There are a number of tasks you typically carry out on your project, such as loading it, creating a saved-state, debugging it, etc. Good practice on large projects is to define small files that hold the commands to execute such a task, name this file after the task and give it a file-extension that makes starting easy (see section 3.1.1). The task *load* is generally central to these tasks. Here is a tentative list.

- *`load.pl`*
  Use this file to set up the environment (prolog flags and file search paths) and load the sources. Quite commonly this file also provides convenient predicates to parse command-line options and start the application.

- *`run.pl`*
  Use this file to start the application. Normally it loads `load.pl` in silent-mode, and calls one of the starting predicates from `load.pl`.

- *save.pl*
  Use this file to create a saved-state of the application by loading `load.pl` and call `qsave_program/2` to generate a saved-state with the proper options.

- *debug.pl*
  Loads the program for debugging. In addition to loading `load.pl` this file defines rules for `portray/1` to modify printing rules for complex terms and customisation rules for the debugger and editing environment. It may start some of these tools.

3.1.3 International source files

As discussed in section 2.17, SWI-Prolog supports international character handling. Its internal encoding is UNICODE. I/O streams convert to/from this internal format. This sections discusses the options for source-files not in US-ASCII.

SWI-Prolog can read files in any of the encodings described in section 2.17. Two encodings are of particular interest. The `text` encoding deals with the current *locale*, the default used by this computer for representing text files. The encodings `utf8`, `unicode_le` and `unicode_be` are *UNICODE* encodings: they can represent—in the same file—characters of virtually any known language. In addition, they do so unambiguously.

If one wants to represent non US-ASCII text as Prolog terms in a source-file there are several options:

- *Use escape sequences*
  This approach describes NON-ASCII as sequences of the form *octal*\. The numerical argument is interpreted as a UNICODE character.[2] The resulting Prolog file is strict 7-bit US-ASCII, but if there are many NON-ASCII characters it becomes very unreadable.

- *Use local conventions*
  Alternatively the file may be specified using local conventions, such as the EUC encoding for Japanese text. The disadvantage is portability. If the file is moved to another machine this machine must be using the same *locale* or the file is unreadable. There is no elegant if files from multiple locales must be united in one application using this technique. In other words, it is fine for local projects in countries with uniform locale conventions.

- *Using UTF-8 files*
  The best way to specify source files with many NON-ASCII characters is definitely the use of UTF-8 encoding. Prolog can be notified two ways of this encoding, using a UTF-8 *BOM* (see section 2.17.1) or using the directive `:- encoding(utf8).`. Many todays text editors, including PceEmacs, are capable of editing UTF-8 files. Projects that started using local conventions can be be re-coded using the Unix `iconv` tool or often using a commands offered by the editor.

3.2 Using modules

Modules have been debated fiercely in the Prolog world. Despite all counter-arguments we feel they are extremely useful because

[2] To my knowledge, the ISO escape sequences is limited to 3 octal digits, which means most characters cannot be represented.

- *They hide local predicates*
  This is the reason they have been invented in the first place. Hiding provides two features. They allow for short predicate names without worrying about conflicts. Given the flat name-space introduced by modules, they still require meaningful module names as well as meaningful names for exported predicates.

- *They document the interface*
  Possibly more important then avoiding name-conflicts is their role in documenting which part of the file is for public usage and which is private. When editing a module you may assume you can reorganise anything but the name and semantics of the exported predicates without worrying.

- *They help the editor*
  The PceEmacs built-in editor does on-the-fly cross-referencing of the current module, colouring predicates based on their origin and usage. Using modules, the editor can quickly find out what is provided by the imported modules by reading just the first term. This allows it to indicate real-time which predicates are not used or not defined.

Using modules is generally easy. Only if you write meta-predicates (predicates reasoning about other predicates) that are exported from a module good understanding of resolution of terms to predicates inside a module is required. Here is a typical example from `readutil`.

```
:- module(read_util,
          [ read_line_to_codes/2,       % +Fd, -Codes
            read_line_to_codes/3,       % +Fd, -Codes, ?Tail
            read_stream_to_codes/2,     % +Fd, -Codes
            read_stream_to_codes/3,     % +Fd, -Codes, ?Tail
            read_file_to_codes/3,       % +File, -Codes, +Options
            read_file_to_terms/3        % +File, -Terms, +Options
          ]).
```

3.3 The test-edit-reload cycle

SWI-Prolog does not enforce the use of a particular editor for writing down Prolog source code. Editors are complicated programs that must be mastered in detail for real productive programming and if you are familiar with a specific editor you should not be forced to change. You may specify your favourite editor using the prolog flag `editor`, the environment variable `EDITOR` or by defining rules for prolog_edit:edit_source/1 (see section 4.4).

The use of a built-in editor, which is selected by setting the prolog-flag `editor` to `pce_emacs`, has advantages. The XPCE *editor* object around which the built-in PceEmacs is built can be opened as a Prolog stream allowing analysis of your source by the real Prolog system.

3.3.1 Locating things to edit

The central predicate for editing something is `edit/1`, an extensible front-end that searches for objects (files, predicates, modules as well as XPCE classes and methods) in the Prolog database. If multiple matches are found it provides a choice. Together with the built-in completion on atoms bound to the `TAB` key this provides a quick way to edit objects:

```
?- edit(country).
Please select item to edit:

  1 chat:country/10       '/staff/jan/lib/prolog/chat/countr.pl':16
  2 chat:country/1        '/staff/jan/lib/prolog/chat/world0.pl':72

Your choice?
```

3.3.2 Editing and incremental compilation

One of the nice features of Prolog is that the code can be modified while the program is running. Using pure Prolog you can trace a program, find it is misbehaving, enter a *break environment*, modify the source code, reload it and finally do *retry* on the misbehaving predicate and try again. This sequence is not uncommon for long-running programs. For faster programs one normally aborts after understanding the misbehaviour, edit the source, reload it and try again.

One of the nice features of SWI-Prolog is the availability of `make/0`, a simple predicate that checks all loaded source files to see which ones you have modified. It then reloads these files, considering the module from which the file was loaded originally. This greatly simplifies the trace-edit-verify development cycle. After the tracer reveals there is something wrong with `prove/3`, you do:

```
?- edit(prove).
```

Now edit the source, possibly switching to other files and making multiple changes. After finishing invoke `make/0`, either through the editor UI () Compile/Make (`Control-C Control-M`)) or on the top-level and watch the files being reloaded.[3]

```
?- make.
% show compiled into photo_gallery 0.03 sec, 3,360 bytes
```

3.4 Using the PceEmacs built-in editor

3.4.1 Activating PceEmacs

Initially `edit/1` uses the editor specified in the `EDITOR` environment variable. There are two ways to force it to use the built-in editor. One is to set the prolog-flag `editor` to `pce_emacs` and the other is by starting the editor explicitly using the `emacs/[0,1]` predicates.

3.4.2 Bluffing through PceEmacs

PceEmacs closely mimics Richard Stallman's GNU-Emacs commands, adding features from modern window-based editors to make it more acceptable for beginners.[4]

[3] Watching these files is a good habit. If expected files are not reloaded you may have forgotten to save them from the editor or you may have been editing the wrong file (wrong directory).

[4] Decent merging with MS-Windows control-key conventions is difficult as many conflict with GNU-Emacs. Especially the cut/copy/paste commands conflict with important GNU-Emacs commands.

At the basis, PceEmacs maps keyboard sequences to methods defined on the extended *editor* object. Some frequently used commands are, with their key-binding, presented in the menu-bar above each editor window. A complete overview of the bindings for the current *mode* is provided through) Help/Show key bindings (`Control-h Control-b`).

Edit modes

Modes are the heart of (Pce)Emacs. Modes define dedicated editing support for a particular kind of (source-)text. For our purpose we want *Prolog mode*. Their are various ways to make PceEmacs use Prolog mode for a file.

- *Using the proper extension*
  If the file ends in `.pl` or the selected alternative (e.g. `.pro`) extension, Prolog mode is selected.
- *Using* `#!/path/to/pl`
  If the file is a *Prolog Script* file, starting with the line `#!/path/to/pl options -s`, Prolog mode is selected regardless of the extension
- *Using* `-*- Prolog -*-`
  If the above sequence appears in the first line of the file (inside a Prolog comment) Prolog mode is selected.
- *Explicit selection*
  Finally, using) File/Mode/Prolog (`y`)ou can switch to Prolog mode explicitly.

Frequently used editor commands

Below we list a few important commands and how to activate them.

- *Cut/Copy/Paste*
  These commands follow Unix/X11 traditions. You're best suited with a three-button mouse. After selecting using the left-mouse (double-click uses word-mode and triple line-mode), the selected text is *automatically* copied to the clipboard (X11 primary selection on Unix). *Cut* is achieved using the DEL key or by typing something else at the location. *Paste* is achieved using the middle-mouse (or wheel) button. If you don't have a middle mouse-button, pressing the left- and right-button at the same time is interpreted as a middle-button click. If nothing helps there is the) Edit/Paste menu-entry. Text is pasted at the caret-location.
- *Undo*
  Undo is bound to the GNU-Emacs Control-_ as well as the MS-Windows Control-Z sequence.
- *Abort*
  Multi-key sequences can be aborted at any stage using Control-G.
- *Find*
  Find (Search) is started using Control-S (forward) or Control-R (backward). PceEmacs implements *incremental search*. This is difficult to use for novices, but very powerful once you get the clue. After one of the above start-keys the system indicates search mode in the status line. As you are typing the search-string, the system searches for it, extending the search with every character you type. It illustrates the current match using a green background.

If the target cannot be found, PceEmacs warns you and no longer extends the search-string.[5] During search some characters have special meaning. Typing anything but these characters commits the search, re-starting normal edit mode. Special commands are:

Control-S
: Search for next forwards.

Control-R
: Search for next backwards.

Control-W
: Extend search to next word-boundary.

Control-G
: Cancel search, go back to where it started.

ESC
: Commit search, leaving caret at found location.

Backspace
: Remove a character from the search string.

- *Dynamic Abbreviation*
  Also called *dabbrev* is an important feature of Emacs clones to support programming. After typing the first few letters of an identifier you may hit Alt-/, causing PceEmacs to search backwards for identifiers that start the same and using it to complete the text you typed. A second Alt-/ searches further backwards. If there are no hits before the caret it starts searching forwards. With some practice, this system allows for very fast entering code with nice and readable identifiers (or other difficult long words).

- *Open (a file)*
  Is called) File/Find file (`Control-x Control-f`). By default the file is loaded into the current window. If you want to keep this window, Hit Alt-s or click the little icon at the bottom-left to make the window *sticky*.

- *Split view*
  Sometimes you want to look at two places of the same file. To do this, use Control-x 2 to create a new window pointing to the same file. Do not worry, you can edit as well as move around in both. Control-x 1 kills all other windows running on the same file.

These were the most commonly used commands. In section section 3.4.3 we discuss specific support for dealing with Prolog source code.

3.4.3 Prolog Mode

In the previous section (section 3.4.2) we explained the basics of PceEmacs. Here we continue with Prolog specific functionality. Possibly the most interesting is *Syntax highlighting*. Unlike most editors where this is based on simple patterns, PceEmacs syntax highlighting is achieved by Prolog itself actually reading and interpreting the source as you type it. There are three moments at which PceEmacs checks (part of) the syntax.

[5] GNU-Emacs keeps extending the string, but why? Adding more text will not make it match.

Clauses	
Blue bold	Head of an exported predicate
Red bold	Head of a predicate that is not called
Black Bold	Head of remaining predicates
Calls in the clause-body	
Blue	Call to built-in or imported predicate
Red	Call to not-defined predicate
Purple	Call to dynamic predicate
Other entities	
Dark green	Comment
Dark blue	Quoted atom or string
Brown	Variable

Table 3.1: Colour conventions

- *After typing a* **.**
  After typing a **.** that is not preceded by a *symbol* character the system assumes you completed a clause, tries to find the start of this clause and verifies the syntax. If this process succeeds it colours the elements of the clause according to the rules given below. Colouring is done using information from the last full check on this file. If it fails, the syntax error is displayed in the status line and the clause is not coloured.

- *After the command* Control-c Control-s
  Acronym for **C**check **S**yntax it performs the same checks as above for the clause surrounding the caret. On a syntax error however, the caret is moved to the expected location of the error.[6]

- *After pausing for two seconds*
  After a short pause (2 seconds), PceEmacs opens the edit-buffer and reads it as a whole, creating an index of defined, called, dynamic, imported and exported predicates. After completing this, it re-reads the file and colours all clauses and calls with valid syntax.

- *After typing* Control-l Control-l
  The Control-l commands re-centers the window (scrolls the window to make the caret the center of the window). Hitting this command twice starts the same process as above.

The colour schema itself is defined in `emacs/prolog_colour`. The colouring can be extended and modified using multifile predicates. Please check this source-file for details. In general, underlined objects have a popup (right-mouse button) associated for common commands such as viewing the documentation or source. **Bold** text is used to indicate the definition of objects (typically predicates when using plain Prolog). Other colours follow intuitive conventions. See table 3.4.3.

Layout support Layout is not 'just nice', it is *essential* for writing readable code. There is much debate on the proper layout of Prolog. PceEmacs, being a rather small project supports only one particular style for layout.[7] Below are examples of typical constructs.

[6] In most cases the location where the parser cannot proceed is further down the file than the actual error-location.
[7] Defined in Prolog in the file `emacs/prolog_mode`, you may wish to extend this. Please contribute your extensions!

```
head(arg1, arg2).

head(arg1, arg2) :- !.

head(Arg1, arg2) :- !,
        call1(Arg1).

head(Arg1, arg2) :-
        (   if(Arg1)
        ->  then
        ;   else
        ).

head(Arg1) :-
        (   a
        ;   b
        ).

head :-
        a(many,
          long,
          arguments(with,
                    many,
                    more),
          and([ a,
                long,
                list,
                with,
                a,
              | tail
              ])).
```

PceEmacs uses the same conventions as GNU-Emacs. The TAB key indents the current line according to the syntax rules. Alt-q indents all lines of the current clause. It provides support for head, calls (indented 1 tab), if-then-else, disjunction and argument-lists broken across multiple lines as illustrated above.

Finding your way around

The command Alt-. extracts name and arity from the caret location and jumps (after conformation or edit) to the definition of the predicate. It does so based on the source-location database of loaded predicates also used by `edit/1`. This makes locating predicates reliable if all sources are loaded and up-to-date (see `make/0`).

In addition, references to files in `use_module/[1,2]`, `consult/1`, etc. are red if the file cannot be found and underlined blue if the file can be loaded. A popup allows for opening the referenced file.

3.5 The Graphical Debugger

SWI-Prolog offers two debuggers. One is the traditional text-console based 4-port Prolog tracer and the other is a window-based source-level debugger. The window-based debugger requires XPCE installed. It operates based on the `prolog_trace_interception/4` hook and other low-level functionality described in chapter B.

Window-based tracing provides much better overview due to the eminent relation to your source-code, a clear list of named variables and their bindings as well as a graphical overview of the call and choice-point stack. There are some drawbacks though. Using a textual trace on the console one can scroll back and examine the past, while the graphical debugger just presents a (much better) overview of the current state.

3.5.1 Invoking the window-based debugger

Whether the text-based or window-based debugger is used is controlled using the predicates `guitracer/0` and `noguitracer/0`. Entering debug mode is controlled using the normal predicates for this: `trace/0` and `spy/1`. In addition, PceEmacs prolog mode provides the command) Prolog/Break at (`Control-c b`) to insert a break-point at a specific location in the source-code.

guitracer
: This predicate installs the above-mentioned hooks that redirect tracing to the window-based environment. No window appears. The debugger window appears as actual tracing is started through `trace/0`, by hitting a spy-point defined by `spy/1` or a break-point defined using PceEmacs command) Prolog/Break at (`Control-c b`).

noguitracer
: Disable the hooks installed by `guitracer/0`, reverting to normal text-console based tracing.

gtrace
: Utility defined as `guitracer,trace`.

gdebug
: Utility defined as `guitracer,debug`.

gspy(+*Predicate*)
: Utility defined as `guitracer,spy(Predicate)`.

3.6 The Prolog Navigator

Another tool is the *Prolog Navigator*. This tool can be started from PceEmacs using the command) Browse/Prolog navigator, from the GUI debugger or using the programmatic IDE interface described in section 3.8.

3.7 Cross referencer

A cross-referencers is a tool examining the caller-callee relation between predicates and using this information to explicate dependency relations between source files, find calls to non-existing predicates and predicates for which no callers can be found. Cross-referencing is useful during program

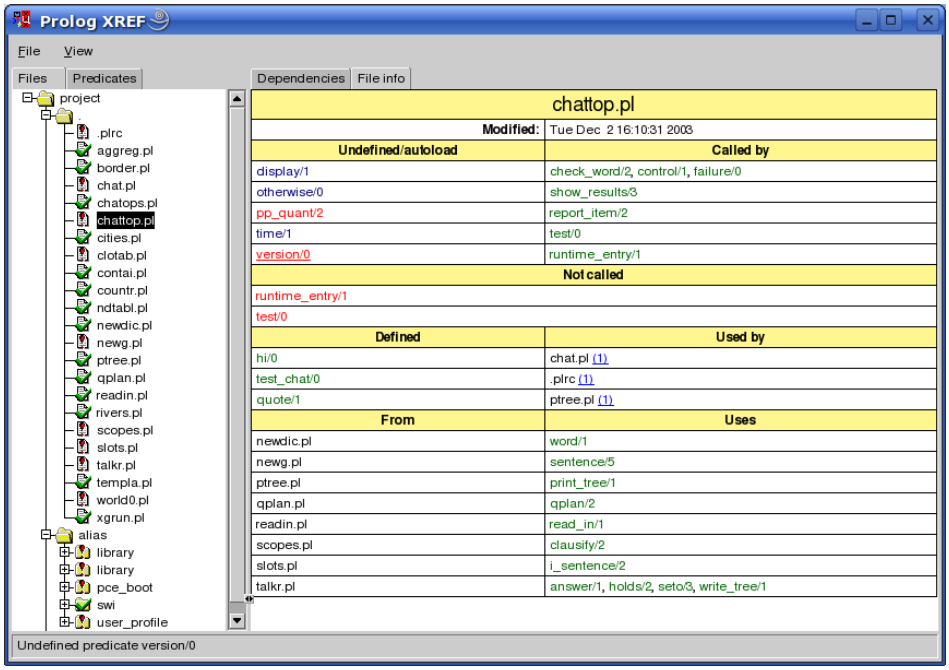

Figure 3.1: File info for `chattop.pl`, part of CHAT80

development, reorganisation, cleanup, porting and other program maintenance tasks. The dynamic nature of Prolog makes the task non-trivial. Goals can be created dynamically `call/1` after construction of a goal term. Abtract interpretation can find some of such calls, but the ultimately they can come from external communication, making it completely impossible to predict the callee. In other words, the cross-referencer has only partial understanding of the program and its results are necessarily incomplete. Still, it provides valuable information to the developer.

SWI-Prolog's cross-referencer is split into two parts. The standard Prolog library `prolog_xref` is an extensible library for information gathering described in section A.17 and the XPCE

library `pce_xref` provides a graphical frontend for the cross-referencer described here. We demonstrate the tool on CHAT80, a natural language question and answer system by Fernando C.N. Pereira and David H.D. Warren.

gxref

Run cross-referencer on all currently loaded files and present a graphical overview of the result. As the predicate operates on the currently loaded application it must be run after loading the application.

The **left window** (see figure 3.1 provides browsers for loaded files and predicates. To avoid long file paths the file hierarchy has three main branches. The first is the current directory holding the sources. The second is marked `alias` and below it are the file-search-path aliases (see `file_search_path/2` and `absolute_file_name/3`). Here you find files loaded from the system as well as modules of the program loaded from other locations using file search path. All loaded files that fall outside these categories are below the last branch called `/`. File where the system found suspicious dependencies are marked with an exclamation mark. This also holds for directories holding such files. Clicking on a file opens a *File info* window in the right pane.

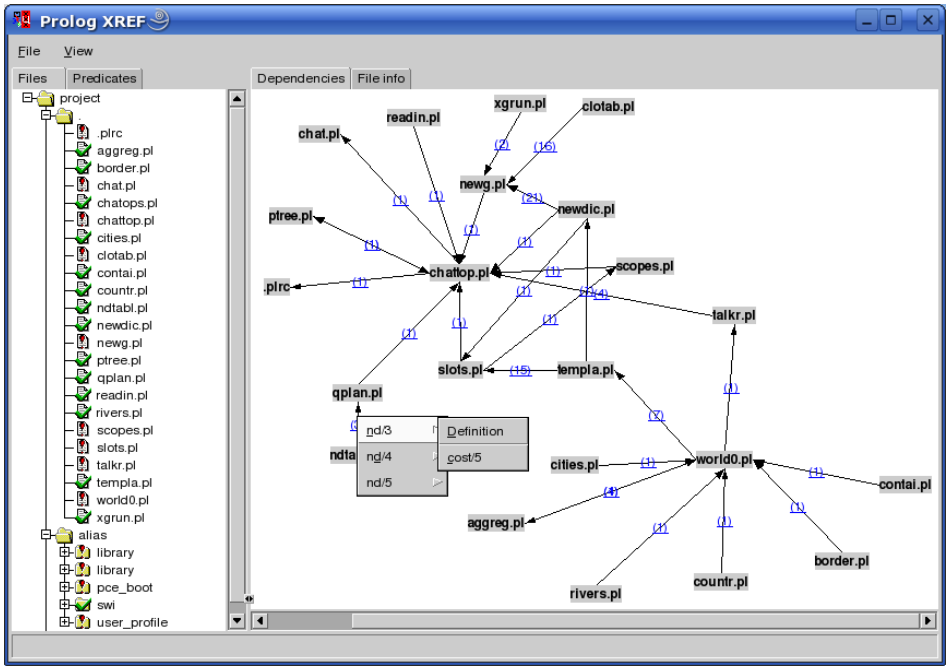

Figure 3.2: Dependencies between source files of CHAT80

The **File info** window shows a file, its main properties, its undefined and not-called predicates and its import- and export relations to other files in the project. Both predicates and files can be opened by clicking on them. The number of callers in a file for a certain predicate is indicated with a blue underlined number. A left-click will open a list and allows to edit the calling predicate.

The **Dependencies** (see figure 3.2) window displays a graphical overview of dependencies between files. Using the background menu a complete graph of the project can be created. It is also possible to drag files onto the graph window and use the menu on the nodes to incrementally expand the graph. The underlined blue text indicates the number of predicates used in the destination file. Left-clicking opens a menu to open the definition or select one of the callers.

Module and non-module files The cross-referencer threads module and non-module project files differently. Module files have explicit import and export relations and the tool shows the usage and consistency of the relations. Using the menu-command Header the tool creates a consistent import list for the module that can be included in the file. The tool computes the dependency relations between the non-module files. If the user wishes to convert the project into a module-based one the Header command generates an appropriate module header and import list. Note that the cross-referencer may have missed dependencies and does not deal with meta-predicates defined in one module and called in another. Such problems must be resolved manually.

Settings The following settings can be controlled from the settings menu:

Warn autoload
By default disabled. If enabled, modules that require predicates to be autoloaded are flagged with a warning and the file info window of a module shows the required autoload predicates.

Warn not called
If enabled (default), the file-overview shows an alert icon for files that have predicates that are not called.

3.8 Accessing the IDE from your program

Over the years a collection of IDE components have been developed, each with their own interface. In addition, some of these components require each other and loading IDE components must be on demand to avoid the IDE being part of a saved-state (see `qsave_program/2`). For this reason, access to the IDE will be concentrated on a single interface called `prolog_ide/1`:

prolog_ide(+*Action*)
This predicate ensures the IDE enabling XPCE component is loaded, creates the XPCE class *prolog_ide* and sends *Action* to its one and only instance `\index{@prolog_ide}\objectname{prolog_ide}`. *Action* is one of the following:

open_navigator(+*Directory*)
Open the Prolog Navigator (see section 3.6) in the given *Directory*.

open_debug_status
Open a window to edit spy- and trace-points.

open_query_window
Opens a little window to run Prolog queries from a GUI component.

thread_monitor
Open a graphical window indicating existing threads and their status.

debug_monitor
Open a graphical front-end for the `debug` library that provides an overview of the topics and catches messages.

xref
Open a graphical front-end for the cross-referencer that provides an overview of predicates and their callers.

3.9 Summary of the iDE

The SWI-Prolog development environment consists of a number of interrelated but not (yet) integrated tools. Here is a list of the most important features and tips.

- *Atom completion*
  The console[8] completes a partial atom on the TAB key and shows alternatives on the command Alt-?.

- *Use* `edit/1` *to finding locations*
  The command `edit/1` takes the name of a file, module, predicate or other entity registered through extensions and starts the users preferred editor at the right location.

[8] On Windows this is realised by plwin.exe, on Unix through the GNU readline library, which is included automatically when found by `configure`.

- *Select editor*
  External editors are selected using the `EDITOR` environment variable, by setting the prolog flag `editor` or by defining the hook prolog_edit:edit_source/1.

- *Update Prolog after editing*
  Using `make/0`, all files you have edited are re-loaded.

- *PceEmacs*
  Offers syntax-highlighting and checking based on real-time parsing of the editor's buffer, layout-support and navigation support.

- *Using the graphical debugger*
  The predicates `guitracer/0` and `noguitracer/0` switch between traditional text-based and window-based debugging. The tracer is activated using the `trace/0`, `spy/1` or menu-items from PceEmacs or the PrologNavigator.

- *The Prolog Navigator*
  Shows the file-structure and structure inside the file. It allows for loading files, editing, setting spy-points, etc.

4 Built-in predicates

4.1 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'. Finally, arguments may have the ':' specifier, which implies the argument is module-sensitive. Normally the argument is a *callable* term referring to a predicate in a specific module. See section 5 for more information on module-handing.

4.2 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. 8-bit character sets have been allowed for a long time, 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.

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

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 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 between 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.

4.3 Loading Prolog source files

This section deals with loading Prolog source-files. A Prolog source file is a plain text 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 5 for details.

An 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 `ordsets.pl` (containing predicates for manipulating ordered sets).

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

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 predicates `term_expansion/2` and `goal_expansion/2`. Transformations by `term_expansion/2` 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 except for `include/1`. It currently supports a subset of the options of Quintus `load_files/2`. *Files* is either a single source-file, or a list of source-files. The specification for a source-file is handed to `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:

autoload(*Bool*)
: If `true` (default `false`), indicate this load is a *demand* load. This implies that, depending on the setting of the prolog-flag `verbose_autoload` the load-action is printed at level `informational` or `silent`. See also `print_message/2` and `current_prolog_flag/2`.

derived_from(*File*)
: Indicate that the loaded file is derived from *File*. Used by `make/0` to time-check and load the original file rather than the derived file.

encoding(*Encoding*)
: Specify the way characters are encoded in the file. Default is taken from the prolog flag `encoding`. See section 2.17.1 for details.

expand(*Bool*)
: If `true`, run the filenames through `expand_file_name/2` and load the returned files. Default is `false`, except for `consult/1` which is intended for interactive use. Flexible location of files is defined by `file_search_path/2`.

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.

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

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

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

qcompile(*Bool*)
If this call appears in a directive of a file that is compiled into Quick Load Format using `qcompile/1` and this flag is `true`, the contents of the argument files are included in the `.qlf` file instead of the loading directive.

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. This option writes the prolog flag `verbose_load` with the negation of *Bool*.

stream(*Input*)
This SWI-Prolog extension compiles the data from the stream *Input*. If this option is used, *Files* must be a single atom which is used to identify the source-location of the loaded clauses as well as remove all clauses if the data is re-consulted.

This option is added to allow compiling from non-file locations such as databases, the web, the *user* (see `consult/1`) or other servers.

The `load_files/2` predicate can be hooked to load other data or data from other objects than files. See `prolog_load_file/2` for a description and `http_load` for an example.

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 Unix shell 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
?- [user].
```

The predicate `consult/1` is equivalent to load_files(Files, []), except for handling the special file `user`, which reads clauses from the terminal. See also the `stream`(*Input*) option of `load_files/2`.

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(not_loaded)]).[3]

include(+*File*)
Pretend the terms in *File* are in the source-file in which `:- include(File)` appears. The include construct is only honoured 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 it does not exist. This will flag the predicate as 'undefined'. See also `check/0` and `autoload/0`.

encoding(+*Encoding*)
This directive can appear anywhere in a source file to define how characters are encoded in the remainder of the file. It can be used in files that are encoded with a superset of US-ASCII, currently UTF-8 and ISO Latin-1. See also section 2.17.1.

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. In addition, `make/0` updates the autoload indices (see section 2.13) and runs `list_undefined/0` from the `check` library to report on undefined predicates.

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.

[3] On older versions the condition used to be if(changed). Poor time management on some machines or due to copying often caused problems. The `make/0` predicate deals with updating the running system after changing the source code.

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]`.

The prolog-flag `verbose_file_search` can be set to `true` to help debugging Prolog's search for files.

expand_file_search_path(+*Spec, -Path***)**
Unifies *Path* with 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***)**
True 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
`source`	File loaded. Returns the original Prolog file when loading a `.qlf` file. Compatible to SICStus Prolog.
`file`	Currently equivalent to `file`. In future versions it may report a different values for files being loaded using `include/1`.
`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)`. See also `stream_position_data/3`.

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 and multifile predicate, normally not defined. When defined by the user all terms read during consulting 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 5 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. If *Goal* is of the form *Module*:*Goal* where *Module* is instantiated, goal_expansion/2 is called on *Goal* using rules from module *Module* followed by user.

Only goals appearing 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, once/1, ignore/1, findall/3, bagof/3, setof/3 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 run 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 run from PL_cleanup(), which is called when the system halts. The hooks are run in the reverse order they were registered (FIFO). Success or failure executing a hook is ignored. If the hook raises an exception this is printed using print_message/2. An attempt to call halt/[0,1] from a hook is ignored.

initialization(+*Goal*)
Call *Goal* and register it using at_initialization/1. Directives that do other things than 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
True if the system is compiling source files with the -c option or qcompile/1 into an intermediate code file. Can be used to perform conditional code optimisations in term_expansion/2 (see also the -O option) or to omit execution of directives during compilation.

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
```

4.3.1 Loading files, active code and threads

Traditionally, Prolog environments allow for reloading files holding currently active code. In particular, the following sequence is valid use of the development environment:

- Trace a goal
- Find unexpected behaviour of a predicate
- Enter a *break* using the **b** command
- Fix the sources and reload them using `make/0`
- Exit the break, *retry* using the **r** command

Goals running during the reload keep running on the old definition, while new goals use the reloaded definition, which is why the *retry* must be used *after* the reload. This implies that clauses of predicates that are active during the reload cannot be reclaimed. Normally a small amount of dead clauses should not be an issue during development. Such clauses can be reclaimed with `garbage_collect_clauses/0`.

garbage_collect_clauses
leanup all *dirty* predicates, where dirty predicates are defined to be predicates that have both old and new definitions due to reloading a source file while the predicate was active. Of course, predicates that are active using `garbage_collect_clauses/0` cannot be reclaimed and remain *dirty*. Predicate are -like atoms- shared resources and therefore all threads are suspended during the execution of this predicate.

Threads and reloading running code

As of version 5.5.30, there is basic thread-safety for reloading source files while other threads are executing code defined in these source files. Reloading a file freezes all threads after marking the active predicates originating from the file being reloaded. The threads are resumed after the file has been loaded. In addition, after completing loading the outermost file the system runs `garbage_collect_clauses/0`.

What does that mean? Unfortunately it does *not* mean we can 'hot-swap' modules. Consider the case where thread A is executing the recursive predicate P. We 'fix' P and reload. The already running goals for P continue to run the old definition, but new recursive calls will use the new definition! Many similar cases can be constructed with dependent predicates.

It provides some basic security for reloading files in multi-threaded applications during development. In the above scenarios the system does not crash uncontrolled, but behaves like any broken program: it may return the wrong bindings, wrong truth value or raise an exception.

Future versions may have an 'update now' facility. Such as facility can be implemented on top of the *logical update view*. It would allow threads to do a controlled update between processing independent jobs.

4.3.2 Quick load files

SWI-Prolog supports compilation of individual or multiple Prolog source files 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 source files 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 4.3. 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 `.pl` file otherwise.

qcompile(+*File*)

Takes a single file specification like `consult/1` (i.e., accepts constructs like `library(LibFile)` and, in addition to the normal compilation, 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)`', '`:- [+File]`' or `:- load_files(+File, [qcompile(true), ...])` 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.

Conditional execution or optimisation may test the predicate `compiling/0`.

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

4.4 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 asked to select from the locations found. Finally, prolog_edit:edit_source/1 is used to invoke the user's preferred editor. Typically, `edit/1` can be handed the name of a predicate, module, basename of a file, XPCE class, XPCE method, etc.

edit
Edit the 'default' file using `edit/1`. The default file is the file loaded with the command-line option `-s` or, in windows, the file loaded by double-clicking from the Windows shell.

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-specified objects.

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

If it fails, `edit/1` uses its internal defaults, which are defined by the prolog-flag `editor` and/or the environment variable `EDITOR`. The following rules apply. If the prolog-flag `editor` is of the format `$`⟨*name*⟩, the editor is determined by the environment variable ⟨*name*⟩. Else, if this flag is `pce_emacs` or `built_in` *and* XPCE is loaded or can be loaded, the built-in Emacs clone is used. Else, if the environment `EDITOR` is set, this editor is used. Finally, `vi` is used as default on Unix systems and `notepad` on Windows.

See the default user preferences file `dotfiles/dotplrc` for examples.

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 `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 losing 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`' or simply ⟨*Head*⟩. Variables in the clause are written as A, B, ... Singleton variables are written as _. See also `portray_clause/2`.

portray_clause(+*Stream, +Clause*)
Pretty print a clause to *Stream*. See `portray_clause/1` for details.

4.5 Verify Type of a Term

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

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

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

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

rational(+*Term*)
True if *Term* is bound to a rational number. Rational numbers include integers.

rational(+*Term, -Numerator, -Denominator*)
True if *Term* is a rational number with given *Numerator* and *Denominator*. The *Numerator* and *Denominator* are in canonical form, which means *Denominator* is a positive integer and there are no common divisors between *Numerator* and *Denominator*.

number(+*Term*)
True if *Term* is bound to an integer or floating point number.[4]

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

string(+*Term*)
True if *Term* is bound to a string. Note that string here refers to the built-in atomic type string as described in section 4.23, Text in double quotes such as `"hello"` creates a *list* of *character codes*. We illustrate the issues in the example queries below.

```
?- write("hello").
[104, 101, 108, 108, 111]
?- string("hello").
No
?- is_list("hello").
Yes
```

atomic(+*Term*)
True if *Term* is bound to an atom, string, integer or floating point number. Note that string refers to the built-in type. See `string/1`. Strings in the classical Prolog sense are lists and therefore compound.

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

callable(+*Term*)
True 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*)
True if *Term* holds no free variables.

cyclic_term(+*Term*)
True if *Term* contains cycles, i.e. is an infinite term. See also `acyclic_term/1` and section 2.16.[5]

acyclic_term(+*Term*)
True if *Term* does not contain cycles, i.e. can be processed recursively in finite time. See also `cyclic_term/1` and section 2.16.

[4] As rational numbers are not atomic in the current implementation and we do not want to break the rule that `number/1` implies `atomic/1`, `number/1` fails on rational numbers. This will change if rational numbers become atomic.

[5] The predicates `cyclic_term/1` and `acyclic_term/1` are compatible to SICStus Prolog. Some Prolog systems supporting cyclic terms use `is_cyclic/1`.

4.6 Comparison and Unification or Terms

4.6.1 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* < *Numbers* < *Atoms* < *Strings* < *Comound Terms*[6]
2. Variables are sorted by address. Attaching attributes (see section 6.1) does not affect the ordering.
3. *Atoms* are compared alphabetically.
4. *Strings* are compared alphabetically.
5. *Numbers* are compared by value. Integers and floats are treated identically. If the prolog_flag (see `current_prolog_flag/2`) `iso` is defined, all floating point numbers precede all integers.
6. *Compound* terms are first checked on their arity, then on their functor-name (alphabetically) and finally recursively on their arguments, leftmost argument first.

+*Term1* **==** +*Term2*
: True 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*. True 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 infinitely 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`.

[6] Strings might be considered atoms in future versions. See also section 4.23

+Term1 **=@=** *+Term2*
: True 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
```

The predicates `=@=/2` and `\=@=/2` are cycle-safe. Attributed variables are considered structurally equal iff their attributes are structurally equal.

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

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

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

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

+Term1 **@>=** *+Term2*
: True 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.

?=(*@Term1, @Term2*)
: Decide whether the equality of *Term1* and *Term2* can be compared safely, i.e. whether the result of `Term1 == Term2` can change due to further instantiation of either term. It is defined as by `?=(A,B) :- (A==B ; A \= B), !.` See also `dif/2`.

unifiable(*@X, @Y, -Unifier*)
: If *X* and *Y* can unify, unify *Unifier* with a list of *Var* = *Value*, representing the bindings required to make *X* and *Y* equivalent.[7] This predicate can handle cyclic terms. Attributed variables are handles as normal variables. Associated hooks are *not* executed.

4.7 Control Predicates

The predicates of this section implement control structures. Normally the constructs in this section, except for `repeat/0`, are translated by the compiler. Please note that complex goals passed as arguments to meta-predicates such as `findall/3` below cause the goal to be compiled to a temporary

[7] This predicate was introduced for the implementation of `dif/2` and `when/2` after discussion with Tom Schrijvers and Bart Demoen. None of us is really happy with the name and therefore suggestions for a new name are welcome.

location before execution. It is faster to define a sub-predicate (i.e. `one_character_atom/1` in the example below) and make a call to this simple predicate.

```
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.[8]

```
t1 :- (a, !, fail ; b).         % cuts a/0 and t1/0
t2 :- (a -> b, !  ; c).         % cuts b/0 and t2/0
t3 :- call((a, !, fail ; b)).   % cuts a/0
t4 :- \+(a, !, fail ; b).       % cuts a/0
```

+Goal1 **,** *+Goal2*
: Conjunction. True 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`.

+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 choice-point of the predicate as a whole (due to multiple clauses) is **not** destroyed. The combination `;/2` and `->/2` acts as if defines by:

[8] Up to version 4.0.6, the sequence X=!, X acted as a true cut. This feature has been deleted for ISO compliance.

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

Please note that (If −> Then) acts as (If −> Then ; **fail**), making the construct *fail* if the condition fails. This unusual semantics is part of the ISO and all de-facto Prolog standards.

+*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*.

The construct *A* ⋆−> *B*, i.e. without an *Else* branch, is translated as the normal conjunction *A*, *B*.[9]

\+ +*Goal*

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

4.8 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`.

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*)

True if *Goal* cannot be proven. Retained for compatibility only. New code should use `\+/1`.

[9] BUG: The decompiler implemented by `clause/2` returns this construct as a normal conjunction too.

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 may 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 infinitely 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 *iterative deepening*. It was implemented after discussion with Steve Moyle `smoyle@ermine.ox.ac.uk`.

call_cleanup(*:Goal, +Catcher, :Cleanup*)
Calls *Goal*. If *Goal* is completely finished, either by deterministic success, failure, its choice-point being cut or raising an exception and *Catcher* unifies to the termination code (see below), *Cleanup* is called. Success or failure of *Cleanup* is ignored and possible choice-points it created are destroyed (as `once/1`). If cleanup throws an exception this is executed as normal.[10]

[10] BUG: During the execution of *Cleanup*, garbage collection and stack-shifts are disabled.

Catcher is unified with a term describing how the call has finished. If this unification fails, *Cleanup* is *not* called.

exit
Goal succeeded without leaving any choice-points.

fail
Goal failed.

!
Goal succeeded with choice-points and these are now discarded by the execution of a cut (or other pruning of the search tree such as if-then-else).

exception(*Exception***)**
Goal raised the given *Exception*.

Typical use of this predicate is cleanup of permanent data storage required to execute *Goal*, close file-descriptors, etc. The example below provides a non-deterministic search for a term in a file, closing the stream as needed.

```
term_in_file(Term, File) :-
        open(File, read, In),
        call_cleanup(term_in_stream(Term, In), _, close(In)).

term_in_stream(Term, In) :-
        repeat,
        read(In, T),
        (   T == end_of_file
        ->  !, fail
        ;   T = Term
        ).
```

Note that this predicate is impossible to implement in Prolog other then reading all terms into a list, close the file and call `member/2` because without `call_cleanup/3` there is no way to gain control if the choice-point left by repeat is killed by a cut.

This predicate is a SWI-Prolog extension. See also `call_cleanup/2` for compatibility to other Prolog implementations.

call_cleanup(*:Goal, :Cleanup***)**
This predicate is equivalent to `call_cleanup`(*Goal, _, Cleanup*), calling *Cleanup* regardless of the reason for termination and without providing information. This predicate provides compatibility to a number of other Prolog implementations.

4.9 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 (4.0.6), most of the built-in predicates generate exceptions, but some obscure predicates merely print a message, start the debugger and fail, which was the normal behaviour before the introduction of exceptions.

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 choice-points generated by *Goal* are cut, the system backtracks to the start of `catch/3` while preserving the thrown exception term 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 is much slower, especially if the exception-term is large due to the copying thereof.

throw(*+Exception*)

Raise an exception. The system looks for the innermost `catch/3` ancestor for which *Exception* unifies with the *Catcher* argument of the `catch/3` call. See `catch/3` for details.

ISO demands `throw/1` to make a copy of *Exception*, walk up the stack to a `catch/3` call, backtrack and try to unify the copy of *Exception* with *Catcher*. SWI-Prolog delays making a copy of *Exception* and backtracking until it actually found a matching `catch/3` goal. The advantage is that we can start the debugger at the first possible location while preserving the entire exception context if there is no matching `catch/3` goal. This approach can lead to different behaviour if *Goal* and *Catcher* of `catch/3` call share variables. We assume this to be highly unlikely and could not think of a scenario where this is useful.[11]

If an exception is raised in a callback from C (see chapter 9) and not caught in the same callback, PL_next_solution() fails and the exception context can be retrieved using PL_exception().

4.9.1 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 top-level 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 top-level 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.

[11] I'd like to acknowledge Bart Demoen for his clarifications on these matters.

- *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 advised to switch on debug mode using the predicate `debug/0`.

4.9.2 The exception term

Built-in 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.

4.9.3 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`, `banner`, `warning`, `error`, `help` or `silent`. A human-readable message is printed to the stream `user_error`.

If the prolog flag (see `current_prolog_flag/2`) `verbose` is `silent`, messages with *Kind* `informational`, or `banner` are treated as silent. See `-q`.

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. If you need to report errors from your own predicates, we advise you to stick to the existing error terms if you can; but should you need to invent new ones, you can define corresponding error messages by asserting clauses for prolog:message. You will need to declare the predicate as multifile.

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 4.23. Primarily intended to write messages to Windows in XPCE (see section 1.5) or other GUI environments.

4.10 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 9.6.12).

Signals are used to handle internal errors (execution of a non-existing CPU instruction, 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 functionality 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 4.9. `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 9):

```
#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 Prolog below. This will run *Goal* just as `once/1`, throwing the exception `error`(*signal(alrm, _), _*) if a timeout occurs.[12]

```
:- load_foreign_library(alarm).

:- on_signal(alrm, _, throw).

:- module_transparent
        call_with_time_limit/2.

call_with_time_limit(MaxTime, Goal) :-
        alarm(MaxTime),
        call_cleanup(Goal, _, alarm(0)), !.
```

The signal names are defined by the POSIX standard 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`).

4.10.1 Notes on signal handling

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

- *Portability*
  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. For the multi-threaded versions this is even worse: signals can easily violate thread synchronisation consistency.

  The C-interface described in section 9.6.12 provides the option `PL_SIGSYNC` for registering a signal handler that delays delivery of signals to a safe point. Unfortunately this may cause signals to be delayed for a long time if Prolog is executing foreign code.

[12] Note that `call_with_time_limit/2` is defined in `time`, part of the 'clib' package. The version provided in the library runs on POSIX systems as well as MS-Windows and can schedule multiple concurrent alarms.

- *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.

4.11 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 possibility 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.

4.12 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 built-in 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]
```

4.13 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 advised 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.[13]

flag(+*Key, -Old, +New*)
Key is an atom, integer or term. As with the recorded database, if *Key* is a term, only the name and arity are used to locate the flag. 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 fast mechanism for storing simple facts in the database. The flag database is shared between threads and updates are atomic, making it suitable for generating unique integer counters.[14]

[13] 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.

[14] The `flag/3` predicate is not portable. Non-backtrackable global variables (`nb_setval/2`) and non-backtrackable assignment (`nb_setarg/3`) are more widely recognised special-purpose alternatives for non-backtrackable and/or global state.

4.13.1 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'.[15] 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.

4.13.2 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 and versions of SWI-Prolog. The `hash_term/2` predicate is cycle-safe.

4.14 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.

dynamic +*Name/+Arity, ...*

Informs the interpreter that the definition of the predicate(s) may change during execution (using `assert/1` and/or `retract/1`). In the multi-threaded version, the clauses of dynamic

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

predicates are shared between the threads. The directive `thread_local/1` provides an alternative where each threads has its own clause-list for the predicate. Dynamic predicates can be turned into static ones using `compile_predicates/1`.

compile_predicates(*:ListOfNameArity*)
Compile a list of specified dynamic predicates (see `dynamic/1` and `assert/1`) into normal static predicates. This call tells the Prolog environment the definition will not change anymore and further calls to `assert/1` or `retract/1` on the named predicates raise a permission error. This predicate is designed to deal with parts of the program that is generated at runtime but does not change during the remainder of the program execution.[16]

multifile *+Name/+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 *+Name/+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 compound terms are considered. Compound terms are indexed on the combination of their name and arity. 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.

Indexing as specified by this predicate uses a quick but linear scan. Without explicit specification the system uses an algorithm depending on the structure of the first argument and number of clauses, In particular, for predicates that can be indexed on the first argument and have many clauses, the system will use an automatically resizing hash-table to provide access time independent from the number of clauses.[17] 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)).
```

[16] The specification of this predicate is from Richard O'Keefe. The implementation is allowed to optimise the predicate. This is not yet implemented. In multi-threaded Prolog however, static code runs faster as it does not require synchronisation. This is particularly true on SMP hardware.

[17] SWI-Prolog indexing is no longer state-of-the-art. Better schemas for multi-argument as well as indexing *inside* compound terms are known. We hope to integrate this in future versions.

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

Note that this type of indexing makes selecting clauses much faster but remains *linear* with respect to the number of clauses, while hashing as described with `hash/1` provides constant access time. See also `hash/1` and `hash_term/2`.

hash(+*Head*)
Index the given predicate by hashing on the first argument. This is done by default on any predicate with more than 5 clauses having a first argument that can be indexed and at most two that can not be indexed. On dynamic predicates the hash-table is resized as the number of clauses grows, providing roughly constant-time access regardless of the number of clauses predicates that can be indexed on the first argument. See also `index/1`, `hash_term/2` and `predicate_property/2`.

4.15 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_blob(*?Blob, ?Type*)
Examine the type or enumerate blobs of the given *Type*. Typed blobs are supported through the foreign language interface for storing arbitrary BLOBS (Binary Large Object) or handles to external entities. See section 9.6.6 for details.

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.

current_predicate(*:Name/Arity*)
ISO compliant implementation of `current_predicate/2`. Unlike `current_predicate/2`, the current implementation of `current_predicate/1` does not consider predicates that can be autoloaded 'current'.

predicate_property(*:Head, ?Property*)

True 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:

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.

dynamic
Is true if `assert/1` and `retract/1` may be used to modify the predicate. This property is set using `dynamic/1`.

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*.

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

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

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.

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.

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`.

multifile
Is true there may be multiple (or no) file providing clauses for the predicate. This property is set using `multifile/1`.

nodebug
Details of the predicate are not shown by the debugger. This is the default for built-in predicates. User predicates can be compiled this way using the Prolog flag `generate_debug_info`.

notrace
Do not show ports of this predicate in the debugger.

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

thread_local
If true (only possible on the multi-threaded version) each thread has its own clauses for the predicate. This property is set using `thread_local/1`.

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

undefined
Is true if a procedure definition block for the predicate exists, but there are no clauses for it and it is not declared dynamic or multifile. 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.

volatile
If true, the clauses are not saved into a saved-state by `qsave_program/[1,2]`. This property is set using `volatile/1`.

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*)
True if *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 source file 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.

4.16 Input and output

SWI-Prolog provides two different packages for input and output. The native I/O system is based on the ISO standard predicates `open/3`, `close/1` and friends.[18] Being more widely portable and equipped with a clearer and more robust specification, new code is encouraged to use these predicates for manipulation of I/O streams.

Section 4.16.2 describes `tell/1`, `see/1` and friends, providing I/O in the spirit of the outdated Edinburgh standard. These predicates are layered on top of the ISO predicates. Both packages are fully integrated; the user may switch freely between them.

4.16.1 ISO Input and Output Streams

The predicates described in this section provide ISO compliant I/O, where 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.

This schema is not vulnerable to filename and stream ambiguities as well as changes to the working directory. New code is advised to use these predicates to manage input and output streams.

open(+*SrcDest, +Mode, -Stream, +Options*)
ISO compliant predicate to open a stream. *SrcDes* is either an atom, specifying a file, or a term '`pipe(`*Command*`)`', 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. *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.[19] The *Options* list can contain the following options:

[18] Actually based on Quintus Prolog, providing this interface before the ISO standard existed.
[19] New code should use the `alias`(*Alias*) option for compatibility to the ISO standard

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. See also the option `encoding`.

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),
        ...
```

encoding(*Encoding*)
Define the encoding used for reading and writing text to this stream. The default encoding for type `text` is derived from the Prolog flag `encoding`. For `binary` streams the default encoding is `octet`. For details on encoding issues, see section 2.17.1.

bom(*Bool*)
Check for a BOM (*Byte Order Marker*) or write one. If omitted, the default is `true` for mode `read` and `false` for mode `write`. See also `stream_property/2` and especially section 2.17.1 for a discussion on this feature.

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 `full` (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).

lock(*LockingMode*)
Try to obtain a lock on the open file. Default is `none`, which does not lock the file. The value `read` or `shared` means other processes may read the file, but not write it. The value `write` or `exclusive` means no other process may read or write the file.

Locks are acquired through the POSIX function fcntl() using the command `F_SETLKW`, which makes a blocked call wait for the lock to be released. Please note that fcntl() locks are *advisory* and therefore only other applications using the same advisory locks honour your lock. As there are many issues around locking in Unix, especially related to NFS (network file system), please study the fcntl() manual page before trusting your locks!

The `lock` option is a SWI-Prolog extension.

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:

alias(*Atom*)
If *Atom* is bound, test of the stream has the specified alias. Otherwise unify *Atom* with the first alias of the stream.[20]

buffer(*Buffering*)
SWI-Prolog extension to query the buffering mode of this stream. *Buffering* is one of `full`, `line` or `false`. See also `open/4`.

bom(*Bool*)
If present and `true`, a BOM (*Byte Order Mark*) was detected while opening the file for reading or a BOM was written while opening the stream. See section 2.17.1 for details.

encoding(*Encoding*)
Query the encoding used for text. See section 2.17.1 for an overview of wide character and encoding issues in SWI-Prolog.

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.

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

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`.

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

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

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`.

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

position(*Term*)
Unify *Term* with the current stream-position. A stream-position is an opaque term whose fields can be extracted using `stream_position_data/3`. See also `set_stream_position/2`.

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.

representation_errors(*Mode*)
Determines behaviour of character output if the stream cannot represent a character. For example, an ISO Latin-1 stream cannot represent cyrillic characters. The behaviour is one of `error` (throw and I/O error exception), `prolog` (write `\...\` escape code or `xml` (write `&#...;` XML character entity). The initial mode is `prolog` for the user streams and `error` for all other streams. See also section 2.17.1 and `set_stream/2`.

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

tty(*Bool*)
This property is reported with *Bool* equals `true` if the stream is associated with a terminal. See also `set_stream/2`.

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

is_stream(+*Term*)
True if *Term* is a stream name or valid stream handle. This predicate realises a safe test for the existence of a stream alias or handle.

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`.

stream_position_data(*?Field, +Position, -Data*)
Extracts information from the opaque stream position term as returned by `stream_property/2` requesting the `position`(*Position*) property. *Field* is one of `line_count`, `line_position`, `char_count` or `byte_count`. See also `line_count/2`, `line_position/2`, `character_count/2` and `byte_count/2`.[21]

[21] Introduced in version 5.6.4 after extending the position term with a byte-count. Compatible with SICStus Prolog.

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 repositioned, 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_property/2` and `set_stream_position/2`. Please note that the use of `seek/4` on non-binary files (see `open/4`) is of limited use as the referred positions are byte offsets.

set_stream(+*Stream, +Attribute*)
Modify an attribute of an existing stream. *Attribute* specifies the stream property to set. See also `stream_property/2` and `open/4`.

alias(*AliasName*)
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`.

buffer(*Buffering*)
Set the buffering mode of an already created stream. Buffering is one of `full`, `line` or `false`.

close_on_abort(*Bool*)
Determine whether or not the stream is closed by `abort/0`. By default streams are closed.

encoding(*Atom*)
Defines the mapping between bytes and character codes used for the stream. See section 2.17.1 for supported encodings.

eof_action(*Action*)
Set end-of-file handling to one of `eof_code`, `reset` or `error`.

timeout(*Seconds*)
This option can be used to make streams generate an exception if it takes longer than *Seconds* before any new data arrives at the stream. The value *infinite* (default) makes the stream block indefinitely. Like `wait_for_input/3`, this call only applies to streams that support the select() system call. For further information about timeout handling, see `wait_for_input/3`. The exception is of the form

`error(timeout_error(`*read, Stream*`), _)`

record_position(*Bool*)
Do/do not record the line-count and line-position (see `line_count/2` and `line_position/2`).

representation_errors(*Mode*)
Change the behaviour when writing characters to the stream that cannot be represented by the encoding. See also `stream_property/2` and section 2.17.1.

file_name(*FileName*)
Set the file name associated to this stream. This call can be used to set the file for error-locations if *Stream* corresponds to *FileName* and is not obtained by opening the file directly but, for example, through a network service.

tty(*Bool*)
Modify whether Prolog thinks there is a terminal (i.e. human interaction) connected to this stream. On Unix systems the initial value comes from isatty(). On Windows, the initial user streams are supposed to be associated to a terminal. See also `stream_property/2`.

set_prolog_IO(+*In*, +*Out*, +*Error*)
Prepare the given streams for interactive behaviour normally associated to the terminal. *In* becomes the `user_input` and `current_input` of the calling thread. *Out* becomes `user_output` and `current_output`. If *Error* equals *Out* an unbuffered stream is associated to the same destination and linked to `user_error`. Otherwise *Error* is used for `user_error`. Output buffering for *Out* is set to `line` and buffering on *Error* is disabled. See also `prolog/0` and `set_stream/2`. The *clib* package provides the library `prolog_server` creating a TCP/IP server for creating an interactive session to Prolog.

4.16.2 Edinburgh-style I/O

The package for implicit input and output destination is (almost) compatible to Edinburgh 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.

Source and destination are either a file, `user`, or a term 'pipe(*Command*)'. The reserved stream name `user` refers to the terminal.[22] 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).         % Start writing to the terminal.
?- tell(pipe(lpr)).    % Start writing to the printer.
```

Another example of using the `pipe/1` construct is shown below.[23] 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).
```

[22] The ISO I/O layer uses `user_input`, `user_output` and `user_error`.

[23] As of version 5.3.15, the pipe construct is supported in the MS-Windows version, both for `plcon.exe` and `plwin.exe`. The implementation uses code from the LUA programming language (`http://www.lua.org`).

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

Compatibility notes

Unlike Edinburgh Prolog systems, `telling/1` and `seeing/1` do not return the filename of the current input/output, but the stream-identifier, to ensure the design pattern below works under all circumstances.[24]

```
        ...,
        telling(Old), tell(x),
        ...,
        told, tell(Old),
        ...,
```

The predicates `tell/1` and `see/1` first check for `user`, the `pipe`(*command*) and a stream-handle. Otherwise, if the argument is an atom it is first compared to open streams associated to a file with *exactly* the same name. If such a stream, created using `tell/1` or `see/1` exists, output (input) is switch to the open stream. Otherwise a file with the specified name is opened.

The behaviour is compatible to Edinburgh Prolog. This is not without problems. Changing directory, non-file streams, multiple names referring to the same file easily lead to unexpected behaviour. New code, especially when managing multiple I/O channels should consider using the ISO I/O predicates defined in section 4.16.1.

see(+*SrcDest*)
: Open *SrcDest* for reading and make it the current input (see `set_input/1`). If *SrcDest* is a stream-handle, just makes this stream the current input. See the introduction of section 4.16.2 for details.

tell(+*SrcDest*)
: Open *SrcDest* for writing and make it the current output (see `set_output/1`). If *SrcDest* is a stream-handle, just makes this stream the current output. See the introduction of section 4.16.2 for details.

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*)
: Same as `current_input/1`, except that `user` is returned if the current input is the stream `user_input` to improve compatibility with traditional Edinburgh I/O. See the introduction of section 4.16.2 for details.

[24] Filenames can be ambiguous and SWI-Prolog streams can refer to much more than just files.

telling(*?SrcDest*)
Same as `current_output/1`, except that `user` is returned if the current output is the stream `user_output` to improve compatibility with traditional Edinburgh I/O. See the introduction of section 4.16.2 for details.

seen
Close the current input stream. The new input stream becomes *user_input*.

told
Close the current output stream. The new output stream becomes *user_output*.

4.16.3 Switching Between Edinburgh and ISO 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*. See also `with_output_to/2`.

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.

4.16.4 Write onto atoms, code-lists, etc.

with_output_to(+*Output, :Goal*)
Run *Goal* as `once/1`, while characters written to the current output is sent to *Output*. The predicate is SWI-Prolog specific, inspired by various posts to the mailinglist. It provides a flexible replacement for predicates such as `sformat/3`, `swritef/3`, `term_to_atom/2`, `atom_number/2` converting numbers to atoms, etc. The predicate `format/3` accepts the same terms as output argument.

Applications should generally avoid creating atoms by breaking and concatenating other atoms as the creation of large numbers of intermediate atoms generally leads to poor performance, even more so in multi-threaded applications. This predicate supports creating difference-lists from character data efficiently. The example below defines the DCG rule term//1 to insert a term in the output:

```
term(Term, In, Tail) :-
        with_output_to(codes(In, Tail), write(Term)).

?- phrase(term(hello), X).

X = [104, 101, 108, 108, 111]
```

A Stream handle or alias
Temporary switch current output to the given stream. Redirection using `with_output_to/2` guarantees the original output is restored, also if *Goal* fails or raises an exception. See also `call_cleanup/2`.

atom(*-Atom*)
Create an atom from the emitted characters. Please note the remark above.

string(*-String*)
Create a string-object as defined in section 4.23.

codes(*-Codes*)
Create a list of character codes from the emitted characters, similar to `atom_codes/2`.

codes(*-Codes, -Tail*)
Create a list of character codes as a difference-list.

chars(*-Chars*)
Create a list of one-character-atoms codes from the emitted characters, similar to `atom_chars/2`.

chars(*-Chars, -Tail*)
Create a list of one-character-atoms as a difference-list.

4.17 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 `infinite`, `wait_for_input/3` waits indefinitely.[25]

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).
```

This predicate relies on the select() call on most operating systems. On Unix this call is implemented for any stream referring to a file-handle, which implies all OS-based streams: sockets, terminals, pipes, etc. On non-Unix systems select() is generally only implemented for socket-based streams. See also `socket` from the `clib` package.

Note that `wait_for_input/3` returns streams that have data waiting. This does not mean you can, for example, call `read/2` on the stream without blocking as the stream might hold an incomplete term. The predicate `set_stream/2` using the option `timeout`(*Seconds*) can be used to make the stream generate an exception if no new data arrives for within the timeout. Suppose two processes communicate by exchanging Prolog terms. The following code makes the server immune for clients that write an incomplete term:

[25] For compatibility reasons, a *Timeout* value of 0 (integer) also waits indefinitely. To call select() without giving up the CPU pass the float 0.0. If compatibility with versions older than 5.1.3 is desired pass the float value 1e-7.

```
...,
tcp_accept(Server, Socket, _Peer),
tcp_open(Socket, In, Out),
set_stream(In, timeout(10)),
catch(read(In, Term), _, (close(Out), close(In), fail)),
...,
```

byte_count(+*Stream, -Count*)
Byte-position in *Stream*. For binary streams this is the same as `character_count/2`. For text files the number may be different due to multi-byte encodings or additional record separators (such as Control-M in Windows).

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*.[26]

With the introduction of exception-handling, it is advised to use `catch/3` to catch possibly file-errors and act accordingly. Note that if *fileerrors* is `off`, *no* exception is generated.

4.18 Primitive character I/O

See section 4.2 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 a character code or an atom of one character. Depreciated. New code should use `put_char/1` or `put_code/1`.

[26] 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(+*Stream, +Char*)
Write *Char* to *Stream*. See `put/1` for details.

put_byte(+*Byte*)
Write a single byte to the output. *Byte* must be an integer between 0 and 255.

put_byte(+*Stream, +Byte*)
Write a single byte to a stream. *Byte* must be an integer between 0 and 255.

put_char(+*Char*)
Write a character to the current output, obeying the encoding defined for the current output stream. Note that this may raise an exception if the encoding of *Stream* cannot represent *Char*.

put_char(+*Stream, +Char*)
Write a character to *Stream*, obeying the encoding defined for *Stream*. Note that this may raise an exception if the encoding of *Stream* cannot represent *Char*.

put_code(+*Code*)
Similar to `put_char/1`, but using a *character code*. *Code* is a non-negative integer. Note that this may raise an exception if the encoding of *Stream* cannot represent *Code*.

put_code(+*Stream, +Code*)
Same as `put_code/1` but directing *Code* to *Stream*.

tab(+*Amount*)
Writes *Amount* spaces on the current output stream. *Amount* should be an expression that evaluates to a positive integer (see section 4.26).

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*, returning an integer between 0 and 255.

get_code(-*Code*)
Read the current input stream and unify *Code* with the character code of the next character. *Code* 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_code/1` predicate. Note that Edinburgh prolog didn't support wide characters and therefore technically speaking `get0/1` should have been mapped to `get_byte/1`. The intention of `get0/1` however is to read character codes.

get0(+*Stream, -Char*)
Edinburgh version of the ISO `get_code/2` predicate. See also `get0/1`.

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

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

get_single_char(-*Code*)
Get a single character from input stream 'user' (regardless of the current input stream). Unlike `get_code/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 character 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* codes from stream *StreamIn* to *StreamOut*. Note that the copy is done using the semantics of `get_code/2` and `put_code/2`, taking care of possibly recoding that needs take place between two text files. See section 2.17.1.

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

read_pending_input(+*StreamIn, -Codes, ?Tail*)
Read input pending in the input buffer of *StreamIn* and return it in the difference list *Codes-Tail*. I.e. the available characters codes are used to create the list *Codes* ending in the tail *Tail*. This predicate is intended for efficient unbuffered copying and filtering of input coming from network connections or devices.

The following code fragment realises efficient non-blocking copy of data from an input- to an output stream. The `at_end_of_stream/1` call checks for end-of-stream and fills the input buffer. Note that the use of a `get_code/2` and `put_code/2` based loop requires a `flush_output/1` call after *each* `put_code/2`. The `copy_stream_data/2` does not allow for inspection of the copied data and suffers from the same buffering issues.

```
copy(In, Out) :-
        repeat,
            (   at_end_of_stream(In)
            ->  !
            ;   read_pending_input(In, Chars, []),
                format(Out, '~s', [Chars]),
                flush_output(Out),
                fail
            ).
```

4.19 Term reading and writing

This section describes the basic term reading and writing predicates. The predicates `format/[1,2]` and `writef/2` provide formatted output. Writing to Prolog datastructures such as atoms or code-lists is supported by `with_output_to/2` and `format/3`.

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:

attributes(*Atom*)

Define how attributed variables (see section 6.1) are written. The default is determined by the prolog flag `write_attributes`. Defined values are `ignore` (ignore the attribute), `dots` (write the attributes as `{...}`), `write` (simply hand the attributes recursively to `write_term/2`) and `portray` (hand the attributes to `attr_portray_hook/2`).

backquoted_string(*Bool*)

If `true`, write a string object (see section 4.23) as `` `... ` ``. The default depends on the prolog flag with the same name.

character_escapes(*Bool*)

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(*Bool*)

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.

max_depth(*Integer*)

If the term is nested deeper 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 top-level and debugger to limit screen output. See also the prolog-flags `toplevel_print_options` and `debugger_print_options`.

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(*Bool***)**
If `true`, terms of the format `$VAR(N)`, where ⟨*N*⟩ is a positive integer, will be written as a variable name. If *N* is an atom it is written without quotes. This extension allows for writing variables with user-provided names. The default is `false`. See also `numbervars/3`.

portray(*Bool***)**
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.

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

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`, `quoted` and `numbervars` after `numbervars/4` using the `singletons` option.

Note that due to the use of `numbervars/4`, non-ground terms must be written using a *single* `write_canonical/1` call. This used to be the case anyhow, as garbage collection between multiple calls to one of the write predicates can change the `_G`⟨*NNN*⟩ identity of the variables.

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, see section 2.15.1) 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:

backquoted_string(*Bool*)
If `true`, read `` `...` `` to a string object (see section 4.23). The default depends on the prolog flag with the same name.

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

comments(-*Comments*)
Unify *Comments* with a list of *Position-Comment*, where *Position* is a stream-position object (see `stream_position_data/3`) indicating the start of a comment and *Comment* is a string-object containing the text including delimiters of a comment. It returns all comments from where the `read_term/2` call started upto the end of the term read.

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

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).

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). If *Vars* is the constant `warning`, singleton variables are reported using `print_message/2`.

syntax_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).

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.

term_position(*Pos*)
Unifies *Pos* with the starting position of the term read. *Pos* if of the same format as use by `stream_property/2`.

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

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`.

4.20 Analysing and Constructing Terms

functor(*?Term, ?Functor, ?Arity*)
True 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[27] 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.[28] The predicate `arg/3` fails silently if $Arg = 0$ or $Arg > arity$ and raises the exception `domain_error(not_less_then_zero, Arg)` if $Arg < 0$.

?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:

[27] 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.

[28] The instantiation pattern (-, +, ?) is an extension to 'standard' Prolog. Some systems provide `genarg/3` that covers this pattern.

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

List = [foo, hello, X]

?- Term =.. [baz, foo(1)]

Term = baz(foo(1))
```

numbervars(+*Term*, +*Start*, -*End*)
Unify the free variables of *Term* with a term `$VAR`(*N*), where *N* 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), 0, End).

A = '$VAR'(0)
B = '$VAR'(1)
End = 2
```

See also the `numbervars` option to `write_term/3` and `numbervars/4`.

numbervars(+*Term*, +*Start*, -*End*, +*Options*)
As `numbervars/3`, but providing the following options:

functor_name(+*Atom*)
Name of the functor to use instead of `$VAR`.

attvar(+*Action*)
What to do if an attributed variable is encountered. Options are `skip`, which causes `numbervars/3` to ignore the attributed variable, `bind` which causes it to thread it as a normal variable and assign the next `'$VAR'`(N) term to it or (default) `error` which raises the a `type_error` exception.[29]

singletons(+*Bool*)
If `true` (default `false`), `numbervars/4` does singleton detection. Singleton variables are unified with `'$VAR'('_')`, causing them to be printed as _ by `write_term/2` using the numbervars option. This option is exploited by `portray_clause/2` and `write_canonical/2`.

term_variables(+*Term*, -*List*)
Unify *List* with a list of variables, each sharing with a unique variable of *Term*.[30] See also `term_variables/3`. For example:

[29] This behaviour was decided after a long discussion between David Reitter, Richard O'Keefe, Bart Demoen and Tom Schrijvers.

[30] This predicate used to be called `free_variables/2`. The name `term_variables/2` is more widely used. The old predicate is still available from the library `backcomp`.

```
?- term_variables(a(X, b(Y, X), Z), L).

L = [G367, G366, G371]
X = G367
Y = G366
Z = G371
```

term_variables(+*Term, -List, ?Tail*)
Difference list version of `term_variables/2`. I.e. *Tail* is the tail of the variable-list *List*.

copy_term(+*In, -Out*)
Create a version if *In* with renamed (fresh) variables and unify it to *Out*. Attributed variables (see section 6.1) have their attributed copied. The implementation of `copy_term/2` can deal with infinite trees (cyclic terms). As pure Prolog cannot distinguish a ground term from another ground term with exactly the same structure, ground sub-terms are *shared* between *In* and *Out*. Sharing ground terms does affect `setarg/3`. SWI-Prolog provides `duplicate_term/2` to create a true copy of a term.

4.20.1 Non-logical operations on terms

Prolog is not capable to *modify* instantiated parts of a term. Lacking that capability makes that language much safer, but unfortunately there are problems that suffer severely in terms of time and/or memory usage. Always try hard to avoid the use of these primitives, but they can be a good alternative to using dynamic predicates. See also section 6.3, discussing the use of global variables.

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. See also `nb_setarg/3`.

This predicate may be used for destructive assignment to terms, using them as an extra-logical storage bin. Always try hard to avoid the use of `setarg/3` as it is not supported by many Prolog systems and one has to be very careful about unexpected copying as well as unexpected not copying of terms.

nb_setarg(+*Arg, +Term, +Value*)
Assigns the *Arg*-th argument of the compound term *Term* with the given *Value* as `setarg/3`, but on backtracking the assignment is *not* reversed. If *Term* is not atomic, it is duplicated using `duplicate_term/2`. This predicate uses the same technique as `nb_setval/2`. We therefore refer to the description of `nb_setval/2` for details on non-backtrackable assignment of terms. This predicate is compatible to GNU-Prolog `setarg`(*A,T,V,false*), removing the type-restriction on *Value*. See also `nb_linkarg/3`. Below is an example for counting the number of solutions of a goal. Note that this implementation is thread-safe, reentrant and capable of handling exceptions. Realising these features with a traditional implementation based on assert/retract or `flag/3` is much more complicated.

```
:- module_transparent succeeds_n_times/2.
```

```
succeeds_n_times(Goal, Times) :-
        Counter = counter(0),
        (   Goal,
            arg(1, Counter, N0),
            N is N0 + 1,
            nb_setarg(1, Counter, N),
            fail
        ;   arg(1, Counter, Times)
        ).
```

nb_linkarg(+*Arg, +Term, +Value*)
As `nb_setarg/3`, but like `nb_linkval/2` it does *not* duplicate *Value*. Use with extreme care and consult the documentation of `nb_linkval/2` before use.

duplicate_term(+*In, -Out*)
Version of `copy_term/2` that also copies ground terms and therefore ensures destructive modification using `setarg/3` does not affect the copy. See also `nb_setval/2`, `nb_linkval/2`, `nb_setarg/3` and `nb_linkarg/3`.

4.21 Analysing and Constructing Atoms

These predicates convert between Prolog constants and lists of character codes. The predicates `atom_codes/2`, `number_codes/2` and `name/2` behave the same when converting from a constant to a list of character codes. 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. Up-to 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 character codes. If *Atom* is instantiated, if will be translated into a list of character codes and the result is unified with *String*. If *Atom* is unbound and *String* is a list of character codes, 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 character codes[31].

[31] Up-to version 3.2.x, `atom_chars/2` behaved as the current `atom_codes/2`. The current definition is compliant with the ISO standard

```
?- atom_chars(hello, X).

X = [h, e, l, l, o]
```

char_code(*?Atom, ?Code*)
Convert between character and character code for a single character.[32]

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 rather than one-character atoms. In the mode -, +, both predicates behave identically to improve handling of non-ISO source.

atom_number(*?Atom, ?Number*)
Realises the popular combination of `atom_codes/2` and `number_codes/2` to convert between atom and number (integer or float) in one predicate, avoiding the intermediate list.

name(*?AtomOrInt, ?String*)
String is a list of character codes representing the same text as *Atom*. Each of the arguments may be a variable, but not both. When *String* is bound to an character code 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).

term_to_atom(*?Term, ?Atom*)
True 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.

[32] This is also called `atom_char/2` in older versions of SWI-Prolog as well as some other Prolog implementations. The `atom_char/2` predicate is available from the library `backcomp.pl`

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*)
True if *Atom* is an atom of *Length* characters long. This predicate also works for strings (see section 4.23). If the prolog flag `iso` is *not* set, it also accepts integers and floats, expressing the number of characters output when given to `write/1` as well as code-lists and character-lists, expressing the length of the list.[33]

atom_prefix(*+Atom, +Prefix*)
True if *Atom* starts with the characters from *Prefix*. Its behaviour is equivalent to `?- sub_atom(Atom, 0, _, _, Prefix)`. Depreciated.

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 minimises non-determinism and creation of atoms. This is a very flexible predicate that can do search, prefix- and suffix-matching, etc.

4.22 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 `ctype` providing compatibility to some other Prolog systems. The predicates of this library are defined in terms of `code_type/2`.

[33] BUG: Note that `[]` is both an atom an empty code/character list. The predicate `atom_length/2` returns 2 for this atom.

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 hexa-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. See also `upcase_atom/2` and `downcase_atom/2`.

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. See also `upcase_atom/2` and `downcase_atom/2`.

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`.

4.22.1 Case conversion

There is nothing in the Prolog standard for converting case in textual data. The SWI-Prolog predicates `code_type/2` and `char_type/2` can be used to test and convert individual characters. We have started some additional support:

downcase_atom(+*AnyCase, -LowerCase***)**
Converts the characters of *AnyCase* into lowercase as `char_type/2` does (i.e. based on the defined *locale* if Prolog provides locale support on the hosting platform) and unifies the lowercase atom with *LowerCase*.

upcase_atom(+*AnyCase, -UpperCase***)**
Converts, similar to `downcase_atom/2`, an atom to upper-case.

4.23 Representing text in strings

SWI-Prolog supports the data type *string*. Strings are a time and space efficient mechanism to handle text in Prolog. Strings are stored 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, offering 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 9) 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 implementations. 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.

String objects by default have no lexical representation and thus can only be created using the predicates below or through the foreign language interface (See chapter 9. There are two ways to make `read/1` read text into strings, both controlled through Prolog flags. One is by setting the `double_quotes` flag to `string` and the other is by setting the `backquoted_string` flag to `true`. In latter case, `` `Hello world` `` is read into a string and `write_term/2` prints strings between back-quotes if `quoted` is `true`. This flag provides compatibility to LPA Prolog string handling.

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 character codes 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`.

4.24 Operators

Operators are defined to improve the readability 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. Operators can be exported from modules using a term `op`(*Precedence, Type, Name*) in the export list as specified by `module/2`. This is an extension specific to SWI-Prolog and the advised mechanism of portability is not an important concern.

The module-table of the module `user` acts as default table for all modules and can be modified explicitly from inside a module to achieve compatibility to other Prolog systems:

```
:- module(prove,
          [ prove/1
          ]).

:- op(900, xfx, user:(=>)).
```

Unlike what many users think, operators and quoted atoms have no relation: defining an 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 4.1. Note that all operators can be redefined by the user.

current_op(*?Precedence, ?Type, ?:Name*)
: True if *Name* is currently defined as an operator of type *Type* with precedence *Precedence*. See also `op/3`.

4.25 Character Conversion

Although I wouldn't really know for what you would like to use these features, they are provided for ISO compliance.

char_conversion(+*CharIn, +CharOut*)
: Define that term-input (see `read_term/3`) maps each character read as *CharIn* to the character

1200	xfx	`-->`, `:-`
1200	fx	`:-`, `?-`
1150	fx	`dynamic`, `discontiguous`, `initialization`, `module_transparent`, `multifile`, `thread_local`, `volatile`
1100	xfy	`;`, `|`
1050	xfy	`->`, op`*->`
1000	xfy	`,`
954	xfy	`\`
900	fy	`\+`
900	fx	`~`
700	xfx	`<`, `=`, `=..`, `=@=`, `=:=`, `=<`, `==`, `=\=`, `>`, `>=`, `@<`, `@=<`, `@>`, `@>=`, `\=`, `\==`, `is`
600	xfy	`:`
500	yfx	`+`, `-`, `/\`, `\/`, `xor`
500	fx	`+`, `-`, `?`, `\`
400	yfx	`*`, `/`, `//`, `rdiv`, `<<`, `>>`, `mod`, `rem`
200	xfx	`**`
200	xfy	`^`

Table 4.1: System operators

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`.

current_char_conversion(*?CharIn, ?CharOut*)
Queries the current character conversion-table. See `char_conversion/2` for details.

4.26 Arithmetic

Arithmetic can be divided into some special purpose integer predicates and a series of general predicates for integer, floating point and rational arithmetic as appropriate. 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 4.26.2.

4.26.1 Special purpose integer arithmetic

The predicates in this section provide more logical operations between integers. They are not covered by the ISO standard, although they are 'part of the community' and found as either library or built-in in many other Prolog systems.

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*. If *High* is `inf`

or `infinite`[34] `between/3` is true iff *Value* $\geq$ *Low*, a feature that is particularly interesting for generating integers from a certain value.

succ(*?Int1, ?Int2*)
True if *Int2* $=$ *Int1* $+ 1$ and *Int1* ≥ 0. At least one of the arguments must be instantiated to a natural number. This predicate raises the domain-error `not_less_than_zero` if called with a negative integer. E.g. `succ`(*X, 0*) fails silently and `succ`(*X, -1*) raises a domain-error.[35]

plus(*?Int1, ?Int2, ?Int3*)
True if *Int3* $=$ *Int1* $+$ *Int2*. At least two of the three arguments must be instantiated to integers.

4.26.2 General purpose arithmetic

The general arithmetic predicates are optionally compiled (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.

+Expr1 **>** *+Expr2*
True if expression *Expr1* evaluates to a larger number than *Expr2*.

+Expr1 **<** *+Expr2*
True if expression *Expr1* evaluates to a smaller number than *Expr2*.

+Expr1 **=<** *+Expr2*
True if expression *Expr1* evaluates to a smaller or equal number to *Expr2*.

+Expr1 **>=** *+Expr2*
True if expression *Expr1* evaluates to a larger or equal number to *Expr2*.

+Expr1 **=\=** *+Expr2*
True if expression *Expr1* evaluates to a number non-equal to *Expr2*.

+Expr1 **=:=** *+Expr2*
True if expression *Expr1* evaluates to a number equal to *Expr2*.

-Number **is** *+Expr*
True if *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` should be used with unbound left operand. If equality is to be tested, =:=/2 should be used. For example:

```
?- 1 is sin(pi/2).      Fails!. sin(pi/2) evaluates to the float 1.0,
                        which does not unify with the integer 1.
?- 1 =:= sin(pi/2).     Succeeds as expected.
```

[34] We prefer `infinite`, but some other Prolog systems already use `inf` for infinity we accept both for the time being.
[35] The behaviour to deal with natural numbers only was defined by Richard O'Keefe to support the common count-down-to-zero in a natural way. Up-to 5.1.8 `succ/2` also accepted negative integers.

Arithmetic types

SWI-Prolog defines the following numeric types:

- *integer*
  If SWI-Prolog is built using the *GNU multiple precision arithmetic library* (GMP), integer arithmetic is *unbounded*, which means that the size of integers is limited by available memory only. Without GMP, SWI-Prolog integers are 64-bits, regardless of the native integer size of the platform. The type of integer support can be detected using the Prolog flags `bounded`, `min_integer` and `max_integer`. As the use of GMP is default, most of the following descriptions assume unbounded integer arithmetic.

  Internally, SWI-Prolog has three integer representations. Small integers (defined by the Prolog flag `max_tagged_integer`) are encoded directly. Larger integers are represented as 64-bit value on the global stack. Integers that do not fit in 64-bit are represented as serialised GNU MPZ structures on the global stack.

- *rational number*
  Rational numbers (Q) are quotients of two integers. Rational arithmetic is only provided if GMP is used (see above). Rational numbers are currently not supported by a Prolog type. They are represented by the compound term `rdiv`(*N,M*). Rational numbers that are returned from `is/2` are *canonical*, which means M is positive and N and M have no common divisors. Rational numbers are introduced in the computation using the `rational/1`, `rationalize/1` or the `rdiv/2` (rational division) function. Using the same functor for rational division and representing rational numbers allow for passing rational numbers between computations as well as to `format/3` for printing.

  On the long term it is likely that rational numbers will become *atomic* as well as subtype of *number*. User code that creates or inspects the `rdiv`(*M,N*) terms will not be portable to future versions. Rationals are created using one of the functions mentioned above and inspected using `rational/3`.

- *float*
  Floating point numbers are represented using the C-type `double`. On most today platforms these are 64-bit IEEE floating point numbers.

Arithmetic functions that require integer arguments accept, in addition to integers, rational numbers with denominator '1' and floating point numbers that can be accurately converted to integers. If the required argument is a float the argument is converted to float. Note that conversion of integers to floating point numbers may raise an overflow exception. In all other cases, arguments are converted to the same type using the order below.

integer → rational number → floating point number

Rational number examples

The use of rational numbers with unbounded integers allows for exact integer or *fixed point* arithmetic under the addition, subtraction, multiplication and division. To exploit rational arithmetic `rdiv/2` should be used instead of '/' and floating point numbers must be converted to rational using `rational/1`. Omitting the `rational/1` on floats will convert a rational operand to float and continue the arithmetic using floating point numbers. Here are some examples.

A is 2 rdiv 6	A = 1 rdiv 3
A is 4 rdiv 3 + 1	A = 7 rdiv 3
A is 4 rdiv 3 + 1.5	A = 2.83333
A is 4 rdiv 3 + rational(1.5)	A = 17 rdiv 6

Note that floats cannot represent all decimal numbers exactly. The function `rational/1` creates an *exact* equivalent of the float, while `rationalize/1` creates a rational number that is within the float rounding error from the original float. Please check the documentation of these functions for details and examples.

Rational numbers can be printed as decimal numbers with arbitrary precision using the `format/3` floating point conversion:

```
?- A is 4 rdiv 3 + rational(1.5),
   format('~50f~n', [A]).
2.83333333333333333333333333333333333333333333333333

A = 17 rdiv 6
```

Arithmetic Functions

Arithmetic functions are terms which are evaluated by the arithmetic predicates described in section 4.26.2. 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 four types of arguments to functions:

Expr	Arbitrary expression, returning either a floating point value or an integer.
IntExpr	Arbitrary expression that must evaluate into an integer.
RatExpr	Arbitrary expression that must evaluate into a rational number.
FloatExpr	Arbitrary expression that must evaluate into a floating point.

For systems using bounded integer arithmetic (default is unbounded, see section 4.26.2 for details), integer operations that would cause overflow automatically convert to floating point arithmetic.

− *+Expr*
: $Result = -Expr$

+Expr1 **+** *+Expr2*
: $Result = Expr1 + Expr2$

+Expr1 **−** *+Expr2*
: $Result = Expr1 - Expr2$

+Expr1 ***** *+Expr2*
: $Result = Expr1 \times Expr2$

+Expr1 **/** *+Expr2*
: *Result* = $\frac{Expr1}{Expr2}$ The the flag `iso` is `true`, both arguments are converted to float and the return value is a float. Otherwise (default), if both arguments are integers the operation returns an integer if the division is exact. If at least one of the arguments is rational and the other argument is integer, the operation returns a rational number. In all other cases the return value is a float. See also `///2` and `rdiv/2`.

+IntExpr1 **mod** *+IntExpr2*
: Modulo: *Result* = *IntExpr1* - (*IntExpr1* // *IntExpr2*) × *IntExpr2* The function `mod/2` is implemented using the C `%` operator. It's behaviour with negative 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*)

+RatExpr **rdiv** *+RatExpr*
: Rational number division. This function is only available if SWI-Prolog has been compiled with rational number support. See section 4.26.2 for details.

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*. Both arguments are compared after converting to the same type, but the return value is in the original type. For example, max(2.5, 3) compares the two values after converting to float, but returns the integer 3.

min(*+Expr1, +Expr2*)
: Evaluates to the smallest of both *Expr1* and *Expr2*. See `max/2` for a description of type-handling.

.(*+Int, []*)
: A list of one element evaluates to the element. This implies `"a"` evaluates to the character code 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 character code of the letter 'a' is `0'a`.

random(*+IntExpr*)
: Evaluates to a random integer *i* for which $0 \leq i <$ *IntExpr*. 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.

rational(+*Expr*)
Convert the *Expr* to a rational number or integer. The function returns the input on integers and rational numbers. For floating point numbers, the returned rational number *exactly* represents the float. As floats cannot exactly represent all decimal numbers the results may be surprising. In the examples below, doubles can represent 0.25 and the result is as expected, in contrast to the result of `rational`(*0.1*). The function `rationalize/1` remedies this. See section 4.26.2 for more information on rational number support.

```
?- A is rational(0.25).

A is 1 rdiv 4
?- A is rational(0.1).
A = 3602879701896397 rdiv 36028797018963968
```

rationalize(+*Expr*)
Convert the *Expr* to a rational number or integer. The function is similar to `rational/1`, but the result is only accurate within the rounding error of floating point numbers, generally producing a much smaller denominator.[36]

```
?- A is rationalize(0.25).

A = 1 rdiv 4
?- A is rationalize(0.1).

A = 1 rdiv 10
```

float_fractional_part(+*Expr*)
Fractional part of a floating-point number. Negative if *Expr* is negative, rational if *Expr* is rational and 0 if *Expr* is integer. The following relation is always true: $X = float_fractional_part(X) + float_integer_part(X)$.

float_integer_part(+*Expr*)
Integer part of floating-point number. Negative if *Expr* is negative, *Expr* if *Expr* is integer.

[36] The names `rational/1` and `rationalize/1` as well as their semantics are inspired by Common Lisp.

truncate(+*Expr*)
Truncate *Expr* to an integer. If *Expr* ≥ 0 this is the same as `floor`(*Expr*). For *Expr* < 0 this is the same as `ceil`(*Expr*). E.i. truncate rounds towards zero.

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. The operation performs *arithmetic shift*, which implies that the inserted most significant bits are copies of the original most significant bit.

+*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. The returned value is the one's complement of *IntExpr*.

sqrt(+*Expr*)
Result $= \sqrt{Expr}$

sin(+*Expr*)
Result $= \sin$ *Expr*. *Expr* is the angle in radians.

cos(+*Expr*)
Result $= \cos$ *Expr*. *Expr* is the angle in radians.

tan(+*Expr*)
Result $= \tan$ *Expr*. *Expr* is the angle in radians.

asin(+*Expr*)
Result $= \arcsin$ *Expr*. *Result* is the angle in radians.

acos(+*Expr*)
Result $= \arccos$ *Expr*. *Result* is the angle in radians.

atan(+*Expr*)
Result $= \arctan$ *Expr*. *Result* is the angle in radians.

atan(+*YExpr, +XExpr*)
: *Result* $= \arctan \frac{YExpr}{XExpr}$. *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*)
: *Result* $= \ln Expr$

log10(+*Expr*)
: *Result* $= \lg Expr$

exp(+*Expr*)
: *Result* $= e^{Expr}$

+*Expr1* ****** +*Expr2*
: *Result* $= Expr1^{Expr2}$. With unbounded integers and integer values for *Expr1* and a non-negative integer *Expr2*, the result is always integer.

+*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`.

Bitvector functions The functions below are not covered by the standard. The `msb/1` function is compatible to hProlog. The others are private extensions that improve handling of —unbounded— integers as bit-vectors.

msb(+*IntExpr*)
: Return the largest integer N such that `(IntExpr >> N) /\ 1 =:= 1`. This is the (zero-origin) index of the most significant 1 bit in the value of *IntExpr*, which must evaluate to a positive integer. Errors for 0, negative integers, and non-integers.

lsb(+*IntExpr*)
: Return the smallest integer N such that `(IntExpr >> N) /\ 1 =:= 1`. This is the (zero-origin) index of the least significant 1 bit in the value of IntExpr, which must evaluate to a positive integer. Errors for 0, negative integers, and non-integers.

popcount(+*IntExpr*)
: Return the number of 1s in the binary representation of the non-negative integer *IntExpr*.

4.27 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 built-in 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*.

4.28 Built-in list operations

Most list operations are defined in the library `lists` described in section A.1. Some that are implemented with more low-level primitives are built-in and described here.

is_list(+*Term*)
True if *Term* is bound to the empty list (`[]`) or a term with functor '`.`' and arity 2 and the second argument is a list.[37] This predicate acts as if defined by the following definition:

[37] In versions before 5.0.1, `is_list/1` just checked for `[]` or `[_|_]` and `proper_list/1` had the role of the current `is_list/1`. The current definition is conform the de-facto standard. Assuming proper coding standards, there should only be very few cases where a quick-and-dirty `is_list/1` is a good choice. Richard O'Keefe pointed at this issue.

```
is_list(X) :-
        var(X), !,
        fail.
is_list([]).
is_list([_|T]) :-
        is_list(T).
```

memberchk(*?Elem, +List*)
Equivalent to `member/2`, but leaves no choice point.

length(*?List, ?Int*)
True if *Int* represents the number of elements of list *List*. Can be used to create a list holding only variables.

sort(*+List, -Sorted*)
True if *Sorted* can be unified with a list holding the elements of *List*, sorted to the standard order of terms (see section 4.6). Duplicates are removed. The implementation is in C, using *natural merge sort*[38]

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. It is 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. The implementation is in C, using *natural merge sort*.

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 <, > or =. If built-in predicate `compare/3` is used, the result is the same as `sort/2`. See also `keysort/2`.[39]

merge(*+List1, +List2, -List3*)
List1 and *List2* are lists, sorted to the standard order of terms (see section 4.6). *List3* will be unified with an ordered list holding both the elements of *List1* and *List2*. Duplicates are **not** removed.

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*.

[38] Contributed by Richard O'Keefe.
[39] Please note that the semantics have changed between 3.1.1 and 3.1.2

4.29 Finding all Solutions to a Goal

findall(+*Template, +Goal, -Bag*)
Creates a list of the instantiations *Template* 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 existential operator (`^`), except that `bagof/3` fails when goal has no solutions.

bagof(+*Template, +Goal, -Bag*)
Unify *Bag* with the alternatives of *Template*, if *Goal* has free variables besides the one sharing with *Template* bagof will backtrack over the alternatives of these free variables, unifying *Bag* with the corresponding alternatives of *Template*. 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 ?-
```

setof(+*Template, +Goal, -Set*)
Equivalent to `bagof/3`, but sorts the result using `sort/2` to get a sorted list of alternatives without duplicates.

4.30 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 ..."

maplist(+*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 `maplist/2` fails.[40]

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.

maplist(+*Pred, ?List1, ?List2, ?List3*)
Apply *Pred* on all successive triples of elements from *List1*, *List2* and *List3*. Fails if *Pred* can not be applied to a triple. See the example above.

sublist(+*Pred, +List1, ?List2*)
Unify *List2* with a list of all elements of *List1* to which *Pred* applies.

4.31 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).
```

4.32 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.

[40] The `maplist/2` predicate replaces the obsolete `checklist/2` predicate.

4.32.1 Writef

writeln(+*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 newline 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 character 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 a character code)
%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, []).

4.32.2 Format

format(*+Format*)
Defined as 'format(Format) :- format(Format, []).'

format(*+Format, +Arguments*)
Format is an atom, list of character codes, 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 character codes 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 character code 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.

Numeric conversion (`d`, `D`, `e`, `E`, `f`, `g` and `G`) accept an arithmetic expression as argument. This is introduced to handle rational numbers transparently (see section 4.26.2. The floating point conversions allow for unlimited precision for printing rational numbers in decimal form.

- `~` Output the tilde itself.
- `a` Output the next argument, which should be an atom. This option is equivalent to **w**. Compatibility reasons only.
- `c` Interpret the next argument as an character code and add it to the output. 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 (`write_canonical/1`).
- `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 text from a list of character codes or a string (see `string/1` and section 4.23) from the next argument.
- `@` Interpret the next argument as a goal and execute it. Output written to the `current_output` stream is inserted at this place. Goal is called in the module calling `format/3`. This option is not present in the original definition by Quintus, but supported by some other Prolog systems.

`t` All remaining space between 2 tab stops is distributed equally over `~t` statements between the tab stops. This space is padded with spaces by default. If an argument is supplied this is taken to be the character code 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 tab stop is assumed at the start of each line.

`|` Set a tab stop on the current position. If an argument is supplied set a tab stop on the position of that argument. This will cause all `~t`'s to be distributed between the previous and this tab stop.

`+` Set a tab stop 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(+*Output, +Format, +Arguments*)

As `format/2`, but write the output on the given *Output*. The de-facto standard only allows *Output* to be a stream. The SWI-Prolog implementation allows all valid arguments for `with_output_to/2`.[41] For example:

```
?- format(atom(A), '~D', [1000000]).
A = '1,000,000'
```

4.32.3 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

[41] Earlier versions defined `sformat/[2,3]`. These predicates have been moved to the library `backcomp`.

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(default) :- !,
        dos_newline(1).
dos_newline(N) :-
        (   N > 0
        ->  write('\r\n'),
            N2 is N - 1,
            dos_newline(N2)
        ;   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.

4.33 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.

tty_size(*-Rows, -Columns*)
Determine the size of the terminal. Platforms:

Unix If the system provides *ioctl* calls for this, these are used and `tty_size/2` properly reflects the actual size after a user resize of the window. As a fallback, the system uses `tty_get_capability/3` using `li` and `co` capabilities. In this case the reported size reflects the size at the first call and is not updated after a user-initiated resize of the terminal.

Windows Getting the size of the terminal is provided for `plwin.exe`. The requested value reflects the current size. For the multi-threaded version the console that is associated with the `user_input` stream is used.

4.34 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 one of the Win32 `SW_*` constants written in lowercase without the `SW_*`: `hide maximize minimize restore show showdefault showmaximized showminimized showminnoactive showna shownoactive shownormal`. In addition, `iconic` is a synonym for `minimize` and `normal` for `shownormal`

win_shell(*+Operation, +File, +Show*)
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(). The *Show* argument determines the initial state of the opened window (if any). See `win_exec/2` for defined values.

win_shell(+*Operation, +File*)
Same as `win_shell`(*Operation, File, normal*)

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 explicitly 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 an environment variable. *Name* and *Value* must 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`. Environment variables are shared between threads. Depending on the underlying C library, `setenv/2` and `unsetenv/1` may not be thread-safe and may cause memory leaks. Only changing the environment once and before starting threads is safe in all versions of SWI-Prolog.

unsetenv(+*Name*)
Remove an environment variable from the environment. Some systems lack the underlying unsetenv() library function. On these systems `unsetenv/1` sets the variable to the empty string.

setlocale(+*Category, -Old, +New*)
Set/Query the *locale* setting which tells the C-library how to interpret text-files, write numbers, dates, etc. Category is one of `all`, `collate`, `ctype`, `messages`, `monetary`, `numeric` or `time`. For details, please consult the C-library locale documentation. See also section 2.17.1. Please note that the locale is shared between all threads and thread-safe usage of `setlocale/3` is in general not possible. Do locale operations before starting threads or thoroughly study threading aspects of locale support in your environment before use in multi-threaded environments.

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. 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 `working_directory/2` to the expansion (see `expand_file_name/2`) of `~`. For compatibility only.

cd(+*Directory*)
Equivalent to calling `working_directory/2`. Use for compatibility only.

argv(-*Argv*)
Unify *Argv* with the list of command-line 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`. See also prolog prolog-flag `argv`.

A stand-alone program could use the following skeleton to handle command-line arguments. See also section 2.10.2.

```
main :-
        current_prolog_flag(argv, Argv),
        append(_PrologArgs, [--|AppArgs], Argv), !,
        main(AppArgs).
```

4.34.1 Dealing with time and date

Representing time in a computer system is surprisingly complicated. There are a large number of time representations in use and the correct choice depends on factors such as compactness, resolution and desired operations. Humans tend to think about time in hours, days, months, years or centuries. Physicists think about time in seconds. But, a month does not have a defined number of seconds. Even a day does not have a defined number of seconds as sometimes a leap-second is introduced to synchronise properly with our earth's rotation. At the same time, resolution demands range from better then pico-seconds to millions of years. Finally, civilizations have a wide range of calendars. Although there exist libraries dealing with most if this complexity, our desire to keep Prolog clean and lean stops us from fully supporting these.

For human-oriented tasks, time can be broken into years, months, days, hours, minutes, seconds and a timezone. Physicists prefer to have time in an arithmetic type representing seconds or fraction thereof, so basic arithmetic deal with comparison and durations. An additional advantage of the physicists approach is that it requires much less space. For these reasons, SWI-Prolog uses an arithmetic type as its prime time representation.

Many C libraries deal with time using fixed-point arithmetic, dealing with a large but finite time interval at constant resolution. In our opinion using a floating point number is a more natural choice as we can use a natural unit and the interface does not need to be changed if a higher resolution is

required in the future. Our unit of choice is the second as it is the scientific unit.[42] e have placed our origin at 1970-1-1T0:0:0Z for compatibility with the POSIX notion of time as well as with older time support provided by SWI-Prolog.

Where older versions of SWI-Prolog relied on the POSIX conversion functions, the current implementation uses libtai to realise conversion between time-stamps and calendar dates for a period of 10 million years.

Time and date data-structures

We use the following time representations

TimeStamp
A TimeStamp is a floating point number expression the time in seconds since the Epoch at 1970-1-1.

date(*Y,M,D,H,Mn,S,Off,TZ,DST*)
We call this term a *date-time* structure. The first 5 fields are integers expressing the year, month (1..12), day (1..31), hour (0..23), Minute (0..59). The *S* field holds the seconds as a floating point number between 0.0 and 60.0. *Off* is an integer representing the offset relative to UTC in seconds where positive values are west of Greenwhich. If converted from local time (see `stamp_date_time/3`, *TZ* holds the name of the local timezone. If the timezone is not known *TZ* is the atom `-`. *DST* is `true` if daylight saving time applies to the current time, `false` if daylight saving time is relevant but not effective and `-` if unknown or the timezone has no daylight saving time.

date(*Y,M.D*)
Date using the same values as described above. Extracted using `date_time_value/3`.

time(*H,Mn,S*)
Time using the same values as described above. Extracted using `date_time_value/3`.

Time and date predicates

get_time(-*TimeStamp*)
Return the current time as a *TimeStamp*. The granularity is system dependent. See section 4.34.1.

stamp_date_time(+*TimeStamp, -DateTime, +TimeZone*)
Convert a *TimeZone* to a *DateTime* in the given time zone. See section 4.34.1 for details on the data-types. *TimeZone* describes the timezone for the conversion. It is one of `local` to extract the local time, `'UTC'` to extract at UTC time or an integer describing the seconds west of Greenwhich.

date_time_stamp(+*DateTime, -TimeStamp*)
Compute the timestamp from a date/9 term. Values for month, day, hour, minute or second need not be normalized. This flexibility allows for easy computation of the time at any given number of these units from a given timestamp. Normalization can be achieved following this call with `stamp_date_time/3`. This example computes the date 200 days after 2006-7-14:

[42] Using Julian days is a choice made by the Eclipse team. As conversion to dates is needed for a human readable notation of time and Julian days cannot deal naturally with leap seconds, we decided for second as our unit.

```
?- date_time_stamp(date(2006,7,214,0,0,0,0,-,-), Stamp),
   stamp_date_time(Stamp, D, 0),
   date_time_value(date, D, Date).
Date = date(2007, 1, 30)
```

date_time_value(*?Key, +DateTime, ?Value*)
Extract values from a date/9 term. Provided keys are:

key	value
`year`	Calendar year as an integer
`month`	Calendar month as an integer 1..12
`day`	Calendar day as an integer 1..31
`hour`	Clock hour as an integer 0..23
`minute`	Clock minute as an integer 0..59
`second`	Clock second as a float 0.0..60.0
`utc_offset`	Offset to UTC in seconds (positive is west)
`time_zone`	Name of timezone; fails if unknown
`daylight_saving`	Bool (true) if dst is effective
`date`	Term `date`(*Y,M,D*)
`time`	Term `time`(*H,M,S*)

format_time(*+Out, +Format, +StampOrDateTime*)
Modelled after POSIX strftime(), using GNU extensions. *Out* is a destination as specified with `with_output_to/2`. *Format* is an atom or string with the following conversions. Conversions start with a tilde (%) character.[43]

- `a` The abbreviated weekday name according to the current locale.
- `A` The full weekday name according to the current locale.
- `b` The abbreviated month name according to the current locale.
- `B` The full month name according to the current locale.
- `c` The preferred date and time representation for the current locale.
- `C` The century number (year/100) as a 2-digit integer.
- `d` The day of the month as a decimal number (range 01 to 31).
- `D` Equivalent to %m/%d/%y. (Yecch for Americans only. Americans should note that in other countries %d/%m/%y is rather common. This means that in international context this format is ambigu ous and should not be used.)
- `e` Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space.
- `E` Modifier. Not implemented.
- `F` Equivalent to %Y-%m-%d (the ISO 8601 date format).
- `g` Like %G, but without century, i.e., with a 2-digit year (00-99).

[43] Descriptions taken from Linux Programmer's Manual

`G` The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see %V). This has the same format and value as %y, except that if the ISO week number belongs to the previous or next year, that year is used instead.

`V` The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also %U and %W.

`h` Equivalent to %b.

`H` The hour as a decimal number using a 24-hour clock (range 00 to 23).

`I` The hour as a decimal number using a 12-hour clock (range 01 to 12).

`j` The day of the year as a decimal number (range 001 to 366).

`k` The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also %H.)

`l` The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also %I.)

`m` The month as a decimal number (range 01 to 12).

`M` The minute as a decimal number (range 00 to 59).

`n` A newline character.

`O` Modifier. Not implemented.

`p` Either ‘AM’ or ‘PM’ according to the given time value, or the corresponding strings for the current locale. Noon is treated as ‘pm’ and midnight as ‘am’.

`P` Like %p but in lowercase: ‘am’ or ‘pm’ or a corresponding string for the current locale.

`r` The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to ‘%I:%M:%S %p’.

`R` The time in 24-hour notation (%H:%M). For a version including the seconds, see %T below.

`s` The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.

`S` The second as a decimal number (range 00 to 60). (The range is up to 60 to allow for occasional leap seconds.)

`t` A tab character.

`T` The time in 24-hour notation (%H:%M:%S).

`u` The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w.

`U` The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also %V and %W.

`w` The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u.

`W` The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.

`x` The preferred date representation for the current locale without the time.

`X` The preferred time representation for the current locale without the date.

`y` The year as a decimal number without a century (range 00 to 99).

`Y` The year as a decimal number including the century.

`z` The time-zone as hour offset from GMT. Required to emit RFC822-conformant dates (using "%a, %d %b %Y %H:%M:%S %z").

`Z` The time zone or name or abbreviation.

`+` The date and time in date(1) format.

`%` A literal '%' character.

4.34.2 Controlling the `PLWIN.EXE` console window

The Windows executable `PLWIN.EXE` console has a number of predicates to control the appearance of the console. Being totally non-portable, we do not advice using it for your own application, but use XPCE or another portable GUI platform instead. We give the predicates for reference here.

window_title(*-Old, +New*)
Unify *Old* with the title displayed in the console and change the title to *New*.[44]

win_window_pos(*+ListOfOptions*)
Interface to the MS-Windows SetWindowPos() function, controlling size, position and stacking order of the window. *ListOfOptions* is a list that may hold any number of the terms below.

size(*W, H*)
Change the size of the window. *W* and *H* are expressed in character-units.

position(*X, Y*)
Change the top-left corner of the window. The values are expressed in pixel units.

zorder(*ZOrder*)
Change the location in the window stacking order. Values are `bottom`, `top`, `topmost` and `notopmost`. *Topmost* windows are displayed above all other windows.

show(*Bool*)
If `true`, show the window, if `false` hide the window.

activate
If present, activate the window.

win_has_menu
True if `win_insert_menu/2` and `win_insert_menu_item/4` are present.

win_insert_menu(*+Label, +Before*)
Insert a new entry (pulldown) in the menu. If the menu already contains this entry, nothing is done. The *Label* is the label and using the Windows conventions, a letter prefixed with `&` is underlined and defines the associated accelerator key. *Before* is the label before which this one must be inserted. Using `-` adds the new entry at the end (right). For example, the call below adds a Application entry just before the Help menu.

```
win_insert_menu('&Application', '&Help')
```

[44] BUG: This predicate should have been called `win_window_title` for consistent naming.

win_insert_menu_item(+*Pulldown, +Label, +Before, :Goal*)
Add an item to the named *Pulldown* menu. *Label* and *Before* are handled as in `win_insert_menu/2`, but the label - inserts a *separator*. *Goal* is called if the user selects the item.

4.35 File System Interaction

access_file(+*File, +Mode*)
True 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*)
True if *File* exists and is a regular file. 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 returned *Directory* name does not end in `/`. There are two special cases. The directory-name of `/` is `/` itself and the directory-name if *File* does not contain any `/` characters is `..`

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*)
True 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*)
True if *Directory* exists and is a directory. 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 a local file-name into an absolute path. The absolute path is canonised: references to `.` and `..` are deleted. This predicate ensures that expanding a file-name it returns the same absolute path regardless of how the file is addressed. SWI-Prolog uses absolute file names to register source files independent of the current working directory. See also `absolute_file_name/3`. See also `absolute_file_name/3` and `expand_file_name/2`.

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`.

relative_to(+*FileOrDir*)

Resolve the path relative to the given directory or directory the holding the given file. Without this option, paths are resolved relative to the working directory (see `working_directory/2`) or, if *Spec* is atomic and `absolute_file_name/[2,3]` is executed in a directive, it uses the current source-file as reference.

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 `['']`. The file-type `source` is an alias for `prolog` for compatibility to SICStus Prolog. See also `prolog_file_type/2`.

file_errors(*fail/error*)

If `error` (default), throw and `existence_error` exception if the file cannot be found. If `fail`, stay silent.[45]

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.

expand(*true/false*)

If `true` (default is `false`) and *Spec* is atomic, call `expand_file_name/2` followed by `member/2` on *Spec* before proceeding. This is a SWI-Prolog extension.

The prolog-flag `verbose_file_search` can be set to `true` to help debugging Prolog's search for files.

Compatibility considerations to common argument-order in ISO as well as SICStus `absolute_file_name/3` forced us to be flexible here. If the last argument is a list and the

[45] Silent operation was the default up to version 3.2.6.

2nd not, the arguments are swapped, making the call `absolute_file_name`(+*Spec, -Path, +Options*) valid as well.

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 wildcards, 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.[46].

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.

[46]On Windows, the home directory is determined as follows: if the environment variable `HOME` exists, this is used. If 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

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.

working_directory(-*Old, +New*)
Unify *Old* with an absolute path to the current working directory and change working directory to *New*. Use the pattern `working_directory`(*CWD, CWD*) to get the current directory. See also `absolute_file_name/2` and `chdir/1`.[47] Note that the working directory is shared between all threads.

chdir(+*Path*)
Compatibility predicate. New code should use `working_directory/2`.

4.36 User Top-level 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)`.[48]

[47] BUG: Some of the file-I/O predicates use local filenames. Changing directory while file-bound streams are open causes wrong results on `telling/1`, `seeing/1` and `current_stream/3`

[48] BUG: In the multi-threaded version, `halt/0` does not work when not called from the *main* thread. In the current system a `permission_error` exception is raised. Future versions may enable `halt/0` from any thread.

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 top-level 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 top-level. This predicate is used by the top-level (`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 top-level query. *Bindings* is the query $\langle Name \rangle = \langle Value \rangle$ binding list from the query. *ExpandedBindings* must be unified with the bindings the top-level should print.

4.37 Creating a Protocol of the User Interaction

SWI-Prolog offers the possibility to log the interaction with the user on a file.[49] All Prolog interaction, including warnings and tracer output, are written on the protocol file.

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*)
True if a protocol was started with `protocol/1` or `protocola/1` and unifies *File* with the current protocol output file.

4.38 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 front-end of the tracer. This front-end is installed using the predicate `guitracer/0`.

[49] A similar facility was added to Edinburgh C-Prolog by Wouter Jansweijer.

trace
Start the tracer. `trace/0` itself cannot be seen in the tracer. Note that the Prolog top-level treats `trace/0` special; it means 'trace the next goal'.

tracing
True if 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 front-end 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 port names (`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 choice-points 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. Implemented 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 4.4.

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*⟩`)` [50] 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:

[50] In older versions '?' was a prefix operator. Inversions after 5.5.13, explicit brackets are needed.

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`).
`charset`	off	Warn on atoms and variables holding non-ASCII characters that are not quoted. See also section 2.15.1.

4.39 Obtaining Runtime Statistics

statistics(+*Key*, -*Value*)
Unify system statistics determined by *Key* with *Value*. The possible keys are given in the table 4.2. The last part of the table contains keys for compatibility to other Prolog implementations (Quintus) for improved portability. Note that the ISO standard does not define methods to collect system statistics.

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.

4.40 Execution profiling

This section describes the hierarchical execution profiler introduced in SWI-Prolog 5.1.10. This profiler is based on ideas from `gprof` described in [Graham *et al.*, 1982]. The profiler consists of two parts: the information-gathering is built into the kernel,[51] and a presentation component which is defined in the `statistics` library. The latter can be hooked, which is used by the XPCE module `swi/pce_profile` to provide an interactive graphical representation of results.

[51] There are two implementations; one based on setitimer() using the `SIGPROF` signal and one using Windows Multi Media (MM) timers. On other systems the profiler is not provided.

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 2.18.1)
heapused	Bytes heap in use by Prolog.
heaplimit	Maximum size of the heap (see section 2.18.1)
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
thread_cputime	MT-version: seconds CPU time used by finished threads. Supported on Windows-NT and later, Linux and possibly a few more. Verify it gives plausible results before using.
Compatibility keys	
runtime	[CPU time, CPU time since last] (milliseconds)
system_time	[System CPU time, System CPU time since last] (milliseconds)
real_time	[Wall time, Wall time since last] (seconds since 1970)
memory	[Total unshared data, free memory] (Uses getrusage() if available, otherwise incomplete own statistics.
stacks	[global use, local use]
program	[heap, 0]
global_stack	[global use, global free]
local_stack	[local use, local free]
trail	[trail use, 0]
garbage_collection	[number of GC, bytes gained, time spent]
stack_shifts	[global shifts, local shifts, time spent] (fails if no shifter in this version)
atoms	[number, memory use, 0]
atom_garbage_collection	[number of AGC, bytes gained, time spent]
core	Same as memory

Table 4.2: Keys for `statistics/2`

4.40.1 Profiling predicates

Currently, the interface is kept compatible with the old profiler. As experience grows, it is likely that the old interface is replaced with one that better reflects the new capabilities. Feel free to examine the internal interfaces and report useful application thereof.

profile(*:Goal*)
Execute *Goal* just like `time/1`, collecting profiling statistics and call `show_profile`(*plain, 25*). With XPCE installed this opens a graphical interface to the collected profiling data.

profile(*:Goal, +Style, +Number*)
Execute *Goal* just like `time/1`. Collect profiling statistics and show the top *Number* procedures on the current output stream (see `show_profile/1`) using *Style*. 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`. The `profile/1` predicate is a backward compatibility interface to `profile/1`. The other predicates in this section are low-level predicates for special cases.

show_profile(*+Style, +Number*)
Show the collected results of the profiler. It shows the top *Number* predicates according the percentage CPU-time used. If *Style* is `plain` the time spent in the predicates itself is displayed. If *Style* is `cumulative` the time spent in its siblings (callees) is added to the predicate.

This predicate first calls prolog:show_profile_hook/2. If XPCE is loaded this hook is used to activate a GUI interface to visualise the profile results.

show_profile(*+Number*)
Compatibility. Same as `show_profile`(*plain, Number*).

profiler(*-Old, +New*)
Query or change the status of the profiler. The status is a boolean (`true` or `false`) stating whether or not the profiler is collecting data. It can be used to enable or disable profiling certain parts of the program.

reset_profiler
Switches the profiler to `false` and clears all collected statistics.

noprofile(*+Name/+Arity, ...*)
Declares the predicate *Name/Arity* to be invisible to the profiler. The time spend in the named predicate is added to the caller and the callees are linked directly to the caller. This is particularly useful for simple meta-predicates such as `call/1`, `ignore/1`, `catch/3`, etc.

4.40.2 Visualizing profiling data

Browsing the annotated call-tree as described in section 4.40.3 itself is not very attractive. Therefore, the results are combined per predicate, collecting all *callers* and and *callees* as well as the propagation of time and activations in both directions. Figure 4.1 illustrates this. The central yellowish line is the 'current' predicate with counts for time spent in the predicate ('Self'), time spent in its children ('Siblings'), activations through the call and redo ports. Above that are the *callers*. Here, the two time fields indicate how much time is spent serving each of the callers. The columns sum to the time in the yellowish line. The caller <*recursive*> are the number of recursive calls. Below the yellowish lines

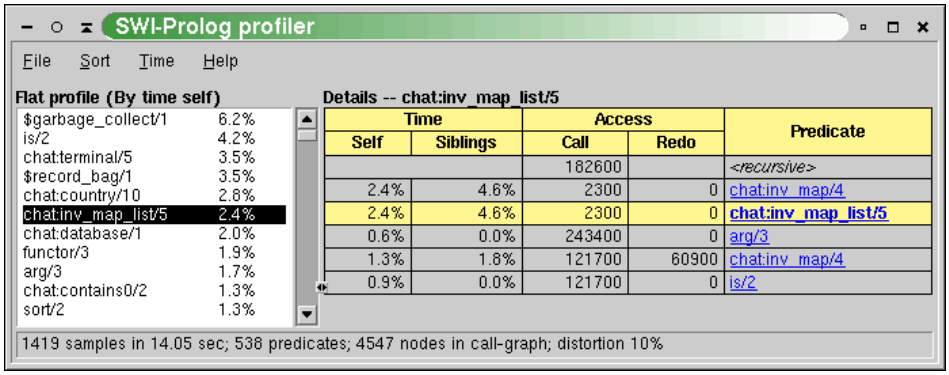

Figure 4.1: Execution profiler showing the activity of the predicate chat:inv_map_list/5.

are the callees, with the time spent in the callee itself for serving the current predicate and the time spent in the callees of the callee ('Siblings'), so the whole time-block adds up to the 'Siblings' field of the current predicate. The 'Access' fields show how many times the current predicate accesses each of the callees.

The predicates have a menu that allows changing the view of the detail window to the given caller or callee, showing the documentation (if it is a built-in) and/or jumping to the source.

The statistics shown in the report-field of figure 4.1 show the following information:

- *samples*
  Number of times the call-tree was sampled for collecting time statistics. On most hardware the resolution of `SIGPROF` is 1/100 second. This number must be sufficiently large to get reliable timing figures. The Time menu allows viewing time as samples, relative time or absolute time.
- *sec*
  Total user CPU time with the profiler active.
- *predicates*
  Total count of predicates that have been called at least one time during the profile.
- *nodes*
  Number of nodes in the call-tree.
- *distortion*
  How much of the time is spend building the call-tree as a percentage of the total execution time. Timing samples while the profiler is building the call-tree are not added to the call-tree.

4.40.3 Information gathering

While the program executes under the profiler, the system builds a *dynamic* call-tree. It does this using three hooks from the kernel: one that starts a new goal (*profCall*), one the tells the system which goal is resumed after an *exit* (*profExit*) and one that tells the system which goal is resumed after a *fail* (i.e. which goal is used to *retry* (*profRedo*)). The profCall() function finds or creates the subnode for the argument predicate below the current node, increments the call-count of this link and returns the sub-node which is recorded in the Prolog stack-frame. Choice-points are marked with the current profiling node. profExit() and profRedo() pass the profiling node where execution resumes.

Just using the above algorithm would create a much too big tree due to recursion. For this reason the system performs detection of recursion. In the simplest case, recursive procedures increment the 'recursive' count on the current node. Mutual recursion however is not easily detected. For example, `call/1` can call a predicate that uses `call/1` itself. This can be viewed as a recursive invocation, but this is generally not desirable. Recursion is currently assumed if the same predicate *with the same parent* appears higher in the call-graph. Early experience with a some arbitrary non-trivial programs are promising.

The last part of the profiler collects statistics on the CPU-time used in each node. On systems providing setitimer() with `SIGPROF`, it 'ticks' the current node of the call-tree each time the timer fires. On Windows a MM-timer in a separate thread checks 100 times per second how much time is spent in the profiled thread and adds this to the current node. See section 4.40.3 for details.

Profiling in the Windows Implementation

Profiling in the Windows version is similar but as profiling is a statistical process it is good to be aware of the implementation[52] for proper interpretation of the results.

Windows does not provide timers that fire asynchronously, frequent and proportional to the CPU time used by the process. Windows does provide multi-media timers that can run at high frequency. Such timers however run in a separate thread of execution and they are fired on the wall-clock rather than the amount of CPU time used. The profiler installs such a timer running, for saving CPU time, rather inaccurately at about 100 Hz. Each time it is fired, it determines the milliseconds CPU time used by Prolog since the last time it was fired. If this value is non-zero, active predicates are incremented with this value.

4.41 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. On multi-threaded versions the actual collection is delayed until there there are no threads performing normal garbage collection. In this case `garbage_collect_atoms/0` returns immediately. Note this implies there is no guarantee it will *ever* happen as there may always be threads performing garbage collection.

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

[52] We hereby acknowledge Lionel Fourquaux, who suggested the design described here after a newsnet enquiry.

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.

4.42 Windows DDE interface

The predicates in this section deal with MS-Windows 'Dynamic Data Exchange' or DDE protocol.[53] A Windows DDE conversation is a form of interprocess communication based on sending reserved window-events between the communicating processes.

See also section 9.4 for loading Windows DLL's into SWI-Prolog.

4.42.1 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).
```

[53] This interface is contributed by Don Dwiggins.

```
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 4.23 for a discussion on this data type.

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`.

4.42.2 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.[54]

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.

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.

4.43 Miscellaneous

dwim_match(+*Atom1, +Atom2*)
True 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:

[54] Up-to 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.

- 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*)
True 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
```

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.

On Unix systems the `sleep/1` predicate is realised —in order of preference— by nanosleep(), usleep(), select() if the time is below 1 minute or sleep(). On Windows systems Sleep() is used.

5 Using Modules

5.1 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 its own predicate space. A module consists of a declaration for its name, its *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.

5.2 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) :-
        atom_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.

5.3 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).
```

5.4 Importing Predicates into a Module

As explained before, in the predicate based approach adapted by SWI-Prolog, each module has its 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. This predicate is equivalent to `ensure_loaded/1`, except that it raises an error if *File* is not a module file.

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 one of the module's *import modules*. On success, normal execution is resumed.

5.4.1 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 import 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 import from the module `user`. This implies they can use all predicates imported into `user` without explicitly importing them.

5.5 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 (library 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.

5.5.1 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 5.6.2 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 5.6.

5.6 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.

5.6.1 Definition and Context Module

Each predicate of the program is assigned a module, called its *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 its 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 successive pairs
%       of elements of List1 and List2.

maplist(_, [], []).
maplist(Goal, [Elem1|Tail1], [Elem2|Tail2]) :-
        apply(Goal, [Elem1, Elem2]),
        maplist(Goal, Tail1, Tail2).
```

5.6.2 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 top-level or to implement a object-oriented system (see above). The latter is useful to create and modify *dynamic modules* (see section 5.7).

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
```

5.7 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
```

5.8 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 predicate indicators (name/arity pairs) or operator declarations using the format `op`(*Precedence, Type, Name*). Operators defined in the export list are available inside the module as well as to modules importing this module. See also section 4.24.

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 `quintus` and provides a partial emulation of the Quintus predicate. See section 5.9.1 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.

strip_module(+*Term, -Module, -Plain*)
: Used in module transparent or meta-predicates to extract the referenced module and plain term. If *Term* is a module-qualified term, i.e. of the format *Module*:*Plain*, *Module* and *Plain* are unified to these values. Otherwise *Plain* is unified to *Term* and *Module* to the context module.

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`.

import_module(+*Module, -Import*)
True if *Import* is defined as an import module for *Module*. All normal modules only import from `user`, which imports from `system`. The predicates `add_import_module/3` and `delete_import_module/2` can be used to manipulate the import list.

add_import_module(+*Module, +Import, +StartOrEnd*)
If *Import* is not already an import module for *Module*, add it to this list at the `start` or `end` depending on *StartOrEnd*. See also `import_module/2` and `delete_import_module/2`.

delete_import_module(+*Module, +Import*)
Delete *Import* from the list of import modules for *Module*. Fails silently if *Import* is not in the list.

default_module(+*Module, -Default*)
Successively 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`.

Backward compatibility. New code should use `import_module/2`.

module(+*Module*)
The call `module(Module)` may be used to switch the default working module for the interactive top-level (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).
...
```

5.9 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.

- Operator declarations are local to a module and may be exported.

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 5.1 gives the solution.

5.9.1 Emulating meta_predicate

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 `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.

[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 5.1: findall/3 using modules

Special Variables and Coroutining

6

This chapter deals with extensions primarily designed to support constraint logic programming (CLP).

6.1 Attributed variables

Attributed variables provide a technique for extending the Prolog unification algorithm [Holzbaur, 1990] by hooking the binding of attributed variables. There is little consensus in the Prolog community on the exact definition and interface to attributed variables. The SWI-Prolog interface is identical to the one realised by Bart Demoen for hProlog [Demoen, 2002].

Binding an attributed variable schedules a goal to be executed at the first possible opportunity. In the current implementation the hooks are executed immediately after a successful unification of the clause-head or successful completion of a foreign language (built-in) predicate. Each attribute is associated to a module and the hook (`attr_unify_hook/2`) is executed in this module. The example below realises a very simple and incomplete finite domain reasoner.

```
:- module(domain,
          [ domain/2                    % Var, ?Domain
          ]).
:- use_module(library(ordsets)).

domain(X, Dom) :-
        var(Dom), !,
        get_attr(X, domain, Dom).
domain(X, List) :-
        list_to_ordset(List, Domain),
        put_attr(Y, domain, Domain),
        X = Y.

%       An attributed variable with attribute value Domain has been
%       assigned the value Y

attr_unify_hook(Domain, Y) :-
        (   get_attr(Y, domain, Dom2)
        ->  ordset_intersection(Domain, Dom2, NewDomain),
            (   NewDomain == []
            ->  fail
            ;   NewDomain = [Value]
            ->  Y = Value
            ;   put_attr(Y, domain, NewDomain)
```

```
        )
    ;   var(Y)
    ->  put_attr( Y, domain, Domain )
    ;   ordset_memberchk(Y, Domain)
    ).
```

Before explaining the code we give some example queries:

```
?- domain(X, [a,b]), X = c                  no
?- domain(X, [a,b]), domain(X, [a,c]).      X = a
?- domain(X, [a,b,c]), domain(X, [a,c]).    X = _G492att(domain, [a, c], [])
```

The predicate `domain/2` fetches (first clause) or assigns (second clause) the variable a *domain*, a set of values it can be unified with. In the second clause first associates the domain with a fresh variable and then unifies X to this variable to deal with the possibility that X already has a domain. The predicate `attr_unify_hook/2` is a hook called after a variable with a domain is assigned a value. In the simple case where the variable is bound to a concrete value we simply check whether this value is in the domain. Otherwise we take the intersection of the domains and either fail if the intersection is empty (first example), simply assign the value if there is only one value in the intersection (second example) or assign the intersection as the new domain of the variable (third example).

attvar(@*Term*)
: Succeeds if *Term* is an attributed variable. Note that `var/1` also succeeds on attributed variables. Attributed variables are created with `put_attr/3`.

put_attr(+*Var, +Module, +Value*)
: If *Var* is a variable or attributed variable, set the value for the attribute named *Module* to *Value*. If an attribute with this name is already associated with *Var*, the old value is replaced. Backtracking will restore the old value (i.e. an attribute is a mutable term. See also `setarg/3`). This predicate raises a type error if *Var* is not a variable or *Module* is not an atom.

get_attr(+*Var, +Module, -Value*)
: Request the current *value* for the attribute named *Module*. If *Var* is not an attributed variable or the named attribute is not associated to *Var* this predicate fails silently. If *Module* is not an atom, a type error is raised.

del_attr(+*Var, +Module*)
: Delete the named attribute. If *Var* loses its last attribute it is transformed back into a traditional Prolog variable. If *Module* is not an atom, a type error is raised. In all other cases this predicate succeeds regardless whether or not the named attribute is present.

attr_unify_hook(+*AttValue, +VarValue*)
: Hook that must be defined in the module an attributed variable refers to. Is is called *after* the attributed variable has been unified with a non-var term, possibly another attributed variable. *AttValue* is the attribute that was associated to the variable in this module and *VarValue* is the new value of the variable. Normally this predicate fails to veto binding the variable to *VarValue*, forcing backtracking to undo the binding. If *VarValue* is another attributed variable the hook often combines the two attribute and associates the combined attribute with *VarValue* using `put_attr/3`.

attr_portray_hook(+*AttValue,* +*Var***)**
Called by `write_term/2` and friends for each attribute if the option `attributes(`*portray*`)` is in effect. If the hook succeeds the attribute is considered printed. Otherwise `Module = ...` is printed to indicate the existence of a variable.

6.1.1 Special purpose predicates for attributes

Normal user code should deal with `put_attr/3`, `get_attr/3` and `del_attr/2`. The routines in this section fetch or set the entire attribute list of a variables. Use of these predicates is anticipated to be restricted to printing and other special purpose operations.

get_attrs(+*Var, -Attributes***)**
Get all attributes of *Var*. *Attributes* is a term of the form `att(`*Module, Value, MoreAttributes*), where *MoreAttributes* is `[]` for the last attribute.

put_attrs(+*Var, -Attributes***)**
Set all attributes of *Var*. See `get_attrs/2` for a description of *Attributes*.

copy_term_nat(+*Term, -Copy***)**
As `copy_term/2`. Attributes however, are *not* copied but replaced by fresh variables.

6.2 Coroutining

Coroutining deals with having Prolog goals scheduled for execution as soon as some conditions is fulfilled. In Prolog the most commonly used conditions is the instantiation (binding) of a variable. Scheduling a goal to execute immediately after a variable is bound allows may be used to avoid instantiation errors for some built-in predicates (e.g. arithmetic), do work *lazy*, prevent the binding of a variable to a particular value, etc. Using `freeze/2` for example we can define a variable can only be assigned an even number:

```
?- freeze(X, X mod 2 =:= 0), X = 3

No
```

freeze(+*Var, :Goal***)**
Delay the execution of *Goal* until *Var* is bound (i.e. is not a variable or attributed variable). If *Var* is bound on entry `freeze/2` is equivalent to `call/1`. The `freeze/2` predicate is realised using an attributed variable associated with the module `freeze`, so `get_attr(Var, freeze, AttVal)` can be used to find out whether and which goals are delayed on *Var*.

frozen(@*Var, -Goal***)**
Unify *Goal* with the goal or conjunction of goals delayed on *Var*. If no goals are frozen on *Var*, *Goal* is unified to `true`.

when(@*Condition, :Goal***)**
Execute *Goal* when *Condition* becomes true. *Condition* is one of `?=(`*X, Y*`)`, `nonvar(`*X*`)`,

`ground`(*X*), `,`(*Cond1, Cond2*) or `;`(*Cond1, Cond2*). See also `freeze/2` and `dif/2`. The implementation can deal with cyclic terms.

The `when/2` predicate is realised using attributed variable associated with the module `when`. It is defined in the autoload library `when`.

dif(*@A, @B*)
The `dif/2` predicate provides a constraint stating that *A* and *B* are different terms. If *A* and *B* can never unify `dif/2` succeeds deterministically. If *A* and *B* are identical it fails immediately and finally, if *A* and *B* can unify, goals are delayed that prevent *A* and *B* to become equal. The `dif/2` predicate behaves as if defined by `dif(X, Y) :- when(?=(X, Y), X \== Y)`. See also `?=/2`. The implementation can deal with cyclic terms.

The `dif/2` predicate is realised using attributed variable associated with the module `dif`. It is defined in the autoload library `dif`.

6.3 Global variables

Global variables are associations between names (atoms) and terms. They differ in various ways from storing information using `assert/1` or `recorda/3`.

- The value lives on the Prolog (global) stack. This implies that lookup time is independent from the size of the term. This is particularly interesting for large data structures such as parsed XML documents or the CHR global constraint store.
- They support both global assignment using `nb_setval/2` and backtrackable assignment using `b_setval/2`.
- Only one value (which can be an arbitrary complex Prolog term) can be associated to a variable at a time.
- Their value cannot be shared among threads. Each thread has its own namespace and values for global variables.
- Currently global variables are scoped globally. We may consider module scoping in future versions.

Both `b_setval/2` and `nb_setval/2` implicitly create a variable if the referenced name does not already refer to a variable.

Global variables may be initialised from directives to make them available during the program lifetime, but some considerations are necessary for saved-states and threads. Saved-states to not store global variables, which implies they have to be declared with `initialization/1` to recreate them after loading the saved state. Each thread has its own set of global variables, starting with an empty set. Using `thread_initialization/1` to define a global variable it will be defined, restored after reloading a saved state and created in all threads that are created *after* the registration. Finally, global variables can be initialised using the exception hook called `exception/3`. The latter technique is by CHR (see chapter 7.

b_setval(+*Name, +Value*)
Associate the term *Value* with the atom *Name* or replaces the currently associated value with *Value*. If *Name* does not refer to an existing global variable a variable with initial value `[]` is created (the empty list). On backtracking the assignment is reversed.

b_getval(+*Name, -Value*)
Get the value associated with the global variable *Name* and unify it with *Value*. Note that this unification may further instantiate the value of the global variable. If this is undesirable the normal precautions (double negation or `copy_term/2`) must be taken. The `b_getval/2` predicate generates errors if *Name* is not an atom or the requested variable does not exist.

nb_setval(+*Name, +Value*)
Associates a copy of *Value* created with `duplicate_term/2` with the atom *Name*. Note that this can be used to set an initial value other than `[]` prior to backtrackable assignment.

nb_getval(+*Name, -Value*)
The `nb_getval/2` predicate is a synonym for `b_getval/2`, introduced for compatibility and symmetry. As most scenarios will use a particular global variable either using non-backtracable or backtrackable assignment, using `nb_getval/2` can be used to document that the variable is used non-backtracable.

nb_linkval(+*Name, +Value*)
Associates the term *Value* with the atom *Name* without copying it. This is a fast special-purpose variation of `nb_setval/2` intended for expert users only because the semantics on backtracking to a point before creating the link are poorly defined for compound terms. The principal term is always left untouched, but backtracking behaviour on arguments is undone if the original assignment was *trailed* and left alone otherwise, which implies that the history that created the term affects the behaviour on backtracking. Please consider the following example:

```
demo_nb_linkval :-
        T = nice(N),
        (   N = world,
            nb_linkval(myvar, T),
            fail
        ;   nb_getval(myvar, V),
            writeln(V)
        ).
```

nb_current(*?Name, ?Value*)
Enumerate all defined variables with their value. The order of enumeration is undefined.

nb_delete(+*Name*)
Delete the named global variable.

6.3.1 Compatibility of SWI-Prolog Global Variables

Global variables have been introduced by various Prolog implementations recently. The implementation of them in SWI-Prolog is based on hProlog by Bart Demoen. In discussion with Bart it was

decided that the semantics if hProlog `nb_setval/2`, which is equivalent to `nb_linkval/2` is not acceptable for normal Prolog users as the behaviour is influenced by how built-in predicates constructing terms (`read/1`, =../2, etc.) are implemented.

GNU-Prolog provides a rich set of global variables, including arrays. Arrays can be implemented easily in SWI-Prolog using `functor/3` and `setarg/3` due to the unrestricted arity of compound terms.

CHR: Constraint Handling Rules

7

This chapter is written by Tom Schrijvers, K.U. Leuven, and adjustments by Jan Wielemaker.

The CHR system of SWI-Prolog is the *K.U.Leuven CHR system*. The runtime environment is written by Christian Holzbaur and Tom Schrijvers while the compiler is written by Tom Schrijvers. Both are integrated with SWI-Prolog and licensed under compatible conditions with permission from the authors.

The main reference for the K.U.Leuven CHR system is:

- T. Schrijvers, and B. Demoen, *The K.U.Leuven CHR System: Implementation and Application*, First Workshop on Constraint Handling Rules: Selected Contributions (Frühwirth, T. and Meister, M., eds.), pp. 1–5, 2004.

On the K.U.Leuven CHR website (`http://www.cs.kuleuven.be/~toms/CHR/`) you can find more related papers, references and example programs.

7.1 Introduction

Constraint Handling Rules (CHR) is a committed-choice rule-based language embedded in Prolog. It is designed for writing constraint solvers and is particularly useful for providing application-specific constraints. It has been used in many kinds of applications, like scheduling, model checking, abduction, type checking among many others.

CHR has previously been implemented in other Prolog systems (SICStus, Eclipse, Yap), Haskell and Java. This CHR system is based on the compilation scheme and runtime environment of CHR in SICStus.

In this documentation we restrict ourselves to giving a short overview of CHR in general and mainly focus on elements specific to this implementation. For a more thorough review of CHR we refer the reader to [Frühwirth, 1998]. More background on CHR can be found at [Frühwirth,].

In section 7.2 we present the syntax of CHR in Prolog and explain informally its operational semantics. Next, section 7.3 deals with practical issues of writing and compiling Prolog programs containing CHR. Section 7.4 explains the currently primitive CHR debugging facilities. Section 7.4.3 provides a few useful predicates to inspect the constraint store and section 7.5 illustrates CHR with two example programs. In section 7.6 some compatibility issues with older versions of this system and SICStus' CHR system. Finally, section 7.7 concludes with a few practical guidelines for using CHR.

7.2 Syntax and Semantics

7.2.1 Syntax

The syntax of CHR rules is the following:

```
rules --> rule, rules.
rules --> [].

rule --> name, actual_rule, pragma, [atom('.')].

name --> atom, [atom('@')].
name --> [].

actual_rule --> simplification_rule.
actual_rule --> propagation_rule.
actual_rule --> simpagation_rule.

simplification_rule --> head, [atom('<=>')], guard, body.
propagation_rule --> head, [atom('==>')], guard, body.
simpagation_rule --> head, [atom('\')], head, [atom('<=>')],
                     guard, body.

head --> constraints.

constraints --> constraint, constraint_id.
constraints --> constraint, constraint_id, [atom(',')], constraints.

constraint --> compound_term.

constraint_id --> [].
constraint_id --> [atom('#')], variable.
constraint_id --> [atom('#')], [atom('passive')] .

guard --> [].
guard --> goal, [atom('|')].

body --> goal.

pragma --> [].
pragma --> [atom('pragma')], actual_pragmas.

actual_pragmas --> actual_pragma.
actual_pragmas --> actual_pragma, [atom(',')], actual_pragmas.

actual_pragma --> [atom('passive(')], variable, [atom(')')].
```

Note that the guard of a rule may not contain any goal that binds a variable in the head of the rule with a non-variable or with another variable in the head of the rule. It may however bind variables that do not appear in the head of the rule, e.g. an auxiliary variable introduced in the guard.

7.2.2 Semantics

In this subsection the operational semantics of CHR in Prolog are presented informally. They do not differ essentially from other CHR systems.

When a constraint is called, it is considered an active constraint and the system will try to apply the rules to it. Rules are tried and executed sequentially in the order they are written.

A rule is conceptually tried for an active constraint in the following way. The active constraint is matched with a constraint in the head of the rule. If more constraints appear in the head they are looked for among the suspended constraints, which are called passive constraints in this context. If the necessary passive constraints can be found and all match with the head of the rule and the guard of the rule succeeds, then the rule is committed and the body of the rule executed. If not all the necessary passive constraint can be found, the matching fails or the guard fails, then the body is not executed and the process of trying and executing simply continues with the following rules. If for a rule, there are multiple constraints in the head, the active constraint will try the rule sequentially multiple times, each time trying to match with another constraint.

This process ends either when the active constraint disappears, i.e. it is removed by some rule, or after the last rule has been processed. In the latter case the active constraint becomes suspended.

A suspended constraint is eligible as a passive constraint for an active constraint. The other way it may interact again with the rules, is when a variable appearing in the constraint becomes bound to either a non-variable or another variable involved in one or more constraints. In that case the constraint is triggered, i.e. it becomes an active constraint and all the rules are tried.

Rule Types There are three different kinds of rules, each with their specific semantics:

- *simplification*
  The simplification rule removes the constraints in its head and calls its body.
- *propagation*
  The propagation rule calls its body exactly once for the constraints in its head.
- *simpagation*
  The simpagation rule removes the constraints in its head after the $\backslash$ and then calls its body. It is an optimization of simplification rules of the form:

  $$constraints_1, constraints_2 <=> constraints_1, body$$

  Namely, in the simpagation form:

  $$constraints_1 \backslash constraints_2 <=> body$$

  The $constraints_1$ constraints are not called in the body.

Rule Names Naming a rule is optional and has no semantical meaning. It only functions as documentation for the programmer.

Pragmas The semantics of the pragmas are:

passive(*Identifier***)**
The constraint in the head of a rule *Identifier* can only match a passive constraint in that rule. There is an abbreviated syntax for this pragma. Instead of:

```
..., c # Id, ... <=> ... pragma passive(Id)
```

you can also write

```
..., c # passive, ... <=> ...
```

Additional pragmas may be released in the future.

:- chr_option(+*Option, +Value*)

It is possible to specify options that apply to all the CHR rules in the module. Options are specified with the `chr_option/2` declaration:

```
:- chr_option(Option,Value).
```

and may appear in the file anywhere after the first constraints declaration.

Available options are:

check_guard_bindings

This option controls whether guards should be checked for (illegal) variable bindings or not. Possible values for this option are `on`, to enable the checks, and `off`, to disable the checks. If this option is on, any guard fails when it binds a variable that appears in the head of the rule. When the option is off (default), the behavior of a binding in the guard is undefined.

optimize

This option controls the degree of optimization. Possible values are `full`, to enable all available optimizations, and `off` (default), to disable all optimizations. The default is derived from the SWI-Prolog flag `optimise`, where `true` is mapped to `full`. Therefore the command-line option `-O` provides full CHR optimization. If optimization is enabled, debugging must be disabled.

debug

This options enables or disables the possibility to debug the CHR code. Possible values are `on` (default) and `off`. See section 7.4 for more details on debugging. The default is derived from the prolog flag `generate_debug_info`, which is `true` by default. See `-nodebug`. If debugging is enabled, optimization must be disabled.

7.3 CHR in SWI-Prolog Programs

7.3.1 Embedding in Prolog Programs

The CHR constraints defined in a `.pl` file are associated with a module. The default module is `user`. One should never load different `.pl` files with the same CHR module name.

7.3.2 Constraint declaration

:- chr_constraint(+*Specifier***)**

Every constraint used in CHR rules has to be declared with a `chr_constraint/1` declaration by the *constraint specifier*. For convenience multiple constraints may be declared at once with the same `chr_constraint/1` declaration followed by a comma-separated list of constraint specifiers.

A constraint specifier is, in its compact form, F/A where F and A are respectively the functor name and arity of the constraint, e.g.:

```
:- chr_constraint foo/1.
:- chr_constraint bar/2, baz/3.
```

In its extended form, a constraint specifier is $c(A_1, \ldots, A_n)$ where c is the constraint's functor, n its arity and the A_i are argument specifiers. An argument specifier is a mode, optionally followed by a type. E.g.

```
:- chr_constraint get_value(+,?).
:- chr_constraint domain(?int, +list(int)),
                  alldifferent(?list(int)).
```

Modes A mode is one of:

–
: The corresponding argument of every occurrence of the constraint is always unbound.

+
: The corresponding argument of every occurrence of the constraint is always ground.

?
: The corresponding argument of every occurrence of the constraint can have any instantiation, which may change over time. This is the default value.

Types A type can be a user-defined type or one of the built-in types. A type comprises a (possibly infinite) set of values. The type declaration for a constraint argument means that for every instance of that constraint the corresponding argument is only ever bound to values in that set. It does not state that the argument necessarily has to be bound to a value.

The built-in types are:

int
: The corresponding argument of every occurrence of the constraint is an integer number.

dense_int
: The corresponding argument of every occurrence of the constraint is an integer that can be used as an array index. Note that if this argument takes values in $[0, n]$, the array takes $O(n)$ space.

float
: . . . a floating point number.

number
. . . a number.

natural
. . . a positive integer.

any
The corresponding argument of every occurrence of the constraint can have any type. This is the default value.

:- chr_type(+*TypeDeclaration***)**
User-defined types are algebraic data types, similar to those in Haskell or the discriminated unions in Mercury. An algebraic data type is defined using `chr_type/1`:

```
:- chr_type type ---> body.
```

If the type term is a functor of arity zero (i.e. one having zero arguments), it names a monomorphic type. Otherwise, it names a polymorphic type; the arguments of the functor must be distinct type variables. The body term is defined as a sequence of constructor definitions separated by semi-colons.

Each constructor definition must be a functor whose arguments (if any) are types. Discriminated union definitions must be transparent: all type variables occurring in the body must also occur in the type.

Here are some examples of algebraic data type definitions:

```
:- chr_type color ---> red ; blue ; yellow ; green.

:- chr_type tree --->  empty ; leaf(int) ; branch(tree, tree).

:- chr_type list(T) ---> [] ; [T | list(T)].

:- chr_type pair(T1, T2) ---> (T1 - T2).
```

Each algebraic data type definition introduces a distinct type. Two algebraic data types that have the same bodies are considered to be distinct types (name equivalence).

Constructors may be overloaded among different types: there may be any number of constructors with a given name and arity, so long as they all have different types.

Aliases can be defined using ==. For example, if your program uses lists of lists of integers, you can define an alias as follows:

```
:- chr_type lli == list(list(int)).
```

Type Checking Currently two complementary forms of type checking are performed:

1. Static type checking is always performed by the compiler. It is limited to CHR rule heads and CHR constraint calls in rule bodies.

   Two kinds of type error are detected. The first is where a variable has to belong to two types. For example, in the program:

   ```
   :-chr_type foo ---> foo.
   :-chr_type bar ---> bar.

   :-chr_constraint abc(?foo).
   :-chr_constraint def(?bar).

   foobar @ abc(X) <=> def(X).
   ```

   the variable `X` has to be of both type `foo` and `bar`. This is reported by the type clash error:

   ```
   CHR compiler ERROR:
       `--> Type clash for variable _G5398 in rule foobar:
                   expected type foo in body goal def(_G5398, _G5448)
                   expected type bar in head def(_G5448, _G5398)
   ```

   The second kind of error is where a functor is used that does not belong to the declared type. For example in:

   ```
   :-chr_type foo ---> foo.
   :-chr_type bar ---> bar.

   :-chr_constraint abc(?foo).

   foo @ abc(bar) <=> true.
   ```

   in the head of the rule `bar` appears where something of type `foo` is expected. This is reported as:

   ```
   CHR compiler ERROR:
       `--> Invalid functor in head abc(bar) of rule foo:
                   found `bar',
                   expected type `foo'!
   ```

   No runtime overhead is incurred in static type checking.

2. Dynamic type checking checks at runtime, during program execution, whether the arguments of CHR constraints respect their declared types. The `when/2` co-routining library is used to delay dynamic type checks until variables are instantiated.

   The kind of error detected by dynamic type checking is where a functor is used that does not belong to the declared type. E.g. for the program:

```
:-chr_type foo ---> foo.

:-chr_constraint abc(?foo).
```

we get the following error in an erroneous query:

```
?- abc(bar).
ERROR: Type error: 'foo' expected, found 'bar' (CHR Runtime Type Error)
```

Dynamic type checking is weaker than static type checking in the sense that it only checks the particular program execution at hand rather than all possible executions. It is stronger in the sense that it tracks types throughout the whole program.

Note that it is enabled only in debug mode, as it incurs some (minor) runtime overhead.

7.3.3 Compilation

The SWI-Prolog CHR compiler exploits `term_expansion/2` rules to translate the constraint handling rules to plain Prolog. These rules are loaded from the library `chr`. They are activated if the compiled file has the `.chr` extension or after finding a declaration of the format below.

```
:- chr_constraint ...
```

It is advised to define CHR rules in a module file, where the module declaration is immediately followed by including the library(chr) library as exemplified below:

```
:- module(zebra, [ zebra/0 ]).
:- use_module(library(chr)).

:- chr_constraint ...
```

Using this style CHR rules can be defined in ordinary Prolog .pl files and the operator definitions required by CHR do not leak into modules where they might cause conflicts.

7.4 Debugging

The CHR debugging facilities are currently rather limited. Only tracing is currently available. To use the CHR debugging facilities for a CHR file it must be compiled for debugging. Generating debug info is controlled by the CHR option `debug`, whose default is derived from the SWI-Prolog flag `generate_debug_info`. Therefore debug info is provided unless the `-nodebug` is used.

7.4.1 Ports

For CHR constraints the four standard ports are defined:

call
A new constraint is called and becomes active.

exit
An active constraint exits: it has either been inserted in the store after trying all rules or has been removed from the constraint store.

fail
An active constraint fails.

redo
An active constraint starts looking for an alternative solution.

In addition to the above ports, CHR constraints have five additional ports:

wake
A suspended constraint is woken and becomes active.

insert
An active constraint has tried all rules and is suspended in the constraint store.

remove
An active or passive constraint is removed from the constraint store.

try
An active constraints tries a rule with possibly some passive constraints. The try port is entered just before committing to the rule.

apply
An active constraints commits to a rule with possibly some passive constraints. The apply port is entered just after committing to the rule.

7.4.2 Tracing

Tracing is enabled with the `chr_trace/0` predicate and disabled with the `chr_notrace/0` predicate.

When enabled the tracer will step through the `call`, `exit`, `fail`, `wake` and `apply` ports, accepting debug commands, and simply write out the other ports.

The following debug commands are currently supported:

```
CHR debug options:

        <cr>    creep           c       creep
        s       skip
        g       ancestors
        n       nodebug
        b       break
        a       abort
        f       fail
        ?       help            h       help
```

Their meaning is:

creep
Step to the next port.

skip
Skip to exit port of this call or wake port.

ancestors
Print list of ancestor call and wake ports.

nodebug
Disable the tracer.

break
Enter a recursive Prolog top-level. See `break/0`.

abort
Exit to the top-level. See `abort/0`.

fail
Insert failure in execution.

help
Print the above available debug options.

7.4.3 CHR Debugging Predicates

The `chr` module contains several predicates that allow inspecting and printing the content of the constraint store.

chr_trace
Activate the CHR tracer. By default the CHR tracer is activated and deactivated automatically by the Prolog predicates `trace/0` and `notrace/0`.

chr_notrace
De-activate the CHR tracer. By default the CHR tracer is activated and deactivated automatically by the Prolog predicates `trace/0` and `notrace/0`.

chr_leash(+*Spec*)
Define the set of CHR ports on which the CHR tracer asks for user intervention (i.e. stops). *Spec* is either a list of ports as defined in section 7.4.1 or a predefined 'alias'. Defined aliases are: `full` to stop at all ports, `none` or `off` to never stop, and `default` to stop at the `call`, `exit`, `fail`, `wake` and `apply` ports. See also `leash/1`.

chr_show_store(+*Mod*)
Prints all suspended constraints of module *Mod* to the standard output. This predicate is automatically called by the SWI-Prolog top-level at the end of each query for every CHR module currently loaded. The prolog-flag `chr_toplevel_show_store` controls whether the top-level shows the constraint stores. The value `true` enables it. Any other value disables it.

7.5 Examples

Here are two example constraint solvers written in CHR.

- The program below defines a solver with one constraint, `leq/2/,` which is a less-than-or-equal constraint, also known as a partial order constraint.

```
:- module(leq,[leq/2]).
:- use_module(library(chr)).

:- chr_constraint leq/2.
reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

  When the above program is saved in a file and loaded in SWI-Prolog, you can call the `leq/2` constraints in a query, e.g.:

```
?- leq(X,Y), leq(Y,Z).
leq(_G23837, _G23841)
leq(_G23838, _G23841)
leq(_G23837, _G23838)

X = _G23837{leq = ...}
Y = _G23838{leq = ...}
Z = _G23841{leq = ...}

Yes
```

  When the query succeeds, the SWI-Prolog top-level prints the content of the CHR constraint store and displays the bindings generate during the query. Some of the query variables may have been bound to attributed variables, as you see in the above example.

- The program below implements a simple finite domain constraint solver.

```
:- module(dom,[dom/2]).
:- use_module(library(chr)).

:- chr_constraint dom(?int,+list(int)).
:- chr_type list(T) ---> [] ; [T|list(T)].

dom(X,[]) <=> fail.
dom(X,[Y]) <=> X = Y.
dom(X,L) <=> nonvar(X) | memberchk(X,L).
dom(X,L1), dom(X,L2) <=> intersection(L1,L2,L3), dom(X,L3).
```

When the above program is saved in a file and loaded in SWI-Prolog, you can call the `dom/2` constraints in a query, e.g.:

```
?- dom(A,[1,2,3]), dom(A,[3,4,5]).

A = 3

Yes
```

7.6 Backwards Compatibility

There are small differences between the current K.U.Leuven CHR system in SWI-Prolog, older versions of the same system and SICStus' CHR system.

The current system maps old syntactic elements onto new ones and ignores a number of no longer required elements. However, for each a *deprecated* warning is issued. You are strongly urged to replace or remove deprecated features.

Besides differences in available options and pragmas, the following differences should be noted:

- *The* `constraints/1` *declaration*
  This declaration is deprecated. It has been replaced with the `chr_constraint/1` declaration.
- *The* `option/2` *declaration*
  This declaration is deprecated. It has been replaced with the `chr_option/2` declaration.
- *The* `handler/1` *declaration*
  In SICStus every CHR module requires a `handler/1` declaration declaring a unique handler name. This declaration is valid syntax in SWI-Prolog, but will have no effect. A warning will be given during compilation.
- *The* `rules/1` *declaration*
  In SICStus, for every CHR module it is possible to only enable a subset of the available rules through the `rules/1` declaration. The declaration is valid syntax in SWI-Prolog, but has no effect. A warning is given during compilation.
- *Guard bindings*
  The `check_guard_bindings` option only turns invalid calls to unification into failure. In SICStus this option does more: it intercepts instantiation errors from Prolog built-ins such as `is/2` and turns them into failure. In SWI-Prolog, we do not go this far, as we like to separate concerns more. The CHR compiler is aware of the CHR code, the Prolog system and programmer should be aware of the appropriate meaning of the Prolog goals used in guards and bodies of CHR rules.

7.7 Programming Tips and Tricks

In this section we cover several guidelines on how to use CHR to write constraint solvers and how to do so efficiently.

- *Check guard bindings yourself*
  It is considered bad practice to write guards that bind variables of the head and to rely on the system to detect this at runtime. It is inefficient and obscures the working of the program.

- *Set semantics*
  The CHR system allows the presence of identical constraints, i.e. multiple constraints with the same functor, arity and arguments. For most constraint solvers, this is not desirable: it affects efficiency and possibly termination. Hence appropriate simpagation rules should be added of the form:
  $$constraint \backslash constraint <=> true$$

- *Multi-headed rules*
  Multi-headed rules are executed more efficiently when the constraints share one or more variables.

- *Mode and type declarations*
  Provide mode and type declarations to get more efficient program execution. Make sure to disable debug (-nodebug) and enable optimization (-O).

- *Compile once, run many times*
  Does consulting your CHR program take a long time in SWI-Prolog? Probably it takes the CHR compiler a long time to compile the CHR rules into Prolog code. When you disable optimizations the CHR compiler will be a lot quicker, but you may loose performance. Alternatively, you can just use SWI-Prolog's `qcompile/1` to generate a `.qlf` file once from your `.pl` file. This `.qlf` contains the generated code of the CHR compiler (be it in a binary format). When you consult the `.qlf` file, the CHR compiler is not invoked and consultation is much faster.

7.8 Compiler Errors and Warnings

In this section we summarize the most important error and warning messages of the CHR compiler.

7.8.1 CHR Compiler Errors

Type clash for variable ... in rule ...

This error indicates an inconsistency between declared types; a variable should belong to two types. See static type checking.

Invalid functor in head ... of rule ...

This error indicates an inconsistency between a declared type and the use of a functor in a rule. See static type checking.

Cyclic alias definition: ... == ...

You have defined a type alias in terms of itself, either directly or indirectly.

Ambiguous type aliases You have defined two overlapping type aliases.

Multiple definitions for type

You have defined the same type multiple times.

Non-ground type in constraint definition: ...

You have declared a non-ground type for a constraint argument.

Could not find type definition for ...

You have used an undefined type in a type declaration.

Illegal mode/type declaration You have used invalid syntax in a constraint declaration.

Constraint multiply defined There is more than one declaration for the same constraint.

Undeclared constraint ... in head of ...

You have used an undeclared constraint in the head of a rule. This often indicates a misspelled constrained name or wrong number of arguments.

Invalid pragma ... in ... Pragma should not be a variable.

You have used a variable as a pragma in a rule. This is not allowed.

Invalid identifier ... in pragma passive in ...

You have used an identifier in a passive pragma that does not correspond to an identifier in the head of the rule. Likely the identifier name is misspelled.

Unknown pragma ... in ...

You have used an unknown pragma in a rule. Likely the pragma is misspelled or not supported.

Something unexpected happened in the CHR compiler

You have most likely bumped into a bug in the CHR compiler. Please contact Tom Schrijvers to notify him of this error.

8 Multi-threaded applications

SWI-Prolog multithreading is based on standard C-language multithreading support. It is not like *ParLog* or other parallel 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.[1]
- *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. See also section 8.7.
- *Natural integration with foreign code*
  Each Prolog thread runs in a native thread of the operating system, 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.

SWI-Prolog multi-threading is based on the POSIX thread standard [Butenhof, 1997] used on most popular systems except for MS-Windows. On Windows it uses the pthread-win32 emulation of POSIX threads mixed with the Windows native API for smoother and faster operation.

8.1 Creating and destroying Prolog threads

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 successfully, the thread-identifier of the created thread is unified to *Id*. *Options* is a list of options. The currently defined options are below. Stack size options can also take the value `inf` or `infinite`, which is mapped to the maximum stack size supported by the platform.

local(*K-Bytes*)
Set the limit to which the local stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the `-L` command-line option.

[1] On an dual AMD-Athlon 1600, SWI-Prolog 5.1.0 creates and joins 4,957 threads per second elapsed time.

global(*K-Bytes*)
Set the limit to which the global stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the `-G` command-line option.

trail(*K-Bytes*)
Set the limit to which the trail stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the `-T` command-line option.

argument(*K-Bytes*)
Set the limit to which the argument stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the `-A` command-line option.

stack(*K-Bytes*)
Set the limit to which the system stack of this thread may grow. The default, minimum and maximum values are system-dependant.

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 note that thread identifiers are freed for reuse after a detached thread finishes or a normal thread has been joined. See also `thread_join/2` and `thread_detach/1`.

If a detached thread dies due to failure or exception of the initial goal the thread prints a message using `print_message/2`. If such termination is considered normal the code must be wrapped using `ignore/1` and/or `catch/3` to ensure successful completion.

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.

thread_join(*+Id, -Status*)
Wait for the termination of thread with given *Id*. Then unify the result-status of the thread with *Status*. After this call, *Id* becomes invalid and all resources associated with the thread are reclaimed. Note that threads with the attribute `detached`(*true*) cannot be joined. 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 representing the exit-status of the thread is retained until `thread_join/2` is called on the thread. Defined values for *Status* are:

true
The goal has been proven successfully.

false
The goal has failed.

exception(*Term*)
The thread is terminated on an exception. See `print_message/2` to turn system exceptions into readable messages.

exited(*Term*)
The thread is terminated on `thread_exit/1` using the argument *Term*.

thread_detach(+*Id*)
Switch thread into detached-state (see `detached`(*Bool*) option at `thread_create/3`) at runtime. *Id* is the identifier of the thread placed in detached state. This may be the result of PL_thread_self/1.

One of the possible applications is to simplify debugging. Threads that are created as *detached* leave no traces if they crash. For not-detached threads the status can be inspected using `current_thread/2`. Threads nobody is waiting for may be created normally and detach themselves just before completion. This way they leave no traces on normal completion and their reason for failure can be inspected.

thread_exit(+*Term*)
Terminates the thread immediately, leaving `exited`(*Term*) as result-state for `thread_join/2`. If the thread has the attribute `detached`(*true*) it terminates, but its exit status cannot be retrieved using `thread_join/2` making the value of *Term* irrelevant. The Prolog stacks and C-thread are reclaimed.

thread_initialization(*:Goal*)
Run *Goal* when thread is started. This predicate must be compared with `initialization/1`, but is intended for initialization operations of the runtime stacks, such as setting global variables as described in section 6.3. *Goal* is run on four occasions: at the call to this predicate, after loading a saved state, on starting a new thread and on creating a Prolog engine through the C interface. On loading a saved state, *Goal* is executed *after* running the `initialization/1` hooks.

thread_at_exit(*:Goal*)
Run *Goal* just before releasing the thread resources. 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` using the result of `thread_self/1` as thread-identifier.

thread_setconcurrency(*-Old, +New*)
Determine the concurrency of the process, which is defined as the maximum number of concurrently active threads. 'Active' here means they are using CPU time. This option is provided if the thread-implementation provides pthread_setconcurrency(). Solaris is a typical example of this family. On other systems this predicate unifies *Old* to 0 (zero) and succeeds silently.

8.2 Monitoring threads

Normal multi-threaded applications should not need these the predicates from this section because almost any usage of these predicates is unsafe. For example checking the existence of a thread before

signalling it is of no use as it may vanish between the two calls. Catching exceptions using `catch/3` is the only safe way to deal with thread-existence errors.

These predicates are provided for diagnosis and monitoring tasks. See also section 8.5, describing more high-level primitives.

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 threads 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. If the underlying native thread has exited (using pthread_exit()) *Term* is unbound.

exception(*Term*)
The *Goal* of the thread has been terminated due to an uncaught exception (see `throw/1` and `catch/3`).

thread_statistics(*+Id, +Key, -Value*)
Obtains statistical information on thread *Id* as `statistics/2` does in single-threaded applications. This call returns all keys of `statistics/2`, although only information statistics about the stacks and CPU time yield different values for each thread.[2]

mutex_statistics
Print usage statistics on internal mutexes and mutexes associated with dynamic predicates. For each mutex two numbers are printed: the number of times the mutex was acquired and the number of *collisions*: the number times the calling thread has to wait for the mutex. The collision-count is not available on Windows as this would break portability to Windows-95/98/ME or significantly harm performance. Generally collision count is close to zero on single-CPU hardware.

8.2.1 Linux: linuxthreads vs. NPTL

Linux has introduces POSIX threads (pthread) using an implementation called *linuxthreads*, where each thread was 'almost' a process. This approach has various disadvantages, such as poor performance and non-compliance with several aspects of POSIX. However, there is one advantage. Where

[2]Getting the CPU-time of a different thread is not supported on all platforms. For Microsoft, it does not work in 95/98/ME. For POSIX systems it requires times() to return values specific for the calling thread. See also section 8.2.1.

pthread does not provide a way to get statistics per thread, we could get this info from the process-oriented times() function. Since the 2.6.x kernels, Linux by default now uses the *NPTL* implementation which is POSIX compliant. Unfortunately, getting per-thread CPU statistics involves reading /proc and is therefore too slow for some applications.

SWI-Prolog is setup to run with the default thread model. Unfortunately there is no way to modify this at runtime, but there is a way to select the old thread model on modern machines at *link time*. This is achieved using the environment variable `LD_ASSUME_KERNEL` which must be set to a pre-nptl kernel version for linking the main executable. The value `2.4.21` is appropriate. When building from source, this flag can be set during the build process. When using a binary distribution one could create a minimal C program and relink the system using the plld utility.

8.3 Thread communication

8.3.1 Message queues

Prolog threads can exchange data using dynamic predicates, database records, and other globally shared data. These provide no suitable means to wait for data or a condition as they can only be checked in an expensive polling loop. *Message queues* provide a means for threads to wait for data or conditions without using the CPU.

Each thread has a message-queue attached to it that is identified by the thread. Additional queues are created using `message_queue_create/1`.

thread_send_message(+*QueueOrThreadId, +Term*)

Place *Term* in the given queue or default 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 the receiving thread and variable-bindings are thus lost. This call returns immediately.

If more than one thread is waiting for messages on the given queue and at least one of these is waiting with a partially instantiated *Term*, the waiting threads are *all* sent a wake-up signal, starting a rush for the available messages in the queue. This behaviour can seriously harm performance with many threads waiting on the same queue as all-but-the-winner perform a useless scan of the queue. If there is only one waiting thread or all waiting threads wait with an unbound variable an arbitrary thread is restarted to scan the queue.[3]

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*, the 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)),
```

[3] See the documentation for the POSIX thread functions pthread_cond_signal() v.s. pthread_cond_broadcastt() for background information.

```
<thread 2>
thread_send_message(Thread_1, b(gnu)),
thread_send_message(Thread_1, 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.

message_queue_create(*?Queue*)
If *Queue* is an atom, create a named queue. To avoid ambiguity of `thread_send_message/2`, the name of a queue may not be in use as a thread-name. If *Queue* is unbound an anonymous queue is created and *Queue* is unified to its identifier.

message_queue_destroy(+*Queue*)
Destroy a message queue created with `message_queue_create/1`. It is *not* allows to destroy the queue of a thread. Neither is it allowed to destroy a queue other threads are waiting for or, for anonymous message queues, may try to wait for later.[4]

thread_get_message(+*Queue, ?Term*)
As `thread_get_message/1`, operating on a given queue. It is allowed (but not advised) to get messages from the queue of other threads.

thread_peek_message(+*Queue, ?Term*)
As `thread_peek_message/1`, operating on a given queue. It is allowed to peek into another thread's message queue, an operation that can be used to check whether a thread has swallowed a message sent to it.

message_queue_size(+*Queue, -Size*)
Unify *Size* with the number of terms waiting in *Queue*. Note that due to concurrent access the returned value may be outdated before it is returned. It can be used for debugging purposes as well as work distribution purposes.

Explicit message queues are designed with the *worker-pool* model in mind, where multiple threads wait on a single queue and pick up the first goal to execute. Below is a simple implementation where the workers execute arbitrary Prolog goals. Note that this example provides no means to tell when all work is done. This must be realised using additional synchronisation.

```
%       create_workers(+Id, +N)
%
%       Create a pool with given Id and number of workers.

create_workers(Id, N) :-
        message_queue_create(Id),
        forall(between(1, N, _),
```

[4] BUG: None of these constraints are properly enforced by the system in the current implementation. It is therefore advised not to delete queues unless you are absolutely sure it is safe.

```
                thread_create(do_work(Id), _, [])).

do_work(Id) :-
        repeat,
          thread_get_message(Id, Goal),
          (   catch(Goal, E, print_message(error, E))
          ->  true
          ;   print_message(error, goal_failed(Goal, worker(Id)))
          ),
        fail.

%       work(+Id, +Goal)
%
%       Post work to be done by the pool

work(Id, Goal) :-
        thread_send_message(Id, Goal).
```

8.3.2 Signalling threads

These predicates provide a mechanism to make another thread execute some goal as an *interrupt*. Signalling threads is safe as these interrupts are only checked at safe points in the virtual machine. Nevertheless, signalling in multi-threaded environments should be handled with care as the receiving thread may hold a *mutex* (see with_mutex). Signalling probably only makes sense to start debugging threads and to cancel no-longer-needed threads with `throw/1`, where the receiving thread should be designed carefully do handle exceptions at any point.

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.

In the Windows version, the receiving thread immediately executes the signal if it reaches a Windows GetMessage() call, which generally happens of the thread is waiting for (user-)input.

8.3.3 Threads and dynamic predicates

Besides queues (section 8.3.1) threads can share and exchange data using dynamic predicates. The multi-threaded version knows about two types of dynamic predicates. By default, a predicate declared *dynamic* (see `dynamic/1`) is shared by all threads. Each thread may assert, retract and run the dynamic predicate. Synchronisation inside Prolog guarantees the consistency of the predicate. Updates are *logical*: visible clauses are not affected by assert/retract after a query started on the predicate. In

many cases primitive from section 8.4 should be used to ensure application invariants on the predicate are maintained.

Besides shared predicates, dynamic predicates can be declared with the `thread_local/1` directive. Such predicates share their attributes, but the clause-list is different in each thread.

thread_local *+Functor/+Arity, ...*

This directive is related to the `dynamic/1` directive. It tells the system that the predicate may be modified using `assert/1`, `retract/1`, etc. during execution of the program. Unlike normal shared dynamic data however each thread has its own clause-list for the predicate. As a thread starts, this clause list is empty. If there are still clauses as the thread terminates these are automatically reclaimed by the system (see also `volatile/1`). The thread_local property implies the properties dynamic and volatile.

Thread-local dynamic predicates are intended for maintaining thread-specific state or intermediate results of a computation.

It is not recommended to put clauses for a thread-local predicate into a file as in the example below as the clause is only visible from the thread that loaded the source-file. All other threads start with an empty clause-list.

```
:- thread_local
        foo/1.

foo(gnat).
```

DISCLAIMER Whether or not this declaration is appropriate in the sense of the proper mechanism to reach the goal is still debated. If you have strong feeling in favour or against, please share them in the SWI-Prolog mailing list.

8.4 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.

with_mutex(*+MutexId, :Goal*)
Execute *Goal* while holding *MutexId*. If *Goal* leaves choice-points, 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/1` and `call_cleanup/3`.

Although described in the thread-section, this predicate is also available in the single-threaded version, where it behaves simply as `once/1`.

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`, which provides a safer way for handling Prolog-level mutexes. The predicate `call_cleanup/[2-3]` is another way to guarantee that the mutex is unlocked while retaining non-determinism.

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 the holding thread and *Count* with the recursive count of the mutex. Otherwise, *ThreadId* is `[]` and *Count* is 0.

8.5 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.

join_threads
Join all terminated threads. For normal applications, dealing with terminated threads must be part of the application logic, either detaching the thread before termination or making sure it will be joined. The predicate `join_threads/0` is intended for interactive sessions to reclaim resources from threads that died unexpectedly during development.

interactor
Create a new console and run the Prolog top-level in this new console. See also `attach_console/0`. In the Windows version a new interactor can also be created from the Run/New thread menu.

8.5.1 Debugging threads

Support for debugging threads is still very limited. Debug and trace mode are flags that are local to each thread. Individual threads can be debugged either using the graphical debugger described in section 3.5 (see `tspy/1` and friends) or by attaching a console to the thread and running the traditional command-line debugger (see `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. On Unix systems the console is an `xterm` application. On Windows systems this requires the GUI version `plwin.exe` rather than the console based `plcon.exe`.

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)).
```

tdebug(+*ThreadId*)
Prepare *ThreadId* for debugging using the graphical tracer. This implies installing the tracer hooks in the thread and switching the thread to debug-mode using `debug/0`. The call is injected into the thread using `thread_signal/2`. We refer to the documentation of this predicate for asynchronous interaction with threads. New threads created inherit their debug-mode from the thread that created them.

tdebug
Call `tdebug/1` in all running threads.

tnodebug(+*ThreadId*)
Disable debugging thread *ThreadId*.

tnodebug
Disable debugging in all threads.

tspy(:*Spec, +ThreadId*)
Set a spy-point as `spy/1` and enable the thread for debugging using `tdebug/1`. Note that a spy-point is a global flag on a predicate that is visible from all threads. Spy points are honoured in all threads that are in debug-mode and ignored in threads that are in nodebug mode.

tspy(:*Spec*)
Set a spy-point as `spy/1` and enable debugging in all threads using `tdebug/0`. Note that removing spy-points can be done using `nospy/1`. Disabling spy-points in a specific thread is achieved by `tnodebug/1`.

8.5.2 Profiling threads

In the current implementation, at most one thread can be profiled at any moment. Any thread can call `profile/1` to profile the execution of some part of its code. The predicate `tprofile/1` allows for profiling the execution of another thread until the user stops collecting profile data.

tprofile(+*ThreadId*)
Start collecting profile data in *ThreadId* and ask the user to hit ⟨*return*⟩ to stop the profiler. See section 4.40 for details on the execution profiler.

8.6 Multi-threaded mixed C and Prolog applications

All foreign-code linked to the multi-threading version of SWI-Prolog should be thread-safe (*reentrant*) or guarded in Prolog using `with_mutex/2` from simultaneous access from multiple Prolog threads. If you want to write mixed multi-threaded C and Prolog application you should first familiarise yourself with writing multi-threaded applications in C (C++).

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()).

The system supports two models. Section 8.6.1 describes the original one-to-one mapping. In this schema a native thread attaches a Prolog thread if it needs to call Prolog and detaches is when finished, as opposed to the model from section 8.6.2 where threads temporary use a Prolog engine.

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.

8.6.1 A Prolog thread for each native thread (one-to-one)

In the one-to-one model, the thread that called PL_initialise() has a Prolog engine attached. If another C-thread in the system wishes to call Prolog it must first attach an engine using

PL_thread_attach_engine() and call PL_thread_destroy_engine() after all Prolog work is finished. This model is especially suitable with long running threads that need to do Prolog work regularly. See section 8.6.2 for the alternative many-to-many model.

`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**(*const 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. The `cancel` field may be filled with a pointer to a function that is called when PL_cleanup() terminates the running Prolog engines. If this function is not present or returns `FALSE` pthread_cancel() 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 */
  int              (*cancel)(int thread);
} PL_thread_attr_t;
```

The structure may be destroyed after PL_thread_attach_engine() has returned. On success it returns the Prolog identifier for the thread (as returned by PL_thread_self()). 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.

`int` **PL_thread_at_exit**(*void (*function)(void *), void *closure, int global*)
Register a handle to be called as the Prolog engine is destroyed. The handler function is called with one `void *` argument holding *closure*. If *global* is `TRUE`, the handler is installed *for all threads*. Globally installed handlers are executed after the thread-local handlers. If the handler is installed local for the current thread only (*global* == `FALSE`) it is stored in the same FIFO queue as used by `thread_at_exit/1`.

8.6.2 Pooling Prolog engines (many-to-many)

In this model Prolog engines live as entities that are independent from threads. If a thread needs to call Prolog it takes one of the engines from the pool and returns the engine when done. This model is suitable in the following identified cases:

- *Compatibility with the single-threaded version*
  In the single-threaded version, foreign threads must serialise access the the one and only thread engine. Functions from this section allow sharing one engine among multiple threads.

- *Many native threads with infrequent Prolog work*
  Prolog threads are expensive in terms of memory and time to create and destroy them. Systems that use a large number of threads that only infrequently need to call Prolog are better take an engine from a pool and return it there.

- *Prolog status must be handed to another thread*
  This situation has been identified by Uwe Lesta when creating a .NET interface for SWI-Prolog. .NET distributes work for active internet connection over a pool of threads. If a Prolog engine contains state for a connection, it must be possible to detach the engine from a thread and re-attach it to another thread handling the same connection.

`PL_engine_t` **PL_create_engine**(*PL_thread_attr_t *attributes*)
Create a new Prolog engine. *attributes* is described with PL_thread_attach_engine(). Any thread can make this call after PL_initialise() returned success. The returned engine is not attached to any thread and lives until PL_destroy_engine() is used on the returned handle.

In the single-threaded version this call always returns `NULL`, indicating failure.

`int` **PL_destroy_engine**(*PL_engine_t e*)
Destroy the given engine. Destroying an engine is only allowed if the engine is not attached to any thread or attached to the calling thread. On success this function returns `TRUE`, on failure the return value is `FALSE`.

`int` **PL_set_engine**(*PL_engine_t engine, PL_engine_t *old*)
Make the calling thread ready to use *engine*. If *old* is non-`NULL` the current engine associated with the calling thread is stored at the given location. If *engine* equals `PL_ENGINE_MAIN` the initial engine is attached to the calling thread. If *engine* is `PL_ENGINE_CURRENT` the engine is not changed. This can be used to query the current engine. This call returns `PL_ENGINE_SET` if the engine was switched successfully, `PL_ENGINE_INVAL` if *engine* is not a valid engine handle and `PL_ENGINE_INUSE` if the engine is currently in use by another thread.

Engines can be changed at any time. For example, it is allowed to select an engine to initiate a Prolog goal, detach it and at a later moment execute the goal from another thread. Note however that the `term_t`, `qid_t` and `fid_t` types are interpreted relative to the engine for which they are created. Behaviour when passing one of these types from one engine to another is undefined.

In the single-threaded version this call only succeeds if *engine* refers to the main engine.

Engines in single-threaded SWI-Prolog

In theory it is possible to port the API of section 8.6.2 to the single-threaded version of SWI-Prolog. This allows C-programs to control multiple Prolog engines concurrently. This has not yet been realised.

8.7 Multithreading and the XPCE graphics system

GUI applications written in XPCE can benefit from the multi-threaded version of XPCE/SWI-Prolog if they need to do expensive computations that block to UI in the single-threaded version.

Due to various technical problems on both Windows and Unix/X11 threading is best exploited by handing long computations to their own thread.

The XPCE message passing system is guarded with a single *mutex*, which synchronises both access from Prolog and activation through the GUI. In MS-Windows, GUI events are processed by the thread that created the window in which the event occurred, whereas in Unix/X11 they are processed by the thread that dispatches messages.

Some tentative work is underway to improve the integration between XPCE and multi-threaded SWI-Prolog. There are two sets of support predicates. The first model assumes that XPCE is running in the main thread and background threads are used for computation. In the second model, XPCE event dispatching runs in the background, while the foreground thread is used for Prolog.

XPCE in the foreground Using XPCE in the foreground simplifies debugging of the UI and generally provides the most comfortable development environment. The GUI creates new threads using `thread_create/3` and, after work in the thread is completed, the sub-thread signals the main thread of the the completion using `in_pce_thread/1`.

in_pce_thread(*:Goal*)
: Assuming XPCE is running in the foreground thread, this call gives background threads the opportunity to make calls to the XPCE thread. A call to `in_pce_thread/1` succeeds immediately, copying *Goal* to the XPCE thread. *Goal* is added to the XPCE event-queue and executed synchronous to normal user events like typing and clicking.

XPCE in the background In this model a thread for running XPCE is created using `pce_dispatch/1` and actions are sent to this thread using `pce_call/1`.

pce_dispatch(+*Options*)
: Create a Prolog thread with the alias-name `pce` for XPCE event-handling. In the X11 version this call creates a thread that executes the X11 event-dispatch loop. In MS-Windows it creates a thread that executes a windows event-dispatch loop. The XPCE event-handling thread has the alias `pce`. *Options* specifies the thread-attributes as `thread_create/3`.

pce_call(*:Goal*)
: Post *Goal* to the `pce` thread, executing it synchronous with the thread's event-loop. The `pce_call/1` predicate returns immediately without waiting. Note that *Goal* is *copied* to the `pce` thread.

For further information about XPCE in threaded applications, please visit `http://gollem.science.uva.nl/twiki/pl/bin/view/Development/MultiThreadsXPCE`

9 Foreign Language Interface

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.

9.1 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.)

9.2 Linking Foreign Modules

Foreign modules may be linked to Prolog in two ways. Using *static linking*, the extensions, a (short) file defining main() which attaches the extensions calls Prolog and the SWI-Prolog kernel distributed as a C-library 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].

9.2.1 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 9.4 for a suitable high-level interface to these predicates.

9.2.2 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 9.7 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.

9.3 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 9.4 in your application.

open_shared_object(+*File, -Handle*)

File is the name of a shared object file (called dynamic load library in MS-Windows). This file is attached to the current process and *Handle* is unified with a handle to the library. Equivalent to `open_shared_object(File, [], 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, -Handle, +Options*)

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().

9.4 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 9.1 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 9.1: MessageBox() example in Windows NT

9.4.1 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
`.../pl/include/SWI-Prolog.h`	Include file
`.../pl/include/SWI-Stream.h`	Stream I/O include file
`.../pl/include/SWI-Exports`	Export declarations (AIX only)
`.../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:

```
#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 }     /* terminating 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 required */

  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. If the linking is performed 'by hand', the command-line option `-dump-runtime-variables` (see section 2.4) can be used to obtain the required paths, libraries and linking options to link the new executable.

9.5 Interface Data types

9.5.1 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 9.3.

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.

9.5.2 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.

int64_t Actually part of the C99 standard rather than Prolog. As of version 5.5.6, Prolog integers are 64-bit on all hardware. The C99 type int64_t is defined in the `stdint.h` standard header and provides platform independent 64-bit integers. Portable code accessing Prolog should use this type to exchange integer values. Please note that PL_get_long() can return `FALSE` on Prolog integers outside the long domain. Robust code should not assume any of the integer fetching functions to succeed if the Prolog term is know to be an integer.

9.6 The Foreign Include File

9.6.1 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:

(return) foreign_t **PL_succeed()**
Succeed deterministically. PL_succeed is defined as `return TRUE`.

(return) foreign_t **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.

(return) foreign_t **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() returns the handle passed via PL_retry(). This handle is a 30 bits signed value (two bits are used for status indication). Defined as `return _PL_retry(n)`. See also PL_succeed().

(return) foreign_t **PL_retry_address**(*void **)
As PL_retry(), but ensures an address as returned by malloc() is correctly recovered by PL_foreign_context_address(). Defined as `return _PL_retry_address(n)`. See also PL_succeed().

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 9.2 shows a skeleton for a non-deterministic foreign predicate definition.

9.6.2 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 9.6.2).

```
typedef struct                  /* define a context structure */
{ ...
} context;

foreign_t
my_function(term_t a0, term_t a1, control_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:
        ctxt = PL_foreign_context_address(handle);
        ...
        free(ctxt);
        PL_succeed;
  }
}
```

Figure 9.2: Skeleton for non-deterministic foreign functions

`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. The returned string becomes invalid if the atom is garbage-collected (see section 9.6.2). Foreign functions that require the text from an atom passed in a `term_t` normally use PL_get_atom_chars() or PL_get_atom_nchars().

`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 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 8, 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 advised to use the *_chars() or *_nchars() functions whenever applicable.

9.6.3 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 destroy 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_ge_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_ground**(*term_t*)

Returns non-zero if *term* is a ground term. See also `ground/1`. This function is cycle-safe.

`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 PL_free() when not needed any longer.

With the introduction of wide-characters (see section 2.17.1), not all atoms can be converted into a `char*`. This function fails if *t* is of the wrong type, but also if the text cannot be represented. See the `REP_*` flags below for details.

`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`
`CVT_EXCEPTION`	If conversion fails due to a type error, raise a Prolog type error exception in addition to failure
`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 `PL_malloc(3)`. When no longer needed the user must call PL_free() on the data.
`REP_ISO_LATIN_1`	(0, default). Text is in ISO Latin-1 encoding and the call fails if text cannot be represented.
`REP_UTF8`	Convert the text to a UTF-8 string. This works for all text.
`REP_MB`	Convert to default locale-defined 8-bit string. Success depends on the locale. Conversion is done using the wcrtomb() C-library function.

`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 that can be represented as a long, assign its value over *i*. If *t* is an integer that cannot be represented by a C long, this function returns `FALSE`. If *t* is a floating point number that can be represented as a long, this function succeeds as well. See also PL_get_int64()

`int` **PL_get_int64**(*term_t* +*t, int64_t *i*)

If *t* is a Prolog integer or float that can be represented as a `int64_t`, assign its value over

i. Currently all Prolog integers can be represented using this type, but this might change if SWI-Prolog introduces unbounded integers.

`int` **PL_get_bool**(*term_t +t, int *val*)
If *t* has the value `true` or `false`, set *val* to the C constant `TRUE` or `FALSE` and return success. otherwise return failure.

`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*)
See PL_get_atom_chars().

`int` **PL_get_list_nchars**(*term_t t, unsigned int *len, char **s*)
See PL_get_list_chars().

`int` **PL_get_nchars**(*term_t t, unsigned int *len, char **s, unsigned int flags*)
See PL_get_chars().

`int` **PL_put_atom_nchars**(*term_t t, unsigned int len, const char *s*)
See PL_put_atom_chars().

`int` **PL_put_string_nchars**(*term_t t, unsigned int len, const char *s*)
See PL_put_string_chars().

`int` **PL_put_list_ncodes**(*term_t t, unsigned int len, const char *s*)
See PL_put_list_codes().

`int` **PL_put_list_nchars**(*term_t t, unsigned int len, const char *s*)
See PL_put_list_chars().

`int` **PL_unify_atom_nchars**(*term_t t, unsigned int len, const char *s*)
See PL_unify_atom_chars().

`int` **PL_unify_string_nchars**(*term_t t, unsigned int len, const char *s*)
See PL_unify_string_chars().

`int` **PL_unify_list_ncodes**(*term_t t, unsigned int len, const char *s*)
See PL_unify_codes().

`int` **PL_unify_list_nchars**(*term_t t, unsigned int len, const char *s*)
See PL_unify_list_chars().

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.

Wide character versions

Support for exchange of wide character strings is still under considerations. The functions dealing with 8-bit character strings return failure when operating on a wide character atom or Prolog string object. The functions below can extract and unify bith 8-bit and wide atoms and string objects. Wide character strings are represented as C arrays of objects of the type `pl_wchar_t`, which is guaranteed to be the same as `wchar_t` on platforms supporting this type. For example, on MS-Windows, this represents 16-bit UCS2 characters, while using the GNU C library (glibc) this represents 32-bit UCS4 characters.

`atom_t` **PL_new_atom_wchars**(*int len, const pl_wchar_t *s*)
Create atom from wide-character string as PL_new_atom_nchars() does for ISO-Latin-1 strings. It *s* only contains ISO-Latin-1 characters a normal byte-array atom is created.

`pl_wchar_t*` **PL_atom_wchars**(*atom_t atom, int *len*)
Extract characters from a wide-character atom. Fails (returns `NULL`) if *atom* is not a wide-character atom. This is the wide-character version of PL_atom_nchars(). Note that only one of these functions succeeds on a particular atom. Especially, after creating an atom with PL_new_atom_wchars(), extracting the text using PL_atom_wchars() will fail of the atom only contains ISO-Latin-1 characters.

`int` **PL_get_wchars**(*term_t t, unsigned int *len, pl_wchar_t **s, unsigned flags*)
Wide-character version of PL_get_chars(). The *flags* argument is the same as for PL_get_chars().

`int` **PL_unify_wchars**(*term_t t, int type, unsigned int len, const pl_wchar_t *s*)
Unify *t* with a textual representation of the C wide character array *s*. The *arg*type argument defines the Prolog representation and is one of `PL_ATOM`, `PL_STRING`, `PL_CODE_LIST` or `PL_CHAR_LIST`.

`int` **PL_unify_wchars_diff**(*term_t +t, term_t -tail, int type, unsigned int len, const pl_wchar_t *s*)
Difference list version of PL_unify_wchars(), only supporting the types `PL_CODE_LIST` and `PL_CHAR_LIST`. It serves two purposes. It allows for returning very long lists from data read from a stream without the need for a resizing buffer in C and the use of difference lists is often practical for further processing in Prolog. Examples can be found in `packages/clib/readutil.c` from the source distribution.

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 elements */
  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 9.3 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.

9.6.4 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 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 9.6.19.

`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_int64**(*term_t -t, int64_t 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.

```
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 9.3: A Foreign definition of `display/1`

`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();
  functor_t animal2;

  /* animal2 is a constant that may be bound to a global
     variable and re-used
  */
  animal2 = PL_new_functor(PL_new_atom("animal"), 2);

  PL_put_atom_chars(a1, "gnu");
  PL_put_integer(a2, 50);
  PL_cons_functor(t, animal2, 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.

9.6.5 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, tmp);
  }

  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_chars**(*term_t ?t, int flags, unsigned int len, const char *chars*)
New function do deal with unification of `char*` with various encodings to a Prolog representation. The *flags* argument is a bitwise *or* specifying the Prolog target type and the encoding of *chars*. Prolog types is one of `PL_ATOM`, `PL_STRING`, `PL_CODE_LIST` or `PL_CHAR_LIST`. Representations is one of `REP_ISO_LATIN_T`, `REP_UTF8` or `REP_MB`. See PL_get_chars() for a definition of the representation types. If *len* is `-1`, *chars* is assumed to be null-terminated.

`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 9.6.19.

`int` **PL_unify_integer**(*term_t ?t, long n*)
Unify *t* with a Prolog integer from *n*.

`int` **PL_unify_int64**(*term_t ?t, int64_t 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.

Special attention is required when passing numbers. C 'promotes' any integral smaller than `int` to `int`. I.e. the types `char`, `short` and `int` are all passed as `int`. In addition, on most 32-bit platforms `int` and `long` are the same. Up-to version 4.0.5, only `PL_INTEGER` could be specified which was taken from the stack as `long`. Such code fails when passing small integral types on machines where `int` is smaller than `long`. It is advised to use `PL_SHORT`, `PL_INT` or `PL_LONG` as appropriate. Similar, C compilers promote `float` to `double` and therefore `PL_FLOAT` and `PL_DOUBLE` are synonyms.

The type identifiers are:

`PL_VARIABLE` ***none***
No op. Used in arguments of `PL_FUNCTOR`.

`PL_BOOL` ***int***
Unify the argument with `true` or `false`.

`PL_ATOM` ***atom_t***
Unify the argument with an atom, as in PL_unify_atom().

PL_CHARS ***const char ****
Unify the argument with an atom, constructed from the C `char *`, as in PL_unify_atom_chars().

PL_NCHARS ***unsigned int, const char ****
Unify the argument with an atom, constructed from length and `char*` as in PL_unify_atom_nchars().

PL_UTF8_CHARS ***const char ****
Create an atom from a UTF-8 string.

PL_UTF8_STRING ***const char ****
Create a packed string object from a UTF-8 string.

PL_MBCHARS ***const char ****
Create an atom from a multi-byte string in the current locale.

PL_MBCODES ***const char ****
Create a list of character codes from a multi-byte string in the current locale.

PL_MBSTRING ***const char ****
Create a packed string object from a multi-byte string in the current locale.

PL_NWCHARS ***unsigned int, const wchar_t ****
Create an atom from a length and a wide character pointer.

PL_NWCODES ***unsigned int, const wchar_t ****
Create an list of character codes from a length and a wide character pointer.

PL_NWSTRING ***unsigned int, const wchar_t ****
Create a packed string object from a length and a wide character pointer.

PL_SHORT ***short***
Unify the argument with an integer, as in PL_unify_integer(). As `short` is promoted to `int`, `PL_SHORT` is a synonym for `PL_INT`.

PL_INT ***int***
Unify the argument with an integer, as in PL_unify_integer().

PL_LONG ***long***
Unify the argument with an integer, as in PL_unify_integer().

PL_INTEGER ***long***
Unify the argument with an integer, as in PL_unify_integer().

PL_DOUBLE ***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_FLOAT ***double***
Unify the argument with a float, as in PL_unify_float().

PL_POINTER ***void ****
Unify the argument with a pointer, as in PL_unify_pointer().

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_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_FUNCTOR_CHARS ***const char *name, int arity, ...***
: Create a functor from the given name and arity and then behave as PL_FUNCTOR.

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 occurrence of *chr* is doubled. In the future the behaviour will depend on the `character_escape` prolog-flag. See `current_prolog_flag/2`.

9.6.6 BLOBS: Using atoms to store arbitrary binary data

SWI-Prolog atoms as well as strings can represent arbitrary binary data of arbitrary length. This facility is attractive for storing foreign data such as images in an atom. An atom is a unique handle to this data and the atom garbage collector is able to destroy atoms that are no longer referenced by the Prolog engine. This property of atoms makes them attractive as a handle to foreign resources, such as Java atoms, Microsoft's COM objects, etc., providing safe combined garbage collection.

To exploit these features safely and in an organised manner the SWI-Prolog foreign interface allows for creating 'atoms' with additional type information. The type is represented by a structure holding C function pointers that tell Prolog how to handle releasing the atom, writing it, sorting it, etc. Two atoms created with different types can represent the same sequence of bytes. Atoms are first ordered on the rank number of the type and then on the result of the `compare()` function. Rank numbers are assigned when the type is registered.

Defining a BLOB type

The type `PL_blob_t` represents a structure with the layout displayed above. The structure contains additional fields at the ... for internal bookkeeping as well as future extension.

```
typedef struct PL_blob_t
{ unsigned long         magic;          /* PL_BLOB_MAGIC */
  unsigned long         flags;          /* Bitwise or of PL_BLOB_* */
  char *                name;           /* name of the type */
  int                   (*release)(atom_t a);
  int                   (*compare)(atom_t a, atom_t b);
  int                   (*write)(IOSTREAM *s, atom_t a, int flags);
```

```
  ...
} PL_blob_t;
```

For each type exactly one such structure should be allocated. Its first field must be initialised to PL_BLOB_MAGIC. The *flags* is a bitwise or of the following constants:

PL_BLOB_TEXT
If specified the blob is assumed to contain text and is considered a normal Prolog atom.

PL_BLOB_UNIQUE
If specified the system ensures that the blob-handle is a unique reference for a blob with the given type, length and content. If this flag is not specified each lookup creates a new blob.

PL_BLOB_NOCOPY
By default the content of the blob is copied. Using this flag the blob references the external data directly. The user must ensure the provided pointer is valid as long as the atom lives. If PL_BLOB_UNIQUE is also specified uniqueness is determined by comparing the pointer rather than the data pointed at.

The *name* field represents the type name as available to Prolog. See also current_blob/2. The other field are function pointers that must be initialised to proper functions or NULL to get the default behaviour of built-in atoms. Below are the defined member functions:

void **acquire**(*atom_t a*)
Called if a new blob of this type is created through PL_put_blob() or PL_unify_blob(). This callback may be used together with the release hook to deal with reference counted external objects.

int **release**(*atom_t a*)
The blob (atom) *a* is about to be released. This function can retrieve the data of the blob using PL_blob_data(). If it returns FALSE the atom garbage collector will *not* reclaim the atom.

int **compare**(*atom_t a, atom_t b*)
Compare the blobs *a* and *b*, both of which are of the type associated to this blob-type. Return values are, as memcmp(), < 0 if *a* is less then *b*, $= 0$ if both are equal and > 0 otherwise.

int **write**(*IOSTREAM *s, atom_t a, int flags*)
Write the content of the blob *a* to the stream *s* and respecting the *flags*. The *flags* are a bitwise or of zero or more of the PL_WRT_* flags defined in SWI-Prolog.h. This prototype is available if the undocumented SWI-Stream.h is included *before* SWI-Prolog.h.

If this function is not provided, write/1 emits the content of the blob for blobs of type PL_BLOB_TEXT or a string of the format <#*hex data*> for binary blobs.

If a blob type is registered from a loadable object (shared object or DLL) the blob-type must be deregistered before the object may be released.

int **PL_unregister_blob_type**(*PL_blob_t *type*)
Unlink the blob type from the registered type and transform the type of possible living blobs to unregistered, avoiding further reference to the type structure, functions referred by it as well as the data. This function returns TRUE if no blobs of this type existed and FALSE otherwise. PL_unregister_blob_type() is intended for the uninstall() hook of foreign modules, avoiding further references to the module.

Accessing blobs

The blob access functions are similar to the atom accessing functions. Blobs being atoms, the atom functions operate on blobs and visa versa. For clarity and possible future compatibility issues however it is not advised to rely on this.

`int` **PL_is_blob**(*term_t t, PL_blob_t **type*)
Succeeds if *t* refers to a blob, in which case *type* is filled with the type of the blob.

`int` **PL_unify_blob**(*term_t t, void *blob, unsigned int len, PL_blob_t *type*)
Unify *t* to a new blob constructed from the given data and associated to the given type. See also PL_unify_atom_nchars().

`int` **PL_put_blob**(*term_t t, void *blob, unsigned int len, PL_blob_t *type*)
Store the described blob in *t*. The return value indicates whether a new blob was allocated (`FALSE`) or the blob is a reference to an existing blob (`TRUE`). Reporting new/existing can be used to deal with external objects having their own reference counts. If the return is `TRUE` this reference count must be incremented and it must be decremented on blob destruction callback. See also PL_put_atom_nchars().

`int` **PL_get_blob**(*term_t t, void **blob, unsigned int *len, PL_blob_t **type*)
If *t* holds a blob or atom get the data and type and return `TRUE`. Otherwise return `FALSE`. Each result pointer may be `NULL`, in which case the requested information is ignored.

`void *` **PL_blob_data**(*atom_t a, unsigned int *len, PL_blob_t **type*)
Get the data and type associated to a blob. This function is mainly used from the callback functions described in section 9.6.6.

9.6.7 Exchanging GMP numbers

If SWI-Prolog is linked with the GNU Multiple Precision Arithmetic Library (GMP, used by default), the foreign interface provides functions for exchanging numeric values to GMP types. To access these functions the header `<gmp.h>` must be included *before* `<SWI-Prolog.h>`. Here is an example exploiting the function mpz_nextprime():

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

static foreign_t
next_prime(term_t n, term_t prime)
{ mpz_t mpz;
  int rc;

  mpz_init(mpz);
  if ( PL_get_mpz(n, mpz) )
  { mpz_nextprime(mpz, mpz);

    rc = PL_unify_mpz(prime, mpz);
  } else
```

```
    rc = FALSE;

  mpz_clear(mpz);
  return rc;
}

install_t
install()
{ PL_register_foreign("next_prime", 2, next_prime, 0);
}
```

`int` **PL_get_mpz**(*term_t t, mpz_t mpz*)
: If *t* represents an integer *mpz* is filled with the value and the function returns `TRUE`. Otherwise *mpz* is untouched and the function returns `FALSE`. Note that *mpz* must have been initialised before calling this function and must be cleared using mpz_clear() to reclaim any storage associated with it.

`int` **PL_get_mpq**(*term_t t, mpq_t mpq*)
: If *t* is an integer or rational number (term `rdiv/2`) *mpq* is filled with the *normalise* rational number and the function returns `TRUE`. Otherwise *mpq* is untouched and the function returns `FALSE`. Note that *mpq* must have been initialised before calling this function and must be cleared using mpq_clear() to reclaim any storage associated with it.

`int` **PL_unify_mpz**(*term_t t, mpz_t mpz*)
: Unify *t* with the integer value represented by *mpz* and return *TRUE* on success. The *mpz* argument is not changed.

`int` **PL_unify_mpq**(*term_t t, mpq_t mpq*)
: Unify *t* with a rational number represented by *mpq* and return *TRUE* on success. Note that *t* is unified with an integer if the denominator is 1. The *mpq* argument is not changed.

9.6.8 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. If the module argument *m* is NULL, the current context module is used.

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 is imported into the module given to PL_predicate(), this function will return the module where the predicate is 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().

[3] Do not pass the integer 0 for normal operation, as this is interpreted as PL_Q_NODEBUG for backward compatibility reasons.

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().

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 9.4 shows an example to obtain the number of defined atoms. All checks are omitted to improve readability.

9.6.9 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 9.4 for an example.

```
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 9.4: Calling Prolog

`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.

9.6.10 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:

```
{ 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*.

9.6.11 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 `arithmetic_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_CHARS, "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].

9.6.12 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 4.10. 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 wishes to handle this signal too, it should use PL_signal() to install its handler after initialising 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.

By default, signals are handled asynchronously (i.e. at the time they arrive). It is inherently dangerous to call extensive code fragments, and especially exception related code from asynchronous handlers. The interface allows for *synchronous* handling of signals. In this

[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

case the native OS handler just schedules the signal using PL_raise(), which is checked by PL_handle_signals() at the call- and redo-port. This behaviour is realised by or-ing *sig* with the constant `PL_SIGSYNC`.[5]

Signal handling routines may raise exceptions using PL_raise_exception(). The use of PL_throw() is not safe. If a synchronous handler raises an exception, the exception is delayed to the next call to PL_handle_signals();

`int` **PL_raise**(*int sig*)
Register *sig* for *synchronous* handling by Prolog. Synchronous signals are handled at the call-port or if foreign code calls PL_handle_signals(). See also `thread_signal/2`.

`int` **PL_handle_signals**(*void*)
Handle any signals pending from PL_raise(). PL_handle_signals() is called at each pass through the call- and redo-port at a safe point. Exceptions raised by the handler using PL_raise_exception() are properly passed to the environment.

The user may call this function inside long-running foreign functions to handle scheduled interrupts. This routine returns the number of signals handled. If a handler raises an exception, the return value is -1 and the calling routine should return with `FALSE` as soon as possible.

9.6.13 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 retrieve 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.

[5] A better default would be to use synchronous handling, but this interface preserves backward compatibility.

`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_external() 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.

Getting file names

The function PL_get_file_name() provides access to Prolog filenames and its file-search mechanism described with `absolute_file_name/3`. Its existence is motivated to realise a uniform interface to deal with file-properties, search, naming conventions etc. from foreign code.

`int` **PL_get_file_name**(*term_t spec, char **name, int flags*)
Translate a Prolog term into a file name. The name is stored in the static buffer ring described with PL_get_chars() option `BUF_RING`. Conversion from the internal UNICODE encoding is done using standard C library functions. *flags* is a bit-mask controlling the conversion process. Options are:

`PL_FILE_ABSOLUTE`
Return an absolute path to the requested file.

`PL_FILE_OSPATH`
Return a the name using the hosting OS conventions. On MS-Windows, `\` is used to separate directories rather than the canonical `/`.

`PL_FILE_SEARCH`
Invoke `absolute_file_name/3`. This implies rules from `file_search_path/2` are used.

`PL_FILE_EXIST`
Demand the path to refer to an existing entity.

`PL_FILE_READ`
Demand read-access on the result.

`PL_FILE_WRITE`
Demand write-access on the result.

`PL_FILE_EXECUTE`
Demand execute-access on the result.

`PL_FILE_NOERRORS`
Do not raise any exceptions.

9.6.14 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 4.9.

`int` **PL_warning**(*format, a1, ...*)
Print an error message starting with '`[WARNING: `', followed by the output from *format*, followed by a '`]`' and a newline. Then start the tracer. *format* and the arguments are the same as for `printf(2)`. Always returns `FALSE`.

9.6.15 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 9.1.

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.
PL_ACTION_ATTACH_CONSOLE	Attach a console to a thread if it does not have one. See attach_console/0.

Table 9.1: PL_action() options

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.
PL_QUERY_MAX_THREADS	Return the maximum number of threads that can be created in this version. Return values of PL_thread_self() are between 0 and this number.
PL_QUERY_ENCODING	Return the default stream encoding of Prolog (of type IOENC).
PL_QUERY_USER_CPU	Get amount of user CPU time of the process in milliseconds.

Table 9.2: PL_query() options

9.6.16 Querying Prolog

long **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.[6] The options are given in table 9.2.

9.6.17 Registering Foreign Predicates

int **PL_register_foreign_in_module**(*const char *module, 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 in module *module*. If *module* is NULL, the predicate is created in the module of the calling context or if no context is present in the module user.

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.

[6] Returning pointers and integers as a long is bad style. The signature of this function should be changed.

Predicates may be registered either before or after PL_initialise(). When registered before initialisation the registration is recorded and executed after installing the system predicates and before loading the saved state.

Default calling (i.e. without PL_FA_VARARGS) *function* is passed the same number of term_t arguments as the arity of the predicate and, if the predicate is non-deterministic, an extra argument of type control_t (see section 9.6.1). If PL_FA_VARARGS is provided, *function* is called with three arguments. The first argument is a term_t handle to the first argument. Further arguments can be reached by adding the offset (see also PL_new_term_refs()). The second argument is the arity, which defines the number of valid term-references in the argument vector. The last argument is used for non-deterministic calls. It is currently undocumented and should be defined of type void*. Here is an example:

```
static foreign_t
atom_checksum(term_t a0, int arity, void* context)
{ char *s;

  if ( PL_get_atom_chars(a0, &s) )
  { int sum;

    for(sum=0; *s; s++)
      sum += *s&0xff;

    return PL_unify_integer(a0+1, sum&0xff);
  }

  return FALSE;
}

install_t
install()
{ PL_register_foreign("atom_checksum", 2, atom_checksum, PL_FA_VARARGS);
}
```

int **PL_register_foreign**(*const char *name, int arity, foreign_t (*function)(), int flags*)
Same as PL_register_foreign_in_module(), passing NULL for the *module*.

void **PL_register_extensions_in_module**(*const char *module, PL_extension *e*)
Register a series of predicates from an array of definitions of the type PL_extension in the given *module*. If *module* is NULL, the predicate is created in the module of the calling context or if no context is present in the module user. The PL_extension type 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;
```

For details, see PL_register_foreign_in_module(). 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_in_module("user", predicates);

  if ( !PL_initialise(argc, argv) )
    PL_halt(1);

  ...
}
```

`void` **PL_register_extensions**(*PL_extension *e*)
Same as PL_register_extensions_in_module() using `NULL` for the *module* argument.

9.6.18 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 top-level 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);
}
```

9.6.19 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 represented 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 chosen 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 efficient encoding required? For examine, a boolean array 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 accessibility 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 9.6.1.

Examples for storing foreign data

In this section, we will 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. Finally, we discuss the outline of the DDE interface.

Integer sets with not-too-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 bit-vectors. 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 naturally expressed using string objects. If the string is wrapped in bitvector/1, lower-bound of the vector is 0, and the upper-bound is not defined, an implementation for getting and putting the sets 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);
}
```

The DDE interface (see section 4.42) 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 approach 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.

9.6.20 Embedding SWI-Prolog in other applications

With embedded Prolog we refer to the situation where the 'main' program is not the Prolog application. Prolog is sometimes embedded in C, C++, Java or other languages to provide logic based services in a larger application. Embedding loads the Prolog engine as a library to the external language. Prolog itself only provides for embedding in the C-language (compatible to C++). Embedding in Java is achieved using JPL using a C-glue between the Java and Prolog C-interfaces.

The most simple embedded program is below. The interface function PL_initialise() **must** be called before any of the other SWI-Prolog foreign language functions described in this chapter, except for PL_initialise_hook(), PL_new_atom(), PL_new_functor() and PL_register_foreign(). PL_initialise() interprets all the command-line arguments, except for the `-t` *`toplevel`* flag that is interpreted by PL_toplevel().

```
int
main(int argc, char **argv)
{
#ifdef READLINE /* Remove if you don't want readline */
  PL_initialise_hook(install_readline);
#endif

  if ( !PL_initialise(argc, argv) )
    PL_halt(1);

  PL_halt(PL_toplevel() ? 0 : 1);
}
```

`int` **PL_initialise**(*int argc, char **argv*)

Initialises the SWI-Prolog heap and stacks, restores the Prolog state, 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 command-line 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 9.7.
- Find the Prolog home directory. This process is described in detail in section 9.8.

PL_initialise() returns 1 if all initialisation succeeded and 0 otherwise.[7]

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:

[7] BUG: Various fatal errors may cause PL_initialise to call PL_halt(1), preventing it from returning at all.

```
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]`. If the Prolog state is attached to a DLL (see the `-dll` option of `plld`, pass the name of this DLL.

`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 top-level should normally delete this line, shrinking the Prolog kernel significantly. Note that the Windows version does not use GNU readline.

`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,).

`int` **PL_halt**(*int status*)
Cleanup the Prolog environment using PL_cleanup() and calls exit() with the status argument.

As PL_cleanup() can only be called from the main thread, this function returns `FALSE` when called from another thread as the main one.[8]

Threading, Signals and embedded Prolog

This section applies to Unix-based environments that have signals or multi-threading. The Windows version is compiled for multi-threading and Windows lacks proper signals.

We can distinguish two classes of embedded executables. There are small C/C++-programs that act as an interfacing layer around Prolog. Most of these programs can be replaced using the normal Prolog executable extended with a dynamically loaded foreign extension and in most cases this is the preferred route. In other cases, Prolog is embedded in a complex application that—like Prolog—wants to control the process environment. A good example is Java. Embedding Prolog is generally the only way to get these environments together in one process image. Java applications however are by nature multi-threaded and appear to do signal-handling (software interrupts).

To make Prolog operate smoothly in such environments it must be told not to alter the process environment. This is partly done at build-time and partly execution time. At build-time we must specify the use of software stack-overflow rather then the default hardware checks. This is done using

```
sh configure --disable-segv-handling
```

The resulting Prolog executable is about 10% slower than the normal executable, but behaves much more reliable in complicated embedded situations. In addition, as the process no longer handles segmentation violations, debugging foreign code linked to it is much easier.

At runtime, it is advised to pass the flag `-nosignals`, which inhibits all default signal handling. This has a few consequences though:

- It is no longer possible to break into the tracer using an interrupt signal (Control-C).
- `SIGPIPE` is normally set to be ignored. Prolog uses return-codes to diagnose broken pipes. Depending on the situation one should take appropriate action if Prolog streams are connected to pipes.
- Fatal errors normally cause Prolog to call PL_cleanup() and exit(). It is advised to call PL_cleanup() as part of the exit-procedure of your application.

9.7 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]
```

[8] BUG: Eventually it may become possible to call PL_halt() from any thread.

`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 command-line. 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 command-line. 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 separately and appended to the kernel using `cat` ⟨*kernel*⟩ ⟨*state*⟩ `>` ⟨*out*⟩ (Win32: `copy /b` ⟨*kernel*⟩`+`⟨*state*⟩ ⟨*out*⟩)

-shared
Link C, C++ or object files into a shared object (DLL) that can be loaded by the `load_foreign_library/1` predicate. If used with `-c` it sets the proper options to compile a C or C++ file ready for linking into a shared object

-dll
Windows only. Embed SWI-Prolog into a DLL rather than an executable.

-c
Compile C or C++ source-files into object files. This turns `plld` into a replacement for the C or C++ compiler where proper options such as the location of the include directory are passed automatically to the compiler.

-E
Invoke the C preprocessor. Used to make `plld` a replacement for the C or C++ compiler.

-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.

9.7.1 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.
```

```
#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 9.5: C-source for the calc application

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 9.5.
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
```

9.8 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 9.7).

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.

9.9 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 9.6 shows the C-source file, figure 9.7 illustrates compiling and loading of foreign code.

```
/*  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 9.6: 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 9.7: Compiling the C-source and loading the object file

9.10 Notes on Using Foreign Code

9.10.1 Memory Allocation

SWI-Prolog's heap memory allocation is based on the `malloc(3)` library routines. The stacks are allocated using mmap() on most Unix machines and using VirtualAlloc() on windows. SWI-Prolog provides the functions below as a wrapper around malloc(). Allocation errors in these functions trap SWI-Prolog's fatal-error handler, in which case PL_malloc() or PL_realloc() do not return.

Portable applications must use PL_free() to release strings returned by PL_get_chars() using the `BUF_MALLOC` argument. Portable applications may use both PL_malloc() and friends or malloc() and friends but should not mix these two sets of functions on the same memory.[9]

`void *` **PL_malloc**(*size_t bytes*)
: Allocate *bytes* of memory. On failure SWI-Prolog's fatal error handler is called and PL_malloc() does not return. Memory allocated using these functions must use PL_realloc() and PL_free() rather than realloc() and free().

`void *` **PL_realloc**(*void *mem, size_t size*)
: Change the size of the allocated chunk, possibly moving it. The *mem* argument must be obtained from a previous PL_malloc() or PL_realloc() call.

`void` **PL_free**(*void *mem*)
: Release memory. The *mem* argument must be obtained from a previous PL_malloc() or PL_realloc() call.

9.10.2 Compatibility between Prolog versions

Great care is taken to ensure binary compatibility of foreign extensions between different Prolog versions. Only much less frequently used stream interface has been responsible for binary incompatibilities.

Source-code that relies on new features of the foreign interface can use the macro `PLVERSION` to find the version of `SWI-Prolog.h` and PL_query() using the option `PL_QUERY_VERSION` to find the version of the attached Prolog system. Both follow the same numbering schema explained with PL_query().

9.10.3 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.
```

[9] These functions were introduced in SWI-Prolog 5.0.9 to realise guaranteed portability. Foreign code that must be compatible with older versions can check the `PLVERSION` macro.

```
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).
```

9.10.4 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.

9.10.5 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.

Generating Runtime Applications

10

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` command-line 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 $\langle Key \rangle = \langle Value \rangle$ or $\langle Key \rangle(\langle Value \rangle)$ pairs. The available keys are described in table 10.1.

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(true)` 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 top-level 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 10.1: ⟨*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 library(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.

10.1 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 10.2 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.

10.2 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.c` 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 }     /* terminating 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`.

10.3 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 source code, 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 incorporated 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 file `help.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.

10.3.1 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 successful 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.

10.3.2 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.

10.4 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 top-level 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 locate config.pl~n'),
            halt(1)
            ).
```

10.4.1 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
```

10.5 The Runtime Environment

10.5.1 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/`	directory for shared libraries
`lib/`⟨*arch*⟩`/`	machine-specific shared libraries

The SWI-Prolog library

A

This chapter documents the SWI-Prolog library. As SWI-Prolog provides auto-loading, there is little difference between library predicates and built-in predicates. Part of the library is therefore documented in the rest of the manual. Library predicates differ from built-in predicates in the following ways.

- User-definition of a built-in leads to a permission-error, while using the name of a library predicate is allowed.
- If autoloading is disabled explicitely or because trapping unknown predicates is disabled (see `unknown/2` and `current_prolog_flag/2`), library predicates must be loaded explicitely.
- Using libraries reduce the footprint of applications that don't need them.

The documentation of the library is just started. Material from the standard packages should be moved here, some material from other parts of the manual should be moved too and various libraries are not documented at all.

A.1 `lists`: List Manipulation

This library provides commonly accepted basic predicates for list manipulation in the Prolog community. Some additional list manipulations are built-in. Their description is in section 4.28.

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.

nextto(*?X, ?Y, ?List*)
Succeeds when *Y* immediatly follows *X* in *List*.

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.[1]

[1] 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.

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(*?List, ?Elem*)
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.[2]

reverse(+*List1, -List2*)
Reverse the order of the elements in *List1* and unify the result with the elements of *List2*.

permutation(*?List1, ?List2*)
Permuation is true when *List1* is a permutation of *List2*. The implementation can solve for *List2* given *List1* or *List1* given *List2*, or even enumerate *List1* and *List2* together.

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]
```

sumlist(+*List, -Sum*)
Unify *Sum* to the result of adding all elements in *List*. *List* must be a proper list holding numbers. See `number/1` and `is/2`. for details on arithmetic.

numlist(+*Low, +High, -List*)
If *Low* and *High* are integers with $Low \leq High$, unify *List* to a list $[Low, Low + 1, \ldots High]$. See also `between/3`.

A.1.1 Set Manipulation

The set predicates listed in this section work on ordinary unsorted lists. Note that this makes many of the operations order N^2. For larger sets consider the use of ordered sets as implemented by library `ordsets.pl`, running most these operations in order N. See section A.2.

is_set(+*Set*)
Succeeds if *Set* is a list (see `is_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]
```

[2] The argument order of this predicate was changed in 5.1.12 for compatibility reasons.

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.

A.2 `ordsets`: Ordered Set Manipulation

Ordered sets are lists with unique elements sorted to the standard order of terms (see `sort/2`). Exploiting ordering, many of the set operations can be expressed in order N rather than N^2 when dealing with unordered sets that may contain duplicates. The `ordsets` is available in a number of Prolog implementations. Our predicates are designed to be compatible with common practice in the Prolog community. The implementation is incomplete and relies partly on `oset`, an older ordered set library distributed with SWI-Prolog. New applications are advices to use `ordsets`.

Some of these predicates match directly to corresponding list operations. It is adviced to use the versions from this library to make clear you are operating on ordered sets.

ord_empty(*?Set*)
True if *Set* is an empty ordered set. *Set* unifies with the empty list.

list_to_ord_set(+*List, -OrdSet*)
Convert a *List* to an ordered set. Same as `sort/2`.

ord_add_element(+*Set, +Element, -NewSet*)
Add an element to an ordered set. *NewSet* is the same as *Set* if *Element* is already part of *Set*.

ord_del_element(+*Set, +Element, -NewSet*)
Delete *Element* from *Set*. Succeeds without changing *Set* if *Set* does not contain *Element*.

ord_intersect(+*Set1, +Set2*)
True if the intersection of *Set1* and *Set2* is non-empty.

ord_intersection(+*Set1, +Set2, -Intersection*)
True if *Intersection* is the intersection of *Set1* and *Set2*.

ord_disjoint(+*Set1, +Set2*)
True if *Set1* and *Set2* have no common element. Negation of `ord_intersect/2`.

ord_subtract(+*Set, +Delete, -Remaining*)
True if *Remaining* contains the elements of *Set* that are not in set *Delete*.

ord_union(+*Set1, +Set2, -Union*)
True if *Union* contains all elements from *Set1* and *Set2*

ord_union(+*Set1, +Set2, -Union, -New*)
Defined as if `ord_union`(*Set1, Set2, Union*), `ord_subtract`(*Set2, Set1, New*).

ord_subset(+*Sub, +Super*)
True if all elements of *Sub* are in *Super*.

ord_memberchk(+*Element, +Set*)
True if *Element* appears in *Set*. Does not backtrack. Same as `memberchk/2`.

A.3 assoc: Association lists

Authors: *Richard A. O'Keefe, L.Damas, V.S.Costa and Markus Triska*

Elements of an association list have 2 components: A (unique) *key* and a *value*. Keys should be ground, values need not be. An association list can be used to fetch elements via their keys and to enumerate its elements in ascending order of their keys. The `assoc` module uses AVL trees to implement association lists. This makes inserting, changing and fetching a single element an O(log(N)) (where N denotes the number of elements in the list) expected time (and worst-case time) operation.

assoc_to_list(+*Assoc, -List*)
List is a list of Key-Value pairs corresponding to the associations in *Assoc* in ascending order of keys.

empty_assoc(-*Assoc*)
Assoc is unified with an empty association list.

gen_assoc(*?Key, +Assoc, ?Value*)
Enumerate matching elements of *Assoc* in ascending order of their keys via backtracking.

get_assoc(+*Key, +Assoc, ?Value*)
Value is the value associated with *Key* in the association list *Assoc*.

get_assoc(+*Key, +Assoc, ?Old, ?NewAssoc, ?New*)
NewAssoc is an association list identical to *Assoc* except that the value associated with *Key* is *New* instead of *Old*.

list_to_assoc(+*List, ?Assoc*)
Assoc is an association list corresponding to the Key-Value pairs in *List*.

map_assoc(*:Goal, +Assoc*)
Goal(V) is true for every value V in *Assoc*.

map_assoc(*:Goal, +AssocIn, ?AssocOut*)
AssocOut is *AssocIn* with *Goal* applied to all corresponding pairs of values.

max_assoc(+*Assoc, ?Key, ?Value*)
Key and *Value* are key and value of the element with the largest key in *Assoc*.

min_assoc(+*Assoc, ?Key, ?Value*)
Key and *Value* are key and value of the element with the smallest key in *Assoc*.

ord_list_to_assoc(+*List, ?Assoc*)
Assoc is an association list correpsond to the Key-Value pairs in *List*, which must occur in ascending order of their keys.

put_assoc(+*Key, +Assoc, +Value, ?NewAssoc*)
NewAssoc is an association list identical to *Assoc* except that *Key* is associated with *Value*. This can be used to insert and change associations.

A.4 `ugraphs`: Unweighted Graphs

Authors: *Richard O'Keefe & Vitor Santos Costa*

Implementation and documentation are copied from YAP 5.0.1. The `ugraph` *library is based on code originally written by Richard O'Keefe. The code was then extended to be compatible with the SICStus Prolog ugraphs library. Code and documentation have been cleaned and style has been changed to be more in line with the rest of SWI-Prolog.*

The ugraphs library was originally released in the public domain. YAP is convered by the Perl artistic license, which does not imply further restrictions on the SWI-Prolog LGPL license.

The routines assume directed graphs, undirected graphs may be implemented by using two edges.

Originally graphs where represented in two formats. The SICStus library and this version of `ugraphs.pl` only uses the *S-representation*. The S-representation of a graph is a list of (vertex-neighbors) pairs, where the pairs are in standard order (as produced by keysort) and the neighbors of each vertex are also in standard order (as produced by sort). This form is convenient for many calculations. Each vertex appears in the S-representation, also if it has no neighbors.

vertices_edges_to_ugraph(+*Vertices, +Edges, -Graph*)
Given a graph with a set of *Vertices* and a set of *Edges*, *Graph* must unify with the corresponding S-representation. Note that the vertices without edges will appear in *Vertices* but not in *Edges*. Moreover, it is sufficient for a vertice to appear in *Edges*.

```
?- vertices_edges_to_ugraph([],[1-3,2-4,4-5,1-5], L).
L = [1-[3,5], 2-[4], 3-[], 4-[5], 5-[]]
```

In this case all vertices are defined implicitly. The next example shows three unconnected vertices:

```
?- vertices_edges_to_ugraph([6,7,8],[1-3,2-4,4-5,1-5], L).
L = [1-[3,5], 2-[4], 3-[], 4-[5], 5-[], 6-[], 7-[], 8-[]] ?
```

vertices(+*Graph, -Vertices*)
Unify *Vertices* with all vertices appearing in graph *Graph*. Example:

```
?- vertices([1-[3,5],2-[4],3-[],4-[5],5-[]], L).
L = [1, 2, 3, 4, 5]
```

edges(+*Graph, -Edges*)
Unify *Edges* with all edges appearing in *Graph*. In the next example:

```
?- edges([1-[3,5],2-[4],3-[],4-[5],5-[]], L).
L = [1-3, 1-5, 2-4, 4-5]
```

add_vertices(+*Graph, +Vertices, -NewGraph*)
Unify *NewGraph* with a new graph obtained by adding the list of *Vertices* to the *Graph*. Example:

```
?- add_vertices([1-[3,5],2-[]], [0,1,2,9], NG).
NG = [0-[], 1-[3,5], 2-[], 9-[]]
```

del_vertices(+*Vertices, +Graph, -NewGraph*)
Unify *NewGraph* with a new graph obtained by deleting the list of *Vertices* and all the edges that start from or go to a vertex in *Vertices* to the *Graph*. Example:

```
?- del_vertices([2,1],
                [1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[2,6],8-[]],
                NL).
NL = [3-[],4-[5],5-[],6-[],7-[6],8-[]]
```

add_edges(+*Graph, +Edges, -NewGraph*)
Unify *NewGraph* with a new graph obtained by adding the list of edges *Edges* to the graph *Graph*. Example:

```
?- add_edges([1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[],8-[]],
             [1-6,2-3,3-2,5-7,3-2,4-5],
             NL).
NL = [1-[3,5,6], 2-[3,4], 3-[2], 4-[5], 5-[7], 6-[], 7-[], 8-[]]
```

del_edges(+*Graph, +Edges, -NewGraph*)
Unify *NewGraph* with a new graph obtained by removing the list of *Edges* from the *Graph*. Notice that no vertices are deleted. In the next example:

```
?- del_edges([1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[],8-[]],
             [1-6,2-3,3-2,5-7,3-2,4-5,1-3],
             NL).
NL = [1-[5],2-[4],3-[],4-[],5-[],6-[],7-[],8-[]]
```

transpose(+*Graph, -NewGraph*)
Unify *NewGraph* with a new graph obtained from *Graph* by replacing all edges of the form V1-V2 by edges of the form V2-V1. The cost is $O(|V|^2)$. Notice that an undirected graph is its own transpose. Example:

```
?- transpose([1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[],8-[]], NL).
NL = [1-[],2-[],3-[1],4-[2],5-[1,4],6-[],7-[],8-[]]
```

neighbours(+*Vertex, +Graph, -Vertices*)
Unify *Vertices* with the list of neighbours of vertex *Vertex* in *Graph*. Example:

```
?- neighbours(4,[1-[3,5],2-[4],3-[],
                 4-[1,2,7,5],5-[],6-[],7-[],8-[]], NL).
NL = [1,2,7,5]
```

neighbors(+*Vertex, +Graph, -Vertices*)
American version of `neighbours/3`.

complement(+*Graph, -NewGraph*)
Unify *NewGraph* with the graph complementary to *Graph*. Example:

```
?- complement([1-[3,5],2-[4],3-[],
               4-[1,2,7,5],5-[],6-[],7-[],8-[]], NL).
NL = [1-[2,4,6,7,8],2-[1,3,5,6,7,8],3-[1,2,4,5,6,7,8],
      4-[3,5,6,8],5-[1,2,3,4,6,7,8],6-[1,2,3,4,5,7,8],
      7-[1,2,3,4,5,6,8],8-[1,2,3,4,5,6,7]]
```

compose(+*LeftGraph, +RightGraph, -NewGraph*)
Compose, by connecting the *drains* of *LeftGraph* to the *sources* of *RightGraph*. Example:

```
?- compose([1-[2],2-[3]],[2-[4],3-[1,2,4]],L).
L = [1-[4], 2-[1,2,4], 3-[]]
```

ugraph_union(+*Graph1, +Graph2, -NewGraph*)
NewGraph is the union of *Graph1* and *Graph2*. Example:

```
?- ugraph_union([1-[2],2-[3]],[2-[4],3-[1,2,4]],L).
L = [1-[2], 2-[3,4], 3-[1,2,4]]
```

top_sort(+*Graph, -Sort*)
Generate the set of nodes *Sort* as a topological sorting of graph *Graph*, if one is possible. A toplogical sort is possible if the graph is connected and acyclic. In the example we show how topological sorting works for a linear graph:

```
?- top_sort([1-[2], 2-[3], 3-[]], L).
L = [1, 2, 3]
```

top_sort(+*Graph, -Sort0, -Sort*)
Generate the difference list Sort-Sort0 as a topological sorting of graph *Graph*, if one is possible.

transitive_closure(+*Graph, -Closure*)
Generate the graph Closure as the transitive closure of graph *Graph*. Example:

```
 ?- transitive_closure([1-[2,3],2-[4,5],4-[6]],L).
L = [1-[2,3,4,5,6], 2-[4,5,6], 4-[6]]
```

reachable(+*Vertex, +Graph, -Vertices*)
Unify *Vertices* with the set of all vertices in graph *Graph* that are reachable from *Vertex*. Example:

```
?- reachable(1,[1-[3,5],2-[4],3-[],4-[5],5-[]],V).
V = [1, 3, 5]
```

A.5 nbset: Non-backtrackable set

The library nb_set defines *non-backtrackable sets*, implemented as binary trees. The sets are represented as compound terms and manipulated using nb_setarg/3. Non-backtrackable manipulation of datastructures is not supported by a large number of Prolog implementation, but it it has several advantages over using the database. It produces less garbage, is thread-safe, reentrant and deals with exceptions without leaking data.

Similar to the assoc library keys can be any Prolog term, but it is not allowed to instantiate or modify a term.

One of the ways to use this library is to generate unique values on backtracking *without* generating *all* solutions first, for example to act as a filter between a generator producing many duplicates and an expensive test routine, as outlines below.

```
generate_and_test(Solution) :-
        empty_nb_set(Set),
        generate(Solution),
        add_nb_set(Solution, Set, true),
        test(Solution).
```

empty_nb_set(*?Set*)
True if *Set* is a non-backtrackable emoty set.

add_nb_set(+*Key, !Set*)
Add *Key* to *Set*. If *Key* is already a member of *Set*, add_nb_set/3 succeeds without modifying *Set*.

add_nb_set(+*Key, !Set, ?New*)
If *Key* is not in *Set* and *New* is unified to true *Key* is added to *Set*. If *Key* is in *Set New* is unified to false. It can be used for many purposes:

add_nb_set(+, +, false)	Test membership
add_nb_set(+, +, true)	Succeed only if new member
add_nb_set(+, +, Var)	Succeed, bindin *Var*

gen_nb_set(+*Set, -Key*)
Generate all members of *Set* on backtracking in the standard order of terms. To test membership, use `add_nb_set/3`.

size_nb_set(+*Set, -Size*)
Unify *Size* with the number of elements in *Set*.

nb_set_to_list(+*Set, -List*)
Unify *List* with a list of all elements in set in the standard order of terms (i.e. and *ordered list*).

A.6 `gensym`: Generate unique identifiers

Gensym (**Gen**erate **Sym**bols) is an old library for generating unique symbols (atoms). Such symbols are generated from a base atom which gets a sequence number appended. Of course there is no guarantee that 'catch22' is not an already defined atom and therefore one must be aware these atoms are only unique in an isolated context.

The SWI-Prolog gensym library is thread-safe. The sequence numbers are global over all threads and therefore generated atoms are unique over all threads.

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.

reset_gensym(+*Base*)
Restart generation of identifiers from *Base* at ⟨*Base*⟩1. Used to make sure a program produces the same results on subsequent runs. Use with care.

reset_gensym
Reset gensym for all registered keys. This predicate is available for compatibility only. New code is strongly advice to avoid the use of reset_gensym or at least to reset only the keys used by your program to avoid unexpected site-effects on other components.

A.7 `check`: Elementary completeness checks

This library defines the predicate `check/0` and a few friends that allow for a quick-and-dirty cross-referencing.

check
Performs the three checking passes implemented by `list_undefined/0`, `list_autoload/0` and `list_redefined/0`. Please check the definition of these predicates for details.

The typical usage of this predicate is right after loading your program to get a quick overview on the completeness and possible conflicts in your program.

list_undefined
Scans the database for predicates that have no definition. A predicate is considered defined if it has clauses, is declared using `dynamic/1` or `multifile/1`. As a program is compiled

calls are translated to predicates. If the called predicate is not yet defined it is created as a predicate without definition. The same happens with runtime generated calls. This predicate lists all such undefined predicates that are referenced and not defined in the library. See also `list_autoload/0`. Below is an example from a real program and an illustration how to edit the referencing predicate using `edit/1`.

```
?- list_undefined.
Warning: The predicates below are not defined. If these are defined
Warning: at runtime using assert/1, use :- dynamic Name/Arity.
Warning:
Warning: rdf_edit:rdfe_retract/4, which is referenced by
Warning:         1-st clause of rdf_edit:undo/4
Warning: rdf_edit:rdfe_retract/3, which is referenced by
Warning:         1-st clause of rdf_edit:delete_object/1
Warning:         1-st clause of rdf_edit:delete_subject/1
Warning:         1-st clause of rdf_edit:delete_predicate/1

?- edit(rdf_edit:undo/4).
```

list_autoload
Lists all undefined (see `list_undefined/0`) predicates that have a definition in the library along with the file from which they will be autoloaded when accessed. See also `autoload/0`.

list_redefined
Lists predicates that are defined in the global module `user` as well as in a normal module. I.e. predicates for which the local definition overrules the global default definition.

A.8 debug: Some reusable code to help debugging applications

This library provides an structured alternative for putting print-statements into your source-code to trace what is going on. Debug messages are organised in *topics* that can be activated and de-activated without changing the source. In addition, if the application is compiled with the `-O` flag these predicates are removed using `goal_expansion/2`.

Although this library can be used through the normal demand-loading mechanism it is adviced to load it explicitely before code using it to profit from goal-expansion, which removes these calls if compiled with optimisation on and records the topics from `debug/3` and `debugging/1` for `list_debug_topics/0`.

debug(+*Topic, +Format, +Args*)
If *Topic* is a selected debugging topic (see `debug/1`) a message is printed using `print_message/2` with level `informational`. *Format* and *Args* are interpreted by `format/2`. Here is a typical example:

```
...,
debug(init, 'Initialised ~w', [Module]),
...,
```

Topic can be any Prolog term. Compound terms can be used to make categories of topics that can be activated using `debug/1`.

debugging(+*Topic*)
Succeeds if *Topic* is a selected debugging topic. It is intended to execute arbitrary code depending on the users debug topic selection. The construct `(debugging(Topic) -> Code ; true)` is removed if the code is compiled in optimise mode.

debug(+*Topic*)
Select all registered topics that unify with *Topic* for debugging. This call is normally used from the toplevel to activate a topic for debugging. Topics are de-activated using `nodebug/1`.

nodebug(+*Topic*)
Deactivates topics for debugging. See `debug/1` for the arguments.

list_debug_topics
List the current status of registered topics. See also `debugging/0`.

assertion(*:Goal*)
This predicate is to be compared to the C-library assert() function. By inserting this goal you explicitely state you expect *Goal* to succeed at this place. As `assertion/1` calls are removed when compiling in optimized mode *Goal* should not have side-effects. Typical examples are type-tests and validating invariants defined by your application.

If *Goal* fails the system prints a message, followed by a stack-trace and starts the debugger.

In older versions of this library this predicate was called `assume/1`. Code using `assume/1` is automatically converted while printing a warning on the first occurrence.

A.9 `readutil`: Reading lines, streams and files

This library contains primitives to read lines, files, multiple terms, etc. The package `clib` provides a shared object (DLL) named `readutil`. If the library can locate this shared object it will use the foreign implementation for reading character codes. Otherwise it will use a Prolog implementation. Distributed applications should make sure to deliver the `readutil` shared object if performance of these predicates is critical.

read_line_to_codes(+*Stream, -Codes*)
Read the next line of input from *Stream* and unify the result with *Codes after* the line has been read. A line is ended by a newline character or end-of-file. Unlike `read_line_to_codes/3`, this predicate removes trailing newline character.

On end-of-file the atom `end_of_file` is returned. See also `at_end_of_stream/[0,1]`.

read_line_to_codes(+*Stream, -Codes, ?Tail*)
Diference-list version to read an input line to a list of character codes. Reading stops at the newline or end-of-file character, but unlike `read_line_to_codes/2`, the newline is retained in the output. This predicate is especially useful for readine a block of lines upto some delimiter. The following example reads an HTTP header ended by a blank line:

```
read_header_data(Stream, Header) :-
        read_line_to_codes(Stream, Header, Tail),
        read_header_data(Header, Stream, Tail).

read_header_data("\r\n", _, _) :- !.
read_header_data("\n", _, _) :- !.
read_header_data("", _, _) :- !.
read_header_data(_, Stream, Tail) :-
        read_line_to_codes(Stream, Tail, NewTail),
        read_header_data(Tail, Stream, NewTail).
```

read_stream_to_codes(+*Stream, -Codes*)
Read all input until end-of-file and unify the result to *Codes*.

read_stream_to_codes(+*Stream, -Codes, ?Tail*)
Difference-list version of `read_stream_to_codes/2`.

read_file_to_codes(+*Spec, -Codes, +Options*)
Read a file to a list of character codes. *Spec* is a file-specification for `absolute_file_name/3`. *Codes* is the resulting code-list. *Options* is a list of options for `absolute_file_name/3` and `open/4`. In addition, the option `tail`(*Tail*) is defined, forming a difference-list.

read_file_to_terms(+*Spec, -Terms, +Options*)
Read a file to a list of prolog terms (see `read/1`). *Spec* is a file-specification for `absolute_file_name/3`. *Terms* is the resulting list of Prolog terms. *Options* is a list of options for `absolute_file_name/3` and `open/4`. In addition, the option `tail`(*Tail*) is defined, forming a difference-list.

A.10 `netscape`: Activating your Web-browser

This library deals with the very system dependent task of opening a web page in a browser. See also `url` and the HTTP package.

www_open_url(+*URL*)
Open *URL* in an external web-browser. The reason to place this in the library is to centralise the maintenance on this highly platform and browser specific task. It distinguishes between the following cases:

- *MS-Windows*
  If it detects MS-Windows it uses `win_shell/2` to open the *URL*. The behaviour and browser started depends on the Window and Windows-shell configuration, but in general it should be the behaviour expected by the user.
- *Other platforms*
  On other platforms it tests the environment variable (see `getenv/2`) named `BROWSER` or uses `netscape` if this variable is not set. If the browser is either `mozilla` or `netscape`, `www_open_url/1` first tries to open a new window on a running using the

`-remote` option of netscape. If this fails or the browser is not `mozilla` or `netscape` the system simply passes the URL as first argument to the program.

A.11 `registry`: Manipulating the Windows registry

The `registry` is only available on the MS-Windows version of SWI-Prolog. It loads the foreign extension `plregtry.dll`, providing the predicates described below. This library only makes the most common operations on the registry available through the Prolog user. The underlying DLL provides a more complete coverage of the Windows registry API. Please consult the sources in `pl/src/win32/foreign/plregtry.c` for further details.

In all these predicates, *Path* refers to a '/' separated path into the registry. This is *not* an atom containing '/'-characters as used for filenames, but a term using the functor `//2`. Windows defines the following roots for the registry: `classes_root`, `current_user`, `local_machine` and `users`

registry_get_key(+*Path, -Value*)
Get the principal (default) value associated to this key. Fails silently of the key does not exist.

registry_get_key(+*Path, +Name, -Value*)
Get a named value associated to this key.

registry_set_key(+*Path, +Value*)
Set the principal (default) value of this key. Creates (a path to) the key if this does not already exist.

registry_set_key(+*Path, +Name, +Value*)
Associated a named value to this key. Creates (a path to) the key if this does not already exist.

registry_delete_key(+*Path*)
Delete the indicated key.

shell_register_file_type(+*Ext, +Type, +Name, +OpenAction*)
Register a file-type. *Ext* is the extension to associate. *Type* is the type name, often something link `prolog.type`. *Name* is the name visible in the Windows file-type browser. Finally, *OpenAction* defines the action to execute when a file with this extension is opened in the Windows explorer.

shell_register_dde(+*Type, +Action, +Service, +Topic, +Command, +IfNotRunning*)
Associate DDE actions to a type. *Type* is the same type as used for the 2nd argument of `shell_register_file_type/4`, *Action* is the a action to perform, *Service* and *Topic* specify the DDE topic to address and *Command* is the command to execute on this topic. Finally, *IfNotRunning* defines the command to execute if the required DDE server is not present.

shell_register_prolog(+*Ext*)
Default registration of SWI-Prolog, which is invoked as part of the initialisation process on Windows systems. As the source also explains the above predicates, it is given as an example:

```
shell_register_prolog(Ext) :-
        current_prolog_flag(argv, [Me|_]),
        concat_atom(['"', Me, '" "%1"'], OpenCommand),
```

```
shell_register_file_type(Ext, 'prolog.type', 'Prolog Source',
                         OpenCommand),
shell_register_dde('prolog.type', consult,
                   prolog, control, 'consult(''%1'')', Me),
shell_register_dde('prolog.type', edit,
                   prolog, control, 'edit(''%1'')', Me).
```

A.12 url: Analysing and constructing URL

This library deals with the analysis and construction of a URL, **U**niversal **R**esource **L**ocator. URL is the basis for communicating locations of resources (data) on the web. A URL consists of a protocol identifier (e.g. HTTP, FTP), and a protocol-specific syntax further defining the location. URLs are standardized in RFC-1738.

The implementation in this library covers only a small portion of the defined protocols. Though the initial implementation followed RFC-1738 strictly, the current is more relaxed to deal with frequent violations of the standard encountered in practical use.

This library contains code by Jan Wielemaker who wrote the initial version and Lukas Faulstich who added various extensions.

parse_url(*?URL, ?Parts*)
Construct or analyse a *URL*. *URL* is an atom holding a URL or a variable. *Parts* is a list of components. Each component is of the format `Name`(*Value*). Defined components are:

protocol(*Protocol*)
The used protocol. This is, after the optional `url:`, an identifier separated from the remainder of the URL using `:`. `parse_url/2` assumes the `http` protocol if no protocol is specified and the URL can be parsed as a valid HTTP url. In addition to the RFC-1738 specified protocols, the `file:` protocol is supported as well.

host(*Host*)
Host-name or IP-address on which the resource is located. Supported by all network-based protocols.

port(*Port*)
Integer port-number to access on the *Host*. This only appears if the port is explicitly specified in the URL. Implicit default ports (e.g. 80 for HTTP) do *not* appear in the part-list.

path(*Path*)
(File-) path addressed by the URL. This is supported for the `ftp`, `http` and `file` protocols. If no path appears, the library generates the path `/`.

search(*ListOfNameValue*)
Search-specification of HTTP URL. This is the part after the `?`, normally used to transfer data from HTML forms that use the '`GET`' protocol. In the URL it consists of a www-form-encoded list of *Name*=*Value* pairs. This is mapped to a list of Prolog *Name*=*Value* terms with decoded names and values.

fragment(*Fragment*)
Fragment specification of HTTP URL. This is the part after the # character.

The example below illustrates the all this for an HTTP UTL.

```
?- parse_url('http://swi.psy.uva.nl/message.cgi?msg=Hello+World%21#x',
             P).
P = [ protocol(http),
      host('swi.psy.uva.nl'),
      fragment(x),
      search([ msg = 'Hello World!'
             ]),
      path('/message.cgi')
    ].
```

By instantiating the parts-list this predicate can be used to create a URL.

parse_url(*?URL, +BaseURL, ?Parts*)
Same as `parse_url/2`, but dealing a url that is relative to the given *BaseURL*. This is used to analyse or construct a URI found in the document behind *BaseURL*.

global_url(*+URL, +BaseURL, -AbsoluteUrl*)
Transform a (possibly) relative URL into a global one.

http_location(*?Parts, ?Location*)
Similar to `parse_url/2`, but only deals with the location part of an HTTP URL. That is, the path, search and fragment specifiers. In the HTTP protocol, the first line of a message is

Action Location [`HTTP/`*HttpVersion*]

Location is either an atom or a code-list.

www_form_encode(*?Value, ?WwwFormEncoded*)
Translate between a string-literal and the x-www-form-encoded representation used in path and search specifications of the HTTP protocol.

Encoding implies mapping space to +, preserving alpha-numercial characters, map newlines to %0D%0A and anything else to %XX. When decoding, newlines appear as a single newline (10) character.

A.13 `clp/bounds`: Integer Bounds Constraint Solver

Author: *Tom Schrijvers*, K.U.Leuven

The bounds solver is a rather simple integer constraint solver, implemented with attributed variables. Its syntax is a subset of the SICStus clp(FD) syntax.

Please note that the `clp/bounds` library is *not* an *autoload* library and therefore this library must be loaded explicitely before using it using:

```
:- use_module(library('clp/bounds')).
```

A.13.1 Constraints

The following constraints are supported:

-*Var* **in** +*Range*
: Varibale *Var* is restricted to be in range *Range*. A range is denoted by `L..U` where both *L* and *U* are integers.

-*Vars* **in** +*Range*
: A list of variables *Vars* are restriced to be in range *Range*.

tuples_in(+*Tuples*, +*Extension*)
: Where *Tuples* is a list of tuples (lists) of variables and integers, each of length N, and *Extension* is a list of tuples of integers, each of length N. Each tuple of *Tuples* is constrained to be in the relation defined by *Extension*. See section A.13.4 for an example.

?Expr **#>** *?Expr*
: The left-hand expression is constrained to be greater than the right-hand expressions.

?Expr **#<** *?Expr*
: The left-hand expression is constrained to be smaller than the right-hand expressions.

?Expr **#>=** *?Expr*
: The left-hand expression is constrained to be greater than or equal to the right-hand expressions.

?Expr **#=<** *?Expr*
: The left-hand expression is constrained to be smaller than or equal to the right-hand expressions.

?Expr **#=** *?Expr*
: The left-hand expression is constrained to be equal to the right-hand expressions.

?Expr **#\=** *?Expr*
: The left-hand expression is constrained to be not equal to the right-hand expressions.

sum(+*Vars*,+*Op*,*?Value*)
: Here *Vars* is a list of variables and integers, *Op* is one of the binary constraint relation symbols above and *Value* is an integer or variable. It represents the constraint ($\sum$*Vars*) *Op Value*.

lex_chain(+*VarsLists*)
: The constraint enforces lexicographic ordering on the lists in the argument. The argument *Vars* is a list of lists of variables and integers. The current implementation was contributed by Markus Triska.

all_different(+*Vars*)
: Constrains all variabls in the list *Vars* to be pairwise not equal.

indomain(+*Var*)
: Assigns a value in its domain to variable *Var*. Backtracks over all possible values from lowest to greatest. Contributed by Markus Triska.

label(+*Vars*)
: All variables are assigned a variable that does not violate the constraint on them.

Here *Expr* can be one of

integer Any integer.

variable A variable.

?Expr **+** *?Expr*
The sum of two expressions.

?Expr ***** *?Expr*
The product of two expressions.

?Expr **-** *?Expr*
The difference of two expressions.

max(*?Expr,?Expr***)**
The maximum of two expressions.

min(*?Expr,?Expr***)**
The minimum of two expressions.

?Expr **mod** *?Expr*
The first expression modulo the second expression.

abs(*?Expr***)**
The absolute value of an expression.

A.13.2 Constraint Implication and Reified Constraints

The following constraint implication predicates are available:

+P **#=>** *+Q*
P implies *Q*, where *P* and *Q* are reifyable constraints.

+Q **#<=** *+P*
P implies *Q*, where *P* and *Q* are reifyable constraints.

+P **#<=>** *+Q*
P and *Q* are equivalent, where *P* and *Q* are reifyable constraints.

In addition, instead of being a reifyable constraint, either *P* or *Q* can be a boolean variable that is the truth value of the corresponding constraint.
The following constraints are reifyable: `#=/2`, `#\=/2`, `#</2`, `#>/2`, `#=</2`, `#>/2`.

For example, to count the number of occurrences of a particular value in a list of constraint variables:

- *Using constraint implication*

```
occurrences(List,Value,Count) :-
        occurrences(List,Value,0,Count).

occurrences([],_,Count,Count).
occurrences([X|Xs],Value,Acc,Count) :-
```

```
            X #= Value  #=> NAcc #= Acc + 1,
            X #\= Value #=> NAcc #= Acc,
            occurrences(Xs,Value,NAcc,Count).
```

- *Using reified constraints*

```
    occurrences(List,Value,Count) :-
            occurrences(List,Value,0,Count).

    occurrences([],_,Count,Count).
    occurrences([X|Xs],Value,Acc,Count) :-
            X #= Value #=> B,
            NAcc #= Acc + B,
            occurrences(Xs,Value,NAcc,Count).
```

A.13.3 Example 1: Send+More=Money

The following is an implementation of the classic alphametics puzzle SEND + MORE = MONEY:

```
:- use_module(library('clp/bounds')).

send([[S,E,N,D], [M,O,R,E], [M,O,N,E,Y]])  :-
             Digits   =  [S,E,N,D,M,O,R,Y],
             Carries  =  [C1,C2,C3,C4],
             Digits  in  0..9,
             Carries in  0..1,

             M                #=              C4,
             O  +  10  *  C4  #=  M  +  S  +  C3,
             N  +  10  *  C3  #=  O  +  E  +  C2,
             E  +  10  *  C2  #=  R  +  N  +  C1,
             Y  +  10  *  C1  #=  E  +  D,

             M  #>=  1,
             S  #>=  1,
             all_different(Digits),
             label(Digits).
```

A.13.4 Example 2: Using tuples_in for a train schedule

This example demonstrates `tuples_in/2`. A train schedule is represented as a list Ts of quadruples, denoting departure and arrival places and times for each train. The *path/3* predicate given below constrains Ps to a feasible journey from A to D via 3 trains that are part of the given schedule.

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

schedule(Ts) :-
        Ts = [[1,2,0,1],[2,3,4,5],[2,3,0,1],[3,4,5,6],[3,4,2,3],[3,4,8,9]].
```

```
path(A, D, Ps) :-
        schedule(Ts),
        Ps = [[A,B,_T0,T1],[B,C,T2,T3],[C,D,T4,_T5]],
        tuples_in(Ps, Ts),
        T2 #> T1,
        T4 #> T3.
```

An example query:

```
?- path(1, 4, Ps), flatten(Ps, Vars), label(Vars).

Ps = [[1, 2, 0, 1], [2, 3, 4, 5], [3, 4, 8, 9]]
```

A.13.5 SICStus clp(FD) compatibility

Apart from the limited syntax, the bounds solver differs in the following ways from the SICStus clp(FD) solver:

- *inf and sup*
  The smallest lowerbound and greatest upperbound in bounds are `max_integer` and `min_integer + 1`.

A.14 `clpqr`: Constraint Logic Programming over Rationals and Reals

Author: *Leslie De Koninck*, K.U. Leuven

This CLP(Q,R) system is a port of the CLP(Q,R) system of Sicstus Prolog by Christian Holzbaur: Holzbaur C.: OFAI clp(q,r) Manual, Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.[3] This manual is roughly based on the manual of the above mentioned CLP(Q,R) implementation.

The CLP(Q,R) system consists of two components: the CLP(Q) library for handling constraints over the rational numbers and the CLP(R) library for handling constraints over the real numbers (using floating point numbers as representation). Both libraries offer the same predicates (with exception of `bb_inf/4` in CLP(Q) and `bb_inf/5` in CLP(R)). It is allowed to use both libraries in one program, but using both CLP(Q) and CLP(R) constraints on the same variable will result in an exception.

Please note that the `clpqr` library is *not* an *autoload* library and therefore this library must be loaded explicitely before using it:

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

or

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

[3] http://www.ai.univie.ac.at/cgi-bin/tr-online?number+95-09

A.14.1 Solver predicates

The following predicates are provided to work with constraints:

{}(+*Constraints*)
Adds the constraints given by *Constraints* to the constraint store.

entailed(+*Constraint*)
Succeeds if *Constraint* is necessarily true within the current constraint store. This means that adding the negation of the constraint to the store results in failure.

inf(+*Expression, -Inf*)
Computes the infimum of *Expression* within the current state of the constraint store and returns that infimum in *Inf*. This predicate does not change the constraint store.

sup(+*Expression, -Sup*)
Computes the supremum of *Expression* within the current state of the constraint store and returns that supremum in *Sup*. This predicate does not change the constraint store.

minimize(+*Expression*)
Minimizes *Expression* within the current constraint store. This is the same as computing the infimum and equation the expression to that infimum.

maximize(+*Expression*)
Maximizes *Expression* within the current constraint store. This is the same as computing the supremum and equating the expression to that supremum.

bb_inf(+*Ints, +Expression, -Inf, -Vertex, +Eps*)
This predicate is offered in CLP(R) only. It computes the infimum of *Expression* within the current constraint store, with the additional constraint that in that infimum, all variables in *Ints* have integral values. *Vertex* will contain the values of *Ints* in the infimum. *Eps* denotes how much a value may differ from an integer to be considered an integer. E.g. when *Eps* = 0.001, then X = 4.999 will be considered as an integer (5 in this case). *Eps* should be between 0 and 0.5.

bb_inf(+*Ints, +Expression, -Inf, -Vertex*)
This predicate is offered in CLP(Q) only. It behaves the same as `bb_inf/5` but does not use an error margin.

bb_inf(+*ints, +Expression, -Inf*)
The same as `bb_inf/5` or `bb_inf/4` but without returning the values of the integers. In CLP(R), an error margin of 0.001 is used.

dump(+*Target, +Newvars, -CodedAnswer*)
Returns the constraints on *Target* in the list *CodedAnswer* where all variables of *Target* have veen replaced by *NewVars*. This operation does not change the constraint store. E.g. in

```
dump([X,Y,Z],[x,y,z],Cons)
```

Cons will contain the constraints on X, Y and Z where these variables have been replaced by atoms x, y and z.

⟨*Constraints*⟩	::=	⟨*Constraint*⟩	single constraint
	|	⟨*Constraint*⟩ , ⟨*Constraints*⟩	conjunction
	|	⟨*Constraint*⟩ ; ⟨*Constraints*⟩	disjunction
⟨*Constraint*⟩	::=	⟨*Expression*⟩ < ⟨*Expression*⟩	less than
	|	⟨*Expression*⟩ > ⟨*Expression*⟩	greater than
	|	⟨*Expression*⟩ =< ⟨*Expression*⟩	less or equal
	|	<=(⟨*Expression*⟩, ⟨*Expression*⟩)	less or equal
	|	⟨*Expression*⟩ >= ⟨*Expression*⟩	greater or equal
	|	⟨*Expression*⟩ =\= ⟨*Expression*⟩	not equal
	|	⟨*Expression*⟩ =:= ⟨*Expression*⟩	equal
	|	⟨*Expression*⟩ = ⟨*Expression*⟩	equal
⟨*Expression*⟩	::=	⟨*Variable*⟩	Prolog variable
	|	⟨*Number*⟩	Prolog number (float, integer)
	|	+⟨*Expression*⟩	unary plus
	|	-⟨*Expression*⟩	unary minus
	|	⟨*Expression*⟩ + ⟨*Expression*⟩	addition
	|	⟨*Expression*⟩ - ⟨*Expression*⟩	substraction
	|	⟨*Expression*⟩ * ⟨*Expression*⟩	multiplication
	|	⟨*Expression*⟩ / ⟨*Expression*⟩	division
	|	abs(⟨*Expression*⟩)	absolute value
	|	sin(⟨*Expression*⟩)	sine
	|	cos(⟨*Expression*⟩)	cosine
	|	tan(⟨*Expression*⟩)	tangent
	|	exp(⟨*Expression*⟩)	exponent
	|	pow(⟨*Expression*⟩)	exponent
	|	⟨*Expression*⟩ ^ ⟨*Expression*⟩	exponent
	|	min(⟨*Expression*⟩, ⟨*Expression*⟩)	minimum
	|	max(⟨*Expression*⟩, ⟨*Expression*⟩)	maximum

Table A.1: CLP(Q,R) constraint BNF

A.14.2 Syntax of the predicate arguments

The arguments of the predicates defined in the subsection above are defined in table A.1. Failing to meet the syntax rules will result in an exception.

A.14.3 Use of unification

Instead of using the `{}/1` predicate, you can also use the standard unification mechanism to store constraints. The following code samples are equivalent:

- *Unification with a variable*

```
{X =:= Y}
{X = Y}
X = Y
```

- *Unification with a number*

$A = B * C$	B or C is ground A and (B or C) are ground	A = 5 * C or A = B * 4 20 = 5 * C or 20 = B * 4
$A = B/C$	C is ground A and B are ground	A = B / 3 4 = 12 / C
$X = min(Y, Z)$ $X = max(Y, Z)$ $X = abs(Y)$	Y and Z are ground Y and Z are ground Y is ground	X = min(4,3) X = max(4,3) X = abs(-7)
$X = pow(Y, Z)$ $X = exp(Y, Z)$ $X = Y \hat{} Z$	X and Y are ground X and Z are ground Y and Z are ground	8 = 2 ^ Z 8 = Y ^ 3 X = 2 ^ 3
$X = sin(Y)$ $X = cos(Y)$ $X = tan(Y)$	X is ground Y is ground	1 = sin(Y) X = sin(1.5707)

Table A.2: CLP(Q,R) isolating axioms

```
{X =:= 5.0}
{X = 5.0}
X = 5.0
```

A.14.4 Non-linear constraints

The CLP(Q,R) system deals only passively with non-linear constraints. They remain in a passive state until certain conditions are satisfied. These conditions, which are called the isolation axioms, are given in table A.2.

A.15 clp/clp_distinct: Weak arc consistent 'all_distinct' constraint

Author: *Markus Triska*

The `clp/clp_distinct` module provides the following constraints:

all_distinct(+*Vars*)
: The variables in *Vars* are constrained to be pairwise distinct. All variables must already be assigned domains (via `vars_in/2` or `vars_in/3`) when this constraint is posted.

vars_in(+*Vars, +Domain*)
: Where *Vars* is a list of variables and *Domain* is a list of non-negative integers. Each variable of *Vars* is constrained to be in *Domain*.

vars_in(+*Vars, +From, +To*)
: Where *Vars* is a list of variables, and $0 \leq From \leq To$. Each variable in *Vars* is constrained to be in the discrete interval [From,To].

A.15.1 Example 1

The `all_distinct/1` constraint can detect various inconsistencies:

```
?- vars_in([X,Y,Z], [1,2]), all_distinct([X,Y,Z]).

No
```

A.15.2 Example 2

In this example, 3 is assigned to Z without labeling any variables:

```
?- vars_in([X,Y], [1,2]), vars_in([Z], [1,2,3]), all_distinct([X,Y,Z]).

X = _G180{1-2}
Y = _G183{1-2}
Z = 3 ;
```

A.15.3 Example 3

The clp_distinct module can be used in conjunction with clp/bounds. All relevant variables must still be assigned domains via one of the *vars_in* predicates before `all_distinct/1` can be posted:

```
:- use_module(library(bounds)).
:- use_module(library(clp_distinct)).

?- [X,Y] in 1..2, vars_in([X,Y], [1,2]), all_distinct([X,Y]), label([X,Y]).

X = 1
Y = 2 ;

X = 2
Y = 1 ;
```

A.16 `simplex`: Solve linear programming problems

Author: *Markus Triska*

A linear programming problem consists of a set of (linear) constraints, a number of variables and a linear objective function. The goal is to assign values to the variables so as to maximize (or minimize) the value of the objective function while satisfying all constraints.

Many optimization problems can be modeled in this way. Consider having a knapsack with fixed capacity C, and a number of items with sizes s(i) and values v(i). The goal is to put as many items as possible in the knapsack (not exceeding its capacity) while maximizing the sum of their values.

As another example, suppose you are given a set of coins with certain values, and you are to find the minimum number of coins such that their values sum up to a fixed amount. Instances of these problems are solved below.

The `simplex` module provides the following predicates:

assignment(+*Cost, -Assignment*)
Cost is a list of lists representing the quadratic cost matrix, where element (i,j) denotes the cost of assigning entity i to entity j. An assignment with minimal cost is computed and unified with *Assignment* as a list of lists, representing an adjacency matrix.

constraint(+*Constraint, +S0, -S*)
Adds a linear or integrality constraint to the linear program corresponding to state *S0*. A linear constraint is of the form "Left Op C", where "Left" is a list of Coefficient*Variable terms (variables in the context of linear programs can be atoms or compound terms) and C is a non-negative numeric constant. The list represents the sum of its elements. *Op* can be =, =¡ or ¿=. The coefficient "1" can be omitted. An integrality constraint is of the form integral(Variable) and constrains Variable to an integral value.

constraint(+*Name, +Constraint, +S0, -S*)
Like `constraint/3`, and attaches the name *Name* (an atom or compound term) to the new constraint.

constraint_add(+*Name, +Left, +S0, -S*)
Left is a list of Coefficient*Variable terms. The terms are added to the left-hand side of the constraint named *Name*. *S* is unified with the resulting state.

gen_state(-*State*)
Generates an initial state corresponding to an empty linear program.

maximize(+*Objective, +S0, -S*)
Maximizes the objective function, stated as a list of "Coefficient*Variable" terms that represents the sum of its elements, with respect to the linear program corresponding to state *S0*. *S* is unified with an internal representation of the solved instance.

minimize(+*Objective, +S0, -S*)
Analogous to `maximize/3`.

objective(+*State, -Objective*)
Unifies *Objective* with the result of the objective function at the obtained extremum. *State* must correspond to a solved instance.

shadow_price(+*State, +Name, -Value*)
Unifies *Value* with the shadow price corresponding to the linear constraint whose name is *Name*. *State* must correspond to a solved instance.

transportation(+*Supplies, +Demands, +Costs, -Transport*)
Supplies and *Demands* are both lists of positive numbers. Their respective sums must be equal. *Costs* is a list of lists representing the cost matrix, where an entry (i,j) denotes the cost of transporting one unit from i to j. A transportation plan having minimum cost is computed and unified with *Transport* in the form of a list of lists that represents the transportation matrix, where element (i,j) denotes how many units to ship from i to j.

variable_value(+*State, +Variable, -Value*)
Value is unified with the value obtained for *Variable*. *State* must correspond to a solved instance.

All numeric quantities are converted to rationals via `rationalize/1`, and rational arithmetic is used throughout solving linear programs. In the current implementation, all variables are implicitly constrained to be non-negative. This may change in future versions, and non-negativity constraints should therefore be stated explicitly.

A.16.1 Example 1

This is the ”radiation therapy” example, taken from ”Introduction to Operations Research” by Hillier and Lieberman. DCG notation is used to implicitly thread the state through posting the constraints:

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

post_constraints -->
        constraint([0.3*x1, 0.1*x2] =< 2.7),
        constraint([0.5*x1, 0.5*x2] = 6),
        constraint([0.6*x1, 0.4*x2] >= 6),
        constraint([x1] >= 0),
        constraint([x2] >= 0).

radiation(S) :-
        gen_state(S0),
        post_constraints(S0, S1),
        minimize([0.4*x1, 0.5*x2], S1, S).
```

An example query:

```
?- radiation(S), variable_value(S, x1, Val1), variable_value(S, x2, Val2).

Val1 = 15 rdiv 2
Val2 = 9 rdiv 2 ;
```

A.16.2 Example 2

Here is an instance of the knapsack problem described above, where C = 8, and we have two types of items: One item with value 7 and size 6, and 2 items each having size 4 and value 4. We introduce two variables, x(1) and x(2) that denote how many items to take of each type.

```
knapsack_constrain(S) :-
        gen_state(S0),
        constraint([6*x(1), 4*x(2)] =< 8, S0, S1),
        constraint([x(1)] =< 1, S1, S2),
        constraint([x(2)] =< 2, S2, S).

knapsack(S) :-
        knapsack_constrain(S0),
        maximize([7*x(1), 4*x(2)], S0, S).
```

An example query yields:

```
?- knapsack(S), variable_value(S, x(1), X1), variable_value(S, x(2), X2).

X1 = 1
X2 = 1 rdiv 2 ;
```

That is, we are to take the one item of the first type, and half of one of the items of the other type to maximize the total value of items in the knapsack.

If items can not be split, integrality constraints have to be imposed:

```
knapsack_integral(S) :-
        knapsack_constrain(S0),
        constraint(integral(x(1)), S0, S1),
        constraint(integral(x(2)), S1, S2),
        maximize([7*x(1), 4*x(2)], S2, S).
```

Now the result is different:

```
?- knapsack_integral(S), variable_value(S, x(1), X1), variable_value(S, x(2), X2)

X1 = 0
X2 = 2
```

That is, we are to take only the two items of the second type. Notice in particular that always choosing the remaining item with best performance (ratio of value to size) that still fits in the knapsack does not necessarily yield an optimal solution in the presence of integrality constraints.

A.16.3 Example 3

We are given 3 coins each worth 1, 20 coins each worth 5, and 10 coins each worth 20 units of money. The task is to find a minimal number of these coins that amount to 111 units of money. We introduce variables c(1), c(5) and c(20) denoting how many coins to take of the respective type:

```
coins -->
        constraint([c(1), 5*c(5), 20*c(20)] = 111),
        constraint([c(1)] =< 3),
        constraint([c(5)] =< 20),
        constraint([c(20)] =< 10),
        constraint([c(1)] >= 0),
        constraint([c(5)] >= 0),
        constraint([c(20)] >= 0),
        constraint(integral(c(1))),
        constraint(integral(c(5))),
        constraint(integral(c(20))),
        minimize([c(1), c(5), c(20)]).

coins(S) :-
        gen_state(S0),
        coins(S0, S).
```

An example query:

```
?- coins(S), variable_value(S, c(1), C1), variable_value(S, c(5), C5), variable_v

C1 = 1
C5 = 2
C20 = 5
```

A.17 `prologxref`: Cross-reference data collection library

This library collects information on defined and used objects in Prolog sourcefiles. Typically these are predicates, but we expect the library to deal with other types of objects in the future. The library is a building block for tools doing dependency tracking in applications. Dependency tracking is useful to reveal the structure of an unknown program or detect missing components at compile-time, but also for program transformation or minimising a program saved-state by only saving the reachable objects.

This section gives a partial description of the library API, providing some insight in how you can use it for analysing your program. The library should be further modularized, moving its knowledge about -for example- XPCE into a different file and allowing for adding knowledge about other libraries such as Logtalk. **Please do not consider this interface rock-solid.**

The library is exploited by two graphical tools in the SWI-Prolog environment. The XPCE frontend started by `gxref/0` and described in section 3.7 and PceEmacs (section 3.4) which exploits this library for its syntax colouring.

For all predicates described below, *Source* is the source that is processed. This is normally a filename in any notation acceptable to the file loading predicates (see `load_files/2`). Using the hooks defined in section A.17.1 it can be anything else that can be translated into a Prolog stream holding Prolog source text. *Callable* is a callable term (see `callable/1`). Callables do not carry a module qualifier unless the referred predicate is not in the module defined *Source*.

xref_source(+*Source*)
: Gather information on *Source*. If *Source* has already been processed and is still up-to-date according to the file timestamp, no action is taken. This predicate must be called on a file before information can be gathered.

xref_current_source(*?Source*)
: *Source* has been processed.

xref_clean(+*Source*)
: Remove the information gathered for *Source*

xref_defined(*?Source, ?Callable, -How*)
: *Callable* is defined in *Source*. *How* is one of

`dynamic`(*Line*)	Declared dynamic at *Line*
`thread_local`(*Line*)	Declared thread local at *Line*
`multifile`(*Line*)	Declared multifile at *Line*
`local`(*Line*)	First clause at *Line*
`foreign`(*Line*)	Foreign library loaded at *Line*
`constraint`(*Line*)	CHR Constraint at *Line*
`imported`(*File*)	Imported from *File*

xref_called(*?Source, ?Callable, ?By*)
Callable is called in *Source* by *By*.

xref_exported(*?Source, ?Callable*)
Callable is public (exported from the module).

xref_module(*?Source, ?Module*)
Source is a module-file defining the given module.

xref_built_in(*?Callable*)
True if *Callable* is a built-in predicate. Currently this is assumed for all predicates defined in the `system` module and having the property `built_in`. Built-in predicates are not registered as 'called'.

A.17.1 Extending the library

The library provides hooks for extending its rules it uses for finding predicates called by some programming construct.

prolog:called_by(+*Goal, -Called*)
Where *Goal* is a non-var subgoal appearing in called object (typically a clause-body). If it succeeds it must return a list of goals called by *Goal*. As a special construct, if a term $Callable+N$ is returned, N variable arguments are added to *Callable* before further processing. For simple meta-calls a single fact suffices. Complex rules as used in the `html_write` library provided by the HTTP package examine the arguments and create a list of called objects.

The current system cannot deal with the same name/arity in different modules that behave differently with respect to called arguments.

Hackers corner

B

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.

B.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 $\langle module \rangle : \langle goal \rangle$. Do not instantiate variables in this goal unless you **know** what you are doing! Note that the returned term may contain references to the frame and should be discarded before the frame terminates.[1]

parent_goal
If *Value* is instantiated to a callable term, find a frame executing the predicate described by *Value* and unify the arguments of *Value* to the goal arguments associated with the frame. This is intended to check the current execution context. The user must ensure the checked parent goal is not removed from the stack due to last-call optimisation and be aware of the slow operation on deeply nested calls.

[1] The returned term is actually an illegal Prolog term that may hold references from the global- to the local stack to preserve the variable names.

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 behalf 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.

prolog_choice_attribute(+*ChoicePoint*, +*Key*, -*Value*)
Extract attributes of a choice-point. *ChoicePoint* is a reference to a choice-point as passed to `prolog_trace_interception/4` on the 3-th argument. *Key* specifies the requested information:

parent
Requests a reference to the first older choice-point.

frame
Requests a reference to the frame to which the choice-point refers.

type
Requests the type. Defined values are `clause` (the goal has alternative clauses), `foreign` (non-deterministic foreign predicate), `jump` (clause internal choice-point), `top` (first dummy choice-point), `catch` (`catch/3` to allow for undo), `debug` (help the debugger), or `none` (has been deleted).

This predicate is used for the graphical debugger to show the choice-point stack.

deterministic(-*Boolean*)
Unifies its argument with `true` if the clause in which is appears has not created any choice-points since it was started. There are few realistic situations for using this predicate. It is used by the `prolog/0` toplevel to check whether Prolog should prompt the user for alternatives.

B.2 Intercepting the Tracer

prolog_trace_interception(+*Port*, +*Frame*, +*Choice*, -*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 denotes the reason to activate the tracer ('port' in the 4/5-port, but with some additions:

call
Normal entry through the call-port of the 4-port debugger.

redo
Normal entry through the call-port of the 4-port debugger. The `redo` port signals resuming a predicate to generate alternative solutions.

unify
The unify-port represents the *neck* instruction, signalling the end of the head-matching process. This port is normally invisible. See `leash/1` and `visible/1`.

exit
The exit-port signals the goal is proved. It is possible for the goal to have alternative. See `prolog_frame_attribute/3` to examine the goal-stack.

fail
The fail-port signals final failure of the goal.

exception(*Except*)
An exception is raised and still pending. This port is activated on each parent frame of the frame generating the exception until the exception is caught or the user restarts normal computation using `retry`. *Except* is the pending exception-term.

break(*PC*)
A `break` instruction is executed. *PC* is program counter. This port is used by the graphical debugger.

cut_call(*PC*)
A cut is encountered at *PC*. This port is used by the graphical debugger. to visualise the effect of the cut.

cut_exit(*PC*)
A cut has been executed. See `cut_call`(*PC*) for more information.

Frame is a reference to the current local stack frame, which can be examined using `prolog_frame_attribute/3`. *Choice* is a reference to the last choice-point and can be examined using `prolog_choice_attribute/3`. *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 4.38 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).

B.3 Adding context to errors: prolog_exception_hook

The hook `prolog_exception_hook/4` has been introduced in SWI-Prolog 5.6.5 to provide dedicated exception handling facilities for application frameworks. For example non-interactive server applications that wish to provide extensive context for exceptions for offline debugging.

prolog_exception_hook(*+ExceptionIn, -ExceptionOut, +Frame, +CatcherFrame*)
This hook predicate, if defined in the module `user`, is between raising an exception and handling it. It is intended to allow a program adding additional context to an exception to simplify diagnosing the problem. *ExceptionIn* is the exception term as raised by `throw/1` or one of the bullt-in predicates. The output argument *ExceptionOut* describes the exception that is actually raised. *Frame* is the innermost frame. See `prolog_frame_attribute/3` and the library `prolog_stack` for getting information from this. *CatcherFrame* is a reference to the frame calling the matching `catch/3` or `none` of the exception is not caught.

The hook is run in 'nodebug' mode. If it succeeds *ExceptionOut* is considered the current exception. If it fails, *ExceptionIn* is used for further processing. The hook is *never* called recursively. The hook is *not* allowed to modify *ExceptionOut* in such as way that it no longer unifies with the catching frame.

Typically, `prolog_exception_hook/4` is used to fill the second argument of `error`(*Formal, Context*) exceptions. *Formal* is defined by the ISO standard, while SWI-Prolog defines *Context* as a term `context`(*Location, Message*). *Location* is bound to a term ⟨*name*⟩/⟨*arity*⟩ by the kernel. This hook can be used to add more information on the calling context, such as a full stack trace.

Applications that use exceptions as part of normal processing must do a quick test of the environment before starting expensive gathering information on the state of the program.

The hook can call `trace/0` to enter trace mode immediately.

B.4 Hooks using the exception predicate

This section describes the predicate `exception/3`, which can be defined by the user in the module `user` as a multifile predicate. Unlike the name suggests, this is actually a *hook* predicate that has no relation to Prolog exceptions as defined by the ISO predicates `catch/3` and `throw/1`.

The predicate `exception/3` is called by the kernel on a couple of events, allowing the user to 'fix' errors just in time events. The mechanism allows for *lazy* creation of objects such as predicates.

exception(+*Exception, +Context, -Action*)
Dynamic predicate, normally not defined. Called by the Prolog system on run-time exceptions that can be repaired 'just in time'. The values for *Exception* are described below. See also `catch/3` and `throw/1`.

If this hook predicate succeeds it must instantiate the *Action* argument to the atom `fail` to make the operation fail silently, `retry` to tell Prolog to retry the operation or `error` to make the system generate an exception. The action `retry` only makes sense if this hook modified the environment such that the operation can now succeed without error.

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 `existence_error` exception.

undefined_global_variable
Context is instantiated to the name of the missing global variable. The hook must call `nb_setval/2` or `b_setval/2` before returning with the action `retry`.

B.5 Hooks for integrating libraries

Some libraries realise an entirely new programming paradigm on top of Prolog. An example is XPCE which adds an object-system to Prolog as well as an extensive set of graphical primitives. SWI-Prolog provides several hooks to improve the integration of such libraries. See also section 4.4 for editing hooks and section 4.9.3 for hooking into the message system.

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`.

prolog:debug_control_hook(*:Action*)
Hook for the debugger-control predicates that allows the creator of more high-level programming languages to use the common front-end predicates to control de debugger. For example, XPCE uses these hooks to allow for spying methods rather then predicates. *Action* is one of:

spy(*Spec*)
Hook in `spy/1`. If the hook succeeds `spy/1` takes no further action.

nospy(*Spec*)
Hook in `nospy/1`. If the hook succeeds `spy/1` takes no further action. If `spy/1` is hooked, it is advised to place a complementary hook for `nospy/1`.

nospyall
Hook in `nospyall/0`. Should remove all spy-points. This hook is called in a failure-driven loop.

debugging
Hook in `debugging/0`. It can be used in two ways. It can report the status of the additional debug-points controlled by the above hooks and fail to let the system report the others or it succeed, overruling the entire behaviour of `debugging/0`.

prolog:help_hook(+*Action*)
Hook into `help/0` and `help/1`. If the hook succeeds, the built-in actions are not executed. For example, `?- help(picture).` is caught by the XPCE help-hook to give help on the class *picture*. Defined actions are:

help
User entered plain `help/0` to give default help. The default performs `help(help/1)`, giving help on help.

help(*What*)
Hook in `help/1` on the topic *What*.

apropos(*What*)
Hook in `apropos/1` on the topic *What*.

B.6 Hooks for loading files

All loading of source-files is achieved by `load_files/2`. The hook `prolog_load_file/2` can be used to load Prolog code from non-files or even load entirely different information, such as foreign files.

prolog_load_file(+*Spec, +Options*)
Load a single object. If this call succeeds, `load_files/2` assumes the action has been taken care of. This hook is only called if *Options* does not contain the `stream`(*Input*) option. The hook must be defined in the module `user`.

The `http_load` provides an example, loading Prolog sources directly from an HTTP server.

prolog:comment_hook(+*Comments, +Pos, +Term*)
This hook allows for processing —structured— comments encountered by the compiler. The reader collects all comments found from the current position to the end of the next term. It calls this hook providing a list of *Position-Comment* in *Comments*, the start-position of the next term in *Pos* and the next term itself in *Term*. All positions are stream-position terms. This hook is exploited by the documentation system. See `stream_position_data/3`. See also `read_term/3`.

B.7 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

C

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 proof 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 crippled 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 5.

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 all implementations of SWI-Prolog integers are at least 64-bit signed values. When linked to the GNU GMP library, integer arithmetic is unbounded. See also `current_prolog_flag/2`, flags `bounded`, `max_integer` and `min_integer`.

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 `multifile/1`.

neck
Operator (`:-`) separating *head* from *body* in a *clause*.

operator
Symbol (*atom*) that may be placed before its *operand* (prefix), after its *operand* (postfix) or between its two *operands* (infix).

In Prolog, the expression `a+b` is exactly the same as the canonical term `+(a,b)`.

operand
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*). Rules for naming a variable and avoiding a warning are given in section 2.15.1.

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 4.23, 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*.

SWI-Prolog License Conditions and Tools D

SWI-Prolog licensing aims at a large audience, combining ideas from the Free Software Foundation and the less principal Open Source Initiative. The license aims at:

- Make SWI-Prolog itself and its libraries are 'As free as possible'.
- Allow for easy integration of contributions. See section D.2.
- Free software can build on SWI-Prolog without limitations.
- Non-free (open or proprietary) software can be produced using SWI-Prolog, although contributed pure GPL-ed components cannot be used.

To achieve this, different parts of the system have different licenses. SWI-Prolog programs consists of a mixture of 'native' code (source compiled to machine instructions) and 'virtual machine' code (Prolog source compiled to SWI-Prolog virtual machine instructions, covering both compiled SWI-Prolog libraries and your compiled application).

For maximal coherence between free licenses, we start with the two prime licenses from the Free Software Foundation, the GNU General Public License (GPL) and the Lesser GNU General Public License (LGPL), after which we add a proven (used by the GNU-C compiler runtime library as well as the GNU *ClassPath* project) exception to deal with the specific nature of compiled virtual machine code in a saved state.

D.1 The SWI-Prolog kernel and foreign libraries

The SWI-Prolog kernel and our foreign libraries are distributed under the **LGPL**. A Prolog executable consists of the combination of these 'native' code components and Prolog virtual machine code. The SWI-Prolog `plrc` utility allows for disassembling and re-assembling these parts, a process satisfying article **6b** of the LGPL.

Under the LGPL SWI-Prolog can be linked to code distributed under arbitrary licenses, provided a number of requirements are fullfilled. The most important requirement is that, if an application replies on a *modified* version of SWI-Prolog, the modified sources must be made available.

D.1.1 The SWI-Prolog Prolog libraries

Lacking a satisfactory technical solution to handle article **6** of the LGPL, this license cannot be used for the Prolog source code that is part of the SWI-Prolog system (both libraries and kernel code). This situation is comparable to `libgcc`, the runtime library used with the GNU C-compiler. Therefore, we use the same proven license terms as this library. The libgcc license is the with a special exception. Below we rephrased this exception adjusted to our needs:

As a special exception, if you link this library with other files, compiled with a Free Software compiler, to produce an executable, this library does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

D.2 Contributing to the SWI-Prolog project

To achieve maximal coherence using SWI-Prolog for Free and Non-Free software we advice the use of the LGPL for contributed foreign code and the use of the GPL with SWI-Prolog exception for Prolog code for contributed modules.

As a rule of thumb it is advised to use the above licenses whenever possible and only use a strict GPL compliant license only if the module contains other code under strict GPL compliant licenses.

D.3 Software support to keep track of license conditions

Given the above, it is possible that SWI-Prolog packages and extensions will rely on the GPL.[1] The predicates below allow for registering license requirements for Prolog files and foreign modules. The predicate `eval_license/0` reports which components from the currenly configured system are distributed under copy-left and open source enforcing licenses (the GPL) and therefore must be replaced before distributing linked applications under non-free license conditions.

eval_license
Evaluate the license conditions of all loaded components. If the system contains one or more components that are licenced under GPL-like restrictions the system indicates this program may only be distributed under the `GPL` license as well as which components prohibit the use of other license conditions.

license(+*LicenseId*, +*Component*)
Register the fact that *Component* is distributed under a license identified by *LicenseId*. The most important *LicenseId*'s are:

swipl
Indicates this module is distributed under the GNU General Public License (GPL) with the SWI-Prolog exception:[2]

As a special exception, if you link this library with other files, compiled with SWI-Prolog, to produce an executable, this library does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

[1] On the Unix version, the default toplevel uses the GNU readline library for command-line editing. This library is distributed under the GPL. In practice this problem is small as most final applications have Prolog embedded, without direct access to the commandline and therefore without need for `libreadline`.

[2] This exception is a straight re-phrasing of the license used for `libgcc`, the GNU-C runtime library facing similar technical issues.

This should be the default for software contributed to the SWI-Prolog project as it allows the community to prosper both in the free and non-free world. Still, people using SWI-Prolog to create non-free applications must contribute sources to improvements they make to the community.

lgpl
This is the default license for foreign-libraries linked with SWI-Prolog. Use PL_license() to register the condition from foreign code.

gpl
Indicates this module is strictly Free Software, which implies it cannot be used together with any module that is incompatible to the GPL. Please only use these conditions when forced by other code used in the component.

Other licenses known to the system are `guile`, `gnu_ada`, `x11`, `expat`, `sml`, `public_domain`, `cryptix`, `bsd`, `zlib`, constlgpl_compatible and `gpl_compatible`. New licenses can be defined by adding clauses for the multifile predicate license:license/3. Below is an example. The second argument is either `gpl` or `lgpl` to indicate compatibility to these licenses. Other values cause the license to interpreted as *proprietary*. Proprietary licenses are reported by `eval_license/0`. See the file `boot/license.pl` for details.

```
:- multifile license:license/3.

license:license(mylicense, lgpl,
                [ comment('My personal license'),
                  url('http://www.mine.org/license.html')
                ]).

:- license(mylicense).
```

license(+*LicenseId*)
Intented as a directive in Prolog source files. It takes the current filename and calls `license/2`.

`void` **PL_license**(*const char *LicenseId, const char *Component*)
Intended for the install() procedure of foreign libraries. This call can be made *before* PL_initialise().

D.4 Library predicates

D.4.1 `check`

check/0	Program completeness and consistency
list_undefined/0	List undefined predicates
list_autoload/0	List predicates that require autoload
list_redefined/0	List locally redefined predicates

D.4.2 `lists`

append/3	Concatenate lists
delete/3	Delete all matching members from a list
flatten/2	Transform nested list into flat list
intersection/3	Set intersection
is_set/1	Type check for a set
list_to_set/2	Remove duplicates
member/2	Element is member of a list
nextto/3	Y follows X in List
nth0/3	N-th element of a list (0-based)
nth1/3	N-th element of a list (1-based)
numlist/3	Create list of integers in interval
permutation/2	Test/generate permutations of a list
reverse/2	Inverse the order of the elements in a list
select/3	Select element of a list
subset/2	Check subset relation for unordered sets
subtract/3	Delete elements that do not satisfy condition
sumlist/2	Add all numbers in a list
union/3	Union of two sets

D.4.3 `ordsets`

ord_empty/1	Test empty ordered set
list_to_ord_set/2	Create ordered set
ord_add_element/3	Add element to ordered set
ord_del_element/3	Delete element from ordered set
ord_intersect/2	Test non-empty intersection
ord_intersection/3	Compute intersection
ord_disjoint/2	Test empty intersection
ord_subtract/3	Delete set from set
ord_union/3	Union of two ordered sets
ord_union/4	Union and difference of two ordered sets
ord_subset/2	Test subset
ord_memberchk/2	Deterministically test membership

D.4.4 ugraphs

vertices_edges_to_ugraph/3	Create unweighted graph
vertices/2	Find vertices in graph
edges/2	Find edges in graph
add_vertices/3	Add vertices to graph
del_vertices/3	Delete vertices from graph
add_edges/3	Add edges to graph
del_edges/3	Delete edges from graph
transpose/2	Invert the direction of all edges
neighbors/3	Find neighbors of vertice
neighbours/3	Find neighbors of vertice
complement/2	Inverse presense of edges
compose/3	
top_sort/2	Sort graph topologically
top_sort/3	Sort graph topologically
transitive_closure/2	Create transitive closure of graph
reachable/3	Find all reachable vertices
ugraph_union/3	Union of two graphs

D.4.5 www_browser

www_open_url/1	Open a web-page in a browser

D.4.6 readutil

read_line_to_codes/2	Read line from a stream
read_line_to_codes/3	Read line from a stream
read_stream_to_codes/2	Read contents of stream
read_stream_to_codes/3	Read contents of stream
read_file_to_codes/3	Read contents of file
read_file_to_terms/3	Read contents of file to Prolog terms

D.4.7 registry

This library is only available on Windows systems.

registry_get_key/2	Get principal value of key
registry_get_key/3	Get associated value of key
registry_set_key/2	Set principal value of key
registry_set_key/3	Set associated value of key
registry_delete_key/1	Remove a key
shell_register_file_type/4	Register a file-type
shell_register_dde/6	Register DDE action
shell_register_prolog/1	Register Prolog

D.4.8 `url`

parse_url/2	Analyse or construct a URL
parse_url/3	Analyse or construct a relative URL
global_url/3	Make relative URL global
http_location/2	Analyse or construct location
www_form_encode/2	Encode or decode form-data

D.4.9 `clp/bounds`

in/2	Define interval for variable
#>/2	Greater than constraint
#</2	Less than constraint
#>=/2	Greater or equal constraint
#=</2	Less of equal constraint
#\=/2	Non-equal constraint
#=/2	Equality constraint
#<=>/2	Constraint equivalence
#<=/2	Constraint implication to the left
#=>/2	Constraint implication to the right
all_different/1	Constraint all values to be unique
indomain/1	Enumerate values from domain
label/1	Solve constraints for variables
lex_chain/1	Constraint on lexicographic ordering
sum/3	Constraint sum of variables
tuples_in/2	Symbolic constraints on tuples

D.4.10 `clp/clp_distinct`

all_distinct/1	Demand distinct values
vars_in/2	Declare domain of variable as set
vars_in/3	Declare domain of variable as interval

D.4.11 `clp/simplex`

assignment/2	Solve assignment problem
constraint/3	Add linear constraint to state
constraint/4	Add named linear constraint to state
gen_state/1	Create empty linear program
maximize/3	Maximize objective function in to linear constraints
minimize/3	Minimize objective function in to linear constraints
objective/2	Fetch value of objective function
shadow_price/3	Fetch shadow price in solved state
transportation/4	Solve transportation problem
variable_value/3	Fetch value of variable in solved state

D.4.12 clpqr

entailed/1	Check if constraint is entailed
inf/2	Find the infimum of an expression
sup/2	Find the supremum of an expression
minimize/1	Minimizes an expression
maximize/1	Maximizes an expression
bb_inf/3	Infimum of expression for mixed-integer problems
bb_inf/4	Infimum of expression for mixed-integer problems
bb_inf/5	Infimum of expression for mixed-integer problems
dump/3	Dump constraints on variables

D.4.13 prologxref

prolog:called_by/2	(hook) Extend cross-referencer
xref_built_in/1	Examine defined built-ins
xref_called/3	Examine called predicates
xref_clean/1	Remove analysis of source
xref_current_source/1	Examine cross-referenced sources
xref_defined/3	Examine defined predicates
xref_exported/2	Examine exported predicates
xref_module/2	Module defined by source
xref_source/1	Cross-reference analysis of source

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.

[Butenhof, 1997] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Reading, MA, USA, 1997.

[Clocksin & Melish, 1987] W. F. Clocksin and C. S. Melish. *Programming in Prolog*. Springer-Verlag, New York, Third, Revised and Extended edition, 1987.

[Demoen, 2002] Bart Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, oct 2002. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.

[Deransart *et al.*, 1996] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.

[Frühwirth,] T. Frühwirth. Thom Fruehwirth's constraint handling rules website. http://www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/chr-intro.html.

[Frühwirth, 1998] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.

[Graham *et al.*, 1982] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[Hodgson, 1998] Jonathan Hodgson. validation suite for conformance with part 1 of the standard, 1998, `http://www.sju.edu/~jhodgson/pub/suite.tar.gz`.

[Holzbaur, 1990] Christian Holzbaur. Realization of forward checking in logic programming through extended unification. Report TR-90-11, Oesterreichisches Forschungsinstitut fuer Artificial Intelligence, Wien, Austria, 1990.

[Kernighan & Ritchie, 1978] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[O'Keefe, 1990] R. A. O'Keefe. *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.