~

~

SWI Prolog

Reference Manual

Updated for version 8.4.2, February 2022

SWI-Prolog developers
https://www.swi-prolog.org

SWI-Prolog is a comprehensive and portable implementation of the Prolog programming language. SWI-Prolog aims to be a robust and scalable implementation supporting a wide range of applications. In particular, it ships with a wide range of interface libraries, providing interfaces to other languages, databases, graphics and networking. It provides extensive support for managing HTML/SGML/XML, JSON, YAML and RDF documents. The system is particularly suited for server applications due to robust support for multithreading and HTTP server libraries.

SWI-Prolog extends Prolog with *tabling* (SGL resolution). Tabling provides better termination properties and avoids repetitive recomputation. Following XSB, SWI-Prolog's tabling supports sound negation using the *Well Founded Semantics*. *Incremental tabling* supports usage as a *Deductive database*.

SWI-Prolog is designed in the 'Edinburgh tradition'. In addition to the ISO Prolog standard it is largely compatible to Quintus, SICStus and YAP Prolog. SWI-Prolog provides a compatibility framework developed in cooperation with YAP and instantiated for YAP, SICStus, IF/Prolog and XSB.

SWI-Prolog aims at providing a rich development environment, including extensive editor support, graphical source-level debugger, autoloading, a 'make' facility to reload edited files and much more. GNU-Emacs, SWI-Prolog editor for Windows, the PDT plugin for Eclipse or a Visual Studio Code plugin provide alternative environments. SWISH provides a web based environment.

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

~

Contents

1 Introduction

This document is a *reference manual*. That means that it documents the system, but it does not explain the basics of the Prolog language and it leaves many details of the syntax, semantics and built-in primitives undefined where SWI-Prolog follows the standards. This manual is intended for people that are familiar with Prolog. For those not familiar with Prolog, we recommend to start with a Prolog textbook such as [Bratko, 1986], [Sterling & Shapiro, 1986] or [Clocksin & Melish, 1987]. For more advanced Prolog usage we recommend [O'Keefe, 1990].

1.1 Positioning SWI-Prolog

Most implementations of the Prolog language are designed to serve a limited set of use cases. SWI-Prolog is no exception to this rule. SWI-Prolog positions itself primarily as a Prolog environment for 'programming in the large' and use cases where it plays a central role in an application, i.e., where it acts as 'glue' between components. At the same time, SWI-Prolog aims at providing a productive rapid prototyping environment. Its orientation towards programming in the large is backed up by scalability, compiler speed, program structuring (modules), support for multithreading to accommodate servers, Unicode and interfaces to a large number of document formats, protocols and programming languages. Prototyping is facilitated by good development tools, both for command line usage and for usage with graphical development tools. Demand loading of predicates from the library and a 'make' facility avoids the *requirement* for using declarations and reduces typing.

SWI-Prolog is traditionally strong in education because it is free and portable, but also because of its compatibility with textbooks and its easy-to-use environment.

Note that these positions do not imply that the system cannot be used with other scenarios. SWI-Prolog is used as an embedded language where it serves as a small rule subsystem in a large application. It is also used as a deductive database. In some cases, this is the right choice because SWI-Prolog has features that are required in the application, such as threading or Unicode support. In general though, for example: GNU-Prolog is more suited for embedding because it is small and can compile to native code; XSB is better for deductive databases because it provides a mature implementation of *tabling* including support for incremental updates and *Well Founded Semantics*[1]; and ECLiPSe is better at constraint handling.

The syntax and set of built-in predicates is based on the ISO standard [Hodgson, 1998]. Most extensions follow the 'Edinburgh tradition' (DEC10 Prolog and C-Prolog) and Quintus Prolog [Qui, 1997]. The infrastructure for constraint programming is based on hProlog [Demoen, 2002]. Some libraries are copied from the YAP[2] system. Together with YAP, we developed a portability framework (see section C). This framework has been filled for SICStus Prolog, YAP, IF/Prolog and

[1] Sponsored by Kyndi and with help from the XSB developers Theresa Swift and David S. Warren, SWI-Prolog now supports many of the XSB features.

[2] http://www.dcc.fc.up.pt/~{}vsc/Yap/

Ciao. SWI-Prolog version 7 introduces various extensions to the Prolog language (see section 5). The *string* data type and its supporting set of built-in predicates is compatible with ECLiPSe.

1.2 Status and releases

This manual describes version 8.4 of SWI-Prolog. SWI-Prolog is widely considered to be a robust and scalable implementation of the Prolog language. It is widely used in education and research. In addition, it is in use for 24×7 mission critical commercial server processes. The site `http://www.swi-prolog.org` is hosted using the SWI-Prolog HTTP server infrastructure. It receives approximately 2.3 million hits and serves approximately 300 Gbytes on manual data and downloads each month. SWI-Prolog applications range from student assignments to commercial applications that count more than one million lines of Prolog code.

SWI-Prolog has two development tracks. *Stable* releases have an even *minor* version number (e.g., 6.2.1) and are released as a branch from the development version when the development version is considered stable and there is sufficient new functionality to justify a stable release. Stable releases often get a few patch updates to deal with installation issues or major flaws. A new *Development* version is typically released every couple of weeks as a snapshot of the public git repository. 'Extra editions' of the development version may be released after problems that severely hindered the user in their progress have been fixed.

Known bugs that are not likely to be fixed soon are described as footnotes in this manual.

1.3 Should I be using SWI-Prolog?

There are a number of reasons why it might be better to choose a commercial, or another free, Prolog system:

- *SWI-Prolog comes with no warranties*
  Although the developers or the community often provide a work-around or a fix for a bug, there is no place you can go to for guaranteed support. However, the full source archive is available and can be used to compile and debug SWI-Prolog using free tools on all major platforms. Users requiring more support should ensure access to knowledgeable developers.

- *Performance is your first concern*
  Various free and commercial systems have better performance. But, 'standard' Prolog benchmarks disregard many factors that are often critical to the performance of large applications. SWI-Prolog is not good at fast calling of simple predicates, but it is fast with dynamic code, meta-calling and predicates that contain large numbers of clauses or require more advanced clauses indexing. Many of SWI-Prolog's built-in predicates are written in C and have excellent performance.

On the other hand, SWI-Prolog offers some facilities that are widely appreciated by users:

- *Comprehensive support of Prolog extensions*
  Many modern Prolog implementations extend the standard SLD resolution mechanism with which Prolog started and that is described in the ISO standard. SWI-Prolog offers most popular extensions.

Attributed variables provide *Constraint Logic Programming* and delayed execution based on instantiation (*coroutining*). *Tabling* or *SGL resolution* provides characteristics normally associated with *bottom up evaluation*: better termination, better predictable performance by avoiding recomputation and Well Founded Semantics for negation. *Delimited continuations* can be used to implement high level new control structures and *Engines* can be used to control multiple Prolog goals, achieving different control structures such as massive numbers of cooperating agents.

- *Nice environment*
  SWI-Prolog provides a good command line environment, including 'Do What I Mean', auto-completion, history and a tracer that operates on single key strokes. The system automatically recompiles modified parts of the source code using the `make/0` command. The system can be instructed to open an arbitrary editor on the right file and line based on its source database. It ships with various graphical tools and can be combined with the SWI-Prolog editor, PDT (Eclipse plugin for Prolog), VScode or GNU-Emacs.

- *Fast compiler*
  Even very large applications can be loaded in seconds on most machines. If this is not enough, there is the Quick Load Format. See `qcompile/1` and `qsave_program/2`.

- *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, and the performance of debugged code is close to that of normal operation.

- *Source level debugger*
  The source level debugger provides a good overview of your current location in the search tree, variable bindings, your source code and open choice points. Choice point inspection provides meaningful insight to both novices and experienced users. Avoiding unintended choice points often provides a huge increase in performance and a huge saving in memory usage.

- *Profiling*
  SWI-Prolog offers an execution profiler with either textual output or graphical output. Finding and improving hotspots in a Prolog program may result in huge speedups.

- *Flexibility*
  SWI-Prolog can easily be integrated with C, supporting non-determinism in Prolog calling C as well as C calling Prolog (see section 12). It can also be *embedded* in external programs (see section 12.5). System predicates can be redefined locally to provide compatibility with other Prolog systems.

- *Threads*
  Robust support for multiple threads may improve performance and is a key enabling factor for deploying Prolog in server applications. Threads also facilitates debugging and maintenance of long running processes and embedded Prolog engines. The native IDE tools run in a separate thread The `prolog_server` library provides `telnet` access and the pack `libssh` provides SSH login. With some restrictions regarding the compatibility of old and new code, code can be replaced while it is being executed in another thread. This allows for injecting `debug/3` statements as well as fixing bugs without downtime.

- *Interfaces*
  SWI-Prolog ships with many extension packages that provide robust interfaces to processes, encryption, TCP/IP, TIPC, ODBC, SGML/XML/HTML, RDF, JSON, YAML, HTTP, graphics and much more.

1.4 Support the SWI-Prolog project

You can support the SWI-Prolog project in several ways. Academics are invited to cite one of the publications[3] on SWI-Prolog. Users can help by identifying and/or fixing problems with the code or its documentation[4]. Users can contribute new features or, more lightweight, contribute packs[5]. Commercial users may consider contacting the developers[6] to sponsor the development of new features or seek for opportunities to cooperate with the developers or other commercial users.

1.5 Implementation history

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. In those days, Prolog systems were not very aware of their 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 focusing 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 simple Prolog virtual machine called ZIP [Bowen *et al.*, 1983, Neumerkel, 1993] 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 virtual machine, 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 in debug mode compared to normal execution. The price we have to pay is some performance degradation (taking out the debugger from the VM 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 *et al.*, 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`).

SWI-Prolog implements *attributed variables* (constraints) and *delimited continuations* following the design in hProlog by Bart Demoen. The *engine* implementation follows the design proposed by Paul Tarau. Tabling was implemented by Benoit Desouter based on delimited continuations. Tabling has been extended with *answer subsumption* by Fabrizio Riguzzi. The implementation of *well founded semantics* and *incremental tabling* follows XSB and has been sponsored by Kyndi and mode possible by technical support from notably Theresa Swift and David S. Warren.

[3] https://www.swi-prolog.org/Publications.html
[4] https://www.swi-prolog.org/howto/SubmitPatch.html
[5] https://www.swi-prolog.org/pack/list
[6] mailto:info@swi-prolog.org

1.6 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.

Our special thanks go to those who had the fate 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! Randy Sharp fixed many issues in the 6.0.x version of the manual.

Bart Demoen and Tom Schrijvers have helped me adding coroutining, constraints, global variables and support for cyclic terms to the kernel. Tom Schrijvers has provided a first clp(fd) constraint solver, the CHR compiler and some of the coroutining predicates. Markus Triska contributed the current clp(fd) implementation as well as the clp(b) implementation.

Tom Schrijvers and Bart Demoen initiated the implementation of *delimited continuations* (section 4.9), which was used by Benoit Desouter and Tom Schrijvers to implement *tabling* (section 7) as a library. Fabrizio Riguzzi added a first implementation for *mode directed tabling* (section 7.3).

The SWI-Prolog 7 extensions (section 5) are the result of a long heated discussion on the mailinglist. Nicos Angelopoulos' wish for a smooth integration with the R language triggered the overall intend of these extensions to enable a smoother integration of Prolog with other languages. Michael Hendrix suggested and helped shaping SWI-Prolog *quasi quotations*.

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 of the SSL library, unbounded integer and rational number arithmetic and many enhancements to the memory management of the system.

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

Jeff Rosenwald contributed the TIPC networking library and Google's protocol buffer handling.

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.

Kyndi (`https://kyndi.com/`) sponsored the development of the *engines* interface (chapter 11). The final API was established after discussion with the founding father of engines, Paul Tarau and Paulo Moura. Kyndi also sponsored JIT indexing on multiple arguments as well as *deep indexing*. Kyndi currently supports the implementation of XSB compatible tabling, including well founded semantics and incremental tabling. Theresa Swift, David S. Warren and Fabrizio Riguzzi provided input to realise advanced tabling.

Overview 2

2.1 Getting started quickly

2.1.1 Starting SWI-Prolog

Starting SWI-Prolog on Unix

By default, SWI-Prolog is installed as 'swipl'. 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:

```
$ swipl
Welcome to SWI-Prolog ...
...

1 ?-
```

After starting Prolog, one normally loads a program into it using `consult/1`, which 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].
true.

?-
```

Alternatively, the source file may be given as command line arguments:

```
$ swipl likes.pl
Welcome to SWI-Prolog ...
...

1 ?-
```

> Both the above assume `likes.pl` is in your *working directory*. If you use the command line version `swipl` the working directory is the same as the shell from which you started SWI-Prolog. If you started the GUI version (`swipl-win`) this depends largely on the OS. You can use `pwd/0` and `cd/0` to find and change the working directory. The utility `ls/0` lists the contents of the working directory.

```
?- pwd.
% /home/janw/src/swipl-devel/linux/
true.
?- cd('~/tmp').
true.

?- pwd.
% /home/janw/tmp/
true.
```

The file `likes.pl` is also installed in a subdirectory `demo` insides SWI-Prolog's installation directory and may be loaded regardless of the working directory using the command below. See `absolute_file_name/3` and `file_search_path/2` for details on how SWI-Prolog specifies file locations.

```
?- [swi(demo/likes)].
true.
```

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 `swipl` containing the executables, libraries, etc., of the system. No files are installed outside this directory.
- A program `swipl-win.exe`, providing a window for interaction with Prolog. The program `swipl.exe` is a version of SWI-Prolog that runs in a console window.
- The file extension `.pl` is associated with the program `swipl-win.exe`. Opening a `.pl` file will cause `swipl-win.exe` to start, change directory to the directory in which the file to open resides, and load this file.

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

2.1.2 Adding rules from the console

Although we strongly advice to put your program in a file, optionally edit it and use `make/0` to reload it (see section 2.1.4), it is possible to manage facts and rules from the terminal. The most convenient way to add a few clauses is by consulting the pseudo file `user`. The input is ended using the system end-of-file character.

```
?- [user].
|: hello :- format('Hello world~n').
|: ^D
true.

?- hello.
Hello world
true.
```

The predicates `assertz/1` and `retract/1` are alternatives to add and remove rules and facts.

2.1.3 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 (;) or spacebar[1] if (s)he wants another solution. Use the RETURN key if you do not want to see more answers. Prolog completes the output with a full stop (.) if the user uses the RETURN key or Prolog *knows* there are no more answers. If Prolog cannot find (more) answers, it writes **false.** Finally, Prolog answers using an error message to indicate the query or program contains an error.

```
?- likes(sam, X).
X = dahl ;
X = tandoori ;
...
X = chips.

?-
```

Note that the answer written by Prolog is a valid Prolog program that, when executed, produces the same set of answers as the original program.[2]

2.1.4 Examining and modifying your program

If properly configured, the predicate `edit/1` starts the built-in or user configured editor on the argument. The argument can be anything that can be linked to a location: a file name, predicate name, module name, etc. If the argument resolves to only one location the editor is started on this location, otherwise the user is presented a choice.

If a graphical user interface is available, the editor normally creates a new window and the system prompts for the next command. The user may edit the source file, save it and run `make/0` to update any modified source file. If the editor cannot be opened in a window, it opens in the same console and leaving the editor runs `make/0` to reload any source files that have been modified.

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

[2] The SWI-Prolog top level differs in several ways from traditional Prolog top level. The current top level was designed in cooperation with Ulrich Neumerkel.

```
?- edit(likes).

true.
?- make.
% /home/jan/src/pl-devel/linux/likes compiled 0.00 sec, 0 clauses

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

The program can also be *decompiled* using `listing/1` as below. The argument of `listing/1` is just a predicate name, a predicate *indicator* of the form *Name/Arity*, e.g., `?- listing(mild/1).` or a *head*, e.g., `?- listing(likes(sam, _)).`, listing all *matching* clauses. The predicate `listing/0`, i.e., without arguments lists the entire program.[3]

```
?- listing(mild).
mild(dahl).
mild(tandoori).
mild(kurma).

true.
```

2.1.5 Stopping Prolog

The interactive toplevel can be stopped in two ways: enter the system end-of-file character (typically *Control-D*) or by executing the `halt/0` predicate:

```
?- halt.
$
```

2.2 The user's initialisation file

After the system initialisation, the system consults (see `consult/1`) the user's *init* file. This file is searched using `absolute_file_name/3` using the path alias (see `file_search_path/2`) `app_config`. This is a directory named `swi-prolog` below the OS default name for placing application configuration data:

- On Windows, the CSIDL folder `CSIDL_APPDATA`, typically `C:\Documents and Settings\username\Application Data`.
- If the environment variable `XDG_DATA_HOME` is set, use this. This follows the free desktop standard.

[3]This lists several *hook* predicates that are defined by default and is typically not very informative.

- The expansion of `~/.config`.

The directory can be found using this call:

```
?- absolute_file_name(app_config(.), Dir, [file_type(directory)]).
Dir = '/home/jan/.config/swi-prolog'.
```

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.

The installation provides a file `customize/init.pl` with (commented) commands that are often used to customize the behaviour of Prolog, such as interfacing to the editor, color selection or history parameters. Many of the development tools provide menu entries for editing the startup file and starting a fresh startup file from the system skeleton.

See also the `-s` (script) and `-F` (system-wide initialisation) in section 2.4 and section 2.3.

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 initialisation goals 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% swipl -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 command line may have multiple `-g goal` occurrences. The goals are executed in order. Possible choice points of individual goals are pruned. If a *goal* fails execution stops with exit status `1`. If a *goal* raises an exception, the exception is printed and the process stops with exit code `2`.

The `-q` may be used to suppress all informational messages as well as the error message that is normally printed if an initialisation goal *fails*.

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 2.11.2 discusses further scripting options, and chapter 13 discusses the generation of runtime executables. Runtime executables are a means to deliver executables that do not require the Prolog system.

2.4 Command line options

SWI-Prolog can be executed in one of the following modes:

`swipl --help`
`swipl --version`
`swipl --arch`
`swipl --dump-runtime-variables`
: These options must appear as only option. They cause Prolog to print an informational message and exit. See section 2.4.1.

`swipl` **[*option* ...]** *script-file* **[*arg* ...]**
: These arguments are passed on Unix systems if file that starts with `#!/path/to/executable` [*option* ...] is executed. Arguments after the script file are made available in the Prolog flag `argv`.

`swipl` **[*option* ...]** *prolog-file* **... [[`--`]** *arg* **...]**
: This is the normal way to start Prolog. The options are described in section 2.4.2, section 2.4.3 and section 2.4.4. The Prolog flag `argv` provides access to *arg* ... If the *options* are followed by one or more Prolog file names (i.e., names with extension `.pl`, `.prolog` or (on Windows) the user preferred extension registered during installation), these files are loaded. The first file is registered in the Prolog flag `associated_file`. In addition, `pl-win[.exe]` switches to the directory in which this primary source file is located using `working_directory/2`.

`swipl` **-o** *output* **-c** *prolog-file* **...**
: The `-c` option is used to compile a set of Prolog files into an executable. See section 2.4.5.

`swipl` **-o** *output* **-b** *bootfile prolog-file* **...**
: Bootstrap compilation. See section 2.4.6.

2.4.1 Informational command line options

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

`--dump-runtime-variables` *[=format]*
: 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 `swipl-ld` (see section 12.5). Below is a typical example of using this feature.

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

The option can be followed by `=sh` to dump in POSIX shell format (default) or `=cmd` to dump in MS-Windows `cmd.exe` compatible format.

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

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

--abi-version
Print a key (string) that represents the binary compatibility on a number of aspects. See section 2.22.

2.4.2 Command line options for running Prolog

Note that *boolean options* may be written as `--name` (true), `--noname` or `--no-name` (false). They are written as `--no-name` below as the default is 'true'.

--debug-on-interrupt
Enable debugging on an interrupt signal (Control-C, `SIGINT`) immediately. Normally debugging on interrupt is enabled when entering the interactive toplevel. This flag can be used to start the debugger on an interrupt while executing goals from `-g` or `initialization/[1,2]`. See also the Prolog flag `debug_on_interrupt`.

--home=DIR
Use *DIR* as home directory. See section 12.6 for details.

--quiet
Set the Prolog flag `verbose` to `silent`, suppressing informational and banner messages. Also available as `-q`.

--no-debug
Disable debugging. See the `current_prolog_flag/2` flag `generate_debug_info` for details.

--no-signals
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 12.4.23 for details. Note that the handler to unblock system calls is still installed. This can be prevented using `--sigalert=0` additionally. See `--sigalert`.

--no-threads
Disable threading for the multi-threaded version at runtime. See also the flags `threads` and `gc_thread`.

--no-packs
Do *not* attach extension packages (add-ons). See also `attach_packs/0` and the Prolog flag `packs`.

--no-pce
Enable/disable the xpce GUI subsystem. The default is to make it available as autoload component if it is installed and the system can access the graphics. Using `--pce` load the xpce system in user space and `--no-pce` makes it unavailable in the session.

--on-error =*style*
How to handle on errors. See the Prolog flag `on_error` for details.

--on-warning =*style*
: How to handle on warnings. See the Prolog flag `on_warning` for details.

--pldoc *[=port]*
: Start the PlDoc documentation system on a free network port and launch the user's browser on `http://localhost:`*port*. If *port* is specified, the server is started at the given port and the browser is *not* launched.

--sigalert=NUM
: Use signal *NUM* (1...31) for alerting a thread. This is needed to make `thread_signal/2`, and derived Prolog signal handling act immediately when the target thread is blocked on an interruptible system call (e.g., `sleep/1`, read/write to most devices). The default is to use `SIGUSR2`. If *NUM* is 0 (zero), this handler is not installed. See `prolog_alert_signal/2` to query or modify this value at runtime.

--no-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. See also `tty_control`.

--win-app
: This option is available only in `swipl-win.exe` and is used for the start-menu item. If causes `plwin` to start in the folder `...\My Documents\Prolog` or local equivalent thereof (see `win_folder/2`). The `Prolog` subdirectory is created if it does not exist.

-O
: Optimised compilation. See `current_prolog_flag/2` flag `optimise` for details.

-l *file*
: Load *file*. This flag provides compatibility with some other Prolog systems.[4] It is used in SWI-Prolog to skip the program initialization specified using `initialization/2` directives. See also section 2.11.2, and `initialize/0`.

-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 `init.pl`. '`-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*
: Select 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`*

[4] YAP, SICStus

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`.

-x *bootfile*
Boot from *bootfile* instead of the system's default boot file. A boot file 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, and argpath1 ... 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.

--traditional
This flag disables the most important extensions of SWI-Prolog version 7 (see section 5) that introduce incompatibilities with earlier versions. In particular, lists are represented in the traditional way, double quoted text is represented by a list of character codes and the functional notation on dicts is not supported. Dicts as a syntactic entity, and the predicates that act on them, are still supported if this flag is present.

--
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.

2.4.3 Controlling the stack sizes

As of version 7.7.14 the stacks are no longer limited individually. Instead, only the combined size is limited. Note that 32 bit systems still pose a 128Mb limit. See section 2.20.1. The combined limit is by default 1Gb on 64 bit machines and 512Mb on 32 bit machines.

For example, to limit the stacks to 32Gb use the command below. Note that the stack limits apply *per thread*. Individual threads may be controlled using the `stack_limit(+Bytes)` option of thread_create. Any thread can call `set_prolog_flag(stack_limit, Limit)` (see `stack_limit`) to adjust the stack limit. This limit is inherited by threads created from this thread.

```
$ swipl --stack-limit=32g
```

--stack-limit=*size[bkmg]*
Limit the combined size of the Prolog stacks to the indicated *size*. The suffix specifies the value as *bytes*, *Kbytes*, *Mbytes* or *Gbytes*.

--table-space=*size[bkmg]*
Limit for the *table space*. This is where tries holding memoized[5] answers for *tabling* are stored. The default is 1Gb on 64 bit machines and 512Mb on 32 bit machines. See the Prolog flag `table_space`.

[5] The letter M is used because the T was already in use. It is a memnonic for **M**emoizing.

--shared-table-space=*size[bkmg]*
: Limit for the table space for *shared* tables. See section 7.9.

2.4.4 Running goals from the command line

-g *goal*
: *Goal* is executed just before entering the top level. This option may appear multiple times. See section 2.3 for details. If no initialization goal is present the system calls `version/0` to print the welcome message. The welcome message can be suppressed with `--quiet`, but also with `-g true`. *goal* can be a complex term. In this case quotes are normally needed to protect it from being expanded by the shell. A safe way to run a goal non-interactively is below. If `go/0` succeeds `-g halt` causes the process to stop with exit code 0. If it fails, the exit code is 1; and if it raises an exception, the exit code is 2.

```
% swipl <options> -g go -g halt
```

-t *goal*
: Use *goal* as interactive top level instead of the default goal `prolog/0`. The *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. If the top level raises an exception, this is printed as an uncaught error and the top level is *restarted*. This flag also determines the goal started by `break/0` and `abort/0`. If you want to prevent the user from entering interactive mode, start the application with '`-g` *goal* `-t halt`'.

2.4.5 Compilation options

-c *file* ...
: Compile files into an 'intermediate code file'. See section 2.11.

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

2.4.6 Maintenance options

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 *token1,token2,...*
: Print debug messages for DEBUG statements tagged with one of the indicated tokens. Only has effect if the system is compiled with the `-DO_DEBUG` flag. System maintenance only.

2.5 UI Themes

UI (colour) themes play a role in two parts: when writing to the *console* and for the xpce-based development tools such as PceEmacs or the graphical debugger. Coloured console output is based on `ansi_format/3`. The central message infra structure based on `print_message/2` labels message (components) with a Prolog term that specifies the role. This is mapped to concrete colours by means of the hook `prolog:console_color/2`. Theming the IDE uses xpce *class variables* that are initialised from Prolog when xpce is loaded.

Themes are implemented as a Prolog file in the file search path library/theme. A theme can be loaded using (for example) the directive below in the user's initialization file (see section 2.2).

```
:- use_module(library(theme/dark)).
```

The theme file `library(theme/auto)` is provided to automatically choose a reasonable theme based on the environment. The current version detects the background color on *xterm* compatible terminal emulators (found on most Unix systems) and loads the `dark` theme if the background is 'darkish'.

The following notes apply to the different platforms on which SWI-Prolog is supported:

Unix/Linux If an xterm compatible terminal emulator is used to run Prolog you may wish to load either an explicit theme or `library(theme/auto)`.

Windows The `swipl-win.exe` graphical application can be themed by loading a theme file. The theme file also sets the foreground and background colours for the console.

2.5.1 Status of theme support

Theme support was added in SWI-Prolog 8.1.11. Only part of the IDE tools are covered and the only additional theme (`dark`) is not net well balanced. The interfaces between the theme file and notably the IDE components is not very well established. Please contribute by improving the `dark` theme. Once that is complete and properly functioning we can start adding new themes.

2.6 GNU Emacs Interface

Unfortunately the default Prolog mode of GNU Emacs is not very good. There are several alternatives though:

- `https://bruda.ca/emacs/prolog_mode_for_emacs`
  Prolog mode for Emacs and XEmacs maintained by Stefan Bruda.
- `https://www.metalevel.at/pceprolog/`
  Recommended configuration options for editing Prolog code with Emacs.
- `https://www.metalevel.at/ediprolog/`
  Interact with SWI-Prolog directly in Emacs buffers.
- `https://www.metalevel.at/etrace/`
  Trace Prolog code with Emacs.

2.7 Online Help

2.7.1 library(help): Text based manual

This module provides `help/1` and `apropos/1` that give help on a topic or searches the manual for relevant topics.

By default the result of `help/1` is sent through a *pager* such as `less`. This behaviour is controlled by the following:

- The Prolog flag `help_pager`, which can be set to one of the following values:

  false
  Never use a pager.

  default
  Use default behaviour. This tries to determine whether Prolog is running interactively in an environment that allows for a pager. If so it examines the environment variable `PAGER` or otherwise tries to find the `less` program.

  Callable
  A *Callable* term is interpreted as `program_name(Arg, ...)`. For example, `less('-r')` would be the default. Note that the program name can be an absolute path if single quotes are used.

help *[det]*
help(+*What*) *[det]*
Show help for *What*. *What* is a term that describes the `topics(s)` to give help for. Notations for *What* are:

Atom
This ambiguous form is most commonly used and shows all matching documents. For example:

```
?- help(append).
```

Name **/** *Arity*
Give help on predicates with matching *Name/Arity*. *Arity* may be unbound.

Name **//** *Arity*
Give help on the matching DCG rule (non-terminal)

f(*Name/Arity*)
Give help on the matching Prolog arithmetic functions.

c(*Name*)
Give help on the matching C interface function

section(*Label*)
Show the section from the manual with matching *Label*.

If an exact match fails this predicates attempts fuzzy matching and, when successful, display the results headed by a warning that the matches are based on fuzzy matching.

If possible, the results are sent through a *pager* such as the `less` program. This behaviour is controlled by the Prolog flag `help_pager`. See section level documentation.

See also `apropos/1` for searching the manual names and summaries.

show_html_hook(+*HTML:string*) *[semidet,multifile]*
Hook called to display the extracted *HTML* document. If this hook fails the *HTML* is rendered to the console as plain text using `html_text/2`.

apropos(+*Query*) *[det]*
Print objects from the manual whose name or summary match with *Query*. *Query* takes one of the following forms:

Type : *Text*
Find objects matching *Text* and filter the results by *Type*. *Type* matching is a case intensitive *prefix* match. Defined types are `section`, `cfunction`, `function`, `iso_predicate`, `swi_builtin_predicate`, `library_predicate`, `dcg` and aliases `chapter`, `arithmetic`, `c_function`, `predicate`, `nonterminal` and `non_terminal`. For example:

```
?- apropos(c:close).
?- apropos(f:min).
```

Text
Text is broken into tokens. A topic matches if all tokens appear in the name or summary of the topic. Matching is case insensitive. Results are ordered depending on the quality of the match.

2.7.2 library(explain): Describe Prolog Terms

The `library(explain)` describes prolog-terms. The most useful functionality is its cross-referencing function.

```
?- explain(subset(_,_)).
"subset(_, _)" is a compound term
    from 2-th clause of lists:subset/2
    Referenced from 46-th clause of prolog_xref:imported/3
    Referenced from 68-th clause of prolog_xref:imported/3
lists:subset/2 is a predicate defined in
    /staff/jan/lib/pl-5.6.17/library/lists.pl:307
    Referenced from 2-th clause of lists:subset/2
    Possibly referenced from 2-th clause of lists:subset/2
```

Note that PceEmacs can jump to definitions and `gxref/0` can be used for an overview of dependencies.

`!!.`	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

explain(@*Term*) *[det]*
Give an explanation on *Term*. 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/1` describes the predicate as well as possible references to it. See also `gxref/0`.

explain(@*Term, -Explanation*) *[nondet]*
True when *Explanation* is an explanation of *Term*.

2.8 Command line history

SWI-Prolog offers a query substitution mechanism similar to what is seen in Unix shells. The availability of this feature is controlled by `set_prolog_flag/2`, using the `history` Prolog flag. By default, history is available if no interactive command line editor is available. To enable history, 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.

2.9 Reuse of top-level bindings

Bindings resulting from the successful execution of a top-level goal are asserted in a database *if they are not too large*. These values may be reused in further top-level queries as $Var. If the same variable name is used in a subsequent query the system associates the variable with the latest binding. Example:

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

```
6 ?- X = statistics.
X = statistics.

7 ?- $X.
% Started at Fri Aug 24 16:42:53 2018
% 0.118 seconds cpu time for 456,902 inferences
% 7,574 atoms, 4,058 functors, 2,912 predicates, 56 modules, 109,791 VM-codes
%
```

```
1 ?- maplist(plus(1), `hello`, X).
X = [105,102,109,109,112].

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

3 ?-
```

Figure 2.1: Reusing top-level bindings

```
%                       Limit    Allocated      In use
% Local  stack:             -        20 Kb    1,888  b
% Global stack:             -        60 Kb       36 Kb
% Trail  stack:             -        30 Kb    4,112  b
%        Total:    1,024 Mb      110 Kb       42 Kb
%
% 3 garbage collections gained 178,400 bytes in 0.000 seconds.
% 2 clause garbage collections gained 134 clauses in 0.000 seconds.
% Stack shifts: 2 local, 2 global, 2 trail in 0.000 seconds
% 2 threads, 0 finished threads used 0.000 seconds
true.
```

2.10 Overview of the Debugger

Imperative languages like C++, Python or JavaScript execute mostly linear code with some branching and subroutine calls. Their debuggers support stepping through the code and pausing on each line, or running the program until it hits a breakpoint and pauses. When paused, the user can inspect the current program state or give the debugger commands.

Prolog has a logical execution model that involves attempting to prove logical predicates and needs a different debugging approach. SWI-Prolog uses the traditional Prolog "Byrd Box Model" or "4 Port Model" debugging approach described by [Byrd, 1980, Clocksin & Melish, 1987] with a couple of extensions to implement its command line debugger. There are two other debuggers available that build on this infrastructure: a graphical debugger and remote debugging in the web interface provided by SWISH.

Reference information to all predicates available for manipulating the debugger is in the debugger section (section 4.39).

2.10.1 The Byrd Box Model And Ports

Standard Prolog debugging tools are built around the so-called "Byrd Box Model" or "4 Port Model" which models each predicate in a Prolog program as a state machine ("box") that transitions through states ("ports") as a program is evaluated. The developer can ask the engine to pause for program inspection when it reaches specific ports or predicates.

As we go through this overview, remember that a ”port” is just another word for a ”state” in the state machine that each predicate transitions through during evaluation. The state machine is called a ”box” because it is drawn like this:

```
             *--------------------------------------*
      Call   |                                      |   Exit
 ---------> +  descendant(X,Y) :- offspring(X,Y).   + --------->
             |                                      |
             |  descendant(X,Z) :-                  |
 <--------- +     offspring(X,Y), descendant(Y,Z). + <---------
      Fail   |                                      |   Redo
             *--------------------------------------*
```

The standard ports are: `call`, `redo`, `exit` and `fail`. SWI-Prolog extends this with two more: `unify` and `exception`. Each trace happens at a particular phase of predicate resolution. Recall that when resolving or ”proving” a predicate, the Prolog engine:

1. Collects all rules that **might** match by having a head with the same name and number of arguments
    - `call` is traced, once, if **any** rules might match.
    - `redo` is also traced when the engine backtracks to find the next matching rule.
2. Finds the next matching rule whose head can be unified with the predicate
    - `unify` is traced with the results of unification if one is found.
    - `fail` is traced if no rule heads can be unified.
3. Applies variable assignments from unification to clauses in the rule body and continues at #1 with the updated clauses
4. After **all** of the body clauses of the matched rule have either succeeded, failed, or thrown an exception:
    - `exit` is traced if all of them succeeded (meaning this rule is true).
    - `fail` is traced if any of them failed (meaning this rule is false).
    - `exception` is traced if any of them threw an exception.

    This means there can be **a lot** of traces between the initial `call` and the end of tracing for a particular predicate.

2.10.2 Trace Mode Example

The `trace/0` predicate turns on ”trace mode”, which, by default, produces a trace and pauses at every port of every predicate to allow inspection of the state of the program. This is normally done from the Prolog console window, but for embedded Prolog systems or when Prolog runs as a daemon it can also be done by getting a prompt via the libssh package.

> Note: If the native graphics plugin (XPCE) is available, the commands `gtrace/0` and `gspy/1` activate the graphical debugger while `tdebug/0` and `tspy/1` allow debugging of arbitrary threads.

Each goal is printed using the Prolog predicate `write_term/2`. The style is defined by the Prolog flag `debugger_write_options` and can be modified using this flag or using the `w`, `p` and `d` commands of the tracer (section 2.10.4).

Here's an example debugging session that shows the basic flow. The `unify` port is off by default since it doesn't add a lot of information in most cases for the command line debugger.

```
is_a(rock1, rock).
is_a(rock2, rock).
color(rock1, red).

noun(X, Type) :- is_a(X, Type).
adjective(X, color, Value) :- color(X, Value).
```

```
?- trace.
true.

[trace]  ?- noun(X, rock), adjective(X, color, red).
   Call: (11) noun(_9774, rock) ? creep
```

The `trace/0` predicate turned on trace mode, which is now indicated at every prompt by `[trace] ?-`. The initial query provided by the user was `noun(X, rock), adjective(X, color, red)` which is asking to find a "red rock". Finally: the first port triggered was a *Call* to the first predicate in the initial query, indicating the engine is about to look for the first rule that matches `noun(_9774, rock)`.

Pressing `spacebar`, `c`, or `enter` caused the tracer to print `creep` followed by the next trace. There are many additional commands available that are described later in the overview.

```
is_a(rock1, rock).
is_a(rock2, rock).
color(rock1, red).

noun(X, Type) :- is_a(X, Type).
adjective(X, color, Value) :- color(X, Value).
```

```
[trace]  ?- noun(X, rock), adjective(X, color, red).
...
   Call: (12) is_a(_9774, rock) ? creep
   Exit: (12) is_a(rock1, rock) ? creep
   Exit: (11) noun(rock1, rock) ? creep
...
```

Next, the first clause of `noun/2` gets a `call` trace since the engine is trying to find the next rule that matches `is_a(_9774, rock)`. Since there **is** a fact that can unify: `is_a(rock1, rock)`, the trace shows `exit` (i.e. succeeded) along with that value. Since that was the final predicate in the body of `noun/2`, `noun/2` also gets an `exit` trace that shows the unified value of its head: `noun(rock1, rock)`.

```
is_a(rock1, rock).
is_a(rock2, rock).
color(rock1, red).

noun(X, Type) :- is_a(X, Type).
adjective(X, color, Value) :- color(X, Value).
```

```
[trace]  ?- noun(X, rock), adjective(X, color, red).
...
   Call: (11) adjective(rock1, color, red) ? creep
   Call: (12) color(rock1, red) ? creep
   Exit: (12) color(rock1, red) ? creep
   Exit: (11) adjective(rock1, color, red) ? creep
   X = rock1 ;
...
```

Prolog then moved to the next predicate in the initial query: `adjective/3` and solved it in a similar way. Since that was the last predicate in the query, an answer was returned. Pressing `;` requested the next answer and began Prolog backtracking.

```
is_a(rock1, rock).
is_a(rock2, rock).
color(rock1, red).

noun(X, Type) :- is_a(X, Type).
adjective(X, color, Value) :- color(X, Value).
```

```
[trace]  ?- noun(X, rock), adjective(X, color, red).
...
   Redo: (12) is_a(_9774, rock) ? creep
   Exit: (12) is_a(rock2, rock) ? creep
   Exit: (11) noun(rock2, rock) ? creep
   Call: (11) adjective(rock2, color, red) ? creep
   Call: (12) color(rock2, red) ? creep
   Fail: (12) color(rock2, red) ? creep
   Fail: (11) adjective(rock2, color, red) ? creep
false.
```

The only choice point to `redo` (i.e. backtrack over) was the `is_a/2` clause of `noun/2` since there was one potential match left to attempt to unify: `is_a(rock2, rock)`. This succeeds with an `exit` trace since it does unify with the `redo` predicate and causes `noun(rock2, rock)` to also succeed with `exit` just as above.

As the traces continue, you can see the `fail` port get activated for `color(rock2, red)` since there is no way to prove that predicate and thus the whole query returns `false`.

Tracing will continue for every query you pose until you enter `notrace.` to turn off trace mode.

2.10.3 Trace Mode Options: `leash/1` and `visible/1`

When you enable trace mode with `trace/0`, the tracer will, by default, pause and wait for a command at every port it hits on every predicate. The `leash/1` predicate can be used to modify the ports to pause at. This is a global setting, so changes will remain until they are changed again or SWI-Prolog is restarted. Disabling the tracer via `notrace/0` doesn't affect which ports are leashed.

The `leash/1` argument must start with + to add, or - to remove, followed by the name of a port such as `call`, `exit`, etc. There are special terms like `all` which can be used instead of manually adding or removing every port.

To stop only at the fail port, use `leash/1` like this:

```
?- leash(-all).
true.

?- leash(+fail).
true.

?- trace.
true.

[trace]  ?- noun(X, rock), adjective(X, color, red).
   Call: (11) noun(_3794, rock)
   Call: (12) is_a(_3794, rock)
   Exit: (12) is_a(rock1, rock)
   Exit: (11) noun(rock1, rock)
   Call: (11) adjective(rock1, color, red)
   Call: (12) color(rock1, red)
   Exit: (12) color(rock1, red)
   Exit: (11) adjective(rock1, color, red)
X = rock1 ;
   Redo: (12) is_a(_3794, rock)
   Exit: (12) is_a(rock2, rock)
   Exit: (11) noun(rock2, rock)
   Call: (11) adjective(rock2, color, red)
   Call: (12) color(rock2, red)
   Fail: (12) color(rock2, red) ? creep
   Fail: (11) adjective(rock2, color, red) ? creep
false.
```

Now, only the lines that start with "Fail:" have "creep" after them because that was the only time the tracer paused for a command. To never pause and just see all the traces, use `leash(-all)` and don't turn any ports back on.

The default ports are still printed out because a different setting, `visible/1`, controls which ports are printed. `visible/1` takes the same form of argument as `leash/1`. To only stop and show the `fail` port, use `leash/1` and `visible/1` like this:

```
?- leash(-all).
true.

?- leash(+fail).
true.

?- visible(-all).
true.

?- visible(+fail).
true.

?- trace.
true.

[trace]  ?- noun(X, rock), adjective(X, color, red).
X = rock1 ;
   Fail: (12) color(rock2, red) ? creep
   Fail: (11) adjective(rock2, color, red) ? creep
false.
```

2.10.4 Trace Mode Commands When Paused

You can do way more than just press `spacebar` when the tracer is paused at a port. All actions are single-character commands which are executed **without** waiting for a return (unless the command line option `--no-tty` is active). Pressing `?` or `h` when paused will print out a list of these commands as well.

Control Flow Commands

Abort	a	Abort Prolog execution (see `abort/0`)
Break	b	Enter a Prolog break environment (see `break/0`)
Creep	c	Continue execution, stop at next port. (Also `return`, `space`)
Exit	e	Terminate Prolog (see `halt/0`)
Fail	f	Force failure of the current goal
Find	/	Search for a port (see below for the description of this command (section 2.10.4))
Ignore	i	Ignore the current goal, pretending it succeeded
Leap	l	Continue execution, stop at next spy point
No debug	n	Continue execution in 'no debug' mode
Repeat find	.	Repeat the last find command (see 'Find' (section 2.10.4))
Retry	r	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
Skip	s	Continue execution, stop at the next port of **this** goal (thus skipping all calls to children of this goal)
Spy	+	Set a spy point (see `spy/1`) on the current predicate. Spy points are described later in the overview (section 2.10.6).
No spy	-	Remove the spy point (see `nospy/1`) from the current predicate. Spy points are described later in the overview (section 2.10.6).
Up	u	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.

Find (`/`) Description and Examples The Find (`/`) command continues execution until a port matching a find pattern is found. 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`.

Informational Commands

Alternatives	A	Show all goals that have alternatives
Goals	g	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.
Help	h	Show available options (also `?`)
Listing	L	List the current predicate with `listing/1`

Formatting Commands

Context	C	Toggle 'Show Context'. If `on`, the context module of the goal is displayed between square brackets (see modules section (section 6)). Default is `off`.
Display	d	Set the `max_depth(Depth)` option of debugger_write_options (section 2.12), limiting the depth to which terms are printed. See also the `w` and `p` options.
Print	p	Set the Prolog flag `debugger_write_options` to `[quoted(true), portray(true), max_depth(10), priority(699)]`. This is the default.
Write	w	Set the Prolog flag `debugger_write_options` to `[quoted(true), attributes(write), priority(699)]`, bypassing `portray/1`, etc.

2.10.5 Trace Mode vs. Trace Point

A slight detour is useful to describe some related predicates that can be confusing: To only trace a single or select set of predicates, the `trace/1` or `trace/2` predicates can be used to set a **trace point**. Even though they use the same base predicate name `trace`, these predicates ignore the `leash/1` and `visible/1` global settings and don't pause when they trace a port. They really are a different feature that also happens to do tracing.

A **trace point** is set on a particular predicate and traces the ports of that predicate **whether or not you are in** `trace/0` **trace mode**. Each trace point can trace different ports if the `trace/2` variant is used.

```
?- trace(is_a/2).
%          is_a/2: [all]
true.

?- noun(X, rock), adjective(X, color, red).
 T Call: is_a(_25702, rock)
 T Exit: is_a(rock1, rock)
X = rock1 ;
 T Redo: is_a(rock1, rock)
 T Exit: is_a(rock2, rock)
false.
```

Notice that **trace mode** did not have to be turned on using `trace/0` **and** that this only traced out the ports hit while executing `is_a/2` **and** that the program was not ever paused.

In fact, if trace mode is turned on while using a trace point, things get very confusing because the trace point infrastructure itself will be traced!

```
?- trace(is_a/2).
%          is_a/2: [all]
true.
```

```
?- trace.
true.

[trace]  ?- noun(X, rock), adjective(X, color, red).
   Call: (11) noun(_29318, rock) ? creep
   Call: (12) is_a(_29318, rock) ? creep
   Call: (13) print_message(debug, frame(user:is_a(_29318, rock), trace(call))) ?
   Call: (18) push_msg(frame(user:is_a(_29318, rock), trace(call))) ? creep
   Call: (21) exception(undefined_global_variable, '$inprint_message', _30046) ?
   Fail: (21) exception(undefined_global_variable, '$inprint_message', _30090) ?
   Exit: (18) push_msg(frame(user:is_a(_29318, rock), trace(call))) ? creep
   Call: (19) prolog:message(frame(user:is_a(_29318, rock), trace(call)), _30140,
   Fail: (19) prolog:message(frame(user:is_a(_29318, rock), trace(call)), _30140,
   Call: (19) message_property(debug, stream(_30192)) ? creep
   Fail: (19) message_property(debug, stream(_30192)) ? creep
   Call: (20) message_property(debug, prefix(_30200)) ? creep
   Fail: (20) message_property(debug, prefix(_30200)) ? creep
 T Call: is_a(_29318, rock)
   Call: (17) pop_msg ? creep
   Exit: (17) pop_msg ? creep
   ...Lots more after this...
```

So, trace **points** are a confusingly named and separate feature from trace **mode**.

2.10.6 Spy Points and Debug Mode

Back to trace mode features: Because the tracing output of a Prolog program can often be quite large, sometimes it is useful to start trace mode at a particular point deep in the program. This is what a **spy point** is for. It specifies a predicate that should turn on trace mode.

A spy point is enabled like this: `spy(mypredicate/2)`. After that command, the first time `mypredicate/2` is encountered, trace mode will turn on and work just like it does normally. This includes paying attention to the global `leash/1` and `visible/1` settings. The spy point can be removed using `nospy/1` or `nospyall/0`.

```
is_a(rock1, rock).
is_a(rock2, rock).
color(rock1, red).

noun(X, Type) :- is_a(X, Type).
adjective(X, color, Value) :- color(X, Value).
```

```
?- spy(is_a/2).
% Spy point on is_a/2
true.

[debug]  ?- noun(X, rock), adjective(X, color, red).
```

```
 * Call: (12) is_a(_1858, rock) ? creep
 * Exit: (12) is_a(rock1, rock) ? creep
   Exit: (11) noun(rock1, rock) ? creep
   Call: (11) adjective(rock1, color, red) ? creep
   Call: (12) color(rock1, red) ? creep
   Exit: (12) color(rock1, red) ? creep
   Exit: (11) adjective(rock1, color, red) ? creep
X = rock1 ;
 * Redo: (12) is_a(_1858, rock) ? creep
 * Exit: (12) is_a(rock2, rock) ? creep
   Exit: (11) noun(rock2, rock) ? creep
   Call: (11) adjective(rock2, color, red) ? creep
   Call: (12) color(rock2, red) ? creep
   Fail: (12) color(rock2, red) ? creep
   Fail: (11) adjective(rock2, color, red) ? creep
false.
```

After the spy point is hit, the output above is identical to the traces generated by running `trace/0` with the initial query, but is obviously missing all of the traces before the spy point.

Note that after `spy/1` is called, there is a new tag in front of `?-`, the `[debug]` tag:

```
?- spy(is_a/2).
% Spy point on is_a/2
true.

[debug] ?-
```

This means the system is in ”debug mode”. Debug mode does two things: it tells the system to watch for spy points and it turns off some optimizations that would make the traces confusing. The ideal 4-port model ([Byrd, 1980]) 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, choice points are only destroyed when removed by the cut and last call optimisation is switched off. [Note: This implies the system can run out of stack in debug mode, while no problems arise when running in non-debug mode.]

Debug mode can be turned off again using `nodebug/0`, but then the spy point will be ignored (but remembered). Turning debug mode back on via `debug/0` will hit the spy point again.

```
is_a(rock1, rock).
is_a(rock2, rock).
color(rock1, red).

noun(X, Type) :- is_a(X, Type).
adjective(X, color, Value) :- color(X, Value).
```

```
?-  spy(is_a/2).
% Spy point on is_a/2
true.

[debug]  ?- nodebug.
true.

?- noun(X, rock).
X = rock1 ;
X = rock2.

?- debug.
true.

[debug]  ?- noun(X, rock).
 * Call: (11) is_a(_47826, rock) ? creep
 * Exit: (11) is_a(rock1, rock) ? creep
   Exit: (10) noun(rock1, rock) ? creep
X = rock1 ;
 * Redo: (11) is_a(_47826, rock) ? creep
 * Exit: (11) is_a(rock2, rock) ? creep
   Exit: (10) noun(rock2, rock) ? creep
X = rock2.
```

So, debug mode allows Prolog to watch for spy points and enable trace mode when it hits one. The `tracing/0` and `debugging/0` predicates will report if the system is in either of those modes.

2.10.7 Breakpoints

Sometimes even spy points aren't enough. There may be a predicate that is used in many different places and it would be helpful to turn on tracing mode only when **one particular** call to it is made. **Breakpoints** allow for turning on trace mode when a specific source file, line number, and character in that line are hit. The predicates used are `set_breakpoint/4` and `set_breakpoint/5`. Many breakpoints can be active at a time.

Note that the interface provided by these predicates is not intended for end-users. The built-in PceEmacs editor that is also embedded in the graphical debugger allow setting break points based on the cursor position.

`Example.pl` has now been modified to have multiple calls to `noun/2`:

```
is_a(rock1, rock).
is_a(rock2, rock).
color(rock1, red).

noun(X, Type) :- is_a(X, Type).
adjective(X, color, Value) :- color(X, Value).
```

```
test_noun1(X, Type) :- noun(X, Type).
test_noun2(X, Type) :- noun(X, Type).
```

To enable tracing just when `noun/2` is called from `test_noun2/2`, `set_breakpoint/4` can be used like this:

```
?- set_breakpoint('/...path.../Example.pl', 8, 24, ID).
% Breakpoint 1 in 1-st clause of test_noun2/2 at Example.pl:8
ID = 1.

?- debug.
true.

[debug]  ?- noun(X, rock).
X = rock1 .

[debug]  ?- test_noun1(X, rock).
X = rock1 .

[debug]  ?- test_noun2(X, rock).
   Call: (11) noun(_44982, rock) ? creep
   Call: (12) is_a(_44982, rock) ? creep
   Exit: (12) is_a(rock1, rock) ? creep
   Exit: (11) noun(rock1, rock) ? creep
   Exit: (10) test_noun2(rock1, rock) ? creep
X = rock1 .

[trace]  ?- notrace.
true.

[debug]  ?-
```

The call to `set_breakpoint/4` had to specify the source file ("`Example.pl`"), the line number (8), and the character within that line (24) to precisely specify what clause should turn on trace mode (this is much easier using the graphical debugger because it shows source code).

The breakpoint won't get triggered if the system isn't in debug mode but, unlike setting a spy point, `set_breakpoint/4` does **not** do this automatically. So, it was turned on manually using `debug/0`.

The output shows that only the call to `test_noun2/2` (where the breakpoint was set) actually turned on trace mode. Note that the `[Trace] ?-` at the end shows that trace mode is left on after being triggered. It can be turned off again via `notrace/0`, which will leave the system in debug mode. All debugging modes can be shut off at once by calling `nodebug/0` since shutting off debug mode automatically turns off trace mode.

2.10.8 Command Line Debugger Summary

In summary, there are really two distinct "tracing" features: trace **mode** and trace **points**. Both write traces to the console using the "Byrd Box Model" but that's where similarity ends.

Trace Mode

Trace mode is the main Prolog command line debugger that allows for tracing the transitions through the resolution states of predicates represented by ports in the "Byrd Box Model" and optionally pausing for a command when certain ports are hit.

It can be turned on manually via `trace/0`, or (when put into debug mode using `debug/0`) when a specific predicate is encountered via `spy/1`, or when a specific call to a predicate is encountered via `set_breakpoint/4` or `set_breakpoint/5`.

When in trace mode, `visible/1` controls which ports are written to the console, and `leash/1` controls which ports cause execution to pause to allow program inspection.

When execution is paused, there are many commands that can be used to inspect the state of the program, cause goals to fail or succeed, etc.

Trace mode is turned off via `notrace/0` and debug mode is turned off via `nodebug/0`.

Trace Points

Trace **points** are a separate feature from trace **mode** that allow writing specified ports to the console when a predicate is being evaluated. It does not ever pause program execution and does not need to be in trace or debug mode to work.

They are turned on via `trace/1` and `trace/2`.

They don't pay attention to `visible/1` (because the ports shown are set in `trace/2`) or `leash/1` (because they don't pause execution).

They can be turned off via `trace/2`.

2.11 Compilation

2.11.1 During program development

During program development, programs are normally loaded using the list abbreviation (`?- [load].`). 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:

```
% swipl
<banner>

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

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

When using Windows, the user may open `load.pl` from the Windows explorer, which will cause `swipl-win.exe` to be started in the directory holding `load.pl`. Prolog loads `load.pl` before entering the top level. If Prolog is started from an interactive shell, one may choose the type `swipl -s load.pl`.

2.11.2 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

A Prolog source file can be used directly as a Unix program using the Unix `#!` magic start. The Unix `#!` magic is allowed because if the first letter of a Prolog file is `#`, the first line is treated as a comment.[6] To create a Prolog script, use one of the two alternatives below as first line. The first can be used to bind a script to a specific Prolog installation, while the latter uses the default prolog installed in `$PATH`.

```
#!/path/to/swipl
#!/usr/bin/env swipl
```

The interpretation of arguments to the executable in the *HashBang* line differs between Unix-derived systems. For portability, the `#!` must be followed immediately with an absolute path to the executable and should have none or one argument. Neither the executable path, nor the argument shall use quotes or spaces. When started this way, the Prolog flag `argv` contains the command line arguments that follow the script invocation.

Starting with version 7.5.8, `initialization/2` support the *When* options `program` and `main`, allowing for the following definition of a Prolog script that evaluates an arithmetic expression on the command line. Note that `main/0` is defined lib the library `main`. It calls `main/1` with the command line arguments after disabling signal handling.

```
#!/usr/bin/env swipl

:- initialization(main, main).

main(Argv) :-
        concat_atom(Argv, ' ', SingleArg),
        term_to_atom(Term, SingleArg),
        Val is Term,
        format('~w~n', [Val]).
```

And here are two example runs:

[6] 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.

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

Prolog script may be launched for debugging or inspection purposes using the `-l` or `-t`. For example, `-l` merely loads the script, ignoring `main` and `program` initialization.

```
swipl -l eval 1+1
<banner>

?- main.
2
true.

?-
```

We can also force the program to enter the interactive toplevel after the application is completed using `-t prolog`:

```
swipl -t prolog eval 1+1
2
?-
```

The Windows version simply ignores the `#!` line.[7]

Creating a shell script

With the introduction of *PrologScript* (see section 2.11.2), 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=swipl

exec $PL -q -f "$base/load" --
```

[7]Older versions extracted command line arguments from the *HashBang* line. As of version 5.9 all relevant setup can be achieved using *directives*. Due to the compatibility issues around *HashBang* line processing, we decided to remove it completely.

```
:- initialization go.

go :-
        current_prolog_flag(argv, Arguments),
        go(Arguments).

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 the `-c` command line option. A saved state is a file containing machine-independent[8] 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 13.

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:

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

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

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

This performs exactly the same as executing

```
% swipl
<banner>

?- [load].
```

[8]The saved state does not depend on the CPU instruction set or endianness. Saved states for 32- and 64-bits are not compatible. Typically, saved states only run on the same version of Prolog on which they have been created.

```
?- qsave_program(myprog,
                 [ goal(main),
                   stand_alone(true)
                 ]).
?- halt.
```

2.12 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*) *[ISO]*

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 *Key* instantiated, it unifies *Value* with the value of the Prolog flag or fails if the *Key* is not a Prolog flag.

Flags marked *changeable* can be modified by the user using `set_prolog_flag/2`. Flag values are typed. Flags marked as `bool` can have the values `true` or `false`. The predicate `create_prolog_flag/3` may be used to create flags that describe or control behaviour of libraries and applications. The library `settings` provides an alternative interface for managing notably application parameters.

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>
        )
```

Some Prolog flags are scoped to a source file. This implies that if they are set using a directive inside a file, the flag value encountered when loading of the file started is restored when loading of the file is completed. Currently, the following flags are scoped to the source file: `generate_debug_info` and `optimise`.

A new thread (see section 10) *copies* all flags from the thread that created the new thread (its *parent*).[9] As a consequence, modifying a flag inside a thread does not affect other threads.

abi_version *(dict)*

The flag value is a dict with keys that describe the version of the various Application Binary Interface (ABI) components. See section 2.22 for details.

[9] This is implemented using the copy-on-write technique.

access_level *(atom, changeable)*
This flag defines a normal 'user' view (`user`, default) or a 'system' view. In system view all system code is fully accessible as if it was normal user code. In user view, certain operations are not permitted and some details are kept invisible. We leave the exact consequences undefined, but, for example, system code can be traced using system access and system predicates can be redefined.

address_bits *(integer)*
Address size of the hosting machine. Typically 32 or 64. Except for the maximum stack limit, this has few implications to the user. See also the Prolog flag `arch`.

agc_margin *(integer, changeable)*
If this amount of atoms possible garbage atoms exist 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()`.[10]

allow_dot_in_atom *(bool, changeable)*
If `true` (default `false`), dots may be embedded into atoms that are not quoted and start with a letter. The embedded dot *must* be followed by an identifier continuation character (i.e., letter, digit or underscore). The dot is allowed in identifiers in many languages, which can make this a useful flag for defining DSLs. Note that this conflicts with cascading functional notation. For example, `Post.meta.author` is read as `.(Post, 'meta.author'` if this flag is set to `true`.

allow_variable_name_as_functor *(bool, changeable)*
If true (default is false), `Functor(arg)` is read as if it were 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.

android *(bool)*
If present and true, it indicates we are running on the Android OS. The flag is not present in other operating systems.

android_api *(integer)*
If running on Android, it indicates the compile-time API Level defined by the C macro `__ANDROID_API__`. It is not defined if running on other operating systems. The API level may or may not match the API level of the running device, since it is the API level at compile time.

answer_write_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)]`.

apple *(bool)*
If present and `true`, the operating system is MacOSX. Defined if the C compiler used

[10]Given that SWI-Prolog has no limit on the length of atoms, 10,000 atoms may still occupy a lot of memory. Applications using extremely large atoms may wish to call `garbage_collect_atoms/0` explicitly or lower the margin.

to compile this version of SWI-Prolog defines `__APPLE__`. Note that the `unix` is also defined for MacOSX.

`apple_universal_binary`boolr If present and `true`, SWI-Prolog has been build as a *universal binary*. Universal binaries contain native executable code for multiple architectures. Currently the supported architectures are `x86_64` and `arm64`. The archirecture prefix for components is `fat-darwin` while the `arch` depends on the actual CPU type.

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 12.2.3 and `file_search_path/2`. For Apple, see also `apple_universal_binary`.

argv *(list, changeable)*
List is a list of atoms representing the application command line arguments. Application command line arguments are those that have *not* been processed by Prolog during its initialization. Note that Prolog's argument processing stops at `--` or the first non-option argument. See also `os_argv`.[11]

associated_file *(atom)*
Set if Prolog was started with a prolog file as argument. Used by e.g., `edit/0` to edit the initial file.

autoload *(atom, changeable)*
This flag controls autoloading predicates based on `autoload/1` and `autoload/2` as well as predicates from *autoload libraries*. It has the following values:

false
Predicates are never auto-loaded. If predicates have been imported before using `autoload/[1,2]`, load the referenced files immediately using `use_module/[1,2]`. Note that most of the development utilities such as `listing/1` have to be explicitly imported before they can be used at the toplevel.

explicit
Do not autoload from *autoload libraries*, but do use lazy loading for predicates imported using `autoload/[1,2]`.

user
As `false`, but to autoload library predicates into the global `user` module. This makes the development tools and library implicitly available to the toplevel, but not to modules.

user_or_explicit
Combines `explicit` with `user`, providing lazy loading of predicates imported using `autoload/[1,2]` and implicit access to the whole library for the toplevel.

true
Provide full autoloading everywhere. This is the default.

back_quotes *(codes,chars,string,symbol_char, changeable)*
Defines the term-representation for back-quoted material. The default is `codes`. If `--traditional` is given, the default is `symbol_char`, which allows using ` in operators composed of symbols.[12] See also section 5.2.

[11] Prior to version 6.5.2, `argv` was defined as `os_argv` is now. The change was made for compatibility reasons and because the current definition is more practical.

[12] Older versions had a boolean flag `backquoted_strings`, which toggled between `string` and `symbol_char`

backtrace *(bool, changeable)*
If `true` (default), print a backtrace on an uncaught exception.

backtrace_depth *(integer, changeable)*
If backtraces on errors are enabled, this flag defines the maximum number of frames that is printed (default 20).

backtrace_goal_depth *(integer, changeable)*
The frame of a backtrace is printed after making a shallow copy of the goal. This flag determines the depth to which the goal term is copied. Default is '3'.

backtrace_show_lines *(bool, changeable)*
If `true` (default), try to reconstruct the line number at which the exception happened.

bounded *(bool)*
ISO Prolog flag. If `true`, integer representation is bound by `min_integer` and `max_integer`. If `false` integers can be arbitrarily large and the `min_integer` and `max_integer` are not present. See section 4.27.2.

break_level *(integer)*
Current break-level. The initial top level (started with `-t`) has value 0. See `break/0`. This flag is absent from threads that are not running a top-level loop.

c_cc *(atom, changeable)*
Name of the C compiler used to compile SWI-Prolog. Normally either gcc or cc. See section 12.5.

c_cflags *(atom, changeable)*
CFLAGS used to compile SWI-Prolog. See section 12.5.

c_ldflags *(atom, changeable)*
LDFLAGS used to link SWI-Prolog. See section 12.5.

c_libplso *(atom, changeable)*
Libraries needed to link extensions (shared object, DLL) to SWI-Prolog. Typically empty on ELF systems and `-lswipl` on COFF-based systems. See section 12.5.

c_libs *(atom, changeable)*
Libraries needed to link executables that embed SWI-Prolog. Typically `-lswipl` if the SWI-Prolog kernel is a shared (DLL). If the SWI-Prolog kernel is in a static library, this flag also contains the dependencies.

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. See section 2.16.1.

character_escapes_unicode *(bool, changeable)*
If `true` (default), `write/1` and friends write escaped characters using the `\uXXXX` or `\UXXXXXXXX` syntax rather than the ISO Prolog `\x<hex>\` syntax. SWI-Prolog reads both.

cmake_build_type *(atom)*
Provides the cmake *build type* used to build this version of SWI-Prolog.

colon_sets_calling_context *(bool)*
Using the construct ⟨*module*⟩:⟨*goal*⟩ sets the *calling context* for executing ⟨*goal*⟩. This flag is defined by ISO/IEC 13211-2 (Prolog modules standard). See section 6.

color_term *(bool, changeable)*
This flag is managed by library `ansi_term`, which is loaded at startup if the two conditions below are both true. Note that this implies that setting this flag to `false` from the system or personal initialization file (see section 2.2 disables colored output. The predicate `message_property/2` can be used to control the actual color scheme depending in the message type passed to `print_message/2`.

- `stream_property(current_output, tty(true))`
- `\+ current_prolog_flag(color_term, false)`

compile_meta_arguments *(atom, changeable)*
This flag controls compilation of arguments passed to meta-calls marked '0' or '^' (see `meta_predicate/1`). Supported values are:

false
(default). Meta-arguments are passed verbatim. If the argument is a control structure ((A,B), (A;B), (A-¿B;C), etc.) it is compile to an temporary clause allocated on the environment stack when the meta-predicate is called.

control
Compile meta-arguments that contain control structures to an auxiliary predicate. This generally improves performance as well as the debugging experience.

always
Always create an intermediate clause, even for system predicates.[13]

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 `swipl-win.exe` to indicate that the console supports menus. See also section 4.35.4.

cpu_count *(integer, changeable)*
Number of physical CPUs or cores in the system. The flag is marked read-write both to allow pretending the system has more or less processors. See also `thread_setconcurrency/2` and the library `thread`. This flag is not available on systems where we do not know how to get the number of CPUs. This flag is not included in a saved state (see `qsave_program/1`).

dde *(bool)*
Set to `true` if this instance of Prolog supports DDE as described in section 4.44.

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

[13] This may be used in the future for replacing the normal head of the generated predicate with a special reference (similar to database references as used by, e.g., `assert/2`) that provides direct access to the executable code, thus avoiding runtime lookup of predicates for meta-calling.

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 the Prolog flag `report_error`. Default is `true`.

debug_on_interrupt *(bool, changeable)*
If `true`, start the debugger on Control-C.[14]. The initial value is `false` and the value is set to `true` when entering the interactive top level. See `--debug-on-interrupt` to start handling interrupts immediately.

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.

debugger_write_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)]`.

determinism_error *(atom, changeable)*
This flag defines the behaviour when the predicate determinism is not according to its declaration. See `det/1`. Possible values are `error` (default), `warning` and `silent`.

dialect *(atom)*
Fixed to `swi`. The code below is a reliable and portable way to detect SWI-Prolog.

```
is_dialect(swi) :-
        catch(current_prolog_flag(dialect, swi), _, fail).
```

double_quotes *(codes,chars,atom,string, changeable)*
This flag determines how double quoted strings are read by Prolog and is —like `character_escapes` and `back_quotes`— maintained for each module. The default is `string`, which produces a string as described in section 5.2. If `--traditional` is given, the default is `codes`, which produces a list of character codes, integers that represent a Unicode code-point. The value `chars` produces a list of one-character atoms and the value `atom` makes double quotes the same as single quotes, creating a atom. See also section 5.

editor *(atom, changeable)*
Determines the editor used by `edit/1`. See section 4.4.1 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 2.19.1 for details.

[14]More precisely when receiving `SIGINT`

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

exit_status *(integer)*
Set by `halt/1` to its argument, making the exit status available to hooks registered with `at_halt/1`.

file_name_case_handling *(atom, changeable)*
This flag defines how Prolog handles the case of file names. The flag is used for case normalization and to determine whether two names refer to the same file.[15] It has one of the following values:

case_sensitive
The filesystem is fully case sensitive. Prolog does not perform any case modification or case insensitive matching. This is the default on Unix systems.

case_preserving
The filesystem is case insensitive, but it preserves the case with which the user has created a file. This is the default on Windows systems.

case_insensitive
The filesystem doesn't store or match case. In this scenario Prolog maps all file names to lower case.

file_name_variables *(bool, changeable)*
If `true` (default `false`), expand `\$\arg{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` can be used to expand environment variables and wildcard patterns. This Prolog flag is intended for backward compatibility with older versions of SWI-Prolog.

file_search_cache_time *(number, changeable)*
Time in seconds for which search results from `absolute_file_name/3` are cached. Within this time limit, the system will first check that the old search result satisfies the conditions. Default is 10 seconds, which typically avoids most repetitive searches for (library) files during compilation. Setting this value to 0 (zero) disables the cache.

float_max *(float)*
The biggest representable floating point number.

float_max_integer *(float)*
The highest integer that can be represented precisely as a floating point number.

float_min *(float)*
The smallest representable floating point number above 0.0. See also `nexttoward/2`.

float_overflow *(atom, changeable)*
One of `error` (default) or `infinity`. The first is ISO compliant. Using `infinity`, floating point overflow is mapped to positive or negative `Inf`. See section 4.27.2.

float_rounding *(atom, changeable)*
Defines how arithmetic rounds to a float. Defined values are `to_nearest` (default), `to_positive`, `to_negative` or `to_zero`. For most scenarios the function `roundtoward/2` provides a safer and faster alternative.

[15] BUG: Note that file name case handling is typically a properly of the filesystem, while Prolog only has a global flag to determine its file handling.

float_undefined *(atom, changeable)*
One of `error` (default) or `nan`. The first is ISO compliant. Using `nan`, undefined operations such as `sqrt`(*-2.0*) is mapped to `NaN`. See section 4.27.2.

float_underflow *(atom, changeable)*
One of `error` or `ignore` (default). The second is ISO compliant, binding the result to 0.0.

float_zero_div *(atom, changeable)*
One of `error` (default) or `infinity`. The first is ISO compliant. Using `infinity`, division by 0.0 is mapped to positive or negative `Inf`. See section 4.27.2.

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.

gc_thread *(bool)*
If `true` (default if threading is enabled), atom and clause garbage collection are executed in a separate thread with the *alias* `gc`. Otherwise the thread that detected sufficient garbage executes the garbage collector. As running these global collectors may take relatively long, using a separate thread improves real time behaviour. The `gc` thread can be controlled using `set_prolog_gc_thread/1`.

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 `--no-debug`. This flag is scoped within a source file. Many of the libraries have `:- set_prolog_flag(generate_debug_info, false)` to hide their details from a normal trace.[16]

gmp_version *(integer)*
If Prolog is linked with GMP, this flag gives the major version of the GMP library used. See also section 12.4.9.

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 2.8. 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*⟩`/boot.prc` and to find its library as ⟨*home*⟩`/library`. Some installations may put architecture independent files in a *shared home* and also define `shared_home`. System files can be found using `absolute_file_name/3` as `swi`(*file*). See `file_search_path/2`.

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

[16]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 further changes to the compiler.

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 with normal SWI-Prolog behaviour. Currently it has the following effect:

- The `//2` (float division) *always* returns a float, even if applied to integers that can be divided.
- In the standard order of terms (see section 4.6.1), all floats are before all integers.
- `atom_length/2` yields a type 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.
- Syntax is closer to the ISO standard:
  - Unquoted commas and bars appearing as atoms are not allowed. Instead of `f(,,a)` now write `f(',',a)`. Unquoted commas can only be used to separate arguments in functional notation and list notation, and as a conjunction operator. Unquoted bars can only appear within lists to separate head and tail, like `[Head|Tail]`, and as infix operator for alternation in grammar rules, like `a --> b | c.`
  - Within functional notation and list notation terms must have priority below 1000. That means that rules and control constructs appearing as arguments need bracketing. A term like `[a :- b, c].` must now be disambiguated to mean `[(a :- b), c].` or `[(a :- b, c)].`
  - Operators appearing as operands must be bracketed. Instead of `X == -, true.` write `X == (-), true.` Currently, this is not entirely enforced.
  - Backslash-escaped newlines are interpreted according to the ISO standard. See section 2.16.1.

large_files *(bool)*
If present and `true`, SWI-Prolog has been compiled with *large file support* (LFS) and is capable of accessing files larger than 2GB. This flag is always `true` on 64-bit hardware and true on 32-bit hardware if the configuration detected support for LFS. Note that it may still be the case that the *file system* on which a particular file resides puts limits on the file size.

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

malloc *(atom)*
Set after a successful identification of the used malloc() implementation. Currently possibly values are `tcmalloc` and `ptmalloc`. See section 4.43.2 for details.

max_answers_for_subgoal *(integer, changeable)*
Limit the number of answers in a table. The atom `infinite` clears the flag. By default this flag is not defined. See section 7.10 for details.

max_answers_for_subgoal_action *(atom, changeable)*
The action taken when a table reaches the number of answers specified in `max_answers_for_subgoal`. Supported values are `bounded_rationality`, `error` (default) or `suspend`.

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

max_char_code *(integer)*
Highest (Unicode) code point that is supported. SWI-Prolog supports all Unicode code points from 0 (zero) upto and including the value of this flag. Currently `0xffff` on Windows (UCS-2) and `0x10ffff` on most other platforms.

max_integer *(integer)*
Maximum integer value if integers are *bounded*. See also the flag `bounded` and section 4.27.2.

max_procedure_arity *(integer)*
Maximum arity for a predicate. An attempt to define or call such a predicate results in a `representation_error(max_procedure_arity)` exception. Currently set to 1024.

max_rational_size *(integer, changeable)*
Limit the size in bytes for rational numbers. This *tripwire* can be used to identify cases where setting the Prolog flag `prefer_rationals` to `true` creates excessively big rational numbers and, if precision is not required, one should use floating point arithmetic.

max_rational_size_action *(atom, changeable)*
Action when the `max_rational_size` tripwire is exceeded. Possible values are `error` (default), which throws a tripwire resource error and `float`, which converts the rational number into a floating point number. Note that rational numbers may exceed the range for floating point numbers.

max_table_answer_size *(integer, changeable)*
Limit the size of an answer substitution for tabling. The atom `infinite` clears the flag. By default this flag is not defined. See section 7.10 for details.

max_table_answer_size_action *(atom, changeable)*
The action taken if an answer substitution larger than `max_table_answer_size` is added to a table. Supported values are `error` (default), `bounded_rationality`, `suspend` and `fail`.

max_table_subgoal_size *(integer, changeable)*
Limit the size of a goal term accessing a table. The atom `infinite` clears the flag. By default this flag is not defined. See section 7.10 for details.

max_table_subgoal_size_action *(atom, changeable)*
The action taken if a tabled goal exceeds `max_table_subgoal_size`. Supported values are `error` (default), `abstract` and `suspend`.

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

message_context *(list(atom), changeable)*
Context information to add to messages of the levels `error` and `warning`. The list may contain the elements `thread` to add the thread that generates the message to the message, `time` or `time`(*Format*) to add a time stamp. The default time format is `%T.%3f`. The default is `[thread]`. See also `format_time/3` and `print_message/2`.

min_integer *(integer)*
Minimum integer value if integers are *bounded*. See also the flag `bounded` and section 4.27.2.

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

mitigate_spectre *(bool, changeable)*
When `true` (default `false`), enforce mitigation against the Spectre timing-based security vulnerability. Spectre based attacks can extract information from memory owned by the process that should remain invisible, such as passwords or the private key of a web server. The attacks work by causing speculative access to sensitive data, and leaking the data via side-channels such as differences in the duration of successive instructions. An example of a potentially vulnerable application is SWISH. SWISH allows users to run Prolog code while the swish server must protect the privacy of other users as well as its HTTPS private keys, cookies and passwords.

Currently, enabling this flag reduces the resolution of `get_time/1` and `statistics/2` CPU time to $20\mu s$.

WARNING: Although a coarser timer makes a successful attack of this type harder, it does not reliably prevent such attacks in general. Full mitigation may require compiler support to disable speculative access to sensitive data.

occurs_check *(atom, changeable)*
This flag controls unification that creates an infinite tree (also called *cyclic term*) and can have three values. Using `false` (default), unification succeeds, creating an infinite tree. Using `true`, unification behaves as `unify_with_occurs_check/2`, failing silently. Using `error`, an attempt to create a cyclic term results in an `occurs_check` exception. The latter is intended for debugging unintentional creations of cyclic terms. Note that this flag is a global flag modifying fundamental behaviour of Prolog. Changing the flag from its default may cause libraries to stop functioning properly.

on_error *(atom, changeable)*
Determines how to act on an error printed using `print_message/2`, i.e., an error that is reported to the user. The possible values are `print` (default), `status` and `halt`. Using `halt` the process halts immediately with status 1. Otherwise execution continues. Using `status halt/0` exits with status 1 if one or more errors were printed by the process. In *compile* mode (see `-c`) the default is `status`. This flag can be set from the commandline using `--on-error`. See also section 4.3.2.

on_warning *(atom, changeable)*
As `on_error`, but for warnings. The default is always `print`. The commandline option is `--on-warning`.

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. The `optimise` flag is scoped to a source file.

Currently optimised 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`).

optimise_unify *(bool, changeable)*
If `true` (default), allow the compiler to (re)move explicit unification calls (`=/2`). While this behaviour can significantly improve performance, it is not yet handled properly by the source-level debugger. See section 2.18.3.

os_argv *(list, changeable)*
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. See `argv` to get the application options.

packs *(bool)*
If `true`, extension packs (add-ons) are attached. Can be set to `false` using the `--no-packs`.

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

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.

portable_vmi *(bool, changeable)*
If `true` (default), generate `.qlf` files and saved states that run both on 32 bit and 64-bit hardware. If `false`, some optimized virtual machine instructions are only used if the integer argument is within the range of a tagged integer for 32-bit machines.

posix_shell *(atom, changeable)*
Path to a POSIX compatible shell. This default is typically `/bin/sh`. This flag is used by `shell/1` and `qsave_program/2`.

prefer_rationals *(bool, changeable)*
Only provided if the system is compiled with unbounded and rational arithmetic support (see `bounded`). If `true`, prefer arithmetic to produce rational numbers over floats. This implies:

- Division (`//2`) of two integers produces a rational number.
- Power (`^/2`) of two integers produces a rational number, *also* if the second operant is a negative number. For example, `2^(-2)` evaluates to `1/4`.

Using `true` can create excessively large rational numbers. The Prolog flag `max_rational_size` can be used to detect and act on this *tripwire*.

If `false`, rational numbers can only be created using the functions `rational/1`, `rationalize/1` and `rdiv/2` or by reading them. See also `rational_syntax`, section 2.16.1 and section 4.27.2.

The current default is `false`. We consider changing this to `true` in the future. Users are strongly encouraged to set this flag to `true` and report issues this may cause.

print_write_options *(term, changeable)*
: Specifies the options for `write_term/2` used by `print/1` and `print/2`.

prompt_alternatives_on *(atom, changeable)*
: Determines prompting for alternatives in the Prolog top level. Default is `determinism`, which implies the system prompts for alternatives if the goal succeeded while leaving choice points. Many classical Prolog systems behave as `groundness`: they prompt for alternatives if and only if the query contains variables.

protect_static_code *(bool, changeable)*
: If `true` (default `false`), `clause/2` does not operate on static code, providing some basic protection from hackers that wish to list the static code of your Prolog program. Once the flag is `true`, it cannot be changed back to `false`. Protection is default in ISO mode (see Prolog flag `iso`). Note that many parts of the development environment require `clause/2` to work on static code, and enabling this flag should thus only be used for production code.

qcompile *(atom, changeable)*
: This option provides the default for the `qcompile`(+*Atom*) option of `load_files/2`.

rational_syntax *(atom, changeable)*
: Determines the read and write syntax for rational numbers. Possible values are `natural` (e.g., `1/3`) or `compatibility` (e.g., `1r3`). The `compatibility` syntax is always accepted. This flag is module sensitive.

The default for this flag is currently `compatibility`, which reads and writes rational numbers as e.g., `1r3`.[17] We will consider `natural` as a default in the future. Users are strongly encouraged to set this flag to `natural` and report issues this may cause.

readline *(atom, changeable)*
: Specifies which form of command line editing is provided. Possible values are below. The flag may be set from the user's init file (see section 2.3) to one of `false`, `readline` or `editline`. This causes the toplevel not to load a command line editor (`false`) or load the specified one. If loading fails the flag is set to `false`.

false
: No command line editing is available.

readline
: The library `readline` is loaded, providing line editing based on the GNU readline library.

editline
: The library `editline` is loaded, providing line editing based on the BSD libedit. This is the default if `editline` is available and can be loaded.

swipl_win
: SWI-Prolog uses its own console (`swipl-win.exe` on Windows, the Qt based `swipl-win` on MacOS) which provides line editing.

[17] There is still some discussion on the separating character. See section 2.16.1.

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.

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`.

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

sandboxed_load *(bool, changeable)*
If `true` (default `false`), `load_files/2` calls hooks to allow library(sandbox) to verify the safety of directives.

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

shared_home *(atom)*
Indicates that part of the SWI-Prolog system files are installed in ⟨*prefix*⟩`/share/swipl` instead of in the home at the ⟨*prefix*⟩`/lib/swipl`. This flag indicates the location of this *shared home* and the directory is added to the file search path `swi`. See `file_search_path/2` and the flag `home`.

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`.

shared_object_search_path *(atom)*
Name of the environment variable used by the system to search for shared objects.

shared_table_space *(integer, changeable)*
Space reserved for storing shared answer tables. See section 7.9 and the Prolog flag `table_space`.

shift_check *(bool, changeable)*
When `true` (default `false`), check for suspicious delimited continuations captured by `shift_for_copy/1`.

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 `--no-signals` is active. See section 12.4.23 for details.

stack_limit *(int, changeable)*
Limits the combined sizes of the Prolog stacks for the current thread. See also `--stack-limit` and section 2.20.1.

stream_type_check *(atom, changeable)*
Defines whether and how strictly the system validates that byte I/O should not be applied to text streams and text I/O should not be applied to binary streams. Values are `false` (no checking), `true` (full checking) and `loose`. Using checking mode `loose` (default),

the system accepts byte I/O from text stream that use ISO Latin-1 encoding and accepts writing text to binary streams.

string_stack_tripwire *(int, changeable)*
Maintenance for foreign language string management. Prints a warning if the string stack depth hits the tripwire value. See section 12.4.12 for details.

system_thread_id *(int)*
Available in multithreaded version (see section 10) where the operating system provides system-wide integer thread identifiers. The integer is the thread identifier used by the operating system for the calling thread. See also `thread_self/1`.

table_incremental *(bool, changeable)*
Set the default for whether to use incremental tabling or not. Initially set to `false`. See `table/1`.

table_shared *(bool, changeable)*
Set the default for whether to use shared tabling or not. Initially set to `false`. See `table/1`.

table_space *(integer, changeable)*
Space reserved for storing answer tables for *tabled predicates* (see `table/1`).[18] When exceeded a `resource_error`(*table_space*) exception is raised.

table_subsumptive *(bool, changeable)*
Set the default choice between *variant* tabling and *subsumptive* tabling. Initially set to `false`. See `table/1`.

threads *(bool, changeable)*
True when threads are supported. If the system is compiled without thread support the value is `false` and read-only. Otherwise the value is `true` unless the system was started with the `--no-threads`. Threading may be disabled only if no threads are running. See also the `gc_thread` flag.

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 `format_time/3`.

tmp_dir *(atom, changeable)*
Path to the temporary directory. initialised from the environment variable `TMP` or `TEMP` in windows. If this variable is not defined a default is used. This default is typically `/tmp` or `c:/temp` in windows.

toplevel_goal *(term, changeable)*
Defines the goal that is executed after running the initialization goals and entry point (see `-g`, `initialization/2` and section 2.11.2. The initial value is `default`, starting a normal interactive session. This value may be changed using the command line option `-t`. The explicit value `prolog` is equivalent to `default`. If `initialization`(*Goal,main*) is used and the toplevel is `default`, the toplevel is set to `halt` (see `halt/0`).

toplevel_list_wfs_residual_program *(bool, changeable)*
If `true` (default) and the answer is *undefined* according to the Well Founded Semantics

[18]BUG: Currently only counts the space occupied by the nodes in the answer tries.

(see section 7.6), list the *residual program* before the answer. Otherwise the answer terminated with **undefined**. See also `undefined/0`.

toplevel_mode *(atom, changeable)*
If `backtracking` (default), the toplevel backtracks after completing a query. If `recursive`, the toplevel is implemented as a recursive loop. This implies that global variables set using `b_setval/2` are maintained between queries. In *recursive* mode, answers to toplevel variables (see section 2.9) are kept in backtrackable global variables and thus **not copied**. In *backtracking* mode answers to toplevel variables are kept in the recorded database (see section 4.14.2).

The recursive mode has been added for interactive usage of CHR (see section 9),[19] which maintains the global constraint store in backtrackable global variables.

toplevel_name_variables *(bool, changeable)*
If `true` (default), give names to variables at the toplevel instead of printing them as *_NNN*. The variables are named *_A*, *_B*, ... Variables that appear only once (singletons) are printed as _.

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_factorized *(bool, changeable)*
If `true` (default `false`) show the internal sharing of subterms in the answer substitution. The example below reveals internal sharing of leaf nodes in *red-black trees* as implemented by the `rbtrees` predicate `rb_new/1`:

```
?- set_prolog_flag(toplevel_print_factorized, true).
?- rb_new(X).
X = t(_S1, _S1), % where
    _S1 = black('', _G387, _G388, '').
```

If this flag is `false`, the `% where` notation is still used to indicate cycles as illustrated below. This example also shows that the implementation reveals the internal cycle length, and *not* the minimal cycle length. Cycles of different length are indistinguishable in Prolog (as illustrated by `S == R`).

```
?- S = s(S), R = s(s(R)), S == R.
S = s(S),
R = s(s(R)).
```

toplevel_prompt *(atom, changeable)*
Define the prompt that is used by the interactive top level. The following ~ (tilde) sequences are replaced:

~m	*Type in* module if not `user` (see `module/1`)
~l	*Break level* if not 0 (see `break/0`)
~d	*Debugging state* if not normal execution (see `debug/0`, `trace/0`)
~!	*History event* if history is enabled (see flag `history`)

[19] Suggested by Falco Nogatz

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 2.9.

trace_gc *(bool, changeable)*
If `true` (default `false`), garbage collections and stack-shifts will be reported on the terminal. May be changed. Values are reported in bytes as G+T, where G is the global stack value and T the trail stack value. 'Gained' describes the number of bytes reclaimed. 'used' the number of bytes on the stack after GC and 'free' the number of bytes allocated, but not in use. Below is an example output.

```
% GC: gained 236,416+163,424 in 0.00 sec;
      used 13,448+5,808; free 72,568+47,440
```

traditional *(bool)*
Available in SWI-Prolog version 7. If `true`, 'traditional' mode has been selected using `--traditional`. Notice that some SWI7 features, like the functional notation on dicts, do not work in this mode. See also section 5.

tty_control *(bool, changeable)*
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. If this flag is `false` at startup, command line editing is disabled. See also the `--no-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. See also `apple` and `windows`.

unknown *(fail,warning,error, changeable)*
Determines the behaviour if an undefined procedure is encountered. If `fail`, the predicate 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 and inherited from the module's *import-module*. Using default setup, this implies that normal modules inherit the flag from `user`, which in turn inherit the value `error` from `system`. The user may change the flag for module `user` to change the default for all application modules or for a specific module. It is strongly advised to keep the `error` default and use `dynamic/1` and/or `multifile/1` to specify possible non-existence of a predicate.

unload_foreign_libraries *(bool, changeable)*
If `true` (default `false`), unload all loaded foreign libraries. Default is `false` because modern OSes reclaim the resources anyway and unloading the foreign code may cause registered hooks to point to no longer existing data or code.

user_flags *(Atom, changeable)*
Define the behaviour of `set_prolog_flag/2` if the flag is not known. Values are `silent`, `warning` and `error`. The first two create the flag on-the-fly, where `warning` prints a message. The value `error` is consistent with ISO: it raises an existence error and does not create the flag. See also `create_prolog_flag/3`. The default is `silent`, but future versions may change that. Developers are encouraged to

use another value and ensure proper use of `create_prolog_flag/3` to create flags for their library.

var_prefix *(bool, changeable)*
If `true` (default `false`), variables must start with an underscore (_). May be changed. This flag is local to the module in which it is changed. See section 2.16.1.

verbose *(atom, changeable)*
This flag 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_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`.

verbose_load *(atom, changeable)*
Determines messages printed for loading (compiling) Prolog files. Current values are `full` (print a message at the start and end of each file loaded), `normal` (print a message at the end of each file loaded), `brief` (print a message at end of loading the toplevel file), and `silent` (no messages are printed, default). The value of this flag is normally controlled by the option `silent`(*Bool*) provided by `load_files/2`.

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

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

version_data *(swi(Major, Minor, Patch, Extra))*
Part of the dialect compatibility layer; see also the Prolog flag `dialect` and section C. *Extra* provides platform-specific version information as a list. *Extra* is used for *tagged versions* such as “7.4.0-rc1”, in which case *Extra* contains a term `tag`(*rc1*).

version_git *(atom)*
Available if created from a git repository. See `git-describe` for details.

warn_override_implicit_import *(bool, changeable)*
If `true` (default), a warning is printed if an implicitly imported predicate is clobbered by a local definition. See `use_module/1` for details.

win_file_access_check *(atom, changeable)*
Controls the behaviour or `access_file/2` under Windows. There is no reliable way to check access to files and directories on Windows. This flag allows for switching between three alternative approximations.

access
Use Windows _waccess() function. This ignores ACLs (Access Control List) and thus may indicate that access is allowed while it is not.

getfilesecurity
Use the Windows GetFileSecurity() function. This does not work on all file systems, but is probably the best choice on file systems that do support it, notably local NTFS volumes.

openclose
Try to open the file and close it. This works reliable for files, but not for directories. Currently directories are checked using _waccess(). This is the default.

windows *(bool)*
If present and `true`, the operating system is an implementation of Microsoft Windows. This flag is only available on MS-Windows based versions. See also `unix`.

wine_version *(atom)*
If present, SWI-Prolog is the MS-Windows version running under the Wine emulator.

write_attributes *(atom, changeable)*
Defines how `write/1` and friends write attributed variables. The option values are described with the `attributes` option of `write_term/2`. 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.

xref *(bool, changeable)*
If `true`, source code is being read for *analysis* purposes such as cross-referencing. Otherwise (default) it is being read to be compiled. This flag is used at several places by `term_expansion/2` and `goal_expansion/2` hooks, notably if these hooks use side effects. See also the libraries `prolog_source` and `prolog_xref`.

set_prolog_flag(*:Key, +Value*) *[ISO]*
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.

Some flags (e.g., `unknown`) are maintained on a per-module basis. The addressed module is determined by the *Key* argument.

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. Defined types are `boolean` (if the initial value is one of `false`, `true`, `on` or `off`), `atom` if the initial value is any other atom, `integer` if the value is an integer that can be expressed as a 64-bit signed value. Any other initial value results in an untyped flag that can represent any valid Prolog term.

The behaviour when *Key* denotes a non-existent key depends on the Prolog flag `user_flags`. The default is to define them silently. New code is encouraged to use `create_prolog_flag/3` for portability.

create_prolog_flag(*+Key, +Value, +Options*) *[YAP]*

Create a new Prolog flag. The ISO standard does not foresee creation of new flags, but many libraries introduce new flags. *Options* is a list of the options below. See also `user_flags`.

access(*+Access*)
Define access rights for the flag. Values are `read_write` and `read_only`. The default is `read_write`.

type(*+Atom*)
Define a type restriction. Possible values are `boolean`, `atom`, `integer`, `float` and `term`. The default is determined from the initial value. Note that `term` restricts the term to be ground.

keep(*+Boolean*)
If `true`, do not modify the flag if it already exists. Otherwise (default), this predicate behaves as `set_prolog_flag/2` if the flag already exists.

2.13 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).
- `message_property/2`
  Hook into `print_message/2` that defines prefix, output stream, color, etc.
- `message_prefix_hook/2`
  Hook into `print_message/2` to add additional prefixes to the message such as the time and thread.
- `library_directory/1`
  Hook into `absolute_file_name/3` to define new library directories (most Prolog systems).
- `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 systems).
- `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 an 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`
  Define 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 8.1 for details.

2.14 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 (see section 6.10. If this fails the *auto loader* is activated.[20] On first activation an index to all library files in all library directories is loaded in core (see `library_directory/1`, `file_search_path/2` and `reload_library_index/0`). If the undefined predicate can be located in 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` key `verbose_autoload` is provided to get verbose loading. The Prolog flag `autoload` can be used to enable/disable the autoload system. A more controlled form of autoloading as well as lazy loading application modules is provided by `autoload/[1,2]`.

[20] Actually, the hook `user:exception/3` is called; only if this hook fails it calls the autoloader.

Autoloading only handles (library) source files that use the module mechanism described in chapter 6. The files are loaded with `use_module/2` and only the trapped undefined predicate is imported into 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 an indexed library directory.

```
% mkdir ~/${XDG_DATA_HOME-.config}/swi-prolog/lib
% cd ~/${XDG_DATA_HOME-.config}/swi-prolog/lib
% swipl -g 'make_library_index(.)' -t halt
```

If there is more than one library file 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.

autoload_path(+*DirAlias***)**

Add *DirAlias* to the libraries that are used by the autoloader. This extends the search path `autoload` and reloads the library index. For example:

```
:- autoload_path(library(http)).
```

If this call appears as a directive, it is term-expanded into a clause for `user:file_search_path/2` and a directive calling `reload_library_index/0`. This keeps source information and allows for removing this directive.

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.

```
:- prolog_load_context(directory, Dir),
   make_library_index(Dir,
                      [ '*.pl',
                        'trace/browse.pl',
                        'swi/*.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.

When creating an executable using either `qsave_program/2` or the `-c` command line options, it is necessary to load all predicates that would normally be autoloaded explicitly. This is discussed in section 13. See `autoload_all/0`.

2.15 Packs: community add-ons

SWI-Prolog has a mechanism for easy incorporation of community extensions. See the pack landing page for details and available packs. This section documents the built-in predicates to attach packs. Predicates for creating, registering and installing packs are provided by the library `prolog_pack`.

attach_packs

Attaches all packs in subdirectories of directories that are accessible through the *file search path* (see `absolute_file_name/3`) `pack`. The default for this search path is given below. See `file_search_path/2` for the `app_data` search path.

```
user:file_search_path(pack, app_data(pack)).
```

The predicate `attach_packs/0` is called on startup of SWI-Prolog.

attach_packs(+*Directory*)

Attach all packs in subdirectories of *Directory*. Same as `attach_packs`(*Directory, []*).

attach_packs(+*Directory*, +*Options*)

Attach all packs in subdirectories of *Directory*. Options is one of:

search(+*Where*)
Determines the order in which pack library directories are searched. Default is to add new packages at the end (`last`). Using `first`, new packages are added at the start.

duplicate(+*Action*)
Determines what happens if a pack with the same name is already attached. Default is `warning`, which prints a warning and ignores the new pack. Other options are `keep`, which is like `warning` but operates silently and `replace`, which detaches the old pack and attaches the new.

The predicate `attach_packs/2` can be used to attach packages that are bundled with an application.

2.16 The SWI-Prolog syntax

SWI-Prolog syntax is close to ISO-Prolog standard syntax, which is based on the Edinburgh Prolog syntax. A formal description can be found in the ISO standard document. For an informal introduction we refer to Prolog text books (see section 1) and online tutorials. In addition to the differences from the ISO standard documented here, SWI-Prolog offers several extensions, some of which also extend the syntax. See section 5 for more information.

2.16.1 ISO Syntax Support

This section lists various extensions w.r.t. the ISO Prolog 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 Unicode. See also section 2.19.

Nested comments

SWI-Prolog allows for nesting `/* ...*/` comments. Where the ISO standard accepts `/* .../* ...*/` as a comment, SWI-Prolog will search for a terminating `*/`. This is useful if some code with `/* ...*/` comment statements in it should be commented out. This modification also avoids unintended commenting in the example below, where the closing `*/` of the first comment has been forgotten.[21]

```
/* comment

code

/* second comment */

code
```

[21]Recent copies of GCC give a style warning if `/*` is encountered in a comment, which suggests that this problem has been recognised more widely.

Character Escape Syntax

Within quoted atoms (using single quotes: `'`⟨*atom*⟩`'`) special characters are represented using escape sequences. An escape sequence is led in by the backslash (`\`) character. The list of escape sequences is compatible with the ISO standard but contains some extensions, and the interpretation of numerically specified characters is slightly more flexible to improve compatibility. Undefined escape characters raise a `syntax_error` exception.[22]

`\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. Not supported by ISO. Example:

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

`\`⟨NEWLINE⟩
: When in ISO mode (see the Prolog flag `iso`), only skip this sequence. In native mode, white space that follows the newline is skipped as well and a warning is printed, indicating that this construct is deprecated and advising to use `\c`. We advise using `\c` or putting the layout *before* the `\`, as shown below. Using `\c` is supported by various other Prolog implementations and will remain supported by SWI-Prolog. The style shown below is the most compatible solution.[23]

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

instead of

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

Note that SWI-Prolog also allows unescaped newlines to appear in quoted material. This is not allowed by the ISO standard, but used to be common practice before.

`\e`
: Escape character (ASCII 27). Not ISO, but widely supported.

`\f`
: Form-feed character.

[22] Up to SWI-Prolog 6.1.9, undefined escape characters were copied verbatim, i.e., removing the backslash.
[23] Future versions will interpret `\`⟨*return*⟩ according to ISO.

`\n`
: Next-line character.

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

`\s`
: Space character. Intended to allow writing `0'\s` to get the character code of the space character. Not ISO.

`\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 with 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, 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.

`\\`
: Escapes the backslash itself. Thus, `'\\'` is an atom consisting of a single `\`.

`\'`
: Single quote. Note that `'\''` and `''''` both describe the atom with a single `'`, i.e., `'\'' == ''''` is true.

`\"`
: Double quote.

`` \` ``
: Back quote.

Character escaping is only available if `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 as octal 40 (decimal 32) by `read`. Also, the `writef/2` sequence `\l` is illegal. 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.

Using digit groups in large integers

SWI-Prolog supports splitting long integers into *digit groups*. Digit groups can be separated with the sequence ⟨*underscore*⟩, ⟨*optional white space*⟩. If the ⟨*radix*⟩ is 10 or lower, they may also be separated with exactly one space. The following all express the integer 1 million:

```
1_000_000
1 000 000
1_000_/*more*/000
```

Integers can be printed using this notation with `format/2`, using the `~I` format specifier. For example:

```
?- format('~I', [1000000]).
1_000_000
```

The current syntax has been proposed by Ulrich Neumerkel on the SWI-Prolog mailinglist.

Rational number syntax

As of version 8.1.22, SWI-Prolog supports rational numbers as a primary citizen atomic data type if SWI-Prolog is compiled with the GMP library. This can be tested using the `bounded` Prolog flag. An atomic type also requires a syntax. Unfortunately there are few options for adding rational numbers without breaking the ISO standard.[24]

ECLiPSe and SWI-Prolog have agreed to define the canonical syntax for rational numbers to be e.g., `1r3`. In addition, ECLiPSe accepts `1_3` and SWI-Prolog can be asked to accept `1/3` using the module sensitive Prolog flag `rational_syntax`, which has the values below. Note that `write_canonical/1` always uses the compatible `1r3` syntax.

natural

This is the default mode where we ignore the ambiguity issue and follow the most natural ⟨*integer*⟩/⟨*nonneg*⟩ alternative. Here, ⟨*integer*⟩ follows the normal rules for Prolog decimal

[24] ECLiPSe uses *numerator_denominator*. This syntax conflicts with SWI-Prolog digit groups (see section 2.16.1) and does not have a recognised link to rational numbers. The notation `1/3r` and `1/3R` have also been proposed. The `1/3r` is compatible to Ruby, but is hard to parse due to the required look-ahead and not very natural. See also https://en.wikipedia.org/wiki/Rational_data_type.

integers and ⟨*nonneg*⟩ does the same, but does not allows for a sign. Note that the parser translates a rational number to its canonical form which implies there are no common divisors in the resulting numerator and denominator. Examples of ration numbers are:

1/2	1/2
2/4	1/2
1 000 000/33 000	1000/33
-3/5	-3/5

We expect very few programs to have text parsed into a rational number while a term was expected. Note that for rationals appearing in an arithmetic expression the only difference is that evaluation moves from runtime to compiletime. The utility `list_rationals/0` may be used on a loaded program to check whether the program contains rational numbers inside clauses and thus may be subject to compatibility issues. If a term is intended this can be written as `/(1,2)`, `(1)/2`, `1 / 2` or some variation thereof.

compatibility
Read and write rational numbers as e.g., `1r3`. In other words, this adheres to the same rules as `natural` above, but using the '`r`' instead of '`/`'. Note that this may conflict with traditional Prolog as '`r`' can be defined as an infix operator. The same argument holds for `0x23` and similar syntax for numbers that are part of the ISO standard.

While the syntax is controlled by the flag `rational_syntax`, behavior on integer division and exponentiation is controlled by the flag `prefer_rationals`. See section section 4.27.2 for arithmetic on rational numbers.

NaN and Infinity floats and their syntax

SWI-Prolog supports reading and printing 'special' floating point values according to Proposal for Prolog Standard core update wrt floating point arithmetic by Joachim Schimpf and available in ECLiPSe Prolog. In particular,

- Infinity is printed as `1.0Inf` or `-1.0Inf`. Any sequence matching the regular expression `[+-]?\sd+[.]\sd+Inf` is mapped to plus or minus infinity.
- `NaN` (Not a Number) is printed as `1.xxxNaN`, where *1.xxx* is the float after replacing the exponent by '1'. Such numbers are read, resulting in the same `NaN`. The `NaN` constant can also be produced using the function `nan/0`, e.g.,

```
?- A is nan.
A = 1.5NaN.
```

By default SWI-Prolog arithmetic (see section 4.27) follows the ISO standard with describes that floating point operations either produce a *normal* floating point number or raise an exception. section 4.27.2 describes the Prolog flags that can be used to support the IEEE special float values. The ability to create, read and write such values facilitates the exchange of data with languages that can represent the full range of IEEE doubles.

Force only underscore to introduce a variable

According to the ISO standard and most Prolog systems, identifiers that start with an uppercase letter or an underscore are variables. In the past, *Prolog by BIM* provided an alternative syntax, where only the underscore (_) introduces a variable. As of SWI-Prolog 7.3.27 SWI-Prolog supports this alternative syntax, controlled by the Prolog flag `var_prefix`. As the `character_escapes` flag, this flag is maintained per module, where the default is `false`, supporting standard syntax.

Having only the underscore introduce a variable is particularly useful if code contains identifiers for case sensitive external languages. Examples are the RDF library where code frequently specifies property and class names[25] and the R interface for specifying functions or variables that start with an uppercase character. Lexical databases where part of the terms start with an uppercase letter is another category were the readability of the code improves using this option.

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 6.0.0 of the Unicode standard. Please note 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 2.16.1) 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.[26] 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 have the notion of character case. In such languages variables *must* be written as `_name`.

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

- *Control and unassigned characters*
  Control and unassigned (C*) characters produce a syntax error if encountered outside quoted atoms/strings and outside comments.

- *Other characters*
  The first 128 characters follow the ISO Prolog standard. Unicode symbol and punctuation characters (general category S* and P*) act as glueing symbol characters (i.e., just like ==: an unquoted sequence of symbol characters are combined into an atom).

[25] Samer Abdallah suggested this feature based on experience with non-Prolog users using the RDF library.

[26] http://www.unicode.org/reports/tr31/

Other characters (this is mainly `No`: *a numeric character of other type*) are currently handled as 'solo'.

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 that a variable is meant 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.[27] 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.[28]

Any normal variable appearing exactly once in the clause *and* any named singleton variables appearing more than once are reported. Below are some examples with warnings in the right column. Singleton messages can be suppressed using the `style_check/1` directive.

test(_).	
test(_a).	Singleton variables: [_a]
test(_12).	Singleton variables: [_12]
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).	

Semantic singletons Starting with version 6.5.1, SWI-Prolog has *syntactic singletons* and *semantic singletons*. The first are checked by `read_clause/3` (and `read_term/3` using the option `singletons(`*warning*`)`). The latter are generated by the compiler for variables that appear alone in a *branch*. For example, in the code below the variable *X* is not a *syntactic* singleton, but the variable *X* does not communicate any bindings and replacing *X* with `_` does not change the semantics.

```
test :-
        (   test_1(X)
```

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

```
        ;   test_2(X)
        ).
```

2.17 Rational trees (cyclic terms)

SWI-Prolog supports rational trees, also known as cyclic terms. 'Supports' is so defined that most relevant built-in predicates terminate when faced with rational trees. Almost all SWI-Prolog's built-in term manipulation predicates process terms in a time that is linear to the amount of memory used to represent the term on the stack. The following set of predicates safely handles rational trees: `=../2`, `==/2`, `=@=/2`, `=/2`, `@</2`, `@=</2`, `@>=/2`, `@>/2`, `\==/2`, `\=@=/2`, `\=/2`, `acyclic_term/1`, `bagof/3`, `compare/3`, `copy_term/2`, `cyclic_term/1`, `dif/2`, `duplicate_term/2`, `findall/3`, `ground/1`, `term_hash/2`, `numbervars/3`, `numbervars/4`, `recorda/3`, `recordz/3`, `setof/3`, `subsumes_term/2`, `term_variables/2`, `throw/1`, `unify_with_occurs_check/2`, `unifiable/3`, `when/2`, `write/1` (and related predicates) .

In addition, some built-ins recognise rational trees and raise an appropriate exception. Arithmetic evaluation belongs to this group. The compiler (`asserta/1`, etc.) also raises an exception. Future versions may support rational trees. Predicates that could provide meaningful processing of rational trees raise a `representation_error`. Predicates for which rational trees have no meaningful interpretation raise a `type_error`. For example:

```
1 ?- A = f(A), asserta(a(A)).
ERROR: asserta/1: Cannot represent due to `cyclic_term'
2 ?- A = 1+A, B is A.
ERROR: is/2: Type error: `expression' expected, found
             `@(S_1,[S_1=1+S_1])' (cyclic term)
```

2.18 Just-in-time clause indexing

SWI-Prolog provides 'just-in-time' indexing over multiple arguments.[29] 'Just-in-time' means that clause indexes are not built by the compiler (or `asserta/1` for dynamic predicates), but on the first call to such a predicate where an index might help (i.e., a call where at least one argument is instantiated). This section describes the rules used by the indexing logic. Note that this logic is not 'set in stone'. The indexing capabilities of the system will change. Although this inevitably leads to some regressing on some particular use cases, we strive to avoid significant slowdowns.

The list below describes the clause selection process for various predicates and calls. The alternatives are considered in the order they are presented.

- *Special purpose code*
  Currently two special cases are recognised by the compiler: static code with exactly one clause and static code with two clauses, one where the first argument is the empty list (`[]`) and one where the first argument is a non-empty list (`[_|_]`).

[29] JIT indexing was added in version 5.11.29 (Oct. 2011).

- *Linear scan on first argument*
  The principal clause list maintains a *key* for the first argument. An indexing key is either a constant or a functor (name/arity reference). Calls with an instantiated first argument and less than 10 clauses perform a linear scan for a possible matching clause using this index key. If the result is deterministic it is used. Otherwise the system looks for better indexes.[30].

- *Hash lookup*
  If none of the above applies, the system considers the available hash tables for which the corresponding argument is instantiated. If a table is found with acceptable characteristics, it is used. Otherwise it assesses the clauses for all instantiated arguments and selects the best candidate for creating a new hash table. If there is no single argument that provides an acceptable hash quality it will search for a combination of arguments.[31] Searching for index candidates is only performed on the first 254 arguments.

  If a single-argument index contains multiple compound terms with the same name and arity and at least one non-variable argument, a *list index* is created. A subsequent query where this argument is bound to a compound causes jiti indexing to be applied *recursively* on the arguments of the term. This is called *deep indexing*.[32] See also section 2.18.1

  Clauses that have a variable at an otherwise indexable argument must be linked into all hash buckets. Currently, predicates that have more than 10% such clauses for a specific argument are not considered for indexing on that argument.

  Disregarding variables, the suitability of an argument for hashing is expressed as the number of unique indexable values divided by the standard deviation of the number of duplicate values for each value plus one.[33]

  The indexes of dynamic predicates are deleted if the number of clauses is doubled since its creation or reduced below 1/4th. The JIT approach will recreate a suitable index on the next call. Indexes of running predicates cannot be deleted. They are added to a 'removed index list' associated to the predicate. Outdated indexes of predicates are reclaimed by `garbage_collect_clauses/0`. The clause garbage collector is scheduled automatically, based on time and space based heuristics. See `garbage_collect_clauses/0` for details.

The library `prolog_jiti` provides `jiti_list/0`,1 to list the characteristics of all or some of the created hash tables.

Dynamic predicates are indexed using the same rules as static predicates, except that the *special purpose* schemes are never applied. In addition, the JITI index is discarded if the number of clauses has doubled since the predicate was last assessed or shrinks below one fourth. A subsequent call reassesses the statistics of the dynamic predicate and, when applicable, creates a new index.

[30]Up to 7.7.2 this result was used also when non-deterministic.
[31]The last step was added in SWI-Prolog 7.5.8.
[32]Deep indexing was added in version 7.7.4.
[33]Earlier versions simply used the number of unique values, but poor distribution of values makes a table less suitable. This was analysed by Fabien Noth and Günter Kniesel.

2.18.1 Deep indexing

As introduced in section 2.18, *deep indexing* creates hash tables distinguish clauses that share a compound with the same name and arity. Deep indexes allow for efficient lookup of arbitrary terms. Without it is advised to *flatten* the term, i.e., turn `F(`*X*`)` into two arguments for the fact, one argument denoting the functor *F* and the second the argument X. This works fine as long as the arity of the each of the terms is the same. Alternatively we can use `term_hash/2` or `term_hash/4` to add a column holding the hash of the term. That approach can deal with arbitrary arities, but requires us to know that the term is ground (`term_hash/2`) or up to which depth we get sufficient selectivity (`term_hash/4`).

Deep indexing does not require this knowledge and leads to efficient lookup regardless of the instantiation of the query and term. The current version does come with some limitations:

- The decision which index to use is taken independently at each level. Future versions may be smarter on this.
- Deep indexing only applies to a *single argument* indexes (on any argument).
- Currently, the depth of indexing is limited to 7 levels.

Note that, when compiling DCGs (see section 4.13) and the first body term is a *literal*, it is included into the clause head. See for example the grammar and its plain Prolog representation below.

```
det(det(a), sg)   --> "a".
det(det(an), pl)  --> "an".
det(det(the), _)  --> "the".
```

```
?- listing(det).
det(det(a), sg, [97|A], A).
det(det(an), pl, [97, 110|A], A).
det(det(the), _, [116, 104, 101|A], A).
```

Deep argument indexing will create indexes for the 3rd list argument, providing speedup and making clause selection deterministic if all rules start with a literal and all literals are unique in the first 6 elements. Note that deep index creation stops as soon as a deterministic choice can be made or there are no two clauses that have the same name/arity combination.

2.18.2 Future directions

- The 'special cases' can be extended. This is notably attractive for static predicates with a relatively small number of clauses where a hash lookup is too costly.
- Create an efficient decision diagram for selecting between low numbers of static clauses.
- Implement a better judgements for selecting between deep and plain indexes.

2.18.3 Indexing for body code

The current SWI-Prolog versions only consider the head for generating clause indexing. This would make it impossible to examine a head argument and pass the argument in the body without copying the argument. Consider the two clauses below. Both have equal semantics under Prolog. The first version would loose clause indexing while the second creates a copy of the `f/1` argument. Neither is desirable.

```
p(X) :- X = f(I), integer(I), q(X).
p(f(I)) :- integer(I), q(f(X)).
```

As of SWI-Prolog 8.3.21, unifications against head arguments that happen before anything else in the body are compiled special. Effectively, the term unified too is moved into the head (providing indexing) and places where this term is used simply use the corresponding argument. The explicit unification is removed. Decompilation (`clause/2`) reverses this process, but may not produce exactly the same term. The re-inserted unfications are ordered according to the argument position and the variable is always on the left hand of the `=/2`. Thus,

```
p(X,Y) :- f(_) = Y, X = g(_), q(X,Y).
```

Is decompiled into the following equivalent clause.

```
p(X,Y) :- X = g(_), Y = f(_), q(X,Y).
```

Additional notes:

- This transformation is only performed on *static* code.
- The unifications must *immediately* follow the head in a *conjunction*.
- As sole exception, calls to `true/0` are skipped. This allows `goal_expansion/2` to convert goals to `true` while preserving this optimization.
- If the head argument is not used the body unification is still moved into the head. The decompiler does not inverse the process in that case. Thus, `p(X) :- X = a.` is fully equivalent to `p(a).`
- Currently this optimziation is enabled regardless of the Prolog flag `optimise`. As this optimization harms source-level debugging, this may not be desirable. On the other hand we do not want determinism to depend on optimization while this optimization affects determinism.

2.18.4 Indexing and portability

The base-line functionality of Prolog implementations provides indexing on constants and functor (name/arity) on the first argument. This must be your assumption if wide portability of your program is important. This can typically be achieved by exploiting `term_hash/2` or `term_hash/4` and/or maintaining multiple copies of a predicate with reordered arguments and wrappers that update all implementations (assert/retract) and selects the appropriate implementation (query).

YAP provides full JIT indexing, including indexing arguments of compound terms. YAP's indexing has been the inspiration for enhancing SWI-Prolog's indexing capabilities.

2.19 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-bit 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 5.2). 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 chapter 12 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.

2.19.1 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 influences `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 `setlocale(LC_CTYPE, NULL)` to one of `text`, `utf8` or `iso_latin_1`. One of the latter two is used if the encoding name is recognized, while `text` is used as default. Using `text`, the translation is left to the wide-character functions of the C library.[34] 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 with 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 2.16.1 for syntax issues. For additional information and Unicode resources, please visit `http://www.unicode.org/`.

SWI-Prolog currently defines and supports the following encodings:

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.

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

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 with `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 writes the characters using Prolog escape sequences while other streams generate an I/O exception.

BOM: Byte Order Mark

From section 2.19.1, you may have got the impression that text files are complicated. This section deals with a related topic, making life 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 off 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 use of a UTF-8 BOM. In other cases there is additional information revealing 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`.

2.20 System limits

2.20.1 Limits on memory areas

The SWI-Prolog engine uses three *stacks* the *local stack* (also called *environment stack*) stores the environment frames used to call predicates as well as choice points. The *global stack* (also called *heap*) contains terms, floats, strings and large integers. Finally, the *trail stack* records variable bindings and assignments to support *backtracking*. The internal data representation limits these stacks to 128 MB (each) on 32-bit processors. More generally to $2^{\text{bits-per-pointer}-5}$ bytes, which implies they are virtually unlimited on 64-bit machines.

As of version 7.7.14, the stacks are restricted by the writeable flag `stack_limit` or the command line option `--stack-limit`. This flag limits the combined size of the three stacks per thread. The default limit is currently 512 Mbytes on 32-bit machines, which imposes no additional limit considering the 128 Mbytes hard limit on 32-bit and 1 Gbytes on 64-bit machines.

Considering portability, applications that need to modify the default limits are advised to do so using the Prolog flag `stack_limit`.

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. No limits are imposed on the addresses returned by malloc() and friends.

2.20.2 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 length of atoms and strings. The number of atoms is limited to 16777216 (16M) on 32-bit machines. On 64-bit machines this is virtually unlimited. See also section 12.4.2.

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.

Nesting of terms Most built-in predicates that process Prolog terms create an explicitly managed stack and perform optimization for processing the last argument of a term. This implies they can process deeply nested terms at constant and low usage of the C stack, and the system raises a resource error if no more stack can be allocated. Currently only `read/1` and `write/1` (and all variations thereof) still use the C stack and may cause the system to crash in an uncontrolled way (i.e., not mapped to a Prolog exception that can be caught).

Integers On most systems SWI-Prolog is compiled with support for unbounded integers by means of the GNU GMP library. In practice this means that integers are bound by the global stack size. Too large integers cause a `resource_error`. On systems that lack GMP, 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 integers that appear in clauses, the value (below `max_tagged_integer` or not) has little impact on the size of the clause.

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

Area name	Description
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 off with the !/0 predicate or no choice points have been created since the invocation and the last subclause is started (last call optimisation).
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).
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 no longer needed. As the trail and global stacks are garbage collected together, a small trail can cause an excessive amount of garbage collections. To avoid this, the trail is automatically resized to be at least 1/6th of the size of the global stack.

Table 2.2: Memory areas

2.20.3 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.

2.21 SWI-Prolog and 64-bit machines

Most of today's 64-bit platforms are capable of running both 32-bit and 64-bit applications. This asks for some clarifications on the advantages and drawbacks of 64-bit addressing for (SWI-)Prolog.

2.21.1 Supported platforms

SWI-Prolog can be compiled for a 32- or 64-bit address space on any system with a suitable C compiler. Pointer arithmetic is based on the type (u)intptr_t from `stdint.h`, with suitable emulation on MS-Windows.

2.21.2 Comparing 32- and 64-bits Prolog

Most of Prolog's memory usage consists of pointers. This indicates the primary drawback: Prolog memory usage almost doubles when using the 64-bit addressing model. Using more memory means copying more data between CPU and main memory, slowing down the system.

What then are the advantages? First of all, SWI-Prolog's addressing of the Prolog stacks does not cover the whole address space due to the use of *type tag bits* and *garbage collection flags*. On 32-bit hardware the stacks are limited to 128 MB each. This tends to be too low for demanding applications on modern hardware. On 64-bit hardware the limit is 2^{32} times higher, exceeding the addressing capabilities of today's CPUs and operating systems. This implies Prolog can be started with stack sizes that use the full capabilities of your hardware.

Multi-threaded applications profit much more because every thread has its own set of stacks. The Prolog stacks start small and are dynamically expanded (see section 2.20.1). The C stack is also dynamically expanded, but the maximum size is *reserved* when a thread is started. Using 100 threads at the maximum default C stack of 8Mb (Linux) costs 800Mb virtual memory![35]

The implications of theoretical performance loss due to increased memory bandwidth implied by exchanging wider pointers depend on the design of the hardware. We only have data for the popular IA32 vs. AMD64 architectures. Here, it appears that the loss is compensated for by an instruction set that has been optimized for modern programming. In particular, the AMD64 has more registers and the relative addressing capabilities have been improved. Where we see a 10% performance degradation when placing the SWI-Prolog kernel in a Unix shared object, we cannot find a measurable difference on AMD64.

2.21.3 Choosing between 32- and 64-bit Prolog

For those cases where we can choose between 32 and 64 bits, either because the hardware and OS support both or because we can still choose the hardware and OS, we give guidelines for this decision.

[35] C-recursion over Prolog data structures is removed from most of SWI-Prolog. When removed from all predicates it will often be possible to use lower limits in threads. See `http://www.swi-prolog.org/Devel/CStack.html`

First of all, if SWI-Prolog needs to be linked against 32- or 64-bit native libraries, there is no choice as it is not possible to link 32- and 64-bit code into a single executable. Only if all required libraries are available in both sizes and there is no clear reason to use either do the different characteristics of Prolog become important.

Prolog applications that require more than the 128 MB stack limit provided in 32-bit addressing mode must use the 64-bit edition. Note however that the limits must be doubled to accommodate the same Prolog application.

If the system is tight on physical memory, 32-bit Prolog has the clear advantage of using only slightly more than half of the memory of 64-bit Prolog. This argument applies as long as the application fits in the *virtual address space* of the machine. The virtual address space of 32-bit hardware is 4GB, but in many cases the operating system provides less to user applications.

The only standard SWI-Prolog library adding significantly to this calculation is the RDF database provided by the *semweb* package. It uses approximately 80 bytes per triple on 32-bit hardware and 150 bytes on 64-bit hardware. Details depend on how many different resources and literals appear in the dataset as well as desired additional literal indexes.

Summarizing, if applications are small enough to fit comfortably in virtual and physical memory, simply take the model used by most of the applications on the OS. If applications require more than 128 MB per stack, use the 64-bit edition. If applications approach the size of physical memory, fit in the 128 MB stack limit and fit in virtual memory, the 32-bit version has clear advantages. For demanding applications on 64-bit hardware with more than about 6GB physical memory the 64-bit model is the model of choice.

2.22 Binary compatibility

SWI-Prolog first of all attempts to maintain *source code* compatibility between versions. Data and programs can often be represented in binary form. This touches a number of interfaces with varying degrees of compatibility. The relevant version numbers and signatures are made available by `PL_version()`, the `--abi-version` and the Prolog flag `abi_version`.

Foreign extensions

Dynamically loadable foreign extensions have the usual dependencies on the architecture, ABI model of the (C) compiler, dynamic link library format, etc. They also depend on the backward compatibility of the PL_* API functions provided lib `libswipl`.

A compatible API allows distribution of foreign extensions in binary form, notably for platforms on which compilation is complicated (e.g., Windows). This compatibility is therefore high on the priority list, but must infrequently be compromised.

`PL_version()`: `PL_VERSION_FLI`, `abi_version` key: `foreign_interface`

Binary terms

Terms may be represented in binary format using `PL_record_external()` and `fast_write/2`. As these formats are used for storing binary terms in databases or communicate terms between Prolog processes in binary form, great care is taken to maintain compatibility.

`PL_version()`: `PL_VERSION_REC`, `abi_version` key: `record`

QLF files

QLF files (see `qcompile/1`) are binary representation of Prolog file or module. They represent clauses as sequences of *virtual machine* (VM) instructions. Their compatibility relies on the QLF file format and the ABI of the VM. Some care is taken to maintain compatibility.

`PL_version()`: `PL_VERSION_QLF`, `PL_VERSION_QLF_LOAD` and `PL_VERSION_VM`, `abi_version` key: `qlf`, `qlf_min_load`, `vmi`

Saved states

Saved states (see `-c` and `qsave_program/2`) is a zip file that contains the entire Prolog database using the same representation as QLF files. A saved state may contain additional resources, such as foreign extensions, data files, etc. In addition to the dependency concerns of QLF files, built-in and core library predicates may call *internal* foreign predicates. The interface between the public built-ins and internal foreign predicates changes frequently. Patch level releases in the *stable branch* will as much as possible maintain compatibility.

The relevant ABI version keys are the same as for QLF files with one addition: `PL_version()`: `PL_VERSION_BUILT_IN`, `abi_version` key: `built_in`

3 Initialising and Managing a Prolog Project

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 explains 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 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 its 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. 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/fileExtension`. 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 its own directory or directory structure. If nobody else will ever touch your files and you use only one computer, there is little to 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 reaching the file system, SWI-Prolog 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. Third, 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)).
```

When 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 with calls to `use_module/1` 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 calling `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 2.19, SWI-Prolog supports international character handling. Its internal encoding is UNICODE. I/O streams convert to/from this internal format. This section discusses the options for source files not in US-ASCII.

SWI-Prolog can read files in any of the encodings described in section 2.19. 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.[1] 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 use the same *locale* or the file is unreadable. There is no elegant way 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 of this encoding in two ways, using a UTF-8 *BOM* (see section 2.19.1) or using the directive `:- encoding(utf8).` Many of today's text editors, including PceEmacs, are capable of editing UTF-8 files. Projects that were started using local conventions can be re-coded using the Unix `iconv` tool or often using 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:

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

- *They hide local predicates*
  This is the reason they were 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 than 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 except the name and the 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 in 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 is a good understanding required of the resolution of terms to predicates inside a module. 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 Prolog source code. Editors are complicated programs that must be mastered in detail for real productive programming. 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`.

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  '/home/jan/.config/swi-prolog/lib/chat/countr.pl':16
  2 chat:country/1   '/home/jan/.config/swi-prolog/lib/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 will normally abort 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. For example, 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.[2]

```
?- 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.

[2]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).

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.[3]

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*. There 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/.../swipl`
  If the file is a *Prolog Script* file, starting with the line `#!/path/to/swipl` *options*, 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 you 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.

[3]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.

- *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.[4] 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 forwards for next.

  Control-R
  : Search backwards for next.

  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*, dynamic abbreviation is an important feature of Emacs clones to support programming. After typing the first few letters of an identifier, you may press Alt-/, causing PceEmacs to search backwards for identifiers that start the same and use 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 entering code very fast 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, press 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 in 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 are the most commonly used commands. In section 3.4.3 we discuss specific support for dealing with Prolog source code.

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

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.

- *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**heck **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.[5]

- *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 command re-centers the window (scrolls the window to make the caret the center of the window). Typing 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 with 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.[6] Below are examples of typical constructs.

```
head(arg1, arg2).

head(arg1, arg2) :- !.

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

head(Arg1, arg2) :-
```

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

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 undefined predicate
Purple	Call to dynamic predicate
Other entities	
Dark green	Comment
Dark blue	Quoted atom or string
Brown	Variable

Table 3.1: Colour conventions

```
        (   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 a 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.

The graphical tracer is particularly useful for debugging threads. The tracer must be loaded from the `main` thread before it can be used from a background thread.

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 the 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-referencer is a tool that examines the caller-callee relation between predicates, and, using this information to explicate dependency relations between source files, finds calls to non-existing predicates and predicates for which no callers can be found. Cross-referencing is useful during program development, reorganisation, clean-up, porting and other program maintenance tasks. The dynamic nature of Prolog makes the task non-trivial. Goals can be created dynamically using `call/1` after construction of a goal term. Abstract interpretation can find some of these calls, but they can also come from external communication, making it 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.38, and the XPCE library `pce_xref` provides a graphical front-end 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 the file search path. All loaded files that fall outside these categories are below the last branch called `/`. Files 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.

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 allow editing 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.

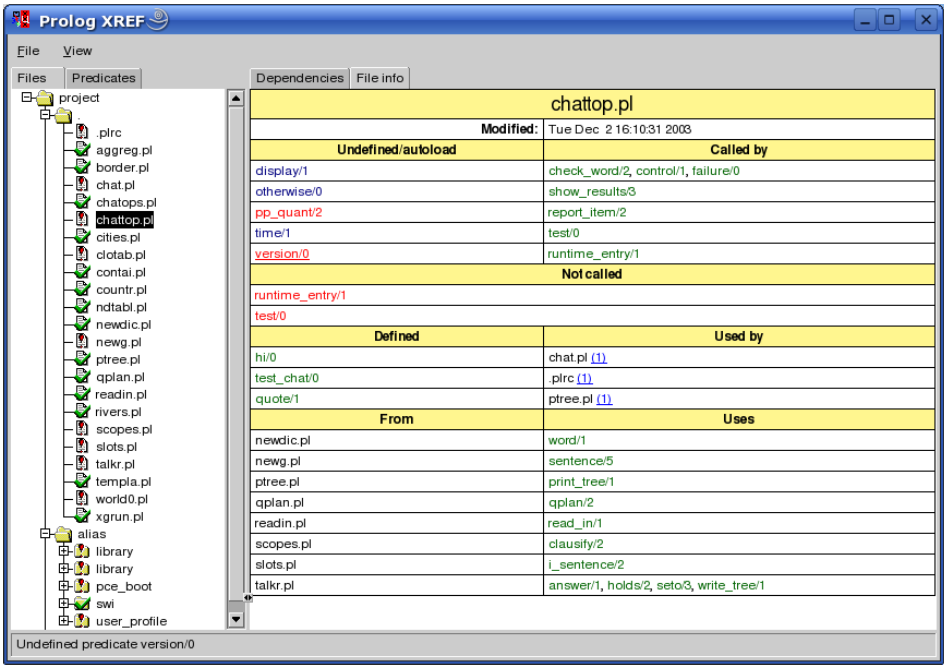

Figure 3.1: File info for `chattop.pl`, part of CHAT80

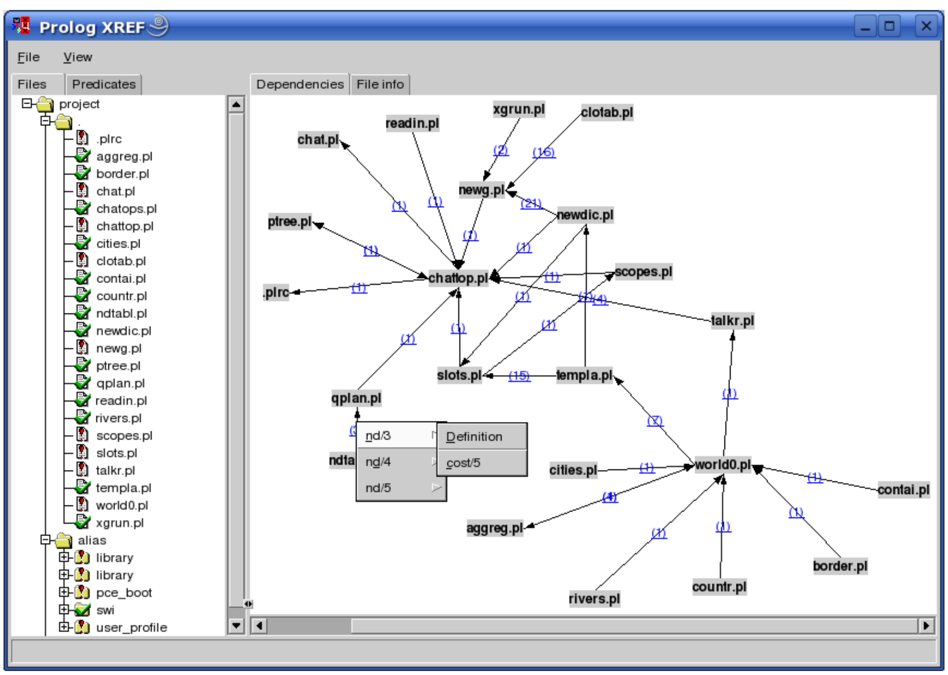

Figure 3.2: Dependencies between source files of CHAT80

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 Header menu command, 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 its 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 is 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 `@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
  : Open 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[7] completes a partial atom on the TAB key and shows alternatives on the command Alt-?.

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

- *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 Prolog Navigator.

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

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

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 *mode indicator*.

4.1.1 The argument mode indicator

An *argument mode indicator* gives information about the intended direction in which information carried by a predicate argument is supposed to flow. Mode indicators (and types) are not a formal part of the Prolog language but help in explaining intended semantics to the programmer. There is no complete agreement on argument mode indicators in the Prolog community. We use the following definitions:[1]

[1] These definitions are taken from the *PlDoc* markup language description. *PldDoc* markup is used for source code markup (as well as for the commenting tool). The current manual has only one mode declaration per predicate and therefore predicates with mode (+,−) and (−,+) are described as `(?,?)`. The `@`-mode is often replaced by
chr+.

++	At call time, the argument must be *ground*, i.e., the argument may not contain any variables that are still unbound.
+	At call time, the argument must be instantiated to a term satisfying some (informal) type specification. The argument need not necessarily be ground. For example, the term `[_]` is a list, although its only member is the anonymous variable, which is always unbound (and thus nonground).
-	Argument is an *output* argument. It may or may not be bound at call-time. If the argument is bound at call time, the goal behaves as if the argument were unbound, and then unified with that term after the goal succeeds. This is what is called being *steadfast*: instantiation of output arguments at call-time does not change the semantics of the predicate, although optimizations may be performed. For example, the goal `findall(X, Goal, [T])` is good style and equivalent to `findall(X, Goal, Xs), Xs = [T]`[2] Note that any *determinism* specification, e.g., `det`, only applies if the argument is unbound. For the case where the argument is bound or involved in constraints, `det` effectively becomes `semidet`, and `multi` effectively becomes `nondet`.
–	At call time, the argument must be unbound. This is typically used by predicates that create 'something' and return a handle to the created object, such as `open/3`, which creates a *stream*.
?	At call time, the argument must be bound to a *partial term* (a term which may or may not be ground) satisfying some (informal) type specification. Note that an unbound variable *is* a partial term. Think of the argument as either providing input or accepting output or being used for both input and output. For example, in `stream_property(S, reposition(Bool))`, the `reposition` part of the term provides input and the unbound-at-call-time *Bool* variable accepts output.
:	Argument is a *meta-argument*, for example a term that can be called as goal. The predicate is thus a *meta-predicate*. This flag implies `+`.
@	Argument will not be further instantiated than it is at call-time. Typically used for type tests.
!	Argument contains a mutable structure that may be modified using `setarg/3` or `nb_setarg/3`.

See also section 4.8 for examples of meta-predicates, and section 6.5 for mode flags to label meta-predicate arguments in module export declarations.

4.1.2 Predicate indicators

Referring to a predicate in running text is done using a *predicate indicator*. The canonical and most generic form of a predicate indicator is a term `[`⟨*module*⟩`:]`⟨*name*⟩`/`⟨*arity*⟩. The module is generally omitted if it is irrelevant (case of a built-in predicate) or if it can be inferred from context.

Non-terminal indicators

Compliant to the ISO standard draft on Definite Clause Grammars (see section 4.13), SWI-Prolog also allows for the *non-terminal indicator* to refer to a *DCG grammar rule*. The non-terminal indicator is written as `[`⟨*module*⟩`]:`⟨*name*⟩`//`⟨*arity*⟩.

A non-terminal indicator ⟨*name*⟩`//`⟨*arity*⟩ is understood to be equivalent to ⟨*name*⟩`/`⟨*arity*⟩`+2`, regardless of whether or not the referenced predicate is defined or can be used as a grammar rule.[3] The `//`-notation can be used in all places that traditionally allow for a predicate indicator, e.g., the module declaration, `spy/1`, and `dynamic/1`.

4.1.3 Predicate behaviour and determinism

To describe the general behaviour of a predicate, the following vocabulary is employed. In source code, structured comments contain the corresponding keywords:

`det`	A *deterministic* predicate always succeeds exactly once and does not leave a choicepoint.
`semidet`	A *semi-deterministic* predicate succeeds at most once. If it succeeds it does not leave a choicepoint.
`nondet`	A *non-deterministic* predicate is the most general case and no claims are made on the number of solutions (which may be zero, i.e., the predicate may *fail*) and whether or not the predicate leaves an choicepoint on the last solution.
`multi`	As `nondet`, but succeeds at least once.
`undefined`	Well founded semantics third value. See `undefined/0`.

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-bit 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.

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 natural representation, hiding encoding problems from the programmer as well as providing much easier debugging.

[3]This, however, makes a specific assumption about the implementation of DCG rules, namely that DCG rules are preprocessed into standard Prolog rules taking two additional arguments, the input list and the output list, in accumulator style. This *need* not be true in all implementations.

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

In SWI-Prolog, character codes are *always* the Unicode equivalent of the encoding. That is, if `get_code/1` reads from a stream encoded as `KOI8-R` (used for the Cyrillic alphabet), it returns the corresponding Unicode code points. Similarly, assembling or disassembling atoms using `atom_codes/2` interprets the codes as Unicode points. See section 2.19.1 for details.

To ease the pain of the two character representations (code and char), 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 character is an `a`.

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

The two character 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 there are `atom_chars/2`, `atom_codes/2`, `number_chars/2` and `number_codes/2`. For character I/O on streams there are `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` 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 Prolog clauses and directives, but no *module declaration* (see `module/1`). They are normally loaded using `consult/1` or `ensure_loaded/1`. Currently, a non-module file can only be loaded into a single module.[4]

A module Prolog source file starts with a module declaration. The subsequent Prolog code is loaded into the specified module, and only the *exported* predicates are made available to the context loading the module. Module files are normally loaded with `use_module/[1,2]`. See chapter 6 for details.

An include Prolog source file is loaded using the `include/1` directive, textually including Prolog text into another Prolog source. A file may be included into multiple source files and is typically used to share *declarations* such as multifile or dynamic between source files.

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

[4]This limitation may be lifted in the future. Existing limitations in SWI-Prolog's source code administration make this non-trivial.

```
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 multifile predicates `term_expansion/2`, `term_expansion/4`, `goal_expansion/2` and `goal_expansion/4`. 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.[5] Code that needs to create additional clauses must use `compile_aux_clauses/1`. See `library(apply_macros)` for an example.

A *directive* is an instruction to the compiler. Directives are used to set (predicate) properties (see section 4.15), set flags (see `set_prolog_flag/2`) and load files (this section). Directives are terms of the form `:-` ⟨*term*⟩.. Here are some examples:

```
:- use_module(library(lists)).
:- dynamic
        store/2.                    % Name, Value
```

The directive `initialization/1` can be used to run arbitrary Prolog goals. The specified goal is started *after* loading the file in which it appears has completed.

SWI-Prolog compiles code as it is read from the file, and directives are executed as *goals*. This implies that directives may call any predicate that has been defined before the point where the directive appears. It also accepts `?-` ⟨*term*⟩. as a synonym.

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.

Advanced topics are handled in subsequent sections: mutually dependent files (section 4.3.2), multithreaded loading (section 4.3.2) and reloading running code (section 4.3.2).

The core of the family of loading predicates is `load_files/2`. The predicates `consult/1`, `ensure_loaded/1`, `use_module/1`, `use_module/2` and `reexport/1` pass the file argument directly to `load_files/2` and pass additional options as expressed in the table 4.1:

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

Predicate	if	must_be_module	import
consult/1	true	false	all
ensure_loaded/1	not_loaded	false	all
use_module/1	not_loaded	true	all
use_module/2	not_loaded	true	specified
reexport/1	not_loaded	true	all
reexport/2	not_loaded	true	specified

Table 4.1: Properties of the file-loading predicates. The *import* column specifies what is imported if the loaded file is a module file.

load_files(*:Files*)
Equivalent to load_files(*Files, []*). Same as consult/1, See load_files/2 for supported options.

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 that 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.

check_script(*Bool*)
If false (default true), do not check the first character to be # and skip the first line when found.

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.

dialect(*+Dialect*)
Load *Files* with enhanced compatibility with the target Prolog system identified by *Dialect*. See expects_dialect/1 and section C for details.

encoding(*Encoding*)
Specify the way characters are encoded in the file. Default is taken from the Prolog flag encoding. See section 2.19.1 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.

format(+*Format*)
Used to specify the file format if data is loaded from a stream using the `stream`(*Stream*) option. Default is `source`, loading Prolog source text. If `qlf`, load QLF data (see `qcompile/1`).

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, and `not_loaded` loads the file if it was not loaded before.

imports(*Import*)
Specify what to import from the loaded module. The default for `use_module/1` is `all`. *Import* is passed from the second argument of `use_module/2`. Traditionally it is a list of predicate indicators to import. As part of the SWI-Prolog/YAP integration, we also support *Pred* as *Name* to import a predicate under another name. Finally, *Import* can be the term `except`(*Exceptions*), where *Exceptions* is a list of predicate indicators that specify predicates that are *not* imported or *Pred* as *Name* terms to denote renamed predicates. See also `reexport/2` and `use_module/2`.[6]

If *Import* equals `all`, all operators are imported as well. Otherwise, operators are *not* imported. Operators can be imported selectively by adding terms `op`(*Pri,Assoc,Name*) to the *Import* list. If such a term is encountered, all exported operators that unify with this term are imported. Typically, this construct will be used with all arguments unbound to import all operators or with only *Name* bound to import a particular operator.

modified(*TimeStamp*)
Claim that the source was loaded at *TimeStamp* without checking the source. This option is intended to be used together with the `stream`(*Input*) option, for example after extracting the time from an HTTP server or database.

module(+*Module*)
Load the indicated file into the given module, overruling the module name specified in the `:- module(Name, ...)` directive. This currently serves two purposes: (1) allow loading two module files that specify the same module into the same process and force and (2): force loading source code in a specific module, even if the code provides its own module name. Experimental.

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

qcompile(*Atom*)
How to deal with quick-load-file compilation by `qcompile/1`. Values are:

never
Default. Do not use qcompile unless called explicitly.

auto
Use qcompile for all writeable files. See comment below.

large
Use qcompile if the file is 'large'. Currently, files larger than 100 Kbytes are considered large.

[6]BUG: *Name/Arity* as *NewName* is currently implemented using a *link clause*. This harms efficiency and does not allow for querying the relation through `predicate_property/2`.

part
If `load_files/2` appears in a directive of a file that is compiled into Quick Load Format using `qcompile/1`, the contents of the argument files are included in the `.qlf` file instead of the loading directive.

If this option is not present, it uses the value of the Prolog flag `qcompile` as default.

optimise(*+Boolean*)
Explicitly set the optimization for compiling this module. See `optimise`.

redefine_module(*+Action*)
Defines what to do if a file is loaded that provides a module that is already loaded from another file. *Action* is one of `false` (default), which prints an error and refuses to load the file, or `true`, which uses `unload_file/1` on the old file and then proceeds loading the new file. Finally, there is `ask`, which starts interaction with the user. `ask` is only provided if the stream `user_input` is associated with a terminal.

reexport(*Bool*)
If `true` re-export the imported predicate. Used by `reexport/1` and `reexport/2`.

register(*Bool*)
If `false`, do not register the load location and options. This option is used by `make/0` and `load_hotfixes/1` to avoid polluting the load-context database. See `source_file_property/2`.

sandboxed(*Bool*)
Load the file in *sandboxed* mode. This option controls the flag `sandboxed_load`. The only meaningful value for *Bool* is `true`. Using `false` while the Prolog flag is set to `true` raises a permission error.

scope_settings(*Bool*)
Scope `style_check/1` and `expects_dialect/1` to the file and files loaded from the file after the directive. Default is `true`. The system and user initialization files (see `-f` and `-F`) are loading with `scope_settings`(*false*).

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 to remove all clauses if the data is reconsulted.

This option is added to allow compiling from non-file locations such as databases, the web, the *user* (see `consult/1`) or other servers. It can be combined with `format`(*qlf*) to load QLF data from a stream.

The `load_files/2` predicate can be hooked to load other data or data from objects other than files. See `prolog_load_file/2` for a description and `http/http_load` for an example. All hooks for `load_files/2` are documented in section B.9.

consult(*:File*)
Read *File* as a Prolog source file. Calls to `consult/1` may be abbreviated by just typing a number of filenames in a list. Examples:

```
?- consult(load).          % consult load or load.pl
?- [library(lists)].       % load library lists
?- [user].                 % Type program on the terminal
```

The predicate `consult/1` is equivalent to `load_files(File, [])`, except for handling the special file `user`, which reads clauses from the terminal. See also the `stream`(*Input*) option of `load_files/2`. Abbreviation using `?- [file1,file2].` does *not* work for the empty list (`[]`). This facility is implemented by defining the list as a predicate. Applications may only rely on using the list abbreviation at the Prolog toplevel and in directives.

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 this semantics, we hope to get as close as 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)]).`[7]

include(*+File*) *[ISO]*

Textually include the content of *File* at the position where the *directive* `:- include(File).` appears. The include construct is only honoured if it appears as a directive in a source file. *Textual* include (similar to C/C++ #include) is obviously useful for sharing declarations such as `dynamic/1` or `multifile/1` by including a file with directives from multiple files that use these predicates.

Textually including files that contain *clauses* is less obvious. Normally, in SWI-Prolog, clauses are *owned* by the file in which they are defined. This information is used to *replace* the old definition after the file has been modified and is reloaded by, e.g., `make/0`. As we understand it, `include/1` is intended to include the same file multiple times. Including a file holding clauses multiple times into the same module is rather meaningless as it just duplicates the same clauses. Including a file holding clauses in multiple modules does not suffer from this problem, but leads to multiple equivalent *copies* of predicates. Using `use_module/1` can achieve the same result while *sharing* the predicates.

If `include/1` is used to load files holding clauses, and if these files are loaded only once, then these `include/1` directives can be replaced by other predicates (such as `consult/1`). However, there are several cases where either `include/1` has no alternative, or using any alternative also requires other changes. An example of the former is using `include/1` to share directives. An example of the latter are cases where clauses of different predicates are distributed over multiple files: If these files are loaded with `include/1`, the directive `discontiguous/1` is appropriate, whereas if they are consulted, one must use the directive `multifile/1`.

To accommodate included files holding clauses, SWI-Prolog distinguishes between the source location of a clause (in this case the included file) and the *owner* of a clause (the file that includes

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

the file holding the clause). The source location is used by, e.g., `edit/1`, the graphical tracer, etc., while the owner is used to determine which clauses are removed if the file is modified. Relevant information is found with the following predicates:

- `source_file/2` describes the owner relation.
- `predicate_property/2` describes the source location (of the first clause).
- `clause_property/2` provides access to both source and ownership.
- `source_file_property/2` can be used to query include relationships between files.

require(+*Predicates*)
Declare that this file/module requires the specified predicates to be defined "with their commonly accepted definition". *Predicates* is either a list of predicate indicators or a *comma-list* of predicate indicators. First, all built-in predicates are removed from the set. The remaining predicates are searched using the library index used for autoloading and mapped to a set of `autoload/2` directives. This implies that the targets will be loaded lazily if autoloading is not completely disabled and loaded using `use_module/2` otherwise. See `autoload`.

The `require/1` directive provides less control over the exact nature and location of the predicate. As `autoload/2`, it prevents a local definition of this predicate. As SWI-Prolog guarantees that the set of built-in predicates and predicates available for autoloading is unambiguous (i.e., has no duplicates) the specification is unambiguous. It provides four advantages over `autoload/2`: (1) the user does not have to remember the exact library, (2) the directive can be supported in other Prolog systems[8], providing compatibility despite differences in library and built-in predicate organization, (3) it is robust against changes to the SWI-Prolog libraries and (4) it is less typing.

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 2.19.1.

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/1` or one of its derivatives. The predicate `make/0` is called after `edit/1`, automatically reloading all modified files. If 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 2.14) and runs `list_undefined/0` from the `check` library to report on undefined predicates.

library_directory(?*Atom*)
Dynamic predicate used to specify library directories. Defaults to `app_config(`*lib*`)` (see `file_search_path/2`) and the system's library (in this order) are defined. The user may add library directories using `assertz/1`, `asserta/1` or remove system defaults using `retract/1`. Deprecated. New code should use `file_search_path/2`.

[8] SICStus provides it

file_search_path(*+Alias, -Path*)

Dynamic multifile hook predicate used to specify 'path aliases'. This hook is called by `absolute_file_name/3` to search files specified as `Alias`(*Name*), e.g., `library`(*lists*). This feature is best described using an example. Given the definition:

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

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

Below is the initial definition of the file search path. This path implies `swi(`⟨*Path*⟩`)` and 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 `use_foreign_library/1`.

```
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)).
user:file_search_path(path, Dir) :-
    getenv('PATH', Path),
    (   current_prolog_flag(windows, true)
    ->  atomic_list_concat(Dirs, (;), Path)
    ;   atomic_list_concat(Dirs, :, Path)
    ),
    member(Dir, Dirs).
user:file_search_path(user_app_data, Dir) :-
    '$xdg_prolog_directory'(data, Dir).
user:file_search_path(common_app_data, Dir) :-
    '$xdg_prolog_directory'(common_data, Dir).
user:file_search_path(user_app_config, Dir) :-
    '$xdg_prolog_directory'(config, Dir).
user:file_search_path(common_app_config, Dir) :-
    '$xdg_prolog_directory'(common_config, Dir).
user:file_search_path(app_data, user_app_data('.')).
user:file_search_path(app_data, common_app_data('.')).
user:file_search_path(app_config, user_app_config('.')).
user:file_search_path(app_config, common_app_config('.')).
```

The `'$xdg_prolog_directory'/2` uses either the XDG Base Directory or `win_folder/2` on Windows. On Windows, user config is mapped to roaming appdata (CSIDL_APPDATA), user data to the non-roaming (CSIDL_LOCAL_APPDATA) and common data to (CSIDL_COMMON_APPDATA).

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*) *[nondet]*
Unifies *Path* with all possible expansions of the filename 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, and *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 can add extensions for Prolog source files to avoid conflicts (for example with `perl`) as well as to be compatible with another Prolog implementation. We suggest using `.pro` for avoiding conflicts with `perl`. Overriding the system definitions can stop the system from finding libraries.

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*)
True if the predicate specified by *Pred* is owned by 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. If *Pred* is a *multifile* predicate this predicate succeeds for all files that contribute clauses to *Pred*.[9] See also `clause_property/2`. Note that the relation between files and predicates is more complicated if `include/1` is used. The predicate describes the *owner* of the predicate. See `include/1` for details.

source_file_property(*?File, ?Property*)
True when *Property* is a property of the loaded file *File*. If *File* is non-var, it can be a file specification that is valid for `load_files/2`. Defined properties are:

derived_from(*Original, OriginalModified*)
File was generated from the file *Original*, which was last modified at time *OriginalModified* at the time it was loaded. This property is available if *File* was loaded using the `derived_from`(*Original*) option to `load_files/2`.

[9]The current implementation performs a linear scan through all clauses to establish this set of files.

includes(*IncludedFile, IncludedFileModified*)
: *File* used `include/1` to include *IncludedFile*. The last modified time of *IncludedFile* was *IncludedFileModified* at the time it was included.

included_in(*MasterFile, Line*)
: *File* was included into *MasterFile* from line *Line*. This is the inverse of the `includes` property.

load_context(*Module, Location, Options*)
: *Module* is the module into which the file was loaded. If *File* is a module, this is the module into which the exports are imported. Otherwise it is the module into which the clauses of the non-module file are loaded. *Location* describes the file location from which the file was loaded. It is either a term ⟨*file*⟩:⟨*line*⟩ or the atom `user` if the file was loaded from the terminal or another unknown source. *Options* are the options passed to `load_files/2`. Note that all predicates to load files are mapped to `load_files/2`, using the option argument to specify the exact behaviour.

load_count(*-Count*)
: *Count* is the number of times the file have been loaded, i.e., 1 (one) if the file has been loaded once.

modified(*Stamp*)
: File modification time when *File* was loaded. This is used by `make/0` to find files whose modification time is different from when it was loaded.

source(*Source*)
: One of `file` if the source was loaded from a file, `resource` if the source was loaded from a resource or `state` if the file was included in the saved state.

module(*Module*)
: *File* is a module file that declares the module *Module*.

number_of_clauses(*Count*)
: *Count* is the number of clauses associated with *File*. Note that clauses loaded from included files are counted as part of the main file.

reloading
: Present if the file is currently being **re**loaded.

exists_source(+*Source*) *[semidet]*

True if *Source* (a term valid for `load_files/2`) exists. Fails without error if this is not the case. The predicate is intended to be used with *conditional compilation* (see section 4.3.1 For example:

```
:- if(exists_source(library(error))).
:- use_module_library(error).
:- endif.
```

The implementation uses `absolute_file_name/3` using `file_type`(*prolog*).

exists_source(+*Source, -File*) *[semidet]*

As `exists_source/1`, binding *File* to an atom describing the full absolute path to the source file.

unload_file(+*File*)
Remove all clauses loaded from *File*. If *File* loaded a module, clear the module's export list and disassociate it from the file. *File* is a canonical filename or a file indicator that is valid for `load_files/2`.

This predicate should be used with care. The multithreaded nature of SWI-Prolog makes removing static code unsafe. Attempts to do this should be reserved for development or situations where the application can guarantee that none of the clauses associated to *File* are active.

prolog_load_context(*?Key, ?Value*)
Obtain context information during compilation. This predicate can be used from directives appearing in a source file to get information about the file being loaded as well as by the `term_expansion/2` and `goal_expansion/2` hooks. See also `source_location/2` and `if/1`. The following keys are defined:

Key	**Description**
`directory`	Directory in which `source` lives
`dialect`	Compatibility mode. See `expects_dialect/1`.
`file`	Similar to `source`, but returns the file being included when called while an include file is being processed
`module`	Module into which file is loaded
`reload`	`true` if the file is being **re**loaded. Not present on first load
`script`	Boolean that indicates whether the file is loaded as a script file (see `-s`)
`source`	File being loaded. If the system is processing an included file, the value is the *main* file. Returns the original Prolog file when loading a `.qlf` file.
`stream`	Stream identifier (see `current_input/1`)
`term_position`	Start position of last term read. See also `stream_property/2` (`position` property and `stream_position_data/3`.[10]
`term`	Term being expanded by `expand_term/2`.
`variable_names`	A list of '*Name* = *Var*' of the last term read. See `read_term/2` for details.

The `directory` is commonly used to add rules to `file_search_path/2`, setting up a search path for finding files with `absolute_file_name/3`. For example:

```
:- dynamic user:file_search_path/2.
:- multifile user:file_search_path/2.

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

    ...
    absolute_file_name(my_program_home('README.TXT'), ReadMe,
                       [ access(read) ]),
    ...
```

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`.

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. Hooks may call `cancel_halt/1`, causing `halt/0` and `PL_halt(0)` to print a message indicating that halting the system has been cancelled.

cancel_halt(*+Reason*)
If this predicate is called from a hook registered with `at_halt/1`, halting Prolog is cancelled and an informational message is printed that includes *Reason*. This is used by the development tools to cancel halting the system if the editor has unsaved data and the user decides to cancel.

:- initialization(*:Goal*) *[ISO]*
Call *Goal* *after* loading the source file in which this directive appears has been completed. In addition, *Goal* is executed if a saved state created using `qsave_program/1` is restored.

The ISO standard only allows for using `:- Term` if *Term* is a *directive*. This means that arbitrary goals can only be called from a directive by means of the `initialization/1` directive. SWI-Prolog does not enforce this rule.

The `initialization/1` directive must be used to do program initialization in saved states (see `qsave_program/1`). A saved state contains the predicates, Prolog flags and operators present at the moment the state was created. Other resources (records, foreign resources, etc.) must be recreated using `initialization/1` directives or from the entry goal of the saved state.

Up to SWI-Prolog 5.7.11, *Goal* was executed immediately rather than after loading the program text in which the directive appears as dictated by the ISO standard. In many cases the exact moment of execution is irrelevant, but there are exceptions. For example, `load_foreign_library/1` must be executed immediately to make the loaded foreign predicates available for exporting. SWI-Prolog now provides the directive `use_foreign_library/1` to ensure immediate loading as well as loading after restoring a saved state. If the system encounters a directive `:- initialization(load_foreign_library(...))`, it will load the foreign library immediately and issue a warning to update your code. This behaviour can be extended by providing clauses for the multifile hook predicate `prolog:initialize_now`(*Term, Advice*), where *Advice* is an atom that gives advice on how to resolve the compatibility issue.

initialization(*:Goal, +When*)
Similar to `initialization/1`, but allows for specifying when *Goal* is executed while loading the program text:

now
Execute *Goal* immediately.

after_load
Execute *Goal* after loading the program text in which the directive appears. This is the same as `initialization/1`.

prepare_state
Execute *Goal* as part of `qsave_program/2`. This hook can be used for example to eagerly execute initialization that is normally done lazily on first usage.

restore_state
Do not execute *Goal* while loading the program, but *only* when restoring a saved state.[11]

program
Execute *Goal* once after executing the `-g` goals at program startup. Registered goals are executed in the order encountered and a failure or exception causes the Prolog to exit with non-zero exit status. These goals are *not* executed if the `-l` is given to merely *load* files. In that case they may be executed explicitly using `initialize/0`. See also section 2.11.2.

main
When Prolog starts, the last goal registered using `initialization`(*Goal, main*) is executed as main goal. If *Goal* fails or raises an exception, the process terminates with non-zero exit code. If not explicitly specified using the `-t` the *toplevel goal* is set to `halt/0`, causing the process to exit with status 0. An explicitly specified toplevel is executed normally. This implies that `-t prolog` causes the application to start the normal interactive toplevel after completing *Goal*. See also the Prolog flag `toplevel_goal` and section 2.11.2.

initialize *[det]*
Run all initialization goals registered using `initialization`(*Goal, program*). Raises an error `initialization_error`(*Reason, Goal, File:Line*) if *Goal* fails or raises an exception. *Reason* is `failed` or the exception raised.

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.

4.3.1 Conditional compilation and program transformation

ISO Prolog defines no way for program transformations such as macro expansion or conditional compilation. Expansion through `term_expansion/2` and `expand_term/2` can be seen as part of the de-facto standard. This mechanism can do arbitrary translation between valid Prolog terms read from the source file to Prolog terms handed to the compiler. As `term_expansion/2` can return a list, the transformation does not need to be term-to-term.

Various Prolog dialects provide the analogous `goal_expansion/2` and `expand_goal/2` that allow for translation of individual body terms, freeing the user of the task to disassemble each clause.

[11] Used to be called `restore`. `restore` is still accepted for backward compatibility.

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 than the read term (*Term1*). *Term2* may be a term of 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 6 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`.

It is possible to act on the beginning and end of a file by expanding the terms `begin_of_file` and `end_of_file`. The latter is supported by most Prolog systems that support term expansion as `read_term/3` returns `end_of_file` on reaching the end of the input. Expanding `begin_of_file` may be used to initialise the compilation, for example base on the file name extension. It was added in SWI-Prolog 8.1.1.

The current macro-expansion mechanism originates from Prolog systems in the 1980s and 1990s. It has several flaws, (1) the hooks act globally (except for definitions in a module), (2) it is hard to deal with interactions between transformations, (3) macros can not be reused between modules using the normal module export/import protocol and (4) it is hard to make source code aware tools such as the graphical debugger act properly in the context of macro expansion. Several Prolog implementations have tried to implement better expansion mechanisms. None of these solve all problems and all are largely incompatible with our current macro expansion. Future versions may provide a new mechanism to solve these issues.

Controlled interaction is provided between macro expansion defined in a module and the `user` and `system` modules. Here, SWI-Prolog uses a *pipeline* where the result of local module expansion is the input for the expansion in `user`, which is the input for the expansion in `system`. See also section 6.10.

Scoping, i.e., make a rule defined in a module only active if this module is imported into the module being compiled, can be emulated by defining the macro globally in the `user` module and using `prolog_load_context/2` and some logic to verify the macro expansion should apply. If (goal) expansion effectively defined *inlining* it is good practice to also define the predicate and have the macro expansion check that the predicate is in scope. Here is an example.

```
:- module(m1, [double/2]).

double(X, D) :- D is X*2.

user:goal_expansion(double(X,D), D is X*2) :-
    prolog_load_context(module, M),
    predicate_property(M:double(_,_), imported_from(m1)).
```

For term expansion that is not related to a specific predicate we can define a sentinel predicate rather than using the goal predicate and check it is imported into the current module to verify that the module that defines the expansion is imported into the current compilation context.

expand_term(*+Term1, -Term2*)

This predicate is normally called by the compiler on terms read from the input to perform preprocessing. It consists of four steps, where each step processes the output of the previous step.

1. Test conditional compilation directives and translate all input to `[]` if we are in a 'false branch' of the conditional compilation. See section 4.3.1.
2. Call `term_expansion/2`. This predicate is first tried in the module that is being compiled and then in modules from which this module inherits according to `default_module/2`. The output of the expansion in a module is used as input for the next module. Using the default setup and when compiling a normal application module *M*, this implies expansion is executed in *M*, `user` and finally in `system`. Library modules inherit directly from `system` and can thus not be re-interpreted by term expansion rules in `user`.
3. Call DCG expansion (`dcg_translate_rule/2`).
4. Call `expand_goal/2` on each body term that appears in the output of the previous steps.

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 follows the module inheritance path as defined by `default_module/2`, i.e., by default `user` and `system`. 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 default modules for *Module*.

Only goals appearing in the body of clauses when reading a source file are expanded using this mechanism, and only if they appear literally in the clause, or as an argument to a defined meta-predicate that is annotated using '0' (see `meta_predicate/1`). Other cases need a real predicate definition.

The expansion hook can use `prolog_load_context/2` to obtain information about the context in which the goal is expanded such as the module, variable names or the encapsulating term.

expand_goal(*+Goal1, -Goal2*)

This predicate is normally called by the compiler to perform preprocessing using `goal_expansion/2`. The predicate computes a fixed-point by applying transformations until there are no more changes. If optimisation is enabled (see `-O` and `optimise`), `expand_goal/2` simplifies the result by removing unneeded calls to `true/0` and `fail/0` as well as trivially unreachable branches.

If `goal_expansion/2` *wraps* a goal as in the example below the system still reaches fixed-point as it prevents re-expanding the expanded term while recursing. It does re-enable expansion on the *arguments* of the expanded goal as illustrated in `t2/1` in the example.[12]

```
:- meta_predicate run(0).

may_not_fail(test(_)).
may_not_fail(run(_)).

goal_expansion(G, (G *-> true ; error(goal_failed(G),_))) :-
    may_not_fail(G).

t1(X) :- test(X).
t2(X) :- run(run(X)).
```

Is expanded into

```
t1(X) :-
    (   test(X)
    *-> true
    ;   error(goal_failed(test(X)), _)
    ).

t2(X) :-
    (   run((run(X)*->true;error(goal_failed(run(X)), _)))
    *-> true
    ;   error(goal_failed(run(run(X))), _)
    ).
```

Note that goal expansion should not bind any variables in the clause. Doing so may impact the semantics of the clause if the variable is also used elsewhere. In the general case this is not verified. It is verified for `\+/1` and `;/2`, resulting in an exception.

compile_aux_clauses(+*Clauses*)
Compile clauses on behalf of `goal_expansion/2`. This predicate compiles the argument clauses into static predicates, associating the predicates with the current file but avoids changing the notion of current predicate and therefore discontiguous warnings.

Note that in some cases multiple expansions of similar goals can share the same compiled auxiliary predicate. In such cases, the implementation of `goal_expansion/2` can use `predicate_property/2` using the property `defined` to test whether the predicate is already defined in the current context.

dcg_translate_rule(+*In*, -*Out*)
This predicate performs the translation of a term `Head-->Body` into a normal Prolog clause. Normally this functionality should be accessed using `expand_term/2`.

[12] After discussion with Peter Ludemann and Paulo Moura on the forum.

var_property(+*Var, ?Property*)
True when *Property* is a property of *Var*. These properties are available during goal- and term-expansion. Defined properties are below. Future versions are likely to provide more properties, such as whether the variable is referenced in the remainder of the term. See also `goal_expansion/2`.

fresh(*Bool*)
Bool has the value `true` if the variable is guaranteed to be unbound at entry of the goal, otherwise its value is *false*. This implies that the variable first appears in this goal or a previous appearance was in a negation (`\+/1`) or a different branch of a disjunction.

singleton(*Bool*)
Bool has the value `true` if the variable is a *syntactic* singleton in the term it appears in. Note that this tests that the variable appears exactly once in the term being expanded without making any claim on the syntax of the variable. Variables that appear only once in multiple branches are *not* singletons according to this property. Future implementations may improve on that.

name(*Name*)
True when variable appears with the given name in the source.

Program transformation with source layout info

This sections documents extended versions of the program transformation predicates that also transform the source layout information. Extended layout information is currently processed, but unused. Future versions will use for the following enhancements:

- More precise locations of warnings and errors
- More reliable setting of breakpoints
- More reliable source layout information in the graphical debugger.

expand_goal(+*Goal1, ?Layout1, -Goal2, -Layout2*)
goal_expansion(+*Goal1, ?Layout1, -Goal2, -Layout2*)
expand_term(+*Term1, ?Layout1, -Term2, -Layout2*)
term_expansion(+*Term1, ?Layout1, -Term2, -Layout2*)

dcg_translate_rule(+*In, ?LayoutIn, -Out, -LayoutOut*)
These versions are called *before* their 2-argument counterparts. The input layout term is either a variable (if no layout information is available) or a term carrying detailed layout information as returned by the `subterm_positions` of `read_term/2`. The output layout should be a variable if no layout information can be computed for the expansion; a sub-term can also be a variable to indicate "don't know".

Conditional compilation

Conditional compilation builds on the same principle as `term_expansion/2`, `goal_expansion/2` and the expansion of grammar rules to compile sections of the source code conditionally. One of the reasons for introducing conditional compilation is to simplify writing portable code. See section C for more information. Here is a simple example:

```
:- if(\+source_exports(library(lists), suffix/2)).

suffix(Suffix, List) :-
        append(_, Suffix, List).

:- endif.
```

Note that these directives can only appear as separate terms in the input. Typical usage scenarios include:

- Load different libraries on different dialects.
- Define a predicate if it is missing as a system predicate.
- Realise totally different implementations for a particular part of the code due to different capabilities.
- Realise different configuration options for your software.

:- if(*:Goal*)
Compile subsequent code only if *Goal* succeeds. For enhanced portability, *Goal* is processed by `expand_goal/2` before execution. If an error occurs, the error is printed and processing proceeds as if *Goal* has failed.

:- elif(*:Goal*)
Equivalent to `:- else. :-if(Goal).` ... `:- endif.` In a sequence as below, the section below the first matching `elif` is processed. If no test succeeds, the else branch is processed.

```
:- if(test1).
section_1.
:- elif(test2).
section_2.
:- elif(test3).
section_3.
:- else.
section_else.
:- endif.
```

:- else
Start 'else' branch.

:- endif
End of conditional compilation.

4.3.2 Reloading files, active code and threads

Traditionally, Prolog environments allow for reloading files holding currently active code. In particular, the following sequence is a 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* executing the now fixed predicate using the **r** command

Reloading a previously loaded file is safe, both in the debug scenario above and when the code is being executed by another *thread*. Executing threads switch atomically to the new definition of modified predicates, while clauses that belong to the old definition are (eventually) reclaimed by `garbage_collect_clauses/0`.[13] Below we describe the steps taken for *reloading* a file to help understanding the limitations of the process.

1. If a file is being reloaded, a *reload context* is associated to the file administration. This context includes a table keeping track of predicates and a table keeping track of the module(s) associated with this source.

2. If a new predicate is found, an entry is added to the context predicate table. Three options are considered:

   (a) The predicate is new. It is handled the same as if the file was loaded for the first time.

   (b) The predicate is foreign or thread local. These too are treated as if the file was loaded for the first time.

   (c) Normal predicates. Here we initialise a pointer to the *current clause*.

3. New clauses for 'normal predicates' are considered as follows:

   (a) If the clause's byte-code is the same as the predicates current clause, discard the clause and advance the current clause pointer.

   (b) If the clause's byte-code is the same as some clause further into the clause list of the predicate, discard the new clause, mark all intermediate clauses for future deletion, and advance the current clause pointer to the first clause after the matched one.

   (c) If the clause's byte-code matches no clause, insert it for *future activation* before the current clause and keep the current clause.

4. *Properties* such as `dynamic` or `meta_predicate` are in part applied immediately and in part during the fixup process after the file completes loading. Currently, `dynamic` and `thread_local` are applied immediately.

5. New modules are recorded in the reload context. Export declarations (the module's public list and `export/1` calls) are both applied and recorded.

[13]As of version 7.3.12. Older versions wipe all clauses originating from the file before loading the new clauses. This causes threads that executes the code to (typically) die with an *undefined predicate* exception.

6. When the end-of-file is reached, the following fixup steps are taken
   (a) For each predicate
      i. The current clause and subsequent clauses are marked for future deletion.
      ii. All clauses marked for future deletion or creation are (in)activated by changing their 'erased' or 'created' *generation*. Erased clauses are (eventually) reclaimed by the *clause garbage collector*, see `garbage_collect_clauses/0`.
      iii. Pending predicate property changes are applied.
   (b) For each module
      i. Exported predicates that are not encountered in the reload context are removed from the export list.

The above generally ensures that changes to the *content* of source files can typically be activated safely using `make/0`. Global changes such as operator changes, changes of module names, changes to multi-file predicates, etc. sometimes require a restart. In almost all cases, the need for restart is indicated by permission or syntax errors during the reload or existence errors while running the program.

In some cases the content of a source file refers 'to itself'. This is notably the case if local rules for `goal_expansion/2` or `term_expansion/2` are defined or goals are executed using *directives*.[14]. Up to version 7.5.12 it was typically needed to reload the file *twice*, once for updating the code that was used for compiling the remainder of the file and once to effectuate this. As of version 7.5.13, conventional *transaction semantics* apply. This implies that for the thread performing the reload the file's content is first wiped and gradually rebuilt, while other threads see an *atomic* update from the old file content to the new.[15]

Errors and warnings during compilation

Errors and warnings reported while compiling a file are reported using `print_message/2`. Typical errors are syntax errors, errors during macro expansion by `term_expansion/2` and `goal_expansion/2`, compiler errors such as illegal clauses or an attempt to redefine a system predicate and errors caused by executing *directives*, notably using `initialization/1` and `initialization/2`.

Merely reporting error messages and warnings is typically desirable for interactive usage. Non-interactive applications often require to be notified of such issues, typically using the *exit code* of the process. We can distinguish two types of errors and warnings: (1) those resulting from loading an invalid program and (2) messages that result from running the program. A typical example is user code that wishes to try something and in case of an error report this and continue.

```
...,
E = error(_,_),
catch(do_something, E,
      print_message(error, E)),
...
```

[14]Note that `initialization/1` directives are executed *after* loading the file. SWI-Prolog allows for directives that are executed *while* loading the file using `:- Goal.` or `initialization/2`

[15]This feature was implemented by Keri Harris.

User code may be (and often is) started from directives, while running user code may involve compilation due to autoloading, loading of data files, etc. As a result, it is unclear whether an error message should merely be printed, should result in a non-zero exit status at the end or should immediately terminate the process.

The default behaviour is defined by the Prolog flags `on_error` and `on_warning`. It can be fine tuned by defining the *hook predicate* `message_hook/3`. The compiler calls `print_message/2` using the level `silent` and the message below if errors or warnings where printed during the execution of `load_files/2`.

load_file_errors(*File, Errors, Warnings*)
: Here, *File* is the raw file specification handed to `load_files/2`, i.e., `'myfile.pl'` or `library(lists)`, *Errors* is the number of errors printed while loading and *Warnings* is the number of warnings printed while loading. Note that these counts include messages from (initialization) directives.

This allows the user to fine tune the behaviour on errors and, for example, halt the process on a non-zero error count right after loading the file wth errors using the code below.

```
:- multifile user:message_hook/3.

user:message_hook(load_file_errors(_File, Errors, _Warnings),
                  _Level, _Lines) :-
    Errors > 0,
    halt(1).
```

Compilation of mutually dependent code

Large programs are generally split into multiple files. If file A accesses predicates from file B which accesses predicates from file A, we consider this a mutual or circular dependency. If traditional load predicates (e.g., `consult/1`) are used to include file B from A and A from B, loading either file results in a loop. This is because `consult/1` is mapped to `load_files/2` using the option `if(true)`(.) Such programs are typically loaded using a *load file* that consults all required (non-module) files. If modules are used, the dependencies are made explicit using `use_module/1` statements. The `use_module/1` predicate, however, maps to `load_files/2` with the option `if(not_loaded)`(.) A `use_module/1` on an already loaded file merely makes the public predicates of the used module available.

Summarizing, mutual dependency of source files is fully supported with no precautions when using modules. Modules can use each other in an arbitrary dependency graph. When using `consult/1`, predicate dependencies between loaded files can still be arbitrary, but the consult relations between files must be a proper tree.

Compilation with multiple threads

This section discusses compiling files for the first time. For reloading, see section 4.3.2.

Multiple threads can compile files concurrently. This requires special precautions only if multiple threads wish to load the same file at the same time. Therefore, `load_files/2` checks whether some other thread is already loading the file. If not, it starts loading the file. If a thread detects that another thread is already loading the file the thread blocks until the other thread finishes loading the file. After waiting, and if the file is a module file, it imports the exported predicates and operators from the module.

Note that this schema does not prevent deadlocks under all situations. Consider two mutually dependent (see section 4.3.2) module files A and B, where thread 1 starts loading A and thread 2 starts loading B at the same time. Both threads will deadlock when trying to load the used module.

The current implementation does not detect such cases and the involved threads will freeze. This problem can be avoided if a mutually dependent collection of files is always loaded from the same start file.

4.3.3 Quick load files

SWI-Prolog supports compilation of individual or multiple Prolog source files into 'Quick Load Files'. A 'Quick Load File' (`.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/1` is given an explicit `.pl` file, it will load the Prolog source. When given a `.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 file specification as `consult/1`, etc., and, in addition to the normal compilation, creates a *Quick Load File* from *File*. The file extension of this file is `.qlf`. The basename of the Quick Load File is the same as the input file.

  If the file contains '`:- consult(`*+File*`)`', '`:- [`*+File*`]`' or '`:- load_files(`*+File*`, [qcompile(part), ...])`' 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 2.11 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.

qcompile(*:File, +Options*)
: As `qcompile/1`, but processes additional options as defined by `load_files/2`.[16]

4.4 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 customized. See section 4.4.1.

[16]BUG: Option processing is currently incomplete.

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

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

edit(+*Specification*)
First, exploit `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 either the first `.pl` file from the commandline (the *associated* file, see the Prolog flag `associated_file` or the first script file specified using the `-s` or `-l` command line option. When using the Windows shell while SWI-Prolog is associated with the `.pl` extension this is the file loaded by double-clicking a `.pl` file. See also section 2.11.2.

4.4.1 Customizing the editor interface

The predicates described in this section are *hooks* that can be defined to disambiguate specifications given to `edit/1`, find the related source, and open an editor at the given source location.

prolog_edit:locate(+*Spec, -FullSpec, -Location*)
Where *Spec* is the specification provided through `edit/1`. This multifile predicate is used to enumerate locations where 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 basename 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 `customize/init.pl` for examples.

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

Determines how *Editor* is to be invoked using `shell/1`. *Editor* is the determined editor (see `prolog_edit:edit_source/1`), without the full path specification, and without a possible (`.exe`) extension. *Command* is an atom describing the command. The following %-sequences are replaced in *Command* before the result is handed to `shell/1`:

%e	Replaced by the (OS) command name of the editor
%f	Replaced by the (OS) full path name of the file
%d	Replaced by the line number

If the editor can deal with starting at a specified line, two clauses should be provided. The first pattern invokes the editor with a line number, while the second is used if the line number is unknown.

The default contains definitions for `vi`, `emacs`, `emacsclient`, `vim`, `notepad`* and `wordpad`*. Starred editors do not provide starting at a given line number.

Please contribute your specifications to `bugs@swi-prolog.org`.

prolog_edit:load

Normally an undefined 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)).
```

4.5 Verify Type of a Term

Type tests are semi-deterministic predicates that succeed if the argument satisfies the requested type. Type-test predicates have no error condition and do not instantiate their argument. See also library `error`.

var(@*Term***)** *[ISO]*

True if *Term* currently is a free variable.

nonvar(*@Term*) *[ISO]*
True if *Term* currently is not a free variable.

integer(*@Term*) *[ISO]*
True if *Term* is bound to an integer.

float(*@Term*) *[ISO]*
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*) *[ISO]*
True if *Term* is bound to a rational number (including integers) or a floating point number.

atom(*@Term*) *[ISO]*
True if *Term* is bound to an atom.

blob(*@Term, ?Type*)
True if *Term* is a *blob* of type *Type*. See section 12.4.8.

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 5.2. Starting with version 7, the syntax for a string object is text between double quotes, such as `"hello"`.[17] See also the Prolog flag `double_quotes`.

atomic(*@Term*) *[ISO]*
True if *Term* is bound (i.e., not a variable) and is not compound. Thus, atomic acts as if defined by:

```
atomic(Term) :-
        nonvar(Term),
        \+ compound(Term).
```

SWI-Prolog defines the following atomic datatypes: atom (`atom/1`), string (`string/1`), integer (`integer/1`), floating point number (`float/1`) and blob (`blob/2`). In addition, the symbol `[]` (empty list) is atomic, but not an atom. See section 5.1.

compound(*@Term*) *[ISO]*
True if *Term* is bound to a compound term. See also `functor/3` =../2, `compound_name_arity/3` and `compound_name_arguments/3`.

[17] In traditional Prolog systems, double quoted text is often mapped to a list of *character codes*.

callable(*@Term*) *[ISO]*

True if *Term* is bound to an atom or a compound term. This was intended as a type-test for arguments to `call/1`, `call/2` etc. Note that callable only tests the *surface term*. Terms such as (22,true) are considered callable, but cause `call/1` to raise a type error. Module-qualification of meta-argument (see `meta_predicate/1`) using `:/2` causes callable to succeed on any meta-argument.[18] Consider the program and query below:

```
:- meta_predicate p(0).

p(G) :- callable(G), call(G).

?- p(22).
ERROR: Type error: `callable' expected, found `22'
ERROR: In:
ERROR:    [6] p(user:22)
```

ground(*@Term*) *[ISO]*

True if *Term* holds no free variables. See also `nonground/2` and `term_variables/2`.

cyclic_term(*@Term*)

True if *Term* contains cycles, i.e. is an infinite term. See also `acyclic_term/1` and section 2.17.[19]

acyclic_term(*@Term*) *[ISO]*

True if *Term* does not contain cycles, i.e. can be processed recursively in finite time. See also `cyclic_term/1` and section 2.17.

4.6 Comparison and Unification of Terms

Although unification is mostly done implicitly while matching the head of a predicate, it is also provided by the predicate =/2.

?Term1 **=** *?Term2* *[ISO]*

Unify *Term1* with *Term2*. True if the unification succeeds. It acts as if defined by the following fact:

```
=(Term, Term).
```

For behaviour on cyclic terms see the Prolog flag `occurs_check`. Calls to `=/2` in a clause body are compiled and may be (re)moved depending on the Prolog flag `optimise_unify`. See also section 2.18.3.

[18] We think that `callable/1` should be deprecated and there should be two new predicates, one performing a test for callable that is minimally module aware and possibly consistent with type-checking in `call/1` and a second predicate that tests for atom or compound.

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

@Term1 **\=** *@Term2* *[ISO]*

Equivalent to `\+Term1 = Term2`.

This predicate is logically sound if its arguments are sufficiently instantiated. In other cases, such as `?- X \= Y.`, the predicate fails although there are solutions. This is due to the incomplete nature of `\+/1`.

To make your programs work correctly also in situations where the arguments are not yet sufficiently instantiated, use `dif/2` instead.

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* < *Strings* < *Atoms* < *Compound Terms*
2. Variables are sorted by address.
3. *Numbers* are compared by value. Mixed integer/float are compared as floats. If the comparison is equal, the float is considered the smaller value. If the Prolog flag `iso` is defined, all floating point numbers precede all integers.
4. *Strings* are compared alphabetically.
5. *Atoms* are compared alphabetically.
6. *Compound* terms are first checked on their arity, then on their functor name (alphabetically) and finally recursively on their arguments, leftmost argument first.

Although variables are ordered, there are some unexpected properties one should keep in mind when relying on variable ordering. This applies to the predicates below as to predicate such as `sort/2` as well as libraries that reply on ordering such as library `assoc` and library `ordsets`. Obviously, an established relation *A* `@<` *B* no longer holds if *A* is unified with e.g., a number. Also unifying *A* with *B* invalidates the relation because they become equivalent (==/2) after unification.

As stated above, variables are sorted by address, which implies that they are sorted by 'age', where 'older' variables are ordered before 'newer' variables. If two variables are unified their 'shared' age is the age of oldest variable. This implies we can examine a list of sorted variables with 'newer' (fresh) variables without invalidating the order. Attaching an *attribute*, see section 8.1, turns an 'old' variable into a 'new' one as illustrated below. Note that the first always succeeds as the first argument of a term is always the oldest. This only applies for the *first* attribute, i.e., further manipulation of the attribute list does *not* change the 'age'.

```
?- T = f(A,B), A @< B.
T = f(A, B).

?- T = f(A,B), put_attr(A, name, value), A @< B.
false.
```

The above implies you *can* use e.g., an assoc (from library `assoc`, implemented as an AVL tree) to maintain information about a set of variables. You must be careful about what you do with the attributes though. In many cases it is more robust to use attributes to register information about variables.

@*Term1* **==** @*Term2* *[ISO]*
: True if *Term1* is equivalent to *Term2*. A variable is only identical to a sharing variable.

@*Term1* **\==** @*Term2* *[ISO]*
: Equivalent to `\+Term1 == Term2`.

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

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

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

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

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

4.6.2 Special unification and comparison predicates

This section describes special purpose variations on Prolog unification. The predicate `unify_with_occurs_check/2` provides sound unification and is part of the ISO standard. The predicate `subsumes_term/2` defines 'one-sided unification' and is part of the ISO proposal established in Edinburgh (2010). Finally, `unifiable/3` is a 'what-if' version of unification that is often used as a building block in constraint reasoners.

unify_with_occurs_check(*+Term1, +Term2*) *[ISO]*
: 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 queries below.

```
1 ?- A = f(A).
A = f(A).
2 ?- unify_with_occurs_check(A, f(A)).
false.
```

The first statement creates a *cyclic term*, also called a *rational tree*. The second executes logically sound unification and thus fails. Note that the behaviour of unification through `=/2` as well as implicit unification in the head can be changed using the Prolog flag `occurs_check`.

The SWI-Prolog implementation of `unify_with_occurs_check/2` is cycle-safe and only guards against *creating* cycles, not against cycles that may already be present in one of the arguments. This is illustrated in the following two queries:

```
?- X = f(X), Y = X, unify_with_occurs_check(X, Y).
X = Y, Y = f(Y).
?- X = f(X), Y = f(Y), unify_with_occurs_check(X, Y).
X = Y, Y = f(Y).
```

Some other Prolog systems interpret `unify_with_occurs_check/2` as if defined by the clause below, causing failure on the above two queries. Direct use of `acyclic_term/1` is portable and more appropriate for such applications.

```
unify_with_occurs_check(X,X) :- acyclic_term(X).
```

+Term1 **=@=** *+Term2*

True if *Term1* is a *variant* of (or *structurally equivalent* to) *Term2*. Testing for a variant is weaker than equivalence (`==/2`), but stronger than unification (`=/2`). Two terms A and B are variants iff there exists a renaming of the variables in A that makes A equivalent (==) to B and vice versa.[20] Examples:

1	`a =@= A`	false
2	`A =@= B`	true
3	`x(A,A) =@= x(B,C)`	false
4	`x(A,A) =@= x(B,B)`	true
5	`x(A,A) =@= x(A,B)`	false
6	`x(A,B) =@= x(C,D)`	true
7	`x(A,B) =@= x(B,A)`	true
8	`x(A,B) =@= x(C,A)`	true

A term is always a variant of a copy of itself. Term copying takes place in, e.g., `copy_term/2`, `findall/3` or proving a clause added with `asserta/1`. In the pure Prolog world (i.e., without attributed variables), `=@=/2` behaves as if defined below. With attributed variables, variant of the attributes is tested rather than trying to satisfy the constraints.

```
A =@= B :-
        copy_term(A, Ac),
        copy_term(B, Bc),
        numbervars(Ac, 0, N),
        numbervars(Bc, 0, N),
        Ac == Bc.
```

The SWI-Prolog implementation is cycle-safe and can deal with variables that are shared between the left and right argument. Its performance is comparable to `==/2`, both on success and (early) failure. [21]

[20] Row 7 and 8 of this table may come as a surprise, but row 8 is satisfied by (left-to-right) $A \rightarrow C$, $B \rightarrow A$ and (right-to-left) $C \rightarrow A$, $A \rightarrow B$. If the same variable appears in different locations in the left and right term, the variant relation can be broken by consistent binding of both terms. E.g., after binding the first argument in row 8 to a value, both terms are no longer variant.

[21] The current implementation is contributed by Kuniaki Mukai.

This predicate is known by the name `variant/2` in some other Prolog systems. Be aware of possible differences in semantics if the arguments contain attributed variables or share variables.[22]

+Term1 **\=@=** *+Term2*

Equivalent to `` `\+Term1 =@= Term2' ``. See `=@=/2` for details.

subsumes_term(*@Generic, @Specific*) *[ISO]*

True if *Generic* can be made equivalent to *Specific* by only binding variables in *Generic*. The current implementation performs the unification and ensures that the variable set of *Specific* is not changed by the unification. On success, the bindings are undone.[23] This predicate respects constraints.

See section 5.6 for defining clauses whose head is unified using *single sided unification*.

term_subsumer(*+Special1, +Special2, -General*)

General is the most specific term that is a generalisation of *Special1* and *Special2*. The implementation can handle cyclic terms.

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.[24] This predicate can handle cyclic terms. Attributed variables are handled as normal variables. Associated hooks are *not* executed.

?=(*@Term1, @Term2*)

Succeeds if the syntactic equality of *Term1* and *Term2* can be decided safely, i.e. if the result of `Term1 == Term2` will not change due to further instantiation of either term. It behaves as if defined by `?=(X,Y) :- \+ unifiable(X,Y,[_|_]).`

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 location before execution. It is faster to define a sub-predicate (i.e., `one_character_atoms/1` in the example below) and make a call to this simple predicate. See also the Prolog flag `compile_meta_arguments`.

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

fail *[ISO]*

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

[22] In many systems variant is implemented using two calls to `subsumes_term/2`.

[23] This predicate is often named `subsumes_chk/2` in older Prolog dialects. The current name was established in the ISO WG17 meeting in Edinburgh (2010). The `chk` postfix was considered to refer to determinism as in e.g., `memberchk/2`.

[24] 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.

false *[ISO]*

Same as fail, but the name has a more declarative connotation.

true *[ISO]*

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

repeat *[ISO]*

Always succeed, provide an infinite number of choice points.

! *[ISO]*

Cut. Discard all choice points created since entering the predicate in which the cut appears. In other words, *commit* to the clause in which the cut appears *and* discard choice points that have been created by goals to the left of the cut in the current clause. Meta calling is opaque to the cut. This implies that cuts that appear in a term that is subject to meta-calling (`call/1`) only affect choice points created by the meta-called term. The following control structures are transparent to the cut: `;/2`, `->/2` and `*->/2`. Cuts appearing in the *condition* part of `->/2` and `*->/2` are opaque to the cut. The table below explains the scope of the cut with examples. *Prunes* here means "prunes X choice point created by X".

`t0 :- (a, !, b).`	% prunes a/0 and t0/0
`t1 :- (a, !, fail ; b).`	% prunes a/0 and t1/0
`t2 :- (a -> b, ! ; c).`	% prunes b/0 and t2/0
`t3 :- (a, !, b -> c ; d).`	% prunes a/0
`t4 :- call((a, !, fail ; b)).`	% prunes a/0
`t5 :- \+(a, !, fail).`	% prunes a/0

:Goal1 `,` *:Goal2* *[ISO]*

Conjunction (*and*). True if both *Goal1* and *Goal2* are true.

:Goal1 `;` *:Goal2* *[ISO]*

Disjunction (*or*). True if either *Goal1* or *Goal2* succeeds. Note that the semantics change if *Goal1* contains `->/2` or `*->/2`. `;/2` is transparent to cuts. See `!/0` for details. For example:

```
?- (between(1,2,X) ; X = a).
X = 1 ;
X = 2 ;
X = a.
```

It is strongly adviced to *always* use parenthesis around disjunctions. Conjunctions inside a disjunction should not use parenthesis. Traditionally the `;` is placed at the start of the line rather than at the end because a `;` at the end of a line is easily overlooked. Below is an example of the preferred style used in SWI-Prolog.[25]

```
p :-
    a,
    (   b,
```

[25] Some users prefer a newline after the `;`.

```
        c
    ;   d
    ).
```

Although `;/2` is a *control structure* that is normally handled by the compiler, SWI-Prolog implements `;/2` as a true predicate to support `call/2` and friends as well as to allow for querying predicate properties, for example to support code analysis.

:Goal1 **|** *:Goal2*
: Equivalent to `;/2`. Retained for compatibility only. New code should use `;/2`.

:Condition **->** *:Action* *[ISO]*
: 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. Disregarding the interaction with `!/0`, the combination `;/2` and `->/2` acts as if defined as:

```
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.

Please note that (if->then;else) is read as ((if->then);else) and that the *combined* semantics of this syntactic construct as defined above is *different* from the simple nesting of the two individual constructs, i.e., the semantics of `->/2` *changes* when embedded in `;/2`. See also `once/1`.

As with `;/2`, this construct is always nested in parenthesis. Here is an example of the preferred layout for SWI-Prolog.

```
p :-
    a,
    (   b,
        c
    ->  d,
        e
    ;   f
    ->  g
    ;   h
    ).
```

: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 (`call`(*Condition*), *Action*).[26] If

[26]Note that the *Condition* is wrapped in `call/1`, limiting the scope of the cut (`!/0`

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 `call`(*Condition*) and *Action*, otherwise execute *Else*. The construct is known under the name `if/3` in some other Prolog implementations.

The construct *A* `*->` *B*, i.e., without an *Else* branch, the semantics is the same as (`call`(*A*), *B*).

This construct is rarely used. An example use case is the implementation of OPTIONAL in SPARQL. The optional construct should preserve all solutions if the argument succeeds as least once but still succeed otherwise. This is implemented as below.

```
optional(Goal) :-
    (   Goal
    *-> true
    ;   true
    ).
```

Now calling e.g., `optional(member(X, [a,b]))` has the solutions $X = a$ and $X = b$, while `optional(member(X,[]))` succeeds without binding X.

\+ *:Goal* *[ISO]*

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

Many Prolog implementations (including SWI-Prolog) provide `not/1`. The `not/1` alternative is deprecated due to its strong link to logical negation.

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***)** *[ISO]*

Call *Goal*. This predicate is normally used for goals that are not known at compile time. For example, the Prolog toplevel essentially performs `read(Goal), call(Goal).` Also a *meta* predicates such as `ignore/1` are defined using call:

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

Note that a plain variable as a body term acts as `call/1` and the above is equivalent to the code below. SWI-Prolog produces the same code for these two progams and `listing/1` prints the program above.

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

Note that `call/1` restricts the scope of the cut (`!/0`). A cut inside *Goal* only affects choice points created by *Goal*.

call(*:Goal, +ExtraArg1, ...*) *[ISO]*

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

The call/[2..] construct is handled by the compiler. The predicates call/[2-8] are defined as real (meta-)predicates and are available to inspection through `current_predicate/1`, `predicate_property/2`, etc.[27] Higher arities are handled by the compiler and runtime system, but the predicates are not accessible for inspection.[28]

apply(*:Goal, +List*) *[deprecated]*

Append the members of *List* to the arguments of *Goal* and call the resulting term. For example: `apply(plus(1), [2, X])` calls `plus(1, 2, X)`. New code should use call/[2..] if the length of *List* is fixed.

not(*:Goal*) *[deprecated]*

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

once(*:Goal*) *[ISO]*

Make a possibly *nondet* goal *semidet*, i.e., succeed at most once. Defined as:

```
once(Goal) :-
    call(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 below are identical. Be careful about the interaction with `;/2`. The `apply_macros` library defines an inline expansion of `once/1`, mapping it to `(Goal->true;fail)`. Using the full if-then-else constructs prevents its semantics from being changed when embedded in a `;/2` disjunction.

```
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(_).
```

[27] Arities 2..8 are demanded by ISO/IEC 13211-1:1995/Cor.2:2012.

[28] Future versions of the reflective predicate may fake the presence of `call/9..`. Full logical behaviour, generating all these pseudo predicates, is probably undesirable and will become impossible if *max_arity* is removed.

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 inline 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*. See also `call_with_inference_limit/3`. It was implemented after discussion with Steve Moyle `smoyle@ermine.ox.ac.uk`.

call_with_inference_limit(*:Goal, +Limit, -Result*)
Equivalent to `call`(*Goal*), but limits the number of inferences *for each solution of Goal*.[29]. Execution may terminate as follows:

- If *Goal* does *not* terminate before the inference limit is exceeded, *Goal* is aborted by injecting the exception `inference_limit_exceeded` into its execution. After termination of *Goal*, *Result* is unified with the atom `inference_limit_exceeded`. *Otherwise*,
- If *Goal* fails, `call_with_inference_limit/3` fails.
- If *Goal* succeeds *without a choice point*, *Result* is unified with `!`.
- If *Goal* succeeds *with a choice point*, *Result* is unified with `true`.
- If *Goal* throws an exception, `call_with_inference_limit/3` re-throws the exception.

An inference is defined as a call or redo on a predicate. Please note that some primitive built-in predicates are compiled to virtual machine instructions for which inferences are not counted. The execution of predicates defined in other languages (e.g., C, C++) count as a single inference. This includes potentially expensive built-in predicates such as `sort/2`.

Calls to this predicate may be nested. An inner call that sets the limit below the current is honoured. An inner call that would terminate after the current limit does not change the effective limit. See also `call_with_depth_limit/3` and `call_with_time_limit/2`.

[29]This predicate was realised after discussion with Ulrich Neumerkel and Markus Triska.

setup_call_cleanup(*:Setup, :Goal, :Cleanup*)

Calls `(once(Setup), Goal)`. If *Setup* succeeds, *Cleanup* will be called exactly once after *Goal* is finished: either on failure, deterministic success, commit, or an exception. The execution of *Setup* is protected from asynchronous interrupts like `call_with_time_limit/2` (package clib) or `thread_signal/2`. In most uses, *Setup* will perform temporary side-effects required by *Goal* that are finally undone by *Cleanup*.

Success or failure of *Cleanup* is ignored, and choice points it created are destroyed (as `once/1`). If *Cleanup* throws an exception, this is executed as normal while it was not triggered as the result of an exception the exception is propagated as normal. If *Cleanup* was triggered by an exception the rules are described in section 4.10.1

Typically, this predicate is used to cleanup 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) :-
        setup_call_cleanup(open(File, read, In),
                           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 it is impossible to implement this predicate in Prolog. The closest approximation would be to read all terms into a list, close the file and call `member/2`. Without `setup_call_cleanup/3` there is no way to gain control if the choice point left by `repeat/0` is removed by a cut or an exception.

`setup_call_cleanup/3` can also be used to test determinism of a goal, providing a portable alternative to `deterministic/1`:

```
?- setup_call_cleanup(true,(X=1;X=2), Det=yes).

X = 1 ;

X = 2,
Det = yes ;
```

This predicate is under consideration for inclusion into the ISO standard. For compatibility with other Prolog implementations see `call_cleanup/2`.

setup_call_catcher_cleanup(*:Setup, :Goal, +Catcher, :Cleanup*)

Similar to `setup_call_cleanup`(*Setup, Goal, Cleanup*) with additional information on the reason for calling *Cleanup*. Prior to calling *Cleanup*, *Catcher* unifies with the termination code (see below). 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*.

external_exception(*Exception*)
: *Goal* succeeded with choice points and these are now discarded due to an exception. For example:

```
?- setup_call_catcher_cleanup(true, (X=1;X=2),
                              Catcher, writeln(Catcher)),
   throw(ball).
external_exception(ball)
ERROR: Unhandled exception: Unknown message: ball
```

call_cleanup(*:Goal, :Cleanup*)
: Same as `setup_call_cleanup`(*true, Goal, Cleanup*). This is provided for compatibility with a number of other Prolog implementations only. Do not use `call_cleanup/2` if you perform side-effects prior to calling that will be undone by *Cleanup*. Instead, use `setup_call_cleanup/3` with an appropriate first argument to perform those side-effects.

call_cleanup(*:Goal, +Catcher, :Cleanup*) *[deprecated]*
: Same as `setup_call_catcher_cleanup`(*true, Goal, Catcher, Cleanup*). The same warning as for `call_cleanup/2` applies.

undo(*:Goal*)
: Add *Goal* to the *trail*. *Goal* is executed as `ignore/1` on the first opportunity after backtracking to a point before the call to *Goal*. This predicate is intended to make otherwise persistent changes to the database or created by foreign procedures backtrackable if it is possible to define a goal that reverts the effect of the initial call. A typical use case is to define a *backtrackable assert*.

```
b_assertz(Term) :-
    assertz(Term, Ref),
    undo(erase(Ref)).
```

Without `undo/1` we can achieve something similar by leaving a choicepoint using the almost portable[30] alternative below.

[30] `assertz/2` is not part of the ISO standard but supported by multiple systems.

```
b_assertz(Term) :-
    assertz(Term, Ref),
    (   true
    ;   erase(Ref),
        fail
    ).
```

The `undo/1` based solution avoids leaving a choice point open and, more importantly, keeps undoing the assert also if the choice point from the second alternative is pruned.

Currently the following remarks apply

- *Goal* is *copied* when it is registered.
- “First opportunity” means after backtracking or at the first call port reached.
- Multiple undo goals may be scheduled that are executed as a batch. If multiple goals raise an exception, the most urgent is preserved after all goals have been executed.
- It is not allowed for *Goal* to call `undo/1`. An attempt to do so results in a `permission_error` exception.
- Note that an exception that is caught higher in the call stack backtracks and therefore ensures *Goal* is called.

See also `snapshot/1` and `transaction/1`.

4.9 Delimited continuations

The predicates `reset/3` and `shift/1` implement *delimited continuations* for Prolog. Delimited continuations for Prolog are described in [Schrijvers *et al.*, 2013] (preprint PDF). The mechanism allows for proper *coroutines*, two or more routines whose execution is interleaved, while they exchange data. Note that coroutines in this sense differ from coroutines realised using attributed variables as described in chapter 8.

Note that `shift/1` captures the *forward continuation*. It notably does not capture choicepoints. Choicepoints created before the continuation is captured remain open, while choicepoints created when the continuation is executed live their normal life. Unfortunately the consequences for *committing* a choicepoint is complicated. In general a cut (`!/0`) in the continuation does not have the expected result. Negation (`\+/1`) and if-then(-else) (`->/2`) behave as expected, *provided the continuation is called immediately*. This works because for `\+/1` and `->/2` the continuation contains a reference to the choicepoint that must be cancelled and this reference is restored when possible. If, as with tabling, the continuation is saved and called later, the commit has no effect. We illustrate the three scenarios using with the programs below.

```
t1 :-
    reset(gbad, ball, Cont),
    (   Cont == 0
    ->  true
    ;   writeln(resuming),
```

```
        call(Cont)
    ).

gbad :-
    n, !, fail.
gbad.

n :-
    shift(ball),
    writeln(n).
```

Here, the !/0 has **no effect**:

```
?- t1.
resuming
n
true.
```

The second example uses \+/1, which is essentially (G->fail;true).

```
t2 :-
    reset(gok, ball, Cont),
    (   Cont == 0
    ->  true
    ;   writeln(resuming),
        call(Cont)
    ).

gok :-
    \+ n.
```

In this scenario the normal semantics of \+/1 is preserved:

```
?- t1.
resuming
n
false.
```

In the last example we illustrate what happens if we assert the continuation to be executed later. We write the negation using if-then-else to make it easier to explain the behaviour.

```
:- dynamic cont/1.

t3 :-
    retractall(cont(_)),
    reset(gassert, ball, Cont),
```

```
    (   Cont == 0
    ->  true
    ;   asserta(cont(Cont))
    ).

c3 :-
    cont(Cont),
    writeln(resuming),
    call(Cont).

gassert :-
    (   n
    ->  fail
    ;   true
    ).
```

Now, `t3/0` succeeds *twice*. This is because `n/0` shifts, so the commit to the `fail/0` branch is not executed and the `true/0` branch is evaluated normally. Calling the continuation later using `c3/0` fails because the choicepoint that realised the if-then-else does not exist in the continuation and thus the effective continuation is the remainder of `n/0` and `fail/0` in `gassert/0`.

```
?- t3.
true ;
true.

?- c3.
resuming
n
false.
```

The suspension mechanism provided by delimited continuations is used to implement *tabling* [Desouter *et al.*, 2015], (available here). See section 7.

reset(*:Goal, ?Ball, -Continuation*)
: Call *Goal*. If *Goal* calls `shift/1` and the argument of `shift/1` can be unified with *Ball*,[31] `shift/1` causes `reset/3` to return, unifying *Continuation* with a goal that represents the *continuation* after `shift/1`. In other words, meta-calling *Continuation* completes the execution where shift left it. If *Goal* does not call `shift/1`, *Continuation* are unified with the integer `0` (zero).[32]

shift(*+Ball*)
: Abandon the execution of the current goal, returning control to just *after* the matching `reset/3` call. This is similar to `throw/1` except that (1) nothing is 'undone' and (2) the

[31] The argument order described in [Schrijvers *et al.*, 2013] is `reset`(*Goal,Continuation,Ball*). We swapped the argument order for compatibility with `catch/3`

[32] Note that older versions also unify *Ball* with `0`. Testing whether or not shift happened on *Ball* however is *always* ambiguous.

3th argument of `reset/3` is unified with the *continuation*, which allows the code calling `reset/3` to *resume* the current goal.

shift_for_copy(+*Ball*) *[experimental]*
Similar to `shift/1`. This version is intended for situations where it is assumed the continuation is copied and saved to be executed one or multiple times in a different context. This notably prevents restoring choice points saved for `\+/1`, *If*`->`*Then*`;`*Else*, etc.

4.10 Exception handling

The predicates `catch/3` and `throw/1` provide ISO compliant raising and catching of exceptions.

catch(*:Goal, +Catcher, :Recover*) *[ISO]*
Behaves as `call/1` if no exception is raised when executing *Goal*. If an 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 comparable to `call/1`. Recovery from an exception is much slower, especially if the exception term is large due to the copying thereof or is decorated with a stack trace using, e.g., the library `prolog_stack` based on the `prolog_exception_hook/4` hook predicate to rewrite exceptions.

throw(+*Exception*) *[ISO]*
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 that `throw/1` 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 backtracking until it actually finds 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 shared variables. We assume this to be highly unlikely and could not think of a scenario where this is useful.[33]

In addition to explicit calls to `throw/1`, many built-in predicates throw exceptions directly from C. If the *Exception* term cannot be copied due to lack of stack space, the following actions are tried in order:

1. If the exception is of the form `error`(*Formal, ImplementationDefined*), try to raise the exception without the *ImplementationDefined* part.
2. Try to raise `error(resource_error`*(stack), global*).
3. Abort (see `abort/0`).

If an exception is raised in a call-back from C (see chapter 12) and not caught in the same call-back, `PL_next_solution()` fails and the exception context can be retrieved using `PL_exception()`.

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

catch_with_backtrace(*:Goal, +Catcher, :Recover*)

As `catch/3`, but if library `prolog_stack` is loaded and an exception of the shape `error`(*Format, Context*) is raised *Context* is extended with a backtrace. To catch an error and print its message including a backtrace, use the following template:

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

    ...,
    catch_with_backtrace(Goal, Error,
                         print_message(error, Error)),
    ...,
```

This is good practice for a *catch-all* wrapper around an application. See also `main/0` from library `main`.

4.10.1 Urgency of exceptions

Under some conditions an exception may be raised as a result of handling another exception. Below are some of the scenarios:

- The predicate `setup_call_cleanup/3` calls the cleanup handler as a result of an exception and the cleanup handler raises an exception itself. In this case the most *urgent* exception is propagated into the environment.
- Raising an exception fails due to lack of resources, e.g., lack of stack space to store the exception. In this case a resource exception is raised. If that too fails the system tries to raise a resource exception without (stack) context. If that fails it will raise the exception `'$aborted'`, also raised by `abort/0`. As no stack space is required for processing this atomic exception, this should always succeed.
- Certain *callback* operations raise an exception while processing another exception or a previous callback already raised an exception before there was an opportunity to process the exception. The most notable *callback* subject to this issue are `prolog_event_hook/1` (supporting e.g., the graphical debugger), `prolog_exception_hook/4` (rewriting exceptions, e.g., by adding context) and `print_message/2` when called from the core facilities such as the internal debugger. As with `setup_call_cleanup/3`, the most *urgent* exception is preserved.

If the most urgent exceptions needs to be preserved, the following exception ordering is respected, preserving the topmost matching error.

1. `'$aborted'` (`abort/0`)
2. `time_limit_exceeded` (`call_with_time_limit/2`)
3. `error`(`resource_error`*(Resource), Context*)
4. `error`(*Formal, Context*)
5. All other exceptions

Note The above resolution is not described in the ISO standard. This is not needed either because ISO does not specify `setup_call_cleanup/3` and does not deal with environment management issues such as (debugger) callbacks. Neither does it define `abort/0` or timeout handling. Notably `abort/0` and timeout are non-logical control structures. They are implemented on top of exceptions as they need to unwind the stack, destroy choice points and call cleanup handlers in the same way. However, the pending exception should not be replaced by another one before the intended handler is reached. The abort exception cannot be caught, something which is achieved by wrapping the *cleanup handler* of `catch/3` into `call_cleanup`(*Handler, abort*).

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

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

- *current_prolog_flag(debug_on_error, true)*
  If the exception is not caught by Prolog in the current context, it will trap the tracer to help analyse 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`. The hook `prolog_exception_hook/4` can be used to add more debugging facilities to exceptions. An example is the library `http/http_error`, generating a full stack trace on errors in the HTTP server library.

4.10.3 The exception term

General form of the ISO standard exception term

The predicate `throw/1` takes a single argument, the *exception term*, and the ISO standard stipulates that the exception term be of the form `error`(*Formal, Context*) with:

- *Formal*
  the 'formal' description of the error, as listed in chapter 7.12.2 pp. 62-63 ("Error classification") of the ISO standard. It indicates the *error class* and possibly relevant *error context* information.

It may be a compound term of arity 1,2 or 3 - or simply an atom if there is no relevant error context information.

- *Context*
  additional context information beyond the one in *Formal*. If may be unset, i.e. a fresh variable, or set to something that hopefully will help the programmer in debugging. The structure of *Context* is left unspecified by the ISO Standard, so SWI-Prolog creates it own convention (see below).

Thus, constructing an error term and throwing it might take this form (although you would not use the illustrative explicit naming given here; instead composing the exception term directly in a one-liner):

```
Exception = error(Formal, Context),
Context   = ... some local convention ...,
Formal    = type_error(ValidType, Culprit), % for "type error" for example
ValidType = integer,                        % valid atoms are listed in the ISO s
Culprit   = ... some value ...,
throw(Exception)
```

Note that the ISO standard formal term expresses *what should be the case* or *what is the expected correct state*, and not *what is the problem*. For example:

- •
  If a variable is found to be uninstantiated but should be instantiated, the error term is `instantiation_error`: The problem is not that there is an unwanted instantiation, but that the correct state is the one with an instantiated variable.

- •
  In case a variable is found to be instantiated but should be uninstantiated (because it will be used for output), the error term is `uninstantiation_error`(*Culprit*): The problem is not that there is lack of instantiation, but that the correct state is the one which *Culprit* (or one of its subterms) is more uninstantiated than is the case.

- •
  If you try to disassemble an empty list with `compound_name_arguments/3`, the error term is `type_error`(*compound,[]*). The problem is not that `[]` is (erroneously) a compound term, but that a compound term is expected and `[]` does not belong to that class.

Throwing exceptions from applications and libraries

User predicates are free to choose the structure of their *exception terms* (i.e., they can define their own conventions) but *should* adhere to the ISO standard if possible, in particular for libraries.

Notably, exceptions of the shape `error`(*Formal,Context*) are recognised by the development tools and therefore expressing unexpected situations using these exceptions improves the debugging experience.

In SWI-Prolog, the second argument of the exception term, i.e., the *Context* argument, is generally of the form `context`(*Location, Message*), where:

- *Location*
  describes the execution context in which the exception occurred. While the *Location* argument may be specified as a predicate indicator (*Name/Arity*), it is typically filled by the `prolog_stack` library. This library recognises uncaught errors or errors caught by `catch_with_backtrace/3` and fills the *Location* argument with a *backtrace*.

- *Message*
  provides an additional description of the error or can be left as a fresh variable if there is nothing appropriate to fill in.

ISO standard exceptions can be thrown via the predicates exported from `error`. Termwise, these predicates look exactly like the *Formal* of the ISO standard error term they throw:

- •
  `instantiation_error/1` (the argument is not used: ISO specifies no argument)
- •
  `uninstantiation_error/1`
- •
  `type_error/2`
- •
  `domain_error/2`
- •
  `existence_error/2`
- •
  `existence_error/3` (a SWI-Prolog extension that is not ISO)
- •
  `permission_error/3`
- •
  `representation_error/1`
- •
  `resource_error/1`
- •
  `syntax_error/1`

4.11 Printing messages

The predicate `print_message/2` is 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. A 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 define `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 by the system and libraries to print messages. *Kind* describes the nature of the message, while *Term* is a Prolog term that describes the content. Printing messages through this indirection instead of using `format/3` to the stream `user_error` allows displaying the message appropriate to the application (terminal, logfile, graphics), acting on messages based on their content instead of a string (see `message_hook/3`) and creating language specific versions of the messages. See also section 4.11.1. The following message kinds are known:

banner
The system banner message. Banner messages can be suppressed by setting the Prolog flag `verbose` to `silent`.

debug(*Topic*)
Message from library(debug). See `debug/3`.

error
The message indicates an erroneous situation. This kind is used to print uncaught exceptions of type `error`(*Formal, Context*). See section introduction (section 4.11). An error message causes the process to halt with status 1 if the Prolog flag `on_error` is set to `halt` and the message is not intercepted by `message_hook/3`. Not intercepted error messages increment the `errors` key for `statistics/2`.

help
User requested help message, for example after entering 'h' or '?' to a prompt.

information
Information that is requested by the user. An example is `statistics/0`.

informational
Typically messages of events and progress that are considered useful to a developer. Such messages can be suppressed by setting the Prolog flag `verbose` to `silent`.

silent
Message that is normally not printed. Applications may define `message_hook/3` to act upon such messages.

trace
Messages from the (command line) tracer.

warning
The message indicates something dubious that is not considered fatal. For example, discontiguous predicates (see `discontiguous/1`). A warning message causes the process to halt with status 1 if the Prolog flag `on_warning` is set to `halt` and the message is not intercepted by `message_hook/3`. Not intercepted warning messages increment the `warnings` key for `statistics/2`.

The predicate `print_message/2` first translates the *Term* into a list of 'message lines' (see `print_message_lines/3` for details). Next, it calls the hook `message_hook/3` to allow the user to intercept the message. If `message_hook/3` fails it prints 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 write messages from your own predicates, it is recommended to reuse the existing message terms if applicable. If no existing message term is applicable, invent a fairly unique term that represents the event and define a rule for the multifile predicate prolog:message//1. See section 4.11.1 for a deeper discussion and examples.

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 arguments. 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. This is combined with a subsequent message that starts with `at_same_line` to complete the line.

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`).

ansi(*+Attributes, +Format, +Args*)
This message may be intercepted by means of the hook `prolog:message_line_element/2`. The library `ansi_term` implements this hook to achieve coloured output. If it is not intercepted it invokes `format`(*Stream, Format, Args*).

nl
A new line is started. If the message is not complete, *Prefix* is printed before the remainder of the message.

begin(*Kind, Var*)
end(*Var*)
The entire message is headed by `begin`(*Kind, Var*) and ended by `end`(*Var*). This feature is used by, e.g., library `ansi_term` to colour entire messages.

⟨*Format*⟩
Handed to `format/3` as `format(`*Stream, Format, []*`)`. Deprecated because it is ambiguous if *Format* collides with one of the atomic commands.

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

message_hook(+*Term*, +*Kind*, +*Lines***)**
Hook predicate that may be defined 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 must be defined dynamic and multifile to allow other modules defining clauses for it too.

thread_message_hook(+*Term*, +*Kind*, +*Lines***)**
As `message_hook/3`, but this predicate is local to the calling thread (see `thread_local/1`). This hook is called *before* `message_hook/3`. The 'pre-hook' is indented to catch messages they may be produced by calling some goal without affecting other threads.

message_property(+*Kind*, ?*Property***)**
This hook can be used to define additional message kinds and the way they are displayed. The following properties are defined:

color(-*Attributes***)**
Print message using ANSI terminal attributes. See `ansi_format/3` for details. Here is an example, printing help messages in blue:

```
:- multifile user:message_property/2.

user:message_property(help, color([fg(blue)])).
```

prefix(-*Prefix***)**
Prefix printed before each line. This argument is handed to `format/3`. The default is `'~N'`. For example, messages of kind `warning` use `'~NWarning: '`.

location_prefix(+*Location*, -*FirstPrefix*, -*ContinuePrefix***)**
Used for printing messages that are related to a source location. Currently, *Location* is a term *File*:*Line*. *FirstPrefix* is the prefix for the first line and -*ContinuePrefix* is the prefix for continuation lines. For example, the default for errors is

```
location_prefix(File:Line,
                '~NERROR: ~w:~d:'-[File,Line], '~N\t')).
```

stream(-*Stream***)**
Stream to which to print the message. Default is `user_error`.

wait(-*Seconds***)**
Amount of time to wait after printing the message. Default is not to wait.

prolog:message_line_element(*+Stream, +Term*)
This hook is called to print the individual elements of a message from `print_message_lines/3`. This hook is used by e.g., library `ansi_term` to colour messages on ANSI-capable terminals.

prolog:message_prefix_hook(*+ContextTerm, -Prefix*)
This hook is called to add context to the message prefix. *ContextTerm* is a member of the list provided by the `message_context`. *Prefix* must be unified with an atomic value that is added to the message prefix.

message_to_string(*+Term, -String*)
Translates a message term into a string object (see section 5.2).

version
Write the SWI-Prolog banner message as well as additional messages registered using `version/1`. This is the default *initialization goal* which can be modified using `-g`.

version(*+Message*)
Register additional messages to be printed by `version/0`. Each registered message is handed to the message translation DCG and can thus be defined using the hook prolog:message//1. If not defined, it is simply printed.

4.11.1 Printing from libraries

Libraries should *not* use `format/3` or other output predicates directly. Libraries that print informational output directly to the console are hard to use from code that depend on your textual output, such as a CGI script. The predicates in section 4.11 define the API for dealing with messages. The idea behind this is that a library that wants to provide information about its status, progress, events or problems calls `print_message/2`. The first argument is the *level*. The supported levels are described with `print_message/2`. Libraries typically use `informational` and `warning`, while libraries should use exceptions for errors (see `throw/1`, `type_error/2`, etc.).

The second argument is an arbitrary Prolog term that carries the information of the message, but *not* the precise text. The text is defined by the grammar rule prolog:message//1. This distinction is made to allow for translations and to allow hooks processing the information in a different way (e.g., to translate progress messages into a progress bar).

For example, suppose we have a library that must download data from the Internet (e.g., based on `http_open/3`). The library wants to print the progress after each downloaded file. The code below is a good skeleton:

```
download_urls(List) :-
        length(List, Total),
        forall(nth1(I, List, URL),
               (   download_url(URL),
                   print_message(informational,
                                 download_url(URL, I, Total)))).
```

The programmer can now specify the default textual output using the rule below. Note that this rule may be in the same file or anywhere else. Notably, the application may come with several rule sets for different languages. This, and the user-hook example below are the reason to represent the message as a compound term rather than a string. This is similar to using message numbers in non-symbolic languages. The documentation of `print_message_lines/3` describes the elements that may appear in the output list.

```
:- multifile
        prolog:message//1.

prolog:message(download_url(URL, I, Total)) -->
        { Perc is round(I*100/Total) },
        [ 'Downloaded ~w; ~D from ~D (~d%)'-[URL, I, Total, Perc] ].
```

A *user* of the library may define rules for `message_hook/3`. The rule below acts on the message content. Other applications can act on the message level and, for example, popup a message box for warnings and errors.

```
:- multifile user:message_hook/3.

message_hook(download_url(URL, I, Total), _Kind, _Lines) :-
        <send this information to a GUI component>
```

In addition, using the command line option `-q`, the user can disable all *informational* messages.

4.12 Handling signals

As of version 3.1.0, SWI-Prolog is able to handle software interrupts (signals) in Prolog as well as in foreign (C) code (see section 12.4.15).

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 with asynchronous interprocess 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 interprocess (or interthread) communication (Unix kill() function).

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

Determines how *Signal* is processed. *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 to ⟨*Name*⟩/1 in module ⟨*Module*⟩. The handler is called with a single argument: the name of the signal as an atom. The Prolog names for signals are explained below.

Three names have special meaning. `throw` implies Prolog will map the signal onto a Prolog exception as described in section 4.10, `debug` specifies the debug interrupt prompt that is initially bound to `SIGINT` (Control-C) and `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 following structure:

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

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, the following signals are handled unless the command line option `--no-signals` is specified:

int
Prompts the user, allowing to inspect the current state of the process and start the tracer.

usr2
Bound to an empty signal handler used to make blocking system calls return. This allows `thread_signal/2` to interrupt threads blocked in a system call. See also `prolog_alert_signal/2`.

hup, term, abrt, quit
Causes normal Prolog cleanup (e.g., `at_halt/1`) before terminating the process with the same signal.

segv, ill, bus, sys
Dumps the C and Prolog stacks and runs cleanup before terminating the process with the same signal.

fpe, alrm, xcpu, xfsz, vtalrm
Throw a Prolog exception (see above).

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`).

prolog_alert_signal(*?Old, +New***)**
Query or set the signal used to unblock blocking system calls on Unix systems and process pending Prolog signals. The default is `SIGUSR2`. See also `--sigalert`.

4.12.1 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. Currently, the system only supports signals numbered 1 to 32[34]. Installing a signal outside the limited set of supported signals in MS-Windows crashes the application.

[34] TBD: the system should support the Unix realtime signals

- *Safety*
  Immediately delivered signals (see below) are unsafe. This implies that foreign functions called from a handler cannot safely use the SWI-Prolog API and cannot use C longjmp(). Handlers defined as `throw` are unsafe. Handlers defined to call a predicate are safe. Note that the predicate can call `throw/1`, but the delivery is delayed until Prolog is in a safe state.

  The C-interface described in section 12.4.15 provides the option `PL_SIGSYNC` to select either safe synchronous or unsafe asynchronous delivery.

- *Time of delivery*
  Using `throw` or a foreign handler, signals are delivered immediately (as defined by the OS). When using a Prolog predicate, delivery is delayed to a safe moment. Blocking system calls or foreign loops may cause long delays. Foreign code can improve on that by calling `PL_handle_signals()`.

  Signals are blocked when the garbage collector is active.

4.13 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 of characters or tokens and for generating a flat list from a term.

Grammar rules look like ordinary clauses using `-->/2` for separating the head and body rather than `:-/2`. Expanding grammar rules is done by `expand_term/2`, which adds two additional arguments to each term for representing the difference list.

The body of a grammar rule can contain three types of terms. A callable term is interpreted as a reference to a grammar rule. Code between `{...}` is interpreted as plain 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.

We illustrate the behaviour by defining a rule set for parsing an integer.

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

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

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

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

phrase(*:DCGBody, ?List*)
Equivalent to `phrase(`*DCGBody*`,` *InputList*`, [])`.

phrase(*:DCGBody, ?List, ?Rest*)
True when *DCGBody* applies to the difference *List*/*Rest*. Although *DCGBody* is typically a *callable* term that denotes a grammar rule, it can be any term that is valid as the body of a DCG rule.

The example below calls the rule set integer//1 defined in section 4.13 and available from `library(dcg/basics)`, binding *Rest* to the remainder of the input after matching the integer.

```
?- [library(dcg/basics)].
?- atom_codes('42 times', Codes),
   phrase(integer(X), Codes, Rest).
X = 42
Rest = [32, 116, 105, 109, 101, 115]
```

The next example exploits a complete body. Given the following definition of `digit_weight//1`, we can pose the query below.

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

```
?- atom_codes('Version 3.4', Codes),
   phrase(("Version ",
           digit_weight(Major),".",digit_weight(Minor)),
          Codes).
Major = 3,
Minor = 4.
```

The SWI-Prolog implementation of `phrase/3` verifies that the *List* and *Rest* arguments are unbound, bound to the empty list or a list *cons cell*. Other values raise a type error.[35] The predicate `call_dcg/3` is provided to use grammar rules with terms that are not lists.

Note that the syntax for lists of codes changed in SWI-Prolog version 7 (see section 5.2). If a DCG body is translated, both `"text"` and `` `text` `` is a valid code-list literal in version 7. A version 7 string (`"text"`) is **not** acceptable for the second and third arguments of `phrase/3`. This is typically not a problem for applications as the input of a DCG rarely appears in the source code. For testing in the toplevel, one must use double quoted text in versions prior to 7 and back quoted text in version 7 or later.

[35] The ISO standard allows for both raising a type error and accepting any term as input and output. Note the tail of the list is not checked for performance reasons.

See also `portray_text/1`, which can be used to print lists of character codes as a string to the top level and debugger to facilitate debugging DCGs that process character codes. The library `apply_macros` compiles `phrase/3` if the argument is sufficiently instantiated, eliminating the runtime overhead of translating *DCGBody* and meta-calling.

call_dcg(*:DCGBody, ?State0, ?State***)**
As `phrase/3`, but without type checking *State0* and *State*. This allows for using DCG rules for threading an arbitrary state variable. This predicate was introduced after type checking was added to `phrase/3`.[36]

A portable solution for threading state through a DCG can be implemented by wrapping the state in a list and use the DCG semicontext facility. Subsequently, the following predicates may be used to access and modify the state:[37]

```
state(S), [S] --> [S].
state(S0, S), [S] --> [S0].
```

As stated above, grammar rules are a general interface to difference lists. To illustrate, we show a DCG-based implementation of `reverse/2`:

```
reverse(List, Reversed) :-
        phrase(reverse(List), Reversed).

reverse([])    --> [].
reverse([H|T]) --> reverse(T), [H].
```

4.14 Database

SWI-Prolog offers several ways to store data in globally accessible memory, i.e., outside the Prolog *stacks*. Data stored this way notably does not change on *backtracking*. Typically it is a bad idea to use any of the predicates in this section for realising global variables that can be assigned to. Typically, first consider representing data processed by your program as terms passed around as predicate arguments. If you need to reason over multiple solutions to a goal, consider `findall/3`, `aggregate/3` and related predicates.

Nevertheless, there are scenarios where storing data outside the Prolog stacks is a good option. Below are the main options for storing data:

Using dynamic predicates Dynamic predicates are predicates for which the list of clauses is modified at runtime using `asserta/1`, `assertz/1`, `retract/1` or `retractall/1`. Following the ISO standard, predicates that are modified this way need to be declared using the `dynamic/1` *directive*. These facilities are defined by the ISO standard and widely supported. The mechanism is often considered slow in the literature. Performance depends on the Prolog implementation. In SWI-Prolog, querying dynamic predicates has the same

[36]After discussion with Samer Abdallah.
[37]This solution was proposed by Markus Triska.

performance as static ones. The manipulation predicates are fast. Using `retract/1` or `retractall/1` on a predicate registers the predicate as 'dirty'. Dirty predicates are cleaned by `garbage_collect_clauses/0`, which is normally automatically invoked. Some workloads may result in significant performance reduction due to skipping retracted clauses and/or clause garbage collection.

Dynamic predicates can be wrapped using library `persistency` to maintain a backup of the data on disk. Dynamic predicates come in two flavours, *shared* between threads and *local* to each thread. The latter version is created using the directive `thread_local/1`.

The recorded database The 'recorded database' registers a list of terms with a *key*, an atom or compound term. The list is managed using `recorda/3`, `recordz/3` and `erase/1`. It is queried using `recorded/3`. The recorded database is not part of the ISO standard but fairly widely supported, notably in implementations building on the 'Edinburgh tradition'. There are few reasons to use this database in SWI-Prolog due to the good performance of dynamic predicates. Advantages are (1) the handle provides a direct reference to a term, (2) cyclic terms can be stored and (3) attributes (section 8.1) are preserved. Disadvantages are (1) the terms in a list associated with a key are not indexed, (2) the poorly specified *immediate update semantics* (see section 4.14.5 applies to the recorded database and (3) reduced portability.

The `flag/3` predicate The predicate `flag/3` associates one simple value (number or atom) with a key (atom, integer or compound). It is an old SWI-Prolog specific predicate that should be considered deprecated, although there is no plan to remove it.

Using global variables The predicates `b_setval/2` and `nb_setval/2` associate a term living on the Prolog stack with a name, either backtrackable or non-backtrackable. Backtrackable and non-backtrackable assignment without using a global name can be realised with `setarg/3` and `nb_setarg/3`. Notably the latter are used to realise aggregation as e.g., `aggregate_all/3` performs.

Tries As of version 7.3.21, SWI-Prolog provides *tries* (prefix trees) to associate a term *variant* with a value. Tries have been introduced to support *tabling* and are described in section 4.14.4.

4.14.1 Managing (dynamic) predicates

abolish(*:PredicateIndicator*) *[ISO]*

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 was 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.

copy_predicate_clauses(*:From, :To*)
Copy all clauses of predicate *From* to *To*. The predicate *To* must be dynamic or undefined. If *To* is undefined, it is created as a dynamic predicate holding a copy of the clauses of *From*. If *To* is a dynamic predicate, the clauses of *From* are added (as in `assertz/1`) to the clauses of *To*. *To* and *From* must have the same arity. Acts as if defined by the program below, but at a much better performance by avoiding decompilation and compilation.

```
copy_predicate_clauses(From, To) :-
        head(From, MF:FromHead),
        head(To, MT:ToHead),
        FromHead =.. [_|Args],
        ToHead =.. [_|Args],
        forall(clause(MF:FromHead, Body),
               assertz(MT:ToHead, Body)).

head(From, M:Head) :-
        strip_module(From, M, Name/Arity),
        functor(Head, Name, Arity).
```

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 contexts is discouraged.

retract(*+Term*) *[ISO,nondet]*
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. The `retract/1` predicate respects the *logical update view*. This implies that `retract/1` succeeds for all clauses that match *Term* when the predicate was *called*. The example below illustrates that the first call to `retract/1` succeeds on `bee` on backtracking despite the fact that `bee` is already retracted.[38]

```
:- dynamic insect/1.
insect(ant).
insect(bee).

?- (   retract(insect(I)),
       writeln(I),
       retract(insect(bee)),
```

[38] Example by Jan Burse

```
        fail
    ;   true
    ).
ant ;
bee.
```

If multiple threads start a retract on the same predicate at the same time their notion of the *entry generation* is adjusted such that they do not retract the same first clause. This implies that, if multiple threads use `once(retract(Term))`, no two threads will retract the same clause. Note that on backtracking over `retract/1`, multiple threads may retract the same clause as both threads respect the logical update view.

retractall(+*Head*) *[ISO,det]*

All facts or clauses in the database for which the *head* unifies with *Head* are removed. If *Head* refers to a predicate that is not defined, it is implicitly created as a dynamic predicate. See also `dynamic/1`.[39]

asserta(+*Term*) *[ISO]*
assertz(+*Term*) *[ISO]*
assert(+*Term*) *[deprecated]*

Assert a clause (fact or rule) into the database. The predicate `asserta/1` asserts the clause as first clause of the predicate while `assertz/1` assert the clause as last clause. The deprecated `assert/1` is equivalent to `assertz/1`. If the program space for the target module is limited (see `set_module/1`), `asserta/1` can raise a `resource_error`(*program_space*) exception. The example below adds two facts and a rule. Note the double parentheses around the rule.

```
?- assertz(parent('Bob', 'Jane')).
?- assertz(female('Jane')).
?- assertz((mother(Child, Mother) :-
                parent(Child, Mother),
                female(Mother))).
```

asserta(+*Term, -Reference*)
assertz(+*Term, -Reference*)
assert(+*Term, -Reference*) *[deprecated]*

Equivalent to `asserta/1`, `assertz/1`, `assert/1`, but in addition unifies *Reference* with a handle to the asserted clauses. The handle can be used to access this clause with `clause/3` and `erase/1`.

Transactions

Traditionally, Prolog database updates add or remove individual clauses. The *Logical Update View* ensures that a goal that is started on a dynamic predicate does not see modifications due to `assert/1` or `retract/1` during its life time. See section 4.14.5. In a multi-threaded context this assumption still holds for individual predicates: concurrent modifications to a dynamic predicate are invisible.

[39]The ISO standard only allows using `dynamic/1` as a *directive*.

Transactions allow running a goal in *isolation*. The goals running inside the transaction 'see' the database as it was when the transaction was started together with database changes done by the transaction goal. Other threads see no changes until the transaction is *committed*. The commit, also if it involved multiple clauses spread over multiple predicates, becomes *atomically* visible to other threads. Transactions have several benefits [Wielemaker, 2013]

- If a database update requires multiple `assert/1` and/or `retract/1` operations, a transaction ensure either all are executed or the database remains unchanged. Notably unexpected exceptions or failures cannot leave the database in an inconsistent state.

- Other threads do not see the intermediate inconsistent states when a database update that consists of multiple assert and/or retract is performed in a transaction. This notably avoids the need to use locks (see `with_mutex/2`) in threads that read the data. A reading thread may still need to use `snapshot/1` if a goal depends on multiple calls to dynamic predicates. Unlike locks, transaction and snapshot based synchronization allows both readers and writers to make progress simultaneously.[40]

  Transactions on their own **do not guarantee consistency**. For example, when running the code below to update the temperature concurrently from multiple threads it is possible for the global state to have multiple `temperature/1` clauses.

  ```
  update_temperature(Temp) :-
      transaction(( retractall(temperature(_)),
                    asserta(temperature(Temp)))).
  ```

  Global *consistency* can be achieved by wrapping the above transaction using `with_mutex/2` or by using `transaction/3` with a *constraint* that demands a single clause for `temperature/1`

- Transactions allow for "what if" reasoning over the dynamic database. This is particularly useful when combined with the deductive database facilities provided by tabling (see section 7).

SWI-Prolog transactions only affect the *dynamic* database. Static predicates are globally visible and shared at all times. In particular, transactions do not affect loading source files and thus, source files loaded inside a transaction (e.g., due to *autoloading*) are immediately globally visible. This may pose problems if loading source files provide clauses for dynamic predicates.

transaction(*:Goal*)
transaction(*:Goal, +Options*)

Run *Goal* as `once/1` in a transaction. This implies that access to dynamic predicates 'sees' the dynamic predicates at the moment the transaction is started, together with the modifications issued by *Goal*. Thus, *Goal* does not see changes to dynamic predicates from other threads and other threads do not see modifications by *Goal* (*isolation*). If *Goal* succeeds, all modifications become *atomically* visible to the other threads. If *Goal* fails or raises an exception all local modifications are discarded and `transaction/1` fails or passes the exception.

[40] *Read-write* locks also provide readers and writers to make progress simultaneously, but readers see all intermediate states rather than a consistent state.

Currently the number of database changes inside a transaction (or snapshot, see `snapshot/1`) is limited to $2^{32}-1$. If this limit is exceeded a `representation_error(transaction_generations)` exception is raised.

Transactions may be nested. The above mentioned limitation for the number of database changes applies to the combined number in nested transactions.

If *Goal* succeeds, the transaction is *committed*. This implies that (1) any clause that is asserted in the transaction and not retracted in the same transaction is made *globally visible* and (2) and clause the existed before the transaction and is retracted in the transaction becomes *globally invisible*. Multiple transactions may retract the same clause and be committed, i.e., committing a retract that was already performed is a no-op. All modifications become *atomically* visible to other threads. The `transaction/3` variation allows for verifying *constraints* just before the commit takes place.

Clause ordering Inside a transaction clauses can be added using `asserta/1` and `assertz/1`. If only a single transaction is active at any point in time transactions preserve the usual ordering of clauses. However, if multiple transactions manipulate the same predicate(s) concurrently (typically using `transaction/3`), the final order of the clauses is the order in which the transactions asserted the clauses and **not** the order in which the transactions are committed.

The `transaction/1` variant is equivalent to `transaction`(*Goal,[]*). The `transaction/2` variant processed the following options:

bulk(*+Boolean*)
When `true`, accumulate events from changes to dynamic predicates (see `prolog_listen/2`) and trigger these events as part of the commit phase. This implies that if the transaction is not committed the events are never triggered. Failure to trigger the events causes the transaction to be discarded. Experimental.

transaction(*:Goal, :Constraint, +Mutex*)
Similar to `transaction/1`, but allows verifying *Constraint* during the commit phase. This predicate follows the steps below. Any failure or exception during this process discards the transaction and releases *Mutex* when applicable. *Constraint* may modify the database. Such modifications follow the semantics that apply for *Goal*.

- Call `once`(*Goal*)
- Lock *Mutex*
- Change the visibility to the *current* global state combined with the changes made by *Goal*
- Call `once`(*Constraint*)
- Commit the changes
- Unlock *Mutex*.

This predicate is intended to execute multiple transactions with a time consuming *Goal* in part concurrently. For example, it can be used for a *Compare And Swap* (CAS) like design. We illustrate this using a simple counter in the code below. Note that the transaction fails if some other thread concurrently updated the counter. This is why we need the `repeat/0` and a final `!/0`. The CAS-style update is in general useful if *Goal* is expensive and conflicts are rare.

```
:- dynamic counter/1.

increment_counter(Delta) :-
    repeat,
      transaction(( counter(Value),
                    Value2 is Value+Delta,
                  ),
                  ( retract(counter(Value)),
                    asserta(counter(Value2))
                  ),
                  counter_lock),
    !.
```

snapshot(*:Goal*)
Similar to `transaction/1`, but *always* discards the local modifications. In other words, `snapshot/1` allows a thread to examine a frozen state of the dynamic predicates and/or make isolated modifications without affecting other threads and without making permanent changes to the database. Where transactions allow the global state to be updated atomically from one consistent state to the next, a snapshot allows reasoning about a consistent state.

current_transaction(*-Goal*) *[nondet]*
True when called inside a transaction running *Goal*. This predicate generates candidates from the current (nested) transaction outward. *Goal* is a plain goal if the calling context module is the same as matching `transaction/1` or `snapshot/1` and a qualified callable term otherwise. Note that this only enumerates transactions in the current thread.

transaction_updates(*-Updates*)
Unify *Updates* with a list of database updates that would be effectuated if the transaction is going to be committed at this stage. *Updates* is a list of terms defined below. The elements are sorted on the change generation, i.e., the order in which the operations were performed.

asserta(*+ClauseRef*)
assertz(*+ClauseRef*)
The given clause will be asserted at the start or end. Note that due to competing transactions the clause may no longer be the first/last clause of the predicate.

erased(*+ClauseRef*)
The given clause will be removed. This may be due to `erase/1`, `retract/1` or `retractall/1`.

Impact of transactions

Transactions interact with other facilities that depend on changing dynamic predicates. This section discusses these interactions.

Last modified generation Using the `predicate_property/2` property `last_modified_generation`(*Generation*) we can determine whether a predicate was modified. When a predicate is changed inside a transaction this generation is not updated. The generation for dynamic predicates that are modified in the transaction is updated to the *commit generation* when the transaction is committed. Asking for the last modified generation *inside* the transaction examines the log of modified clauses and reports the generation as one of

- The global modified generation if the predicate was not modified in the transaction and not modified outside the transaction to beyond the start generation of the transaction. If the modified generation is higher than the transaction start generation, this generation is reported. [41]
- The transaction start generation plus the local generation of the last change if the predicate is modified inside the transaction.

Wait for database changes The predicate `thread_wait/2` does not wakeup threads for changes inside a transaction. The wakeup is delayed until the transaction is committed. Note that `thread_wait/2` cannot be meaningfully called from inside a transaction because no external entities can cause changes to the dynamic database inside the transaction.

Incremental tabling Consistency of tables must be restored if the transaction is rolled back. For local tables this is realised as follows:

- Tables are either marked to be *invalidated* on rollback or, for *monotonic* tabling individual answers are marked to be removed on rollback.
- A table is marked to be *invalidated* if, while it is created or reevaluated, at least one dependent dynamic predicate has been modified inside the transaction.
- Answers are marked to be retracted when they result from monotonic reevaluation based on changes *inside* the transaction.

In other words: tables being reevaluated inside a transaction that do not depend on predicates modified inside the transaction remain valid. Monotonic tables that get new answers due to asserts inside the transaction have these answers removed during the rollback while the table remains valid. Monotonic tables that are for some reason invalidated inside the transaction are invalidated during the rollback.

Correct interaction between tabling and transaction currently **only deals with local tables**. *Shared* tables should not be combined with transactions. Future versions may improve on that. A possible route is to make a local copy from a shared table when (re)evaluation is performed inside a transaction.

Status SWI-Prolog transaction basics and API are stable. Interaction with other parts of the system that depend on dynamic predicates is still unsettled. Future versions may support non-determinism through transactions and snapshots.

[41] BUG: Note that the above implies that inside a transaction we observe a changing last modified generation for predicates that have only been modified outside the transaction while these changes are not visible.

4.14.2 The recorded database

recorda(+*Key*, +*Term*, -*Reference*)
Assert *Term* in the recorded database under key *Key*. *Key* is a small integer (range `min_tagged_integer` ...`max_tagged_integer`, atom or compound term. If the key is a compound term, only the name and arity define the key. *Reference* is unified with an opaque handle to the record (see `erase/1`).

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

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, Term, _)`.

recorded(?*Key*, ?*Value*, ?*Reference*)
True if *Value* is recorded under *Key* and has the given database *Reference*. If *Reference* is given, this predicate is semi-deterministic. Otherwise, it must be considered non-deterministic. If neither *Reference* nor *Key* is given, the triples are generated as in the code snippet below.[42] See also `current_key/1`.

```
        current_key(Key),
        recorded(Key, Value, Reference)
```

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

erase(+*Reference*)
Erase a record or clause from the database. *Reference* is a db-reference returned by `recorda/3`, `recordz/3` or `recorded/3`, `clause/3`, `assert/2`, `asserta/2` or `assertz/2`. Fail silently if the referenced object no longer exists. Notably, if multiple threads attempt to erase the same clause one will succeed and the others will fail.

instance(+*Reference*, -*Term*)
Unify *Term* with the referenced clause or database record. Unit clauses are represented as *Head* :- `true`.

4.14.3 Flags

The predicate `flag/3` is the oldest way to store global non-backtrackable data in SWI-Prolog. Flags are global and shared by all threads. Their value is limited to atoms, small (64-bit) integers and floating point numbers. Flags are thread-safe. The flags described in this section must not be confused with *Prolog flags* described in section 2.12.

[42] Note that, without a given *Key*, some implementations return triples in the order defined by `recorda/2` and `recordz/2`.

get_flag(*+Key, -Value*)
True when *Value* is the value currently associated with *Key*. If *Key* does not exist, a new flag with value '0' (zero) is created.

set_flag(*+Key, Value*)
Set flag *Key* to *Value*. Value must be an atom, small (64-bit) integer or float.

flag(*+Key, -Old, +New*)
True when *Old* is the current value of the flag *Key* and the flag has been set to *New*. *New* can be an arithmetic expression. The update is *atomic*. This predicate can be used to create a *shared* global counter as illustrated in the example below.

```
next_id(Id) :-
    flag(my_id, Id, Id+1).
```

4.14.4 Tries

Tries (also called *digital tree*, *radix tree* or *prefix tree* maintain a mapping between a variant of a term (see `=@=/2`) and a value. They have been introduced in SWI-Prolog 7.3.21 as part of the implementation of *tabling*. The current implementation is rather immature. In particular, the following limitations currently apply:

- Tries are not thread-safe.
- Tries should not be modified while non-deterministic predicates such as `trie_gen/3` are running on the trie.
- Terms cannot have *attributed variables*.
- Terms cannot be *cyclic*. Possibly this will not change because cyclic terms can only be supported after creating a canonical form of the term.

We give the definition of these predicates for reference and debugging tabled predicates. Future versions are likely to get a more stable and safer implementation. The API to tries should not be considered stable.

trie_new(*-Trie*)
Create a new trie and unify *Trie* with a handle to the trie. The trie handle is a *blob*. Tries are subject to atom garbage collection.

trie_destroy(*+Trie*)
Destroy *Trie*. This removes all nodes from the trie and causes further access to *Trie* to raise an existence_error exception. The handle itself is reclaimed by atom garbage collection.

is_trie(*@Trie*) *[semidet]*
True when *Trie* is a trie object. See also `current_trie/1`.

current_trie(*-Trie*) *[nondet]*
True if *Trie* is a currently existing trie. As this enumerates and then filters all known atoms this predicate is slow and should only be used for debugging purposes. See also `is_trie/1`.

trie_insert(+*Trie*, +*Key*)
Insert the term *Key* into *Trie*. If *Key* is already part of *Trie* the predicates *fails* silently. This is the same as `trie_insert/3`, but using a fixed reserved *Value*.

trie_insert(+*Trie*, +*Key*, +*Value*)
Insert the term *Key* into *Trie* and associate it with *Value*. *Value* can be any term. If *Key*-*Value* is already part of *Trie*, the predicates *fails* silently. If *Key* is in *Trie* associated with a different value, a `permission_error` is raised.

trie_update(+*Trie*, +*Key*, +*Value*)
As `trie_insert/3`, but if *Key* is in *Trie*, its associated value is *updated*.

trie_insert(+*Trie*, +*Term*, +*Value*, -*Handle*)
As `trie_insert/3`, returning a handle to the trie node. This predicate is currently unsafe as *Handle* is an integer used to encode a pointer. It was used to implement a pure Prolog version of the `tabling` library.

trie_delete(+*Trie*, +*Key*, ?*Value*)
Delete *Key* from *Trie* if the value associated with *Key* unifies with *Value*.

trie_lookup(+*Trie*, +*Key*, -*Value*)
True if the term *Key* is in *Trie* and associated with *Value*.

trie_term(+*Handle*, -*Term*)
True when *Term* is a copy of the term associated with *Handle*. The result is undefined (including crashes) if *Handle* is not a handle returned by `trie_insert_new/3` or the node has been removed afterwards.

trie_gen(+*Trie*, ?*Key*) *[nondet]*
True when *Key* is a member of *Trie*. See also `trie_gen_compiled/2`.

trie_gen(+*Trie*, ?*Key*, -*Value*) *[nondet]*
True when *Key* is associated with *Value* in *Trie*. Backtracking retrieves all pairs. Currently scans the entire trie, even if *Key* is partly known. Currently unsafe if *Trie* is modified while the values are being enumerated. See also `trie_gen_compiled/3`.

trie_gen_compiled(+*Trie*, ?*Key*) *[nondet]*
trie_gen_compiled(+*Trie*, ?*Key*, -*Value*) *[nondet]*
Similar to `trie_gen/3`, but uses a *compiled* representation of *Trie*. The compiled representation is created lazily and manipulations of the trie (insert, delete) invalidate the current compiled representation. The compiled representation generates answers faster and, as it runs on a snapshot of the trie, is immune to concurrent modifications of the trie. This predicate is used to generate answers from *answer tries* as used for tabled execution. See section 7.

trie_property(?*Trie*, ?*Property*) *[nondet]*
True if *Trie* exists with *Property*. Intended for debugging and statistical purposes. Retrieving some of these properties visit all nodes of the trie. Defined properties are

value_count(-*Count*)
Number of key-value pairs in the trie.

node_count(*-Count*)
: Number of nodes in the trie.

size(*-Bytes*)
: Required storage space of the trie.

compiled_size(*-Bytes*)
: Required storage space for the compiled representation as used by `trie_gen_compiled/2,3`.

hashed(*-Count*)
: Number of nodes that use a hashed index to its children.

lookup_count(*-Count*)
: Number of `trie_lookup/3` calls (only when compiled with `O_TRIE_STATS`).

gen_call_count(*-Count*)
: Number of `trie_gen/3` calls (only when compiled with `O_TRIE_STATS`).

wait(*-Count*)
: Number of times a thread waited on this trie for another thread to complete it (shared tabling, only when compiled with `O_TRIE_STATS`).

deadlock(*-Count*)
: Number of times this trie was part of a deadlock and its completion was abandoned (shared tabling, only when compiled with `O_TRIE_STATS`).

In addition, a number of additional properties are defined on *answer tries*.

invalidated(*-Count*)
: Number of times the trie was invalidated (incremental tabling).

reevaluated(*-Count*)
: Number of times the trie was re-evaluated (incremental tabling).

idg_affected_count(*-Count*)
: Number of answer tries affected by this one (incremental tabling).

idg_dependent_count(*-Count*)
: Number of answer tries this one depends on (incremental tabling).

idg_size(*-Bytes*)
: Number of bytes in the IDG node representation.

4.14.5 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 to 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'.[43] 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

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

database. Each goal is tagged with the generation in which it was started. Each clause is flagged with the generation it was created in as well as the generation it was erased from. Only clauses with a 'created' ... 'erased' interval that encloses the generation of the current goal are considered visible.

4.14.6 Indexing databases

The indexing capabilities of SWI-Prolog are described in section 2.18. Summarizing, SWI-Prolog creates indexes for any applicable argument, pairs of arguments and indexes on the arguments of compound terms when applicable. Extended JIT indexing is not widely supported among Prolog implementations. Programs that aim at portability should consider using `term_hash/2` and `term_hash/4` to design their database such that indexing on constant or functor (name/arity reference) on the first argument is sufficient. In some cases, using the predicates below to add one or more additional columns (arguments) to a database predicate may improve performance. The overall design of code using these predicates is given below. Note that as `term_hash/2` leaves the hash unbound if *Term* is not ground. This causes the lookup to be fast if *Term* is ground and correct (but slow) otherwise.

```
:- dynamic
    x/2.

assert_x(Term) :-
    term_hash(Term, Hash),
    assertz(x(Hash, Term)).

x(Term) :-
    term_hash(Term, Hash),
    x(Hash, Term).
```

term_hash(+*Term*, -*HashKey*) *[det]*

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 leaves *HashKey* an unbound variable. Hash keys are in the range $0 \ldots 16,777,215$, the maximal integer that can be stored efficiently on both 32 and 64 bit platforms.

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.[44] Hashes differ between big and little endian machines. The `term_hash/2` predicate is cycle-safe.[45]

term_hash(+*Term*, +*Depth*, +*Range*, -*HashKey*) *[det]*

As `term_hash/2`, but only considers *Term* to the specified *Depth*. The top-level term has depth 1, its arguments have depth 2, etc. That is, $Depth = 0$ hashes nothing; $Depth = 1$ hashes atomic values or the functor and arity of a compound term, not its arguments; $Depth = 2$ also indexes the immediate arguments, etc.

[44] Last change: version 5.10.4

[45] BUG: All arguments that (indirectly) lead to a cycle have the same hash key.

HashKey is in the range $[0 \ldots \mathit{Range} - 1]$. *Range* must be in the range $[1 \ldots 2147483647]$.

variant_sha1(*+Term, -SHA1*) *[det]*
Compute a SHA1-hash from *Term*. The hash is represented as a 40-byte hexadecimal atom. Unlike `term_hash/2` and friends, this predicate produces a hash key for non-ground terms. The hash is invariant over variable-renaming (see `=@=/2`) and constants over different invocations of Prolog.[46]

This predicate raises an exception when trying to compute the hash on a cyclic term or attributed term. Attributed terms are not handled because `subsumes_chk/2` is not considered well defined for attributed terms. Cyclic terms are not supported because this would require establishing a canonical cycle. That is, given A=[a—A] and B=[a,a—B], *A* and *B* should produce the same hash. This is not (yet) implemented.

This hash was developed for lookup of solutions to a goal stored in a table. By using a cryptographic hash, heuristic algorithms can often ignore the possibility of hash collisions and thus avoid storing the goal term itself as well as testing using `=@=/2`.

variant_hash(*+Term, -HashKey*) *[det]*
Similar to `variant_sha1/2`, but using a non-cryptographic hash and produces an integer result like `term_hash/2`. This version does deal with attributed variables, processing them as normal variables. This hash is primarily intended to speedup finding variant terms in a set of terms. [47]

4.15 Declaring predicate properties

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

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

In 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.

Notably with the introduction of tabling (see section 7) it is common that a set of predicates require multiple options to be set. SWI-Prolog offers two mechanisms to cope with this. The predicate `dynamic/2` can be used to make a list of predicates dynamic and set additional options. In addition and for compatibility with XSB,[48] all the predicates below accept a term `as`(*(:PredicateIndicator, ...), (+Options)*), where *Options* is a *comma-list* of one of more of the following options:

incremental
Include a dynamic predicate into the incremental tabling dependency graph. See section 7.7.

[46]BUG: The hash depends on word order (big/little-endian) and the wordsize (32/64 bits).
[47]BUG: As `variant_sha1/2`, cyclic terms result in an exception.
[48]Note that `as` is in XSB a high-priority operator and in SWI a low-priority and therefore both the sets of predicate indicators as multiple options require parenthesis.

opaque
Opposite of `incremental`. For XSB compatibility.[49]

abstract(*Level*)
Used together with `incremental` to reduce the dependency graph. See section 7.7.

volatile
Do not save this predicate. See `volatile/1`.

multifile
Predicate may have clauses in multiple clauses. See `multifile/1`.

discontiguous
Predicate clauses may not be contiguous in the file. See `discontiguous/1`.

shared
Dynamic predicate is shared between all threads. This is currently the default.

local
private
Dynamic predicate has distinct set of clauses in each thread. See `thread_local/1`.

Below are some examples, where the last two are semantically identical.

```
:- dynamic person/2 as incremental.
:- dynamic (person/2,organization/2) as (incremental, abstract(0)).
:- dynamic([ person/2,
             organization/2
           ],
           [ incremental(true),
             abstract(0)
           ]).
```

dynamic *:PredicateIndicator, . . .* *[ISO]*
Informs the interpreter that the definition of the predicate(s) may change during execution (using `assert/1` and/or `retract/1`). In the multithreaded version, the clauses of dynamic predicates are shared between the threads. The directive `thread_local/1` provides an alternative where each thread has its own clause list for the predicate. Dynamic predicates can be turned into static ones using `compile_predicates/1`.

dynamic(*:ListOfPredicateIndicators, +Options*)
As `dynamic/1`, but allows for setting additional properties. This predicate allows for setting multiple properties on multiple predicates in a single call. SWI-Prolog also offers the XSB compatible `:- dynamic (p/1) as (incremental,abstract(0)).` syntax. See the introduction of section 4.15. Defined *Options* are:

[49] In XSB, `opaque` is distinct from the default in the sense that dynamic switching between `opaque` and `incremental` is allowed.

incremental(*+Boolean*)
Make the dynamic predicate signal depending *tables*. See section 7.7.

abstract(*0*)
This option must be used together with `incremental`. The only supported value is `0`. With this option a call to the incremental dynamic predicate is recorded as the most generic term for the predicate rather than the specific variant.

thread(*+Local*)
Local is one of `shared` (default) or `local`. See also `thread_local/1`.

multifile(*+Boolean*)
discontiguous(*+Boolean*)
volatile(*+Boolean*)
Set the corresponding property. See `multifile/1`, `discontiguous/1` and `volatile/1`.

compile_predicates(*:ListOfPredicateIndicators*)
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 are generated at runtime but do not change during the remainder of the program execution.[50]

multifile *:PredicateIndicator, ...* *[ISO]*
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 *:PredicateIndicator, ...* *[ISO]*
Informs the system that the clauses of the specified predicate(s) might not be together in the source file. See also `style_check/1`.

public *:PredicateIndicator, ...*
Instructs the cross-referencer that the predicate can be called. It has no semantics.[51] The public declaration can be queried using `predicate_property/2`. The `public/1` directive does *not* export the predicate (see `module/1` and `export/1`). The public directive is used for (1) direct calls into the module from, e.g., foreign code, (2) direct calls into the module from other modules, or (3) flag a predicate as being called if the call is generated by meta-calling constructs that are not analysed by the cross-referencer.

non_terminal *:PredicateIndicator, ...*
Sets the `non_terminal` property on the predicate. This indicates that the predicate implements a *grammar rule*. See `predicate_property/2`. The `non_terminal` property is set for predicates exported as *Name//Arity* as well as predicates that have at least one clause written using the `-->/2` notation.

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

[51]This declaration is compatible with SICStus. In YAP, `public/1` instructs the compiler to keep the source. As the source is always available in SWI-Prolog, our current interpretation also enhances the compatibility with YAP.

4.16 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 12.4.8 for details.

current_functor(*?Name, ?Arity*)

True when *Name/Arity* is a known functor. This means that at some point in time a term with name *Name* and *Arity* arguments was created. Functor objects are currently not subject to garbage collection. Due to timing, `t/2` below with instantiated *Name* and *Arity* can theoretically fail, i.e., a functor may be visible in instantiated mode while it is not yet visible in unbound mode. Considering that the only practical value of `current_functor/2` we are aware of is to analyse resource usage we accept this impure behaviour.

```
t(Name, Arity) :-
    (   current_functor(Name, Arity)
    ->  current_functor(N, A), N == Name, A == Arity
    ;   true
    ).
```

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(*:PredicateIndicator*) *[ISO]*

True if *PredicateIndicator* is a currently defined predicate. A predicate is considered defined if it exists in the specified module, is imported into the module or is defined in one of the modules from which the predicate will be imported if it is called (see section 6.10). Note that `current_predicate/1` does *not* succeed for predicates that can be *autoloaded* unless they are imported using `autoload/2`. See also `current_predicate/2` and `predicate_property/2`.

If *PredicateIndicator* is not fully specified, the predicate only generates values that are defined in or already imported into the target module. Generating all callable predicates therefore requires enumerating modules using `current_module/1`. Generating predicates callable in a given module requires enumerating the import modules using `import_module/2` and the autoloadable predicates using the `predicate_property/2` `autoload`.

current_predicate(*?Name, :Head*)

Classical pre-ISO implementation of `current_predicate/1`, where the predicate is represented by the head term. The advantage is that this can be used for checking the existence of a predicate before calling it without the need for `functor/3`:

```
call_if_exists(G) :-
        current_predicate(_, G),
        call(G).
```

Because of this intended usage, `current_predicate/2` also succeeds if the predicate can be autoloaded. Unfortunately, checking the autoloader makes this predicate relatively slow, in particular because a failed lookup of the autoloader will cause the autoloader to verify that its index is up-to-date.

predicate_property(*:Head, ?Property***)**

True when *Head* refers to a predicate that has property *Property*. With sufficiently instantiated *Head*, `predicate_property/2` tries to resolve the predicate the same way as calling it would do: if the predicate is not defined it scans the default modules (see `default_module/2`) and finally tries the autoloader. Unlike calling, failure to find the target predicate causes `predicate_property/2` to fail silently. If *Head* is not sufficiently bound, only currently locally defined and already imported predicates are enumerated. See `current_predicate/1` for enumerating all predicates. A common issue concerns *generating* all built-in predicates. This can be achieved using the code below:

```
generate_built_in(Name/Arity) :-
    predicate_property(system:Head, built_in),
    functor(Head, Name, Arity),
    \+ sub_atom(Name, 0, _, _, $).   % discard reserved names
```

The predicate `predicate_property/2` is covered by part-II of the ISO standard (modules). Although we are not aware of any Prolog system that implements part-II of the ISO standard, `predicate_property/2` is available in most systems. There is little consensus on the implemented properties though. SWI-Prolog's *auto loading* feature further complicate this predicate.

Property is one of:

autoload(*File***)**

True if the predicate can be autoloaded from the file *File*. Like `undefined`, this property is *not* generated.

built_in

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.

defined

True if the predicate is defined. This property is aware of sources being *reloaded*, in which case it claims the predicate defined only if it is defined in another source or it has seen a definition in the current source. See `compile_aux_clauses/1`.

det

The predicate is defined to be deterministic using `det/1`.

discontiguous
True after `discontiguous/1` was used to flag that the clauses of the predicates may not be contiguous.

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

exported
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` and the property `line_count`. Note that this reports the file of the first clause of a predicate. A more robust interface can be achieved using `nth_clause/3` and `clause_property/2`.

foreign
True if the predicate is defined in the C language.

implementation_module(*-Module*)
True when *Module* is the module in which *Head* is or will be defined. Resolving this property goes through the same search mechanism as when an undefined predicate is encountered, but does not perform any loading. It searches (1) the module inheritance hierarchy (see `default_module/2`) and (2) the autoload index if the `unknown` flag is not set to `fail` in the target module.

indexed(*Indexes*)
Indexes is a list of additional (hash) indexes on the predicate. Each element of the list is a term *ArgSpec-Index*. *ArgSpec* denotes the indexed argument(s) and is one of

single(*Argument*)
Hash on a single argument. *Argument* is the 1-based argument number.

multi(*ArgumentList*)
Hash on a combination of arguments.

deep(*Position*)
Index on a sub-argument. Position is a list holding first the argument of the predicate then the argument into the compound and recursively into deeper compound terms.

Index is a term `hash`(*Buckets, Speedup, Size, IsList*). Here *Buckets* is the number of buckets in the hash and *Speedup* is the expected speedup relative to trying all clauses linearly, *Size* is the size of the index in memory in bytes and finally, *IsList* indicates that a list is created for all clauses with the same key. This is used to create *deep indexes* for the arguments of compound terms.

Note: This predicate property should be used for analysis and statistics only. The exact representation of *Indexes* may change between versions. The utilities `jiti_list/0` `jiti_list/1` list the *jit* indexes of matching predicates in a user friendly way.

interpreted
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.

iso
True if the predicate is covered by the ISO standard (ISO/IEC 13211-1).

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`. See also the `file` property above, notably the reference to `clause_property/2`.

multifile
True if there may be multiple (or no) files providing clauses for the predicate. This property is set using `multifile/1`.

meta_predicate(*Head*)
If the predicate is declared as a meta-predicate using `meta_predicate/1`, unify *Head* with the head-pattern. The head-pattern is a compound term with the same name and arity as the predicate where each argument of the term is a meta-predicate specifier. See `meta_predicate/1` for details.

monotonic
True if the predicate is tabled or dynamic using monotonic propagation. See section 7.8.

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`.

non_terminal
True if the predicate implements a *grammar rule*. See `non_terminal/1`.

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. This property respects the *logical update view* and counts visible clauses at the moment the predicate was started.

number_of_rules(*RuleCount*)
Similar to `number_of_clauses`(*ClauseCount*), but only counts *rules*. A *rule* is defined as a clauses that has a body that is not just `true` (i.e., a *fact*).

last_modified_generation(*Generation*)
Database generation at which the predicate was modified for the last time. Intended to quickly assesses the validity of caches.

opaque
This property applies to dynamic and tabled predicates. For dynamic predicates it (temporary) stops propagating updates to dependent incrementally or monotonic tabled predicates. For tabled predicates it is not an error for an opaque predicate to depend on incremental or monotonic dynamic or tabled predicates.

public
Predicate is declared public using `public/1`. Note that without further definition, public predicates are considered undefined and this property is *not* reported.

quasi_quotation_syntax
The predicate (with arity 4) is declared to provide quasi quotation syntax with `quasi_quotation_syntax/1`.

size(*Bytes*)
Memory used for this predicate. This includes the memory of the predicate header, the combined memory of all clauses including erased but not yet garbage collected clauses (see `garbage_collect_clauses/0` and `clause_property/2`) and the memory used by clause indexes (see the `indexed`(*Indexes*) property. *Excluded* are *lingering* data structures. These are garbage data structures that have been detached from the predicate but cannot yet be reclaimed because they may be in use by some thread.

ssu
The predicate has been defined using *single sided unification* rules. See section 5.6.

static
The definition can *not* be modified using `assertz/1` and friends. This property is the opposite from `dynamic`, i.e., for each defined predicate, either `static` or `dynamic` is true but never both.

tabled
True of the predicate is *tabled*. The `tabled`(*?Flag*) property can be used to obtain details about how the predicate is tabled.

tabled(*?Flag*)
True of the predicate is *tabled* and *Flag* applies. Any tabled predicate has one of the mutually exclusive flags `variant` or `subsumptive`. In addition, tabled predicates may have one or more of the following flags

shared
The table is shared between threads. See section 7.9.

incremental
The table is subject to *incremental tabling*. See section 7.7

Use the `tabled` property to enumerate all tabled predicates. See `table/1` for details.

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

transparent
True if the predicate is declared transparent using the `module_transparent/1` or `meta_predicate/1` declaration. In the latter case the property `meta_predicate`(*Head*) is also provided. See chapter 6 for details.

undefined
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, the predicate has been declared as e.g., a meta-predicate or the predicate had a definition in the past. Originally used to find missing predicate definitions. The current implementation of `list_undefined/0` used cross-referencing. Deprecated.

visible
True when predicate can be called without raising a predicate existence error. This means

that the predicate is (1) defined, (2) can be inherited from one of the default modules (see `default_module/2`) or (3) can be autoloaded. The behaviour is logically consistent iff the property `visible` is provided explicitly. If the property is left unbound, only defined predicates are enumerated.

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, whose 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 the access level is `system` (see `access_level`). Backtracking provides all alternative matches.

clause(*:Head, ?Body*) *[ISO]*
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*.

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 indexes and the references of all clauses of the predicate. The following example finds the 2nd clause of `append/3`:

```
?- use_module(library(lists)).
...
?- nth_clause(append(_,_,_), 2, Ref), clause(Head, Body, Ref).
Ref = <clause>(0x994290),
Head = lists:append([_G23|_G24], _G21, [_G23|_G27]),
Body = append(_G24, _G21, _G27).
```

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`. Unlike most other predicates that access clause references, `clause_property/2` may be used to get information about erased clauses that have not yet been reclaimed. *Property* is one of the following:

file(*FileName*)
Unify *FileName* with the name of the file from which the clause is loaded. Fails if the clause was not created by loading a file (e.g., clauses added using `assertz/1`). See also `source`.

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

size(*SizeInBytes*)
True when *SizeInBytes* is the size that the clause uses in memory in bytes. The size required by a predicate also includes the predicate data record, a linked list of clauses, clause selection instructions and optionally one or more clause indexes.

source(*FileName*)
Unify *FileName* with the name of the source file that created the clause. This is the same as the `file` property, unless the file is loaded from a file that is textually included into source using `include/1`. In this scenario, `file` is the included file, while the `source` property refers to the *main* 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.

predicate(*PredicateIndicator*)
PredicateIndicator denotes the predicate to which this clause belongs. This is needed to obtain information on erased clauses because the usual way to obtain this information using `clause/3` fails for erased clauses.

module(*Module*)
Module is the context module used to execute the body of the clause. For normal clauses, this is the same as the module in which the predicate is defined. However, if a clause is compiled with a module qualified *head*, the clause belongs to the predicate with the qualified head, while the body is executed in the context of the module in which the clause was defined.

4.17 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.[52] 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 4.17.3 describes `tell/1`, `see/1` and friends, providing I/O in the spirit of the traditional Edinburgh standard. These predicates are layered on top of the ISO predicates. Both packages are fully integrated; the user may switch freely between them.

[52] Actually based on Quintus Prolog, providing this interface before the ISO standard existed.

4.17.1 Predefined stream aliases

Each thread has five stream aliases: `user_input`, `user_output`, `user_error`, `current_input`, and `current_output`. Newly created threads inherit these stream aliases from their parent. The `user_input`, `user_output` and `user_error` aliases of the `main` thread are initially bound to the standard operating system I/O streams (*stdin*, *stdout* and *stderr*, normally bound to the POSIX file handles 0, 1 and 2). These aliases may be re-bound, for example if standard I/O refers to a window such as in the `swipl-win.exe` GUI executable for Windows. They can be re-bound by the user using `set_prolog_IO/3` and `set_stream/2` by setting the alias of a stream (e.g, `set_stream(S, alias(user_output))`). An example of rebinding can be found in library `prolog_server`, providing a `telnet` service. The aliases `current_input` and `current_output` define the source and destination for predicates that do not take a stream argument (e.g., `read/1`, `write/1`, `get_code/1`, ...). Initially, these are bound to the same stream as `user_input` and `user_error`. They are re-bound by `see/1`, `tell/1`, `set_input/1` and `set_output/1`. The `current_output` stream is also temporary re-bound by `with_output_to/2` or `format/3` using e.g., `format(atom(A), ...`. Note that code which explicitly writes to the streams `user_output` and `user_error` will not be redirected by `with_output_to/2`.

Compatibility Note that the ISO standard only defines the `user_*` streams. The 'current' streams can be accessed using `current_input/1` and `current_output/1`. For example, an ISO compatible implementation of `write/1` is

```
write(Term) :- current_output(Out), write_term(Out, Term).
```

while SWI-Prolog additionally allows for

```
write(Term) :- write(current_output, Term).
```

4.17.2 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. On the other hand, using the notion of current-I/O simplifies reusability of code without the need to pass arguments around. E.g., see `with_output_to/2`.

SWI-Prolog streams are, compatible with the ISO standard, either input or output streams. To accommodate portability to other systems, a pair of streams can be packed into a *stream-pair*. See `stream_pair/3` for details.

SWI-Prolog stream handles are unique symbols that have no syntactical representation. They are written as `<stream>(hex-number)`, which is not valid input for `read/1`. They are realised using a *blob* of type `stream` (see `blob/2` and section 12.4.8).

open(*+SrcDest, +Mode, –Stream, +Options*) *[ISO]*

True when *SrcDest* can be opened in *Mode* and *Stream* is an I/O stream to/from the object.

SrcDest is normally the name of a file, represented as an atom or string. *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.[53]

SWI-Prolog also allows *SrcDest* to be a term `pipe`(*Command*). In this form, *Command* is started as a child process and if *Mode* is `write`, output written to *Stream* is sent to the standard input of *Command*. Vice versa, if *Mode* is `read`, data written by *Command* to the standard output may be read from *Stream*. On Unix systems, *Command* is handed to popen() which hands it to the Unix shell. On Windows, *Command* is executed directly. See also `process_create/3` from `process`.

If *SrcDest* is an *IRI*, i.e., starts with ⟨*scheme*⟩`://`, where ⟨*scheme*⟩ is a non-empty sequence of lowercase ASCII letters `open/3`,4 calls hooks registered by `register_iri_scheme/3`. Currently the only predefined IRI scheme is `res`, providing access to the *resource database*. See section 13.4.

The following *Options* are recognised by `open/4`:

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),
        ...
```

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 2.19.1 for a discussion of this feature.

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, however, it will be flushed. Useful for logfiles and if the stream is associated to a process (using the `pipe/1` construct).

create(*+List*)
: Specifies how a new file is created when opening in `write`, `append` or `update` mode. Currently, *List* is a list of atoms that describe the permissions of the created file.[54] Defined

[53] New code should use the `alias`(*Alias*) option for compatibility with the ISO standard.

[54] Added after feedback from Joachim Shimpf and Per Mildner.

values are below. Not recognised values are silently ignored, allowing for adding platform specific extensions to this set.

read
Allow read access to the file.

write
Allow write access to the file.

execute
Allow execution access to the file.

default
Allow read and write access to the file.

all
Allow any access provided by the OS.

Note that if *List* is empty, the created file has no associated access permissions. The create options map to the POSIX *mode* option of open(), where `read` map to 0444, `write` to 0222 and `execute` to 0111. On POSIX systems, the final permission is defined as (mode & ~umask).

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 2.19.1.

eof_action(*Action*)
Defines what happens if the end of the input stream is reached. The default value for Action is `eof_code`, which 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.

locale(+*Locale*)
Set the locale that is used by notably `format/2` for output on this stream. See section 4.23.

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.

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`.

wait(*Bool*)
This option can be combined with the `lock` option. If `false` (default `true`), the open call returns immediately with an exception if the file is locked. The exception has the format `permission_error`(*lock, source_sink, SrcDest*).

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

open(*+SrcDest, +Mode, –Stream*) *[ISO]*
Equivalent to `open/4` with an empty option list.

open_null_stream(*–Stream*)
Open an output stream that produces no output. All counting functions are enabled on such a stream. It can be used to discard output (like Unix `/dev/null`) or exploit the counting properties. The initial encoding of *Stream* is `utf8`, enabling arbitrary Unicode output. The encoding can be changed to determine byte counts of the output in a particular encoding or validate if output is possible in a particular encoding. For example, the code below determines the number of characters emitted when writing *Term*.

```
write_length(Term, Len) :-
        open_null_stream(Out),
        write(Out, Term),
        character_count(Out, Len0),
        close(Out),
        Len = Len0.
```

close(*+Stream*) *[ISO]*
Close the specified stream. If *Stream* is not open, an existence error is raised. See `stream_pair/3` for the implications of closing a *stream pair*.

If the closed stream is the current input, output or error stream, the stream alias is bound to the initial standard I/O streams of the process. Calling `close/1` on the initial standard I/O streams of the process is a no-op for an input stream and flushes an output stream without closing it.[55]

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

stream_property(*?Stream, ?StreamProperty*) *[ISO]*
True when *StreamProperty* is a property of *Stream*. If enumeration of streams or properties is demanded because either *Stream* or *StreamProperty* are unbound, the implementation enumerates all candidate streams and properties while locking the stream database. Properties are fetched without locking the stream and may be outdated before this predicate returns due to asynchronous activity.

[55] This behaviour was defined with purely interactive usage of Prolog in mind. Applications should not count on this behaviour. Future versions may allow for closing the initial standard I/O streams.

alias(*Atom*)
If *Atom* is bound, test if the stream has the specified alias. Otherwise unify *Atom* with the first alias of the stream.[56]

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`.

buffer_size(*Integer*)
SWI-Prolog extension to query the size of the I/O buffer associated to a stream in bytes. Fails if the stream is not buffered.

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 2.19.1 for details.

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

close_on_exec(*Bool*)
Determine whether or not the stream is closed when executing a new process (exec() in Unix, CreateProcess() in Windows). Default is to close streams. This maps to fcntl() `F_SETFD` using the flag `FD_CLOEXEC` on Unix and (negated) `HANDLE_FLAG_INHERIT` on Windows.

encoding(*Encoding*)
Query the encoding used for text. See section 2.19.1 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`.

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

locale(*Locale*)
True when *Locale* is the current locale associated with the stream. See section 4.23.

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`.

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

newline(*NewlineMode*)
One of `posix` or `dos`. If `dos`, text streams will emit `\r\n` for `\n` and discard `\r` from input streams. Default depends on the operating system.

nlink(*-Count*)
Number of hard links to the file. This expresses the number of 'names' the file has. Not supported on all operating systems and the value might be bogus. See the documentation of fstat() for your OS and the value `st_nlink`.

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

position(*Pos*)
Unify *Pos* 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 an I/O error exception), `prolog` (write `\x<hex>\`), `unicode` (write `\uXXXX` or `\UXXXXXXXX` escape sequences) or `xml` (write `&#...;` XML character entity). The initial mode is `unicode` for the user streams and `error` for all other streams. See also section 2.19.1 and `set_stream/2`.

timeout(*-Time*)
Time is the timeout currently associated with the stream. See `set_stream/2` with the same option. If no timeout is specified, *Time* is unified to the atom `infinite`.

type(*Type*)
Unify *Type* with `text` or `binary`.

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

write_errors(*Atom*)
Atom is one of `error` (default) or `ignore`. The latter is intended to deal with service processes for which the standard output handles are not connected to valid streams. In these cases write errors may be ignored on `user_error`.

current_stream(*?Object, ?Mode, ?Stream*) *[deprecated]*
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`.

This predicate is deprecated. New code should use the ISO predicate `stream_property/2`.

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.

stream_pair(*?StreamPair, ?Read, ?Write*)
This predicate can be used in mode (-,+,+) to create a *stream-pair* from an input stream and an output stream. Mode (+,-,-) can be used to get access to the underlying streams. If a stream has already been closed, the corresponding argument is left unbound. If mode (+,-,-) is used on a single stream, either *Read* or *Write* is unified with the stream while the other argument is left unbound. This behaviour simplifies writing code that must operate both on streams and stream pairs.

Stream-pairs can be used by all I/O operations on streams, where the operation selects the appropriate member of the pair. The predicate `close/1` closes the still open streams of the pair.[57] The output stream is closed before the input stream. If closing the output stream results in an error, the input stream is still closed. Success is only returned if both streams were closed successfully.

set_stream_position(+*Stream*, +*Pos*) *[ISO]*
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*, +*Pos*, *-Data*)
Extracts information from the opaque stream position term as returned by `stream_property/2` requesting the `position`(*Pos*) 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`.[58]

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.

Positions are counted in 'units'. A unit is 1 byte, except for text files using 2-byte Unicode encoding (2 bytes) or *wchar* encoding (sizeof(wchar_t)). The latter guarantees comfortable interaction with wide-character text objects. Otherwise, the use of `seek/4` on non-binary files (see `open/4`) is of limited use, especially when using multi-byte text encodings (e.g. UTF-8) or multi-byte newline files (e.g. DOS/Windows). On text files, SWI-Prolog offers reliable backup to an old position using `stream_property/2` and `set_stream_position/2`. Skipping N character codes is achieved calling `get_code/2` N times or using `copy_stream_data/3`, directing the output to a null stream (see `open_null_stream/1`). 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 `permission_error` is raised. If applying the offset would result in a file position less than zero, a `domain_error` is raised. Behaviour when seeking to positions beyond the size of the underlying object depend on the object and possibly the operating system. The predicate `seek/4` is compatible with Quintus Prolog, though

[57] As of version 7.1.19, it is allowed to close one of the members of the stream directly and close the pair later.
[58] Introduced in version 5.6.4 after extending the position term with a byte count. Compatible with SICStus Prolog.

the error conditions and signalling is ISO compliant. See also `stream_property/2` and `set_stream_position/2`.

set_stream(+*Stream*, +*Attribute*)
Modify an attribute of an existing stream. *Attribute* specifies the stream property to set. If stream is a *pair* (see `stream_pair/3`) both streams are modified, unless the property is only meaningful on one of the streams or setting both is not meaningful. In particular, `eof_action` only applies to the *read* stream, `representation_errors` only applies to the *write* stream and trying to set `alias` or `line_position` on a pair results in a `permission_error` exception. 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, 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`.

buffer_size(+*Size*)
Set the size of the I/O buffer of the underlying stream to *Size* bytes.

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

close_on_exec(*Bool*)
Set the `close_on_exec` property. See `stream_property/2`.

encoding(*Atom*)
Defines the mapping between bytes and character codes used for the stream. See section 2.19.1 for supported encodings. The value `bom` causes the stream to check whether the current character is a Unicode BOM marker. If a BOM marker is found, the encoding is set accordingly and the call succeeds. Otherwise the call fails.

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

file_name(*FileName*)
Set the filename 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.

line_position(*LinePos*)
Set the line position attribute of the stream. This feature is intended to correct position management of the stream after sending a terminal escape sequence (e.g., setting ANSI character attributes). Setting this attribute raises a permission error if the stream does not record positions. See `line_position/2` and `stream_property/2` (property `position`).

locale(+*Locale*)
Change the locale of the stream. See section 4.23.

newline(*NewlineMode*)
Set input or output translation for newlines. See corresponding `stream_property/2` for details. In addition to the detected modes, an input stream can be set in mode `detect`. It will be set to `dos` if a `\r` character was removed.

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*`), _)`

type(*Type*)
Set the type of the stream to one of `text` or `binary`. See also `open/4` and the `encoding` property of streams. Switching to `binary` sets the encoding to `octet`. Switching to `text` sets the encoding to the default text encoding.

record_position(*Bool*)
Do/do not record the line count and line position (see `line_count/2` and `line_position/2`). Calling `set_stream(S, record_position(true))` resets the position the start of line 1.

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 2.19.1.

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.

set_system_IO(*+In, +Out, +Error*)
Bind the given streams to the operating system I/O streams 0-2 using POSIX dup2() API. *In* becomes `stdin`. *Out* becomes `stdout`. If *Error* equals *Out* an unbuffered stream is associated to the same destination and linked to `stderr`. Otherwise *Error* is used for `stderr`. Output buffering for *Out* is set to line and buffering on *Error* is disabled. The operating system I/O streams are shared across all threads. The three streams must be related to a *file descriptor* or a `domain_error file_stream` is raised. See also `stream_property/2`, property `file_no`(*Fd*).

Where set_prolog_IO/3 rebinds the Prolog streams user_input, user_output and user_error for a specific thread providing a private interactive session, set_system_IO/3 rebinds the shared console I/O and also captures Prolog kernel events (e.g., low-level debug messages, unexpected events) as well as messages from foreign libraries that are directly written to stdout or stderr.

This predicate is intended to capture all output in situations where standard I/O is normally lost, such as when Prolog is running as a service on Windows.

4.17.3 Edinburgh-style I/O

The package for implicit input and output destinations is (almost) compatible with Edinburgh DEC-10 and C-Prolog. The reading and writing predicates refer to, resp., the *current* input and output streams. 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 stream's current value can be obtained using telling/1 for output and seeing/1 for input.

Source and destination are either a file, user, or a term 'pipe(*Command*)'. The reserved stream name user refers to the terminal.[59] 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.[60] Note that the pipe/1 construct is not part of Prolog's standard I/O repertoire.

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

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

The effect of tell/1 is not undone on backtracking, and since the stream handle is not specified explicitly in further I/O operations when using Edinburgh-style I/O, you may write to unintended streams more easily than when using ISO compliant I/O. For example, the following query writes both "a" and "b" into the file 'out' :

```
?- (tell(out), write(a), false ; write(b)), told.
```

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

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

Compatibility notes

Unlike Edinburgh Prolog systems, `telling/1` and `seeing/1` do not return the filename of the current input/output but rather the stream identifier, to ensure the design pattern below works under all circumstances:[61]

```
...,
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 exists, created using `tell/1` or `see/1`, output (input) is switched to the open stream. Otherwise a file with the specified name is opened.

The behaviour is compatible with Edinburgh Prolog. This is not without problems. Changing directory, non-file streams, and 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 4.17.2.

see(+*SrcDest*)
Open *SrcDest* for reading and make it the current input (see `set_input/1`). If *SrcDest* is a stream handle, just make this stream the current input. See the introduction of section 4.17.3 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 make this stream the current output. See the introduction of section 4.17.3 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 4.17.3 for details.

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 4.17.3 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`.

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

4.17.4 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*) *[ISO]*
: Set the current input stream to become *Stream*. Thus, `open(file, read, Stream), set_input(Stream)` is equivalent to `see(file)`.

set_output(+*Stream*) *[ISO]*
: Set the current output stream to become *Stream*. See also `with_output_to/2`.

current_input(-*Stream*) *[ISO]*
: Get the current input stream. Useful for getting access to the status predicates associated with streams.

current_output(-*Stream*) *[ISO]*
: Get the current output stream.

4.17.5 Adding IRI schemas

The file handling predicates may be *hooked* to deal with *IRIs*. An IRI starts with ⟨*scheme*⟩`://`, where ⟨*scheme*⟩ is a non-empty sequence of lowercase ASCII letters. After detecting the scheme the file manipulation predicates call a hook that is registered using `register_iri_scheme/3`.

Hooking the file operations using extensible IRI schemas allows us to place any resource that is accessed through Prolog I/O predicates on arbitrary devices such as web servers or the ZIP archive used to store program resources (see section 13.2). This is typically combined with `file_search_path/2` declarations to switch between accessing a set of resources from local files, from the program resource database, from a web-server, etc.

register_iri_scheme(+*Scheme, :Hook, +Options*)
: Register *Hook* to be called by all file handling predicates if a name that starts with *Scheme*:// is encountered. The *Hook* is called by `call/4` using the *operation*, the *IRI* and a term that receives the *result* of the operation. The following operations are defined:

    open(*Mode,Options*)
    : Called by `open/3`,4. The result argument must be unified with a stream.

    access(*Mode*)
    : Called by `access_file/2`, `exists_file/1` (*Mode* is `file`) and `exists_directory/1` (*Mode* is `directory`). The result argument must be unified with a boolean.

    time
    : Called by `time_file/2`. The result must be unified with a time stamp.

    size
    : Called by `size_file/2`. The result must be unified with an integer representing the size in bytes.

4.17.6 Write onto atoms, code-lists, etc.

with_output_to(+*Output*, :*Goal*)
Run *Goal* as `once/1`, while characters written to the current output are 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 multithreaded 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]
```

Output takes one of the shapes below. Except for the first, the system creates a temporary stream using the `wchar_t` internal encoding that points at a memory buffer. The encoding cannot be changed and an attempt to call `set_stream/2` using `encoding`(*Encoding*) results in a `permission_error` exception.

A Stream handle or alias
Temporarily 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 5.2.

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 from the emitted characters, similar to `atom_chars/2`.

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

4.17.7 Fast binary term I/O

The predicates in this section provide fast binary I/O of arbitrary Prolog terms, including cyclic terms and terms holding attributed variables. Library `fastrw` is a SICSTus/Ciao compatible library that extends the core primitives described below.

The binary representation the same as used by `PL_record_external()`. The use of these primitives instead of using `write_canonical/2` has advantages and disadvantages. Below are the main considerations:

- Using `write_canonical/2` allows or exchange of terms with other Prolog systems. The format is stable and, as it is text based, it can be inspected and corrected.
- Using the binary format improves the performance roughly 3 times.
- The size of both representations is comparable.
- The binary format can deal with cycles, sharing and attributes. Special precautions are needed to transfer such terms using `write_canonical/2`. See `term_factorized/3` and `copy_term/3`.
- In the current version, reading the binary format has only incomplete consistency checks. This implies a user must be able to **trust the source** as crafted messages may compromise the reading Prolog system.

fast_term_serialized(*?Term, ?String*)
(De-)serialize *Term* to/from *String*.

fast_write(*+Output, +Term*)
Write *Term* using the fast serialization format to the *Output* stream. *Output must* be a binary stream.

fast_read(*+Input, -Term*)
Read *Term* using the fast serialization format from the *Input* stream. *Input must* be a binary stream.[62]

4.18 Status of streams

wait_for_input(*+ListOfStreams, -ReadyList, +TimeOut*) *[det]*
Wait for input on one of the streams in *ListOfStreams* and return a list of streams on which input is available in *ReadyList*. Each element of *ListOfStreams* is either a stream or an integer. Integers are consider waitable OS handles. This can be used to (also) wait for handles that are not associated with Prolog streams such as UDP sockets. See `tcp_setopt/2`.

This predicate 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. If *Timeout* is 0 or 0.0 this predicate returns without waiting.[63]

This predicate can be used to implement timeout while reading and to handle input from multiple sources and is typically used to wait for multiple (network) sockets. On Unix systems it may

[62]BUG: The predicate `fast_read/2` may crash on arbitrary input.
[63]Prior to 7.3.23, the integer value '0' was the same as `infinite`.

be used on any stream that is associated with a system file descriptor. On Windows it can only be used on sockets. If *ListOfStreams* contains a stream that is not associated with a supported device, a `domain_error`(*waitable_stream, Stream*) is raised.

The example below waits for input from the user and an explicitly opened secondary terminal stream. On return, *Inputs* may hold `user_input` or *P4* or both.

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

When available, the implementation is based on the poll() system call. The poll() puts no additional restriction on the number of open files the process may have. It does limit the time to $2^{31} - 1$ milliseconds (a bit less than 25 days). Specifying a too large timeout raises a `representation_error`(*timeout*) exception. If poll() is not supported by the OS, select() is used. The select() call can only handle file descriptors up to `FD_SETSIZE`. If the set contains a descriptor that exceeds this limit a `representation_error`(*'FD_SETSIZE'*) is raised.

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 within the timeout period. Suppose two processes communicate by exchanging Prolog terms. The following code makes the server immune for clients that write an incomplete term:

```
    ...,
    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.

4.19 Primitive character I/O

See section 4.2 for an overview of supported character representations.

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

nl(+*Stream*) *[ISO]*
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. Deprecated. New code should use `put_char/1` or `put_code/1`.

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

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

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

put_char(+*Char*) *[ISO]*
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 the output stream cannot represent *Char*.

put_char(+*Stream, +Char*) *[ISO]*
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*) *[ISO]*
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 the output stream cannot represent *Code*.

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

tab(+*Amount*)
Write *Amount* spaces on the current output stream. *Amount* should be an expression that evaluates to a positive integer (see section 4.27).

tab(+*Stream, +Amount*)
Write *Amount* spaces to *Stream*.

flush_output *[ISO]*
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*) *[ISO]*
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*) *[ISO]*
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*) *[ISO]*
Read the next byte from *Stream* and unify *Byte* with an integer between 0 and 255.

get_code(-*Code*) *[ISO]*
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*) *[ISO]*
Read the next character code from *Stream*.

get_char(-*Char*) *[ISO]*
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*) *[ISO]*
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*) *[deprecated]*
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*) *[deprecated]*
Edinburgh version of the ISO `get_code/2` predicate. See also `get0/1`.

get(-*Char*) *[deprecated]*
Read the current input stream and unify the next non-blank character with *Char*. *Char* is unified with -1 on end of file. The predicate `get/1` operates on character *codes*. See also `get0/1`.

get(+*Stream, -Char*) *[deprecated]*
Read the next non-blank character from *Stream*. See also `get/1`, `get0/1` and `get0/2`.

peek_byte(-*Byte*) *[ISO]*
peek_byte(+*Stream, -Byte*) *[ISO]*
peek_code(-*Code*) *[ISO]*
peek_code(+*Stream, -Code*) *[ISO]*
peek_char(-*Char*) *[ISO]*
peek_char(+*Stream, -Char*) *[ISO]*
Read the next byte/code/char from the input without removing it. These predicates do not

modify the stream's position or end-of-file status. These predicates require a buffered stream (see `set_stream/2`) and raise a permission error if the stream is unbuffered or the buffer is too small to hold the longest multi-byte sequence that might need to be buffered.

peek_string(*+Stream, +Len, -String*)
Read the next *Len* characters (if the stream is a text stream) or bytes (if the stream is binary) from Stream without removing the data. If *Len* is larger that the stream buffer size, the buffer size is increased to *Len*. *String* can be shorter than *Len* if the stream contains less data. This predicate is intended to guess the content type of data read from non-repositionable streams.

skip(*+Code*)
Read the input until *Code* 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 `--no-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. See also `with_tty_raw/1`.

with_tty_raw(*:Goal*)
Run goal with the user input and output streams set in *raw mode*, which implies the terminal makes the input available immediately instead of line-by-line and input that is read is not echoed. As a consequence, line editing does not work. See also `get_single_char/1`.

at_end_of_stream *[ISO]*
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*) *[ISO]*
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 streams (unbuffered input streams are rarely used; see `open/4`).

set_end_of_stream(*+Stream*)
Set the size of the file opened as *Stream* to the current file position. This is typically used in combination with the open-mode `update`.

copy_stream_data(*+StreamIn, +StreamOut, +Len*)
Copy *Len* codes from *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 to take place between two text files. See section 2.19.1.

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

fill_buffer(*+Stream*) *[det]*

Fill the *Stream*'s input buffer. Subsequent calls try to read more input until the buffer is completely filled. This predicate is used together with `read_pending_codes/3` to process input with minimal buffering.

read_pending_codes(*+StreamIn, -Codes, ?Tail*)

Read input pending in the input buffer of *StreamIn* and return it in the difference list *Codes-Tail*. That is, the available characters codes are used to create the list *Codes* ending in the tail *Tail*. On encountering end-of-file, both *Codes* and *Tail* are unified with the empty list (`[]`).

This predicate is intended for efficient unbuffered copying and filtering of input coming from network connections or devices. It also enables the library `pure_input`, which processes input from files and streams using a DCG.

The following code fragment realises efficient non-blocking copying 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,
            fill_buffer(In),
            read_pending_codes(In, Chars, Tail),
            \+ \+ ( Tail = [],
                    format(Out, '~s', [Chars]),
                    flush_output(Out)
                  ),
            (   Tail == []
            ->  !
            ;   fail
            ).
```

read_pending_chars(*+StreamIn, -Chars, ?Tail*)

As `read_pending_codes/3`, but returns a difference list of one-character atoms.

4.20 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 data structures such as atoms or code-lists is supported by `with_output_to/2` and `format/3`.

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*) *[ISO]*

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 8.1) 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`).

back_quotes(*Atom*)
Fulfills the same role as the `back_quotes` prolog flag. Notably, the value `string` causes string objects to be printed between back quotes and `symbol_char` causes the backquote to be printed unquoted. In all other cases the backquote is printed as a quoted atom.

brace_terms(*Bool*)
If `true` (default), write `{}(X)` as `{X}`. See also `dotlists` and `ignore_ops`.

blobs(*Atom*)
Define how non-text blobs are handled. By default, this is left to the write handler specified with the blob type. Using `portray`, `portray/1` is called for each blob encountered. See section 12.4.8.

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).

character_escapes_unicode(*Bool*)
If `true` and `character_escapes`(*true*) and `quoted`(*true*) are active escaped characters are written using `\uXXXX` or `\UXXXXXXXX` syntax. The default depends on the Prolog flag `character_escapes_unicode`

cycles(*Bool*)
If `true` (default), cyclic terms are written as `@`(*Template, Substitutions*), where *Substitutions* is a list *Var* = *Value*. If `cycles` is `false`, `max_depth` is not given, and *Term* is cyclic, `write_term/2` raises a `domain_error`.[64] See also the `cycles` option in `read_term/2`.

dotlists(*Bool*)
If `true` (default `false`), write lists using the dotted term notation rather than the list notation.[65] Note that as of version 7, the list constructor is `'[|]'`. Using `dotlists`(*true*), `write_term/2` writes a list using '.' as constructor. This is intended for communication with programs such as other Prolog systems, that rely on this notation. See also the option `no_lists`(*true*) to use the actual SWI-Prolog list functor.

fullstop(*Bool*)
If `true` (default `false`), add a fullstop token to the output. The dot is preceded by a space if needed and followed by a space (default) or newline if the `nl`(*true*) option is also given.[66]

ignore_ops(*Bool*)
If `true`, the generic term representation (⟨*functor*⟩(⟨*args*⟩ ...)) will be used for all terms.

[64] The cycles option and the cyclic term representation using the @-term are copied from SICStus Prolog. However, the default in SICStus is set to `false` and SICStus writes an infinite term if not protected by, e.g., the `depth_limit` option.
[65] Copied from ECLiPSe.
[66] Compatible with ECLiPSe

Otherwise (default), operators will be used where appropriate.[67].

max_depth(*Integer*)
If the term is nested deeper than *Integer*, print the remainder as ellipses (...). A 0 (zero) value (default) imposes no depth limit. This option also delimits the number of printed items in 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|...])
true.
```

Used by the top level and debugger to limit screen output. See also the Prolog flags `answer_write_options` and `debugger_write_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. If *Module* does not exist it is *not* created and the `user` module is used. See also `op/3` and `read_term/2`, providing the same option.

nl(*Bool*)
Add a newline to the output. See also the `fullstop` option.

no_lists(*Bool*)
Do not use list notation. This is similar to `dotlists`(*true*), but uses the SWI-Prolog list functor, which is by default `'[|]'` instead of the ISO Prolog `'.'`. Used by `display/1`.

numbervars(*Bool*)
If `true`, terms of the format `$VAR(N)`, where *N* is an integer that fits in 64-bit,[68] will be written as a variable name. For *N* in 0..25 it emits A..Z. For higher numbers it emits An..Zn, where *n* is *N*//26. For negative numbers it emits S_*N*, which is used for representing shared sub-terms and cyclic terms.

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` and the option `variable_names`.

partial(*Bool*)
If `true` (default `false`), do not reset the logic that inserts extra spaces that separate tokens where needed. This is intended to solve the problems with the code below. Calling `write_value(.)` writes `..`, which cannot be read. By adding `partial`(*true*) to the option list, it correctly emits `. .`. Similar problems appear when emitting operators using multiple calls to `write_term/3`.

```
write_value(Value) :-
        write_term(Value, [partial(true)]),
        write('.'), nl.
```

[67]In traditional systems this flag also stops the syntactic sugar notation for lists and brace terms. In SWI-Prolog, these are controlled by the separate options `dotlists` and `brace_terms`

[68]Larger integers are ignored. As no term that fits into memory can have that many variables, this is not a restriction.

portray(*Bool*)
Same as portrayed(*Bool*). Deprecated.

portray_goal(*:Goal*)
Implies portray(*true*), but calls *Goal* rather than the predefined hook portray/1. *Goal* is called through call/3, where the first argument is *Goal*, the second is the term to be printed and the 3rd argument is the current write option list. The write option list is copied from the write_term call, but the list is guaranteed to hold an option priority that reflects the current priority.

portrayed(*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.

priority(*Integer*)
An integer between 0 and 1200 representing the 'context priority'. Default is 1200. Can be used to write partial terms appearing as the argument to an operator. For example:

```
format('~w = ', [VarName]),
write_term(Value, [quoted(true), priority(699)])
```

quoted(*Bool*)
If true, atoms and strings that need quotes will be quoted. The default is false. If character_escapes is true (default) characters in the quoted atom or string are escaped using backslash (\) sequences. To the minimum, the quote itself, newlines and backslash characters are escaped to make the output valid for read/1. All unassigned unicode characters and characters in the Unicode *separator* (Z*) and *control* (C*) classes except for the ASCII space (\u0020) are escaped. For those characters for which an ISO Prolog single character escape, e.g., \t is defined, this is used. Otherwise the output depends on the option character_escapes_unicode. If this flag applies(default) the widely accepted \uXXXX or \UXXXXXXXX is used. Otherwise the ISO Prolog \x<hex>\ syntax is used.

quote_non_ascii(*Bool*)
Quote an atom that contains non-ASCII, i.e., larger than 127 code points. The Prolog standard only describes non-quoted atom syntax containing ASCII characters. While SWI-Prolog extends this to Unicode (see section 2.16.1), transferring atoms holding non-ASCII text to other Prolog implementations may cause problems. This flag is used by write_canonical/1.

spacing(*+Spacing*)
Determines whether and where extra white space is added to enhance readability. The default is standard, adding only space where needed for proper tokenization by read_term/3. Currently, the only other value is next_argument, adding a space after a comma used to separate arguments in a term or list.

variable_names(*+List*)
Assign names to variables in *Term*. *List* is a list of terms *Name* = *Var*, where *Name* is an atom that represents a valid Prolog variable name. Terms where *Var* is bound or is a variable that does not appear in *Term* are ignored. Raises an error if *List* is not a list,

one of the members is not a term *Name* = *Var*, *Name* is not an atom or *Name* does not represent a valid Prolog variable name.

The implementation binds the variables from *List* to a term `'$VAR'`(*Name*). Like `write_canonical/1`, terms that where already bound to `'$VAR'`(*X*) before `write_term/2` are printed normally, unless the option `numbervars`(*true*) is also provided. If the option `numbervars`(*true*) is used, the user is responsible for avoiding collisions between assigned names and numbered names. See also the `variable_names` option of `read_term/2`.

Possible variable attributes (see section 8.1) are ignored. In most cases one should use `copy_term/3` to obtain a copy that is free of attributed variables and handle the associated constraints as appropriate for the use-case.

write_term(*+Stream, +Term, +Options*) *[ISO]*

As `write_term/2`, but output is sent to *Stream* rather than the current output.

write_length(*+Term, -Length, +Options*) *[semidet]*

True when *Length* is the number of characters emitted for `write_term`(*Term, Options*). In addition to valid options for `write_term/2`, it processes the option:

max_length(*+MaxLength*)

If provided, fail if *Length* would be larger than *MaxLength*. The implementation ensures that the runtime is limited when computing the length of a huge term with a bounded maximum.

write_canonical(*+Term*) *[ISO]*

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`, `quote_non_ascii`, `brace_terms`(*false*) 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 _⟨*NNN*⟩ identity of the variables.

write_canonical(*+Stream, +Term*) *[ISO]*

Write *Term* in canonical form on *Stream*.

write(*+Term*) *[ISO]*

Write *Term* to the current output, using brackets and operators where appropriate.

write(*+Stream, +Term*) *[ISO]*

Write *Term* to *Stream*.

writeq(*+Term*) *[ISO]*

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*) *[ISO]*

Write *Term* to *Stream*, inserting quotes.

writeln(+*Term*)

Equivalent to `write(Term), nl.`. The output stream is locked, which implies no output from other threads can appear between the term and newline.

writeln(+*Stream*, +*Term*)

Equivalent to `write(Stream, Term), nl(Stream).`. The output stream is locked, which implies no output from other threads can appear between the term and newline.

print(+*Term*)

Print a term for debugging purposes. The predicate `print/1` acts as if defined as below.

```
print(Term) :-
    current_prolog_flag(print_write_options, Options), !,
    write_term(Term, Options).
print(Term) :-
    write_term(Term, [ portray(true),
                       numbervars(true),
                       quoted(true)
                     ]).
```

The `print/1` predicate is used primarily through the `~p` escape sequence of `format/2`, which is commonly used in the recipes used by `print_message/2` to emit messages.

The classical definition of this predicate is equivalent to the ISO predicate `write_term/2` using the options `portray`(*true*) and `numbervars`(*true*). The `portray`(*true*) option allows the user to implement application-specific printing of terms printed during debugging to facilitate easy understanding of the output. See also `portray/1` and `portray_text`. SWI-Prolog adds `quoted`(*true*) to (1) facilitate the copying/pasting of terms that are not affected by `portray/1` and to (2) allow numbers, atoms and strings to be more easily distinguished, e.g., `42`, `'42'` and `"42"`.

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/1` succeeds `print/1` assumes the term has been written.

read(-*Term*) *[ISO]*

Read the next **Prolog term** from the current input stream and unify it with *Term*. On reaching end-of-file *Term* is unified with the atom `end_of_file`. This is the same as `read_term/2` using an empty option list.

[NOTE] You might have found this while looking for a predicate to read input from a file or the user. Quite likely this is not what you need in this case. This predicate is for reading a **Prolog**

term which may span multiple lines and must end in a *full stop* (dot character followed by a layout character). The predicates for reading and writing Prolog terms are particularly useful for storing Prolog data in a file or transferring them over a network communication channel (socket) to another Prolog process. The libraries provide a wealth of predicates to read data in other formats. See e.g., `readutil`, `pure_input` or libraries from the extension packages to read XML, JSON, YAML, etc.

read(*+Stream, -Term*) *[ISO]*
Read the next **Prolog term** from *Stream*. See `read/1` and `read_term/2` for details.

read_clause(*+Stream, -Term, +Options*)
Equivalent to `read_term/3`, but sets options according to the current compilation context and optionally processes comments. Defined options:

syntax_errors(*+Atom*)
See `read_term/3`, but the default is `dec10` (report and restart).

term_position(*-TermPos*)
Same as for `read_term/3`.

subterm_positions(*-TermPos*)
Same as for `read_term/3`.

variable_names(*-Bindings*)
Same as for `read_term/3`.

process_comment(*+Boolean*)
If `true` (default), call `prolog:comment_hook`(*Comments, TermPos, Term*) if this multifile hook is defined (see `prolog:comment_hook/3`). This is used to drive PlDoc.

comments(*-Comments*)
If provided, unify *Comments* with the comments encountered while reading *Term*. This option implies `process_comment`(*false*).

The `singletons` option of `read_term/3` is initialised from the active style-checking mode. The `module` option is initialised to the current compilation module (see `prolog_load_context/2`).

read_term(*-Term, +Options*) *[ISO]*
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 with 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 5.2). The default depends on the Prolog flag `back_quotes`.

character_escapes(*Bool*)
Defines how to read \ escape sequences in quoted atoms. See the Prolog flag `character_escapes` in `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 up to the end of the term read.

cycles(*Bool*)
If `true` (default `false`), re-instantiate templates as produced by the corresponding `write_term/2` option. Note that the default is `false` to avoid misinterpretation of `@`(*Template, Substitutions*), while the default of `write_term/2` is `true` because emitting cyclic terms without using the template construct produces an infinitely large term (read: it will generate an error after producing a huge amount of output).

dotlists(*Bool*)
If `true` (default `false`), read `.(a,[])` as a list, even if lists are internally constructed a different functor (`'[|]'(Head,Tail)`). This is primarily intended to read the output from `write_canonical/1` from other Prolog systems. See section 5.1.

double_quotes(*Atom*)
Defines how to read "..." strings. See the Prolog flag `double_quotes`. (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. If the options is provided but the target module does not exist, module `user` is used because new modules by default inherit from `user`

quasi_quotations(*-List*)
If present, unify *List* with the quasi quotations (see section A.39) instead of evaluating quasi quotations. Each quasi quotation is a term `quasi_quotation`(*+Syntax, +Quotation, +VarDict, -Result*), where *Syntax* is the term in `{|Syntax||..|}`, *Quotation* is a list of character codes that represent the quotation, *VarDict* is a list of *Name*=*Variable* and *Result* is a variable that shares with the place where the quotation must be inserted. This option is intended to support tools that manipulate Prolog source text.

singletons(*Vars*)
As `variable_names`, but only reports the variables occurring only once in the *Term* read (ISO). If *Vars* is the constant `warning`, singleton variables are reported using `print_message/2`. The variables appear in the order they have been read. The latter option provides backward compatibility and is used to read terms from source files. Not all singleton variables are reported as a warning. See section 2.16.1 for the rules that apply for warning about a singleton variable.[69]

syntax_errors(*Atom*)
If `error` (default), throw an 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

[69] As of version 7.7.17, *all* variables starting with an underscore except for the truly anonymous variable are returned in *Vars*. Older versions only reported those that would have been reported if `warning` is used.

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 are 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.

dict_position(*From, To, TagFrom, TagTo, KeyValuePosList*)
Used for a dict (see section 5.4). The position of the key-value pairs is described by *KeyValuePosList*, which is a list of `key_value_position/7` terms. The `key_value_position/7` terms appear in the order of the input. Because maps to not preserve ordering, the key is provided in the position description.

key_value_position(*From, To, SepFrom, SepTo, Key, KeyPos, ValuePos*)
Used for key-value pairs in a map (see section 5.4). It is similar to the `term_position/5` that would be created, except that the key and value positions do not need an intermediate list and the key is provided in *Key* to enable synchronisation of the file position data with the data structure.

parentheses_term_position(*From, To, ContentPos*)
Used for terms between parentheses. This is an extension compared to the original Quintus specification that was considered necessary for secure refactoring of terms.

quasi_quotation_position(*From, To, SyntaxTerm, SyntaxPos, ContentPos*)
Used for quasi quotations. Given the input `{|Syntax||Content|}`, *SyntaxTerm* is the parsed term representation from *Syntax*, e.g., `{|string(X)||Hello {{X}}|}` produces *Syntax* `string`(*X*) and *SyntaxPos* describes the layout of this term. *ContentPos* is always a term *From*-*To* describing the character range of *Content*.[70]

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

[70]The layout of the term produced by the quasi quotation parser is not available. Future versions may provide an interface that allows contributing a layout term.

var_prefix(*Bool*)
If `true`, demand variables to start with an underscore. See section 2.16.1.

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). The variables appear in the order they have been read.

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

read_term_from_atom(*+Atom, -Term, +Options*)
Use `read_term/3` to read the next term from *Atom*. *Atom* is either an atom or a string object (see section 5.2). It is not required for *Atom* to end with a full-stop. This predicate supersedes `atom_to_term/3`.

read_term_with_history(*-Term, +Options*)
Read a term while providing history substitutions. `read_term_with_history/2` is used by the top level to read the user's actions. In addition to the options recognised by `read_term/2`, the following options are recognised:

prompt(*+Prompt*)
Define the prompt to use. The default is `~! ?-`. A sequence `~!` is replaced by the current history event number.

show(*+Command*)
Using *Command* lists the saved history events. Default is `!history`.

help(*+Command*)
Using *Command* shows help on the history system. Default is `!help`.

no_save(*+Commands*)
Do not save the command into the history if it appears in the list *Commands*.

module(*+Module*)
Defines the module from which to extract module-specific syntax such as operators and handling of the various quotes. Default is the *typein* module which is set using `module/1` and is initially set to `user`.

input(*+Stream*)
Stream from which to read *Term*. Default is `user_input`.

Most applications will use the `read_term/2` option `variable_names` to get access to the names of the variables in *Term*. SWI-Prolog calls `read_term_with_history/2` as follows:

```
read_term_with_history(
    Goal,
    [ show(h),
      help('!h'),
```

```
          no_save([trace, end_of_file]),
          prompt('~! ?-'),
          variable_names(Bindings)
        ]).
```

prompt(*-Old, +New*)

Set prompt associated with reading from the `user_input` stream. *Old* is first unified with the current prompt. On success the prompt will be set to *New* (an atom). A prompt is printed if data is read from `user_input`, the cursor is at the left margin and the `user_input` is considered to be connected to a terminal. See the `tty`(*Bool*) property of `stream_property/2` and `set_stream/2`.

The default prompt is `'|: '`. Note that the toplevel loop (see `prolog/0`) sets the prompt for the first prompt (see `prompt1/1`) to `'?- '`, possibly decorated by the history event number, *break level* and debug mode. If the first line does not complete the term, subsequent lines are prompted for using the prompt as defined by `prompt/2`.

prompt1(*+Prompt*)

Sets the prompt for the next line to be read. Continuation lines will be read using the prompt defined by `prompt/2`.

4.21 Analysing and Constructing Terms

functor(*?Term, ?Name, ?Arity*) *[ISO]*

True when *Term* is a term with functor *Name/Arity*. If *Term* is a variable it is unified with a new term whose arguments are all different variables (such a term is called a skeleton). If *Term* is atomic, *Arity* will be unified with the integer 0, and *Name* will be unified with *Term*. Raises `instantiation_error` if *Term* is unbound and *Name/Arity* is insufficiently instantiated.

SWI-Prolog also supports terms with arity 0, as in `a()` (see section 5). Such terms must be processed using `functor/4` or `compound_name_arity/3`. The predicate `functor/3` and `=../2` raise a `domain_error` when faced with these terms. Without this precaution a *round trip* of a term with arity 0 over `functor/3` would create an atom.

functor(*?Term, ?Name, ?Arity, ?Type*)

As `functor/3`, but designed to work with zero-arity terms (e.g., `a()`, see section 5). *Type* is one of `atom`, `compound`, `callable` or `atomic`. *Type must* be instantiated if *Name* is an atom and *Arity* is 0 (zero). In other cases *Type* may be a variable. This predicate is true if *Term* (either initially or after haveing been created from *Name* and *Type*) and *Type* are related as below

- If *Term* is compound (including zero-arity compounds), *Type* must be `compound` or `callable`. If *Type* is unbound is is unified with `compound`.
- If *Term* is an atom, *Type* must be `atom` or `callable`. If *Type* is unbound is is unified with `atom`.
- Else *Type* is unified with `atomic`.

This predicate provides a safe *round trip* for zero-arity compounds and atoms. It can also be used as a variant of `functor/3` that only processes compound or callable terms. See also `compound/1`, `callable/1` and `compound_name_arity/3`.

arg(*?Arg, +Term, ?Value*) *[ISO]*

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.[71] The predicate `arg/3` fails silently if $Arg = 0$ or $Arg > arity$ and raises the exception `domain_error(not_less_than_zero, Arg)` if $Arg < 0$.

?Term **=..** *?List* *[ISO]*

List is a list whose head is the functor of *Term* and the remaining arguments are the arguments of the term. Either side of the predicate may be a variable, but not both. This predicate is called 'Univ'.

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

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

SWI-Prolog also supports terms with arity 0, as in `a()` (see section 5). Such terms must be processed using `compound_name_arguments/3`. This predicate raises a domain error as shown below. See also `functor/3`.

```
?- a() =.. L.
ERROR: Domain error: `compound_non_zero_arity' expected, found `a()'
```

compound_name_arity(*?Compound, ?Name, ?Arity*)

Version of `functor/3` that only works for compound terms and can examine and create compound terms with zero arguments (e.g, `name()`). See also `compound_name_arguments/3`. See also `functor/4`.

compound_name_arguments(*?Compound, ?Name, ?Arguments*)

Rationalized version of `=../2` that can compose and decompose compound terms with zero arguments. See also `compound_name_arity/3`.

numbervars(*+Term, +Start, -End*)

Unify the free variables in *Term* with a term `$VAR`(*N*), where *N* is the number of the variable. Counting starts at *Start*. *End* is unified with the number that should be given to the next variable.[72] The example below illustrates this. Note that the toplevel prints `'$VAR'(0)` as *A* due to the `numbervars`(*true*) option used to print answers.

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

[72] BUG: Only *tagged integers* are supported (see the Prolog flag `max_tagged_integer`). This suffices to count all variables that can appear in the largest term that can be represented, but does not support arbitrary large integer values for *Start*. On overflow, a `representation_error`(*tagged_integer*) exception is raised.

```
?- Term = f(X,Y,X),
   numbervars(Term, 0, End),
   write_canonical(Term), nl.
f('$VAR'(0),'$VAR'(1),'$VAR'(0))
Term = f(A, B, A),
X = A,
Y = B,
End = 2.
```

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

numbervars(+*Term, +Start, -End, +Options*)
As `numbervars/3`, 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 treat it as a normal variable and assign the next `'$VAR'`(N) term to it, or (default) `error` which raises a `type_error` exception.[73]

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`.[74]

var_number(@*Term, -VarNumber*)
True if *Term* is numbered by `numbervars/3` and *VarNumber* is the number given to this variable. This predicate avoids the need for unification with `'$VAR'(X)` and opens the path for replacing this valid Prolog term by an internal representation that has no textual equivalent.

term_variables(+*Term, -List*) *[ISO]*
Unify *List* with a list of variables, each sharing with a unique variable of *Term*.[75] The variables in *List* are ordered in order of appearance traversing *Term* depth-first and left-to-right. See also `term_variables/3` and `nonground/2`. For example:

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

[73]This behaviour was decided after a long discussion between David Reitter, Richard O'Keefe, Bart Demoen and Tom Schrijvers.
[74]BUG: Currently this option is ignored for cyclic terms.
[75]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`.

nonground(+*Term*, -*Var*) *[semidet]*

True when *Var* is a variable in *Term*. Fails if *Term* is *ground* (see `ground/1`). This predicate is intended for coroutining to trigger a wakeup if *Term* becomes ground, e.g., using `when/2`. The current implementation always returns the first variable in depth-first left-right search. Ideally it should return a random member of the set of variables (see `term_variables/2`) to realise logarithmic complexity for the ground trigger. Compatible with ECLiPSe and hProlog.

term_variables(+*Term*, -*List*, ?*Tail*)

Difference list version of `term_variables/2`. That is, *Tail* is the tail of the variable list *List*.

term_singletons(+*Term*, -*List*)

Unify *List* with a list of variables, each sharing with a variable that appears only once in *Term*.[76] Note that, if a variable appears in a shared subterm, it is *not* considered singleton. Thus, *A* is *not* a singleton in the example below. See also the `singleton` option of `numbervars/4`.

```
?- S = a(A), term_singletons(t(S,S), L).
L = [].
```

is_most_general_term(@*Term*)

True if *Term* is a callable term where all arguments are non-sharing variables or *Term* is a list whose members are all non-sharing variables. This predicate is used to reason about call subsumption for tabling and is compatible with XSB. See also `subsumes_term/2`. Examples:

1	`is_most_general_term(1)`	false
2	`is_most_general_term(p)`	true
3	`is_most_general_term(p(_))`	true
4	`is_most_general_term(p(_,a))`	false
5	`is_most_general_term(p(X,X))`	false
6	`is_most_general_term([])`	true
7	`is_most_general_term([_|_])`	false
8	`is_most_general_term([_,_])`	true
9	`is_most_general_term([X,X])`	false

copy_term(+*In*, -*Out*) *[ISO]*

Create a version of *In* with renamed (fresh) variables and unify it to *Out*. Attributed variables (see section 8.1) have their attributes 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.

[76]BUG: In the current implementation *Term* must be acyclic. If not, a `representation_error` is raised.

copy_term(*+VarsIn, +In, -VarsOut, -Out*)
Similar to `copy_term/2`, but only rename the variables in *VarsIn* that appear in *In*.[77] Variables in *In* that do not appear in *VarsIn* are *shared* between *In* and *Out*. Sub terms that only contain such shared variables are shared as a whole between *In* and *Out*. *VarsIn* is often a list, but can be an arbitrary term. For example:

```
?- copy_term([X], q(X,Y), Vars, Term).
Vars = [_A],
Term = q(_A, Y).
```

Note that if *VarsIn* and *In* do not share any variables, *Out* is equivalent to *In* and *VarsOut* is a copy (as `copy_term/2`) of *VarsIn*. If *In* does not contain any variables not in *VarsIn* the result is the same as `copy_term(VarsIn-In, VarsOut-Out)`.

copy_term_nat(*+VarsIn, +In, -VarsOut, -Out*)
As `copy_term/4`, using the attributed variable semantics of `copy_term_nat/2`. This implies that attributed variables that appear in *VarsIn* appear as renamed plain variables in *VarsOut* and *Out*. Attributed variables in *In* that do *not* appear in *VarsIn* are shared between *In* and *Out*.

4.21.1 Non-logical operations on terms

Prolog is not able to *modify* instantiated parts of a term. Lacking that capability makes the 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 4.33, 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 noncopying of terms. A good practice to improve somewhat on this situation is to make sure that terms whose arguments are subject to `setarg/3` have one unused and unshared variable in addition to the used arguments. This variable avoids unwanted sharing in, e.g., `copy_term/2`, and causes the term to be considered as non-ground. An alternative is to use `put_attr/3` to attach information to attributed variables (see section 8.1).

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 *Value* 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 with GNU-Prolog `setarg`(*A,T,V,false*),

[77] This predicate is based on a similar predicate in s(CASP) by Joaquin Arias.

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.

```
:- meta_predicate
        succeeds_n_times(0, -).

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 that 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`.

same_term(@*T1*, @*T2*) *[semidet]*
True if *T1* and *T2* are equivalent and will remain equivalent, even if `setarg/3` is used on either of them. This means *T1* and *T2* are the same variable, equivalent atomic data or a compound term allocated at the same address.

4.22 Analysing and Constructing Atoms

These predicates convert between certain Prolog atomic values on one hand and lists of *character codes* (or, for `atom_chars/2`, *characters*) on the other. The Prolog atomic values can be atoms, *character*s (which are atoms of length 1), SWI-Prolog strings, as well as numbers (integers, floats and non-integer rationals).

The *character codes*, also known as *code values*, are integers. In SWI-Prolog, these integers are Unicode code points.[78]

To ease the pain of all text representation variations in the Prolog community, all SWI-Prolog predicates behave as *flexible as possible*. This implies the 'list-side' accepts both a character-code-list and a character-list and the 'atom-side' accepts all atomic types (atom, number and string). For example, the predicates `atom_codes/2`, `number_codes/2` and `name/2` behave the same in mode

[78]BUG: On Windows the range is limited to UCS-2, 0..65535.

(+,-), i.e., 'listwards', 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 throw an exception;
- `name/2` will generate a number if possible and an atom otherwise.

atom_codes(*?Atom, ?CodeList*) *[ISO]*

Convert between an atom and a list of *character codes* (integers denoting characters).

- If *Atom* is instantiated, it will be translated into a list of character codes, which are unified with *CodeList*.
- If *Atom* is uninstantiated and *CodeList* is a list of character codes, then *Atom* will be unified with an atom constructed from this list.

```
?- atom_codes(hello, X).
X = [104, 101, 108, 108, 111].
```

The 'listwards' call to `atom_codes/2` can also be written (functionally) using backquotes instead:

```
?- Cs = `hello`.
Cs = [104, 101, 108, 108, 111].
```

Backquoted strings can be mostly found in the body of DCG rules that process lists of character codes.

Note that this is the default interpretation for backquotes. It can be changed on a per-module basis by setting the value of the Prolog flag `back_quotes`.

atom_chars(*?Atom, ?CharList*) *[ISO]*

Similar to `atom_codes/2`, but *CharList* is a list of *character*s (atoms of length 1) rather than a list of *character codes* (integers denoting characters).

```
?- atom_chars(hello, X).
X = [h, e, l, l, o]
```

char_code(*?Atom, ?Code*) *[ISO]*

Convert between a single *character* (an atom of length 1), and its *character code* (an integer denoting the corresponding character). The predicate alternatively accepts an SWI-Prolog string of length 1 at *Atom* place.

number_chars(*?Number, ?CharList*) *[ISO]*

Similar to `atom_chars/2`, but converts between a number and its representation as a list of *characters* (atoms of length 1).

- If *CharList* is a *proper list*, i.e., not unbound or a *partial list*, *CharList* is parsed according to the Prolog syntax for numbers and the resulting number is unified with *Number*. A `syntax_error` exception is raised if *CharList* is instantiated to a ground, proper list but does not represent a valid Prolog number.
- Otherwise, if *Number* is indeed a number, *Number* is serialized and the result is unified with *CharList*.

Following the ISO standard, the Prolog syntax for number allows for *leading* white space (including newlines) and does not allow for *trailing* white space.[79]

Prolog syntax-based conversion can also be achieved using `format/3` and `read_from_chars/2`.

number_codes(*?Number, ?CodeList*) *[ISO]*

As `number_chars/2`, but converts to a list of character codes rather than characters. 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, float or non-integer rational) in one predicate, avoiding the intermediate list. Unlike the ISO standard `number_codes/2` predicates, `atom_number/2` fails silently in mode (+,-) if *Atom* does not represent a number.

name(*?Atomic, ?CodeList*)

CodeList is a list of character codes representing the same text as *Atomic*. Each of the arguments may be a variable, but not both.

- When *CodeList* describes an integer or floating point number and *Atomic* is a variable, *Atomic* will be unified with the numeric value described by *CodeList* (e.g., `name(N, "300"), 400 is N + 100` succeeds).
- If *CodeList* is not a representation of a number, *Atomic* will be unified with the atom with the name given by the character code list.
- If *Atomic* is an atom or number, the unquoted print representation of it as a character code list is unified with *CodeList*.

This predicate is part of the Edinburgh tradition. It should be considered *deprecated* although, given its long tradition, it is unlikely to be removed from the system. It still has some value for converting input to a number or an atom (depending on the syntax). New code should consider the ISO predicates `atom_codes/2`, `number_codes/2` or the SWI-Prolog predicate `atom_number/2`.

term_to_atom(*?Term, ?Atom*)

True if *Atom* describes a term that unifies with *Term*. When *Atom* is instantiated, *Atom* is parsed and the result unified with *Term*. If *Atom* has no valid syntax, a `syntax_error` exception is raised. Otherwise *Term* is "written" on *Atom* using `write_term/2` with the option `quoted(`*true*`)`. See also `format/3`, `with_output_to/2` and `term_string/2`.

[79] ISO also allows for Prolog comments in leading white space. We–and most other implementations–believe this is incorrect. We also believe it would have been better not to allow for white space, or to allow for both leading and trailing white space.

atom_to_term(*+Atom, -Term, -Bindings*) *[deprecated]*

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. New code should use `read_term_from_atom/3`.

atom_concat(*?Atom1, ?Atom2, ?Atom3*) *[ISO]*

Atom3 forms the concatenation of *Atom1* and *Atom2*. At least two of the arguments must be instantiated to atoms. This predicate also allows for the mode (-,-,+), non-deterministically splitting the 3rd argument into two parts (as `append/3` does for lists). SWI-Prolog allows for atomic arguments. Portable code must use `atomic_concat/3` if non-atom arguments are involved.

atomic_concat(*+Atomic1, +Atomic2, -Atom*)

Atom represents the text after converting *Atomic1* and *Atomic2* to text and concatenating the result:

```
?- atomic_concat(name, 42, X).
X = name42.
```

atomic_list_concat(*+List, -Atom*) *[commons]*

List is a list of strings, atoms, integers, floating point numbers or non-integer rationals. Succeeds if *Atom* can be unified with the concatenated elements of *List*. Equivalent to `atomic_list_concat`(*List, '', Atom*).

atomic_list_concat(*+List, +Separator, -Atom*) *[commons]*

Creates an atom just like `atomic_list_concat/2`, but inserts *Separator* between each pair of inputs. For example:

```
?- atomic_list_concat([gnu, gnat], ', ', A).

A = 'gnu, gnat'
```

The 'atomwards' transformation is usually called a *string join* operation in other programming languages.

The SWI-Prolog version of this predicate can also be used to split atoms by instantiating *Separator* and *Atom* as shown below. We kept this functionality to simplify porting old SWI-Prolog code where this predicate was called `concat_atom/3`. When used in mode (-,+,+), *Separator* must be a non-empty atom. See also `split_string/4`.

```
?- atomic_list_concat(L, -, 'gnu-gnat').

L = [gnu, gnat]
```

atom_length(*+Atom, -Length*) *[ISO]*

True if *Atom* is an atom of *Length* characters. The SWI-Prolog version accepts all atomic types, as well as code-lists and character-lists. New code should avoid this feature and use `write_length/3` to get the number of characters that would be written if the argument was handed to `write_term/3`.

atom_prefix(*+Atom, +Prefix*) *[deprecated]*

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

sub_atom(*+Atom, ?Before, ?Length, ?After, ?SubAtom*) *[ISO]*

ISO predicate for breaking atoms. It maintains the following relation: *SubAtom* is a sub-atom of *Atom* that starts at (0-based index) *Before*, has *Length* characters, and *Atom* contains *After* characters after the match. The implementation minimises non-determinism and creation of atoms. This is a flexible predicate that can do search, prefix- and suffix-matching, etc.

Pick out a sub-atom of length 3 starting a 0-based index 2:

```
?- sub_atom(aaxyzbbb, 2, 3, After, SubAtom).
After = 3,
SubAtom = xyz.
```

On backtracking, matches are delivered in order left-to-right (i.e. *Before* increases monotonically):

```
?- sub_atom('xATGATGAxATGAxATGAx', Before, Length, After, 'ATGA').
Before = 1, Length = 4, After = 14 ;
Before = Length, Length = 4, After = 11 ;
Before = 9, Length = 4, After = 6 ;
Before = 14, Length = 4, After = 1 ;
false.
```

See also `sub_string/5`, the corresponding predicate for SWI-Prolog strings.

sub_atom_icasechk(*+Haystack, ?Start, +Needle*) *[semidet]*

True when *Needle* is a sub atom of *Haystack* starting at *Start*. The match is 'half case insensitive', i.e., uppercase letters in *Needle* only match themselves, while lowercase letters in *Needle* match case insensitively. *Start* is the first 0-based offset inside *Haystack* where *Needle* matches.[80]

4.23 Localization (locale) support

SWI-Prolog provides (currently limited) support for localized applications.

- The predicates `char_type/2` and `code_type/2` query character classes depending on the locale.

[80]This predicate replaces `$apropos_match/2`, used by the help system, while extending it with locating the (first) match and performing case insensitive prefix matching. We are still not happy with the name and interface.

- The predicates `collation_key/2` and `locale_sort/2` can be used for locale dependent sorting of atoms.
- The predicate `format_time/3` can be used to format time and date representations, where some of the specifiers are locale dependent.
- The predicate `format/2` provides locale-specific formatting of numbers. This functionality is based on a more fine-grained localization model that is the subject of this section.

A locale is a (optionally named) read-only object that provides information to locale specific functions.[81] The system creates a default locale object named `default` from the system locale. This locale is used as the initial locale for the three standard streams as well as the `main` thread. Locale sensitive output predicates such as `format/3` get their locale from the stream to which they deliver their output. New streams get their locale from the thread that created the stream. Threads get their locale from the thread that created them.

locale_create(*-Locale, +Default, +Options*)
Create a new locale object. *Default* is either an existing locale or a string that denotes the name of a locale provided by the system, such as `"en_EN.UTF-8"`. The values read from the default locale can be modified using *Options*. *Options* provided are:

alias(*+Atom*)
Give the locale a name.

decimal_point(*+Atom*)
Specify the decimal point to use.

thousands_sep(*+Atom*)
Specify the string that delimits digit groups. Only effective is `grouping` is also specified.

grouping(*+List*)
Specify the grouping of digits. Groups are created from the right (least significant) digits, left of the decimal point. *List* is a list of integers, specifying the number of digits in each group, counting from the right. If the last element is `repeat`(*Count*), the remaining digits are grouped in groups of size *Count*. If the last element is a normal integer, digits further to the left are not grouped.

For example, the English locale uses

```
[ decimal_point('.'), thousands_sep(','), grouping([repeat(3)]) ]
```

Named locales exists until they are destroyed using `locale_destroy/1` and they are no longer referenced. Unnamed locales are subject to (atom) garbage collection.

locale_destroy(*+Locale*)
Destroy a locale. If the locale is named, this removes the name association from the locale, after which the locale is left to be reclaimed by garbage collection.

[81]The locale interface described in this section and its effect on `format/2` and reading integers from digit groups was discussed on the SWI-Prolog mailinglist. Most input in this discussion is from Ulrich Neumerkel and Richard O'Keefe. The predicates in this section were designed by Jan Wielemaker.

locale_property(*?Locale, ?Property*)
True when *Locale* has *Property*. Properties are the same as the *Options* described with `locale_create/3`.

set_locale(*+Locale*)
Set the default locale for the current thread, as well as the locale for the standard streams (`user_input`, `user_output`, `user_error`, `current_output` and `current_input`. This locale is used for new streams, unless overruled using the `locale`(*Locale*) option of `open/4` or `set_stream/2`.

current_locale(*-Locale*)
True when *Locale* is the locale of the calling thread.

4.24 Character properties

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

char_type(*?Char, ?Type*)
Tests or generates alternative *Type*s or *Char*s. The character types are inspired by the standard C `<ctype.h>` primitives. The types are sensititve to the active *locale*, see `setlocale/3`. Most of the *Type*s are mapped to the Unicode classification functions from `<wctype.h>`, e.g., `alnum` uses iswalnum(). The types `prolog_var_start`, `prolog_atom_start`, `prolog_identifier_continue` and `prolog_symbol` are based on the locale-independent built-in classification routines that are also used by `read/1` and friends.

Note that the mode (-,+) is only efficient if the *Type* has a parameter, e.g., `char_type`(*C, digit(8)*). If *Type* is a atomic, the whole unicode range (0..0x1ffff) is generated and tested against the character classification function.

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-bit ASCII character (0..127).

white
Char is a space or tab, i.e. white space inside a line.

cntrl
Char is an ASCII control character (0..31), ASCII DEL character (127), or non-ASCII character in the range 128..159 or 8232..8233.

digit
Char is a digit.

digit(*Weight*)
Char is a digit with value *Weight*. I.e. `char_type(X, digit(6))` yields *X* = `'6'`. Useful for parsing numbers.

xdigit(*Weight*)
Char is a hexadecimal digit with value *Weight*. I.e. `char_type(a, xdigit(X))` yields *X* = `'10'`. Useful for parsing numbers.

print
Char is printable character.

graph
Char produces a visible mark on a page when printed. Note that the space is not included!

lower
Char is a lowercase letter.

lower(*Upper*)
Char is a lowercase version of *Upper*. Only true if *Char* is lowercase and *Upper* uppercase.

to_lower(*Upper*)
Char is a lowercase 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 uppercase letter.

upper(*Lower*)
Char is an uppercase version of *Lower*. Only true if *Char* is uppercase and *Lower* lowercase.

to_upper(*Lower*)
Char is an uppercase 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 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.

prolog_var_start
Char can start a Prolog variable name.

prolog_atom_start
Char can start a unquoted Prolog atom that is not a symbol.

prolog_identifier_continue
Char can continue a Prolog variable name or atom.

prolog_symbol
Char is a Prolog symbol character. Sequences of Prolog symbol characters glue together to form an unquoted atom. Examples are `=..`, \=, etc.

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 will instantiate only with an integer code or a one-character atom, depending of the version used. See also the Prolog flag `double_quotes`, `atom_chars/2` and `atom_codes/2`.

4.24.1 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 uppercase.

4.24.2 White space normalization

normalize_space(*-Out, +In*)
Normalize white space in *In*. All leading and trailing white space is removed. All non-empty sequences for Unicode white space characters are replaced by a single space (`\u0020`) character. *Out* uses the same conventions as `with_output_to/2` and `format/3`.

4.24.3 Language-specific comparison

This section deals with predicates for language-specific string comparison operations.

collation_key(*+Atom, -Key*)
: Create a *Key* from *Atom* for locale-specific comparison. The key is defined such that if the key of atom A precedes the key of atom B in the standard order of terms, A is alphabetically smaller than B using the sort order of the current locale.

  The predicate `collation_key/2` is used by `locale_sort/2` from library(sort). Please examine the implementation of `locale_sort/2` as an example of using this call.

  The *Key* is an implementation-defined and generally unreadable string. On systems that do not support locale handling, *Key* is simply unified with *Atom*.

locale_sort(*+List, -Sorted*)
: Sort a list of atoms using the current locale. *List* is a list of atoms or string objects (see section 5.2). *Sorted* is unified with a list containing all atoms of *List*, sorted to the rules of the current locale. See also `collation_key/2` and `setlocale/3`.

4.25 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 using many operators can make it hard to understand the limits of your syntax.

In SWI-Prolog, 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`. Many modern Prolog systems have module specific operators. Unfortunately, there is no established interface for exporting and importing operators. SWI-Prolog's convention has been adopted by YAP.

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 with other Prolog that do not have module-local operators:

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

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

Although operators are module-specific and the predicates that define them (`op/3`) or rely on them such as `current_op/3`, `read/1` and `write/1` are module sensitive, they are not proper meta-predicates. If they were proper meta predicates `read/1` and `write/1` would use the module from which they are called, breaking compatibility with other Prolog systems. The following rules apply:

1. If the module is explicitly specified by qualifying the third argument (`op/3`, `current_op/3`) or specifying a `module(`*Module*`)` option (`read_term/3`, `write_term/3`), this module is used.

2. While compiling, the module into which the compiled code is loaded applies.

3. Otherwise, the *typein module* applies. This is normally `user` and may be changed using `module/1`.

In SWI-Prolog, a *quoted atom* never acts as an operator. Note that the portable way to stop an atom acting as an operator is to enclose it in parentheses like this: (myop). See also section 5.3.1.

op(*+Precedence, +Type, :Name*) *[ISO]*

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`, `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 parentheses `( ... )` has precedence 0.

The predefined operators are shown in table 4.2. Operators can be redefined, unless prohibited by one of the limitations below. Applications must be careful with (re-)defining operators because changing operators may cause (other) files to be interpreted **differently**. Often this will lead to a syntax error. In other cases, text is read silently into a different term which may lead to subtle and difficult to track errors.

- It is not allowed to redefine the comma (`','`).
- The bar (`|`) can only be (re-)defined as infix operator with priority not less than 1001.
- It is not allowed to define the empty list (`[]`) or the curly-bracket pair (`{}`) as operators.

In SWI-Prolog, operators are *local* to a module (see also section 6.9). Keeping operators in modules and using controlled import/export of operators as described with the `module/2` directive keep the issues manageable. The module `system` provides the operators from table 4.2 and these operators cannot be modified. Files that are loaded from the SWI-Prolog directories resolve operators and predicates from this `system` module rather than `user`, which makes the semantics of the library and development system modules independent of operator changes to the `user` module. See section 4.25 for details about the relation between operators and modules.

current_op(*?Precedence, ?Type, ?:Name*) *[ISO]*

True if *Name* is currently defined as an operator of type *Type* with precedence *Precedence*. See also `op/3`. Note that an *unqualified Name* does **not** resolve to the calling context but, when compiling, to the compiler's target module and otherwise to the *typein module*. See section 4.25 for details.

1200	xfx	`-->`, `:-`
1200	fx	`:-`, `?-`
1150	fx	`dynamic`, `discontiguous`, `initialization`, `meta_predicate`, `module_transparent`, `multifile`, `public`, `thread_local`, `thread_initialization`, `volatile`
1105	xfy	`|`
1100	xfy	`;`
1050	xfy	`->`, `*->`
1000	xfy	`,`
990	xfx	`:=`
900	fy	`\+`
700	xfx	`<`, `=`, `=..`, `=@=`, `\=@=`, `=:=`, `=<`, `==`, `=\=`, `>`, `>=`, `@<`, `@=<`, `@>`, `@>=`, `\=`, `\==`, `as`, `is`, `>:<`, `:<`
600	xfy	`:`
500	yfx	`+`, `-`, `/\`, `\/`, `xor`
500	fx	`?`
400	yfx	`*`, `/`, `//`, `div`, `rdiv`, `<<`, `>>`, `mod`, `rem`
200	xfx	`**`
200	xfy	`^`
200	fy	`+`, `-`, `\`
100	yfx	`.`
1	fx	`$`

Table 4.2: System operators

4.26 Character Conversion

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

char_conversion(*+CharIn, +CharOut*) *[ISO]*

Define that term input (see `read_term/3`) maps each character read as *CharIn* to the character *CharOut*. Character conversion is only executed if the Prolog flag `char_conversion` is set to `true` and not inside quoted atoms or strings. The initial table maps each character onto itself. See also `current_char_conversion/2`.

current_char_conversion(*?CharIn, ?CharOut*) *[ISO]*

Queries the current character conversion table. See `char_conversion/2` for details.

4.27 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 4.27.2.

4.27.1 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`[82] `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 $Int1 \geq 0$. 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.[83]

plus(*?Int1, ?Int2, ?Int3*)

True if $Int3 = Int1 + Int2$. At least two of the three arguments must be instantiated to integers.

divmod(*+Dividend, +Divisor, -Quotient, -Remainder*)

This predicate is a shorthand for computing both the *Quotient* and *Remainder* of two integers in a single operation. This allows for exploiting the fact that the low level implementation for computing the quotient also produces the remainder. Timing confirms that this predicate

[82] We prefer `infinite`, but some other Prolog systems already use `inf` for infinity; we accept both for the time being.

[83] 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.

is almost twice as fast as performing the steps independently. Semantically, `divmod/4` is defined as below.

```
divmod(Dividend, Divisor, Quotient, Remainder) :-
        Quotient  is Dividend div Divisor,
        Remainder is Dividend mod Divisor.
```

Note that this predicate is only available if SWI-Prolog is compiled with unbounded integer support. This is the case for all packaged versions.

nth_integer_root_and_remainder(*+N, +I, -Root, -Remainder*)
True when $Root^N + Remainder = I$. *N* and *I* must be integers.[84] *N* must be one or more. If *I* is negative and *N* is *odd*, *Root* and *Remainder* are negative, i.e., the following holds for $I < 0$:

```
%   I < 0,
%   N mod 2 =\= 0,
    nth_integer_root_and_remainder(
        N, I, Root, Remainder),
    IPos is -I,
    nth_integer_root_and_remainder(
        N, IPos, RootPos, RemainderPos),
    Root =:= -RootPos,
    Remainder =:= -RemainderPos.
```

4.27.2 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* *[ISO]*
True if expression *Expr1* evaluates to a larger number than *Expr2*.

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

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

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

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

[84] This predicate was suggested by Markus Triska. The final name and argument order is by Richard O'Keefe. The decision to include the remainder is by Jan Wielemaker. Including the remainder makes this predicate about twice as slow if *Root* is not exact.

+Expr1 **=:=** *+Expr2* *[ISO]*

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

-Number **is** *+Expr* *[ISO]*

True when *Number* is the value to which *Expr* evaluates. Typically, `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.

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 values on the global stack. Integers that do not fit in 64 bits are represented as serialised GNU MPZ structures on the global stack.

- *rational number*

  Rational numbers (Q) are quotients of two integers (N/M). Rational arithmetic is only provided if GMP is used (see above). Rational numbers satisfy the type tests `rational/1`, `number/1` and `atomic/1` and may satisfy the type test `integer/1`, i.e., integers are considered rational numbers. Rational numbers are always kept in *canonical representation*, which means M is positive and N and M have no common divisors. Rational numbers are introduced into the computation using the functions `rational/1`, `rationalize/1` or the `rdiv/2` (rational division) function. If the Prolog flag `prefer_rationals` is `true` (default), division (`//2`) and integer power (`^/2`) also produce a rational number.

- *float*

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

Arithmetic functions that require integer arguments accept, in addition to integers, rational numbers with (canonical) denominator '1'. 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 $\rightarrow$ rational number $\rightarrow$ floating point number

Rational number examples

The use of rational numbers with unbounded integers allows for exact integer or *fixed point* arithmetic under addition, subtraction, multiplication, division and exponentiation (`^/2`). Support for rational numbers depends on the Prolog flag `prefer_rationals`. If this is `true` (default), the number division function (`//2`) and exponentiation function (`^/2`) generate a rational number on integer and rational arguments and `read/1` and friends read `[-+][0-9_ ]+/[0-9_ ]+` into a rational number. See also section 2.16.1. Here are some examples.

A is 2/6	A = 1/3
A is 4/3 + 1	A = 7/3
A is 4/3 + 1.5	A = 2.83333
A is 4/3 + rationalize(1.5)	A = 17/6

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 4/3 + rational(1.5),
   format('~50f~n', [A]).
2.83333333333333333333333333333333333333333333333333

A = 17/6
```

Rational numbers or floats

SWI-Prolog uses rational number arithmetic if the Prolog flag `prefer_rationals` is `true` and if this is defined for a function on the given operants. This results in perfectly precise answers. Unfortunately rational numbers can get really large and, if a precise answer is not needed, a big waste of memory and CPU time. In such cases one should use floating point arithmetic. The Prolog flag `max_rational_size` provides a *tripwire* to detect cases where rational numbers get big and react on these events.

Floating point arithmetic can be forced by forcing a float into an argument at any point, i.e., the result of a function with at least one float is always float except for the float-to-integer rounding and truncating functions such as `round/1`, `rational/1` or `float_integer_part/1`.

Float arithmetic is typically forced by using a floating point constant as initial value or operant. Alternatively, the `float/1` function forces conversion of the argument.

IEEE 754 floating point arithmetic

The Prolog ISO standard defines that floating point arithmetic returns a valid floating point number or raises an exception. IEEE floating point arithmetic defines two modes: raising exceptions and propagating the special float values `NaN`, `Inf`, `-Inf` and `-0.0`. SWI-Prolog implements a part of the ECLiPSe proposal to support non-exception based processing of floating point numbers. There are four flags that define handling the four exceptional events in floating point arithmetic, providing the choice between `error` and returning the IEEE special value. Note that these flags *only* apply for floating point arithmetic. For example rational division by zero always raises an exception.

Flag	**Default**	**Alternative**
`float_overflow`	error	infinity
`float_zero_div`	error	infinity
`float_undefined`	error	nan
`float_underflow`	ignore	error

The Prolog flag `float_rounding` and the function `roundtoward/2` control the rounding mode for floating point arithmetic. The default rounding is `to_nearest` and the following alternatives are provided: `to_positive`, `to_negative` and `to_zero`.

float_class(*+Float, -Class*) *[det]*

Wraps C99 fpclassify() to access the class of a floating point number. Raises a type error if *Float* is not a float. Defined classes are below.

nan

Float is "Not a number". See `nan/0`. May be produced if the Prolog flag `float_undefined` is set to `nan`. Although IEEE 754 allows NaN to carry a *payload* and have a sign, SWI-Prolog has only a single NaN values. Note that two NaN *terms* compare equal in the standard order of terms (`==/2`, etc.), they compare non-equal for arithmetic (`=:=/2`, etc.).

infinite

Float is positive or negative infinity. See `inf/0`. May be produced if the Prolog flag `float_overflow` or the flag `float_zero_div` is set to `infinity`.

zero

Float is zero (0.0 or -0.0)

subnormal

Float is too small to be represented in normalized format. May **not** be produced if the Prolog flag `float_underflow` is set to `error`.

normal

Float is a normal floating point number.

float_parts(*+Float, -Mantissa, -Base, -Exponent*) *[det]*

True when *Mantissa* is the normalized fraction of *Float*, *Base* is the *radix* and *Exponent* is the exponent. This uses the C function frexp(). If *Float* is NaN or ±Inf *Mantissa* has the same value and *Exponent* is 0 (zero). In the current implementation *Base* is always 2. The following relation is always true:

$$Float =:= Mantissa \times Base^{Exponent}$$

bounded_number(*?Low, ?High, +Num*) *[det]*

True if *Low* ¡ *Num* ¡ *High*. Raises a type error if *Num* is not a number. This predicate can be used both to check and generate bounds across the various numeric types. Note that a number cannot be bounded by itself and `NaN`, `Inf`, and `-Inf` are not bounded numbers.

If *Low* and/or *High* are variables they will be unified with *tightest* values that still meet the bounds criteria. The generated bounds will be integers if *Num* is an integer; otherwise they will be floats (also see `nexttoward/2` for generating float bounds). Some examples:

```
?- bounded_number(0,10,1).
true.

?- bounded_number(0.0,1.0,1r2).
true.

?- bounded_number(L,H,1.0).
L = 0.9999999999999999,
H = 1.0000000000000002.

?- bounded_number(L,H,-1).
L = -2,
H = 0.

?- bounded_number(0,1r2,1).
false.

?- bounded_number(L,H,1.0Inf).
false.
```

Arithmetic Functions

Arithmetic functions are terms which are evaluated by the arithmetic predicates described in section 4.27.2. 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 to an integer.
RatExpr	Arbitrary expression that must evaluate to a rational number.
FloatExpr	Arbitrary expression that must evaluate to a floating point.

For systems using bounded integer arithmetic (default is unbounded, see section 4.27.2 for details), integer operations that would cause overflow automatically convert to floating point arithmetic.

SWI-Prolog provides many extensions to the set of floating point functions defined by the ISO standard. The current policy is to provide such functions on 'as-needed' basis if the function is widely supported elsewhere and notably if it is part of the C99 mathematical library. In addition, we try to maintain compatibility with YAP.

- *+Expr* *[ISO]*

Result $= -$*Expr*

+ *+Expr* *[ISO]*

Result $=$ *Expr*. Note that if + is followed by a number, the parser discards the +. I.e. `?- integer(+1)` succeeds.

+Expr1 **+** *+Expr2* *[ISO]*

Result = *Expr1* + *Expr2*

+Expr1 **-** *+Expr2* *[ISO]*

Result = *Expr1* − *Expr2*

+Expr1 ***** *+Expr2* *[ISO]*

Result = *Expr1* × *Expr2*

+Expr1 **/** *+Expr2* *[ISO]*

$Result = \frac{Expr1}{Expr2}$. If the flag `iso` is `true` or one of the arguments is a float, both arguments are converted to float and the return value is a float. Otherwise the result type depends on the Prolog flag `prefer_rationals`. If `true`, the result is always a rational number. If `false` the result is rational if at least one of the arguments is rational. Otherwise (both arguments are integer) the result is integer if the division is exact and float otherwise. See also section 4.27.2, `///2`, and `rdiv/2`.

The current default for the Prolog flag `prefer_rationals` is `false`. Future version may switch this to `true`, providing precise results when possible. The pitfall is that in general rational arithmetic is slower and can become very slow and produce huge numbers that require a lot of (global stack) memory. Code for which the exact results provided by rational numbers is not needed should force float results by making one of the operants float, for example by dividing by `10.0` rather than `10` or by using `float/1`. Note that when one of the arguments is forced to a float the division is a float operation while if the result is forced to the float the division is done using rational arithmetic.

+IntExpr1 **mod** *+IntExpr2* *[ISO]*

Modulo, defined as *Result* = *IntExpr1* - (*IntExpr1* div *IntExpr2*) × *IntExpr2*, where `div` is *floored* division.

+IntExpr1 **rem** *+IntExpr2* *[ISO]*

Remainder of integer division. Behaves as if defined by *Result* is *IntExpr1* - (*IntExpr1* // *IntExpr2*) × *IntExpr2*

+IntExpr1 **//** *+IntExpr2* *[ISO]*

Integer division, defined as *Result* is rnd_I(*Expr1*/*Expr2*). The function rnd_I is the default rounding used by the C compiler and available through the Prolog flag `integer_rounding_function`. In the C99 standard, C-rounding is defined as `towards_zero`.[85]

div(*+IntExpr1, +IntExpr2*) *[ISO]*

Integer division, defined as *Result* is (*IntExpr1* - *IntExpr1* mod *IntExpr2*) // *IntExpr2*. In other words, this is integer division that rounds towards -infinity. This function guarantees behaviour that is consistent with `mod/2`, i.e., the following holds for every pair of integers X, Y where `Y =\= 0`.

```
        Q is div(X, Y),
        M is mod(X, Y),
        X =:= Y*Q+M.
```

[85] Future versions might guarantee rounding towards zero.

+RatExpr **rdiv** *+RatExpr*
: Rational number division. This function is only available if SWI-Prolog has been compiled with rational number support. See section 4.27.2 for details.

+IntExpr1 **gcd** *+IntExpr2*
: Result is the greatest common divisor of *IntExpr1* and *IntExpr2*. The GCD is always a positive integer. If either expression evaluates to zero the GCD is the result of the other expression.

+IntExpr1 **lcm** *+IntExpr2*
: Result is the least common multiple of *IntExpr1*, *IntExpr2*.[86] If either expression evaluates to zero the LCM is zero.

abs(*+Expr*) *[ISO]*
: Evaluate *Expr* and return the absolute value of it.

sign(*+Expr*) *[ISO]*
: Evaluate to -1 if $Expr < 0$, 1 if $Expr > 0$ and 0 if $Expr = 0$. If *Expr* evaluates to a float, the return value is a float (e.g., -1.0, 0.0 or 1.0). In particular, note that sign(-0.0) evaluates to 0.0. See also `copysign/2`.

copysign(*+Expr1, +Expr2*) *[ISO]*
: Evaluate to *X*, where the absolute value of *X* equals the absolute value of *Expr1* and the sign of *X* matches the sign of *Expr2*. This function is based on copysign() from C99, which works on double precision floats and deals with handling the sign of special floating point values such as -0.0. Our implementation follows C99 if both arguments are floats. Otherwise, `copysign/2` evaluates to *Expr1* if the sign of both expressions matches or -*Expr1* if the signs do not match. Here, we use the extended notion of signs for floating point numbers, where the sign of -0.0 and other special floats is negative.

nexttoward(*+Expr1, +Expr2*)
: Evaluates to floating point number following *Expr1* in the direction of *Expr2*. This relates to `epsilon/0` in the following way:

```
?- epsilon =:= nexttoward(1,2)-1.
true.
```

roundtoward(*+Expr1, +RoundMode*)
: Evaluate *Expr1* using the floating point rounding mode *RoundMode*. This provides a local alternative to the Prolog flag `float_rounding`. This function can be nested. The supported values for *RoundMode* are the same as the flag values: `to_nearest`, `to_positive`, `to_negative` or `to_zero`.

max(*+Expr1, +Expr2*) *[ISO]*
: Evaluate to the larger of *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. If both values are numerical

[86] BUG: If the system is compiled for bounded integers only `lcm/2` produces an integer overflow if the product of the two expressions does not fit in a 64 bit signed integer. The default build with unbounded integer support has no such limit.

equal the returned max is of the type used for the comparison. For example, the max of 1 and 1.0 is 1.0 because both numbers are converted to float for the comparison. However, the special float -0.0 is smaller than 0.0 as well as the integer 0.

min(+*Expr1, +Expr2*) *[ISO]*

Evaluate to the smaller of *Expr1* and *Expr2*. See `max/2` for a description of type handling.

.(+*Char, []*) *[deprecated]*

A list of one element evaluates to the character code of this element.[87] This implies `"a"` evaluates to the character code of the letter 'a' (97) using the traditional mapping of double quoted string to a list of character codes. *Char* is either a valid code point (non-negative integer up to the Prolog flag `max_char_code`) or a one-character atom. Arithmetic evaluation also translates a string object (see section 5.2) of one character length into the character code for that character. This implies that expression `"a"` works if the Prolog flag `double_quotes` is set to one of `codes`, `chars` or `string`.

Getting access to character codes this way originates from DEC10 Prolog. ISO has the `0'a` syntax and the predicate `char_code/2`. Future versions may drop support for `X is "a"`.

random(+*IntExpr*)

Evaluate to a random integer *i* for which $0 \leq i <$ *IntExpr*. The system has two implementations. If it is compiled with support for unbounded arithmetic (default) it uses the GMP library random functions. In this case, each thread keeps its own random state. The default algorithm is the *Mersenne Twister* algorithm. The seed is set when the first random number in a thread is generated. If available, it is set from `/dev/random`.[88] Otherwise it is set from the system clock. If unbounded arithmetic is not supported, random numbers are shared between threads and the seed is initialised from the clock when SWI-Prolog was started. The predicate `set_random/1` can be used to control the random number generator.

Warning! Although properly seeded (if supported on the OS), the Mersenne Twister algorithm does *not* produce cryptographically secure random numbers. To generate cryptographically secure random numbers, use `crypto_n_random_bytes/2` from library `crypto` provided by the `ssl` package.

random_float

Evaluate to a random float *I* for which $0.0 < i < 1.0$. This function shares the random state with `random/1`. All remarks with the function `random/1` also apply for `random_float/0`. Note that both sides of the domain are *open*. This avoids evaluation errors on, e.g., `log/1` or `//2` while no practical application can expect 0.0.[89]

round(+*Expr*) *[ISO]*

Evaluate *Expr* and round the result to the nearest integer. According to ISO, `round/1` is defined as `floor`(*Expr+1/2*), i.e., rounding *down*. This is an unconventional choice under which the relation `round(Expr) == -round(-Expr)` does not hold. SWI-Prolog rounds *outward*, e.g., `round(1.5) =:= 2` and `round(-1.5) =:= -2`.

[87] The function is documented as `./2`. Using SWI-Prolog v7 and later the actual functor is `'[|]'/2`.

[88] On Windows the state is initialised from CryptGenRandom().

[89] Richard O'Keefe said: "If you *are* generating IEEE doubles with the claimed uniformity, then 0 has a 1 in $2^{53} = 1 in 9,007,199,254,740,992$ chance of turning up. No program that expects [0.0,1.0) is going to be surprised when 0.0 fails to turn up in a few millions of millions of trials, now is it? But a program that expects (0.0,1.0) could be devastated if 0.0 did turn up."

integer(+*Expr*)
Same as `round/1` (backward compatibility).

float(+*Expr*) *[ISO]*
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 4.27.2 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
```

For every *normal* float X the relation $X =:=$ rational(X) holds.

This function raises an `evaluation_error`(*undefined*) if *Expr* is NaN and `evaluation_error`(*rational_overflow*) if *Expr* is Inf.

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.[90][91]

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

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

A = 1 rdiv 10
```

For every *normal* float X the relation $X =:=$ rationalize(X) holds.

This function raises the same exceptions as `rational/1` on non-normal floating point numbers.

[90] The names `rational/1` and `rationalize/1` as well as their semantics are inspired by Common Lisp.
[91] The implementation of rationalize as well as converting a rational number into a float is copied from ECLiPSe and covered by the *Cisco-style Mozilla Public License Version 1.1*.

numerator(*+RationalExpr*)

If *RationalExpr* evaluates to a rational number or integer, evaluate to the top/left value. Evaluates to itself if *RationalExpr* evaluates to an integer. See also `denominator/1`. The following is true for any rational *X*.

```
X =:= numerator(X)/denominator(X).
```

denominator(*+RationalExpr*)

If *RationalExpr* evaluates to a rational number or integer, evaluate to the bottom/right value. Evaluates to 1 (one) if *RationalExpr* evaluates to an integer. See also `numerator/1`. The following is true for any rational *X*.

```
X =:= numerator(X)/denominator(X).
```

float_fractional_part(*+Expr*) *[ISO]*

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: $X is float_fractional_part(X) + float_integer_part(X)$.

float_integer_part(*+Expr*) *[ISO]*

Integer part of floating point number. Negative if *Expr* is negative, *Expr* if *Expr* is integer.

truncate(*+Expr*) *[ISO]*

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*). That is, `truncate/1` rounds towards zero.

floor(*+Expr*) *[ISO]*

Evaluate *Expr* and return the largest integer smaller or equal to the result of the evaluation.

ceiling(*+Expr*) *[ISO]*

Evaluate *Expr* and return the smallest integer larger or equal to the result of the evaluation.

ceil(*+Expr*)

Same as `ceiling/1` (backward compatibility).

+IntExpr1 **>>** *+IntExpr2* *[ISO]*

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 bits.

+IntExpr1 **<<** *+IntExpr2* *[ISO]*

Bitwise shift *IntExpr1* by *IntExpr2* bits to the left.

+IntExpr1 **\/** *+IntExpr2* *[ISO]*

Bitwise 'or' *IntExpr1* and *IntExpr2*.

+IntExpr1 **/** *+IntExpr2* *[ISO]*

Bitwise 'and' *IntExpr1* and *IntExpr2*.

+IntExpr1 **xor** *+IntExpr2* *[ISO]*
Bitwise 'exclusive or' *IntExpr1* and *IntExpr2*.

**** *+IntExpr* *[ISO]*
Bitwise negation. The returned value is the one's complement of *IntExpr*.

sqrt(*+Expr*) *[ISO]*
Result $= \sqrt{Expr}$.

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

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

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

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

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

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

atan2(*+YExpr, +XExpr*) *[ISO]*
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.

Note that the ISO Prolog standard demands `atan2`(*0.0,0.0*) to raise an evaluation error, whereas the C99 and POSIX standards demand this to evaluate to 0.0. SWI-Prolog follows C99 and POSIX.

atan(*+YExpr, +XExpr*)
Same as `atan2/2` (backward compatibility).

sinh(*+Expr*)
Result $= \sinh Expr$. The hyperbolic sine of X is defined as $\frac{e^X - e^{-X}}{2}$.

cosh(*+Expr*)
Result $= \cosh Expr$. The hyperbolic cosine of X is defined as $\frac{e^X + e^{-X}}{2}$.

tanh(*+Expr*)
Result $= \tanh Expr$. The hyperbolic tangent of X is defined as $\frac{\sinh X}{\cosh X}$.

asinh(*+Expr*)
Result $= arcsinh(Expr)$ (inverse hyperbolic sine).

acosh(*+Expr*)
Result $= arccosh(Expr)$ (inverse hyperbolic cosine).

atanh(+*Expr*)
Result = $arctanh$(*Expr*). (inverse hyperbolic tangent).

log(+*Expr*) *[ISO]*
Natural logarithm. *Result* = $\ln$ *Expr*

log10(+*Expr*)
Base-10 logarithm. *Result* = $\lg$ *Expr*

exp(+*Expr*) *[ISO]*
Result = e^{Expr}

+*Expr1* ** +*Expr2* *[ISO]*
Result = $Expr1^{Expr2}$. The result is a float, unless SWI-Prolog is compiled with unbounded integer support and the inputs are integers and produce an integer result. The integer expressions 0^I, 1^I and -1^I are guaranteed to work for any integer I. Other integer base values generate a `resource` error if the result does not fit in memory.

The ISO standard demands a float result for all inputs and introduces `^/2` for integer exponentiation. The function `float/1` can be used on one or both arguments to force a floating point result. Note that casting the *input* result in a floating point computation, while casting the *output* performs integer exponentiation followed by a conversion to float.

+*Expr1* ^ +*Expr2* *[ISO]*
In SWI-Prolog, `^/2` is equivalent to `**/2`. The ISO version is similar, except that it produces a evaluation error if both *Expr1* and *Expr2* are integers and the result is not an integer. The table below illustrates the behaviour of the exponentiation functions in ISO and SWI. Note that if the exponent is negative the behavior of *Int^Int* depends on the flag `prefer_rationals`, producing either a rational number or a floating point number.

Expr1	*Expr2*	Function	SWI	ISO
Int	Int	`**/2`	Int or Rational	Float
Int	Float	`**/2`	Float	Float
Rational	Int	`**/2`	Rational	-
Float	Int	`**/2`	Float	Float
Float	Float	`**/2`	Float	Float
Int	Int	`^/2`	Int or Rational	Int or error
Int	Float	`^/2`	Float	Float
Rational	Int	`^/2`	Rational	-
Float	Int	`^/2`	Float	Float
Float	Float	`^/2`	Float	Float

powm(+*IntExprBase*, +*IntExprExp*, +*IntExprMod*)
Result = ($IntExprBase^{IntExprExp}$) modulo *IntExprMod*. Only available when compiled with unbounded integer support. This formula is required for Diffie-Hellman key-exchange, a technique where two parties can establish a secret key over a public network. *IntExprBase* and *IntExprExp* must be non-negative ($>= 0$), *IntExprMod* must be positive (> 0).[92]

[92]The underlying GMP mpz_powm() function allows negative values under some conditions. As the conditions are expensive to pre-compute, error handling from GMP is non-trivial and negative values are not needed for Diffie-Hellman key-exchange we do not support these.

lgamma(+*Expr*)
Return the natural logarithm of the absolute value of the Gamma function.[93]

erf(+*Expr*)
Wikipedia: "In mathematics, the error function (also called the Gauss error function) is a special function (non-elementary) of sigmoid shape which occurs in probability, statistics and partial differential equations."

erfc(+*Expr*)
Wikipedia: "The complementary error function."

pi *[ISO]*
Evaluate to the mathematical constant π (3.14159...).

e
Evaluate to the mathematical constant e (2.71828...).

epsilon
Evaluate to the difference between the float 1.0 and the first larger floating point number. Deprecated. The function `nexttoward/2` provides a better alternative.

inf
Evaluate to positive infinity. See section 2.16.1 and section 4.27.2. This value can be negated using `-/1`.

nan
Evaluate to *Not a Number*. See section 2.16.1 and section 4.27.2.

cputime
Evaluate to a floating point number expressing the CPU time (in seconds) used by Prolog up till now. See also `statistics/2` and `time/1`.

eval(+*Expr*)
Evaluate *Expr*. Although ISO standard dictates that 'A=1+2, B is A' works and unifies B to 3, it is widely felt that source level variables in arithmetic expressions should have been limited to numbers. In this view the eval function can be used to evaluate arbitrary expressions.[94]

Bitvector functions The functions below are not covered by the standard. The `msb/1` function also appears in hProlog and SICStus Prolog. The `getbit/2` function also appears in ECLiPSe, which also provides `setbit`(*Vector,Index*) and `clrbit`(*Vector,Index*). The others are SWI-Prolog 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.

[93] Some interfaces also provide the sign of the Gamma function. We cannot do that in an arithmetic function. Future versions may provide a *predicate* `lgamma/3` that returns both the value and the sign.
[94] The `eval/1` function was first introduced by ECLiPSe and is under consideration for YAP.

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*.

getbit(*+IntExprV, +IntExprI*)
Evaluates to the bit value (0 or 1) of the *IntExprI*-th bit of *IntExprV*. Both arguments must evaluate to non-negative integers. The result is equivalent to `(IntExprV >> IntExprI)/\1`, but more efficient because materialization of the shifted value is avoided. Future versions will optimise `(IntExprV >> IntExprI)/\1` to a call to `getbit/2`, providing both portability and performance.[95]

4.28 Misc arithmetic support predicates

set_random(*+Option*)
Controls the random number generator accessible through the *functions* `random/1` and `random_float/0`. Note that the library `random` provides an alternative API to the same random primitives.

seed(*+Seed*)
Set the seed of the random generator for this thread. *Seed* is an integer or the atom `random`. If `random`, repeat the initialization procedure described with the function `random/1`. Here is an example:

```
?- set_random(seed(111)), A is random(6).
A = 5.
?- set_random(seed(111)), A is random(6).
A = 5.
```

state(*+State*)
Set the generator to a state fetched using the state property of `random_property/1`. Using other values may lead to undefined behaviour.[96]

random_property(*?Option*)
True when *Option* is a current property of the random generator. Currently, this predicate provides access to the state. This predicate is not present on systems where the state is inaccessible.

state(*-State*)
Describes the current state of the random generator. State is a normal Prolog term that can be asserted or written to a file. Applications should make no other assumptions about its

[95] This issue was fiercely debated at the ISO standard mailinglist. The name *getbit* was selected for compatibility with ECLiPSe, the only system providing this support. Richard O'Keefe disliked the name and argued that efficient handling of the above implementation is the best choice for this functionality.

[96] The limitations of the underlying (GMP) library are unknown, which makes it impossible to validate the *State*.

representation. The only meaningful operation is to use as argument to `set_random/1` using the `state(`*State*`)` option.[97]

current_arithmetic_function(?*Head***)**
True when *Head* is an evaluable function. For example:

```
?- current_arithmetic_function(sin(_)).
true.
```

4.29 Built-in list operations

Most list operations are defined in the library `lists` described in section A.22. 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 compound term with name '`[|]`'[98] and arity 2 and the second argument is a list.[99] This predicate acts as if defined by the definition below on *acyclic* terms. The implementation safely *fails* if *Term* represents a cyclic list.

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

memberchk(?*Elem*, +*List***)** *[semidet]*
True when *Elem* is an element of *List*. This 'chk' variant of `member/2` is semi deterministic and typically used to test membership of a list. Raises a `type` error if scanning *List* encounters a non-list. Note that `memberchk/2` does *not* perform a full list typecheck. For example, `memberchk(a, [a|b])` succeeds without error. If *List* is cyclic and *Elem* is not a member of *List*, `memberchk/2` eventually raises a `type` error.[100]

length(?*List*, ?*Length***)** *[ISO]*
True if *Length* represents the number of elements in *List*. This predicate is a true relation and can be used to find the length of a list or produce a list (holding variables) of length *Length*.

[97] BUG: GMP provides no portable mechanism to fetch and restore the state. The current implementation works, but the state depends on the platform. I.e., it is generally not possible to reuse the state with another version of GMP or on a CPU with different datasizes or endian-ness.

[98] The traditional list functor name is the dot (`'.'`). This is still the case of the command line option `--traditional` is given. See also section 5.1.

[99] 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 conforms to 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.

[100] *Eventually* here means it will scan as many elements as the longest list that may exist given the current stack usage before raising the exception.

The predicate is non-deterministic, producing lists of increasing length if *List* is a *partial list* and *Length* is a variable.

```
?- length(List,4).
List = [_27940,_27946,_27952,_27958].

?- length(List,Length).
List = [], Length = 0 ;
List = [_24698], Length = 1 ;
List = [_24698,_25826], Length = 2
...
```

It raises errors if *Length* is bound to a non-integer or a negative integer or if *List* is neither a list nor a partial list. This error condition includes cyclic lists:[101]

```
?- A=[1,2,3|A], length(A,L).
ERROR: Type error: `list' expected ...
```

Covering an edge case, the predicate fails if the tail of *List* is equivalent to *Length*:[102]

```
?- List=[1,2,3|Length],length(List,Length).
false.

?- length(Length,Length).
false.
```

sort(+*List*, -*Sorted***)** *[ISO]*

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*.[103] The `sort/2` predicate can sort a cyclic list, returning a non-cyclic version with the same elements.

Note that *List* may contain non-ground terms. If *Sorted* is unbound at call-time, for each consecutive pair of elements in *Sorted*, the relation `E1 @< E2` will hold. However, unifying a variable in *Sorted* may cause this relation to become invalid, *even* unifying a variable in *Sorted* with another (older) variable. See also section 4.6.1.

sort(+*Key*, +*Order*, +*List*, -*Sorted***)**

True when *Sorted* can be unified with a list holding the element of *List*. *Key* determines which part of each element in *List* is used for comparing two term and *Order* describes the relation between each set of consecutive elements in *Sorted*.[104]

[101] ISO demands failure here. We think an error is more appropriate.

[102] This is logically correct. An exception would be more appropriate, but to our best knowledge, current practice in Prolog does not describe a suitable candidate exception term.

[103] Contributed by Richard O'Keefe.

[104] The definition of this predicate was established after discussion with Joachim Schimpf from the ECLiPSe team. ECLiPSe currently only accepts <, =<, > and >= for the *Order* argument but this is likely to change. SWI-Prolog extends this predicate to deal with dicts.

If *Key* is the integer zero (0), the entire term is used to compare two elements. Using *Key*=0 can be used to sort arbitrary Prolog terms. Other values for *Key* can only be used with compound terms or dicts (see section 5.4). An integer key extracts the *Key*-th argument from a compound term. An integer or atom key extracts the value from a dict that is associated with the given key. A type_error is raised if the list element is of the wrong type and an existence_error is raised if the compound has not enough argument or the dict does not contain the requested key.

Deeper nested elements of structures can be selected by using a list of keys for the *Key* argument.

The *Order* argument is described in the table below:[105]

Order	Ordering	Duplicate handling
`@<`	ascending	remove
`@=<`	ascending	keep
`@>`	descending	remove
`@>=`	descending	keep

The sort is *stable*, which implies that, if duplicates are kept, the order of duplicates is not changed. If duplicates are removed, only the first element of a sequence of duplicates appears in *Sorted*.

This predicate supersedes most of the other sorting primitives, for example:

```
sort(List, Sorted)      :- sort(0,  @<, List,  Sorted).
msort(List, Sorted)     :- sort(0, @=<, List,  Sorted).
keysort(Pairs, Sorted) :- sort(1, @=<, Pairs, Sorted).
```

The following example sorts a list of rows, for example resulting from `csv_read_file/2`) ascending on the 3th column and descending on the 4th column:

```
    sort(4, @>=, Rows0, Rows1),
    sort(3, @=<, Rows1, Sorted).
```

See also `sort/2` (ISO), `msort/2`, `keysort/2`, `predsort/3` and `order_by/2`.

msort(*+List, -Sorted*)
Equivalent to `sort/2`, but does not remove duplicates. Raises a `type_error` if *List* is a cyclic list or not a list.

keysort(*+List, -Sorted*) *[ISO]*
Sort a list of *pairs*. *List* must be a list of *Key*−*Value* pairs, terms whose principal functor is (-)/2. *List* is sorted on *Key* according to the standard order of terms (see section 4.6.1). Duplicates are *not* removed. Sorting is *stable* with regard to the order of the *Values*, i.e., the order of multiple elements that have the same *Key* is not changed.

The `keysort/2` predicate is often used together with library `pairs`. It can be used to sort lists on different or multiple criteria. For example, the following predicates sorts a list of atoms according to their length, maintaining the initial order for atoms that have the same length.

[105] For compatibility with ECLiPSe, the values <, =<, > and >= are allowed as synonyms.

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

sort_atoms_by_length(Atoms, ByLength) :-
        map_list_to_pairs(atom_length, Atoms, Pairs),
        keysort(Pairs, Sorted),
        pairs_values(Sorted, ByLength).
```

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 =. Duplicates are removed (i.e. equivalence classes of elements as defined by *Pred* are collapsed to a single element in *Sorted*) If the built-in predicate `compare/3` is used, the result is the same as `sort/2`. See also `keysort/2`.

4.30 Finding all Solutions to a Goal

findall(*+Template, :Goal, -Bag*) *[ISO]*
: Create a list of the instantiations *Template* gets successively on backtracking over *Goal* and unify 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 appearing in *Goal* scoped to the *Goal* with an existential (caret) operator (`^`), except that `bagof/3` fails when *Goal* has no solutions.

findall(*+Template, :Goal, -Bag, +Tail*)
: As `findall/3`, but returns the result as the difference list *Bag-Tail*. The 3-argument version is defined as

```
findall(Templ, Goal, Bag) :-
        findall(Templ, Goal, Bag, [])
```

findnsols(*+N, @Template, :Goal, -List*) *[nondet]*
findnsols(*+N, @Template, :Goal, -List, ?Tail*) *[nondet]*
: As `findall/3` and `findall/4`, but generates at most *N* solutions. If *N* solutions are returned, this predicate succeeds with a choice point if *Goal* has a choice point. Backtracking returns the next chunk of (at most) *N* solutions. In addition to passing a plain integer for *N*, a term of the form `count`(*N*) is accepted. Using `count`(*N*), the size of the next chunk can be controlled using `nb_setarg/3`. The non-deterministic behaviour used to implement the *chunk* option in `pengines`. Based on Ciao, but the Ciao version is deterministic. Portability can be achieved by wrapping the goal in `once/1`. Below are three examples. The first illustrates standard chunking of answers. The second illustrates that the chunk size can be adjusted dynamically and the last illustrates that no choice point is left if *Goal* leaves no choice-point after the last solution.

```
?- findnsols(5, I, between(1, 12, I), L).
L = [1, 2, 3, 4, 5] ;
L = [6, 7, 8, 9, 10] ;
L = [11, 12].

?- State = count(2),
   findnsols(State, I, between(1, 12, I), L),
   nb_setarg(1, State, 5).
State = count(5), L = [1, 2] ;
State = count(5), L = [3, 4, 5, 6, 7] ;
State = count(5), L = [8, 9, 10, 11, 12].

?- findnsols(4, I, between(1, 4, I), L).
L = [1, 2, 3, 4].
```

bagof(+*Template, :Goal, -Bag*) *[ISO]*

Unify *Bag* with the alternatives of *Template*. If *Goal* has free variables besides the one sharing with *Template*, `bagof/3` will backtrack over the alternatives of these free variables, unifying *Bag* with the corresponding alternatives of *Template*. The construct +*Var*^*Goal* tells `bagof/3` 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).
true.

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].

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].

5 ?-
```

setof(+*Template, +Goal, -Set*) *[ISO]*

Equivalent to `bagof/3`, but sorts the result using `sort/2` to get a sorted list of alternatives without duplicates.

4.31 Forall

forall(*:Cond, :Action*) *[semidet]*

For all alternative bindings of *Cond*, *Action* can be proven. The example verifies that all arithmetic statements in the given list 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).
```

The predicate `forall/2` is implemented as `\+ ( Cond, \+ Action)`, i.e., *There is no instantiation of Cond for which Action is false.*. The use of double negation implies that `forall/2` *does not change any variable bindings*. It proves a relation. The `forall/2` control structure can be used for its side-effects. E.g., the following asserts relations in a list into the dynamic database:

```
?- forall(member(Child-Parent, ChildPairs),
          assertz(child_of(Child, Parent))).
```

Using `forall/2` as `forall(`*Generator, SideEffect*`)` is preferred over the classical *failure driven loop* as shown below because it makes it explicit which part of the construct is the generator and which part creates the side effects. Also, unexpected failure of the side effect causes the construct to fail. Failure makes it evident that there is an issue with the code, while a failure driven loop would succeed with an erroneous result.

```
        ...,
        (   Generator,
            SideEffect,
            fail
        ;   true
        )
```

If your intent is to create variable bindings, the `forall/2` control structure is inadequate. Possibly you are looking for `maplist/2`, `findall/3` or `foreach/2`.

4.32 Formatted Write

The current version of SWI-Prolog provides two formatted write predicates. The 'writef' family (`writef/1`, `writef/2`, `swritef/3`), is compatible with Edinburgh C-Prolog and should be considered *deprecated*. The 'format' family (`format/1`, `format/2`, `format/3`), was defined by Quintus Prolog and currently available in many Prolog systems, although the details vary.

4.32.1 Writef

writef(*+Atom*) *[deprecated]*

Equivalent to `writef(Atom, [])`. See `writef/2` for details.

writef(*+Format, +Arguments*) *[deprecated]*

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* 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 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 2.16.1.

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*) *[deprecated]*

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*) *[deprecated]*

Equivalent to `swritef(String, Format, []).`

4.32.2 Format

The format family of predicates is the most versatile and portable[106] way to produce textual output.

format(*+Format*)

Defined as '`format(Format) :- format(Format, []).`'. See `format/2` for details.

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 single argument is not a list, 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, optionally followed by a colon modifier (:), [107] 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 which case the numeric argument is taken from the next argument of the argument list, which should be a positive integer. E.g., the following three examples all pass 46 (`.`) to `~t`:

```
?- format('~w ~46t ~w~72|~n', ['Title', 'Page']).
?- format('~w ~`.t ~w~72|~n', ['Title', 'Page']).
?- format('~w ~*t ~w~72|~n', ['Title', 46, 'Page']).
```

Some format expressions may call back Prolog, i.e., `~p`, `~W`, `~@` and user defined extensions registered with `format_predicate/2`. Output written to the stream `current_output` is merged into the `format/2` output. If there is no pending *rubber* (`~t`) and the the position notation aligns, only the output is switched. Otherwise the output is captured in a temporary memory buffer and emitted after the callback finishes. The system attempts to preserve the position and alignment promises. It sets the `tty` property of the temporary stream to reflect the main stream and uses the position information of the temporary stream to update its notion of the position. Notable `ansi_format/3` cooperates properly in callbacks.[108].

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 4.27.2). The floating point conversions allow for unlimited precision for printing rational numbers in decimal form. E.g., the following will write as many 3's as you want by changing the '50'.

[106] Unfortunately not covered by any standard.
[107] The colon modifiers is a SWI-Prolog extension, proposed by Richard O'Keefe.
[108] As of version 8.3.30

```
?- format('~50f', [10 rdiv 3]).
3.33333333333333333333333333333333333333333333333333
```

- `~` Output the tilde itself.
- `a` Output the next argument, which must be an atom. This option is equivalent to **w**, except that it requires the argument to be an atom.
- `c` Interpret the next argument as a character code and add it to the output. This argument must be a valid Unicode character code. Note that the actually emitted bytes are defined by the character encoding of the output stream and an exception may be raised if the output stream is not capable of representing the requested Unicode character. See section 2.19.1 for details.
- `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).

  The colon modifier (e.g., `~:d`) causes the number to be printed according to the locale of the output stream. See section 4.23.
- `D` Same as **d**, but makes large values easier to read by inserting a comma every three digits left or right of the dot. This is the same as `~:d`, but using the fixed English locale.
- `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. The numeric argument defines the number of digits right of the decimal point. If the colon modifier (:) is used, the float is formatted using conventions from the current locale, which may define the decimal point as well as grouping of digits left of the decimal point.
- `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.
- `I` Emit a decimal number using Prolog digit grouping (the underscore, _). The argument describes the size of each digit group. The default is 3. See also section 2.16.1. For example:

  ```
  ?- A is 1<<100, format('~10I', [A]).
  1_2676506002_2822940149_6703205376
  ```
- `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 numeric argument notation (default 8). Thus `~16r` prints its argument hexadecimal. The argument should be in the range $[2, \ldots, 36]$. Lowercase letters are used for digits above 9. The colon modifier may be used to form locale-specific digit groups.

R Same as **r**, but uses uppercase letters for digits above 9.

s Output text from a list of character codes, characters, string (see `string/1` and section 5.2) or atom from the next argument. If an numeric argument is given the string is truncated to this number of characters.

@ 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, it 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.

If the current column is at or past the requested tabstop and the modifier (:) is used, a newline is inserted and the padding character of the last `~t` is used to pad to the requested position.

+ Set a tab stop (as `~|`) relative to the last tab stop or the beginning of the line if no tab stops are set before the `~+`. This constructs can be used to fill fields. The partial format sequence below prints an integer right-aligned and padded with zeros in 6 columns. The ... sequences in the example illustrate that the integer is aligned in 6 columns regardless of the remainder of the format specification.

```
format('...~|~`0t~d~6+...', [..., Integer, ...])
```

w Give the next argument to `write/1`.

W Give the next two arguments to `write_term/2`. For example, `format('~W', [Term, [numbervars(true)]])`. 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`.[109] 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/3` and friends 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 a character code or a one-character atom, specifying the letter to be (re)defined. *Head* is a term, whose name and arity are used to determine the predicate to call for the redefined formatting character. The first argument to the predicate is the numeric argument of the format command, or the atom `default` if no argument is specified. The remaining arguments are filled from the argument list. The example below defines `~T` to print a timestamp in ISO8601 format (see `format_time/3`). The subsequent block illustrates a possible call.

```
:- format_predicate('T', format_time(_Arg,_Time)).

format_time(_Arg, Stamp) :-
        must_be(number, Stamp),
        format_time(current_output, '%FT%T%z', Stamp).
```

```
?- get_time(Now),
   format('Now, it is ~T~n', [Now]).
Now, it is 2012-06-04T19:02:01+0200
Now = 1338829321.6620328.
```

current_format_predicate(*?Code, ?:Head*)
True when `~`*Code* is handled by the user-defined predicate specified by *Head*.

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

4.33 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 of 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 do 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 `exception/3`. See also `nb_current/2` and `nb_delete/1`.

b_setval(+*Name*, +*Value*)
: Associate the term *Value* with the atom *Name* or replace the currently associated value with *Value*. On backtracking the assignment is reversed. If the variable *Name* did not exist before calling `b_setval/2`, backtracking causes the variable to be deleted.[110]

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.

[110] Prior to version 8.3.28 backtracking over the variable creation caused the variable to get the value `[]`, i.e., the empty list. If this is desirable use `nb_setval(Var, [])` before `b_setval/2`.

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 using either non-backtrackable or backtrackable assignment, using nb_getval/2 can be used to document that the variable is non-backtrackable. Raises existence_error(*variable, Name*) if the variable does not exist. Alternatively, nb_current/2 can used to query a global variable. This version *fails* if the variable does not exist rather than raising an exception.

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. 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. Note that nb_current/2 can be used as an alternative for nb_getval/2 to request the value of a variable and fail silently if the variable does not exists. Note that if the variable is not defined, exception/3 is called attempting to define it. As of version 8.3.28, a failure of exception/3 to define the variable causes the variable to be defined with a reserved valued to avoid subsequent calls to exception/3.

nb_delete(*+Name*)
Delete the named global variable. Succeeds also if the named variable does not exist. Deleting a global variable ensures the variable is associated to a reserved value to avoid subsequent calls to exception/3. Note that this implies that the resources associated with a global variable are never fully reclaimed.

4.33.1 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 of 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 that construct 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.

4.34 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 console 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 results 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).

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. The *ioctl* is issued on the file descriptor associated with the `user_input` stream. 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 `swipl-win.exe`. The requested value reflects the current size. For the multithreaded version the console that is associated with the `user_input` stream is used.

4.35 Operating System Interaction

The predicates in this section provide basic access to the operating system that has been part of the Prolog legacy tradition. Note that more advanced access to low-level OS features is provided by several libraries from the `clib` package, notably library `process`, `socket`, `unix` and `filesex`.

shell(+*Command***)**
: Equivalent to '`shell(Command, 0)`'. See `shell/2` for details.

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 Windows, `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 (`cmd.exe` and therefore commands that are built in the command interpreter can only be activated using the command interpreter. For example, a file can be copied using the command below.

```
?- shell('cmd.exe /C copy file1.txt file2.txt').
```

Note that many of the operations that can be achieved using the shell built-in commands can easily be achieved using Prolog primitives. See `make_directory/1`, `delete_file/1`, `rename_file/2`, etc. The clib package provides `filesex`, implementing various high level file operations such as `copy_file/2`. Using Prolog primitives instead of shell commands improves the portability of your program.

The library `process` provides `process_create/3` and several related primitives that support more fine-grained interaction with processes, including I/O redirection and management of asynchronous processes.

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 2.19.1. 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 using in multi-threaded environments. Locale settings are used by `format_time/3`, `collation_key/2` and `locale_sort/2`.

4.35.1 Windows-specific Operating System Interaction

The predicates in this section are only available on the Windows version of SWI-Prolog. Their use is discouraged if there are portable alternatives. For example, `win_exec/2` and `win_shell/2` can often be replaced by the more portable `shell/2` or the more powerful `process_create/3`.

win_exec(+*Command, +Show*)
Windows 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*)
Windows 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*)
Windows only. Fetches the value of a Windows 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
```

win_folder(*?Name, -Directory*)
True if *Name* is the Windows 'CSIDL' of *Directory*. If *Name* is unbound, all known Windows special paths are generated. *Name* is the CSIDL after deleting the leading `CSIDL_` and mapping the constant to lowercase. Check the Windows documentation for the function SHGetSpecialFolderPath() for a description of the defined constants. This example extracts the 'My Documents' folder:

```
?- win_folder(personal, MyDocuments).

MyDocuments = 'C:/Documents and Settings/jan/My Documents'
```

win_add_dll_directory(+*AbsDir*)
This predicate adds a directory to the search path for dependent DLL files. If possible, this is achieved with `win_add_dll_directory/2`. Otherwise, `%PATH%` is extended with the provided directory. *AbsDir* may be specified in the Prolog canonical syntax. See `prolog_to_os_filename/2`. Note that `use_foreign_library/1` passes an absolute path to the DLL if the destination DLL can be located from the specification using `absolute_file_name/3`.

win_add_dll_directory(+*AbsDir, -Cookie*)
This predicate adds a directory to the search path for dependent DLL files. If the call is successful it unifies *Cookie* with a handle that must be passed to `win_remove_dll_directory/1` to remove the directory from the search path. Error conditions:

- This predicate is available in the Windows port of SWI-Prolog starting from 6.3.8/6.2.6.
- This predicate *fails* if Windows does not yet support the underlying primitives. These are available in recently patched Windows 7 systems and later.
- This predicate throws an exception if the provided path is invalid or the underlying Windows API returns an error.

If `open_shared_object/2` is passed an *absolute* path to a DLL on a Windows installation that supports AddDllDirectory() and friends,[111] SWI-Prolog uses LoadLibraryEx() with the flags `LOAD_LIBRARY_SEARCH_DLL_LOAD_DIR` and `LOAD_LIBRARY_SEARCH_DEFAULT_DIRS`. In this scenario, directories from `%PATH%` and *not* searched. Additional directories can be added using `win_add_dll_directory/2`.

win_remove_dll_directory(-*Cookie*)
Remove a DLL search directory installed using `win_add_dll_directory/2`.

win_process_modules(-*FileNames*)
This predicate is a wrapper around EnumProcessModules(). *FileNames* is unified with a list of absolute paths for all *modules* of the Windows process. Modules are the main executable file and all DLLs loaded into the process, *except data DLLs*. The returned file names are in canonical Prolog representation. This predicate may be used to debug loading a DLL from an unexpected location and as a helper for packaging all dependencies when creating a distribution. According to the Windows documentation this API may return incorrect results if DLLs are loaded or unloaded while EnumProcessModules() is in progress. See also `qsave_program/2`.

win_get_user_preferred_ui_languages(+*Format, -Languages*)
Unifies *Languages* with a list of the user preferred languages (*Windows Display Languages*) in order of preference. If *Format* is `name`, the list elements are atoms. See Language Names

[111] Windows 7 with up-to-date patches or Windows 8.

for details. If *Format* is `id`, *Languages* is a list of numeric language ids represented as Prolog integers.

4.35.2 Apple specific Operating System Interaction

Non-portable Apple MacOS specific predicates are prefixed woth `apple_`.

apple_current_locale_identifier(*-Identifier*)
Unify *Identifier* with the value for CFLocaleGetIdentifier() of the Apple current locale. The *Identifier* is an atom that consists of the primary language identifier, e.g., `en` for english followed by an undercore and an identifier for the *Region* in the MacOS *Language & Region* preferences. For example, with the primary language set to "English (UK)" and the *Region* to "United Kingdom" we get `en_GB`. This relates to the locale identifier `en_GB.UTF-8`. Unfortunately it is not that simple. For example, we can combine the primary language "English (UK)" with the *Region* "Netherlands" to end up with `en_NL` which is not a valid MacOS locale.

4.35.3 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 a range from better than pico-seconds to millions of years. Finally, civilizations have a wide range of calendars. Although there exist libraries dealing with most of 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 deals with comparison and durations. An additional advantage of the physicist's 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.[112] We have placed our origin at 1970-01-01T0:0:0Z 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

[112] 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 the second as our unit.

TimeStamp
A TimeStamp is a floating point number expressing the time in seconds since the Epoch at 1970-01-01.

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) and 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 Greenwich. 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 4.35.3.

stamp_date_time(*+TimeStamp, -DateTime, +TimeZone*)
Convert a *TimeStamp* to a *DateTime* in the given timezone. See section 4.35.3 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 a UTC time or an integer describing the seconds west of Greenwich.

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 2006-07-14:

```
?- 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)
```

When computing a time stamp from a local time specification, the UTC offset (arg 7), TZ (arg 8) and DST (arg 9) argument may be left unbound and are unified with the proper information. The example below, executed in Amsterdam, illustrates this behaviour. On the 25th of March at 01:00, DST does not apply. At 02.00, the clock is advanced by one hour and thus both 02:00 and 03:00 represent the same time stamp.

```
1 ?- date_time_stamp(date(2012,3,25,1,0,0,UTCOff,TZ,DST),
                     Stamp).
UTCOff = -3600,
TZ = 'CET',
DST = false,
Stamp = 1332633600.0.

2 ?- date_time_stamp(date(2012,3,25,2,0,0,UTCOff,TZ,DST),
                     Stamp).
UTCOff = -7200,
TZ = 'CEST',
DST = true,
Stamp = 1332637200.0.

3 ?- date_time_stamp(date(2012,3,25,3,0,0,UTCOff,TZ,DST),
                     Stamp).
UTCOff = -7200,
TZ = 'CEST',
DST = true,
Stamp = 1332637200.0.
```

Note that DST and offset calculation are based on the POSIX function mktime(). If mktime() returns an error, a representation_error `dst` is generated.

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 in effect
`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 percent (%) character.[113] *StampOrDateTime* is either a numeric time-stamp, a term `date`(*Y,M,D,H,M,S,O,TZ,DST*) or a term `date`(*Y,M,D*).

[113]Descriptions taken from Linux Programmer's Manual

a The abbreviated weekday name according to the current locale. Use `format_time/4` for POSIX locale.

A The full weekday name according to the current locale. Use `format_time/4` for POSIX locale.

b The abbreviated month name according to the current locale. Use `format_time/4` for POSIX locale.

B The full month name according to the current locale. Use `format_time/4` for POSIX 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. (For Americans only. Americans should note that in other countries %d/%m/%y is rather common. This means that in an international context this format is ambiguous 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 Number of microseconds. The `f` can be prefixed by an integer to print the desired number of digits. E.g., `%3f` prints milliseconds. This format is not covered by any standard, but available with different format specifiers in various incarnations of the strftime() function.

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).

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 to select locale-specific output. 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'.[114]

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 timezone as hour offset from GMT using the format HHmm. Required to emit RFC822-conforming dates (using `'%a, %d %b %Y %T %z'`). Our implementation supports `%:z`, which modifies the output to HH:mm as required by XML-Schema. Note that both notations are valid in ISO 8601. The sequence `%:z` is compatible to the GNU date(1) command.

Z The timezone or name or abbreviation.

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

% A literal '%' character.

The table below gives some format strings for popular time representations. RFC1123 is used by HTTP. The full implementation of `http_timestamp/2` as available from `http/http_header` is here.

```
http_timestamp(Time, Atom) :-
        stamp_date_time(Time, Date, 'UTC'),
        format_time(atom(Atom),
                    '%a, %d %b %Y %T GMT',
                    Date, posix).
```

[114]Despite the above claim, some locales yield `am` or `pm` in lower case.

Standard	Format string
xsd	`'%FT%T%:z'`
ISO8601	`'%FT%T%z'`
RFC822	`'%a, %d %b %Y %T %z'`
RFC1123	`'%a, %d %b %Y %T GMT'`

format_time(+*Out*, +*Format*, +*StampOrDateTime*, +*Locale***)**
Format time given a specified *Locale*. This predicate is a work-around for lacking proper portable and thread-safe time and locale handling in current C libraries. In its current implementation the only value allowed for *Locale* is `posix`, which currently only modifies the behaviour of the `a`, `A`, `b` and `B` format specifiers. The predicate is used to be able to emit POSIX locale week and month names for emitting standardised time-stamps such as RFC1123.

parse_time(+*Text*, -*Stamp***)**
Same as `parse_time(`*Text*, *_Format*, *Stamp*`)`. See `parse_time/3`.

parse_time(+*Text*, ?*Format*, -*Stamp***)**
Parse a textual time representation, producing a time-stamp. Supported formats for *Text* are in the table below. If the format is known, it may be given to reduce parse time and avoid ambiguities. Otherwise, *Format* is unified with the format encountered.

Name	Example
rfc_1123	`Fri, 08 Dec 2006 15:29:44 GMT` `Fri, 08 Dec 2006 15:29:44 +0000`

iso_8601	`2006-12-08T17:29:44+02:00`
	`20061208T172944+0200`
	`2006-12-08T15:29Z`
	`2006-12-08`
	`20061208`
	`2006-12`
	`2006-W49-5`
	`2006-342`

day_of_the_week(+*Date*,-*DayOfTheWeek***)**
Computes the day of the week for a given date. *Date* = `date(`*Year*,*Month*,*Day*`)`. Days of the week are numbered from one to seven: Monday = 1, Tuesday = 2, ..., Sunday = 7.

4.35.4 Controlling the `swipl-win.exe` console window

The Windows executable `swipl-win.exe` console has a number of predicates to control the appearance of the console. Being totally non-portable, we do not advise 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*.[115]

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

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_window_color(+*Which*, +*RGB***)**
Change the color of the console window. *Which* is one of `foreground`, `background`, `selection_foreground` or `selection_background`. *RGB* is a term `rgb`(*Red,Green,Blue*) where the components are values between 0 and 255. The defaults are established using the Windows API GetSysColor().

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 convention, 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 an Application entry just before the Help menu.

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

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.

4.36 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`, `execute`, `search`, `exist`, or `none`. Fails silently otherwise. *File* may also be the name of a directory. `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.

The mode `execute` is only intended for use with regular files and the mode `search` only with directories. However, the two modes are currently equivalent and both can be used with either files or directories. This may change in the future, so the results of checking `execute` access on directories or `search` access on regular files should not be relied on.

The behaviour is backed up by the POSIX access() API. The Windows replacement (_waccess()) returns incorrect results because it does not consider ACLs (Access Control Lists). The Prolog flag `win_file_access_check` may be used to control the level of checking performed by Prolog. Please note that checking access never provides a guarantee that a subsequent open succeeds without errors due to inherent concurrency in file operations. It is generally more robust to try and open the file and handle possible exceptions. See `open/4` and `catch/3`.

exists_file(+*File*)
True if *File* exists and is a *regular* file. This does not imply the user has read or write access to the file. See also `exists_directory/1` and `access_file/2`.

file_directory_name(+*File, -Directory*)
Extracts the directory part of *File*. This predicate removes the longest match for the regular expression `/*[^/]*/*$`. If the result is empty it binds *Directory* to `/` if the first character of *File* is `/` and `.` otherwise. The behaviour is consistent with the POSIX `dirname` program.[116]

See also `directory_file_path/3` from `filesex`. The system ensures that for every valid *Path* using the Prolog (POSIX) directory separators, following is true on systems with a sound implementation of `same_file/2`.[117] See also `prolog_to_os_filename/2`.

```
        ...,
        file_directory_name(Path, Dir),
        file_base_name(Path, File),
        directory_file_path(Dir, File, Path2),
        same_file(Path, Path2).
```

file_base_name(+*Path, -File*)
Extracts the file name part from a path. Similar to `file_directory_name/2` the extraction is based on the regex `/*([^/]*)/*$`, now capturing the non-`/` segment. If the segment is empty it unifies *File* with `/` if *Path* starts with `/` and the empty atom (`''`) otherwise. The behaviour is consistent with the POSIX `basename` program.[118]

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). On systems that provide stat() with meaningful values for `st_dev` and `st_inode`, `same_file/2` is implemented by comparing the device and inode identifiers.

[116] Before SWI-Prolog 7.7.13 trailing `/` where *not* removed, translation `/a/b/` into `/a/b`. Volker Wysk pointed at this incorrect behaviour.

[117] On some systems, *Path* and *Path2* refer to the same entry in the file system, but `same_file/2` may fail.

[118] Before SWI-Prolog 7.7.13, if argPath ended with a `/` *File* was unified with the empty atom.

On Windows, `same_file/2` compares the strings returned by the GetFullPathName() system call.

exists_directory(+*Directory***)**
True if *Directory* exists and is a directory. This does not imply the user has read, search or write permission for the directory.

delete_file(+*File***)**
Remove *File* from the file system.

rename_file(+*File1*, +*File2***)**
Rename *File1* as *File2*. The semantics is compatible to the POSIX semantics of the rename() system call as far as the operating system allows. Notably, if *File2* exists, the operation succeeds (except for possible permission errors) and is *atomic* (meaning there is no window where *File2* does not exist).

size_file(+*File*, -*Size***)**
Unify *Size* with the size of *File* in bytes.

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 filename into an absolute path. The absolute path is canonicalised: references to `.`, `..` and repeated directory separators (`/`) are deleted. This predicate ensures that expanding a filename returns the same absolute path regardless of how the file is addressed. Notably, if a file appears in multiple directories due to symbolic or hard links `absolute_file_name/2` returns the same absolute filename. SWI-Prolog uses absolute filenames to register source files independent of the current working directory.

This predicate has a different history than `absolute_file_name/3` and should primarily be used to get an absolute canonical name from a relative name. If *File* is a term Alias(Relative) is behaviour is defined as below, i.e., if an accessible file can be found using the provided search path this is returned. Otherwise it returns the the expansion of the alias path.[119] Users are adviced to use `absolute_file_name/3` with a appropriate options for resolving an Alias(Relative) term.

```
absolute_file_name(Spec, AbsFile) :-
    absolute_file_name(Spec, File, [access(read), file_errors(fail)]),
    !,
    AbsFile = File.
absolute_file_name(Spec, AbsFile) :-
    absolute_file_name(Spec, AbsFile, []).
```

See also `absolute_file_name/3` and `expand_file_name/2`.

[119]The SICStus implementation behaves as `absolute_file_name/3` with an empty option list.

absolute_file_name(*+Spec, -Absolute, +Options*)
Convert the given file specification into an absolute path. *Spec* is a term Alias(Relative) (e.g., `(library(lists))`, a relative filename or an absolute filename. The primary intention of this predicate is to resolve files specified as Alias(Relative). This predicate *only returns non-directories*, unless the option `file_type`(*directory*) is specified or the requested access is `none`. Supported *Options* are:

extensions(*ListOfExtensions*)
List of file extensions to try. Default is exam['']. 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 on 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 the directory 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 one of `read`, `write`, `append`, `execute`, `search`, `exist` or `none`. See also `access_file/2`. The default is `none` which, if `file_type` is **not** specified as `directory` or `regular`, returns absolute file names that result from expanding aliases without inspecting the actual file system.

file_type(*Type*)
Defines extensions. Current mapping: `txt` implies `['']`, `prolog` implies `['.pl', '']`, `executable` implies `['.so', '']` and `qlf` implies `['.qlf', '']`. The *Type* `directory` implies `['']` and causes this predicate to generate (only) directories. The *Type* `regular` is the opposite of `directory` and is the default if no file type is specified and the effective access mode is `none`.

The file type `source` is an alias for `prolog` for compatibility with SICStus Prolog. See also `prolog_file_type/2`.

file_errors(*fail/error*)
If `error` (default), throw an `existence_error` exception if the file cannot be found. If `fail`, stay silent.[120]

solutions(*first/all*)
If `first` (default), the predicate leaves no choice point. Otherwise a choice point will be left and backtracking may yield more solutions.

expand(*Boolean*)
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 intended to minimise porting effort after SWI-Prolog stopped expanding environment variables and the `~` by default. This option should be considered deprecated. In particular the use of *wildcard* patterns such as `*` should be avoided.

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

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

This predicate is derived from Quintus Prolog. In Quintus Prolog, the argument order was `absolute_file_name`(*+Spec, +Options, -Path*). The argument order has been changed for compatibility with ISO and SICStus. The Quintus argument order is still accepted.

is_absolute_file_name(*+File*)
True if *File* specifies an 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 also be done case-insensitive. *Extension* may be specified with or without a leading dot (`.`). If an *Extension* is generated, it will not have a leading dot.

directory_files(*+Directory, -Entries*)
Unify *Entries* with a list of entries in *Directory*. Each member of *Entries* is an atom denoting an entry relative to *Directory*. *Entries* contains all entries, including hidden files and, if supplied by the OS, the special entries `.` and `..`. See also `expand_file_name/2`.[121]

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 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 contains wildcard characters, only existing files and directories are returned. Expanding a 'pattern' without wildcard characters returns the argument, regardless of whether or not it exists.

Before expanding wildcards, the construct `\$\arg{var}` is expanded to the value of the environment variable *var*, and a possible leading `~` character is expanded to the user's home directory.[122]

prolog_to_os_filename(*?PrologPath, ?OsPath*)
Convert between the internal Prolog path name conventions and the operating system path name conventions. The internal conventions follow the POSIX standard, which implies that this predicate is equivalent to =/2 (unify) on POSIX (e.g., Unix) systems. On Windows systems it changes the directory separator from `\` into `/`.

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

[121] This predicate should be considered a misnomer because it returns entries rather than files. We stick to this name for compatibility with, e.g., SICStus, Ciao and YAP.

[122] 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.

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*) *[deprecated]*
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. *Base* is used as part of the final filename. Portable applications should limit themselves to alphanumeric characters.

Because it is possible to guess the generated filename, attackers may create the filesystem entry as a link and possibly create a security issue. New code should use `tmp_file_stream/3`.

tmp_file_stream(*+Encoding, -FileName, -Stream*)
tmp_file_stream(*-FileName, -Stream, +Options*)
Create a temporary filename *FileName*, open it for writing and unify *Stream* with the output stream. If the OS supports it, the created file is only accessible to the current user and the file is created using the open()-flag `O_EXCL`, which guarantees that the file did not exist before this call. The following options are processed:

encoding(*+Encoding*)
Encoding of *Stream*. Default is the value of the Prolog flag `encoding`. The value `binary` opens the file in binary mode.

extension(*+Ext*)
Ensure the created file has the given extension. Default is no extension. Using an extension may be necessary to run external programs on the file.

This predicate is a safe replacement of `tmp_file/2`. Note that in those cases where the temporary file is needed to store output from an external command, the file must be closed first. E.g., the following downloads a file from a URL to a temporary file and opens the file for reading (on Unix systems you can delete the file for cleanup after opening it for reading):

```
open_url(URL, In) :-
        tmp_file_stream(text, File, Stream),
        close(Stream),
        process_create(curl, ['-o', File, URL], []),
        open(File, read, In),
        delete_file(File).              % Unix-only
```

Temporary files created using this call are removed if the Prolog process terminates *gracefully*. Calling `delete_file/1` using *FileName* removes the file and removes the entry from the administration of files-to-be-deleted.

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`.[123] Note that the working directory is shared between all threads.

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

4.37 User Top-level Manipulation

break
: Recursively start a new Prolog top level. This Prolog top level shares everything from the environment it was started in. 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 that is somehow not functional, the term `end_of_file.` can be entered to return from the break environment. If the `-t` *`toplevel`* command line option is given, this goal is started instead of entering the default interactive top level (`prolog/0`).

  Notably the gui based versions (`swipl-win` on Windows and MacOS) provide the menu Run/New thread that opens a new toplevel that runs concurrently with the initial toplevel. The concurrent toplevel can be used to examine the program, in particular global dynamic predicates. It can not access *global variables* or thread-local dynamic predicates (see `thread_local/1`) of the main thread.

abort
: Abort the Prolog execution and restart the top level. If the `-t` *`toplevel`* command line option is given, this goal is restarted instead of entering the default interactive top level.

  Aborting is implemented by throwing the reserved exception `'$aborted'`. This exception can be caught using `catch/3`, but the recovery goal is wrapped with a predicate that prunes the choice points of the recovery goal (i.e., as `once/1`) and re-throws the exception. This is illustrated in the example below, where we press control-C and 'a'. See also section 4.10.1.

```
?- catch((repeat,fail), E, true).
^CAction (h for help) ? abort
% Execution Aborted
```

halt *[ISO]*
: Terminate Prolog execution with default exit code using `halt/1`. The default exit code is normally 0, but can be 1 if one of the Prolog flags `on_error` or `on_warning` is set to `status` and there have been errors or warnings.

[123] 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`.

halt(+*Status*) *[ISO]*

Terminate Prolog execution with *Status*. This predicate calls `PL_halt()` which preforms the following steps:

1. Set the Prolog flag `exit_status` to *Status*.
2. Call all hooks registered using `at_halt/1`. If *Status* equals 0 (zero), any of these hooks calls `cancel_halt/1`, termination is cancelled.
3. Call all hooks registered using `PL_at_halt()`. In the future, if any of these hooks returns non-zero, termination will be cancelled. Currently, this only prints a warning.
4. Perform the following system cleanup actions:
   - Cancel all threads, calling `thread_at_exit/1` registered termination hooks. Threads not responding within 1 second are cancelled forcefully.
   - Flush I/O and close all streams except for standard I/O.
   - Reset the terminal if its properties were changed.
   - Remove temporary files and incomplete compilation output.
   - Reclaim memory.
5. Call exit(Status) to terminate the process

`halt/1` has been extended in SWI-Prolog to accept the arg `abort`. This performs as `halt/1` above except that:

- Termination cannot be cancelled with `cancel_halt/1`.
- abort() is called instead of exit(Status).

prolog

This goal starts the default interactive top level. Queries are read from the stream `user_input`. See also the Prolog flag `history`. The `prolog/0` predicate is terminated (succeeds) by typing the end-of-file character (typically 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 2.9.

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.

4.38 Creating a Protocol of the User Interaction

SWI-Prolog offers the possibility to log the interaction with the user on a file.[124] All Prolog interaction, including warnings and tracer output, are written to 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.

4.39 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 2.10.

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`.

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 cannot be seen in the tracer.

notrace
Stop the tracer. `notrace/0` itself cannot be seen in the 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. Tracing a predicate is achieved by *wrapping* the predicate using `wrap_predicate/4`.

Each time a port (of the 4-port model) is passed that has a trace point set, the goal is printed. Unlike `trace/0`, however, the execution is continued without asking for further information. Examples:

[124] A similar facility was added to Edinburgh C-Prolog by Wouter Jansweijer.

`?- 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.

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 break points, disables last-call optimisation and aggressive destruction of choice points to make debugging information accessible. Implemented by the Prolog flag `debug`.

Note that the `min_free` parameter of all stacks is enlarged to 8 K cells if debugging is switched off in order to avoid excessive GC. GC complicates tracing because it renames the _⟨*NNN*⟩ variables and replaces unreachable variables with the atom <garbage_collected>. Calling `nodebug/0` does *not* reset the initial free-margin because several parts of the top level and debugger disable debugging of system code regions. See also `set_prolog_stack/2`.

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 A.21.

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 *Ports* 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*)
Modify/query style checking options. *Spec* is one of the terms below or a list of these.

- +*Style* enables a style check
- -*Style* disables a style check
- ?(*Style*) queries a style check (note the brackets). If *Style* is unbound, all active style check options are returned on backtracking.

Loading a file using `load_files/2` or one of its derived predicates reset the style checking options to their value before loading the file, scoping the option to the remainder of the file and all files loaded *after* changing the style checking.

singleton(*true*)
The predicate `read_clause/3` (used by the compiler to read source code) warns on variables appearing only once in a term (clause) which have a name not starting with an underscore. See section 2.16.1 for details on variable handling and warnings.

no_effect(*true*)
This warning is generated by the compiler for BIPs (built-in predicates) that are inlined by the compiler and for which the compiler can prove that they are meaningless. An example is using `==/2` against a not-yet-initialised variable as illustrated in the example below. This comparison is always `false`.

```
always_false(X) :-
        X == Y,
        write(Y).
```

var_branches(*false*)
Verifies that if a variable is introduced in a branch and used *after* the branch, it is introduced in all branches. This code aims at bugs following the skeleton below, where `p`(*Next*) may be called with *Next* unbound.

```
p(Arg) :-
        (   Cond
        ->  Next = value1
        ;   true
        ),
        p(Next).
```

If a variable *V* is intended to be left unbound, one can use `V=_`. This construct is removed by the compiler and thus has no implications for the performance of your program.

This check was suggested together with *semantic* singleton checking. The SWI-Prolog libraries contain about a hundred clauses that are triggered by this style check. Unlike semantic singleton analysis, only a tiny fraction of these clauses proofed faulty. In most cases, the branches failing to bind the variable fail or raise an exception or the caller handles the case where the variable is unbound. The status of this style check is unclear. It might be removed in the future or it might be enhanced with a deeper analysis to be more precise.

discontiguous(*true*)
Warn if the clauses for a predicate are not together in the same source file. It is advised to disable the warning for discontiguous predicates using the `discontiguous/1` directive.

charset(*false*)
Warn on atoms and variable names holding non-ASCII characters that are not quoted. See also section 2.16.1.

4.40 Debugging and declaring determinism

A common issue with Prolog programs of a *procedural* nature is to guarantee deterministic behaviour and debug possible problems with determinism. SWI-Prolog provides several mechanisms to make writing, debugging and maintaining deterministic code easier. One of them is *Single Sided Unification* using `=>/2` rules as described in section 5.6. This section deals with annotating your program.

> **WARNING:** The primitives in this section are experimental. The naming and exact semantics may change. If you are interested in this, please follow and contribute to discussion on the Discourse forum.

det(*+PredicateIndicators*) *[experimental,directive]*
Declare a number of predicates as `det`, *deterministic*. As a result, both failure and success with a choicepoint is considered an error. The behaviour if the declaration is violated is controlled with the Prolog flag `determinism_error`. The default is to raise an exception (`error`). Consider the following program:

```
:- det(p/1).

p(1).
p(2).
```

Now, a call `?- p(1).` behaves normally. However:

```
?- p(X).
ERROR: Deterministic procedure p/1 succeeded with a choicepoint
ERROR: In:
ERROR:   [10] p(1)

?- p(a).
ERROR: Deterministic procedure p/1 failed
ERROR: In:
ERROR:   [10] p(a)
```

$ *[experimental]*
The `$/0` constructs acts similar to the `!/0`, but in addition declares that the remainder of the clause body shall succeed deterministically. It exploits the same underlying mechanism as the `det/1` declaration. See also `$/1`.

$(*:Goal*) *[experimental]*

Verify that *Goal* succeeds deterministically. This predicate has no effect if *Goal* succeeds without a choicepoint. Otherwise the result depends on the Prolog flag `determinism_error`:

silent
Act as `once/1`.

warning
Print a warning and act as `once/1`.

error
Raise a `determinism_error` exception.

Note that if `$/1` is used for the last call, last call optimization is not effective. This behaviour ensures consistent errors or warnings. Last call optimization with determinism checking can be realised using `..., $, Last.`, i.e. by executing `$/0` before the last call rather than wrapping the last call in `$/1`.

A deterministic predicate may call normal predicates. No error is triggered as long as the deterministic predicate either ignores a possible failure, e.g., using `\+/1` and prunes possible choice points created by called predicates. If the last predicate is a normal predicate the requirement to succeed deterministically is transferred to the new goal. As last-call optimization causes the information which predicate initially claimed to be deterministic to be lost, the error is associated with the called predicate. Debug mode (see `debug/0` or the Prolog flag `debug`) may be used to avoid last call optimization and find the call stack that causes the issue.

4.41 Obtaining Runtime Statistics

statistics(*+Key, -Value*)

Unify system statistics determined by *Key* with *Value*. The possible keys are given in the table 4.3. This predicate supports additional keys for compatibility reasons. These keys are described in table 4.4.

statistics

Display a table of system statistics on the stream `user_error`.

time(*:Goal*)

Execute *Goal* just like `call/1` and print time used, 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. If *Goal* is non-deterministic, print statistics for each solution, where the reported values are relative to the previous answer.

4.42 Execution profiling

This section describes the hierarchical execution profiler. This profiler is based on ideas from `gprof` described in [Graham *et al.*, 1982]. The profiler consists of two parts: the information-gathering com-

Native keys (times as float in seconds)	
agc	Number of atom garbage collections performed
agc_gained	Number of atoms removed
agc_time	Time spent in atom garbage collections
atoms	Total number of defined atoms
atom_space	Bytes used to represent atoms
c_stack	System (C-) stack limit. 0 if not known.
cgc	Number of clause garbage collections performed
cgc_gained	Number of clauses reclaimed
cgc_time	Time spent in clause garbage collections
clauses	Total number of clauses in the program
codes	Total size of (virtual) executable code in words
cputime	(User) CPU time since thread was started in seconds
epoch	Time stamp when thread was started
errors	Number of error mesages printed
functors	Total number of defined name/arity pairs
functor_space	Bytes used to represent functors
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
global_shifts	Number of global stack expansions
heapused	Bytes of heap in use by Prolog (0 if not maintained)
inferences	Total number of passes via the call and redo ports since Prolog was started
modules	Total number of defined modules
local	Allocated size of the local stack in bytes
local_shifts	Number of local stack expansions
locallimit	Size to which the local stack is allowed to grow
localused	Number of bytes in use on the local stack
table_space_used	Amount of bytes in use by the thread's answer tables
trail	Allocated size of the trail stack in bytes
trail_shifts	Number of trail stack expansions
traillimit	Size to which the trail stack is allowed to grow
trailused	Number of bytes in use on the trail stack
shift_time	Time spent in stack-shifts
stack	Total memory in use for stacks in all threads
predicates	Total number of predicates. This includes predicates that are undefined or not yet resolved.
indexes_created	Number of clause index tables creates.
indexes_destroyed	Number of clause index tables destroyed.
process_epoch	Time stamp when Prolog was started
process_cputime	(User) CPU time since Prolog was started in seconds
thread_cputime	MT-version: Seconds CPU time used by **finished** threads. The implementation requires non-portable functionality. Currently works on Linux, MacOSX, Windows and probably some more.
threads	MT-version: number of active threads
threads_created	MT-version: number of created threads
engines	MT-version: number of existing engines
engines_created	MT-version: number of created engines
threads_peak	MT-version: highest id handed out. This is a fair but possibly not 100% accurate value for the highest number of threads since the process was created.

Compatibility keys (times in milliseconds)	
runtime	[CPU time, CPU time since last] (milliseconds, excluding time spent in garbage collection)
system_time	[System CPU time, System CPU time since last] (milliseconds)
real_time	[Wall time, Wall time since last] (integer seconds. See `get_time/1`)
walltime	[Wall time since start, Wall time since last] (milliseconds, SICStus compatibility)
memory	[Total unshared data, free memory] (Used is based on `ru_idrss` from getrusage(). Free is based on `RLIMIT_DATA` from getrlimit(). Both are reported as zero if the OS lacks support. Free is -1 if getrlimit() is supported but returns infinity.)
stacks	[global use, local use]
program	[heap use, 0]
global_stack	[global use, global free]
local_stack	[local use, local free]
trail	[trail use, trail free]
garbage_collection	[number of GC, bytes gained, time spent, bytes left] The last column is a SWI-Prolog extension. It contains the sum of the memory left after each collection, which can be divided by the count to find the average working set size after GC. Use `[Count, Gained, Time|_]` for compatibility.
stack_shifts	[global shifts, local shifts, time spent]
atoms	[number, memory use, 0]
atom_garbage_collection	[number of AGC, bytes gained, time spent]
clause_garbage_collection	[number of CGC, clauses gained, time spent]
core	Same as memory

Table 4.4: Compatibility keys for `statistics/2`. Time is expressed in milliseconds.

ponent built into the kernel,[125] 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 frontend for the results.

4.42.1 Profiling predicates

The following predicates are defined to interact with the profiler.

profile(*:Goal*)
Execute *Goal* just like `once/1`, collecting profiling statistics, and call `show_profile([])`. With XPCE installed this opens a graphical interface to examine the collected profiling data.

profile(*:Goal, +Options*)
Execute *Goal* just like `once/1`. Collect profiling statistics according to *Options* and call `show_profile/1` with *Options*. The default collects CPU profiling and opens a graphical interface when provided, printing the 'plain' time usage of the top 25 predicates as a ballback. Options are described below. Remaining options are passed to `show_profile/1`.

time(*+Which*)
If *Which* is `cpu` (default), collect CPU timing statistics. If `wall`, collect wall time statistics based on a 5 millisecond sampling rate. Wall time statistics can be useful if *Goal* calls blocking system calls.

show_profile(*+Options*)
This predicate first calls `prolog:show_profile_hook/1`. If XPCE is loaded, this hook is used to activate a GUI interface to visualise the profile results. If not, a report is printed to the terminal according to *Options*:

top(*+N*)
Show the only top *N* predicates. Default is 25.

cumulative(*+Bool*)
If `true` (default `false`), include the time spent in children in the time reported for a predicate.

profiler(*-Old, +New*)
Query or change the status of the profiler. The status is one of

false
The profiler is not activated.

cputime
The profiler collects CPU statistics.

walltime
The profiler collects wall time statistics.

The value `true` is accepted as a synonym for `cputime` for compatibility reasons.

[125] 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.

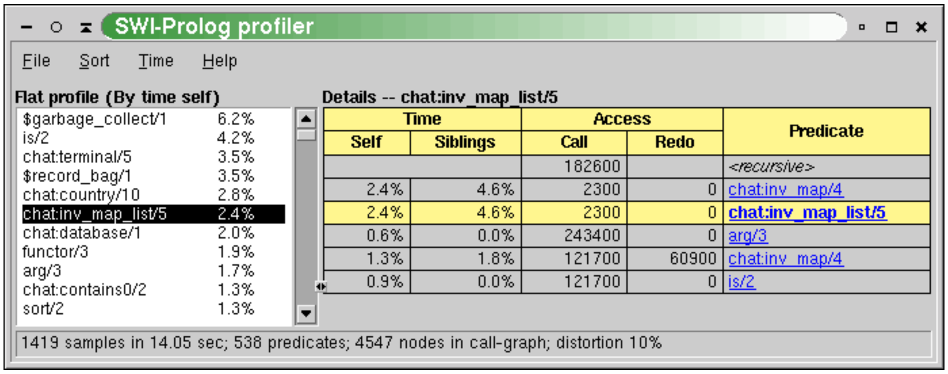

Figure 4.1: Execution profiler showing the activity of the predicate `chat:inv_map_list/5`.

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 spent 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.

4.42.2 Visualizing profiling data

Browsing the annotated call-tree as described in section 4.42.3 itself is not very attractive. Therefore, the results are combined per predicate, collecting all *callers* 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*> is the number of recursive calls. Below the yellowish lines 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 spent 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.

4.42.3 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 that 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 some 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 4.42.3 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[126] 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 CPU time in milliseconds used by Prolog since the last time it was fired. If this value is non-zero, active predicates are incremented with this value.

4.43 Memory Management

4.43.1 Garbage collection

garbage_collect
Invoke the global and trail stack garbage collector. Normally the garbage collector is invoked

[126] We hereby acknowledge Lionel Fourquaux, who suggested the design described here after a newsnet enquiry.

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 multithreaded versions the actual collection is delayed until there are no threads performing normal garbage collection. In this case `garbage_collect_atoms/0` returns immediately. Note that there is no guarantee it will *ever* happen, as there may always be threads performing garbage collection.

garbage_collect_clauses

Reclaim retracted clauses. During normal operation, retracting a clause implies setting the *erased generation* to the current *generation* of the database and increment the generation. Keeping the clause around is both needed to realise the *logical update view* and deal with the fact that other threads may be executing the clause. Both static and dynamic code is processed this way.[127].

The clause garbage collector (CGC) scans the environment stacks of all threads for referenced dirty predicates and at which generation this reference accesses the predicate. It then removes the references for clauses that have been retracted before the oldest access generation from the clause list as well as the secondary clauses indexes of the predicate. If the clause list is not being scanned, the clause references and ultimately the clause itself is reclaimed.

The clause garbage collector is called under three conditions, (1) after *reloading* a source file, (2) if the memory occupied by retracted but not yet reclaimed clauses exceeds 12.5% of the program store, or (3) if skipping dead clauses in the clause lists becomes too costly. The cost of clause garbage collection is proportional with the total size of the local stack of all threads (the scanning phase) and the number of clauses in all 'dirty' predicates (the reclaiming phase).

set_prolog_gc_thread(+*Status*)

Control whether or not atom and clause garbage collection are executed in a dedicated thread. The default is `true`. Values for *Status* are `true`, `false` and `stop`. The latter stops the `gc` thread but allows is to be recreated lazily. This is use by e.g., `fork/1` to avoid forking a multi-threaded application. See also `gc_thread`.

trim_stacks

Release stack memory resources that are not in use at this moment, returning them to the operating system. 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 can be written as follows:

```
loop :-
        generator,
            trim_stacks,
            potentially_expensive_operation,
        stop_condition, !.
```

[127] Up to version 7.3.11, dynamic code was handled using *reference counts*.

The Prolog top-level loop is written this way, reclaiming memory resources after every user query. See also `trim_heap/0` and `thread_idle/2`.

set_prolog_stack(*+Stack, +KeyValue*)
Set a parameter for one of the Prolog runtime stacks. *Stack* is one of `local`, `global` or `trail`. The table below describes the *Key*(*Value*) pairs.

Current settings can be retrieved with `prolog_stack_property/2`.

min_free(*+Cells*)
Minimum amount of free space after trimming or shifting the stack. Setting this value higher can reduce the number of garbage collections and stack-shifts at the cost of higher memory usage. The amount is reported and specified in *cells*. A cell is 4 bytes in the 32-bit version and 8 bytes on the 64-bit version. See `address_bits`. See also `trim_stacks/0` and `debug/0`.

low(*+Cells*)
factor(*+Number*)
These two figures determine whether, if the stacks are low, a stack *shift* (expansion) or garbage collection is performed. This depends on these two parameters, the current stack usage and the amount of stack used after the last garbage collection. A garbage collection is started if $used > factor \times lastused + low$.

spare(*+Cells*)
All stacks trigger overflow before actually reaching the limit, so the resulting error can be handled gracefully. The spare stack is used for `print_message/2` from the garbage collector and for handling exceptions. The default suffices, unless the user redefines related hooks. Do **not** specify large values for this because it reduces the amount of memory available for your real task.

Related hooks are `message_hook/3` (redefining GC messages), `prolog_trace_interception/4` and `prolog_exception_hook/4`.

prolog_stack_property(*?Stack, ?KeyValue*)
True if *KeyValue* is a current property of *Stack*. See `set_prolog_stack/2` for defined properties.

The total space limit for all stacks is controlled using the prolog flag `stack_limit`.

4.43.2 Heap memory (malloc)

SWI-Prolog's memory management is based on the C runtime malloc() function and related functions. The characteristics of the malloc() implementation may affect performance and overall memory usage of the system. For most Prolog programs the performance impact of the allocator is small.[128] The impact on total memory usage can be significant though, in particular for multi-threaded applications. This is due to two aspects of SWI-Prolog memory management:

- The Prolog stacks are allocated using malloc(). The stacks can be extremely large. SWI-Prolog assumes malloc() will use a mechanism that allows returning this memory to the OS. Most todays allocators satisfy this requirement.

[128] Multi-threaded applications may suffer from allocators that do not effectively avoid *false sharing* that affect CPU cache behaviour or operate using a single lock to provide thread safety. Such allocators should be rare in modern OSes.

- Atoms and clauses are allocated by the thread that requires them, but this memory is freed by the thread running the atom or clause garbage collector (see `garbage_collect_atoms/0` and `garbage_collect_clauses/0`). Normally these run in the thread `gc`, which means that all deallocation happens in this thread. Notably the ptmalloc implementation used by the GNU C library (glibc) seems to handle this poorly.

Starting with version 8.1.27, SWI-Prolog by default links against tcmalloc when available. Note that changing the allocator can only be done by linking the main executable (`swipl`) to an alternative library. When embedded (see section 12.4.23) the main program that embeds `libswipl` must be linked with tcmalloc. On ELF based systems (Linux), this effect can also be achieved using the environment variable `LD_PRELOAD`:

```
% LD_PRELOAD=/path/to/libtcmalloc.so swipl ...
```

SWI-Prolog attempts to detect the currently active allocator and sets the Prolog flag `malloc` if the detection succeeds. regardless of the malloc implementation, `trim_heap/0` is provided.

trim_heap *[det]*

his predicate attempts to return heap memory to the operating system. There is no portable way of doing so. If the system detects tcmalloc it calls MallocExtension_ReleaseFreeMemory(). If the system detects ptmalloc as provided by the GNU runtime library it calls malloc_trim(). In other cases this predicate simply succeeds. See also `trim_stacks/0`

TCMalloc control predicates

If SWI-Prolog core detects that tcmalloc is the current allocator and provides the following additional predicates.

malloc_property(*?Property*) *[nondet]*

True when *Property* is a property of the current allocator. The properties are defined by the allocator. The properties of tcmalloc are defined in `gperftools/malloc_extension.h`:[129]

'generic.current_allocated_bytes'(*-Int*)
: Number of bytes currently allocated by application.

'generic.heap_size'(*-Int*)
: Number of bytes in the heap (= current_allocated_bytes + fragmentation + freed memory regions).

'tcmalloc.max_total_thread_cache_bytes'(*-Int*)
: Upper limit on total number of bytes stored across all thread caches.

'tcmalloc.current_total_thread_cache_bytes'(*-Int*)
: Number of bytes used across all thread caches.

'tcmalloc.central_cache_free_bytes'(*-Int*)
: Number of free bytes in the central cache that have been assigned to size classes. They always count towards virtual memory usage, and unless the underlying memory is swapped out by the OS, they also count towards physical memory usage.

[129] Documentation copied from the header.

'tcmalloc.transfer_cache_free_bytes'(*-Int*)
Number of free bytes that are waiting to be transferred between the central cache and a thread cache. They always count towards virtual memory usage, and unless the underlying memory is swapped out by the OS, they also count towards physical

'tcmalloc.thread_cache_free_bytes'(*-Int*)
Number of free bytes in thread caches. They always count towards virtual memory usage, and unless the underlying memory is swapped out by the OS, they also count towards physical memory usage.

'tcmalloc.pageheap_free_bytes'(*-Int*)
Number of bytes in free, mapped pages in page heap. These bytes can be used to fulfill allocation requests. They always count towards virtual memory usage, and unless the underlying memory is swapped out by the OS, they also count towards physical memory usage. This property is not writable.

'tcmalloc.pageheap_unmapped_bytes'(*-Int*)
Number of bytes in free, unmapped pages in page heap. These are bytes that have been released back to the OS, possibly by one of the MallocExtension "Release" calls. They can be used to fulfill allocation requests, but typically incur a page fault. They always count towards virtual memory usage, and depending on the OS, typically do not count towards physical memory usage.

set_malloc(*+Property*) *[det]*
Set properties described in `malloc_property/1`. Currently the only writable property is `tcmalloc.max_total_thread_cache_bytes`. Setting an unknown property raises a `domain_error` and setting a read-only property raises a `permission_error` exception.

thread_idle(*:Goal, +Duration*) *[semidet]*
Indicates to the system that the calling thread will idle for some time while calling *Goal* as `once/1`. This call releases resources to the OS to minimise the footprint of the calling thread while it waits. Despite the name this predicate is always provided, also if the system is not configured with tcmalloc or is single threaded. *Duration* is one of

short
Calls `trim_stacks/0` and, if tcmalloc is used, calls MallocExtension_MarkThreadTemporarilyIdle() which empties the thread's malloc cache but preserves the cache itself.

long
Calls `garbage_collect/0` and `trim_stacks/0` and, if tcmalloc is used, calls MallocExtension_MarkThreadIdle() which releases all thread-specific allocation data structures.

4.44 Windows DDE interface

The predicates in this section deal with MS-Windows 'Dynamic Data Exchange' or DDE protocol.[130] A Windows DDE conversation is a form of interprocess communication based on sending reserved window events between the communicating processes.

[130] This interface is contributed by Don Dwiggins.

Failing DDE operations raise an error of the structure below, where *Operation* is the name of the (partial) operation that failed and *Message* is a translation of the operator error code. For some errors, *Context* provides additional comments.

```
        error(dde_error(Operation, Message), Context)
```

4.44.1 DDE client interface

The DDE client interface allows Prolog to talk to DDE server programs. We will demonstrate the use of the DDE interface using the Windows PROGMAN (Program Manager) application:

```
1 ?- open_dde_conversation(progman, progman, C).

C = 0
2 ?- dde_request(0, groups, X)

--> Unifies X with description of groups

3 ?- dde_execute(0, '[CreateGroup("DDE Demo")]').
true.

4 ?- close_dde_conversation(0).
true.
```

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.

dde_execute(+*Handle*, +*Command***)**
Request the DDE server to execute the given command string. Succeeds if the command could be executed and fails with an 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`.

4.44.2 DDE server mode

The `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 as 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.[131]

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.

[131] 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.

dde_current_service(*-Service, -Topic*)
Find currently registered services and the topics served on them.

dde_current_connection(*-Service, -Topic*)
Find currently open conversations.

4.45 Miscellaneous

dwim_match(*+Atom1, +Atom2*)
True if *Atom1* matches *Atom2* in the 'Do What I Mean' sense. Both *Atom1* and *Atom2* may also be integers or floats. The two atoms match if:

- They are identical
- They differ by one character (spy ≡ spu)
- One character is inserted/deleted (debug ≡ deug)
- Two characters are transposed (trace ≡ tarce)
- 'Sub-words' are glued differently (existsfile ≡ existsFile ≡ exists_file)
- Two adjacent sub-words are transposed (existsFile ≡ fileExists)

dwim_match(*+Atom1, +Atom2, -Difference*)
Equivalent to `dwim_match/2`, but unifies *Difference* with an atom identifying the 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*)
wildcard_match(*+Pattern, +String, +Options*)
True if *String* matches the wildcard pattern *Pattern*. *Pattern* is very similar to 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%').
true.
```

The `wildcard_match/3` version processes the following option:

case_sensitive(*+Boolean*)
When `false` (default `true`), match case insensitively.

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.

SWI-Prolog extensions 5

This chapter describes extensions to the Prolog language introduced with SWI-Prolog version 7 in 2014. The changes bring more modern syntactical conventions to Prolog such as key-value maps, called *dicts*, as primary citizens and a restricted form of *functional notation*. They also extend Prolog basic types with strings, providing a natural notation to textual material as opposed to identifiers (atoms) and lists.

These extensions make the syntax more intuitive to new users, simplify the integration of domain specific languages (DSLs) and facilitate a more natural Prolog representation for popular exchange languages such as XML and JSON.

While many programs run unmodified in SWI-Prolog version 7, some require modifications, especially those that pass double quoted strings to general purpose list processing predicates. See section 5.2.4 and section 5.2.5 for information and tools on porting. We provide a tool (`list_strings/0`) that we used to port a huge code base in half a day.

5.1 Lists are special

As of version 7, SWI-Prolog lists can be distinguished unambiguously at runtime from `./2` terms and the atom `'[]'`.

```
  Traditional list                    SWI-Prolog 7 list

     '.'                                  '[|]'
    /   \                                 /   \
   1     '.'                             1    '[|]'
        /   \                                 /   \
       2     '.'                             2    '[|]'
            /   \                                 /   \
           3   '[]'                              3      []

           terminated with                       terminated with
           the atom '[]',                        a special constant
           indistinguishable from text           which is printed as []
```

The constant `[]` is special constant that is not an atom. It has the following properties:

```
atom([]).          fails
atomic([]).        succeeds
[] == '[]'.        fails
[] == [].          succeeds
```

The 'cons' operator for creating *list cells* has changed from the pretty atom '`.`' to the ugly atom '`[|]`', so we can use the '`.`' for other purposes, notably functional notation on *dicts*. See section 5.4.1.

This modification has minimal impact on typical Prolog code. It does affect foreign code (see section 12) that uses the normal atom and compound term interface for manipulating lists. In most cases this can be avoided by using the dedicated list functions. For convenience, the macros `ATOM_nil` and `ATOM_dot` are provided by `SWI-Prolog.h`.

Another place that is affected is `write_canonical/1`. Impact is minimized by using the list syntax for lists. The predicates `read_term/2` and `write_term/2` support the option `dotlists`(*true*), which causes `read_term/2` to read `.(a,[])` as `[a]` and `write_term/2` to write `[a]` as `.(a,[])`.

5.1.1 Motivating '`[|]`' and `[]` for lists

Representing lists the conventional way using `./2` as list cell and the atom `'[]'` as list terminator both (independently) pose conflicts, while these conflicts are easily avoided.

- Using `./2` prevents using this commonly used symbol as an operator because `a.B` cannot be distinguished from `[a|B]`. Freeing `./2` provides us with a unique term that we can use for functional notation on dicts as described in section 5.4.1.

- Using the atom `'[]'` as list terminator prevents dynamic distinction between atoms and the empty list. As a result, we cannot use type polymorphism that involve both atoms and lists. For example, we cannot use *multi lists* (arbitrary deeply nested lists) of atoms. Multi lists of atoms are in some situations a good representation of a flat list that is assembled from sub sequences. The alternative, using difference lists or DCGs, is often less natural and sometimes requires 'opening' proper lists (i.e., copying the list while replacing the terminating atom `'[]'` with a variable) that have to be added to the sequence. The ambiguity of atom and list is particularly painful when mapping external data representations that do not suffer from this ambiguity.

  At the same time, avoiding atom `'[]'` as a list terminator makes the various text representations unambiguous, which allows us to write predicates that require a textual argument to accept any of atoms, strings, lists of character codes or characters. Traditionally, the empty list, as an atom, is afflicted with an ambiguous interpretation as it can stand for any of the strings `"[]"` and `""`.

5.2 The string type and its double quoted syntax

As of SWI-Prolog version 7, text enclosed in double quotes (e.g., `"Hello world"`) is read as objects of the type *string*. Strings are distinct from lists, which makes it possible to recognize them at runtime and print them using the string syntax:

```
?- write("Hello world!").
Hello world!

?- writeq("Hello world!").
"Hello world!"
```

A string is a compact representation of a character sequence that lives on the global (term) stack. Strings are represented by sequences of Unicode character codes including the character code 0

Mode	double_quotes	back_quotes
Version 7 default	string	codes
--traditional	codes	symbol_char

Table 5.1: Mapping of double and back quoted text in the two modes.

(zero). The length of strings is limited by the available space on the global (term) stack (see `set_prolog_stack/2`). Section 5.2.3 motivates the introduction of strings and mapping double quoted text to this type.

Whereas in version 7, double-quoted text is mapped to strings, *back-quoted* text (as in `` `text` ``) is mapped to a list of *character codes*, i.e. integers that are Unicode code points. In a traditional setting, back-quoted would be mapped to a list of *characters* (also known as *chars*), which are atoms of length 1.

The settings for the flags that control how double- and back-quoted text is read is summarised in table 5.1. Programs that aim for compatibility should realise that the ISO standard defines back-quoted text, but does not define the `back_quotes` Prolog flag and does not define the term that is produced by back-quoted text.

5.2.1 Representing text: strings, atoms and code lists

With the introduction of strings as a Prolog data type, there are three main ways to represent text: using strings, using atoms and using lists of character codes. As a fourth way, one may also use lists of chars. This section explains what to choose for what purpose. Both strings and atoms are *atomic* objects: you can only look inside them using dedicated predicates, while lists of character codes or chars are compound data structures forming an extended structure that must follow a convention.

Lists of character codes is what you need if you want to *parse* text using Prolog grammar rules (DCGs, see `phrase/3`). Most of the text reading predicates (e.g., `read_line_to_codes/2`) return a list of character codes because most applications need to parse these lines before the data can be processed. As said above, the *back-quoted text* notation (`` `hello` ``) can be used to easily specify a list of character codes. The `0'c` notation can be used to specify a single character code.

Atoms are *identifiers*. They are typically used in cases where identity comparison is the main operation and that are typically not composed nor taken apart. Examples are RDF resources (URIs that identify something), system identifiers (e.g., `'Boeing 747'`), but also individual words in a natural language processing system. They are also used where other languages would use *enumerated types*, such as the names of days in the week. Unlike enumerated types, Prolog atoms do not form a fixed set and the same atom can represent different things in different contexts.

Strings typically represents text that is processed as a unit most of the time, but which is not an identifier for something. Format specifications for `format/3` is a good example. Another example is a descriptive text provided in an application. Strings may be composed and decomposed using e.g., `string_concat/3` and `sub_string/5` or converted for parsing using `string_codes/2` or created from codes generated by a generative grammar rule, also using `string_codes/2`.

5.2.2 Predicates that operate on strings

Strings are manipulated using a set of predicates that mirrors the set of predicates used for manipulating atoms. In addition to the list below, `string/1` performs the type check for this type and is described in section 4.5.

SWI-Prolog's string primitives are being synchronized with ECLiPSe. We expect the set of predicates documented in this section to be stable, although it might be expanded. In general, SWI-Prolog's text manipulation predicates accept any form of text as input argument - they accept *anytext* input. *anytext* comprises:

- atoms
- strings
- lists of *character codes*
- list of *characters*
- number types: integers, floating point numbers and non-integer rationals. Under the hood, these must first be formatted into a text representation according to some inner convention before they can be used.

The predicates produce the type indicated by the predicate name as output. This policy simplifies migration and writing programs that can run unmodified or with minor modifications on systems that do not support strings. Code should avoid relying on this feature as much as possible for clarity as well as to facilitate a more strict mode and/or type checking in future releases.

atom_string(*?Atom, ?String*)
: Bi-directional conversion between an atom and a string. At least one of the two arguments must be instantiated. An initially uninstantiated variable on the "string side" is always instantiated to a string. An initially uninstantiated variable on the "atom side" is always instantiated to an atom. If both arguments are instantiated, their list-of-character representations must match, but the types are not enforced. The following all succeed:

```
atom_string("x",'x').
atom_string('x',"x").
atom_string(3.1415,3.1415).
atom_string('3r2',3r2).
atom_string(3r2,'3r2').
atom_string(6r4,3r2).
```

number_string(*?Number, ?String*)
: Bi-directional conversion between a number and a string. At least one of the two arguments must be instantiated. Besides the type used to represent the text, this predicate differs in several ways from its ISO cousin:[1]

  - If *String* does not represent a number, the predicate *fails* rather than throwing a syntax error exception.

[1]Note that SWI-Prolog's syntax for numbers is not ISO compatible either.

- Leading white space and Prolog comments are *not* allowed.
- Numbers may start with + or -.
- It is *not* allowed to have white space between a leading + or - and the number.
- Floating point numbers in exponential notation do not require a dot before exponent, i.e., "1e10" is a valid number.

Unlike other predicates of this family, if instantiated, *String* cannot be an atom.

The corresponding 'atom-handling' predicate is atom_number/2, with reversed argument order.

term_string(*?Term, ?String*)
Bi-directional conversion between a term and a string. If *String* is instantiated, it is parsed and the result is unified with *Term*. Otherwise *Term* is 'written' using the option quoted(*true*) and the result is converted to *String*.

term_string(*?Term, ?String, +Options*)
As term_string/2, passing *Options* to either read_term/2 or write_term/2. For example:

```
?- term_string(Term, 'a(A)', [variable_names(VNames)]).
Term = a(_9674),
VNames = ['A'=_9674].
```

string_chars(*?String, ?Chars*)
Bi-directional conversion between a string and a list of characters. At least one of the two arguments must be instantiated.

See also: atom_chars/2.

string_codes(*?String, ?Codes*)
Bi-directional conversion between a string and a list of character codes. At least one of the two arguments must be instantiated.

text_to_string(*+Text, -String*) *[det]*
Converts *Text* to a string. *Text* is *anytext* excluding the number types. When running in --traditional mode, '[]' is ambiguous and interpreted as an empty string.

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 *anytext* as its first argument. Number types must first be formatted into strings before the length of their string representation can be determined.

string_code(*?Index, +String, ?Code*)
True when *Code* represents the character at the 1-based *Index* position in *String*. If *Index* is unbound the string is scanned from index 1. Raises a domain error if *Index* is negative. Fails silently if *Index* is zero or greater than the length of *String*. The mode string_code(-,+,+) is deterministic if the searched-for *Code* appears only once in *String*. See also sub_string/5.

get_string_code(*+Index, +String, -Code*)
Semi-deterministic version of `string_code/3`. In addition, this version provides strict range checking, throwing a domain error if *Index* is less than 1 or greater than the length of *String*. ECLiPSe provides this to support `String[Index]` notation.

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.

split_string(*+String, +SepChars, +PadChars, -SubStrings*) *[det]*
Break *String* into *SubStrings*. The *SepChars* argument provides the characters that act as separators and thus the length of *SubStrings* is one more than the number of separators found if *SepChars* and *PadChars* do not have common characters. If *SepChars* and *PadChars* are equal, sequences of adjacent separators act as a single separator. Leading and trailing characters for each substring that appear in *PadChars* are removed from the substring. The input arguments can be either atoms, strings or char/code lists. Compatible with ECLiPSe. Below are some examples:

A simple split wherever there is a '.':

```
?- split_string("a.b.c.d", ".", "", L).
L = ["a", "b", "c", "d"].
```

Consider sequences of separators as a single one:

```
?- split_string("/home//jan///nice/path", "/", "/", L).
L = ["home", "jan", "nice", "path"].
```

Split and remove white space:

```
?- split_string("SWI-Prolog, 7.0", ",", " ", L).
L = ["SWI-Prolog", "7.0"].
```

Only remove leading and trailing white space (*trim* the string):

```
?- split_string("  SWI-Prolog  ", "", "\s\t\n", L).
L = ["SWI-Prolog"].
```

In the typical use cases, *SepChars* either does not overlap *PadChars* or is equivalent to handle multiple adjacent separators as a single (often white space). The behaviour with partially overlapping sets of padding and separators should be considered undefined. See also `read_string/5`.

sub_string(*+String, ?Before, ?Length, ?After, ?SubString*)
This predicate is functionally equivalent to `sub_atom/5`, but operates on strings. Note that this implies the string *input* arguments can be either strings or atoms. If *SubString* is unbound (output) it is unified with a string. The following example splits a string of the form ⟨*name*⟩=⟨*value*⟩ into the name part (an atom) and the value (a string).

```
name_value(String, Name, Value) :-
    sub_string(String, Before, _, After, "="),
    !,
    sub_atom(String, 0, Before, _, Name),
    sub_string(String, _, After, 0, Value).
```

atomics_to_string(*+List, -String*)
List is a list of strings, atoms, or number types. Succeeds if *String* can be unified with the concatenated elements of *List*. Equivalent to `atomics_to_string`(*List, '', String*).

atomics_to_string(*+List, +Separator, -String*)
Creates a string just like `atomics_to_string/2`, but inserts *Separator* between each pair of inputs. For example:

```
?- atomics_to_string([gnu, "gnat", 1], ', ', A).
A = "gnu, gnat, 1"
```

string_upper(*+String, -UpperCase*)
Convert *String* to upper case and unify the result with *UpperCase*.

string_lower(*+String, LowerCase*)
Convert *String* to lower case and unify the result with *LowerCase*.

read_string(*+Stream, ?Length, -String*)
Read at most *Length* characters from *Stream* and return them in the string *String*. If *Length* is unbound, *Stream* is read to the end and *Length* is unified with the number of characters read. Note that *characters* must be read as *Unicode code points*, *not* bytes.

read_string(*+Stream, +SepChars, +PadChars, -Sep, -String*)
Read a string from *Stream*, providing functionality similar to `split_string/4`. The predicate performs the following steps:

1. Skip all characters that match *PadChars*
2. Read up to a character that matches *SepChars* or end of file
3. Discard trailing characters that match *PadChars* from the collected input
4. Unify *String* with a string created from the input and *Sep* with the code of the separator character read. If input was terminated by the end of the input, *Sep* is unified with -1.

The predicate `read_string/5` called repeatedly on an input until *Sep* is -1 (end of file) is equivalent to reading the entire file into a string and calling `split_string/4`, provided that *SepChars* and *PadChars* are not *partially overlapping*.[2] Below are some examples:

Read a line:

```
read_string(Input, "\n", "\r", Sep, String)
```

Read a line, stripping leading and trailing white space:

```
read_string(Input, "\n", "\r\t ", Sep, String)
```

Read up to '`,`' or '`)`', unifying *Sep* with `0',` i.e. Unicode 44, or `0')`, i.e. Unicode 41:

```
read_string(Input, ",)", "\t ", Sep, String)
```

open_string(*+String, -Stream*)
True when *Stream* is an input stream that accesses the content of *String*. *String* can be any text representation, i.e., string, atom, list of codes or list of characters. The created *Stream* has the `reposition` property (see `stream_property/2`). Note that the internal encoding of the data is either ISO Latin 1 or UTF-8.

5.2.3 Why has the representation of double quoted text changed?

Prolog defines two forms of quoted text. Traditionally, single quoted text is mapped to atoms while double quoted text is mapped to a list of *character codes* (integers) or characters (atoms of length 1). Representing text using atoms is often considered inadequate for several reasons:

- It hides the conceptual difference between text and program symbols. Where content of text often matters because it is used in I/O, program symbols are merely identifiers that match with the same symbol elsewhere. Program symbols can often be consistently replaced, for example to obfuscate or compact a program.

- Atoms are globally unique identifiers. They are stored in a shared table. Volatile strings represented as atoms come at a significant price due to the required cooperation between threads for creating atoms. Reclaiming temporary atoms using *Atom garbage collection* is a costly process that requires significant synchronisation.

- Many Prolog systems (not SWI-Prolog) put severe restrictions on the length of atoms or the maximum number of atoms.

Representing text as lists, be it of character codes or characters, also comes at a price:

- It is not possible to distinguish (at runtime) a list of integers or atoms from a string. Sometimes this information can be derived from (implicit) typing. In other cases the list must be embedded

[2]Behaviour that is fully compatible would require unlimited look-ahead.

in a compound term to distinguish the two types. For example, `s("hello world")` could be used to indicate that we are dealing with a string.

Lacking runtime information, debuggers and the toplevel can only use heuristics to decide whether to print a list of integers as such or as a string (see `portray_text/1`).

While experienced Prolog programmers have learned to cope with this, we still consider this an unfortunate situation.

- Lists are expensive structures, taking 2 cells per character (3 for SWI-Prolog in its current form). This stresses memory consumption on the stacks while pushing them on the stack and dealing with them during garbage collection is unnecessarily expensive.

5.2.4 Adapting code for double quoted strings

We observe that in many programs, most strings are only handled as a single unit during their lifetime. Examining real code tells us that double quoted strings typically appear in one of the following roles:

A DCG literal Although represented as a list of codes is the correct representation for handling in DCGs, the DCG translator can recognise the literal and convert it to the proper representation. Such code need not be modified.

A format string This is a typical example of text that is conceptually not a program identifier. Format is designed to deal with alternative representations of the format string. Such code need not be modified.

Getting a character code The construct `[X] = "a"` is a commonly used template for getting the character code of the letter 'a'. ISO Prolog defines the syntax `0'a` for this purpose. Code using this must be modified. The modified code will run on any ISO compliant Prolog Processor.

As argument to list predicates to operate on strings Here, we might see code similar to `append("name:", Rest, Codes)`. Such code needs to be modified. In this particular example, the following is a good portable alternative: `phrase("name:", Codes, Rest)`

Checks for a character to be in a set Such tests are often performed with code such as this: `memberchk(C, "~!@#$")`. This is a rather inefficient check in a traditional Prolog system because it pushes a list of character codes cell-by-cell onto the Prolog stack and then traverses this list cell-by-cell to see whether one of the cells unifies with *C*. If the test is successful, the string will eventually be subject to garbage collection. The best code for this is to write a predicate as below, which pushes nothing on the stack and performs an indexed lookup to see whether the character code is in 'my_class'.

```
my_class(0'~).
my_class(0'!).
...
```

An alternative to reach the same effect is to use term expansion to create the clauses:

```
term_expansion(my_class(_), Clauses) :-
        findall(my_class(C),
                string_code(_, "~!@#$", C),
                Clauses).

my_class(_).
```

Finally, the predicate `string_code/3` can be exploited directly as a replacement for the `memberchk/2` on a list of codes. Although the string is still pushed onto the stack, it is more compact and only a single entity.

5.2.5 Predicates to support adapting code for double quoted strings

The predicates in this section can help adapting your program to the new convention for handling double quoted strings. We have adapted a huge code base with which we were not familiar in about half a day.

list_strings
: This predicate may be used to assess compatibility issues due to the representation of double quoted text as string objects. See section 5.2 and section 5.2.3. To use it, load your program into Prolog and run `list_strings/0`. The predicate lists source locations of string objects encountered in the program that are not considered safe. Such string need to be examined manually, after which one of the actions below may be appropriate:

- Rewrite the code. For example, change `[X] = "a"` into `X = 0'a`.
- If a particular module relies heavily on representing strings as lists of character code, consider adding the following directive to the module. Note that this flag only applies to the module in which it appears.

```
:- set_prolog_flag(double_quotes, codes).
```

- Use a back quoted string (e.g., `` `text` ``). Note that this will not make your code run regardless of the `--traditional` command line option and code exploiting this mapping is also not portable to ISO compliant systems.
- If the strings appear in facts and usage is safe, add a clause to the multifile predicate `check:string_predicate/1` to silence `list_strings/0` on all clauses of that predicate.
- If the strings appear as an argument to a predicate that can handle string objects, add a clause to the multifile predicate `check:valid_string_goal/1` to silence `list_strings/0`.

check:string_predicate(*:PredicateIndicator*)
: Declare that *PredicateIndicator* has clauses that contain strings, but that this is safe. For example, if there is a predicate `help_info/2`, where the second argument contains a double quoted string that is handled properly by the predicates of the applications' help system, add the following declaration to stop `list_strings/0` from complaining:

```
:- multifile check:string_predicate/1.

check:string_predicate(user:help_info/2).
```

check:valid_string_goal(*:Goal***)**

Declare that calls to *Goal* are safe. The module qualification is the actual module in which *Goal* is defined. For example, a call to `format/3` is resolved by the predicate `system:format/3`. and the code below specifies that the second argument may be a string (system predicates that accept strings are defined in the library).

```
:- multifile check:valid_string_goal/1.

check:valid_string_goal(system:format(_,S,_)) :- string(S).
```

5.3 Syntax changes since SWI-Prolog 7

5.3.1 Operators and quoted atoms

As of SWI-Prolog version 7, quoted atoms lose their operator property. This means that expressions such as `A = 'dynamic'/1` are valid syntax, regardless of the operator definitions. From questions on the mailinglist this is what people expect.[3] To accommodate for real quoted operators, a quoted atom that *needs* quotes can still act as an operator.[4] A good use-case for this is a unit library[5], which allows for expressions such as below.

```
?- Y isu 600kcal - 1h*200'W'.
Y = 1790400.0'J'.
```

5.3.2 Compound terms with zero arguments

As of SWI-Prolog version 7, the system supports compound terms that have no arguments. This implies that e.g., `name()` is valid syntax. This extension aims at functions on dicts (see section 5.4) as well as the implementation of domain specific languages (DSLs). To minimise the consequences, the classic predicates `functor/3` and `=../2` have not been modified. The predicates `compound_name_arity/3` and `compound_name_arguments/3` have been added. These predicates operate only on compound terms and behave consistently for compounds with zero arguments. Code that *generalises* a term using the sequence below should generally be changed to use `compound_name_arity/3`.

[3]We believe that most users expect an operator declaration to define a new token, which would explain why the operator name is often quoted in the declaration, but not while the operator is used. We are afraid that allowing for this easily creates ambiguous syntax. Also, many development environments are based on tokenization. Having dynamic tokenization due to operator declarations would make it hard to support Prolog in such editors.

[4]Suggested by Joachim Schimpf.

[5]https://groups.google.com/d/msg/comp.lang.prolog/ozqdzI-gi_g/2G16GYLIS0IJ

```
    ...,
    functor(Specific, Name, Arity),
    functor(General, Name, Arity),
    ...,
```

Replacement of `=../2` by `compound_name_arguments/3` is typically needed to deal with code that follow the skeleton below.

```
    ...,
    Term0 =.. [Name|Args0],
    maplist(convert, Args0, Args),
    Term =.. [Name|Args],
    ...,
```

For predicates, goals and arithmetic functions (evaluable terms), ⟨*name*⟩ and ⟨*name*⟩() are *equivalent*. Below are some examples that illustrate this behaviour.

```
go() :- format('Hello world~n').

?- go().
Hello world

?- go.
Hello world

?- Pi is pi().
Pi = 3.141592653589793.

?- Pi is pi.
Pi = 3.141592653589793.
```

Note that the *canonical* representation of predicate heads and functions without arguments is an atom. Thus, `clause`(*go(), Body*) returns the clauses for `go/0`, but `clause`(*-Head, -Body, +Ref*) unifies *Head* with an atom if the clause specified by *Ref* is part of a predicate with zero arguments.

5.3.3 Block operators

Introducing curly bracket and array subscripting.[6] The symbols `[]` and `{}` may be declared as an operator, which has the following effect:

[ ]

This operator is typically declared as a low-priority `yf` postfix operator, which allows for `array[index]` notation. This syntax produces a term `[]([index],array)`.

[6]Introducing block operators was proposed by Jose Morales. It was discussed in the Prolog standardization mailing list, but there were too many conflicts with existing extensions (ECLiPSe and B-Prolog) and doubt about their need to reach an agreement. Increasing need to get to some solution resulted in what is documented in this section. These extensions are also implemented in recent versions of YAP.

{}
: This operator is typically declared as a low-priority `xf` postfix operator, which allows for `head(arg) { body }` notation. This syntax produces a term `{}({body},head(arg))`.

Below is an example that illustrates the representation of a typical 'curly bracket language' in Prolog.

```
?- op(100, xf, {}).
?- op(100, yf, []).
?- op(1100, yf, ;).

?- displayq(func(arg)
            { a[10] = 5;
              update();
            }).
{}({;(=([]([10],a),5),;(update()))},func(arg))
```

5.4 Dicts: structures with named arguments

SWI-Prolog version 7 introduces dicts as an abstract object with a concrete modern syntax and functional notation for accessing members and as well as access functions defined by the user. The syntax for a dict is illustrated below. *Tag* is either a variable or an atom. As with compound terms, there is **no** space between the tag and the opening brace. The keys are either atoms or small integers (up to `max_tagged_integer`). The values are arbitrary Prolog terms which are parsed using the same rules as used for arguments in compound terms.

Tag{Key1:Value1, Key2:Value2, ...}

A dict can *not* hold duplicate keys. The dict is transformed into an opaque internal representation that does *not* respect the order in which the key-value pairs appear in the input text. If a dict is written, the keys are written according to the standard order of terms (see section 4.6.1). Here are some examples, where the second example illustrates that the order is not maintained and the third illustrates an anonymous dict.

```
?- A = point{x:1, y:2}.
A = point{x:1, y:2}.

?- A = point{y:2, x:1}.
A = point{x:1, y:2}.

?- A = _{first_name:"Mel", last_name:"Smith"}.
A = _G1476{first_name:"Mel", last_name:"Smith"}.
```

Dicts can be unified following the standard symmetric Prolog unification rules. As dicts use an internal canonical form, the order in which the named keys are represented is not relevant. This behaviour is illustrated by the following example.

```
?- point{x:1, y:2} = Tag{y:2, x:X}.
Tag = point,
X = 1.
```

Note In the current implementation, two dicts unify only if they have the same set of keys and the tags and values associated with the keys unify. In future versions, the notion of unification between dicts could be modified such that two dicts unify if their tags and the values associated with *common* keys unify, turning both dicts into a new dict that has the union of the keys of the two original dicts.

5.4.1 Functions on dicts

The infix operator dot (`op`(*100, yfx, .*) is used to extract values and evaluate functions on dicts. Functions are recognised if they appear in the argument of a *goal* in the source text, possibly nested in a term. The keys act as field selector, which is illustrated in this example.

```
?- X = point{x:1,y:2}.x.
X = 1.

?- Pt = point{x:1,y:2}, write(Pt.y).
2
Pt = point{x:1,y:2}.

?- X = point{x:1,y:2}.C.
X = 1,
C = x ;
X = 2,
C = y.
```

The compiler translates a goal that contains `./2` terms in its arguments into a conjunction of calls to `./3` defined in the `system` module. Terms functor`.`2 that appears in the head are replaced with a variable and calls to `./3` are inserted at the start of the body. Below are two examples, where the first extracts the `x` key from a dict and the second extends a dict containing an address with the postal code, given a `find_postal_code/4` predicate.

```
dict_x(X, X.x).

add_postal_code(Dict, Dict.put(postal_code, Code)) :-
        find_postal_code(Dict.city,
                         Dict.street,
                         Dict.house_number,
                         Code).
```

Note that expansion of `./2` terms implies that such terms cannot be created by writing them explicitly in your source code. Such terms can still be created with `functor/3`, `=../2`, `compound_name_arity/3` and `compound_name_arguments/3`.[7]

.(*+Dict, +Function, -Result*)

This predicate is called to evaluate `./2` terms found in the arguments of a goal. This predicate evaluates the field extraction described above, raising an exception if *Function* is an atom (*key*) and *Dict* does not contain the requested key. If *Function* is a compound term, it checks for the predefined functions on dicts described in section 5.4.1 or executes a user defined function as described in section 5.4.1.

User defined functions on dicts

The tag of a dict associates the dict to a module. If the dot notation uses a compound term, this calls the goal below.

⟨*module*⟩:⟨*name*⟩(Arg1, ..., +Dict, -Value)

Functions are normal Prolog predicates. The dict infrastructure provides a more convenient syntax for representing the head of such predicates without worrying about the argument calling conventions. The code below defines a function `multiply`(*Times*) on a point that creates a new point by multiplying both coordinates. and `len`[8] to compute the length from the origin. The `.` and `:=` operators are used to abstract the location of the predicate arguments. It is allowed to define multiple a function with multiple clauses, providing overloading and non-determinism.

```
:- module(point, []).

M.multiply(F) := point{x:X, y:Y} :-
        X is M.x*F,
        Y is M.y*F.

M.len() := Len :-
        Len is sqrt(M.x**2 + M.y**2).
```

After these definitions, we can evaluate the following functions:

```
?- X = point{x:1, y:2}.multiply(2).
X = point{x:2, y:4}.

?- X = point{x:1, y:2}.multiply(2).len().
X = 4.47213595499958.
```

[7] Traditional code is unlikely to use `./2` terms because they were practically reserved for usage in lists. We do not provide a quoting mechanism as found in functional languages because it would only be needed to quote `./2` terms, such terms are rare and term manipulation provides an escape route.

[8] as `length` would result in a predicate `length/2`, this name cannot be used. This might change in future versions.

Predefined functions on dicts

Dicts currently define the following reserved functions:

get(*?KeyPath*)
Return the value associates with *KeyPath*. *KeyPath* is either a single key or a term `Key1/Key2/...`. Each key is either an atom, small integer or a variable. While `Dict.Key` throws an existence error, this function *fails* silently if a key does not exist in the target dict. See also `:<`/2, which can be used to test for existence and unify multiple key values from a dict. For example:

```
?- write(t{a:x}.get(a)).
x
?- write(t{a:x}.get(b)).
false.
?- write(t{a:t{b:x}}.get(a/b)).
x
```

put(*+New*)
Evaluates to a new dict where the key-values in *New* replace or extend the key-values in the original dict. See `put_dict/3`.

get(*?KeyPath, +Default*)
Same as `get/1`, but if no match is found the function evaluates to *Default*. If *KeyPath* contains variables possible choice points are respected and the function only evaluates to *Default* if the pattern has no matches.

put(*+KeyPath, +Value*)
Evaluates to a new dict where the *KeyPath-Value* replaces or extends the key-values in the original dict. *KeyPath* is either a key or a term *KeyPath/Key*,[9] replacing the value associated with *Key* in a sub-dict of the dict on which the function operates. See `put_dict/4`. Below are some examples:

```
?- A = _{}.put(a, 1).
A = _G7359{a:1}.

?- A = _{a:1}.put(a, 2).
A = _G7377{a:2}.

?- A = _{a:1}.put(b/c, 2).
A = _G1395{a:1, b:_G1584{c:2}}.

?- A = _{a:_{b:1}}.put(a/b, 2).
A = _G1429{a:_G1425{b:2}}.
```

[9] Note that we do not use the '.' functor here, because the `./2` would *evaluate*.

```
?- A = _{a:1}.put(a/b, 2).
A = _G1395{a:_G1578{b:2}}.
```

5.4.2 Predicates for managing dicts

This section documents the predicates that are defined on dicts. We use the naming and argument conventions of the traditional `assoc`.

is_dict(*@Term*)
True if *Term* is a dict. This is the same as `is_dict(Term,_)`.

is_dict(*@Term, -Tag*)
True if *Term* is a dict of *Tag*.

get_dict(*?Key, +Dict, -Value*)
Unify the value associated with *Key* in dict with *Value*. If *Key* is unbound, all associations in *Dict* are returned on backtracking. The order in which the associations are returned is undefined. This predicate is normally accessed using the functional notation `Dict.Key`. See section 5.4.1.

Fails silently if Key does not appear in Dict. This is different from the behavior of the functional '.'-notation, which throws an existence error in that case.

get_dict(*+Key, +Dict, -Value, -NewDict, +NewValue*) *[semidet]*
Create a new dict after updating the value for *Key*. Fails if *Value* does not unify with the current value associated with *Key*. *Dict* is either a dict or a list the can be converted into a dict.

Has the behavior as if defined in the following way:

```
get_dict(Key, Dict, Value, NewDict, NewValue) :-
        get_dict(Key, Dict, Value),
        put_dict(Key, Dict, NewValue, NewDict).
```

dict_create(*-Dict, +Tag, +Data*)
Create a dict in *Tag* from *Data*. *Data* is a list of attribute-value pairs using the syntax `Key:Value`, `Key=Value`, `Key-Value` or `Key(Value)`. An exception is raised if *Data* is not a proper list, one of the elements is not of the shape above, a key is neither an atom nor a small integer or there is a duplicate key.

dict_pairs(*?Dict, ?Tag, ?Pairs*)
Bi-directional mapping between a dict and an ordered list of pairs (see section A.30).

put_dict(*+New, +DictIn, -DictOut*)
DictOut is a new dict created by replacing or adding key-value pairs from *New* to *Dict*. *New* is either a dict or a valid input for `dict_create/3`. This predicate is normally accessed using the functional notation. Below are some examples:

```
?- A = point{x:1, y:2}.put(_{x:3}).
A = point{x:3, y:2}.

?- A = point{x:1, y:2}.put([x=3]).
A = point{x:3, y:2}.

?- A = point{x:1, y:2}.put([x=3,z=0]).
A = point{x:3, y:2, z:0}.
```

put_dict(*+Key, +DictIn, +Value, -DictOut*)

DictOut is a new dict created by replacing or adding *Key*-*Value* to *DictIn*. For example:

```
?- A = point{x:1, y:2}.put(x, 3).
A = point{x:3, y:2}.
```

This predicate can also be accessed by using the functional notation, in which case Key can also be a *path* of keys. For example:

```
?- Dict = _{}.put(a/b, c).
Dict = _6096{a:_6200{b:c}}.
```

del_dict(*+Key, +DictIn, ?Value, -DictOut*)

True when *Key*-*Value* is in *DictIn* and *DictOut* contains all associations of *DictIn* except for *Key*.

+Select **:<** *+From* *[semidet]*

True when *Select* is a 'sub dict' of *From*: the tags must unify and all keys in *Select* must appear with unifying values in *From*. *From* may contain keys that are not in *Select*. This operation is frequently used to *match* a dict and at the same time extract relevant values from it. For example:

```
plot(Dict, On) :-
        _{x:X, y:Y, z:Z} :< Dict, !,
        plot_xyz(X, Y, Z, On).
plot(Dict, On) :-
        _{x:X, y:Y} :< Dict, !,
        plot_xy(X, Y, On).
```

The goal `Select :< From` is equivalent to `select_dict(`*Select, From, _*`)`.

select_dict(*+Select, +From, -Rest*) *[semidet]*

True when the tags of *Select* and *From* have been unified, all keys in *Select* appear in *From* and the corresponding values have been unified. The key-value pairs of *From* that do not appear in *Select* are used to form an anonymous dict, which us unified with *Rest*. For example:

```
?- select_dict(P{x:0, y:Y}, point{x:0, y:1, z:2}, R).
P = point,
Y = 1,
R = _G1705{z:2}.
```

See also `:</2` to ignore *Rest* and `>:</2` for a symmetric partial unification of two dicts.

+Dict1 **>:<** *+Dict2*
This operator specifies a *partial unification* between *Dict1* and *Dict2*. It is true when the tags and the values associated with all *common* keys have been unified. The values associated to keys that do not appear in the other dict are ignored. Partial unification is symmetric. For example, given a list of dicts, find dicts that represent a point with X equal to zero:

```
    member(Dict, List),
    Dict >:< point{x:0, y:Y}.
```

See also `:</2` and `select_dict/3`.

Destructive assignment in dicts

This section describes the destructive update operations defined on dicts. These actions can only *update* keys and not add or remove keys. If the requested key does not exist the predicate raises `existence_error`(*key, Key, Dict*). Note the additional argument.

Destructive assignment is a non-logical operation and should be used with care because the system may copy or share identical Prolog terms at any time. Some of this behaviour can be avoided by adding an additional unbound value to the dict. This prevents unwanted sharing and ensures that `copy_term/2` actually copies the dict. This pitfall is demonstrated in the example below:

```
?- A = a{a:1}, copy_term(A,B), b_set_dict(a, A, 2).
A = B, B = a{a:2}.

?- A = a{a:1,dummy:_}, copy_term(A,B), b_set_dict(a, A, 2).
A = a{a:2, dummy:_G3195},
B = a{a:1, dummy:_G3391}.
```

b_set_dict(*+Key, !Dict, +Value*) *[det]*
Destructively update the value associated with *Key* in *Dict* to *Value*. The update is trailed and undone on backtracking. This predicate raises an existence error if *Key* does not appear in *Dict*. The update semantics are equivalent to `setarg/3` and `b_setval/2`.

nb_set_dict(*+Key, !Dict, +Value*) *[det]*
Destructively update the value associated with *Key* in *Dict* to a copy of *Value*. The update is *not* undone on backtracking. This predicate raises an existence error if *Key* does not appear in *Dict*. The update semantics are equivalent to `nb_setarg/3` and `nb_setval/2`.

nb_link_dict(*+Key, !Dict, +Value*) *[det]*
Destructively update the value associated with *Key* in *Dict* to *Value*. The update is *not* undone on backtracking. This predicate raises an existence error if *Key* does not appear in *Dict*. The update semantics are equivalent to `nb_linkarg/3` and `nb_linkval/2`. Use with extreme care and consult the documentation of `nb_linkval/2` before use.

5.4.3 When to use dicts?

Dicts are a new type in the Prolog world. They compete with several other types and libraries. In the list below we have a closer look at these relations. We will see that dicts are first of all a good replacement for compound terms with a high or not clearly fixed arity, library `record` and option processing.

Compound terms Compound terms with positional arguments form the traditional way to package data in Prolog. This representation is well understood, fast and compound terms are stored efficiently. Compound terms are still the representation of choice, provided that the number of arguments is low and fixed or compactness or performance are of utmost importance.

A good example of a compound term is the representation of RDF triples using the term `rdf`(*Subject, Predicate, Object*) because RDF triples are defined to have precisely these three arguments and they are always referred to in this order. An application processing information about persons should probably use dicts because the information that is related to a person is not so fixed. Typically we see first and last name. But there may also be title, middle name, gender, date of birth, etc. The number of arguments becomes unmanageable when using a compound term, while adding or removing an argument leads to many changes in the program.

Library `record` Using library `record` relieves the maintenance issues associated with using compound terms significantly. The library generates access and modification predicates for each field in a compound term from a declaration. The library provides sound access to compound terms with many arguments. One of its problems is the verbose syntax needed to access or modify fields which results from long names for the generated predicates and the restriction that each field needs to be extracted with a separate goal. Consider the example below, where the first uses library `record` and the second uses dicts.

```
...,
person_first_name(P, FirstName),
person_last_name(P, LastName),
format('Dear ~w ~w,~n~n', [FirstName, LastName]).

...,
format('Dear ~w ~w,~n~n', [Dict.first_name, Dict.last_name]).
```

Records have a fixed number of arguments and (non-)existence of an argument must be represented using a value that is outside the normal domain. This lead to unnatural code. For example, suppose our person also has a title. If we know the first name we use this and else we use the title. The code samples below illustrate this.

```
salutation(P) :-
    person_first_name(P, FirstName), nonvar(FirstName), !,
    person_last_name(P, LastName),
    format('Dear ~w ~w,~n~n', [FirstName, LastName]).
salutation(P) :-
    person_title(P, Title), nonvar(Title), !,
    person_last_name(P, LastName),
    format('Dear ~w ~w,~n~n', [Title, LastName]).

salutation(P) :-
    _{first_name:FirstName, last_name:LastName} :< P, !,
    format('Dear ~w ~w,~n~n', [FirstName, LastName]).
salutation(P) :-
    _{title:Title, last_name:LastName} :< P, !,
    format('Dear ~w ~w,~n~n', [Title, LastName]).
```

Library `assoc` This library implements a balanced binary tree. Dicts can replace the use of this library if the association is fairly static (i.e., there are few update operations), all keys are atoms or (small) integers and the code does not rely on ordered operations.

Library `option` Option lists are introduced by ISO Prolog, for example for `read_term/3`, `open/4`, etc. The `option` library provides operations to extract options, merge options lists, etc. Dicts are well suited to replace option lists because they are cheaper, can be processed faster and have a more natural syntax.

Library `pairs` This library is commonly used to process large name-value associations. In many cases this concerns short-lived data structures that result from `findall/3`, `maplist/3` and similar list processing predicates. Dicts may play a role if frequent random key lookups are needed on the resulting association. For example, the skeleton 'create a pairs list', 'use `list_to_assoc/2` to create an assoc', followed by frequent usage of `get_assoc/3` to extract key values can be replaced using `dict_pairs/3` and the dict access functions. Using dicts in this scenario is more efficient and provides a more pleasant access syntax.

5.4.4 A motivation for dicts as primary citizens

Dicts, or key-value associations, are a common data structure. A good old example are *property lists* as found in Lisp, while a good recent example is formed by JavaScript *objects*. Traditional Prolog does not offer native property lists. As a result, people are using a wide range of data structures for key-value associations:

- Using compound terms and positional arguments, e.g., `point(1,2)`.
- Using compound terms with library `record`, which generates access predicates for a term using positional arguments from a description.
- Using lists of terms `Name=Value`, `Name-Value`, `Name:Value` or `Name(Value)`.
- Using library `assoc` which represents the associations as a balanced binary tree.

This situation is unfortunate. Each of these have their advantages and disadvantages. E.g., compound terms are compact and fast, but inflexible and using positional arguments quickly breaks down. Library `record` fixes this, but the syntax is considered hard to use. Lists are flexible, but expensive and the alternative key-value representations that are used complicate the matter even more. Library `assoc` allows for efficient manipulation of changing associations, but the syntactical representation of an assoc is complex, which makes them unsuitable for e.g., *options lists* as seen in predicates such as `open/4`.

5.4.5 Implementation notes about dicts

Although dicts are designed as an abstract data type and we deliberately reserve the possibility to change the representation and even use multiple representations, this section describes the current implementation.

Dicts are currently represented as a compound term using the functor `'dict'`. The first argument is the tag. The remaining arguments create an array of sorted key-value pairs. This representation is compact and guarantees good locality. Lookup is order $\log N$, while adding values, deleting values and merging with other dicts has order N. The main disadvantage is that changing values in large dicts is costly, both in terms of memory and time.

Future versions may share keys in a separate structure or use a binary trees to allow for cheaper updates. One of the issues is that the representation must either be kept canonical or unification must be extended to compensate for alternate representations.

5.5 Integration of strings and dicts in the libraries

While lacking proper string support and dicts when designed, many predicates and libraries use interfaces that must be classified as suboptimal. Changing these interfaces is likely to break much more code than the changes described in this chapter. This section discusses some of these issues. Roughly, there are two cases. There where key-value associations or text is required as *input*, we can facilitate the new features by overloading the accepted types. Interfaces that produce text or key-value associations as their *output* however must make a choice. We plan to resolve that using either options that specify the desired output or provide an alternative library.

5.5.1 Dicts and option processing

System predicates and predicates based on library `options` process dicts as an alternative to traditional option lists.

5.5.2 Dicts in core data structures

Some predicates now produce structured data using compound terms and access predicates. We consider migrating these to dicts. Below is a tentative list of candidates. Portable code should use the provided access predicates and not rely on the term representation.

- Stream position terms
- Date and time records

5.5.3 Dicts, strings and XML

The XML representation could benefit significantly from the new features. In due time we plan to provide an set of alternative predicates and options to existing predicates that can be used to exploit the new types. We propose the following changes to the data representation:

- The attribute list of the `element`(*Name, Attributes, Content*) will become a dict.
- Attribute values will remain atoms
- CDATA in element content will be represented as strings

5.5.4 Dicts, strings and JSON

The JSON representation could benefit significantly from the new features. In due time we plan to provide an set of alternative predicates and options to existing predicates that can be used to exploit the new types. We propose the following changes to the data representation:

- Instead of using `json`(*KeyValueList*), the new interface will translate JSON objects to a dict. The type of this dict will be `json`.
- String values in JSON will be mapped to strings.
- The values `true`, `false` and `null` will be represented as atoms.

5.5.5 Dicts, strings and HTTP

The HTTP library and related data structures would profit from exploiting dicts. Below is a list of data structures that might be affected by future changes. Code can be made more robust by using the `option` library functions for extracting values from these structures.

- The HTTP request structure
- The HTTP parameter interface
- URI components
- Attributes to HTML elements

5.6 Single Sided Unification rules

For the execution of a normal Prolog clause, the goal term is unified with the head of the clause. This allows us to write facts such as below and use this relation in all four possible *modes*. This is the basis of SLD resolution that turns Prolog into a logic programming language.

```
parent('Bob', 'Susan').
```

In practice though, Prolog is both a logic programming language and a language for expressing computations in a near *procedural* style. The first is used to solve (notably) combinatorial problems while the latter is used for I/O, data transformation and the many non-logical operations that are involved in many applications.

Many Prolog programmers experience writing procedural style Prolog as fighting non-determinism and dealing with hard to debug silent failures because no clause matches some goal. Below are two typical queries on library predicates that have a procedural nature, i.e., are *single moded*.

```
?- sum_list(a, X).
false.

?- sum_list([1|T], X).
T = [],
X = 1 ;
ERROR: Arguments are not sufficiently instantiated
```

The definition of `sum_list/2` is it appears in library(lists) is below. This implementation can be considered elegant. Note that `sum_list/2` has only one meaningful mode: (+,-). A general (logical) implementation would allow for a partial list or a list holding one or more variables, With a proper list that holds a single variable we can still make a sound logical implementation. In all other cases the number of solutions is infinite and even *uncountable* for a partial list, making the predicate useless as a *generator* of solutions.

```
sum_list(Xs, Sum) :-
    sum_list(Xs, 0, Sum).

sum_list([], Sum, Sum).
sum_list([X|Xs], Sum0, Sum) :-
    Sum1 is Sum0 + X,
    sum_list(Xs, Sum1, Sum).
```

If we want to avoid the above dubious behaviour we have two options. First, we can verify that the first argument is a list before entering the recursion, changing the first clause as below. The disadvantage is that we process the list twice.

```
sum_list(Xs, Sum) :-
    must_be(list, Xs),
    sum_list(Xs, 0, Sum).
```

Alternatively, we can rewrite the second clause to verify the list on the fly. That leads to the code below. Most likely the overhead of this alternative compared to the above is even worse in many Prolog implementations. Most people would also consider this code rather inelegant.

```
sum_list(Var, _, _) :-
    var(Var),
```

```
    instantiation_error(Var).
sum_list([], Sum, Sum) :-
    !.
sum_list([X|Xs], Sum0, Sum) :-
    !,
    Sum1 is Sum0 + X,
    sum_list(Xs, Sum1, Sum).
sum_list(NoList, _, _) :-
    type_error(list, NoList).
```

Another example is a relation `max/3`, expressing the maximum of two numbers. A classical textbook definition could be as below. This code has two drawbacks. First it leaves an open choice points in most Prolog implementations if *X* is the largest and second it compares the two numbers twice. Some Prolog systems detect this particular case, but in general it needs two know that one test is the strict negation of the other.

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- Y > X.
```

As a result people use a cut and might come up with the **wrong** solution below. Consider the query `?- max(5,2,2).` to see why this code is broken.

```
max(X,Y,X) :- X >= Y, !.
max(_,Y,Y).
```

A correct solution is below, *delaying* binding the output until after the cut.

```
max(X,Y,M) :- X >= Y, !, M = X.
max(_,Y,Y).
```

Some people may prefer using if-then-else as below. This is arguable the cleanest efficient solution in standard Prolog.

```
max(X,Y,M) :- ( X >= Y -> M = X ; M = Y ).
```

As we have seen from these examples, writing procedural code in Prolog requires us to follow the two basic principles below. Both principles have been properly described in *The Craft of Prolog* [O'Keefe, 1990].

- Structure every clause as `Head :- Guard, !, Body`. Every clause has the cut as early as possible. *Guard* can be empty. The last clause often does not need a cut.
- Avoid that the head unification binds values in the goal term. We see this may lead to undesirable results such as sum_list(L,S) binding *L* to '[]' and *S* to '0' as well as loss of *steadfastness*, causing max(5,2,2) to succeed. The first requires additional `var/1` or `nonvar/1` tests. The second requires delaying unification until after the cut.

Picat provides the `=>/2` alternative for the Prolog *neck* (`:-/2`) to force the above practices. A Picat rule has the following shape:

```
Head, Guard => Body.
```

This is semantically equivalent to the Prolog clause below. The `subsumes_term/2` guarantees the clause head is more *generic* than the goal term and thus unifying the two does not affect any of the arguments of the goal. This implies all output unification _must_ be done after the head unification.

```
p(V1,V2,...,Vn) :-
    Pattern = p(A1,A2,...,An),
    Args = p(V1,V2,...,Vn),
    subsumes_term(Pattern, Args),
    Pattern = Args,
    Guard,
    !,
    Body.
```

SWI-Prolog as of version 8.3.19 support `=>/2` as an alternative to normal Prolog clauses. The construct comes with the following properties.

- A predicate either uses `:-/2` for all its clauses or `=>/2`. Mixing is **not** allowed and raises a permission error for a clause that does not use the same *neck* as the first clause.
- Unlike Picat, it is an error if no clause matches.

Given `=>/2` rules, we can rewrite `sum_list/2` as below. The first clause can be written using `:-/2` or `=>/2`. As the head is the most general head and there is only one clause these are fully equivalent. The `sum_list/3` helper needs a small modification: we need to delay the unification against *Sum* to the body. The last clause is equivalent.

```
sum_list(Xs, Sum) =>
    sum_list(Xs, 0, Sum).

sum_list([], Sum0, Sum) =>
    Sum = Sum0.
sum_list([X|Xs], Sum0, Sum) =>
    Sum1 is Sum0 + X,
    sum_list(Xs, Sum1, Sum).
```

Given this definition, `sum_list(L,S)` no longer matches a rule and neither does e.g., `sum_list(a,S)`. Both raise an error. Currently the error is defined as below.

```
existence_error(matching_rule, Head)
```

Should silent failure be desired if no rule matches, this is easily encoding by adding a rule at the end using the most general head and `fail/0` as body:

```
sum_list(_,_,_) => fail.
```

5.6.1 Single Sided Unification Guards

Using the construction `Head, Guard => Body`, the *Guard* is executed *after* the single sided head unification. If the *Guard* succeeds the clause executes a cut (`!/0`) and proceeds normally. There are no restrictions on the guard code. A well behaved guard is a *test*. Notably:

- Though not enforced[10], guard code shall not instantiate variables in the *Head* because this breaks the promise of SSU and may make the node non-steadfast.
- It is bad style (but again, not enforced) to have any type of side effects (output, database change, etc.)
- Typically, guard calls are `semidet`. Non-deterministic calls are allowed. If the guard succeeds with choicepoints these are pruned before the body is entered.

As a special exception, explicit unification against a variable in the head is moved into the head. See section 2.18.3. In the example below, the `X = f(I)` is moved into the head and (thus) is executed using single sided unification.

```
p(X), X = f(I), integer(I) => q(X).
```

> **Warning** Moving the guard unification into the head changes the semantics of the unification. This may be defended by the rules above that claim one should not unify against the head arguments in the guard. Future versions may use a dedicated operator to indicate that the unification may be moved into the head.

5.6.2 Consequenses of => single sided unification rules

The `=>/2` construct is handled by the low-level compiler if no *guard* is present. If a guard is present it is currently compiled into the construct below. The Picat `?=>/2` *neck* operator is like `=>/2`, but does not *commit* to this rule. We are not yet sure whether or not SWI-Prolog will remain supporting `?=>/2`.[11]

```
Head ?=> Guard, !, Body.
```

The main consequence is that `clause/2` cannot distinguish between a normal clause and a `=>/2` clause. In the current implementation it operates on both without distinguishing the two. This implies e.g., *cross referencing* still works. Meta interpretation however does not work. In future versions `clause/2` may fail on these rules. As an alternative we provide `rule/2`,3.

rule(*:Head, -Rule*)
rule(*:Head, -Rule, -Ref*)
: True when *Rule* is a rule/clause that implements *Head*. *Rule* is a complete rule term. For a normal clause this is a term `Head :- Body` and for a single sided unification rule it is a term `Head => Body`.

[10] We do not know about an efficient way to enforce unification against head arguments

[11] `?=>/2` is currently implemented but not defined as an operator.

5.6.3 SSU: Future considerations

The current implementation is a rather simple. Single sided unification is achieved doing normal head unification and backtrack if this unification bound variables in the goal term. Future versions are likely to backtrack as soon as we find a variable in the goal that needs to be unified.

It is likely that in due time significant parts of the libraries will be migrated to use SSU rules, turning many silent failures on type errors into errors.

5.7 Remaining issues

The changes and extensions described in this chapter resolve many limitations of the Prolog language we have encountered. Still, there are remaining issues for which we seek solutions in the future.

Text representation Although strings resolve this issue for many applications, we are still faced with the representation of text as lists of characters which we need for parsing using DCGs. The ISO standard provides two representations, a list of *character codes* ('codes' for short) and a list of *one-character atoms* ('chars' for short). There are two sets of predicates, named *_code(s) and *_char(s) that provide the same functionality (e.g., `atom_codes/2` and `atom_chars/2`) using their own representation of characters. Codes can be used in arithmetic expressions, while chars are more readable. Neither can unambiguously be interpreted as a representation for text because codes can be interpreted as a list of integers and chars as a list of atoms.

We have not found a convincing way out. One of the options could be the introduction of a 'char' type. This type can be allowed in arithmetic and with the 0'⟨*char*⟩ syntax we have a concrete syntax for it.

Arrays Although lists are generally a much cleaner alternative for Prolog, real arrays with direct access to elements can be useful for particular tasks. The problem of integrating arrays is twofold. First of all, there is no good one-size-fits-all data representation for arrays. Many tasks that involve arrays require *mutable* arrays, while Prolog data is immutable by design. Second, standard Prolog has no good syntax support for arrays. SWI-Prolog version 7 has 'block operators' (see section 5.3.3) which can resolve the syntactic issues. Block operators have been adopted by YAP.

Lambda expressions Although many alternatives[12] have been proposed, we still feel uneasy with them.

Loops Many people have explored routes to avoid the need for recursion in Prolog for simple iterations over data. ECLiPSe have proposed *logical loops* [Schimpf, 2002], while B-Prolog introduced *declarative loops* and *list comprehension* [Zhou, 2010]. The above mentioned lambda expressions, combined with `maplist/2` can achieve similar results.

[12] See e.g., `http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/ISO-Hiord`

6 Modules

A Prolog module is a collection of predicates which defines a public interface by means of a set of provided predicates and operators. Prolog modules are defined by an ISO standard. Unfortunately, the standard is considered a failure and, as far as we are aware, not implemented by any concrete Prolog implementation. The SWI-Prolog module system syntax is derived from the Quintus Prolog module system. The Quintus module system has been the starting point for the module systems of a number of mainstream Prolog systems, such as SICStus, Ciao and YAP. The underlying primitives of the SWI-Prolog module system differ from the mentioned systems. These primitives allow for multiple modules in a file, hierarchical modules, emulation of other modules interfaces, etc.

This chapter motivates and describes the SWI-Prolog module system. Novices can start using the module system after reading section 6.2 and section 6.3. The primitives defined in these sections suffice for basic usage until one needs to export predicates that call or manage other predicates dynamically (e.g., use `call/1`, `assert/1`, etc.). Such predicates are called *meta predicates* and are discussed in section 6.5. Section 6.6 to section 6.9 describe more advanced issues. Starting with section 6.10, we discuss more low-level aspects of the SWI-Prolog module system that are used to implement the visible module system, and can be used to build other code reuse mechanisms.

6.1 Why Use Modules?

In classic Prolog systems, all predicates are organised in a single namespace and any predicate can call any predicate. Because each predicate in a file can be called from anywhere in the program, it becomes very hard to find the dependencies and enhance the implementation of a predicate without risking to break the overall application. This is true for any language, but even worse for Prolog due to its frequent need for 'helper predicates'.

A Prolog module encapsulates a set of predicates and defines an *interface*. Modules can import other modules, which makes the dependencies explicit. Given explicit dependencies and a well-defined interface, it becomes much easier to change the internal organisation of a module without breaking the overall application.

Explicit dependencies can also be used by the development environment. The SWI-Prolog library `prolog_xref` can be used to analyse completeness and consistency of modules. This library is used by the built-in editor PceEmacs for syntax highlighting, jump-to-definition, etc.

6.2 Defining a Module

Modules are normally 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` and hiding the helper predicate `rev/3`. A module can use all built-in predicates and, by default, cannot redefine system predicates.

```
:- module(reverse, [reverse/2]).

reverse(List1, List2) :-
        rev(List1, [], List2).

rev([], List, List).
rev([Head|List1], List2, List3) :-
        rev(List1, [Head|List2], List3).
```

The module is named `reverse`. Typically, the name of a module is the same as the name of the file by which it is defined without the filename extension, but this naming is not enforced. Modules are organised in a single and flat namespace and therefore module names must be chosen with some care to avoid conflicts. As we will see, typical applications of the module system rarely use the name of a module explicitly in the source text.

:- 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 a module named *Module*. Note that a module name is an atom. The module exports the predicates of *PublicList*. *PublicList* is a list of predicate indicators (name/arity or 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 4.25.

Compatible to Ciao Prolog, if *Module* is unbound, it is unified with the basename without extension of the file being loaded.

:- module(+*Module, +PublicList, +Dialect*)

Same as `module/2`. The additional *Dialect* argument provides a list of *language options*. Each atom in the list *Dialect* is mapped to a `use_module/1` goal as given below. See also section C. The third argument is supported for compatibility with the Prolog Commons project.

```
:- use_module(library(dialect/LangOption)).
```

6.3 Importing Predicates into a Module

Predicates can be added to a module by *importing* them from another module. Importing adds predicates to the namespace of a module. An imported predicate can be called exactly the same as a locally defined predicate, although its implementation remains part of the module in which it has been defined.

Importing the predicates from another module is achieved using the directives `use_module/1` or `use_module/2`. Note that both directives take *filename(s)* as arguments. That is, modules are imported based on their filename rather than their module name.

use_module(*+Files*)

Load the file(s) specified with *Files* just like `ensure_loaded/1`. The files must all be module files. All exported predicates from the loaded files are imported into the module from which this predicate is called. This predicate is equivalent to `ensure_loaded/1`, except that it raises an error if *Files* are not module files.

The imported predicates act as *weak symbols* in the module into which they are imported. This implies that a local definition of a predicate overrides (clobbers) the imported definition. If the flag `warn_override_implicit_import` is `true` (default), a warning is printed. Below is an example of a module that uses library(lists), but redefines `flatten/2`, giving it a totally different meaning:

```
:- module(shapes, []).
:- use_module(library(lists)).

flatten(cube, square).
flatten(ball, circle).
```

Loading the above file prints the following message:

```
Warning: /home/janw/Bugs/Import/t.pl:5:
        Local definition of shapes:flatten/2
        overrides weak import from lists
```

This warning can be avoided by (1) using `use_module/2` to only import the predicates from the `lists` library that are actually used in the 'shapes' module, (2) using the `except([flatten/2])` option of `use_module/2`, (3) use `:- abolish(flatten/2).` before the local definition or (4) setting `warn_override_implicit_import` to `false`. Globally disabling this warning is only recommended if overriding imported predicates is common as a result of design choices or the program is ported from a system that silently overrides imported predicates.

Note that it is always an error to import two modules with `use_module/1` that export the same predicate. Such conflicts must be resolved with `use_module/2` as described above.

use_module(*+File, +ImportList*)

Load *File*, which must be a module file, and import the predicates as specified by *ImportList*. *ImportList* is a list of predicate indicators specifying the predicates that will be imported from the loaded module. *ImportList* also allows for renaming or import-everything-except. See also the `import` option of `load_files/2`. The first example below loads `member/2` from the `lists` library and `append/2` under the name `list_concat`, which is how this predicate is named in YAP. The second example loads all exports from library `option` except for `meta_options/3`. These renaming facilities are generally used to deal with portability issues with as few changes as possible to the actual code. See also section C and section 6.8.

```
:- use_module(library(lists), [ member/2,
                                append/2 as list_concat
```

```
                                         ]).
:- use_module(library(option), except([meta_options/3])).
```

In most cases a module is imported because some of its predicates are being used. However, sometimes a module is imported for other reasons, e.g., for its declarations. In such cases it is best practice to use `use_module/2` with empty ImportList. This distinguishes an imported module that is used, although not for its predicates, from a module that is needlessly imported.

The `module/2`, `use_module/1` and `use_module/2` directives are sufficient to partition a simple Prolog program into modules. The SWI-Prolog graphical cross-referencing tool `gxref/0` can be used to analyse the dependencies between non-module files and propose module declarations for each file.

6.4 Controlled autoloading for modules

SWI-Prolog by default support *autoloading* from its standard library. Autoloading implies that when a predicate is found missing during execution the library is searched and the predicate is imported lazily using `use_module/2`. See section 2.14 for details.

The advantage of autoloading is that it requires less typing while it reduces the startup time and reduces the memory footprint of an application. It also allows moving old predicates or emulation thereof the module `backcomp` without affecting existing code. This procedure keeps the libraries and system clean. We make sure that there are not two modules that provide the same predicate as autoload predicate.

Nevertheless, a disadvantage of this autoloader is that the dependencies of a module on the libraries are not explicit and tooling such as PceEmacs or `gxref/0` are required to find these dependencies. Some users want explicit control over which library predicates are accessed from where, preferably by using `use_module/2` which explicitly states which predicates are imported from which library.[1]

Large applications typically contain source files that are not immediately needed and often are not needed at all in many runs of the program. This can be solved by creating an application-specific autoload library, but with multiple parties providing autoloadable predicates the maintenance becomes fragile. For these two reasons we added `autoload/1` and `autoload/2` that behave similar to `use_module/[1,2]`, but do not perform the actual loading. The generic autoloader now proceeds as follows if a missing predicate is encountered:

1. Check `autoload/2` declarations. If one specifies the predicate, import it using `use_module/2`.

2. Check `autoload/1` declarations. If the specified file is loaded, check its export list. Otherwise read the module declaration of the target file to find the exports. If the target predicate is found, import it using `use_module/2`.

3. Perform autoloading from the library if the `autoload` is `true`.

autoload(*:File*)
autoload(*:File, +Imports*)

[1]Note that built-in predicates still add predicates for general use to all name spaces.

Declare that possibly missing predicates in the module in which this declaration occurs are to be resolved by using `use_module/2` on *File* to (possibly) load the file and make the target predicate available. The `autoload/2` variant is tried before `autoload/1`. It is not allowed for two `autoload/2` declarations to provide the same predicate and it is not allowed to define a predicate provided in this way locally. See also `require/1`, which allows specifying predicates for autoloading from their default location.

Predicates made available using `autoload/2` behave as defined predicates, which implies that any operation on them will perform autoloading if necessary. Notably `predicate_property/2`, `current_predicate/1` and `clause/2` are supported.

Currently, neither the existence of *File*, nor whether it actually exports the given predicates (`autoload/2`) is verified when the file is loaded. Instead, the declarations are verified when searching for a missing predicate.

If the Prolog flag `autoload` is set to `false`, these declarations are interpreted as `use_module/[1,2]`.

6.5 Defining a meta-predicate

A meta-predicate is a predicate that calls other predicates dynamically, modifies a predicate, or reasons about properties of a predicate. Such predicates use either a compound term or a *predicate indicator* to describe the predicate they address, e.g., `assert(name(jan))` or `abolish(name/1)`. With modules, this simple schema no longer works as each module defines its own mapping from name+arity to predicate. This is resolved by wrapping the original description in a term ⟨*module*⟩:⟨*term*⟩, e.g., `assert(person:name(jan))` or `abolish(person:name/1)`.

Of course, when calling `assert/1` from inside a module, we expect to assert to a predicate local to this module. In other words, we do not wish to provide this `:/2` wrapper by hand. The `meta_predicate/1` directive tells the compiler that certain arguments are terms that will be used to look up a predicate and thus need to be wrapped (qualified) with ⟨*module*⟩:⟨*term*⟩, unless they are already wrapped.

In the example below, we use this to define `maplist/3` inside a module. The argument '2' in the meta_predicate declaration means that the argument is module-sensitive and refers to a predicate with an arity that is two more than the term that is passed in. The compiler only distinguishes the values 0..9 and `:`, which denote module-sensitive arguments, from `+`, `-` and `?`, which denote *modes*. The values 0..9 are used by the *cross-referencer* and syntax highlighting. Note that the helper predicate `maplist_/3` does not need to be declared as a meta-predicate because the `maplist/3` wrapper already ensures that *Goal* is qualified as ⟨*module*⟩:*Goal*. See the description of `meta_predicate/1` for details.

```
:- module(maplist, [maplist/3]).
:- meta_predicate maplist(2, ?, ?).

%%      maplist(:Goal, +List1, ?List2)
%
%       True if Goal can successfully be applied to all
%       successive pairs of elements from List1 and List2.

maplist(Goal, L1, L2) :-
```

```
        maplist_(L1, L2, Goal).

maplist_([], [], _).
maplist_([H0|T0], [H|T], Goal) :-
        call(Goal, H0, H),
        maplist_(T0, T, Goal).
```

meta_predicate +*Head, ...*
Define the predicates referenced by the comma-separated list *Head* as *meta-predicates*. Each argument of each head is a *meta argument specifier*. Defined specifiers are given below. Only 0..9, `:`, `^` and `//` are interpreted; the mode declarations `+`, `-`, `*` and `?` are ignored.

0..9
The argument is a term that is used to reference a predicate with N more arguments than the given argument term. For example: `call(0)` or `maplist(1, +)`.

:
The argument is module-sensitive, but does not directly refer to a predicate. For example: `consult(:)`.

^
This extension is used to denote the possibly `^`-annotated goal of `setof/3`, `bagof/3`, `aggregate/3` and `aggregate/4`. It is processed similar to '0', but leaving the `^`/2 intact.

//
The argument is a DCG body. See `phrase/3`.

-

?

*****

+
All these have the same semantics, declaring the argument to be not module sensitive. The `*` notation is an alias for `?` for compatibility with e.g., Logtalk. The specific mode has merely documentation value. See section 4.1.1 for details.

Each argument that is module-sensitive (i.e., marked 0..9, `:` or `^`) is qualified with the context module of the caller if it is not already qualified. The implementation ensures that the argument is passed as ⟨*module*⟩:⟨*term*⟩, where ⟨*module*⟩ is an atom denoting the name of a module and ⟨*term*⟩ itself is not a `:/2` term where the first argument is an atom. Below is a simple declaration and a number of queries.

```
:- meta_predicate
        meta(0, +).
```

```
meta(Module:Term, _Arg) :-
        format('Module=~w, Term = ~q~n', [Module, Term]).
```

```
?- meta(test, x).
Module=user, Term = test
?- meta(m1:test, x).
Module=m1, Term = test
?- m2:meta(test, x).
Module=m2, Term = test
?- m1:meta(m2:test, x).
Module=m2, Term = test
?- meta(m1:m2:test, x).
Module=m2, Term = test
?- meta(m1:42:test, x).
Module=42, Term = test
```

The `meta_predicate/1` declaration is the portable mechanism for defining meta-predicates and replaces the old SWI-Prolog specific mechanism provided by the deprecated predicates `module_transparent/1`, `context_module/1` and `strip_module/3`. See also section 6.16.

6.6 Overruling Module Boundaries

The module system described so far is sufficient to distribute programs over multiple modules. 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 into some module. Calling in a particular module is useful for debugging from the user's top level or to access multiple implementations of the same interface that reside in multiple modules. Accessing the same interface from multiple modules cannot be achieved using importing because importing a predicate with the same name and arity from two modules results in a name conflict. Asserting in a different module can be used to create models dynamically in a new module. See section 6.13.

Direct addressing of modules is achieved using a `:/2` explicitly in a program and relies on the module qualification mechanism described in section 6.5. Here are a few examples:

```
?- assert(world:done).    % asserts done/0 into module world
?- world:asserta(done).   % the same
?- world:done.            % calls done/0 in module world
```

Note that the second example is the same due to the Prolog flag `colon_sets_calling_context`. The system predicate `asserta/1` is called in the module `world`, which is possible because system predicates are *visible* in all modules. At the same time, the *calling context* is set to `world`. Because meta arguments are qualified with the calling context, the resulting call is the same as the first example.

6.6.1 Explicit manipulation of the calling context

Quintus' derived module systems have no means to separate the lookup module (for finding predicates) from the calling context (for qualifying meta arguments). Some other Prolog implementations (e.g., ECLiPSe and IF/Prolog) distinguish these operations, using `@/2` for setting the calling context of a goal. This is provided by SWI-Prolog, currently mainly to support compatibility layers.

@(*:Goal, +Module***)**

Execute *Goal*, setting the calling context to *Module*. Setting the calling context affects meta-predicates, for which meta arguments are qualified with *Module* and transparent predicates (see `module_transparent/1`). It has no implications for other predicates.

For example, the code `asserta(done)@world` is the same as `asserta(world:done)`. Unlike in `world:asserta(done)`, `asserta/1` is resolved in the current module rather than the module `world`. This makes no difference for system predicates, but usually does make a difference for user predicates.

Not that SWI-Prolog does not define `@` as an operator. Some systems define this construct using `op(900, xfx, @)`.

6.7 Interacting with modules from the top level

Debugging often requires interaction with predicates that reside in modules: running them, setting spy points on them, etc. This can be achieved using the ⟨*module*⟩:⟨*term*⟩ construct explicitly as described above. In SWI-Prolog, you may also wish to omit the module qualification. Setting a spy point (`spy/1`) on a plain predicate sets a spy point on any predicate with that name in any module. Editing (`edit/1`) or calling an unqualified predicate invokes the DWIM (Do What I Mean) mechanism, which generally suggests the correct qualified query.

Mainly for compatibility, we provide `module/1` to switch the module with which the interactive top level interacts:

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 when debugging a module. The example below lists the clauses of `file_of_label/2` in the module `tex`.

```
1 ?- module(tex).
true.
tex: 2 ?- listing(file_of_label/2).
...
```

6.8 Composing modules from other modules

The predicates in this section are intended to create new modules from the content of other modules. Below is an example to define a *composite* module. The example exports all public predicates of `module_1`, `module_2` and `module_3`, `pred/1` from `module_4`, all predicates from `module_5` except `do_not_use/1` and all predicates from `module_6` while renaming `pred/1` into `mypred/1`.

```
:- module(my_composite, []).
:- reexport([ module_1,
              module_2,
              module_3
            ]).
:- reexport(module_4, [ pred/1 ]).
:- reexport(module_5, except([do_not_use/1])).
:- reexport(module_6, except([pred/1 as mypred])).
```

reexport(*+Files*)
: Load and import predicates as `use_module/1` and re-export all imported predicates. The reexport declarations must immediately follow the module declaration.

reexport(*+File, +Import*)
: Import from *File* as `use_module/2` and re-export the imported predicates. The reexport declarations must immediately follow the module declaration.

6.9 Operators and modules

Operators (section 4.25) are local to modules, where the initial table behaves as if it is copied from the module `user` (see section 6.11). A specific operator can be disabled inside a module using `:- op(0, Type, Name)`. Inheritance from the public table can be restored using `:- op(-1, Type, Name)`.

In addition to using the `op/3` directive, operators can be declared in the `module/2` directive as shown below. Such operator declarations are visible inside the module, and importing such a module makes the operators visible in the target module. Exporting operators is typically used by modules that implement sub-languages such as chr (see chapter 9). The example below is copied from the library `clpfd`.

```
:- module(clpfd,
          [ op(760, yfx, #<==>),
            op(750, xfy, #==>),
            op(750, yfx, #<==),
            op(740, yfx, #\/),
            ...
            (#<==>)/2,
            (#==>)/2,
            (#<==)/2,
            (#\/)/2,
            ...
          ]).
```

6.10 Dynamic importing using import modules

Until now we discussed the public module interface that is, at least to some extent, portable between Prolog implementations with a module system that is derived from Quintus Prolog. The remainder of this chapter describes the underlying mechanisms that can be used to emulate other module systems or implement other code-reuse mechanisms.

In addition to built-in predicates, imported predicates and locally defined predicates, SWI-Prolog modules can also call predicates from its *import modules*. Each module has a (possibly empty) list of import modules. In the default setup, each new module has a single import module, which is `user` for all normal user modules and `system` for all system library modules. Module `user` imports from `system` where all built-in predicates reside. These special modules are described in more detail in section 6.11.

In general, the import relations between modules form an acyclic directed graph. The import relation affects the following mechanisms:

Predicate visibility When looking for a specific predicate definition the system starts in the target module. If the predicate is undefined there it walks the module import relations depth-first left-to-right searching for a module that defines the predicate. The first encountered definition is used. Note that using the default setup this means it searches the `user` and `system` modules (in that order).

Operators Operators are also searched through the import relations. System operators are defined in the module `system`. The user may define operators in `user` to make them globally visible for compatibility with e.g., SICStus Prolog that has no local operators. Normally operators are defined in a module and, when applicable, exported using the `module/2` module header.

The `unknown` flag This flag controls the response to encountering an undefined predicate in the target module.

Term and goal expansion The hooks `term_expansion/2` and `goal_expansion/2` (see section 4.3.1) are *chained* over the import modules that define these hooks. This implies we collect all modules that provide definitions for these hook predicates by traversing the import module relation depth-first and left-to-right. Next, we perform the transformations in a *pipeline*, starting at the target module.

The list of import modules for a specific module can be manipulated and queried using the following predicates, as well as using `set_module/1`.

import_module(+*Module, -Import*) *[nondet]*

True if *Module* inherits directly from *Import*. 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. See also `default_module/2`.

default_module(+*Module, -Default*) *[multi]*

True if predicates and operators in *Default* are visible in *Module*. Modules are returned in the same search order used for predicates and operators. That is, *Default* is first unified with *Module*, followed by the depth-first transitive closure of `import_module/2`.

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.

6.11 Reserved Modules and using the 'user' module

As mentioned above, SWI-Prolog contains two special modules. The first one is the module `system`. This module contains all built-in predicates. Module `system` has no import module, i.e., is a *root* of the module graph. The second special module is the module `user`. This module forms the initial working space of the user. Initially it is empty.[2]. The import module of module `user` is `system`, making all built-in predicates available.

All normal application modules import from the module `user`. This implies they can use all predicates imported into `user` without explicitly importing them. If an application loads all modules from the `user` module using `use_module/1`, one achieves a scoping system similar to the C-language, where every module can access all exported predicates without any special precautions.

All *library* modules (see `module_property/2`) import directly from `system`. Library modules are modules loaded from the SWI-Prolog installation. As they import from `system`, the functionality of a library is not affected by operator or predicate definitions in the `user` module.

6.12 An alternative import/export interface

The `use_module/1` predicate from section 6.3 defines import and export relations based on the filename from which a module is loaded. If modules are created differently, such as by asserting predicates into a new module as described in section 6.13, this interface cannot be used. The interface below provides for import/export from modules that are not created using a module file.

export(*+PredicateIndicator, ...*)
Add predicates 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.

import(*+PredicateIndicator, ...*)
Import predicates *PredicateIndicator* into the current context module. *PredicateIndicator* must specify the source module using the ⟨*module*⟩:⟨*pi*⟩ construct. Note that predicates are normally imported using one of the directives `use_module/[1,2]`. The `import/1` alternative is meant for handling imports into dynamically created modules. See also `export/1` and `export_list/2`.

[2]Unfortunately some *hooks* are traditionally defined in the user module

6.13 Dynamic Modules

So far, we discussed modules that were created by loading a module file. These modules have been introduced to 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. For example:

```
?- assert(world_a:consistent),
   set_prolog_flag(world_a:unknown, fail).
```

These calls create a module called 'world_a' and make the call 'world_a:consistent' succeed. Undefined predicates will not raise an exception for this module (see `unknown`).

Import and export from a dynamically created world can be achieved using `import/1` and `export/1` or by specifying the import module as described in section 6.10.

```
?- world_b:export(solve/2).          % exports solve/2 from world_b
?- world_c:import(world_b:solve/2).  % and import it to world_c
```

6.14 Transparent predicates: definition and context module

The 'module-transparent' mechanism is still underlying the actual implementation. Direct usage by programmers is deprecated. Please use `meta_predicate/1` *to deal with meta-predicates.*

The qualification of module-sensitive arguments described in section 6.5 is realised using *transparent* predicates. It is now deprecated to use this mechanism directly. However, studying the underlying mechanism helps to understand SWI-Prolog's modules. In some respect, the transparent mechanism is more powerful than meta-predicate declarations.

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 for a Prolog term. By default, the context module is the definition module of the predicate running the goal. For transparent predicates, however, this is the context module of the goal inherited from the parent goal. Below, we implement `maplist/3` using the transparent mechanism. The code of `maplist/3` and `maplist_/3` is the same as in section 6.5, but now we must declare both the main predicate and the helper as transparent to avoid changing the context module when calling the helper.

```
:- module(maplist, maplist/3).

:- module_transparent
        maplist/3,
        maplist_/3.
```

```
maplist(Goal, L1, L2) :-
        maplist_(L1, L2, G).

maplist_([], [], _).
maplist_([H0|T0], [H|T], Goal) :-
        call(Goal, H0, H),
        maplist_(T0, T, Goal).
```

Note that *any* call that translates terms into predicates is subject to the transparent mechanism, not just the terms passed to module-sensitive arguments. For example, the module below counts the number of unique atoms returned as bindings for a variable. It works as expected. If we use the directive `:- module_transparent count_atom_results/3.` instead, `atom_result/2` is called wrongly in the module *calling* `count_atom_results/3`. This can be solved using `strip_module/3` to create a qualified goal and a non-transparent helper predicate that is defined in the same module.

```
:- module(count_atom_results,
          [ count_atom_results/3
          ]).
:- meta_predicate count_atom_results(-,0,-).

count_atom_results(A, Goal, Count) :-
        setof(A, atom_result(A, Goal), As), !,
        length(As, Count).
count_atom_results(_, _, 0).

atom_result(Var, Goal) :-
        call(Goal),
        atom(Var).
```

The following predicates support the module-transparent interface:

:- 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.

context_module(-*Module*)
: Unify *Module* with the context module of the current goal. `context_module/1` itself is, of course, transparent.

strip_module(+*Term, -Module, -Plain*)
: Used in module-transparent predicates 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.

6.15 Module properties

The following predicates can be used to query the module system for reflexive programming:

current_module(*?Module*) *[nondet]*
True if *Module* is a currently defined module. This predicate enumerates all modules, whether loaded from a file or created dynamically. Note that modules cannot be destroyed in the current version of SWI-Prolog.

module_property(*?Module, ?Property*)
True if *Property* is a property of *Module*. Defined properties are:

class(*-Class*)
True when *Class* is the class of the module. Defined classes are

user
Default for user-defined modules.

system
Module `system` and modules from ⟨*home*⟩`/boot`.

library
Other modules from the system directories.

temporary
Module is temporary.

test
Modules that create tests.

development
Modules that only support the development environment.

file(*?File*)
True if *Module* was loaded from *File*.

line_count(*-Line*)
True if *Module* was loaded from the N-th line of file.

exports(*-ListOfPredicateIndicators*)
True if *Module* exports the given predicates. Predicate indicators are in canonical form (i.e., always using name/arity and never the DCG form name//arity). Future versions may also use the DCG form. See also `predicate_property/2`. Succeeds with an empty list if the module exports no predicates.

exported_operators(*-ListOfOperators*)
True if *Module* exports the given operators. Each exported operator is represented as a term `op`(*Pri,Assoc,Name*). Succeeds with an empty list if the module exports no operators.

size(*-Bytes*)
Total size in bytes used to represent the module. This includes the module itself, its (hash) tables and the summed size of all predicates defined in this module. See also the `size`(*Bytes*) property in `predicate_property/2`.

program_size(*-Bytes*)
Memory (in bytes) used for storing the predicates of this module. This figure includes the predicate header and clauses.

program_space(*-Bytes*)
If present, this number limits the `program_size`. See `set_module/1`.

last_modified_generation(*-Generation*)
Integer expression the last database generation where a clause was added or removed from a predicate that is implemented in this module. See also `predicate_property/2`.

set_module(*:Property*)
Modify properties of the module. Currently, the following properties may be modified:

base(*+Base*)
Set the default import module of the current module to *Module*. Typically, *Module* is one of `user` or `system`. See section 6.10.

class(*+Class*)
Set the class of the module. See `module_property/2`.

program_space(*+Bytes*)
Maximum amount of memory used to store the predicates defined inside the module. Raises a permission error if the current usage is above the requested limit. Setting the limit to 0 (zero) removes the limit. An attempt to assert clauses that causes the limit to be exceeded causes a `resource_error`(*program_space*) exception. See `assertz/1` and `module_property/2`.

6.16 Compatibility of the Module System

The SWI-Prolog module system is largely derived from the Quintus Prolog module system, which is also adopted by SICStus, Ciao and YAP. Originally, the mechanism for defining meta-predicates in SWI-Prolog was based on the `module_transparent/1` directive and `strip_module/3`. Since 5.7.4 it supports the de-facto standard `meta_predicate/1` directive for implementing meta-predicates, providing much better compatibility.

The support for the `meta_predicate/1` mechanism, however, is considerably different. On most systems, the *caller* of a meta-predicate is compiled differently to provide the required ⟨*module*⟩:⟨*term*⟩ qualification. This implies that the meta-declaration must be available to the compiler when compiling code that calls a meta-predicate. In practice, this implies that other systems pose the following restrictions on meta-predicates:

- Modules that provide meta-predicates for a module to be compiled must be loaded explicitly by that module.
- The meta-predicate directives of exported predicates must follow the `module/2` directive immediately.
- After changing a meta-declaration, all modules that *call* the modified predicates need to be recompiled.

In SWI-Prolog, meta-predicates are also *module-transparent*, and qualifying the module-sensitive arguments is done inside the meta-predicate. As a result, the caller need not be aware that it is calling a meta-predicate and none of the above restrictions hold for SWI-Prolog. However, code that aims at portability must obey the above rules.

Other differences are listed below.

- If a module does not define a predicate, it is searched for in the *import modules*. By default, the import module of any user-defined module is the `user` module. In turn, the `user` module imports from the module `system` that provides all built-in predicates. The auto-import hierarchy can be changed using `add_import_module/3` and `delete_import_module/2`.

  This mechanism can be used to realise a simple object-oriented system or a hierarchical module system.

- Operator declarations are local to a module and may be exported. In Quintus and SICStus all operators are global. YAP and Ciao also use local operators. SWI-Prolog provides global operator declarations from within a module by explicitly qualifying the operator name with the `user` module. I.e., operators are inherited from the *import modules* (see above).

```
:- op(precedence, type, user:(operatorname)).
```

Tabled execution (SLG resolution) 7

This chapter describes SWI-Prolog's support for *Tabled execution* for one or more Prolog predicates, also called *SLG resolution*. Tabling a predicate provides two properties:

1. Re-evaluation of a tabled predicate is avoided by *memoizing* the answers. This can realise huge performance enhancements as illustrated in section 7.1. It also comes with two downsides: the memoized answers are not automatically updated or invalidated if the world (set of predicates on which the answers depend) changes and the answer tables must be stored (in memory).

2. *Left recursion*, a goal calling a *variant* of itself recursively and thus *looping* under the normal Prolog SLD resolution is avoided by suspending the variant call and resuming it with answers from the table. This is illustrated in section 7.2.

Tabling is particularly suited to simplify inference over a highly entangled set of predicates that express axioms and rules in a static (not changing) world. When using SLD resolution for such problems, it is hard to ensure termination and avoid frequent recomputation of intermediate results. A solution is to use Datalog style bottom-up evaluation, i.e., applying rules on the axioms and derived facts until a fixed point is reached. However, bottom-up evaluation typically derives many facts that are never used. Tabling provides a *goal oriented* resolution strategy for such problems and is enabled simply by adding a `table/1` directive to the program.

7.1 Example 1: using tabling for memoizing

As a first classical example we use tabling for *memoizing* intermediate results. We use Fibonacci numbers to illustrate the approach. The Fibonacci number I is defined as the sum of the Fibonacci numbers for $I - 1$ and $I - 2$, while the Fibonacci number of 0 and 1 are both defined to be 1. This can be translated naturally into Prolog:

```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        N > 1,
        N1 is N-1,
        N2 is N-2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1+F2.
```

The complexity of executing this using SLD resolution however is 2^N and thus becomes prohibitively slow rather quickly, e.g., the execution time for $N = 30$ is already 0.4 seconds. Using tabling, `fib`(*N,F*) for each value of N is computed only once and the algorithm becomes linear. Tabling effectively inverts the execution order for this case: it suspends the final addition (F is F1+F2) until the two preceding Fibonacci numbers have been added to the answer tables. Thus, we can reduce the complexity from the show-stopping 2^N to linear by adding a tabling directive and otherwise not changing the algorithm. The code becomes:

```
:- table fib/2.

fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        N > 1,
        N1 is N-1,
        N2 is N-2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1+F2.
```

The price that we pay is that a table `fib`(*I,F*) is created for each I in $0..N$. The execution time for $N = 30$ is now 1 millisecond and computing the Fibonacci number for $N = 1000$ is doable (output edited for readability).

```
1 ?- time(fib(1000, X)).
% 52,991 inferences, 0.013 CPU in 0.013 seconds
X = 70330367711422815821835254877183549770181269836358
    73274260490508715453711819693357974224949456261173
    34877504492417659910881863632654502236471060120533
    74121273867339111198139373125598767690091902245245
    323403501.
```

In the case of Fibonacci numbers we can still rather easily achieve linear complexity using program transformation, where we use bottom-up instead of top-down evaluation, i.e., we compute `fib`(*N,F*) for growing N, where we pass the last two Fibonacci numbers to the next iteration. Not having to create the tables and not having to suspend and resume goals makes this implementation about 25 times faster than the tabled one. However, even in this simple case the transformation is not obvious and it is far more difficult to recognise the algorithm as an implementation of Fibonacci numbers.

```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        fib(1,1,1,N,F).

fib(_F, F1, N, N, F1) :- !.
fib(F0, F1, I, N, F) :-
        F2 is F0+F1,
        I2 is I + 1,
        fib(F1, F2, I2, N, F).
```

7.2 Example 2: avoiding non-termination

SLD resolution easily results in an infinite loop due to *left recursion*, a goal that (indirectly) calls a variant of itself or cycles in the input data. Thus, if we have a series of `connection/2` statements that define railway connections between two cities, we cannot use the most natural logical definition to express that we can travel between two cities:

```
% :- table connection/2.

connection(X, Y) :-
        connection(X, Z),
        connection(Z, Y).
connection(X, Y) :-
        connection(Y, X).

connection('Amsterdam', 'Schiphol').
connection('Amsterdam', 'Haarlem').
connection('Schiphol', 'Leiden').
connection('Haarlem', 'Leiden').
```

After enabling tabling however, the above works just fine as illustrated in the session below. Where is the magic and what is the price we paid? The magic is, again, the fact that new goals to the tabled predicate suspend. So, all recursive goals are suspended. Eventually, a table for `connection`(*'Amsterdam', X*) is created with the two direct connections from Amsterdam. Now, it resumes the first clause using the tabled solutions, continuing the last `connection/2` subgoal with `connection`(*'Schiphol', X*) and `connection`(*'Haarlem', X*). These two go through the same process, creating new suspended recursive calls and creating tables for the connections from Schiphol and Haarlem. Eventually, we end up with a set of tables for each call variant that is involved in computing the transitive closure of the network starting in Amsterdam. However, if the Japanese rail network would have been in our data as well, we would not have produced tables for that.

```
1 ?- connection('Amsterdam', X).
X = 'Haarlem' ;
X = 'Schiphol' ;
X = 'Amsterdam' ;
X = 'Leiden'.
```

Again, the fact that a simple `table/1` directive turns the pure logical specification into a fairly efficient algorithm is a clear advantage. Without tabling the program needs to be *stratified*, introducing a base layer with the raw connections, a second layer that introduces the *commutative* property of a railway (if you can travel from A to B you can also travel from B to A and a final layer that realises *transitivity* (if you can travel from A to B and from B to C you can also travel from A to C). The third and final layer must keep track which cities you have already visited to avoid traveling in circles. The transformed program however uses little memory (the list of already visited cities and the still open choices) and does not need to deal with maintaining consistency between the tables and ground facts.

7.3 Answer subsumption or mode directed tabling

Tabling as defined above has a serious limitation. Although the definition of `connection/2` from section section 7.2 can compute the transitive closure of connected cities, it cannot provide you with a route to travel. The reason is that there are infinitely many routes if there are cycles in the network and each new route found will be added to the answer table and cause the tabled execution's completion algorithm to search for more routes, eventually running out of memory.

The solution to this problem is called *mode directed tabling* or *answer subsumption*.[1] In this execution model one or more arguments are *not* added to the table. Instead, we remember a single *aggregated* value for these arguments. The example below is derived from section 7.2 and returns the connection as a list of cities. This argument is defined as a *moded* argument using the `lattice`(*PI*) mode.[2] This causes the tabling engine each time that it finds an new path to call `shortest/3` and keep the shortest route.

```
:- table
    connection(_,_,lattice(shortest/3)).

shortest(P1, P2, P):-
    length(P1, L1),
    length(P2, L2),
    (   L1 < L2
    ->  P = P1
    ;   P = P2
    ).

connection(X, Y, [X,Y]) :-
    connection(X, Y).
connection(X, Y, P) :-
    connection(X, Z, P0),
    connection(Z, Y),
    append(P0, [Y], P).
```

The mode declaration scheme is equivalent to XSB with partial compatibility support for YAP and B-Prolog. The `lattice`(*PI*) mode is the most general mode. The YAP `all` (B-Prolog `@`) mode is not yet supported. The list below describes the supported modes and indicates the portability.

Var
+
index

A variable (XSB), the atom `index` (YAP) or a + (B-Prolog, YAP) declare that the argument is tabled normally.

lattice(*Pred*)

Pred denotes a predicate with arity 3. It may be specified as an predicate indicator (*Name*/3),

[1] The term *answer subsumption* is used by XSB and *mode directed tabling* by YAP and B-Prolog. The idea is that some arguments are considered 'outputs', where multiple values for the same 'input' are combined. Possibly *answer aggregation* would have been a better name.

[2] This mode is compatible to XSB Prolog.

plain predicate name (*Name*) or a head term `Name(_,_,_)`. On each answer, *PI* is called with three arguments: the current aggregated answer and the new answer are inputs. The last argument must be unified with a term that represents the new aggregated answer.

po(*PI*)
: *Partial Ordering*. The new answer is added iff `call`(*PI, +Old, +Answer*) succeeds. For example, `po('<'/2)` accumulates the largest result. In SWI-Prolog the arity (2) may be omitted, resulting in `po(<)`.

-
first
: The atom `-` (B-Prolog, YAP) and `first` (YAP) declare to keep the first answer for this argument.

last
: The atom `last` (YAP) declares to keep the last answer.

min
: The atom `min` (YAP) declares to keep the smallest answer according to the standard order of terms (see `@</2`). Note that in SWI-Prolog the standard order of terms orders numbers by value.

max
: The atom `max` (YAP) declares to keep the largest answer according to the standard order of terms (see `@>/2`). Note that in SWI-Prolog the standard order of terms orders numbers by value.

sum
: The atom `sum` (YAP) declares to sum numeric answers.

7.4 Tabling for impure programs

Tabling guarantees logically correct results and termination provided the computation only involves terms of bounded size on *pure* Prolog programs, i.e., Prolog programs without side effects or pruning of choice points (cut, `->/2`, etc.). Notably pruning choice points of an incomplete tabled goal may cause an incomplete table and thus cause subsequent queries for the same goal to return an incomplete set of answers. The current SWI-Prolog implementation provides several mechanisms to improve on this situation.

- *Dynamic Strongly Connected Components (SCC)*
  Tabled goals are *completed* as soon as possible. Each fresh tabled goal creates a scheduling component which the system attempts to solve immediately. If a subgoal of the fresh goal refers to an incomplete tabled goal the scheduling components for both goals are merged such that the related goals are completed together. Dynamic rather than static determination of strongly connected components guarantees that the components are minimal because only actually reached code needs to be considered rather than maximally reachable code.

  Minimal SCCs imply that goals are completed as early as possible. This implies that tabled goals may be embedded in e.g., `findall/3` or be used as a condition as long as there is no dependency (*loop*) with goals outside the `findall/3` or condition. For example, the code below misbehaves when called as p(*X*) because the argument of `findall/3` calls a *variant* of the goal that initiated the findall goal. A call p(*1*) however is ok as p(*1*) is not a variant of p(*X*).

```
p(X) :-
    findall(Y, p(Y), Ys),
    ...
```

- *Early completion*
  Ground goals, i.e., goals without variables, are subject to early completion. This implies they are considered completed after the first solution.

7.5 Variant and subsumptive tabling

By default, SWI-Prolog (and other Prolog systems with tabling) create a table per call *variant*. A call (term) is a variant of another call (term) if there is a renaming of variables that makes the two terms equal. See `=@=/2` for details. Consider the following program:

```
:- table p/1.

p(X) :- p(Y), Y < 10 000, X is Y+1.
p(1).
```

Calling p(*X*) creates a table for this variant with 10,000 answers. Calling p(*42*) creates a new table where the recursive call (p(*Y*)) uses the previously created table to enumerate all values $1 \ldots 41$ before deriving p(*42*) is true. *Early completion* (see section 7.4) in this case prevents enumerating all answers for p(*Y*) ($1 \ldots 10,000$). As a result, the query below runs in quadratic time and creates a 10,000 additional tables.

```
?- time(forall(between(1, 10 000, X), p(X))).
% 150,370,553 inferences, 13.256 CPU in 13.256 seconds
```

Subsumptive tabling answers a query using answers from a more general table. In this case, this means it uses basically `trie_gen/2` to get the answer p(*42*) from the table p(_). This leads to the program and results shown below.

```
:- table p/1 as subsumptive.

p(X) :- p(Y), Y < 10 000, X is Y+1.
p(1).
```

```
?- time(p(_)).
% 140,066 inferences, 0.015 CPU in 0.015 seconds
?- time(t).
% 170,005 inferences, 0.016 CPU in 0.016 seconds
```

Subsumptive tabling can be activated in two ways. Per table assign the `... as subsumptive` option and globally by setting the `table_subsumptive` flag to `true`.

One may wonder why subsumptive tabling is not the default. There are also some drawbacks:

- Subsumptive tabling only provides correct support if instances (more specific) queries indeed provides answers that are consistent with the more general query. This is true for *pure programs*, but not guaranteed for arbitrary Prolog programs.
- Finding more generic tables is more complicated and expensive than finding the call variant table and extracting the subset of answers that match the more specific query can be expensive.
- Using subsumptive tables can create more dependencies in the call graph which can slow down the table completion process. Larger dependent components also negatively impact the issues discussed in section 7.4.

7.6 Well Founded Semantics

According to Wikipedia, "*Well Founded Semantics* is one definition of how we can make conclusions from a set of logical rules". Well Founded Semantics (WFS) defines a *three valued logic* representing *true*, *false* and something that is neither true or false. This latter value is often referred to as *bottom*, *undefined* or *unknown*. SWI-Prolog uses `undefined/0`.

Well Founded Semantics allows for reasoning about programs with contradictions or multiple answer sets. It allows for obtaining true/false results for literals that do not depend on the sub program that has no unambiguous solution, propagating the notion of *undefined* to literals that cannot be resolved otherwise and obtaining the *residual* program that expresses why an answer is not unambiguous.

The notion of *Well Founded Semantics* is only relevant if the program uses *negation* as implemented by `tnot/1`. The argument of `tnot/1`, as the name implies, must be a goal associated with a tabled predicate (see `table/1`). In a nutshell, resolving a goal that implies `tnot/1` is implemented as follows:

Consider the following partial *body term*:

```
...,
tnot(p),
q.
```

1. If p has an unconditional answer in its table, fail.
2. Else, *delay* the negation. If an unconditional answer arrives at some time, resume with failure.
3. If at the end of the traditional tabled evaluation we can still not decide on p, execute the *continuation* (q above) while maintaining the *delay list* set to `tnot`(p). If executing the continuation results in an answer for some tabled predicate, record this answer as a *conditional* answer, in this case with the condition `tnot`(p).
4. If a conditional answer is added to a table, it is propagated to its *followers*, say f, adding the pair $\{f$,answer$\}$ to the delay list. If this leads to an answer, the answer is conditional on this pair.

5. After the continuations of all unresolved `tnot/1` calls have been executed the various tables may have conditional answers in addition to normal answers.

6. If there are negative literals that have neither conditional answers nor unconditional answers, the condition `tnot`(*g*) is true. This conclusion is propagated by simplifying the conditions for all answers that depend on `tnot`(*g*). This may result in a definite *false* condition, in which case the answer is removed or a definite *true* condition in which case the answer is made unconditional. Both events can make other conditional answers definitely true or false, etc.

7. At the end of the simplifying process some answers may still be conditional. A final *answer completion* step analyses the graph of depending nodes, eliminating *positive loops*, e.g., "*p* :- *q*. *q* :- *p*". The answers in such a loop are removed, possibly leading to more simplification. This process is executed until *fixed point* is reached, i.e., no further positive loops exist and no further simplification is possible.

The above process may complete without any remaining conditional answers, in which case we are back in the normal Prolog world. It is also possible that some answers remain conditional. The most obvious case is represented by `undefined/0`. The toplevel responds with **undefined** instead of **true** if an answer is conditional.

undefined

Unknown represents neither `true` nor `false` in the well formed model. It is implemented as

```
:- table undefined/0.

undefined :- tnot(undefined).
```

Solving a set of predicates under well formed semantics results in a *residual program*. This program contains clauses for all tabled predicates with condition answers where each clause head represents and answer and each clause body its condition. The condition is a disjunction of conjunctions where each literal is either a tabled goal or `tnot/1` of a tabled goal. The remaining model has at least a cycle through a negative literal (`tnot/1`) and has no single solution in the *stable model semantics*, i.e., it either expresses a contradiction (as `undefined/0`, i.e., there is no stable model) or a multiple stable models as in the program below, where both {p} and {q} are stable models.

```
:- table p/0, q/0.

p :- tnot(q).
q :- tnot(p).
```

Note that it is possible that some literals have the same truth value in all stable models but are still *undefined* under the stable model semantics.

The residual program is an explanation of why an answer is undefined. SWI-Prolog offers the following predicates to access the residual program.

call_residual_program(*:Goal, -Program*)

True when *Goal* is an answer according to the Well Founded Semantics. If *Program* is the empty list, *Goal* is unconditionally true. Otherwise this is a program as described by `delays_residual_program/2`.

call_delays(*:Goal, -Condition*)
True when *Goal* is an answer that is true when Condition can be satisfied. If *Condition* is `true`, *Answer* is unconditional. Otherwise it is a conjunction of goals, each of which is associated with a tabled predicate.

delays_residual_program(*:Condition, -Program*)
Program is a list of clauses that represents the connected program associated with *Condition*. Each clause head represents a conditional answer from a table and each corresponding clause body is the condition that must hold for this answer to be true. The body is a disjunction of conjunctions. Each leaf in this condition is either a term `tnot`(*Goal*) or a plain *Goal*. Each *Goal* is associated with a tabled predicate. The program always contains at least one cycle that involves `tnot/1`.

7.6.1 Well founded semantics and the toplevel

The toplevel supports two modes for reporting that it is undefined whether the current answer is true. The mode is selected by the Prolog flag `toplevel_list_wfs_residual_program`. If `true`, the toplevel uses `call_delays/2` and `delays_residual_program/2` to find the conditional answers used and the *residual* program associated with these answers. It then prints the residual program, followed by the answer and the conditional answers. For `undefined/0`, this results in the following output:

```
?- undefined.
% WFS residual program
    undefined :-
        tnot(undefined).
undefined.
```

If the `toplevel_list_wfs_residual_program` is false, any undefined answer is a conjunction with `undefined/0`. See the program and output below.

```
:- table p/0, q/0.

p :- tnot(q).
q :- tnot(p).
```

```
?- p.
% WFS residual program
    p :-
        tnot(q).
    q :-
        tnot(p).
p.

?- set_prolog_flag(toplevel_list_wfs_residual_program, false).
true.
```

```
?- p.
undefined.
```

7.7 Incremental tabling

Incremental tabling maintains the consistency of a set of tabled predicates that depend on a set of dynamic predicates. Both the tabled and dynamic predicates must have the property `incremental` set. See `dynamic/1` and `table/1`.

Incremental tabling causes the engine to connect the *answer tries* and incremental dynamic predicates in an *Incremental Dependency Graph* (IDG). Modifications (`asserta/1`, `retract/1`, `retractall/1` and friends) of an incremental dynamic predicate mark all depending tables as invalid. Subsequent usage of these tables forces re-evaluation.

Re-evaluation of invalidated tables happens on demand, i.e., on access to an invalid table. First the dependency graph of invalid tables that lead to dynamic predicates is established. Next, tables are re-evaluated in *bottom-up* order. For each re-evaluated table the system determines whether the new table has changed. If the table has not changed, this event is propagated to the *affected* nodes of the IDG and no further re-evaluation may be needed. Consider the following program:

```
:- table (p/1, q/1) as incremental.
:- dynamic([d/1], [incremental(true)]).

p(X) :- q(X).
q(X) :- d(X), X < 10.

d(1).
```

Executing this program creates tables for $X = 1$ for `p/1` and `q/1`. After calling `assert(d(100))` the tables for `p/1` and `q/1` have an *invalid count* of 1. Re-running `p(X)` first re-evaluates `q/1` (bottom-up) which results to the same table as $X = 100$ does not lead to a new answer. Re-evaluation clears the invalid count for `q/1` and, because the `q/1` tables is not changed, decrements the invalid count of affected tables. This sets the *invalid count* for `p/1` to zero, completing the re-evaluation.

Note that invalidating and re-evaluation is done at the level of tables. Notably asserting a clause invalidates all affected tables and may lead to re-evaluating of all these tables. Incremental tabling automates manual abolishing of invalid tables in a changing world and avoids unnecessary re-evaluation if indirectly affected tables prove unaffected because the answer set of dependent tables is unaffected by the change. This is the same policy as implemented in XSB [Swift, 2014]. Future versions may implement a more fine grained approach.

7.8 Monotonic tabling

Incremental tabling (section 7.7) maintains the consistency of tables that depend directly or indirectly on (incremental) dynamic predicates. This is done by *invalidating* dependent tables on an assert or retract and lazily *re-evaluate* invalid tables when their content is needed. Incremental tabling preserves all normal tabling properties, including well founded semantics. The downside is that re-evaluation recomputes the table from scratch. This section deals with *monotonic tabling*, a mechanism that propagates the consequences of `assert/1` and friends without recomputing the dependent tables from scratch. Unlike incremental tabling though, monotonic tabling can only deal with monotonic programs, in particular it does *not* deal with negation.

The example below defines the transitive closure of a bi-directional graph using monotonic tabling. This program builds tables for the `connected/2` and maintains these tables when new facts are added for `link/2`.

```
:- table connected/2 as monotonic.
:- dynamic link/2 as monotonic.

connected(X, Y) :-
    connected(Y, X).
connected(X, Z) :-
    connected(X, Y),
    connected(Y, Z).
connected(X, Y) :-
    link(X, Y).
```

abolish_monotonic_tables
Abolish all monotonic tables and their dependencies.

The list below describes properties of monotonic tabling and relation to other tabling primitives:

- When using `retract/1` on a dynamic monotonic predicate, all dependent tables and dependency links are invalidated and marked for normal *incremental* update.
- `abolish_all_tables/0` destroys all monotonic dependency relations.
- Dynamic predicates can be declared as both `monotonic` and `incremental` and it allowed to have both incremental and monotonic tabled predicates that depend on such dynamic predicates.
- A tabled predicate that depends on a monotonic tabled predicate must be tabled monotonic or incremental. If the dependent predicate is incremental a new answer invalidates the incremental table.

7.8.1 Eager and lazy monotonic tabling

There are two types of monotonic tabling. The default is *eager*, which implies that an asserted clause is immediately propagated through the dependency network. Possibly resulting new answers can be tracked as described in section 7.8.2. The alternative is *lazy*. A predicate is marked for lazy using the `lazy` option as shown below, or by setting the `table_monotonic` flag to `lazy`.

```
:- table p/1 as (monotonic,lazy).
```

If a predicate is tabled as monotonic and lazy and an answer is added to one of the monotonic dynamic predicates, all dependent monotonic or incremental tables are invalidated and the answer is queued together with the dependency. A subsequent call to one of the invalidated tabled predicates re-evaluates the tables. For a monotonic table this implies pushing the queued answers through the dependencies. Removing a clause from one of a monotonic dynamic predicates invalidates all dependent tables and marks all these tables for *forced reevaluation*, which implies they are reevaluated using the same algorithm as used for *incremental* tabling.

Lazy monotonic tables may depend on eager monotonic tables. There is no point in making an eager monotonic table depend on a lazy monotonic table as one would have to reevaluate the lazy table to make the eager table consistent. Therefore, a dependency of an eager table on a lazy table is silently converted into a lazy dependency.

7.8.2 Tracking new answers to monotonic tables

The `prolog_listen/2` interface allows for tracking new facts that are added to monotonic tables. For example, we can print new possible connections from the above program using this code:

```
:- prolog_listen(connected/2, connection_change).

connection_change(new_answer, _:connected(From, To)) :-
    format('~p and ~p are now connected~n', [From, To]).
```

Currently, *failure* of the hook are ignored. If the hook throws an exception this is propagated. The hook is executed outside the current tabling context.[3]

After loading the `connected/2` program and the above declarations we can observe the interaction below. Note that query 1 establishes the dependencies and fills the tables using normal tabling. In the current implementation, possibly discovered connections do not trigger the hook.[4]. Adding a single `link/2` fact links both locations to itself and to each other in both directions. Adding a second fact extends the network.

```
1 ?- connected(_,_).
false.

2 ?- assert(link('Amsterdam', 'Haarlem')).
'Amsterdam' and 'Haarlem' are now connected
'Amsterdam' and 'Amsterdam' are now connected
'Haarlem' and 'Amsterdam' are now connected
'Haarlem' and 'Haarlem' are now connected
true.

3 ?- assert(link('Leiden', 'Haarlem')).
'Leiden' and 'Haarlem' are now connected
'Haarlem' and 'Leiden' are now connected
'Amsterdam' and 'Leiden' are now connected
'Leiden' and 'Amsterdam' are now connected
```

[3] The final behavior may be different in both aspects.
[4] This is likely to change in the future.

```
'Haarlem' and 'Leiden' are now connected
'Leiden' and 'Haarlem' are now connected
'Leiden' and 'Amsterdam' are now connected
'Leiden' and 'Leiden' are now connected
'Amsterdam' and 'Leiden' are now connected
true.
```

7.8.3 Monotonic tabling with external data

Monotonic tables depend on monotonic dynamic predicates. In some situations there is external dynamic data such as a database. One solution is to maintain a shadow copy of all the external data in a dynamic predicate. This wastes resources and introduces maintenance problems. The system allows to use this information directly from the external source. To do this, create a dynamic and monotonic predicate that accesses the data:

```
:- dynamic my_data/2 as monotonic.

my_data(X, Y) :-
    <access external data>.
```

Any monotonic table that depends on `my_data/2` will be populated correctly and build a dependency. Next, if a new answer is added to the external data the user must call `incr_propagate_calls/1` from the Prolog library `increval`. Similarly, when an answer is removed from the extenal data we use `incr_invalidate_calls/1`. Both notification calls must be made *after* the external data has been updated, i.e., `my_data/2` must reflect the new situation before calling `incr_propagate_calls/1` or `incr_invalidate_calls/1`.

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

on_new_my_data(X, Y) :-
    incr_propagate_calls(my_data(X, Y)).

on_removed_my_data(X,Y) :-
    incr_invalidate_calls(my_data(X, Y)).
```

incr_propagate_calls(*:Answer*)
Activate the monotonic answer propagation similarly to when a new fact is asserted for a monotonic dynamic predicate. The *Answer* term must match a monotonic dynamic predicate. See section 7.8.3 for an example.

Status Monotonic tabling is experimental and incomplete. Notably support for *answer subsumption* and *call subsumption* is probably possible and may greatly improve the application domain and resource usage. Monotonic tabling should work with both shared and private tables. Concurrency issues have not yet been tested though.

7.9 Shared tabling

Tables can both be *private* to a thread or *shared* between all threads. Private tables are used only by the calling threads and are discarded as the thread terminates. Shared tables are used by all threads and can only be discarded explicitly. Tables are declared as shared using, e.g.,

```
:- table (p/1, q/2) as shared.
```

A thread may find a table for a particular variant of a shared tabled predicate in any of the following states:

Complete If the table is complete we can simply use its answers.

Fresh/non-existent If the table is still fresh, claim ownership for it and start filling the table. When completed, the ownership relation is terminated.

Incomplete If the table is incomplete and owned by the calling thread, simply continue. If it is owned by another thread we *wait* for the table *unless there is a cycle of threads waiting for each others table*. The latter situation would cause a deadlock and therefore we raise a `deadlock` exception. This exception causes the current SCC to be abandoned and gives other threads the opportunity to claim ownership of the tables that were owned by this thread. The thread that raised the exception and abandoned the SCC simply restarts the leader goal of the SCC. As other threads now have claimed more variants of the SCC it will, in most cases, wait for these threads instead of creating a new deadlock.

A thread that waits for a table may be faced with three results. If the table is complete it can use the answers. It is also possible that the thread that was filling the table raised an exception (either a `deadlock` or any other exception), in which case we find a *fresh* table for which we will try to claim ownership. Finally, some thread may have abolished the table. This situation is the same as when the owning thread raised an exception.

7.9.1 Abolishing shared tables

This section briefly explains the interaction between deleting shared tables and running threads. The core rule is that *abolishing a shared table has no effect on the semantics of the tabled predicates.* An attempt to abolish an incomplete table results in the table to be marked for destruction on completion. The thread that is completing the table continues to do so and continues execution with the computed table answers. Any other thread blocks, waiting for the table to complete. Once completed, the table is destroyed and the waiting threads see a *fresh* table[5].

The current implementation never reclaims shared tables. Instead, they remain part of the global variant table and only the answers of the shared table are reclaimed. Future versions may garbage collect such tables. See also `abolish_shared_tables/0`.

[5]Future versions may avoid waiting by converting the abolished shared table to a private table.

7.9.2 Status and future of shared tabling

Currently, shared tabling has many restrictions. The implementation does not verify that the limitations are met and violating these restrictions may cause incorrect results or crashes. Future versions are expected to resolve these issues.

- Shared tabling currently only handles the basic scenario and cannot yet deal with well formed semantics or incremental tabling.
- As described in section 7.9.1, abolishing shared tables may cause unnecessary waiting for threads to complete the table.
- Only the answers of shared tables can be reclaimed, not the answer table itself.

SWI-Prolog's *continuation based* tabling offers the opportunity to perform *completion* using multiple threads.

7.10 Tabling restraints: bounded rationality and tripwires

Tabling avoids non-termination due to *self-recursion*. As Prolog allows for infinitely nested *compound terms* (*function symbols* in logic) and arbitrary numbers, the set of possible answers is not finite and thus there is no guaranteed termination.

This section describes *restraints* [Grosof & Swift, 2013] that can be enforced to specific or all tabled predicates. Currently there are three defined restraints, limiting (1) the size of (the arguments to) goals, (2) the size of the answer substitution added to a table and (3) the number of answers allowed in any table. If any of these events occurs we can specify the action taken. We distinguish two classes of actions. First, these events can trap a *tripwire* which can be handled using a hook or a predefined action such as raising an exception, printing a warning or enter a *break level*. This can be used for limiting resources, be notified of suspicious events (debugging) or dynamically adjust the (tabling) strategy of the program. Second, they may continue the computation that results in a partial answer (*bounded rationality*). Unlike just not exploring part of the space though, we use the third truth value of well founded semantics to keep track of answers that have not been affected by the restraints and those that have been affected.

The tripwire actions apply for all restraints. If a tripwire action is triggered, the system takes the steps below.

1. Call the `prolog:tripwire/2` hook.
2. If `prolog:tripwire/2` fails, take one of the predefined actions:

   warning
   Print a message indicating the trapped tripwire and continue execution as normal, i.e., the final answer is the same as if no restraint was active.

   error
   Throw an exception `error(resource_error(tripwire(Wire,Context)))`.

   suspend
   Print a message and start a *break level* (see `break/0`).

prolog:tripwire(*Wire, Context***)** *[multifile]*

Called when tripwire *Wire* is trapped. *Context* provides additional context for interpreting the tripwire. The hook can take one of three actions:

- Succeed. In this case the tripwire is considered handled and execution proceeds as if there was no tripwire. Note that tripwires only trigger at the exact value, which implies that a wire on a count will be triggered only once. The hook can install a new tripwire at a higher count.
- Fail. In this case the default action is taken.
- Throw an exception. Exceptions are propagated normally.

Radial restraints limit the sizes of subgoals or answers. Abstraction of a term according to the size limit is implemented by `size_abstract_term/3`.

size_abstract_term(*+Size, +Term, -Abstract***)** *[det]*

The size of a term is defined as the number of compound subterms (*function symbols*) that appear in term. *Abstract* is an abstract copy of *Term* where each argument is abstracted by copying only the first *Size* function symbols and constants. Excess function symbols are replaced by fresh variables.

This predicate is a helper for tabling where *Term* is the `ret/N` *answer skeleton* that is added to the *answer table*. Examples:

Size	Term	Abstract
0	ret(f(x), a)	ret(_, a)
1	ret(f(x), a)	ret(f(x), a)
1	ret(f(A), a)	ret(f(A), a)
1	ret(f(x), x(y(Z)))	ret(f(x), x(_))

radial_restraint *[undefined]*

This predicate is *undefined* in the sense of well founded semantics (see section 7.6 and `undefined/0`). Any answer that depends on this condition is undefined because either the restraint on the subgoal size or answer size was violated.

7.10.1 Restraint subgoal size

Using the `subgoal_abstract`(*Size*) attribute, a tabled subgoal that that is too large is *abstracted* by replacing compound subterms of the goal with variables. In a nutshell, a goal `p`(*s(s(s(s(s(0))))))*) is converted into the semantically equivalent subgoal if the subgoal size is limited to 3.

```
...,
p(s(s(s(X)))), X = s(s(0)),
...,
```

As a result of this, terms stored in the *variant trie* that maps goal variants into *answer tables* is limited. Note that does not limit the number of answer tables as atomic values are never abstracted and there are, for example, an infinite number of integers. Note that restraining the subgoal size does not affect the semantics, provided more general queries on the predicate include all answers that more specific queries do. See also *call substitution* as described in section 7.5. In addition to the tripwire actions, the `max_table_subgoal_size_action` can be set to the value `abstract`:

abstract
Abstract the goal as described above and provide correctness by adding the required unification instructions after the goal.

7.10.2 Restraint answer size

Using the `answer_abstract`(*Size*) attribute, a tabled subgoal that produces answer substitutions (instances of the variables in the goal) whose size exceed *Size* are trapped. In addition to the tripwire actions, answer abstraction defines two additional modes for dealing with too large answers as defines by the Prolog flag `max_table_answer_size_action`:

fail
Ignore the too large answer. Note that this is semantically incorrect.

bounded_rationality
In this mode, the large answer is *abstracted* in the same way as subgoals are abstracted (see section 7.10.1). This is semantically incorrect, but our third truth value *undefined* is used to remedy this problem. In other words, the abstracted value is added to the table as *undefined* and all conclusions that depend on usage of this abstracted value are thus undefined unless they can also be proved some other way.

7.10.3 Restraint answer count

Finally, using "as `max_answers`(*Count*)" or the Prolog flag `max_answers_for_subgoal`, the number of answers in a table is restrained. In addition to the tripwire actions this restraint supports the action `bounded_rationality`[6]. If the restraint is reached in the bounded rationality mode the system takes the following actions:

- Ignore the answer that triggered the restraint.
- Prune the choice points of the tabled goal to avoid more answers.
- Add an new answer to the table that does not bind any variables, i.e., an empty answer substitution. This answer is conditional on `answer_count_restraint/0`.

answer_count_restraint *[undefined]*
This predicate is *undefined* in the sense of well founded semantics (see section 7.6 and `undefined/0`). Any answer that depends on this condition is undefined because the `max_answers` restraint on some table was violated.

The program and subsequent query below illustrate the behavior.

[6] The action `complete_soundly` is supported as a synonym for XSB compatibility

```
:- table p/2 as max_answers(3).

p(M,N) :-
    between(1,M,N).
```

```
?- p(1 000 000, X).
X = 3 ;
X = 2 ;
X = 1 ;
% WFS residual program
    p(1000000, X) :-
        answer_count_restraint.
p(1000000, X).
```

7.11 Tabling predicate reference

:- table(*:Specification*)

Prepare the predicates specified by *Specification* for tabled execution. *Specification* is a *comma-list*, each member specifying tabled execution for a specific predicate. The individual specification is either a *predicate indicator* (name/arity or name//arity) or head specifying tabling with *answer subsumption*.

Although `table/1` is normally used as a directive, SWI-Prolog allows calling it as a runtime predicate to prepare an existing predicate for tabled execution. The predicate `untable/1` can be used to remove the tabling instrumentation from a predicate.

The example below prepares the predicate `edge/2` and the non-terminal statement//1 for tabled execution.

```
:- table edge/2, statement//1.
```

Below is an example declaring a predicate to use tabling with *answer subsumption*. Answer subsumption or *mode directed tabling* is discussed in section 7.3.

```
:- table connection(_,_,min).
```

Additional tabling options can be provided using a term `as/2`, which can be applied to a single specification or a comma list of specifications. The options themselves are a comma-list of one or more of the following atoms:

variant

Default. Create a table for each call variant.

subsumptive
Instead of creating a new table for each call variant, check whether there is a completed table for a more general goal and if this is the case extract the answers from this table. See section 7.5.

shared
Declare that the table shall be shared between threads. See section 7.9

private
Declare that the table shall be local to the calling thread. See section 7.9

incremental
Declare that the table depends on other tables and *incremental* dynamic predicates. See section 7.7.

dynamic
Declare that the predicate is dynamic. Often used together with `incremental`.

This syntax is closely related to the table declarations used in XSB Prolog. Where in XSB `as` is an operator with priority above the priority of the comma, it is an operator with priority below the comma in SWI-Prolog. Therefore, multiple predicates or options must be enclosed in parenthesis. For example:

```
:- table p/1 as subsumptive.
:- table (q/1, r/2) as subsumptive.
```

tnot(*:Goal*)
The `tnot/1` predicate implements *tabled negation*. This predicate realises *Well Founded Semantics*. See section 7.6 for details.

not_exists(*:Goal*)
Handles tabled negation for non-ground (*floundering*) *Goal* as well as non tabled goals. If *Goal* is ground and tabled `not_exists/1` calls `tnot/1`. Otherwise it used `tabled_call`(*Goal*) to create a table and subsequently uses `tnot/1` on the created table.

Logically, `not_exists`(*p(X)*) is defined as tnot($\exists X$(p(X)))

Note that each *Goal* variant populates a table for `tabled_call/1`. Applications may need to abolish such tables to limit memory usage or guarantee consistency 'after the world changed'.

tabled_call(*:Goal*)
Helper predicate for `not_exists/1`. Defined as below. The helper is public because application may need to abolish its tables.

```
:- table tabled_call/1 as variant.
tabled_call(Goal) :- call(Goal).
```

current_table(*:Variant, -Trie*)
True when *Trie* is the answer table for *Variant*.

untable(*:Specification*)
Remove the tables and tabling instrumentation for the specified predicates. *Specification* is compatible with `table/1`, although tabling with *answer subsumption* may be removed using a name/arity specification. The `untable/1` predicate is first of all intended for examining the effect of various tabling scenarios on a particuar program interactively from the toplevel.

Note that although using `untable/1` followed by `table/1` may be used to flush all tables associated with the given predicate(s), flushing tables should be done using one of the table abolish predicates both for better performance and compatibility with other Prolog implementations: `abolish_all_tables/0`, `abolish_private_tables/0`, `abolish_shared_tables/0`, `abolish_module_tables/1` or `abolish_table_subgoals/1`. For example, to remove all tables for `p/3`, run the goal below. The predicate `functor/3` may be used to create a *head term* from a given name and arity.

```
?- abolish_table_subgoals(p(_,_,_)).
```

abolish_all_tables
Remove all tables, both *private* and *shared* (see section 7.9). Since the introduction of *incremental tabling* (see section 7.7) abolishing tables is rarely required to maintain consistency of the tables with a changed environment. Tables may be abolished regardless of the current state of the table. *Incomplete* tables are flagged for destruction when they are completed. See section 7.9.1 for the semantics of destroying shared tables and the following predicates for destroying a subset of the tables: `abolish_private_tables/0`, `abolish_shared_tables/0`, `abolish_table_subgoals/1` and `abolish_module_tables/1`.

abolish_private_tables
Abolish all tables that are private to this thread.

abolish_shared_tables
Abolish all tables that are shared between threads. See also section 7.9.1

abolish_table_subgoals(*:Subgoal*)
Abolish all tables that unify with *SubGoal*. Tables that have undefined answers that depend of the abolished table are abolished as well (recursively). For example, given the program below, `abolish_table_subgoals`(*und*) will also abolish the table for `p/0` because its answer refers to `und/0`.

```
p :- und.
und :- tnot(und).
```

abolish_module_tables(*+Module*)
Remove all tables that belong to predicates in *Module*.

abolish_nonincremental_tables
abolish_nonincremental_tables(*+Options*)
Similar to `abolish_all_tables/0`, but does not abolish *incremental* tables as their consistency is maintained by the system. Options:

on_incomplete(*Action*)
Action is one of `skip` or `error`. If *Action* is `skip`, do not delete the table.[7]

7.12 About the tabling implementation

The SWI-Prolog implementation uses *Delimited continuations* (see section 4.9 to realise suspension of variant calls. The initial version was written by Benoit Desouter and described in [Desouter *et al.*, 2015]. We moved the main data structures required for tabling, the *answer tables* (see section 4.14.4) and the *worklist* to SWI-Prolog's C core. *Mode directed tabling* (section 7.3) is based on a prototype implementation by Fabrizio Riguzzi.

The implementation of dynamic SCCs, dynamically stratified negation and Well Founded Semantics was initiated by Benjamin Grosof from Kyndi and was realised with a lot of help by Theresa Swift, David Warren and Fabrizio Riguzzi, as well as publications about XSB [Sagonas & Swift, 1998, Sagonas *et al.*, 2000].

The `table/1` directive causes the creation of a wrapper calling the renamed original predicate. For example, the program in section 7.2 is translated into the following program. We give this information to improve your understanding of the current tabling implementation. Future versions are likely to use a more low-level translation that is not based on wrappers.

```
connection(A, B) :-
        start_tabling(user:connection(A, B),
                      'connection tabled'(A, B)).

'connection tabled'(X, Y) :-
        connection(X, Z),
        connection(Z, Y).
'connection tabled'(X, Y) :-
        connection(Y, X).

'connection tabled'('Amsterdam', 'Schiphol').
'connection tabled'('Amsterdam', 'Haarlem').
'connection tabled'('Schiphol', 'Leiden').
'connection tabled'('Haarlem', 'Leiden').
```

Status of tabling

The current implementation is merely a first prototype. It needs several enhancements before we can consider it a serious competitor to Prolog systems with mature tabling such as XSB, YAP and B-Prolog. In particular,

- The performance needs to be improved.
- Memory usage needs to be reduced.
- Tables must be shared between threads, both to reduce space and avoid recomputation.

[7]BUG: XSB marks such tables for deletion after completion. That is not yet implemented.

- Tables must be invalidated and reclaimed automatically.
- Notably XSB supports incremental tabling and well-founded semantics under negation.

Constraint Logic Programming 8

This chapter describes the extensions primarily designed to support **constraint logic programming** (CLP), an important declarative programming paradigm with countless practical applications.

CLP(X) stands for constraint logic programming over the domain X. Plain Prolog can be regarded as CLP(H), where H stands for *Herbrand terms*. Over this domain, `=/2` and `dif/2` are the most important constraints that express, respectively, equality and disequality of terms. Plain Prolog can thus be regarded as a special case of CLP.

There are dedicated constraint solvers for several important domains:

- CLP(FD) for **integers** (section A.9)
- CLP(B) for **Boolean** variables (section A.8)
- CLP(Q) for **rational** numbers (section A.10)
- CLP(R) for **floating point** numbers (section A.10).

In addition, CHR (chapter 9) provides a general purpose constraint handling language to reason over user-defined constraints.

Constraints blend in naturally into Prolog programs, and behave exactly like plain Prolog predicates in those cases that can also be expressed without constraints. However, there are two key differences between constraints and plain Prolog predicates:

- Constraints can *delay* checks until their truth can be safely decided. This feature can significantly increase the *generality* of your programs, and preserves their relational nature.
- Constraints can take into account everything you state about the entities you reason about, independent of the order in which you state it, both *before* and also *during* any search for concrete solutions. Using available information to prune parts of the search space is called constraint *propagation*, and it is performed automatically by the available constraint solvers for their respective domains. This feature can significantly increase the *performance* of your programs.

Due to these two key advantages over plain Prolog, CLP has become an extremely important declarative programming paradigm in practice.

Among its most important and typical instances is CLP(FD), constraint logic programming over *integers*. For example, using constraints, you can state in the most general way that a variable *X* is an integer greater than 0. If, later, *X* is bound to a concrete integer, the constraint solver automatically ensures this. If you in addition constrain *X* to integers less than 3, the constraint solver combines the existing knowledge to infer that *X* is either 1 or 2 (see below). To obtain concrete values for *X*, you can ask the solver to *label* *X* and produce 1 and 2 on backtracking. See section A.9.

```
?- use_module(library(clpfd)).
...
true.

?- X #> 0, X #< 3.
X in 1..2.

?- X #> 0, X #< 3, indomain(X).
X = 1 ;
X = 2.
```

Contrast this with plain Prolog, which has no efficient means to deal with (integer) $X > 0$ and $X < 3$. At best it could translate $X > 0$ to `between`(*1, infinite, X*) and a similar primitive for $X < 3$. If the two are combined it has no choice but to generate and test over this infinite two-dimensional space.

Using constraints therefore makes your program more *declarative* in that it frees you from some procedural aspects and limitations of Prolog.

When working with constraints, keep in mind the following:

- As with plain Prolog, `!/0` also destroys the declarative semantics of constraints. A cut after a goal that is delayed may prematurely prune the search space, because the truth of delayed goals is not yet established. There are several ways to avoid cuts in constraint logic programs, retaining both generality and determinism of your programs. See for example `zcompare/3`.
- Term-copying operations (`assertz/1`, `retract/1`, `findall/3`, `copy_term/2`, etc.) generally also copy constraints. The effect varies from ok, silent copying of huge constraint networks to violations of the internal consistency of constraint networks. As a rule of thumb, copying terms holding attributes must be deprecated. If you need to reason about a term that is involved in constraints, use `copy_term/3` to obtain the constraints as Prolog goals, and use these goals for further processing.

All of the mentioned constraint solvers are implemented using the attributed variables interface described in section 8.1. These are lower-level predicates that are mainly intended for library authors, not for typical Prolog programmers.

8.1 Attributed variables

Attributed variables provide a technique for extending the Prolog unification algorithm [Holzbaur, 1992] by hooking the binding of attributed variables. There is no 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]. This interface is simple and available on all Prolog systems that can run the Leuven CHR system (see chapter 9 and the Leuven CHR page).

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_ord_set(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)
        ->  ord_intersection(Domain, Dom2, NewDomain),
            (   NewDomain == []
            ->  fail
            ;   NewDomain = [Value]
            ->  Y = Value
            ;   put_attr(Y, domain, NewDomain)
            )
        ;   var(Y)
        ->  put_attr( Y, domain, Domain )
        ;   ord_memberchk(Y, Domain)
        ).

%       Translate attributes from this module to residual goals

attribute_goals(X) -->
        { get_attr(X, domain, List) },
        [domain(X, List)].
```

Before explaining the code we give some example queries:

`?- domain(X, [a,b]), X = c`	fail
`?- domain(X, [a,b]), domain(X, [a,c]).`	X = a
`?- domain(X, [a,b,c]), domain(X, [a,c]).`	domain(X, [a, c])

The predicate `domain/2` fetches (first clause) or assigns (second clause) the variable a *domain*, a set of values the variable can be unified with. In the second clause, `domain/2` first associates the domain with a fresh variable (Y) and then unifies X to this variable to deal with the possibility that X already has a domain. The predicate `attr_unify_hook/2` (see below) 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), 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). The nonterminal attribute_goals//1 is used to translate remaining attributes to user-readable goals that, when called, reinstate these attributes or attributes that correspond to equivalent constraints.

Implementing constraint solvers (chapter 8) is the most common, but not the only use case for attributed variables: If you implement algorithms that require efficient destructive modifications, then using attributed variables is often a more convenient and somewhat more declarative alternative for `setarg/3` and related predicates whose sharing semantics are harder to understand. In particular, attributed variables make it easy to express graph networks and graph-oriented algorithms, since each variable can store pointers to further variables in its attributes. In such cases, the use of attributed variables should be confined within a module that exposes its functionality via more declarative interface predicates.

8.1.1 Attribute manipulation predicates

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 an uninstantiation error if *Var* is not a variable, and a type error if *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 of whether or not the named attribute is present.

8.1.2 Attributed variable hooks

Attribute names are linked to modules. This means that certain operations on attributed variables cause *hooks* to be called in the module whose name matches the attribute name.

attr_unify_hook(+*AttValue,* +*VarValue***)**
: A hook that must be defined in the module to which an attributed variable refers. It 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. If this predicate fails, the unification fails. If *VarValue* is another attributed variable the hook often combines the two attributes and associates the combined attribute with *VarValue* using `put_attr/3`.

> **To be done** The way in which this hook currently works makes the implementation of important classes of constraint solvers impossible or at least extremely impractical. For increased generality and convenience, simultaneous unifications as in `[X,Y]=[0,1]` should be processed sequentially by the Prolog engine, or a more general hook should be provided in the future. See [Triska, 2016] for more information.

attribute_goals(+*Var*) //

This nonterminal is the main mechanism in which residual constraints are obtained. It is called in every module where it is defined, and *Var* has an attribute. Its argument is that variable. In each module, attribute_goals//1 must describe a list of Prolog goals that are declaratively equivalent to the goals that caused the attributes of that module to be present and in their current state. It is always possible to do this (since these attributes stem from such goals), and it is the responsibility of constraint library authors to provide this mapping without exposing any library internals. Ideally and typically, remaining relevant attributes are mapped to *pure* and potentially simplified Prolog goals that satisfy both of the following:

- They are declaratively equivalent to the constraints that were originally posted.
- They use only predicates that are themselves exported and documented in the modules they stem from.

The latter property ensures that users can reason about residual goals, and see for themselves whether a constraint library behaves correctly. It is this property that makes it possible to thoroughly test constraint solvers by contrasting obtained residual goals with expected answers.

This nonterminal is used by `copy_term/3`, on which the Prolog top level relies to ensure the basic invariant of pure Prolog programs: The answer is *declaratively equivalent* to the query.

The `copy_term/3` primitive uses attribute_goals//1 inside a `findall/3` call. This implies that attribute_goals//1 can unify variables and modify attributes, for example, to tell other hooks that some attribute has already been taken care of. This nonterminal is also used by `frozen/2` which does *not* create a copy. Ideally attribute_goals//1 should not modify anything to allow direct application in `frozen/2`. In the current implementation `frozen/2` backtracks over attribute_goals//1 to tolerate the current behavior. This work-around harms the performance of `frozen/2`. New implementations of attribute_goals//1 should avoid relying on backtracking when feasible. Future versions of `frozen/2` and `copy_term/3` may require attribute_goals//1 not to modify any variables or attributes.

Note that instead of *defaulty* representations, a Prolog *list* is used to represent residual goals. This simplifies processing and reasoning about residual goals throughout all programs that need this functionality.

project_attributes(+*QueryVars, +ResidualVars*)

A hook that can be defined in each module to project constraints on newly introduced variables back to the query variables. *QueryVars* is the list of variables occurring in the query and *ResidualVars* is a list of variables that have attributes attached. There may be variables that occur in both lists. If possible, `project_attributes/2` should change the attributes so that all constraints are expressed as residual goals that refer only to *QueryVars*, while other variables are existentially quantified.

attr_portray_hook(+*AttValue*, +*Var***)** *[deprecated]*
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. This predicate is deprecated because it cannot work with pure interface predicates like `copy_term/3`. Use attribute_goals//1 instead to map attributes to residual goals.

8.1.3 Operations on terms with attributed variables

copy_term(+*Term*, -*Copy*, -*Gs***)**
Create a regular term *Copy* as a copy of *Term* (without any attributes), and a list *Gs* of goals that represents the attributes. The goal `maplist(call, Gs)` recreates the attributes for *Copy*. The nonterminal attribute_goals//1, as defined in the modules the attributes stem from, is used to convert attributes to lists of goals.

This building block is used by the top level to report pending attributes in a portable and understandable fashion. This predicate is the preferred way to reason about and communicate terms with constraints.

The form `copy_term(Term, Term, Gs)` can be used to reason about the constraints in which `Term` is involved.

copy_term_nat(+*Term*, -*Copy***)**
As `copy_term/2`. Attributes, however, are *not* copied but replaced by fresh variables.

term_attvars(+*Term*, -*AttVars***)**
AttVars is a list of all attributed variables in *Term* and its attributes. That is, `term_attvars/2` works recursively through attributes. This predicate is cycle-safe. The goal `term_attvars`(*Term*, *[]*) in an efficient test that *Term* has *no* attributes; scanning the term is aborted after the first attributed variable is found.

8.1.4 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 variable. 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*.

del_attrs(+*Var***)**
If *Var* is an attributed variable, delete *all* its attributes. In all other cases, this predicate succeeds without side-effects.

8.2 Coroutining

Coroutining allows us to delay the execution of Prolog goals until their truth can be safely decided.

Among the most important coroutining predicates is `dif/2`, which expresses *disequality* of terms in a sound way. The actual test is delayed until the terms are sufficiently different, or have become identical. For example:

```
?- dif(X, Y), X = a, Y = b.
X = a,
Y = b.

?- dif(X, Y), X = a, Y = a.
false.
```

There are also lower-level coroutining predicates that are intended as building blocks for higher-level constraints. For example, we can use `freeze/2` to define a variable that can only be assigned an atom:

```
?- freeze(X, atom(X)), X = a.
X = a.
```

In this case, calling `atom/1` earlier causes the whole query to fail:

```
?- atom(X), X = a.
false.
```

If available, domain-specific constraints should be used in such cases. For example, to state that a variable can only assume even integers, use the CLP(FD) constraint `#=/2`:

```
?- X mod 2 #= 0.
X mod 2#=0.
```

Importantly, domain-specific constraints can apply stronger propagation by exploiting logical properties of their respective domains. For example:

```
?- X mod 2 #= 0, X in 1..3.
X = 2.
```

Remaining constraints, such as `X mod 2#=0` in the example above, are called *residual* goals. They are said to *flounder*, because their truth is not yet decided. Declaratively, the query is only true if all residual goals are satisfiable. Use `call_residue_vars/2` to collect all variables that are involved in constraints.

dif(*@A*, *@B*)

The `dif/2` predicate is a *constraint* that is true if and only if *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. Finally, if *A* and *B* can unify, goals are delayed that prevent *A* and *B* to become equal. It is this last property that makes `dif/2` a more general and more declarative alternative for `\=/2` and related predicates.

This 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 variables associated with the module `dif`. It is an autoloaded predicate that is defined in the library `dif`.

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`. See also `frozen/2`.

frozen(@*Term, -Goal***)** *[det]*

Unify *Goal* with the goal or conjunction of goals delayed on some attributed variable in *Term*. If *Term* is free of attributed variables, *Goal* is unified to `true`. Note that `frozen/2` reports all delayed goals, not only those delayed due to `freeze/2`. The goals are extracted using `copy_term/3`.[1] See also `term_attvars/2` and `call_residue_vars/2`.

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 in *X* and *Y*.

The `when/2` predicate is realised using attributed variables associated with the module `when`. It is defined in the autoload library `when`.

call_residue_vars(*:Goal, -Vars***)**

Find residual attributed variables left by *Goal*. This predicate is intended for reasoning about and debugging programs that use coroutining or constraints. To see why this predicate is necessary, consider a predicate that poses contradicting constraints on a variable, and where that variable does not appear in any argument of the predicate and hence does not yield any residual goals on the toplevel when the predicate is invoked. Such programs should fail, but sometimes succeed because the constraint solver is too weak to detect the contradiction. Ideally, delayed goals and constraints are all executed at the end of the computation. The meta predicate `call_residue_vars/2` finds variables that are given attributes or whose attributes are modified by *Goal*, regardless of whether or not these variables are reachable from the arguments of *Goal*.[2].

[1] Versions prior to 8.3.7 only report goals delayed using `freeze/2` on a plain variable. The new behaviour is compatible with SICStus.

[2] The implementation of `call_residue_vars/2` is completely redone in version 7.3.2 (7.2.1) after discussion with Bart Demoen. The current implementation no longer performs full scans of the stacks. The overhead is proportional to the number of attributed variables on the stack, dead or alive.

CHR: Constraint Handling Rules 9

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://dtai.cs.kuleuven.be/CHR/`) you can find more related papers, references and example programs.

9.1 Introduction to CHR

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, and 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, 2009]. More background on CHR can be found at [Frühwirth,].

In section 9.2 we present the syntax of CHR in Prolog and explain informally its operational semantics. Next, section 9.3 deals with practical issues of writing and compiling Prolog programs containing CHR. Section 9.4 explains the (currently primitive) CHR debugging facilities. Section 9.4.3 provides a few useful predicates to inspect the constraint store, and section 9.5 illustrates CHR with two example programs. Section 9.6 describes some compatibility issues with older versions of this system and SICStus' CHR system. Finally, section 9.7 concludes with a few practical guidelines for using CHR.

9.2 CHR Syntax and Semantics

9.2.1 Syntax of CHR rules

```
rules --> rule, rules ; [].

rule --> name, actual_rule, pragma, [atom('.')].

name --> atom, [atom('@')] ; [].

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 --> [] ; 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.

9.2.2 Semantics of CHR

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 constraints can be found, or the matching 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 its 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 semantic 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 behaviour 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 option enables or disables the possibility to debug the CHR code. Possible values are `on` (default) and `off`. See section 9.4 for more details on debugging. The default is derived from the Prolog flag `generate_debug_info`, which is `true` by default. See `--no-debug`. If debugging is enabled, optimization must be disabled.

9.3 CHR in SWI-Prolog Programs

9.3.1 Embedding CHR 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.

9.3.2 CHR 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. Example:

```
:- 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 as a type clash error:

```
CHR compiler ERROR:
    `--> Type clash for variable _ in rule foobar:
                expected type foo in body goal def(_, _)
                expected type bar in head def(_, _)
```

   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.
```

   `bar` appears in the head of the rule 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. For example, 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.

9.3.3 CHR 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 in the following format:

```
:- 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.

9.4 Debugging CHR programs

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 `--no-debug` is used.

9.4.1 CHR debug 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 constraint tries a rule with possibly some passive constraints. The try port is entered just before committing to the rule.

apply
An active constraint commits to a rule with possibly some passive constraints. The apply port is entered just after committing to the rule.

9.4.2 Tracing CHR programs

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.

9.4.3 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
Deactivate 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 9.4.1 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.

find_chr_constraint(-*Constraint*)
Returns a constraint in the constraint store. Via backtracking, all constraints in the store can be enumerated.

9.5 CHR 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)
true .
```

  When the query succeeds, the SWI-Prolog top level prints the content of the CHR constraint store and displays the bindings generated 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.
```

9.6 CHR compatibility

9.6.1 The Old SICStus CHR implementation

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 the programmer should be aware of the appropriate meaning of the Prolog goals used in guards and bodies of CHR rules.

9.6.2 The Old ECLiPSe CHR implementation

The old ECLiPSe CHR implementation features a `label_with/1` construct for labeling variables in CHR constraints. This feature has long since been abandoned. However, a simple transformation is all that is required to port the functionality.

```
label_with Constraint1 if Condition1.
...
label_with ConstraintN if ConditionN.
Constraint1 :- Body1.
...
ConstraintN :- BodyN.
```

is transformed into

```
:- chr_constraint my_labeling/0.

my_labeling \ Constraint1 <=> Condition1 | Body1.
...
my_labeling \ ConstraintN <=> ConditionN | BodyN.
my_labeling <=> true.
```

Be sure to put this code after all other rules in your program! With `my_labeling/0` (or another predicate name of your choosing) the labeling is initiated, rather than ECLiPSe's `chr_labeling/0`.

9.7 CHR 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 (`--no-debug`) 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 lose 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.

- *Finding Constraints*
  The `find_chr_constraint/1` predicate is fairly expensive. Avoid it, if possible. If you must use it, try to use it with an instantiated top-level constraint symbol.

9.8 CHR Compiler Errors and Warnings

In this section we summarize the most important error and warning messages of the CHR compiler.

9.8.1 CHR Compiler Errors

Type clash for variable ... in rule ...

This error indicates an inconsistency between declared types; a variable can not 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 constraint 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.

Multithreaded applications 10

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.

- *Multithreaded server applications*
  Today's 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 10.7.

- *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 multithreading 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. The SWI-Prolog thread implementation has been discussed in the ISO WG17 working group and is largely adopted by YAP and XSB Prolog.[2]

10.1 Creating and destroying Prolog threads

thread_create(*:Goal, -Id*)
Shorthand for `thread_create(Goal, Id, [])`. See `thread_create/3`.

thread_create(*:Goal, -Id, +Options*)
Create a new Prolog thread (and underlying operating system thread) and start it by executing

[1] On an Intel i7-2600K, running Ubuntu Linux 12.04, SWI-Prolog 6.2 creates and joins 32,000 threads per second elapsed time.

[2] The latest version of the ISO draft can be found at `http://logtalk.org/plstd/threads.pdf`. It appears to have dropped from the ISO WG17 agenda.

Goal. If the thread is created successfully, the thread identifier of the created thread is unified to *Id*.

Id is the *alias* name if the option `alias`(*name*) is given. Otherwise it is a *blob* of type `thread`. The anonymous blobs are subject to atom garbage collection. If a thread handle is garbage collected and the thread is not *detached*, it is *joined* if it has already terminated (see `thread_join/2`) and detached otherwise (see `thread_detach/1`).[3] The thread identifier blobs are printed as `<thread>(`*I*,*Ptr*`)`, where *I* is the internal thread identifier and *Ptr* is the unique address of the identifier. The *I* is accepted as input argument for all thread APIs that accept a thread identifier for convenient interaction from the toplevel. See also `thread_property/2`.

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.

affinity(+*CpuSet***)**
: Specify that the thread should only run on the specified CPUs (cores). *CpuSet* is a list of integers between 0 (zero) and the known number of CPUs (see `cpu_count`). If *CpuSet* is empty a `domain_error` is raised. Referring to CPUs equal to or higher than the known number of CPUs returns an `existence_error`.

  This option is currently implemented for systems that provide pthread_attr_setaffinity_np(). The option is silently ignored on other systems.[4]

alias(*AliasName***)**
: Associate an 'alias name' with the thread. This name may be used to refer to the thread and remains valid until the thread is joined (see `thread_join/2`). If the OS supports it (e.g., Linux), the operating system thread is named as well.

at_exit(*:AtExit***)**
: Register *AtExit* as using `thread_at_exit/1` before entering the thread goal. Unlike calling `thread_at_exit/1` as part of the normal *Goal*, this *ensures* the *AtExit* is called. Using `thread_at_exit/1`, the thread may be signalled or run out of resources before `thread_at_exit/1` is reached. See `thread_at_exit/1` for details.

debug(+*Bool***)**
: Enable/disable debugging the new thread. If `false` (default `true`), the new thread is created with the property `debug`(*false*) and debugging is disabled before the new thread is started. The thread debugging predicates such as `tspy/1` and `tdebug/0` do not signal threads with the debug property set to `false`.[5]

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 all resources associated with the thread. If `true`, the system will reclaim all associated resources automatically

[3] Up to version 7.3.23, anonymous thread handles were integers. Using integers did not allow for safe checking of the thread's status as the thread may have died and the handle may have been reused and did not allow for garbage collection to take care of forgotten threads.

[4] BUG: There is currently no way to discover whether this option is supported.

[5] Currently, the flag is only used as a hint for the various debugging primitives, i.e., the system does not really enforce that the target thread stays in *nodebug* mode.

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.

inherit_from(+*ThreadId*)
Inherit defaults from the given *ThreadId* instead of the calling thread. This option was added to ensure that the `__thread_pool_manager` (see `thread_create_in_pool/4`), which is created lazily, has a predictable state. The following properties are inherited:

- The prompt (see `prompt/2`)
- The *typein* module (see `module/1`)
- The standard streams (`user_input`, etc.)
- The default encoding (see `encoding`)
- The default locale (see `set_locale/1`)
- All prolog flags
- The stack limit (see Prolog flag `stack_limit`).

queue_max_size(*Size*)
Enforces a maximum to the number of terms in the input queue. See `message_queue_create/2` with the `max_size`(*o*)ption for details.

stack_limit(*Bytes*)
Set the size limit for the Prolog stacks. See the Prolog flag `stack_limit`. The default is inherited from the calling thread or the thread specified using `inherit_from`(*ThreadId*).

c_stack(*K-Bytes*)
Set the limit to which the C stack of this thread may grow. The default, minimum and maximum values are system-dependent.

The *Goal* argument is *copied* to the new Prolog engine. This implies that 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*)
Calls `thread_join/2` and succeeds if thread *Id* terminated with success. Otherwise the exception `error(thread_error(`*Id, Status*`), _)` is raised, where *Status* is the status as returned by `thread_join/2`.

thread_join(+*Id, -Status*)
Wait for the termination of the thread with the 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. It is not allowed for two threads to join the same thread and the thread

being joined cannot be *detached* (see the `detached(true)` option for `thread_create/3` and `thread_detach/1`).

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*.

Note that the pthread primitive pthread_join() cannot be interrupted. Some systems provide pthread_timedjoin_np(). If this is provided `thread_join/2` is implemented as a loop of timed joins with a 0.25 sec timeout while signals are being tested after each timeout. Such systems allow combining `thread_join/2` with `call_with_time_limit/2` as well as signalling threads blocked in `thread_join/2` using `thread_signal/2`.

thread_alias(+*Alias*)

Set the alias name of the calling thread to *Alias*. An error is raised if the calling thread already has an alias or *Alias* is in use for a thread or message queue.

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 `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 non-detached threads the status can be inspected using `thread_property/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*) *[deprecated]*

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.

The current implementation does not guarantee proper releasing of all mutexes and proper cleanup in `setup_call_cleanup/3`, etc. Please use the exception mechanism (`throw/1`) to abort execution using non-standard control.[6]

[6] BUG: The Windows port does not properly cleanup for *detached* threads while the cleanup for other threads is executed

thread_initialization(*:Goal*)
Run *Goal* when thread is started. This predicate is similar to `initialization/1`, but is intended for initialization operations of the runtime stacks, such as setting global variables as described in section 4.33. *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 run regardless of why the execution of the thread has been completed. When these hooks are run, the return code is already available through `thread_property/2` using the result of `thread_self/1` as thread identifier. Note that there are two scenarios for using exit hooks. Using `thread_at_exit/1` is typically used if the thread creates a side-effect that must be reverted if the thread dies. Another scenario is where the creator of the thread wants to be informed when the thread ends. That cannot be guaranteed by means of `thread_at_exit/1` because it is possible that the thread cannot be created or dies almost instantly due to a signal or resource error. The `at_exit`(*Goal*) option of `thread_create/3` is designed to deal with this scenario.

The *Goal* is executed with signal processing disabled. This avoids that e.g., `thread_signal(Thread, abort)` kills the exit handler rather than the thread in the case the body of *Thread* has just finished when the signal arrives.

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.

thread_affinity(*+ThreadID, -Current, +New*)
True when *Current* is unified with the current thread affinity and the thread affinity is successfully set to *New*. The *thread affinity* specifies the set of CPUs on which this thread is allowed to run. The affinity is represented as a list of non-negative integers. See also the option `affinity`(*+Affinity*) of `thread_create/3`.

This predicate is only present if this functionality can be supported and has been ported to the target operating system. Currently, only Linux support is provided.

10.2 Monitoring threads

Normal multithreaded applications should not need 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 10.5, describing more high-level primitives.

by the thread running `thread_join/2` using the exited thread as *engine*. This is due to a bug in the MinGW pthread implementation.

is_thread(*@Term*)
: True if *Term* is a handle to an existing thread.

thread_property(*?Id, ?Property*)
: True if thread *Id* has *Property*. Either or both arguments may be unbound, enumerating all relations on backtracking. Calling `thread_property/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 opaque thread identifier. Defined properties are:

alias(*Alias*)
: *Alias* is the alias name of thread *Id*.

detached(*Boolean*)
: Current detached status of the thread.

id(*Integer*)
: Integer identifier for the thread. Can be used as argument to the thread predicates, but applications must be aware that these references are reused.

status(*Status*)
: Current status of the thread. *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.

suspended
: Only if the thread is an engine (see section 11). Indicates that the engine is currently not associated with an OS thread.

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`).

engine(*Boolean*)
: If the thread is an engine (see chapter 11), *Boolean* is `true`. Otherwise the property is not present.

thread(*ThreadId*)
: If the thread is an engine that is currently attached to a thread, *ThreadId* is the thread that executes the engine.

size(*Bytes*)
: The amount of memory associated with this thread. This includes the thread structure, its stacks, its default message queue, its clauses in its thread local dynamic predicates (see `thread_local/1`) and memory used for representing thread-local answer tries (see section 7).

system_thread_id(*Integer*)
Thread identifier used by the operating system for the calling thread. Not available on all OSes. This is the same as the Prolog flag `system_thread_id` for the calling thread. Access to the system thread identifier can, on some systems, be used to gain additional control over or information about Prolog threads.

See also `thread_statistics/3` to obtain resource usage information and `message_queue_property/2` to get the number of queued messages for a thread.

thread_statistics(*+Id, +Key, -Value*)
Obtains statistical information on thread *Id* as `statistics/2` does in single-threaded applications. This call supports all keys of `statistics/2`, although only stack sizes, `cputime`, `inferences`, `epoch`, `errors` and `warnings` yield different values for each thread. For `errors` and `warnings` `statistics/2` gives the global process count and this predicate gives the counts for the calling thread.[7]

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 of times the calling thread has to wait for the mutex. The output is written to `current_output` and can thus be redirected using `with_output_to/2`.

10.3 Thread communication

10.3.1 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`. Explicitly created queues come in two flavours. When given an *alias*, they must be destroyed by the user. *Anonymous* message queues are identified by a *blob* (see section 12.4.8) and subject to garbage collection.

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

[7] There is no portable interface to obtain thread-specific CPU time and some operating systems provide no access to this information at all. On such systems the total process CPU is returned. Thread CPU time is supported on MS-Windows, Linux and MacOSX.

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.[8]

thread_send_message(+*Queue*, +*Term*, +*Options*) *[semidet]*

As `thread_send_message/2`, but providing additional *Options*. These are to deal with the case that the queue has a finite maximum size and is full: whereas `thread_send_message/2` will block until the queue has drained sufficiently to accept a new message, `thread_send_message/3` can accept a time-out or deadline analogously to `thread_get_message/3`. The options are:

deadline(+*AbsTime*)
The call fails (silently) if no space has become available before *AbsTime*. See `get_time/1` for the representation of absolute time. If *AbsTime* is earlier then the current time, `thread_send_message/3` fails immediately. Both resolution and maximum wait time is platform-dependent.[9]

timeout(+*Time*)
Time is a float or integer and specifies the maximum time to wait in seconds. This is a relative-time version of the `deadline` option. If both options are provided, the earlier time is effective.

If *Time* is 0 or 0.0, `thread_send_message/3` examines the queue and sends the message if space is available, but does not suspend if no space is available, failing immediately instead.

If *Time* < 0, `thread_send_message/3` fails immediately without sending the message.

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 to *Term*, the term is deleted from the queue.

Please note that non-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* = `gnat`.

```
<thread 1>
thread_get_message(a(A)),

<thread 2>
thread_send_message(Thread_1, b(gnu)),
thread_send_message(Thread_1, a(gnat)),
```

Term may contain attributed variables (see section 8), in which case only terms for which the constraints successfully execute are returned. Handle constraints applies for all predicates that extract terms from message queues. For example, we can get the even numbers from a queue using this code:

[8] See the documentation for the POSIX thread functions pthread_cond_signal() v.s. pthread_cond_broadcast() for background information.

[9] The implementation uses MsgWaitForMultipleObjects() on MS-Windows and pthread_cond_timedwait() on other systems.

```
get_matching_messages(Queue, Pattern, [H|T]) :-
    copy_term(Pattern, H),
    thread_get_message(Queue, H, [timeout(0)]),
    !,
    get_matching_messages(Queue, Pattern, T).
get_matching_messages(_, _, []).

even_numbers(Q, List) :-
    freeze(Even, Even mod 2 =:= 0),
    get_matching_messages(Q, Even, List).
```

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. I.e., `thread_peek_message/1` never waits and does not remove any term from the queue. See also `thread_get_message/3`.

message_queue_create(*?Queue*)
Equivalent to `message_queue_create(Queue,[])`. For compatibility, calling `message_queue_create(`+*Atom*`)` is equivalent to `message_queue_create(`*Queue, [alias(Atom)]*`)`. New code should use `message_queue_create/2` to create a named queue.

message_queue_create(*-Queue, +Options*)
Create a message queue from *Options*. Defined options are:

alias(+*Alias*)
Create a message queue that is identified by the atom *Alias*. Message queues created this way must be explicitly destroyed by the user. If the alias option is omitted, an *Anonymous* queue is created that is identified by a *blob* (see section 12.4.8) and subject to garbage collection.[10]

max_size(+*Size*)
Maximum number of terms in the queue. If this number is reached, `thread_send_message/2` will suspend until the queue is drained. The option can be used if the source, sending messages to the queue, is faster than the drain, consuming the messages.

message_queue_destroy(+*Queue*) *[det]*
Destroy a message queue created with `message_queue_create/1`. A permission error is raised if *Queue* refers to (the default queue of) a thread. Other threads that are waiting for *Queue* using `thread_get_message/2` receive an existence error.

[10]Garbage collecting anonymous message queues is not part of the ISO proposal and most likely not a widely implemented feature.

thread_get_message(+*Queue, ?Term*) *[det]*

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. This predicate raises an existence error exception if *Queue* doesn't exist or is destroyed using `message_queue_destroy/1` while this predicate is waiting.

thread_get_message(+*Queue, ?Term, +Options*) *[semidet]*

As `thread_get_message/2`, but providing additional *Options*:

deadline(+*AbsTime*)

The call fails (silently) if no message has arrived before *AbsTime*. See `get_time/1` for the representation of absolute time. If *AbsTime* is earlier then the current time, `thread_get_message/3` fails immediately. Both resolution and maximum wait time is platform-dependent.[11]

timeout(+*Time*)

Time is a float or integer and specifies the maximum time to wait in seconds. This is a relative-time version of the `deadline` option. If both options are provided, the earlier time is effective.

If *Time* is 0 or 0.0, `thread_get_message/3` examines the queue but does not suspend if no matching term is available. Note that unlike `thread_peek_message/2`, a matching term is removed from the queue.

If *Time* < 0, `thread_get_message/3` fails immediately without removing any message from the queue.

thread_peek_message(+*Queue, ?Term*) *[semidet]*

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_property(*?Queue, ?Property*)

True if *Property* is a property of *Queue*. Defined properties are:

alias(*Alias*)

Queue has the given alias name.

max_size(*Size*)

Maximum number of terms that can be in the queue. See `message_queue_create/2`. This property is not present if there is no limit (default).

size(*Size*)

Queue currently contains *Size* terms. 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.

waiting(*-Count*)

Number of threads waiting for this queue. This property is not present if no threads waits for this queue.

[11]The implementation uses MsgWaitForMultipleObjects() on MS-Windows and pthread_cond_timedwait() on other systems.

The `size`(*Size*) property is always present and may be used to enumerate the created message queues. Note that this predicate does *not enumerate* threads, but can be used to query the properties of the default queue of a thread.

message_queue_set(+*Queue*, +*Property*)
Set a property on the queue. Supported properties are:

max_size(+*Size*)
Change the number of terms that may appear in the message queue before the next `thread_send_message/[2,3]` blocks on it. If the value is higher then the current maximum and the queue has writers waiting, wakeup the writers. The value can be lower than the current number of terms in the queue. In that case writers will block until the queue is drained below the new maximum.

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 Id and number of workers.
%       After the pool is created, post_job/1 can be used to
%       send jobs to the pool.

create_workers(Id, N) :-
        message_queue_create(Id),
        forall(between(1, N, _),
               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.

%%      post_job(+Id, +Goal)
%
%       Post a job to be executed by one of the pool's workers.

post_job(Id, Goal) :-
        thread_send_message(Id, Goal).
```

10.3.2 Waiting for events

While message queues realizes communicating *agents* sharing the same program and optionally dynamic data, the predicate `thread_wait/2` facilitates agents that communicate based on a *shared blackboard*. An important difference is were message queues require the sender and receiver to know about the queue used to communicate and every message can wakeup at most one thread, the blackboard model allows any number (including zero) of threads to *listen* to changes on the blackboard. Any module can act as a blackboard. The blackboard can be updated using the standard Prolog database update predicates (`assert/1`, `retract/1` and friends).

Waiting is implemented using a POSIX *condition variable* and matching *mutex*. On a matching database change the condition variable is signalled using a *broadcast*, waking up all threads waiting in `thread_wait/2`. Multiple database updates can be grouped and cause a single wakeup using `thread_update/2`. This predicate also allows signalling the module condition variable without updating the database and controlling whether all or a single thread is activated.

The blackboard architecture is a good match for an intelligent agent system that has to react on a changing world. Input threads gather sensor data from the world and update a shared world view in a set of dynamic predicates in one or more modules. Agent threads listen to this data or a subset thereof and trigger actions. This is notably a good match with *tabling*, in particular incremental tabling (see section 7.7) and *Well Founded Semantics* (see section 7.6).[12]

thread_wait(*:Goal, :Options*)

Block execution of the calling thread until *Goal* becomes true. The application must be prepared to handle spurious calls to *Goal*, i.e., more calls than asked for based on the *Options* list. A possible exception in *Goal* is propagated and thus terminates `thread_wait/2`.

The wait is associated with a module. This module is derived from the *Options* argument.

The *Options* list specifies when *Goal* is re-evaluated and optionally when the call terminates due to a timeout.

deadline(+*AbsTime*)
timeout(+*Time*)

Timeout and deadline handling. See `thread_get_message/3` for details. This predicate fails when it terminates due to one of these options.

retry_every(+*Time*)

Retry goal every *Time* seconds regardless of whether an event happened. The default is 1 second. This ensures that signals (see `thread_signal/2`) and time limits are respected with an optional delay.[13]

db(+*Boolean*)

Wakeup on arbitrary changes to any dynamic predicate that is defined in the associated module. This is the default if `wait_preds(`+*Preds*`)` is not provided.

wait_preds(+*List*)

Only call *Goal* if at least one of the predicates in *List* has been modified. Each element of *List* is a *predicate indicator* (*Name/Arity* or *Name//Arity* that is resolved to a predicate in

[12]Future versions may provide additional triggers, for example to learn about invalidated tables. Please share your experience.

[13]Some operating systems process such signals immediately, while others only check for such events synchronously.

the module this wait is associated with. If the element is +(*PI*)[14], *Goal* is only triggered if a clause was added (`assert/1`). If the element is −(*PI*), *Goal* is only triggered if a clause was retracted (`retract/1` or `erase/1`). Default is to wakeup on both assert and retract.

modified(*-List*)
The *List* variable normally also appears in *Goal* and is unified with a list of predicates from the `wait_preds` option that have been modified. *List* must be unbound at entry.

module(*+Module*)
Specifies the module to act on explicitly.

The execution of *Goal* is synchronized between all threads calling this predicate on the same module, changes to dynamic predicates in this module and calls to `thread_update/2` on the same module.

This predicate raises a `permision_error` exception when called recursively or called from inside a transaction. See section 4.14.1 for details about interaction with transactions.

thread_update(*:Goal, :Options*)
Update a module (typically using `assert/1` and/or `retract/1` and friends) and on completion signal threads waiting for this module using `thread_wait/2` to reevaluate their *Goal*. *Goal* is synchronized between updating and waiting threads. *Options*:

module(*+Module*)
Determines the module to operate on. Default is the context module associated with the *Options* argument.

notify(*+Atom*)
Determines whether all waiting threads are activated (`broadcast`, default) or a single thread (`signal`).

Compatibility The `thread_wait/2` predicate is modelled after the Qu-Prolog predicate `thread_wait_on_goal/2`. It is largely compatible. Our current implementation does not support predicate time stamps.[15] We made this predicate act on a specific module rather than the entire database. The timeout specification follows that of the other thread waiting predicates and may be combined with the `retry_every` option. The default retry-time is also 1 second rather than *infinite*.

10.3.3 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 multithreaded environments should be handled with care as the receiving thread may hold a *mutex* (see `with_mutex/2`). 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 to handle exceptions at any point.

thread_signal(*+ThreadId, :Goal*)
Make thread *ThreadId* execute *Goal* at the first opportunity. In the current implementation, this

[14] Note that `+p/1` is read as /(+(p),1).
[15] See `predicate_property/2`, property `generation`.

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 multithreading, 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 if the thread is waiting for (user) input.

sig_atomic(*E*)

xecute *Goal* as `once/1` while blocking both thread signals (see `thread_signal/2`) and OS signals (see `on_signal/3`). The system executes some goals while blocking signals. These are:

- The *Setup* call of `setup_call_cleanup/3` and friends.
- The *Cleanup* call of `call_cleanup/2` and friends.
- Compiling a file or loading a *quick load file*.

10.3.4 Threads and dynamic predicates

Besides queues (section 10.3.1) threads can share and exchange data using dynamic predicates. The multithreaded 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 primitives from section 10.4 should be used to ensure that 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 when 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, because 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 feelings in favour or against, please share them in the SWI-Prolog mailing list.

10.4 Thread synchronisation

All internal Prolog operations are thread-safe. This implies that 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 that 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.

The code below provides a solution to this problem based on `with_mutex/2`. It also illustrates the problem of mutexes. The predicate `with_mutex/2` behaves as `once/1` with respect to the guarded goal. This means that our predicate `address/2` is no longer a nice logical non-deterministic relation. This could be solved by explicit locking and unlocking a mutex using `setup_call_cleanup/3`, but at the risk of deadlocking the program if the choice point is left open by accident.

```
change_address(Id, Address) :-
        with_mutex(addressbook,
                   ( retractall(address(Id, _)),
                     asserta(address_db(Id, Address))
                   )).

address(Id, Address) :-
        with_mutex(addressbook,
                   address_db(Id, Address)).
```

Message queues (see `message_queue_create/2`) often provide simpler and more robust ways for threads to communicate. Still, mutexes can be a sensible solution and are therefore provided.

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. Anonymous mutexes are subject to (atom) garbage collection.

mutex_create(*-MutexId, +Options*)
: Create a mutex using options. Defined options are:

alias(*Alias*)
Set the alias name. Using `mutex_create`(*X, [alias(name)]*) is preferred over the equivalent `mutex_create`(*name*).

mutex_destroy(*+MutexId*)
Destroy a mutex. If the mutex is not locked, it is destroyed and further access yields an `existence_error` exception. As of version 7.1.19, this behaviour is reliable. If the mutex is locked, the mutex is scheduled for *delayed destruction*: it will be destroyed when it becomes unlocked.

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 `setup_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 does the mutex become available for locking by other threads. If another thread has locked the mutex the calling thread is suspended until the 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 `setup_call_cleanup/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 *[deprecated]*
Unlock all mutexes held by the current thread. This predicate should not be needed if mutex unlocking is guaranteed with `with_mutex/2` or `setup_call_cleanup/3`.[16]

mutex_property(*?MutexId, ?Property*)
True if *Property* is a property of *MutexId*. Defined properties are:

[16]The also deprecated `thread_exit/1` bypasses the automatic cleanup.

alias(*Alias*)
: Mutex has the defined alias name. See `mutex_create/2` using the 'alias' option.

status(*Status*)
: Current status of the mutex. One of `unlocked` if the mutex is currently not locked, or `locked`(*Owner, Count*) if mutex is locked *Count* times by thread *Owner*. Note that unless *Owner* is the calling thread, the locked status can change at any time. There is no useful application of this property, except for diagnostic purposes.[17]

10.5 Thread support library(threadutil)

This library defines a couple of useful predicates for demonstrating and debugging multithreaded 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.

10.5.1 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`). When using the graphical debugger, the debugger must be *loaded* from the main thread (for example using guitracer) before `gtrace/0` can be called from a thread.

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 `swipl-win.exe` rather than the console-based `swipl.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)).
```

[17]BUG: As *Owner* and *Count* are fetched separately from the mutex, the values may be inconsistent.

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`.

10.5.2 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 4.42 for details on the execution profiler.

10.6 Multithreaded mixed C and Prolog applications

All foreign code linked to the multithreading 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 multithreaded C and Prolog applications you should first familiarise yourself with writing multithreaded applications in C (C++).

If you are using SWI-Prolog as an embedded engine in a multithreaded 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 10.6.1 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 it when finished, as opposed to the model from section 10.6.2, where threads temporarily use a Prolog engine.

10.6.1 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 10.6.2 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_unify_thread_id**(*term_t t, int i*)

Unify *t* with the Prolog thread identifier for thread *i*. Thread identifiers are normally returned from `PL_thread_self()`. Returns -1 if the thread does not exist or the unification fails.

`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.[18] 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. The `flags` field defines the following flags:

PL_THREAD_NO_DEBUG

If this flag is present, the thread starts in normal no-debug status. By default, the debug status is inherited from the main thread.

PL_THREAD_NOT_DETACHED

By default the new thread is created in *detached* mode. With this flag it is created normally, allowing Prolog to *join* the thread.

```
typedef struct
{ size_t    stack_limit;              /* Total stack limit (bytes) */
  size_t    table_space;              /* Total tabling space limit (bytes)
  char *    alias;                    /* alias name */
  int       (*cancel)(int thread);    /* cancel function */
  intptr_t  flags;                    /* PL_THREAD_* flags */
  size_t    max_queue_size;           /* Max size of associated queue */
} 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 multithreading, -2 is returned.

[18]The structure layout changed in version 7.7.14.

`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`.

10.6.2 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 to 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. For systems that use a large number of threads that only infrequently need to call Prolog, it is better to 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 an active internet connection over a pool of threads. If a Prolog engine contains the 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()` returns 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.

10.7 Multithreading and the XPCE graphics system

GUI applications written in XPCE can benefit from Prolog threads if they need to do expensive computations that would otherwise block the UI. 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. In practice, the most feasible approach to graphical Prolog implementations is to control XPCE from a single thread and deploy other threads for (long) computations.

Traditionally, XPCE runs in the foreground (main) thread. We are working towards a situation where XPCE can run comfortably in a separate thread. A separate XPCE thread can be created using pce_dispatch/1. It is also possible to create this thread as the (pce) is loaded by setting the xpce_threaded to true.

Threads other than the thread in which XPCE runs are provided with two predicates to communicate with XPCE.

in_pce_thread(*:Goal*) *[det]*

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.

in_pce_thread_sync(*:Goal*) *[semidet]*

Same as in_pce_thread/1, but wait for *Goal* to be completed. Success depends on the success of executing *Goal*. Variable bindings inside *Goal* are visible to the caller, but it should be noted that the values are being *copied*. If *Goal* throws an exception, this exception is re-thrown by in_pce_thread/1. If the calling thread is the 'pce thread', in_pce_thread_sync/1 executes a direct meta-call. See also in_pce_thread/1.

Note that in_pce_thread_sync/1 is expensive because it requires copying and thread communication. For example, in_pce_thread_synctrue runs at approximately 50,000 calls per second (AMD Phenom 9600B, Ubuntu 11.04).

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`.

Coroutining using Prolog engines 11

Where the term *coroutine* in Prolog typically refer to hooks triggered by *attributed variables* (section 8.1), SWI-Prolog provides two other forms of coroutines. Delimited continuations (see section 4.9) allow creating coroutines that run in the same Prolog engine by capturing and restarting the *continuation*. This section discusses *engines*, also known as *interactors*. The idea was developed by Paul Tarau [Tarau, 2011]. The API described in this chapter has been established together with Paul Tarau and Paulo Moura.

Engines are closely related to *threads* (section 10). An engine is a Prolog virtual machine that has its own stacks and (virtual) machine state. Unlike normal Prolog threads though, they are not associated with an operating system thread. Instead, you *ask* an engine for a next answer (`engine_next/2`). Asking an engine for the next answer attaches the engine to the calling operating system thread and cause it to run until the engine calls `engine_yield/1` or its associated goal completes with an answer, failure or an exception. After the engine yields or completes, it is detached from the operating system thread and the answer term is made available to the calling thread. Communicating with an engine is similar to communicating with a Prolog system though the terminal. In this sense engines are related to *Pengines* as provided by library `pengines`, but where Pengines aim primarily at accessing Prolog engines through the network, engines are in-process entities.

11.1 Examples using engines

We introduce engines by describing application areas and providing simple example programs. The predicates are defined in section 11.3. We identify the following application areas for engines.

1. Aggregating solutions from one or more goals. See section 11.1.1.
2. Access the terms produced in *forward execution* through backtracking without collecting all of them first. Section 11.1.1 illustrates this as well.
3. State accumulation and sharing. See section 11.1.2.
4. Scalable many-agent applications. See section 11.1.3.

11.1.1 Aggregation using engines

Engines can be used to reason about solutions produced by a goal through backtracking. In this scenario we create an engine with the goal we wish to backtrack over and we enumerate all its solution using `engine_next/2`. This usage scenario competes with the all solution predicates (`findall/3`, `bagof/3`, etc.) and the predicates from library `aggregate`. Below we implement `findall/3` using engines.

```
findall(Templ, Goal, List) :-
        setup_call_cleanup(
            engine_create(Templ, Goal, E),
            get_answers(E, List),
            engine_destroy(E)).

get_answers(E, [H|T]) :-
        engine_next(E, H), !,
        get_answers(E, T).
get_answers(_, []).
```

The above is not a particularly attractive alternative for the built-in `findall/3`. It is mostly slower due to time required to create and destroy the engine as well as the (currently[1]) higher overhead of copying terms between engines than the overhead required by the dedicated representation used by `findall/3`.

It gets more interesting if we wish to combine answers from multiple backtracking predicates. Assume we have two predicates that, on backtracking, return ordered solutions and we wish to merge the two answer streams into a single ordered stream of answers. The solution in classical Prolog is below. It collects both answer sets, merges them using ordered set merging and extract the answers. The code is clean and short, but it doesn't produce any answers before both generators are fully enumerated and it uses memory that is proportional to the combined set of answers.

```
:- meta_predicate merge(?,0, ?,0, -).

merge_answers(T1,G1, T2,G2, A) :-
        findall(T1, G1, L1),
        findall(T2, G2, L2),
        ord_union(L1, L2, Ordered),
        member(A, Ordered).
```

We can achieve the same using engines. We create two engines to generate the solutions to both our generators. Now, we can ask both for an answer, put the smallest in the answer set and ask the engine that created the smallest for its next answer, etc. This way we can create an ordered list of answers as above, but now without creating intermediate lists. We can avoid creating the intermediate list by introducing a third engine that controls the two generators and *yields* the answers rather than putting them in a list. This is a general example of turning a predicate that builds a set of terms into a non-deterministic predicate that produces the results on backtracking. The full code is below. Merging the answers of two generators, each generating a set of 10,000 integers is currently about 20% slower than the code above, but the engine-based solution runs in constant space and generates the first solution immediately after both our generators have produced their first solution.

```
:- meta_predicate merge(?,0, ?,0, -).
```

[1]The current implementation of engines is built on top of primitives that are not optimal for the engine use case. There is considerable opportunity to reduce the overhead.

```
merge(T1,G1, T2,G2, A) :-
        engine_create(A, merge(T1,G1, T2,G2), E),
        repeat,
            (   engine_next(E, A)
            ->  true
            ;   !, fail
            ).

merge(T1,G1, T2,G2) :-
        engine_create(T1, G1, E1),
        engine_create(T2, G2, E2),
        (   engine_next(E1, S1)
        ->  (   engine_next(E2, S2)
            ->  order_solutions(S1, S2, E1, E2)
            ;   yield_remaining(S1, E1)
            )
        ;   engine_next(E2, S2),
            yield_remaining(S2, E2)
        ).

order_solutions(S1, S2, E1, E2) :- !,
        (   S1 @=< S2
        ->  engine_yield(S1),
            (   engine_next(E1, S11)
            ->  order_solutions(S11, S2, E1, E2)
            ;   yield_remaining(S2, E2)
            )
        ;   engine_yield(S2),
            (   engine_next(E2, S21)
            ->  order_solutions(S1, S21, E1, E2)
            ;   yield_remaining(S1, E1)
            )
        ).

yield_remaining(S, E) :-
        engine_yield(S),
        engine_next(E, S1),
        yield_remaining(S1, E).
```

11.1.2 State accumulation using engines

Applications that need to manage a state can do so by passing the state around in an additional argument, storing it in a global variable or update it in the dynamic database using `assertz/1` and `retract/1`. Both using an additional argument and a global variable (see `b_setval/2`), make the state subject to backtracking. This may or may not be desirable. If having a state is that subject to backtracking is required, using an additional argument or backtrackable global variable is the right approach. Otherwise, non-backtrackable global variables (nb_setval/2) and dynamic database come into the picture, where global variables are always local to a thread and the dynamic database may or may not be shared between threads (see thread_local/1).

Engines bring an alternative that packages a state inside the engine where it is typically represented in a (threaded) Prolog variable. The state may be updated, while controlled backtracking to a previous state belongs to the possibilities. It can be accessed and updated by anyone with access to the engines' handle. Using an engine to represent state has the following advantages:

- The programming style needed inside the engine is much more 'Prolog friendly', using engine_fetch/1 to read a request and engine_yield/1 to reply to it.
- The state is packaged and subject to (atom) garbage collection.
- The state may be accessed from multiple threads. Access to the state is synchronized without the need for explicit locks.

The example below implements a shared priority heap based on library heaps. The predicate update_heap/1 shows the typical update loop for maintaining state inside an engine: fetch a command, update the state, yield with the reply and call the updater recursively. The update step is guarded against failure. For robustness one may also guard it against exceptions using catch/3. Note that heap_get/3 passes the *Priority* and *Key* it wishes to delete from the heap such that if the unification fails, the heap remains unchanged.

The resulting heap is a global object with either a named or anonymous handle that evolves independently from the Prolog thread(s) that access it. If the heap is anonymous, it is subject to (atom) garbage collection.

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

create_heap(E) :-
        empty_heap(H),
        engine_create(_, update_heap(H), E).

update_heap(H) :-
        engine_fetch(Command),
        (   update_heap(Command, Reply, H, H1)
        ->  true
        ;   H1 = H,
            Reply = false
        ),
        engine_yield(Reply),
        update_heap(H1).

update_heap(add(Priority, Key), true, H0, H) :-
        add_to_heap(H0, Priority, Key, H).
update_heap(get(Priority, Key), Priority-Key, H0, H) :-
        get_from_heap(H0, Priority, Key, H).
```

```
heap_add(Priority, Key, E) :-
        engine_post(E, add(Priority, Key), true).

heap_get(Priority, Key, E) :-
        engine_post(E, get(Priority, Key), Priority-Key).
```

11.1.3 Scalable many-agent applications

The final application area we touch are agent systems were we wish to capture an agent in a Prolog goal. Such systems can be implemented using threads (see section 10) that use `thread_send_message/2` and `thread_get_message/1` to communicate. The main problem is that each thread is associated by an operating system thread. OS threads are, depending on the OS, relatively expensive. Scalability of this design typically ends, depending on OS and hardware, somewhere between 1,000 and 100,000 agents.

Engines provide an alternative. A detached Prolog engine currently requires approximately 20 Kbytes memory on 64 bit hardware, growing with the size of the Prolog stacks. The Prolog stacks may be minimised by calling `garbage_collect/0` followed by `trim_stacks/0`, providing a *deep sleep* mode. The set of agents, each represented by an engine can be controlled by a static or dynamic pool of threads. Scheduling the execution of agents and their communication is completely open and can be optimised to satisfy the requirements of the application.

> This section needs an example. Preferably something that fits on one page and would not scale using threads. Engines might work nice to implement *Antrank: An ant colony algorithm for ranking web pages*.[2]

11.2 Engine resource usage

A Prolog engine consists of a virtual machine state that includes the Prolog stacks. An 'empty' engine requires about 20 KBytes of memory. This grows when the engine requires additional stack space. Anonymous engines are subject to atom garbage collection (see `garbage_collect_atoms/0`). Engines may be reclaimed immediately using `engine_destroy/1`. Calling `engine_destroy/1` destroys the virtual machine state, while the handle itself is left to atom garbage collection. The virtual machine is reclaimed as soon as an engine produced its last result, failed or raised an exception. This implies that it is only advantageous to call `engine_destroy/1` explicitly if you are not interested in further answers.

Engines that are expected to be left in inactive state for a prolonged time can be minimized by calling `garbage_collect/0` and `trim_stacks/0` (in that order) before calling `engine_yield/1` or succeeding.

11.3 Engine predicate reference

This section documents the built-in predicates that deal with engines. In addition to these, most predicates dealing with threads and message queue can be used to access engines.

[2] http://www.ijettcs.org/Volume3Issue2/IJETTCS-2014-04-23-113.pdf

engine_create(+*Template, :Goal, ?Engine*) *[det]*
engine_create(+*Template, :Goal, -Engine, +Options*) *[det]*

Create a new engine and unify *Engine* with a handle to it. *Template* and *Goal* form a pair similar to `findall/3`: the instantiation of *Template* becomes available through `engine_next/2` after *Goal* succeeds. *Options* is a list of the following options. See `thread_create/3` for details.

alias(+*Name*)
Give the engine a name. *Name* must be an atom. If this option is provided, *Engine* is unified with *Name*. The name space for engines is shared with threads and mutexes.

stack(+*Bytes*)
Set the stack limit for the engine. The default is inherited from the calling thread.

The *Engine* argument of `engine_create/3` may be instantiated to an atom, creating an engine with the given alias.

engine_destroy(+*Engine*) *[det]*
Destroy *Engine*.

engine_next(+*Engine, -Term*) *[semidet]*
Ask the engine *Engine* to produce a next answer. On this first call on a specific engine, the *Goal* of the engine is started. If a previous call returned an answer through completion, this causes the engine to backtrack and finally, if the engine produces a previous result using `engine_yield/1`, execution proceeds after the `engine_yield/1` call.

engine_next_reified(+*Engine, -Term*) *[det]*
Similar to `engine_next/2`, but instead of success, failure or or raising an exception, *Term* is unified with one of terms below. This predicate is provided primarily for compatibility with Lean Prolog.

the(*Answer*)
Goal succeeded with *Template* bound to *Answer* or Goal yielded with a term *Answer*.

no
Goal failed.

exception(*Exception*)
Goal raises the error *Exception*.

engine_post(+*Engine, +Term*) *[det]*
Make *Term* available to `engine_fetch/1` inside the *Engine*. This call must be followed by a call to `engine_next/2` and the engine must call `engine_fetch/1`.

engine_post(+*Engine, +Term, -Reply*) *[det]*
Combines `engine_post/2` and `engine_next/2`.

engine_yield(+*Term*) *[det]*
Called from within the engine, causing `engine_next/2` in the caller to return with *Term*. A subsequent call to `engine_next/2` causes `engine_yield/1` to 'return'. This predicate can only be called if the engine is not involved in a callback from C, i.e., when the engine calls a predicate defined in C that calls back Prolog it is not possible to use this predicate. Trying to do so results in a `permission_error` exception.

engine_fetch(*-Term*) *[det]*

Called from within the engine to fetch the term made available through `engine_post/2` or `engine_post/3`. If no term is available an existence_error exception is raised.

engine_self(*-Engine*) *[det]*

Called from within the engine to get access to the handle to the engine itself.

is_engine(@*Term*) *[semidet]*

True if *Term* is a reference to or the alias name of an existing engine.

current_engine(*-Engine*) *[nondet]*

True when *Engine* is an existing engine.

Foreign Language Interface 12

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 a query interface and an interface to extract multiple solutions from a 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.

12.1 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.)

12.2 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 extension calls to 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 running Prolog process.[1]

12.2.1 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 12.2.3 for a suitable high-level interface to these predicates.

12.2.2 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 `swipl-ld` described in section 12.5 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.

12.2.3 library(shlib): Utility library for loading foreign objects (DLLs, shared objects)

This section discusses the functionality of the (autoload) `library(shlib)`, providing an interface to manage shared libraries. We describe the procedure for using a foreign resource (DLL in Windows and shared object in Unix) called `mylib`.

First, one must assemble the resource and make it compatible to SWI-Prolog. The details for this vary between platforms. The swipl-`ld(1)` utility can be used to deal with this in a portable manner. The typical commandline is:

```
swipl-ld -o mylib file.{c,o,cc,C} ...
```

Make sure that one of the files provides a global function `install_mylib()` that initialises the module using calls to PL_register_foreign(). Here is a simple example file mylib.c, which creates a Windows MessageBox:

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

static foreign_t
pl_say_hello(term_t to)
{ char *a;
```

[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 quickly gets smaller and this interface is difficult to port and slow, it is no longer described in this manual. The best alternative would be to use the dld package on machines that do not have shared libraries.

```
  if ( PL_get_atom_chars(to, &a) )
  { MessageBox(NULL, a, "DLL test", MB_OK|MB_TASKMODAL);

    PL_succeed;
  }

  PL_fail;
}

install_t
install_mylib()
{ PL_register_foreign("say_hello", 1, pl_say_hello, 0);
}
```

Now write a file `mylib.pl`:

```
:- module(mylib, [ say_hello/1 ]).
:- use_foreign_library(foreign(mylib)).
```

The file `mylib.pl` can be loaded as a normal Prolog file and provides the predicate defined in C.

use_foreign_library(*+FileSpec*) *[det]*
use_foreign_library(*+FileSpec, +Entry:atom*) *[det]*

Load and install a foreign library as `load_foreign_library/1`,2 and register the installation using `initialization/2` with the option `now`. This is similar to using:

```
:- initialization(load_foreign_library(foreign(mylib))).
```

but using the `initialization/1` wrapper causes the library to be loaded *after* loading of the file in which it appears is completed, while `use_foreign_library/1` loads the library *immediately*. I.e. the difference is only relevant if the remainder of the file uses functionality of the C-library.

As of SWI-Prolog 8.1.22, `use_foreign_library/1`,2 is in provided as a built-in predicate that, if necessary, loads `library(shlib)`. This implies that these directives can be used without explicitly loading `library(shlib)` or relying on demand loading.

qsave:compat_arch(*Arch1, Arch2*) *[semidet,multifile]*

User definable hook to establish if *Arch1* is compatible with *Arch2* when running a shared object. It is used in saved states produced by `qsave_program/2` to determine which shared object to load at runtime.

See also `foreign` option in `qsave_program/2` for more information.

load_foreign_library(*:FileSpec*) *[det]*
load_foreign_library(*:FileSpec, +Entry:atom*) *[det]*

Load a *shared object* or *DLL*. After loading the *Entry* function is called without arguments. The default entry function is composed from =install_=, followed by the file base-name. E.g., the load-call below calls the function `install_mylib()`. If the platform prefixes extern functions with =_=, this prefix is added before calling.

```
...
load_foreign_library(foreign(mylib)),
...
```

Arguments

FileSpec	is a specification for `absolute_file_name/3`. If searching the file fails, the plain name is passed to the OS to try the default method of the OS for locating foreign objects. The default definition of `file_search_path/2` searches <prolog home>/lib/<arch> on Unix and <prolog home>/bin on Windows.

See also `use_foreign_library/1`,2 are intended for use in directives.

unload_foreign_library(*+FileSpec*) *[det]*
unload_foreign_library(*+FileSpec, +Exit:atom*) *[det]*

Unload a *shared object* or *DLL*. After calling the *Exit* function, the shared object is removed from the process. The default exit function is composed from =uninstall_=, followed by the file base-name.

current_foreign_library(*?File, ?Public*)

Query currently loaded shared libraries.

reload_foreign_libraries

Reload all foreign libraries loaded (after restore of a state created using `qsave_program/2`.

win_add_dll_directory(*+AbsDir*) *[det]*

Add *AbsDir* to the directories where dependent DLLs are searched on Windows systems.

Errors `domain_error(operating_system, windows)` if the current OS is not Windows.

12.2.4 Low-level operations on 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 12.2.3 in your application.

open_shared_object(*+File, -Handle*)

File is the name of a shared object file (DLL 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 open_shared_object/3 and load_foreign_library/1.

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.

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().

12.2.5 Static Linking

Older versions of SWI-Prolog were shipped by default with a *static library*. In recent versions we no longer ship a static library because practically every OS properly supports dynamic linking without serious drawbacks and dynamic linking has several advantages. It is on many platforms required to be able to load SWI-Prolog foreign libraries (see use_foreign_library/1). Only on ELF based systems such as Linux we can load foreign libraries *if* the main executable is linked to export its global symbols (gcc -rdynamic option). Another advantage of dynamic libraries is that the user does not have to worry about libraries that this particular build of SWI-Prolog requires such as libgmp as well as OS specific libraries.

If one really wants a static library, use the CMake flag -DSWIPL_STATIC_LIB=ON while configuring a build from source. This causes building and installing libswipl_static.a. Note the _static postfix to avoid a name conflict on Windows between the *import library* and the *static library*.[2].

12.3 Interface Data Types

12.3.1 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.

[2]As is, the Windows build is cross-compiled using MinGW which produces libswipl_static.a. This file can, as far as we know, *not* be used by MSVC.

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 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.

WARNING Term references that are accessed in 'write' (-) mode will refer to an invalid term if the term is allocated on the global stack and backtracking takes us back to a point before the term was written.[3] Compounds, large integers, floats and strings are all allocated on the global stack. Below is a typical scenario where this may happen. The first solution writes a term extracted from the solution into *a*. After the system backtracks due to `PL_next_solution()`, *a* becomes a reference to a term that no longer exists.

```
term_t a = PL_new_term_ref();
...
query = PL_open_query(...);
while(PL_next_solution(query))
{ PL_get_arg(i, ..., a);
}
PL_close_query(query);
```

There are two solutions to this problem. One is to scope the term reference using `PL_open_foreign_frame()` and `PL_close_foreign_frame()` and makes sure it goes out of scope before backtracking happens. The other is to clear the term reference using `PL_put_variable()` before backtracking.

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 to be written by one of the interface functions. For example, `PL_get_arg()` writes a reference to the term argument in its last argument.

[3] This could have been avoided by *trailing* term references when data is written to them. This seriously hurts performance in some scenarios though. If this is desired, use `PL_put_variable()` followed by one of the PL_unify_*() functions.

- *Created by* `PL_new_term_refs(`*int n*`)`
  This function returns a set of term references 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 12.2.

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 of 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 the information necessary to perform its memory management without special precautions from the C programmer.

12.3.2 Other foreign interface types

atom_t An atom in Prolog's 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 a character string C 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 lose their definition).

qid_t Query identifier. Used by `PL_open_query()`, `PL_next_solution()` and `PL_close_query()` to handle backtracking from C.

fid_t Frame identifier. Used by `PL_open_foreign_frame()` and `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 section 12.2.3.

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 that cannot be represented as a C long. Robust code should not assume any of the integer fetching functions to succeed, *even* if the Prolog term is known to be an integer.

PL_ARITY_AS_SIZE

As of SWI-Prolog 7.3.12, the arity of terms has changed from `int` to `size_t`. To deal with this transition, all affecting functions have two versions, where the old name exchanges the arity as `int` and a new function with name *_sz() exchanges the arity as `size_t`. Op to 8.1.28, the default was to use the old `int` functions. As of 8.1.29/8.2.x, the default is to use `size_t` and the old behaviour can be restored by defining `PL_ARITY_AS_SIZE` to `0` (zero). This makes old code compatible, but the following warning is printed when compiling:

```
#warning "Term arity has changed from int to size_t."
#warning "Please update your code or use #define PL_ARITY_AS_SIZE 0."
```

To make the code compile silently again, change the types you use to represent arity from `int` to `size_t`. Please be aware that `size_t` is *unsigned*. At some point representing arity as `int` will be dropped completely.

12.4 The Foreign Include File

12.4.1 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_PRUNED`*)*
  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(***intptr_t value***)**
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 signed value two bits smaller than a pointer, i.e., 30 or 62 bits (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_PRUNED` case and should be aware that the other arguments are not valid in this case.

intptr_t **PL_foreign_context(***control_t***)**
Extracts the context from the context argument. If 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` or `PL_PRUNED`).

*void ** **PL_foreign_context_address(***control_t***)**
Extracts an address as passed in by `PL_retry_address()`.

predicate_t **PL_foreign_context_predicate(***control_t***)**
Fetch the Prolog predicate that is executing this function. Note that if the predicate is imported, the returned predicate refers to the final definition rather than the imported predicate, i.e., the module reported by `PL_predicate_info()` is the module in which the predicate is defined rather than the module where it was called. See also `PL_predicate_info()`.

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_PRUNED` will follow.

The code of figure 12.1 shows a skeleton for a non-deterministic foreign predicate definition.

12.4.2 Atoms and functors

The following functions provide for communication using atoms and functors.

```
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_PRUNED:
        ctxt = PL_foreign_context_address(handle);
        ...
        free(ctxt);
        PL_succeed;
  }
}
```

Figure 12.1: Skeleton for non-deterministic foreign functions

`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 12.4.2). The following atoms are provided as macros, giving access to the empty list symbol and the name of the list constructor. Prior to version 7, `ATOM_nil` is the same as `PL_new_atom(`*"[]"*`)` and `ATOM_dot` is the same as `PL_new_atom(`*"."*`)`. This is no long the case in SWI-Prolog version 7.

atom_t **ATOM_nil(***A***)**

tomic constant that represents the empty list. It is advised to use `PL_get_nil()`, `PL_put_nil()` or `PL_unify_nil()` where applicable.

atom_t **ATOM_dot(***A***)**

tomic constant that represents the name of the list constructor. The list constructor itself is created using `PL_new_functor(`*ATOM_dot,2*`)`. It is advised to use `PL_get_list()`, `PL_put_list()` or `PL_unify_list()` where applicable.

`atom_t` **PL_new_atom_mbchars**(*int rep, size_t len, const char *s*)

This function generalizes `PL_new_atom()` and `PL_new_atom_nchars()` while allowing for multiple encodings. The *rep* argument is one of `REP_ISO_LATIN_1`, `REP_UTF8` or `REP_MB`. If *len* is `(size_t)-1`, it is computed from *s* using strlen().

`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 12.4.2). 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.

`size_t` **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 multithreaded version (see chapter 10), 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.[4]

[4] Otherwise asynchronous atom garbage collection might destroy the atom before it is used.

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.

12.4.3 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.

Testing the type of a term

int **PL_term_type**(*term_t*)
Obtain the type of a term, which should be a term returned by one of the other interface predicates or passed as an argument. The function returns the type of the Prolog term. The type identifiers are listed below. Note that the extraction functions `PL_get_*()` 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) )
        { ...;
        }
```

Version 7 added `PL_NIL`, `PL_BLOB`, `PL_LIST_PAIR` and `PL_DICT`. Older versions classify `PL_NIL` and `PL_BLOB` as `PL_ATOM`, `PL_LIST_PAIR` as `PL_TERM` and do not have dicts.

PL_VARIABLE	A variable or attributed variable
PL_ATOM	A Prolog atom
PL_NIL	The constant `[]`
PL_BLOB	A blob (see section 12.4.8)
PL_STRING	A string (see section 5.2)
PL_INTEGER	A integer
PL_RATIONAL	A rational number
PL_FLOAT	A floating point number
PL_TERM	A compound term
PL_LIST_PAIR	A list cell (`[H|T]`)
PL_DICT	A dict (see section 5.4))

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_rational**(*term_t*)
: Returns non-zero if *term* is a rational number (P/Q). Note that all integers are considered rational and this test thus succeeds for any term for which `PL_is_integer()` succeeds. See also `PL_get_mpq()` and `PL_unify_mpq()`.

`int` **PL_is_float**(*term_t*)
: Returns non-zero if *term* is a float. Note that the corresponding `PL_get_float()` converts rationals (and thus integers).

`int` **PL_is_callable**(*term_t*)
: Returns non-zero if *term* is a callable term. See `callable/1` for details.

`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 using the list constructor or the list terminator. See also `PL_is_pair()` and `PL_skip_list()`.

int **PL_is_pair**(*term_t*)
: Returns non-zero if *term* is a compound term using the list constructor. See also `PL_is_list()` and `PL_skip_list()`.

int **PL_is_dict**(*term_t*)
: Returns non-zero if *term* is a dict. See also `PL_put_dict()` and `PL_get_dict_key()`.

int **PL_is_atomic**(*term_t*)
: Returns non-zero if *term* is atomic (not a variable or compound).

int **PL_is_number**(*term_t*)
: Returns non-zero if *term* is an rational (including integers) or float.

int **PL_is_acyclic**(*term_t*)
: Returns non-zero if *term* is acyclic (i.e. a finite tree).

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, size_t *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 safe 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 be converted and the second how the argument is stored. Below is a specification of these constants. `BUF_STACK` implies, if the data is not static (as from an atom), that the data is pushed on a stack. If BUF_MALLOC is used, the data must be freed using `PL_free()` when no longer needed.

With the introduction of wide characters (see section 2.19.1), 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 of character codes.

CVT_INTEGER
Convert if term is an integer.

CVT_FLOAT
Convert if term is a float. The characters returned are the same as `write/1` would write for the floating point number.

CVT_NUMBER
Convert if term is an 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_STACK` is implied.

CVT_WRITE_CANONICAL
As `CVT_WRITE`, but using `write_canonical/2`.

CVT_WRITEQ
As `CVT_WRITE`, but using `writeq/2`.

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_STACK
Data is stored on a stack. The older `BUF_RING` is an alias for `BUF_STACK`. See section 12.4.12.

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
Text is in ISO Latin-1 encoding and the call fails if text cannot be represented. This flag has the value 0 and is thus the default.

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 none 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*.

`int` **PL_get_uint64**(*term_t +t, uint64_t *i*)
If *t* is a Prolog integer that can be represented as a `uint64_t`, assign its value over *i*. Note that this requires GMP support for representing `uint64_t` values with the high bit set.

`int` **PL_get_intptr**(*term_t +t, intptr_t *i*)
Get an integer that is at least as wide as a pointer. On most platforms this is the same as `PL_get_long()`, but on Win64 pointers are 8 bytes and longs only 4. Unlike `PL_get_pointer()`, the value is not modified.

`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()` and `PL_get_pointer()` guarantee pointers in the range of malloc() are handled without truncating.

`int` **PL_get_float**(*term_t +t, double *f*)
If *t* is a float, integer or rational number, its value is assigned over *f*. Note that if *t* is an integer or rational conversion may fail because the number cannot be represented as a float.

`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, size_t *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()`. See section 12.3.2.

`int` **PL_get_compound_name_arity**(*term_t +t, atom_t *name, size_t *arity*)
If *t* is compound term, the functor name will be assigned over *name* and the arity over *arity*. This is the same as `PL_get_name_arity()`, but this function fails if *t* is an atom.

`int` **PL_get_module**(*term_t +t, module_t *module*)
If *t* is an atom, the system will look up or create the corresponding module and assign an opaque pointer to it over *module*.

`int` **PL_get_arg**(*size_t index, term_t +t, term_t -a*)
If *t* is compound and index is between 1 and arity (inclusive), assign *a* with a term reference to the argument.

`int` **_PL_get_arg**(*size_t index, term_t +t, term_t -a*)
Same as `PL_get_arg()`, but no checking is performed, neither whether *t* is actually a term nor whether *index* is a valid argument index.

`int` **PL_get_dict_key**(*atom_t key, term_t +dict, term_t -value*)
If *dict* is a dict, get the associated value in *value*. Fails silently if *key* does not appear in *dict* or if if *dict* is not a dict.

Exchanging text using length and string

All internal text representation in 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, size_t *len, char **s*)
See `PL_get_atom_chars()`.

`int` **PL_get_list_nchars**(*term_t t, size_t *len, char **s*)
See `PL_get_list_chars()`.

`int` **PL_get_nchars**(*term_t t, size_t *len, char **s, unsigned int flags*)
See `PL_get_chars()`.

`int` **PL_put_atom_nchars**(*term_t t, size_t len, const char *s*)
See `PL_put_atom_chars()`.

`int` **PL_put_string_nchars**(*term_t t, size_t len, const char *s*)
See `PL_put_string_chars()`.

`int` **PL_put_list_ncodes**(*term_t t, size_t len, const char *s*)
See `PL_put_list_codes()`.

`int` **PL_put_list_nchars**(*term_t t, size_t len, const char *s*)
See `PL_put_list_chars()`.

`int` **PL_unify_atom_nchars**(*term_t t, size_t len, const char *s*)
See `PL_unify_atom_chars()`.

`int` **PL_unify_string_nchars**(*term_t t, size_t len, const char *s*)
See `PL_unify_string_chars()`.

`int` **PL_unify_list_ncodes**(*term_t t, size_t len, const char *s*)
See `PL_unify_codes()`.

`int` **PL_unify_list_nchars**(*term_t t, size_t 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**(*size_t len, const char *s*)
Create a new atom as `PL_new_atom()`, but using the given length and characters. If *len* is `(size_t)-1`, it is computed from *s* using strlen().

`const char *` **PL_atom_nchars**(*atom_t a, size_t *len*)
Extract the text and length of an atom.

Wide-character versions

Support for exchange of wide-character strings is still under consideration. 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 both 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**(*size_t len, const pl_wchar_t *s*)
Create atom from wide-character string as `PL_new_atom_nchars()` does for ISO-Latin-1 strings. If *s* only contains ISO-Latin-1 characters a normal byte-array atom is created. If *len* is `(size_t)-1`, it is computed from *s* using wcslen().

`pl_wchar_t*` **PL_atom_wchars**(*atom_t atom, int *len*)
Extract characters from a wide-character atom. Succeeds on any atom marked as 'text'. If the underlying atom is a wide-character atom, the returned pointer is a pointer into the atom structure. If it is an ISO-Latin-1 character, the returned pointer comes from Prolog's 'buffer ring' (see `PL_get_chars()`).

`int` **PL_get_wchars**(*term_t t, size_t *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, size_t len, const pl_wchar_t *s*)
Unify *t* with a textual representation of the C wide-character array *s*. The *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, size_t 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. Also, 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();   /* the elements */
  term_t list = PL_copy_term_ref(l); /* copy (we modify list) */

  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 [] */
}
```

Note that as of version 7, lists have a new representation unless the option `--traditional` is used. see section 5.1.

int **PL_get_list**(*term_t +l, term_t -h, term_t -t*)
If *l* is a list and not the empty list, 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 the empty list, 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 the empty list, assign a term reference to the tail to *t*.

int **PL_get_nil**(*term_t +l*)
Succeeds if *l* represents the list termination constant.

int **PL_skip_list**(*term_t +list, term_t -tail, size_t *len*)
This is a multi-purpose function to deal with lists. It allows for finding the length of a list, checking whether something is a list, etc. The reference *tail* is set to point to the end of the list, *len* is filled with the number of list-cells skipped, and the return value indicates the status of the list:

PL_LIST
The list is a 'proper' list: one that ends in the list terminator constant and *tail* is filled with the terminator constant.

PL_PARTIAL_LIST
The list is a 'partial' list: one that ends in a variable and *tail* is a reference to this variable.

PL_CYCLIC_TERM
The list is cyclic (e.g. X = [a—X]). *tail* points to an arbitrary cell of the list and *len* is at most twice the cycle length of the list.

PL_NOT_A_LIST
The term *list* is not a list at all. *tail* is bound to the non-list term and *len* is set to the number of list-cells skipped.

It is allowed to pass 0 for *tail* and `NULL` for *len*.

An example: defining `write/1` in C

Figure 12.2 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.

12.4.4 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()`.

`int` **PL_put_variable**(*term_t -t*)
Put a fresh variable in the term, resetting the term reference to its initial state.[5]

`int` **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()`.

`int` **PL_put_bool**(*term_t -t, int val*)
Put one of the atoms `true` or `false` in the term reference See also `PL_put_atom()`, `PL_unify_bool()` and `PL_get_bool()`.

`int` **PL_put_chars**(*term_t -t, int flags, size_t len, const char *chars*)
New function to deal with setting a term from a `char*` with various encodings. The *flags* argument is a bitwise *or* specifying the Prolog target type and the encoding of *chars*. A Prolog type is one of `PL_ATOM`, `PL_STRING`, `PL_CODE_LIST` or `PL_CHAR_LIST`. A representation is one of `REP_ISO_LATIN_1`, `REP_UTF8` or `REP_MB`. See `PL_get_chars()` for a definition of the representation types. If *len* is `-1` *chars* must be zero-terminated and the length is computed from *chars* using strlen().

[5]Older versions created a variable on the global stack.

```
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 12.2: A Foreign definition of `display/1`

`int` **PL_put_atom_chars**(*term_t -t, const char *chars*)
Put an atom in the term reference constructed from the zero-terminated string. The string itself will never be referenced by Prolog after this function.

`int` **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()`.

`int` **PL_put_string_nchars**(*term_t -t, size_t 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 12.4.22.

`int` **PL_put_list_chars**(*term_t -t, const char *chars*)
Put a list of ASCII values in the term reference.

`int` **PL_put_integer**(*term_t -t, long i*)
Put a Prolog integer in the term reference.

`int` **PL_put_int64**(*term_t -t, int64_t i*)
Put a Prolog integer in the term reference.

`int` **PL_put_uint64**(*term_t -t, uint64_t i*)
Put a Prolog integer in the term reference. Note that unbounded integer support is required for `uint64_t` values with the highest bit set to 1. Without unbounded integer support, too large values raise a `representation_error` exception.

`int` **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.

`int` **PL_put_float**(*term_t -t, double f*)
Put a floating-point value in the term reference.

`int` **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()`.

`int` **PL_put_list**(*term_t -l*)
As `PL_put_functor()`, using the list-cell functor. Note that on classical Prolog systems or in SWI-Prolog using the option `--traditional`, this is `./2`, while on SWI-Prolog version 7 this is `[|]/2`.

`int` **PL_put_nil**(*term_t -l*)
Put the list terminator constant in *l*. Always returns `TRUE`. Note that in classical Prolog systems or in SWI-Prolog using the option `--traditional`, this is the same as `PL_put_atom_chars("[]")`. See section 5.1.

`int` **PL_put_term**(*term_t -t1, term_t +t2*)
Make *t1* point to the same term as *t2*. Under the unusual condition that *t2* is a fresh term reference this function requires a global stack cell and may thus return *FALSE* and leave a resource exception in the environment.

int **PL_cons_functor**(*term_t -h, functor_t f, . . .*)

Create a term whose arguments are filled from a 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.

int **PL_cons_functor_v**(*term_t -h, functor_t f, term_t a0*)

Create 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.

int **PL_cons_list**(*term_t -l, term_t +h, term_t +t*)

Create a list (cons-) cell in *l* from the head *h* and tail *t*. 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.

int **PL_put_dict**(*term_t -h, atom_t tag, size_t len, const atom_t *keys, term_t values*)

Create a dict from a *tag* and vector of atom-value pairs and put the result in *h*. The dict's key is set by *tag*, which may be `0` to leave the tag unbound. The *keys* vector is a vector of atoms of at least *len* long. The *values* is a term vector allocated using `PL_new_term_refs()` of at least *len* long. This function returns `TRUE` on success, `FALSE` on a resource error (leaving a resource error exception in the environment), `-1` if some key or the *tag* is invalid and `-2` if there are duplicate keys.

12.4.5 Unifying data

The functions of this section *unify* terms with other terms or translated C data structures. Except for `PL_unify()`, these functions are specific to SWI-Prolog. They have been introduced because they shorten the code for returning data to Prolog and at the same time make this more efficient by avoiding the need to allocate temporary term references and reduce the number of calls to the Prolog API. 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;
}
```

Note that unification functions that perform multiple bindings may leave part of the bindings in case of failure. See `PL_unify()` for details.

`int` **PL_unify**(*term_t ?t1, term_t ?t2*)
Unify two Prolog terms and return `TRUE` on success.

Care is needed if `PL_unify()` returns `FAIL` and the foreign function does not *immediately* return to Prolog with `FAIL`. Unification may perform multiple changes to either *t1* or *t2*. A failing unification may have created bindings before failure is detected. *Already created bindings are not undone*. For example, calling `PL_unify()` on `a`(*X, a*) and `a`(*c,b*) binds *X* to `c` and fails when trying to unify `a` to `b`. If control remains in C or even if we want to return success to Prolog, we *must* undo such bindings. This is achieved using `PL_open_foreign_frame()` and `PL_rewind_foreign_frame()`, as shown in the snippet below.

```
{ fid_t fid = PL_open_foreign_frame();

  ...
  if ( !PL_unify(t1, t2) )
    PL_rewind_foreign_frame(fid);
  ...

  PL_close_foreign_frame(fid);
}
```

In addition, `PL_unify()` may have failed on an **exception**, typically a resource (stack) overflow. This can be tested using `PL_exception()`, passing 0 (zero) for the query-id argument. Foreign functions that encounter an exception must return `FAIL` to Prolog as soon as possible or call `PL_clear_exception()` if they wish to ignore the exception.

`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_bool**(*term_t ?t, int a*)
: Unify *t* with either `true` or `false`.

`int` **PL_unify_chars**(*term_t ?t, int flags, size_t len, const char *chars*)
: New function to 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*. A Prolog type is one of `PL_ATOM`, `PL_STRING`, `PL_CODE_LIST` or `PL_CHAR_LIST`. A representation is one of `REP_ISO_LATIN_1`, `REP_UTF8` or `REP_MB`. See `PL_get_chars()` for a definition of the representation types. If *len* is `-1` *chars* must be zero-terminated and the length is computed from *chars* using strlen().

  If *flags* includes `PL_DIFF_LIST` and type is one of `PL_CODE_LIST` or `PL_CHAR_LIST`, the text is converted to a *difference list*. The tail of the difference list is $t + 1$.

`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()`.

`int` **PL_unify_integer**(*term_t ?t, intptr_t 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_uint64**(*term_t ?t, uint64_t n*)
Unify *t* with a Prolog integer from *n*. Note that unbounded integer support is required if *n* does not fit in a *signed* `int64_t`. If unbounded integers are not supported a `representation_error` is raised.

`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 fail. Note that this function does not create a term if the argument is already instantiated. If *f* is a functor with arity 0, *t* is unified with an atom. See also `PL_unify_compound()`.

`int` **PL_unify_compound**(*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 fail. Note that this function does not create a term if the argument is already instantiated. If *f* is a functor with arity 0, *t* is unified with compound without arguments. See also `PL_unify_functor()`.

`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 into *h* and a reference to the tail of the list into *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 are 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`. That is, 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. Similarly, 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` *size_t, 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 *size_t, const wchar_t **
: Create an atom from a length and a wide character pointer.

PL_NWCODES *size_t, const wchar_t **
: Create a list of character codes from a length and a wide character pointer.

PL_NWSTRING *size_t, 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_INTEGER *long*
: Unify the argument with an integer, as in `PL_unify_integer()`.

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_INT64 *int64_t*
: Unify the argument with a 64-bit integer, as in `PL_unify_int64()`.

PL_INTPTR *intptr_t*
: Unify the argument with an integer with the same width as a pointer. On most machines this is the same as `PL_LONG`. but on 64-bit MS-Windows pointers are 64 bits while longs are only 32 bits.

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 be 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 remaining 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 builds 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_chars_to_term(goal, g) )
    rval = PL_call(goal, NULL);
  else
    rval = FALSE;

  PL_discard_foreign_frame(fid);
  return rval;
}
  ...
  call_chars("consult(load)");
  ...
```

`PL_chars_to_term()` is defined using `PL_put_term_from_chars()` which can deal with not null-terminated strings as well as strings using different encodings:

```
int
PL_chars_to_term(const char *s, term_t t)
{ return PL_put_term_from_chars(t, REP_ISO_LATIN_1, (size_t)-1, s);
}
```

`int` **PL_wchars_to_term**(*const pl_wchar_t *chars, term_t -t*)
Wide character version of `PL_chars_to_term()`.

`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_STACK` 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_escapes` Prolog flag.

12.4.6 Convenient functions to generate Prolog exceptions

The typical implementation of a foreign predicate first uses the PL_get_*() functions to extract C data types from the Prolog terms. Failure of any of these functions is normally because the Prolog term is of the wrong type. The *_ex() family of functions are wrappers around (mostly) the PL_get_*() functions, such that we can write code in the style below and get proper exceptions if an argument is uninstantiated or of the wrong type.

```
/** set_size(+Name:atom, +Width:int, +Height:int) is det.

static foreign_t
set_size(term_t name, term_t width, term_t height)
{ char *n;
  int w, h;

  if ( !PL_get_chars(name, &n, CVT_ATOM|CVT_EXCEPTION) ||
```

```
        !PL_get_integer_ex(with, &w) ||
        !PL_get_integer_ex(height, &h) )
    return FALSE;

  ...

}
```

int **PL_get_atom_ex**(*term_t t, atom_t *a*)
As `PL_get_atom()`, but raises a type or instantiation error if *t* is not an atom.

int **PL_get_integer_ex**(*term_t t, int *i*)
As `PL_get_integer()`, but raises a type or instantiation error if *t* is not an integer, or a representation error if the Prolog integer does not fit in a C `int`.

int **PL_get_long_ex**(*term_t t, long *i*)
As `PL_get_long()`, but raises a type or instantiation error if *t* is not an atom, or a representation error if the Prolog integer does not fit in a C `long`.

int **PL_get_int64_ex**(*term_t t, int64_t *i*)
As `PL_get_int64()`, but raises a type or instantiation error if *t* is not an integer, or a representation error if the Prolog integer does not fit in a C `int64_t`.

int **PL_get_uint64_ex**(*term_t t, uint64_t *i*)
As `PL_get_uint64()`, but raises a type, domain or instantiation error if *t* is not an integer or *t* is less than zero, or a representation error if the Prolog integer does not fit in a C `int64_t`.

int **PL_get_intptr_ex**(*term_t t, intptr_t *i*)
As `PL_get_intptr()`, but raises a type or instantiation error if *t* is not an atom, or a representation error if the Prolog integer does not fit in a C `intptr_t`.

int **PL_get_size_ex**(*term_t t, size_t *i*)
As `PL_get_size()`, but raises a type or instantiation error if *t* is not an atom, or a representation error if the Prolog integer does not fit in a C `size_t`.

int **PL_get_bool_ex**(*term_t t, int *i*)
As `PL_get_bool()`, but raises a type or instantiation error if *t* is not an boolean.

int **PL_get_float_ex**(*term_t t, double *f*)
As `PL_get_float()`, but raises a type or instantiation error if *t* is not a float.

int **PL_get_char_ex**(*term_t t, int *p, int eof*)
Get a character code from *t*, where *t* is either an integer or an atom with length one. If *eof* is `TRUE` and *t* is -1, *p* is filled with -1. Raises an appropriate error if the conversion is not possible.

int **PL_get_pointer_ex**(*term_t t, void **addrp*)
As `PL_get_pointer()`, but raises a type or instantiation error if *t* is not a pointer.

`int` **PL_get_list_ex**(*term_t l, term_t h, term_t t*)
As `PL_get_list()`, but raises a type or instantiation error if *t* is not a list.

`int` **PL_get_nil_ex**(*term_t l*)
As `PL_get_nil()`, but raises a type or instantiation error if *t* is not the empty list.

`int` **PL_unify_list_ex**(*term_t l, term_t h, term_t t*)
As `PL_unify_list()`, but raises a type error if *t* is not a variable, list-cell or the empty list.

`int` **PL_unify_nil_ex**(*term_t l*)
As `PL_unify_nil()`, but raises a type error if *t* is not a variable, list-cell or the empty list.

`int` **PL_unify_bool_ex**(*term_t t, int val*)
As `PL_unify_bool()`, but raises a type error if *t* is not a variable or a boolean.

The second family of functions in this section simplifies the generation of ISO compatible error terms. Any foreign function that calls this function must return to Prolog with the return code of the error function or the constant `FALSE`. If available, these error functions add the name of the calling predicate to the error context. See also `PL_raise_exception()`.

`int` **PL_instantiation_error**(*term_t culprit*)
Raise `instantiation_error`. *Culprit* is ignored, but should be bound to the term that is insufficiently instantiated. See `instantiation_error/1`.

`int` **PL_uninstantiation_error**(*term_t culprit*)
Raise `uninstantiation_error(culprit)`. This should be called if an argument that must be unbound at entry is bound to *culprit*. This error is typically raised for a pure output arguments such as a newly created stream handle (e.g., the third argument of `open/3`).

`int` **PL_representation_error**(*const char *resource*)
Raise `representation_error(resource)`. See `representation_error/1`.

`int` **PL_type_error**(*const char *expected, term_t culprit*)
Raise `type_error(expected, culprit)`. See `type_error/2`.

`int` **PL_domain_error**(*const char *expected, term_t culprit*)
Raise `domain_error(expected, culprit)`. See `domain_error/2`.

`int` **PL_existence_error**(*const char *type, term_t culprit*)
Raise `existence_error(type, culprit)`. See `type_error/2`.

`int` **PL_permission_error**(*const char *operation, const char *type, term_t culprit*)
Raise `permission_error(operation, type, culprit)`. See `permission_error/3`.

`int` **PL_resource_error**(*const char *resource*)
Raise `resource_error(resource)`. See `resource_error/1`.

`int` **PL_syntax_error**(*const char *message, IOSTREAM *in*)
Raise `syntax_error(message)`. If *arg* is not `NULL`, add information about the current position of the input stream.

12.4.7 Serializing and deserializing Prolog terms

int **PL_put_term_from_chars**(*term_t t, int flags, size_t len, const char *s*)

Parse the text from the C-string *s* holding *len* bytes and put the resulting term in *t*. *len* can be `(size_t)-1`, assuming a 0-terminated string. The *flags* argument controls the encoding and is currently one of `REP_UTF8` (string is UTF8 encoded), `REP_MB` (string is encoded in the current locale) or 0 (string is encoded in ISO latin 1). The string may, but is not required, to be closed by a full stop (.).

If parsing produces an exception the behaviour depends on the `CVT_EXCEPTION` flag. If present, the exception is propagated into the environment. Otherwise, the exception is placed in *t* and the return value is `FALSE`.[6].

12.4.8 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 below. The structure contains additional fields at the ... for internal bookkeeping as well as future extensions.

```
typedef struct PL_blob_t
{ uintptr_t     magic;          /* PL_BLOB_MAGIC */
  uintptr_t     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);
  void          (*acquire)(atom_t a);
  ...
} 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:

[6] The `CVT_EXCEPTION` was added in version 8.3.12

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 fields 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 than *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* 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 vice 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, size_t 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, size_t 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, size_t *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, size_t *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 12.4.8.

12.4.9 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>`. Foreign code using GMP linked to SWI-Prolog asks for some considerations.

- SWI-Prolog normally rebinds the GMP allocation functions using mp_set_memory_functions(). This means SWI-Prolog must be initialised before the foreign code touches any GMP function. You can call `PL_action(PL_GMP_SET_ALLOC_FUNCTIONS, TRUE)` to force Prolog's GMP initialization without doing the rest of the Prolog initialization. If you do not want Prolog rebinding the GMP allocation, call `PL_action(PL_GMP_SET_ALLOC_FUNCTIONS, FALSE)` *before* initializing Prolog.

- On Windows, each DLL has its own memory pool. To make exchange of GMP numbers between Prolog and foreign code possible you must either let Prolog rebind the allocation functions (default) or you must recompile SWI-Prolog to link to a DLL version of the GMP library.

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 *normalised* 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.

12.4.10 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 then `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 be defined or not, redefined, etc., a Prolog predicate has a handle that is neither 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 as `PL_pred()`, but provides a more convenient interface to the C programmer.

`void` **PL_predicate_info**(*predicate_t p, atom_t *n, size_t *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. Any of the arguments *n*, *a* and *m* can be `NULL`.

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 that 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. *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 *meta-predicates*. See `meta_predicate/1`, `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.[7]

`PL_Q_NODEBUG`
: Switch off the debugger while executing the goal. This option is used by many calls to hook-predicates to avoid tracing the hooks. An example is `print/1` calling `portray/1` from foreign code.

`PL_Q_CATCH_EXCEPTION`
: If an exception is raised while executing the goal, do not report it, but make it available for `PL_exception()`.

`PL_Q_PASS_EXCEPTION`
: As `PL_Q_CATCH_EXCEPTION`, but do not invalidate the exception-term while calling `PL_close_query()`. This option is experimental.

`PL_Q_ALLOW_YIELD`
: Support the `I_YIELD` instruction for engine-based coroutining. See `$engine_yield/2` in `boot/init.pl` for details.

`PL_Q_EXT_STATUS`
: Make `PL_next_solution()` return extended status. Instead of only `TRUE` or `FALSE` extended status as illustrated in the following table:

Extended	**Normal**	
PL_S_EXCEPTION	FALSE	Exception available through `PL_exception()`
PL_S_FALSE	FALSE	Query failed
PL_S_TRUE	TRUE	Query succeeded with choicepoint
PL_S_LAST	TRUE	Query succeeded without choicepoint

`PL_open_query()` can return the query identifier '0' if there is not enough space on the environment stack. This function succeeds, even if the referenced predicate is not defined. In this case, running the query using `PL_next_solution()` will return an existence_error. See `PL_exception()`.

The example below opens a query to the predicate `is_a/2` to find the ancestor of 'me'. The reference to the predicate is valid for the duration of the process and may be cached by the client.

```
char *
ancestor(const char *me)
{ term_t a0 = PL_new_term_refs(2);
  static predicate_t p;
```

[7]Do not pass the integer 0 for normal operation, as this is interpreted as `PL_Q_NODEBUG` for backward compatibility reasons.

```
  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.

int **PL_cut_query**(*qid_t qid*)

Discards the query, but does not delete any of the data created by the query. It just invalidates *qid*, allowing for a new call to `PL_open_query()` in this context. `PL_cut_query()` may invoke cleanup handlers (see `setup_call_cleanup/3`) and therefore may experience exceptions. If an exception occurs the return value is `FALSE` and the exception is accessible through `PL_exception(0)`.

int **PL_close_query**(*qid_t qid*)

As `PL_cut_query()`, but all data and bindings created by the query are destroyed.

qid_t **PL_current_query**(*void*)

Returns the query id of of the current query or `0` if the current thread is not executing any queries.

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 t, module_t m*)

Call term *t* just like the Prolog predicate `once/1`. *t* is called in the module *m*, or in the context module if *m* == NULL. Returns `TRUE` if the call succeeds, `FALSE` otherwise. Figure 12.3 shows an example to obtain the number of defined atoms. All checks are omitted to improve readability.

12.4.11 Discarding Data

The Prolog data created and term references needed to set up the call and/or analyse the result can in most cases be discarded right after the call. `PL_close_query()` allows for destroying the data, while leaving the term references. The calls below may be used to destroy term references and data. See figure 12.3 for an example.

fid_t **PL_open_foreign_frame**()

Create 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.

```
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 12.3: Calling Prolog

`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 do not pop the frame. That is, the same frame can be rewound 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.

12.4.12 String buffering

Many of the functions of the foreign language interface involve strings. Some of these strings point into static memory like those associated with atoms. These strings are valid as long as the atom is protected against atom garbage collection, which generally implies the atom must be locked using `PL_register_atom()` or be part of an accessible term. Other strings are more volatile. Several functions provide a BUF_* flag that can be set to either `BUF_STACK` (default) or `BUF_MALLOC`. Strings returned by a function accepting `BUF_MALLOC` **must** be freed using `PL_free()`. Strings returned using BUF_STACK are pushed on a stack that is cleared when a foreign predicate returns control back to Prolog. More fine grained control may be needed if functions that return strings are called outside the context of a foreign predicate or a foreign predicate creates many strings during its execution. Temporary strings are scoped using these macros:

void **PL_STRINGS_MARK()**
void **PL_STRINGS_RELEASE()**

These macros must be paired and create a C *block* ({...}). Any string created using BUF_STACK after PL_STRINGS_MARK() is released by the corresponding PL_STRINGS_RELEASE(). These macros should be used like below

```
...
PL_STRINGS_MARK();
<operations involving strings>
PL_STRINGS_RELEASE();
...
```

The Prolog flag string_stack_tripwire may be used to set a *tripwire* to help finding places where scoping strings may help reducing resources.

12.4.13 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 innermost module is returned via *module* *. If *raw* is not a module construct, *raw* will simply be put in *plain*. The value pointed to by *m* must be initialized before calling PL_strip_module(), either to the default module or to NULL. A NULL value is replaced by the current context module if *raw* carries no module. 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 module*)

Return the name of *module* as an atom.

module_t **PL_new_module**(*atom_t name*)

Find an existing module or create a new module with the name *name*.

12.4.14 Prolog exceptions in foreign code

This section discusses `PL_exception()` and `PL_raise_exception()`, the interface functions to detect and generate Prolog exceptions from C code. `PL_raise_exception()` from the C interface registers the exception term and returns `FALSE`. If a foreign predicate returns `FALSE`, while an exception term is registered, a Prolog exception will be raised by the virtual machine. This implies for a foreign function that implements a predicate and wishes to raise an exception, the function shall call `PL_raise_exception()`, perform any necessary cleanup and return the return code of `PL_raise_exception()` or explicitly `FALSE`. Calling `PL_raise_exception()` outside the context of a function implementing a foreign predicate results in undefined behaviour.

Note that many of the C API functions may call `PL_raise_exception()` and return `FALSE`. The user shall test for this, cleanup and make the foreign function return `FALSE`.

`PL_exception()` may be used to inspect the currently registered exception. It is normally called after a call to `PL_next_solution()` returns `FALSE`, and returns a term reference to an exception term if an exception is pedning, and `(term_t)0` otherwise. It may also be called after, e.g., `PL_unify()` to distinguish a normal failing unification from a unification that raised an resource error exception.

If a C function implementing a predicate that calls Prolog should use `PL_open_query()` with the flag `PL_Q_PASS_EXCEPTION` and make the function return FALSE if `PL_next_solution()` returns `FALSE` and `PL_exception()` indicates an exception is pending.

Both for C functions implementing a predicate and when Prolog is called while the main control of the process is in C, user code should always check for exceptions. As explained above, C functions implementing a predicate should normally cleanup and return with `FALSE`. If the C function whishes to continue it may call `PL_clear_exception()`. Note that this may cause any exeption to be ignored, including *time outs* and *abort*. Typically the user should check the exeption details before ignoring an exception. If the C code does not implement a predicate it normally prints the exception and calls `PL_clear_exception()` to discard it. Exceptions may be printed by calling `print_message/2` through the C interface.

`int` **PL_raise_exception**(*term_t exception*)

Generate an exception (as `throw/1`) and return `FALSE`. This function is rarely used directly. Instead, errors are typically raised using the functions in section 12.4.6 or the C api functions that end in `_ex` such as `PL_get_atom_ex()`. Below we give an example returning an exception from a foreign predicate the verbose way. Note that the exception is raised in a sequence of actions connected using `&&`. This ensures that a proper exception is raised should any of the calls used to build or raise the exception themselves raise an exception. In this simple case `PL_new_term_ref()` is guaranteed to succeed because the system guarantees at least 10 available term references before entering the foreign predicate. `PL_unify_term()` however may raise a resource exception for the global stack.

```
foreign_t
pl_hello(term_t to)
{ char *s;

  if ( PL_get_atom_chars(to, &s) )
  { Sprintf("Hello \"%s\"\n", s);

```

```
    return TRUE;
  } else
  { term_t except;

    return  ( (except=PL_new_term_ref()) &&
              PL_unify_term(except,
                            PL_FUNCTOR_CHARS, "type_error", 2,
                              PL_CHARS, "atom",
                              PL_TERM, to) &&
              PL_raise_exception(except) );
  }
}
```

For reference, the preferred implementation of the above is below. The `CVT_EXCEPTION` tells the system to generate an exception if the conversion fails. The other `CVT_` flags define the admissible types and `REP_MB` requests the string to be provided in the current *locale* representation. This implies that Unicode text is printed correctly if the current environment can represent it. If not, a `representation_error` is raised.

```
foreign_t
pl_hello(term_t to)
{ char *s;

  if ( PL_get_chars(to, &s, CVT_ATOM|CVT_STRING|CVT_EXCEPTION|REP_MB) )
  { Sprintf("Hello \"%s\"\n", s);

    return TRUE;
  }

  return FALSE;
}
```

`int` **PL_throw**(*term_t exception*)

Similar to `PL_raise_exception()`, but returns using the C longjmp() function to the innermost `PL_next_solution()`. This function is deprecated as it does not provide the opportunity to cleanup.

`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 `(term_t)0` if the Prolog goal simply failed. If there is an exception, `PL_exception()` returns a term reference that contains the exception term.

Additionally, `PL_exception(0)` returns the pending exception in the current query or `(term_t)0` if no exception is pending. This can be used to check the error status after a failing call to, e.g., one of the unification functions.

`void` **PL_clear_exception**(*void*)
Tells Prolog that the encountered exception must be ignored. This function must be called if control remains in C after a previous API call fails with an exception.[8]

12.4.15 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 4.12. 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.

Any handler that wishes to call one of the Prolog interface functions should call `PL_sigaction()` to install the handler. `PL_signal()` provides a deprecated interface that is notably not capable of properly restoring the old signal status if the signal was previously handled by Prolog.

`int` **PL_sigaction**(*int sig, pl_sigaction_t *act, pl_sigaction_t *oldact*)
Install or query the status for signal *sig*. The signal is an integer between 1 and 64, where the where the signals up to 32 are mapped to OS signals and signals above that are handled by Prolog's synchronous signal handling. The `pl_sigaction_t` is a struct with the following definition:

```
typedef struct pl_sigaction
{ void         (*sa_cfunction)(int);     /* traditional C function */
  predicate_t sa_predicate;              /* call a predicate */
  int          sa_flags;                 /* additional flags */
} pl_sigaction_t;
```

The `sa_flags` is a bitwise or of `PLSIG_THROW`, `PLSIG_SYNC` and `PLSIG_NOFRAME`. Signal handling is enabled if `PLSIG_THROW` is provided, `sa_cfunction` or `sa_predicate` is provided. `sa_predicate` is a predicate handle for a predicate with arity 1. If no action is provided the signal handling for this signal is restored to the default before `PL_initialise()` was called.

Finally, 0 (zero) may be passed for *sig*. In that case the system allocates a free signal in the *Prolog range* (32...64). Such signal handler are activated using `PL_thread_raise()`.

`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.

[8] This feature is non-portable. Other Prolog systems (e.g., YAP) have no facilities to ignore raised exceptions, and the design of YAP's exception handling does not support such a facility.

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 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`.[9]

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.

`int` **PL_get_signum_ex**(*term_t t, int *sig*)
Extract a signal specification from a Prolog term and store as an integer signal number in *sig*. The specification is an integer, a lowercase signal name without `SIG` or the full signal name. These refer to the same: `9`, `kill` and `SIGKILL`. Leaves a typed, domain or instantiation error if the conversion fails.

12.4.16 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.

[9] A better default would be to use synchronous handling, but this interface preserves backward compatibility.

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.

`record_t` **PL_duplicate_record**(*record_t record*)
Return a duplicate of *record*. As records are read-only objects this function merely increments the records reference count.

`int` **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. Returns `TRUE` on success, and `FALSE` if there is not enough space on the stack to accommodate the term. 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 and 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 zero 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, size_t *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.

Database

`int` **PL_assert**(*term_t t, module_t m, int flags*)
Provides direct access to `asserta/1` and `assertz/1` by asserting *t* into the database in the module *m*. Defined flags are:

PL_ASSERTZ
Add the new clause as last. Calls `assertz/1`. This macros is defined as 0 and thus the default.

PL_ASSERTA
Add the new clause as first. Calls `asserta/1`.

PL_CREATE_THREAD_LOCAL
If the predicate is not defined, create it as thread-local. See `thread_local/1`.

PL_CREATE_INCREMENTAL
If the predicate is not defined, create it as *incremental* see `table/1` and section 7.7.

On success this function returns `TRUE`. On failure `FALSE` is returned and an exception is left in the environment that describes the reason of failure. See `PL_exception()`.

This predicate bypasses creating a Prolog callback environment and is faster than setting up a call to `assertz/1`. It may be used together with `PL_chars_to_term()`, but the typical use case will create a number of clauses for the same predicate. The fastest way to achieve this is by creating a term that represents the invariable structure of the desired clauses using variables for the variable sub terms. Now we can loop over the data, binding the variables, asserting the term and undoing the bindings. Below is an example loading words from a file that contains a word per line.

```
#include <SWI-Prolog.h>
#include <stdio.h>
#include <string.h>

#define MAXWLEN 256

static foreign_t
load_words(term_t name)
{ char *fn;

  if ( PL_get_file_name(name, &fn, PL_FILE_READ) )
  { FILE *fd = fopen(fn, "r");
    char word[MAXWLEN];
    module_t m = PL_new_module(PL_new_atom("words"));
    term_t cl = PL_new_term_ref();
    term_t w  = PL_new_term_ref();
    fid_t fid;

    if ( !PL_unify_term(cl, PL_FUNCTOR_CHARS, "word", 1, PL_TERM, w) )
      return FALSE;

    if ( (fid = PL_open_foreign_frame()) )
    { while(fgets(word, sizeof(word), fd))
      { size_t len;

        if ( (len=strlen(word)) )
        { word[len-1] = '\0';
          if ( !PL_unify_chars(w, PL_ATOM|REP_MB, (size_t)-1, word) ||
               !PL_assert(cl, m, 0) )
            return FALSE;
          PL_rewind_foreign_frame(fid);
        }
      }

      PL_close_foreign_frame(fid);
    }

    fclose(fd);
    return TRUE;
  }

  return FALSE;
}

install_t
install(void)
```

```
{ PL_register_foreign("load_words", 1, load_words, 0);
}
```

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 buffer stack described with the `PL_get_chars()` option `BUF_STACK`. 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 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.

`int` **PL_get_file_nameW**(*term_t spec, wchar_t **name, int flags*)
Same as `PL_get_file_name()`, but returns the filename as a wide-character string. This is intended for Windows to access the Unicode version of the Win32 API. Note that the flag `PL_FILE_OSPATH` must be provided to fetch a filename in OS native (e.g., `C:\x\y`) notation.

Dealing with Prolog flags from C

Foreign code can set or create Prolog flags using `PL_set_prolog_flag()`. See `set_prolog_flag/2` and `create_prolog_flag/3`. To retrieve the value of a flag you can use `PL_current_prolog_flag()`.

`int` **PL_set_prolog_flag**(*const char *name, int type, ...*)
Set/create a Prolog flag from C. *name* is the name of the affected flag. *type* is one of the values below, which also dictates the type of the final argument. The function returns `TRUE` on success and `FALSE` on failure. This function can be called *before* `PL_initialise()`, making the flag available to the Prolog startup code.

`PL_BOOL`
Create a boolean (`true` or `false`) flag. The argument must be an `int`.

`PL_ATOM`
Create a flag with an atom as value. The argument must be of type `const char *`.

`PL_INTEGER`
Create a flag with an integer as value. The argument must be of type `intptr_t *`.

`int` **PL_current_prolog_flag**(*atom_t name, int type, void *value*)
Retrieve the value of a Prolog flag from C. *name* is the name of the flag as an `atom_t` (see `current_prolog_flag/2`). *type* specifies the kind of value to be retrieved, it is one of the values below. *value* is a pointer to a location where to store the value. The user is responsible for making sure this memory location is of the appropriate size/type (see the returned types below to determine the size/type). The function returns `TRUE` on success and `FALSE` on failure.

`PL_ATOM`
Retrieve a flag whose value is an `atom`. The returned value is an atom handle of type `atom_t`.

`PL_INTEGER`
Retrieve a flag whose value is an `integer`. The returned value is an integer of type `int64_t`.

`PL_FLOAT`
Retrieve a flag whose value is a `float`. The returned value is a floating point number of type `double`.

`PL_TERM`
Retrieve a flag whose value is a `term`. The returned value is a term handle of type `term_t`.

12.4.17 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 4.10.

`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`.

12.4.18 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 below:

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
Windows: Used to indicate to 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 otherwise simply prints the error and exits.

PL_ACTION_TRADITIONAL
Same effect as using `--traditional`. Must be called *before* `PL_initialise()`.

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`.

PL_GMP_SET_ALLOC_FUNCTIONS
Takes an integer argument. If `TRUE`, the GMP allocations are immediately bound to the Prolog functions. If `FALSE`, SWI-Prolog will never rebind the GMP allocation functions. See mp_set_memory_functions() in the GMP documentation. The action returns `FALSE` if there is no GMP support or GMP is already initialised.

`unsigned int` **PL_version**(*int key*)

Query version information. This function may be called before `PL_initialise()`. If the key is unknown the function returns 0. See section 2.22 for a more in-depth discussion on binary compatibility. Defined keys are:

PL_VERSION_SYSTEM
SWI-Prolog version as $10,000 \times major + 100 \times minor + patch$.

PL_VERSION_FLI
Incremented if the foreign interface defined in this chapter changes in a way that breaks backward compatibility.

PL_VERSION_REC
Incremented if the binary representation of terms as used by `PL_record_external()` and `fast_write/2` changes.

PL_VERSION_QLF
Incremented if the QLF file format changes.

PL_VERSION_QLF_LOAD
Represents the oldest loadable QLF file format version.

PL_VERSION_VM
A hash that represents the VM instructions and their arguments.

PL_VERSION_BUILT_IN
A hash that represents the names, arities and properties of all built-in predicates defined in C. If this function is called before `PL_initialise()` it returns 0.

12.4.19 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.[10] The options are given in table 12.1.

12.4.20 Registering Foreign Predicates

`int` **PL_register_foreign_in_module**(*char *mod, char *name, int arity, foreign_t (*f)(), int flags, ...*)
Register the C function *f* 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 *mod*. If *mod* 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* form a bitwise *or*'ed list of options for the installation. These are:

`PL_FA_META`	Provide meta-predicate info (see below)
`PL_FA_TRANSPARENT`	Predicate is module transparent (deprecated)
`PL_FA_NONDETERMINISTIC`	Predicate is non-deterministic. See also `PL_retry()`.
`PL_FA_NOTRACE`	Predicate cannot be seen in the tracer
`PL_FA_VARARGS`	Use alternative calling convention.

If `PL_FA_META` is provided, `PL_register_foreign_in_module()` takes one extra argument. This argument is of type `const char*`. This string must be exactly as long as the number of arguments of the predicate and filled with characters from the set `0-9:^-+?`.

[10] Returning pointers and integers as a long is bad style. The signature of this function should be changed.

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_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 12.1: `PL_query()` options

See `meta_predicate/1` for details. `PL_FA_TRANSPARENT` is implied if at least one meta-argument is provided (`0-9:^`). Note that meta-arguments are *not always* passed as ⟨*module*⟩:⟨*term*⟩. Always use `PL_strip_module()` to extract the module and plain term from a meta-argument.[11]

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 12.4.1). 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;
```

[11]It is encouraged to pass an additional `NULL` pointer for non-meta-predicates.

```
    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.

12.4.21 Foreign Code Hooks

For various specific applications some hooks are 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**(*int (*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) and the *closure* argument passed to the `PL_on_halt()` call. Handlers *must* return 0. Other return values are reserved for future use. See also `at_halt/1`.[12] These handlers are called *before* system cleanup and can therefore access all normal Prolog resources. See also `PL_exit_hook()`.

void **PL_exit_hook**(*int (*f)(int, void *), void *closure*)

Similar to `PL_on_halt()`, but the hooks are executed by `PL_halt()` instead of `PL_cleanup()` just before calling exit().

[12]BUG: Although both `PL_on_halt()` and `at_halt/1` are called in FIFO order, *all* `at_halt/1` handlers are called before *all* `PL_on_halt()` handlers.

`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);
}
```

12.4.22 Storing foreign data

When combining foreign code with Prolog, it can be necessary to make data represented in the foreign language available to Prolog. For example, to pass it to another foreign function. At the end of this section, there is a partial implementation of using foreign functions to manage bit-vectors. Another example is the SGML/XML library that manages a 'parser' object, an object that represents the current state of the parser and that can be directed to perform actions such as parsing a document or make queries about the document content.

This section provides some hints for handling foreign data in Prolog. There are four options for storing such data:

- *Natural Prolog data*
  Uses the representation one would choose if no foreign interface was required. For example, a bitvector representing a list of small integers can be represented as a Prolog list of integers.

- *Opaque packed data on the stacks*
  It is possible to represent the raw binary representation of the foreign object as a Prolog string (see section 5.2). Strings may be created from foreign data using `PL_put_string_nchars()` and retrieved using `PL_get_string_chars()`. It is good practice to wrap the string in a compound term with arity 1, so Prolog can identify the type. The hook `portray/1` rules may be used to streamline printing such terms during development.

- *Opaque packed data in a blob*
  Similar to the above solution, binary data can be stored in an atom. The blob interface (section 12.4.8) provides additional facilities to assign a type and hook-functions that act on creation and destruction of the underlying atom.

- *Natural foreign data, passed as a pointer*
  An alternative is to pass a pointer to the foreign data. Again, the pointer is often 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 where Prolog never examines the contents of the data itself. If the data is opaque to Prolog, the selection will be driven solely by simplicity of the interface and performance.

- *What is the lifetime of the data*
  With 'lifetime' we refer to how it is decided that the object is (or can be) destroyed. We can distinguish three cases:

  1. The object must be destroyed on backtracking and normal Prolog garbage collection (i.e., it acts as a normal Prolog term). In this case, representing the object as a Prolog string (second option above) is the only feasible solution.
  2. The data must survive Prolog backtracking. This leaves two options. One is to represent the object using a pointer and use explicit creation and destruction, making the programmer responsible. The alternative is to use the blob-interface, leaving destruction to the (atom) garbage collector.
  3. The data lives as during the lifetime of a foreign function that implements a 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 12.4.1.

Examples for storing foreign data

In this section, we 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. Then, 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 using Prolog's unbounded integers.

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`, the 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 4.44) 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 use `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, nor to verify that 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.

12.4.23 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 with 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)
{ 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 `initialization/1` hooks and finally runs the initialization goals registered using `-g` *`goal`*.

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 `libswipl.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 12.5.
- Find the Prolog home directory. This process is described in detail in section 12.6.

`PL_initialise()` returns 1 if all initialisation succeeded and 0 otherwise.[13]

In most cases, *argc* and *argv* will be passed from the main program. It is allowed to create your own argument vector, provided `argv[0]` is constructed according to the rules above. For example:

```
int
main(int argc, char **argv)
{ char *av[10];
```

[13]BUG: Various fatal errors may cause `PL_initialise()` to call `PL_halt(1)`, preventing it from returning at all.

```
  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 `"libswipl.dll"` as `argv[0]`. If the Prolog state is attached to a DLL (see the `-dll` option of `swipl-ld`), pass the name of this DLL.

`int` **PL_winitialise**(*int argc, wchar_t **argv*)
Wide character version of `PL_initialise()`. Can be used in Windows combined with the wmain() entry point.

`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*.

`int` **PL_set_resource_db_mem**(*const unsigned char *data, size_t size*)
This function must be called at most once and *before* calling `PL_initialise()`. The memory area designated by *data* and *size* must contain the resource data and be in the format as produced by `qsave_program/2`. The memory area is accessed by `PL_initialise()` as well as calls to `open_resource/3`.[14]

For example, we can include the bootstrap data into an embedded executable using the steps below. The advantage of this approach is that it is fully supported by any OS and you obtain a single file executable.

1. Create a saved state using `qsave_program/2` or

```
% swipl -o state -c file.pl ...
```

2. Create a C source file from the state using e.g., the Unix utility `xxd`(1):

```
% xxd -i state > state.h
```

3. Embed Prolog as in the example below. Instead of calling the toplevel you probably want to call your application code.

[14]This implies that the data must remain accessible during the lifetime of the process if `open_resource/3` is used. Future versions may provide a function to detach the resource database and cause `open_resource/3` to raise an exception.

```
#include <SWI-Prolog.h>
#include "state.h"

int
main(int argc, char **argv)
{ if ( !PL_set_resource_db_mem(state, state_len) ||
       !PL_initialise(argc, argv) )
    PL_halt(1);

  return PL_toplevel();
}
```

Alternative to `xxd`, it is possible to use inline assembler, e.g. the `gcc` `incbin` instruction. Code for `gcc` was provided by Roberto Bagnara on the SWI-Prolog mailinglist. Given the state in a file `state`, create the following assembler program:

```
        .globl _state
        .globl _state_end
_state:
        .incbin "state"
_state_end:
```

Now include this as follows:

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

#if __linux
#define STATE _state
#define STATE_END _state_end
#else
#define STATE state
#define STATE_END state_end
#endif

extern unsigned char STATE[];
extern unsigned char STATE_END[];

int
main(int argc, char **argv)
{ if ( !PL_set_resource_db_mem(STATE, STATE_END - STATE) ||
       !PL_initialise(argc, argv) )
    PL_halt(1);
  return PL_toplevel();
}
```

As Jose Morales pointed at https://github.com/graphitemaster/incbin, which contains a portability layer on top of the above idea.

int **PL_toplevel**()
Runs the goal of the `-t` *toplevel* switch (default `prolog/0`) and returns 1 if successful, 0 otherwise.

int **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 if *status* equals '0' and restores all signal handlers. The *status* argument is passed to the various termination hooks and indicates the *exit-status*.

The function returns `TRUE` if successful and `FALSE` otherwise. Currently, `FALSE` is returned when an attempt is made to call `PL_cleanup()` recursively or if one of the exit handlers cancels the termination using `cancel_halt/1`. Exit handlers may only cancel termination if *status* is 0.

In theory, this function allows deleting and restarting the Prolog system in the same process. In practice, SWI-Prolog's cleanup process is far from complete, and trying to revive the system using `PL_initialise()` will leak memory in the best case. It can also crash the application.

In this state, there is little practical use for this function. If you want to use Prolog temporarily, consider running it in a separate process. If you want to be able to reset Prolog, your options are (again) a separate process, modules or threads.

void **PL_cleanup_fork**()
Stop intervaltimer that may be running on behalf of `profile/1`. The call is intended to be used in combination with fork():

```
if ( (pid=fork()) == 0 )
{ PL_cleanup_fork();
  <some exec variation>
}
```

The call behaves the same on Windows, though there is probably no meaningful application.

int **PL_halt**(*int status*)
Clean up the Prolog environment using `PL_cleanup()` and if successful call exit() with the status argument. Returns `FALSE` if exit was cancelled by `PL_cleanup()`.

Threading, Signals and embedded Prolog

This section applies to Unix-based environments that have signals or multithreading. The Windows version is compiled for multithreading, 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 VMs, however, are by nature multithreaded and appear to do signal handling (software interrupts).

On Unix systems, SWI-Prolog installs handlers for the following signals:

SIGUSR2 has an empty signal handler. This signal is sent to a thread after sending a thread-signal (see `thread_signal/2`). It causes blocking system calls to return with `EINTR`, which gives them the opportunity to react to thread-signals.

In some cases the embedded system and SWI-Prolog may both use `SIGUSR2` without conflict. If the embedded system redefines `SIGUSR2` with a handler that runs quickly and no harm is done in the embedded system due to spurious wakeup when initiated from Prolog, there is no problem. If SWI-Prolog is initialised *after* the embedded system it will call the handler set by the embedded system and the same conditions as above apply. SWI-Prolog's handler is a simple function only chaining a possibly previously registered handler. SWI-Prolog can handle spurious `SIGUSR2` signals.

SIGINT is used by the top level to activate the tracer (typically bound to control-C). The first control-C posts a request for starting the tracer in a safe, synchronous fashion. If control-C is hit again before the safe route is executed, it prompts the user whether or not a forced interrupt is desired.

SIGTERM, SIGABRT and SIGQUIT are caught to cleanup before killing the process again using the same signal.

SIGSEGV, SIGILL, SIGBUS, SIGFPE and SIGSYS are caught by to print a backtrace before killing the process again using the same signal.

SIGHUP is caught and causes the process to exit with status 2 after cleanup.

The `--no-signals` option can be used to inhibit all signal processing except for `SIGUSR2`. The handling of `SIGUSR2` is vital for dealing with blocking system call in threads. The used signal may be changed using the `--sigalert=NUM` option or disabled using `--sigalert=0`.

12.5 Linking embedded applications using swipl-ld

The utility program `swipl-ld` (Win32: swipl-ld.exe) may be used to link a combination of C files and Prolog files into a stand-alone executable. `swipl-ld` automates most of what is described in the previous sections.

In normal usage, a copy is made of the default embedding template `.../swipl/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, `swipl-ld` is typically invoked as:

```
swipl-ld -o output stubfile.c [other-c-or-o-files] [plfiles]
```

`swipl-ld` 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 accessible through the Prolog flag `c_cc` (Win32: `cl.exe`).

-c++ *C++-compiler*
: Compiler for C++ source file (extensions `.cpp`, `.cxx`, `.cc` or `.C`) 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 `swipl-ld`. 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 `swipl-ld` 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 `swipl-ld` 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` passes `-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, -lswipl (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. swipl-ld 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.

*
All other options. These are passed as linker options to the C compiler.

12.5.1 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 <string.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 12.4: 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 12.4.
The application is now created using the command line below. The option `-goal true` sets the Prolog initialization goal to suppress the banner. Note that the `-o calc` does not specify an extension. If the platform uses a file extension for executables, `swipl-ld` will add this (e.g., `.exe` on Windows).

```
% swipl-ld -goal true -o calc calc.c calc.pl
```

The created program `calc` is a native executable with the Prolog code attached to it. Note that the program typically depends on the shared object `libswipl` and, depending on the platform and configuration, on several external shared objects.

```
% ./calc pi/2
1.5708
```

12.6 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 12.5).

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 `--home=DIR` is provided, use this.
2. If the environment variable `SWI_HOME_DIR` is defined and points to an existing directory, use this.
3. If the environment variable `SWIPL` is defined and points to an existing directory, use this.
4. 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.
5. 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.

12.7 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 lowercase letters. Figure 12.5 shows the C source file, figure 12.6 illustrates compiling and loading of foreign code.

```
/*  Include file depends on local installation */
#include <SWI-Prolog.h>
#include <stdlib.h>
#include <string.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 12.5: Lowercase source file

```
% gcc -I/usr/local/lib/swipl-\plversion/include -fpic -c lowercase.c
% gcc -shared -o lowercase.so lowercase.o
% swipl
Welcome to SWI-Prolog (...)
...

1 ?- load_foreign_library(lowercase).
true.

2 ?- lowercase('Hello World!', L).
L = 'hello world!'.
```

Figure 12.6: Compiling the C source and loading the object file

12.8 Notes on Using Foreign Code

12.8.1 Foreign debugging functions

The functions in this section are primarily intended for debugging foreign extensions or embedded Prolog. Violating the constraints of the foreign interface often leads to crashes in a subsequent garbage collection. If this happens, the system needs to be compiled for debugging using `cmake -DCMAKE_BUILD_TYPE=Debug`, after which all functions and predicates listed below are available to use from the debugger (e.g. `gdb`) or can be placed at critical location in your code or the system code.

`void` **PL_backtrace**(*int depth, int flags*)

Dump a Prolog backtrace to the `user_error` stream. *Depth* is the number of frames to dump. *Flags* is a bitwise or of the following constants:

PL_BT_SAFE

(0x1) Do not try to print *goals*. Instead, just print the predicate name and arity. This reduces the likelihood to crash if `PL_backtrace()` is called in a damaged environment.

PL_BT_USER

(0x2) Only show 'user' frames. Default is to also show frames of hidden built-in predicates.

`char *` **PL_backtrace_string**(*int depth, int flags*)

As `PL_backtrace()`, but returns the stack as a string. The string uses UTF-8 encoding. The returned string must be freed using `PL_free()`. This function is was added to get stack traces from running servers where I/O is redirected or discarded. For example, using `gdb`, a stack trace is printed in the gdb console regardless of Prolog I/O redirection using the following command:

```
(gdb) printf "%s", PL_backtrace_string(25,0)
```

The source distribution provides the script `scripts/swipl-bt` that exploits `gdb` and `PL_backtrace_string()` to print stack traces in various formats for a SWI-Prolog process, given its process id.

`int` **PL_check_data**(*term_t data*)

Check the consistency of the term *data*. Returns `TRUE` this is actually implemented in the current version and `FALSE` otherwise. The actual implementation only exists if the system is compiled with the cflag `-DO_DEBUG` or `-DO_MAINTENANCE`. This is *not* the default.

`int` **PL_check_stacks**()

Check the consistency of the runtime stacks of the calling thread. Returns `TRUE` this is actually implemented in the current version and `FALSE` otherwise. The actual implementation only exists if the system is compiled with the cflag `-DO_DEBUG` or `-DO_MAINTENANCE`. This is *not* the default.

The Prolog kernel sources use the macro `DEBUG(`*Topic, Code*`)`. These macros are disabled in the production version and must be enabled by recompiling the system as described above. Specific topics can be enabled and disabled using the predicates `prolog_debug/1` and `prolog_nodebug/1`. In addition, they can be activated from the commandline using commandline option `-d topics`, where *topics* is a comma-separated list of debug topics to enable. For example, the code below adds many consistency checks and prints messages if the Prolog signal handler dispatches signals.

```
$ swipl -d chk_secure,msg_signal
```

prolog_debug(+*Topic***)**
prolog_nodebug(+*Topic***)**

Enable/disable a debug topic. *Topic* is an atom that identifies the desired topic. The available topics are defined in `src/pl-debug.h`. Please search the sources to find out what is actually printed and when. We highlight one topic here:

chk_secure(*A***)**

dd many expensive consistency checks to the system. This should typically be used when the system crashes, notably in the garbage collector. Garbage collection crashes are in most cases caused by invalid data on the Prolog stacks. This debug topic may help locating how the invalid data was created.

These predicates require the system to be compiled for debugging using `cmake -DCMAKE_BUILD_TYPE=Debug`.

`int` **PL_prolog_debug**(*const char *topic*)
`int` **PL_prolog_nodebug**(*const char *topic*)

(De)activate debug topics. The *topics* argument is a comma-separated string of topics to enable or disable. Matching is case-insensitive. See also `prolog_debug/1` and `prolog_nodebug/1`.

These functions require the system to be compiled for debugging using `cmake -DCMAKE_BUILD_TYPE=Debug`.

12.8.2 Memory Allocation

SWI-Prolog's heap memory allocation is based on the `malloc(3)` library routines. 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.

`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.

12.8.3 Compatibility between Prolog versions

Great care is taken to ensure binary compatibility of foreign extensions between different Prolog versions. Only the 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()`.

12.8.4 Foreign hash tables

As of SWI-Prolog 8.3.2 the foreign API provides access to the internal thread-safe and lock-free hash tables that associate pointers or objects that fit in a pointer such as atoms (`atom_t`). An argument against providing these functions is that they have little to do with Prolog. The argument in favor is that it is hard to implement efficient lock-free tables without low-level access to the underlying Prolog threads and exporting this interface has a low cost.

The functions below **can only be called if the calling thread is associated with a Prolog thread**. Failure to do so causes the call to be ignored or return the failure code where applicable.

`hash_table_t` **PL_new_hash_table**(*int size, void (*free_symbol)(void *n, void *v)*)
Create a new table for *size* key-value pairs. The table is resized when needed. If you know the table will hold 10,000 key-value pairs, providing a suitable initial size avoids resizing. The *free_symbol* function is called whenever a key-value pair is removed from the table. This can be `NULL`.

`int` **PL_free_hash_table**(*hash_table_t table*)
Destroy the hash table. First calls `PL_clear_hash_table()`.

`void*` **PL_lookup_hash_table**(*hash_table_t table, void *key*)
Return the value matching *key* or `NULL` if *key* does not appear in the table.

`void*` **PL_add_hash_table**(*hash_table_t table, void *key, void *value, int flags*)
Add *key-value* to the table. The behaviour if *key* is already in the table depends on *flags*. If `0`, this function returns the existing value without updating the table. If `PL_HT_UPDATE` the old *value* is *replaced* and the function returns the old value. If `PL_HT_NEW`, a message and backtrace are printed and the function returns *NULL* if *key* is already in the table.

`void*` **PL_del_hash_table**(*hash_table_t table, void *key*)
Delete *key* from the table, returning the old associated value or `NULL`

`int` **PL_clear_hash_table**(*hash_table_t table*)
Delete all key-value pairs from the table. Call *free_symbol* for each deleted pair.

`hash_table_enum_t` **PL_new_hash_table_enum**(*hash_table_t table*)

Return a table *enumerator* (cursor) that can be used to enumerate all key-value pairs using `PL_advance_hash_table_enum()`. The enumerator must be discarded using `PL_free_hash_table_enum()`. It is safe for another thread to add symbols while the table is being enumerated, but undefined whether or not these new symbols are visible. If another thread deletes a key that is not yet enumerated it will not be enumerated.

`void` **PL_free_hash_table_enum**(*hash_table_enum_t e*)

Discard an enumerator object created using `PL_new_hash_table_enum()`. Failure to do so causes the table to use more and more memory on subsequent modifications.

`int` **PL_advance_hash_table_enum**(*hash_table_enum_t e, void **key, void **value*)

Get the next key-value pair from a cursor.

12.8.5 Debugging and profiling foreign code (valgrind)

This section is only relevant for Unix users on platforms supported by valgrind. Valgrind is an excellent binary instrumentation platform. Unlike many other instrumentation platforms, valgrind can deal with code loaded through dlopen().

The `callgrind` tool can be used to profile foreign code loaded under SWI-Prolog. Compile the foreign library adding `-g` option to `gcc` or `swipl-ld`. By setting the environment variable `VALGRIND` to `yes`, SWI-Prolog will *not* release loaded shared objects using dlclose(). This trick is required to get source information on the loaded library. Without, valgrind claims that the shared object has no debugging information.[15] Here is the complete sequence using `bash` as login shell:

```
% VALGRIND=yes valgrind --tool=callgrind pl <args>
<prolog interaction>
% kcachegrind callgrind.out.<pid>
```

12.8.6 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 you 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.

[15] Tested using valgrind version 3.2.3 on x64.

12.8.7 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 use ⟨*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.

13 Deploying applications

This chapter describes the features of SWI-Prolog for delivering applications using *saved states*.

13.1 Deployment options

There are several ways to make a Prolog application available to your users. By far the easiest way is to require the user to install SWI-Prolog and deliver the application as a directory holding source files, other resources the application may need and a *Prolog Script* file that provides the executable. See section 2.11.2. The two-step installation may be slightly less convenient for the end user, but enables the end-user to conveniently run your program on a different operating system or architecture. This mechanism is obviously not suitable if you want to keep the source of your program secret.

Another solution is to use *saved states*, the main topic of this chapter, together with the installed development system and disable *autoloading* requirements into the state using `--no-autoload` or the `autoload`(*false*) option of `qsave_program/2`. This allows creating the application as a single file, while avoiding the need to ensure that the state is self-contained. For large programs this technique typically reduces startup time by an order of magnitude. This mechanism is particularly suitable for in-house and cloud deployment. It provides some protection against inspecting the source. See section 13.6 for details.

The final solution is to make sure all required resources are present in the saved state. In this case the state may be added to the *emulator* and the application consists of the emulator with state and the shared objects/DLLs required to make the emulator work. If the emulator can be statically linked for the target platform this creates a single file executable that does not require SWI-Prolog installed on the target computer.

13.2 Understanding saved states

A SWI-Prolog *saved state* is a *resource archive* that contains the compiled program in a machine-independent format,[1] startup options, optionally shared objects/DLLs and optionally additional *resource* files. As of version 7.7.13, the resource archive format is ZIP. A resource file is normally **created** using the commandline option `-c`:

```
swipl -o mystate option ... -c file.pl ...
```

The above causes SWI-Prolog to load the given Prolog files and call `qsave_program/2` using options created from the *option* ... in the command above.

[1]Although the compiled code is independent from the CPU and operating system, 32-bit compiled code does not run on the 64-bit emulator, nor the other way around. Conditionally compiled code (see `if/1`) may also reduce platform independence.

A saved state may be **executed** in several ways. The basic mechanism is to use the `-x`:

```
swipl -x mystate app-arg ...
```

Saved states may have an arbitrary payload at the *start*. This allows combining a (shell) script or the emulator with the state to turn the state into a single file executable. By default a state starts with a shell script (Unix) or the emulator (Windows).[2] The options `emulator`(*File*) and `stand_alone`(*Bool*) control what is added at the start of the state. Finally, C/C++ programs that embed Prolog may use a static C string that embeds the state into the executable. See `PL_set_resource_db_mem()`.

13.2.1 Creating a saved state

The predicates in this section support creating a saved state. Note that states are commonly created from the commandline using the `-c`, for example:

```
swipl -o mystate --foreign=save -c load.pl
```

Long (`--`) options are translated into options for `qsave_program/2`. This transformation uses the same conventions as used by `argv_options/3`, except that the transformation is guided by the option type. This implies that integer and callable options need to have valid syntax and boolean options may be abbreviated to simply `--autoload` or `--no-autoload` as shorthands for `--autoload=true` and `--autoload=false`.

qsave_program(+*File*, +*Options*)
Saves the current state of the program to the file *File*. The result is a resource archive *File* containing expresses all Prolog data from the running program, all user-defined resources (see `resource/2` and `open_resource/2`) and optionally all shared objects/DLLs required by the program for the current architecture. Depending on the `stand_alone` option, the resource is headed by the emulator, a Unix shell script or nothing. *Options* is a list of additional options:

stack_limit(+*Bytes*)
Sets default stack limit for the new process. See the command line option `--stack-limit` and the Prolog flag `stack_limit`.

goal(:*Callable*)
Initialization goal for the new executable (see `-g`). Two values have special meaning: `prolog` starts the Prolog toplevel and `default` runs `halt/0` if there are initialization goals and the `prolog/0` toplevel otherwise.

toplevel(:*Callable*)
Top-level goal for the new executable (see `-t`). Similar to `initialization/2` using `main`, the default toplevel is to enter the Prolog interactive shell unless a goal has been specified using `goal`(*Callable*).

init_file(+*Atom*)
Default initialization file for the new executable. See `-f`.

[2] As the default emulator is a short program while the true emulator is in a DLL this keeps the state short.

class(*+Class*)
If `runtime` (default), read resources from the state and disconnect the code loaded into the state from the original source. If `development`, save the predicates in their current state and keep reading resources from their source (if present). See also `open_resource/3`.

autoload(*+Boolean*)
If `true` (default), run `autoload/0` first. If the class is `runtime` and `autoload` is `true`, the state is supposed to be self contained and autoloading is disabled in the restored state.

map(*+File*)
Dump a human-readable trace of what has been saved in *File*.

op(*+Action*)
One of `save` (default) to save the current operator table or `standard` to use the initial table of the emulator.

stand_alone(*+Boolean*)
If `true`, the emulator is the first part of the state. If the emulator is started it tests whether a saved state is attached to itself and load this state. Provided the application has all libraries loaded, the resulting executable is completely independent from the runtime environment or location where it was built. See also section 2.11.2.

emulator(*+File*)
File to use for the emulator. Default is the running Prolog image.

foreign(*+Action*)
If `save`, include shared objects (DLLs) for the current architecture into the saved state. See `current_foreign_library/2`, and current_prolog_flag(arch, Arch). If the program `strip` is available, this is first used to reduce the size of the shared object. If a state is started, `use_foreign_library/1` first tries to locate the foreign resource in the resource database. When found it copies the content of the resource to a temporary file and loads it. If possible (Unix), the temporary object is deleted immediately after opening.[34]

If *Action* is of the form `arch`(*ListOfArches*) then the shared objects for the specified architectures are stored in the saved state. On the command line, the list of architectures can be passed as `--foreign=`⟨*CommaSepArchesList*⟩. In order to obtain the shared object file for the specified architectures, `qsave_program/2` calls a user defined hook: `qsave:arch_shlib`(*+Arch, +FileSpec, -SoPath*). This hook needs to unify `SoPath` with the absolute path to the shared object for the specified architecture. `FileSpec` is of the form `foreign(Name)`.

At runtime, SWI-Prolog will try to load the shared library which is compatible with the current architecture, obtained by calling `current_prolog_flag`(*arch, Arch*). An architecture is compatible if one of the two following conditions is true (tried in order):

1. There is a shared object in the saved state file which matches the current architecture name (from `current_prolog_flag/2`) exactly.

[3] This option is experimental and currently disabled by default. It will become the default if it proves robust.

[4] Creating a temporary file is the most portable way to load a shared object from a zip file but requires write access to the file system. Future versions may provide shortcuts for specific platforms that bypass the file system.

2. The user definable `qsave:compat_arch`(*Arch1, Arch2*) hook succeeds.

This last one is useful when one wants to produce one shared object file that works for multiple architectures, usually compiling for the lowest common denominator of a certain CPU type. For example, it is common to compile for armv7 if even if the code will be running on newer arm CPUs. It is also useful to provide highly-optimized shared objects for particular architectures.

undefined(+*Value*)
Defines what happens if an undefined predicate is found during the code analysis. Values are `ignore` (default) or `error`. In the latter case creating the state is aborted with a message that indicates the undefines predicates and from where they are called.

obfuscate(+*Boolean*)
If `true` (default `false`), replace predicate names with generated symbols to make the code harder to assess for reverse engineering. See section 13.6.1.

verbose(+*Boolean*)
If `true` (default `false`), report progress and status, notably regarding auto loading.

qsave_program(+*File*)
Equivalent to `qsave_program(File, [])`.

autoload_all
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.

This predicate is used by `qsave_program/[1,2]` to ensure the saved state does not depend on availability of the libraries. The predicate `autoload/0` examines all clauses of the loaded program (obtained with `clause/2`) and analyzes the body for referenced goals. Such an analysis cannot be complete in Prolog, which allows for the creation of arbitrary terms at runtime and the use of them as a goal. The current analysis is limited to the following:

- Direct goals appearing in the body
- Arguments of declared meta-predicates that are marked with an integer (0..9). See `meta_predicate/1`.

The analysis of meta-predicate arguments is limited to cases where the argument appears literally in the clause or is assigned using =/2 before the meta-call. That is, the following fragment is processed correctly:

```
        ...,
        Goal = prove(Theory),
        forall(current_theory(Theory),
               Goal)),
```

But, the calls to `prove_simple/1` and `prove_complex/1` in the example below are *not* discovered by the analysis and therefore the modules that define these predicates must be loaded explicitly using `use_module/[1,2]`.

```
        ...,
        member(Goal, [ prove_simple(Theory),
                       prove_complex(Theory)
                     ]),
        forall(current_theory(Theory),
               Goal)),
```

It is good practice to use `gxref/0` to make sure that the program has sufficient declarations such that the analysis tools can verify that all required predicates can be resolved and that all code is called. See `meta_predicate/1`, `dynamic/1`, `public/1` and `prolog:called_by/2`.

volatile *+Name/Arity, ...*
Declare that the clauses of specified predicates should **not** be saved to the program. The volatile declaration is normally used to prevent the clauses of dynamic predicates that represent data for the current session from being saved in the state file.

13.2.2 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 13.2.3 for details.
- If the program uses directives (`:- goal.` lines) that perform other actions than setting predicate attributes (`dynamic/1`, `volatile/1`, etc.) or loading files (`use_module/1`, etc.). Goals that need to be executed when the state is started must use `initialization/1` (ISO standard) or `initialization/2` (SWI extension that provides more control over when the goal is executed). For example, `initialization/2` can be used to start the application:

```
:- initialization(go, main).
```

- *Blobs* used as references to the database (see `clause/3`, `recorded/3`), streams, threads, etc. can not be saved. This implies that (dynamic) clauses may not contain such references at the moment the `qsave_program/2` is called. Note that the required foreign context (stream, etc.) cannot be present in the state anyway, making it pointless to save such references. An attempt to save such objects results in a warning.

  The `volatile/1` directive may be used to prevent saving the clauses of predicates that hold such references. The saved program must reinitialise such references using the normal program initialization techniques: use `initialization/1`,2 directives, explicitly create them by the entry point or make the various components recreate the context lazily when required.

13.2.3 Runtimes and Foreign Code

Many applications use packages that include foreign language components compiled to shared objects or DLLs. This code is normally loaded using `use_foreign_library/1` and the `foreign` file search path. Below is an example from the `socket` library.

```
:- use_foreign_library(foreign(socket)).
```

There are two options to handle shared objects in runtime applications. The first is to use the `foreign(`*save*`)` option of `qsave_program/2` or the `--foreign=save` commandline option. This causes the dependent shared objects to be included into the resource archive. The `use_foreign_library/1` directive first attempts to find the foreign file in the resource archive. Alternatively, the shared objects may be placed in a directory that is distributed with the application. In this cases the file search path `foreign` must be setup to point at this directory. For example, we can place the shared objects in the same directory as the executable using the definition below. This may be refined further by adding subdirectories depending on the architecture as available from the Prolog flag `arch`.

```
:- multifile user:file_search_path/2.

user:file_search_path(foreign, Dir) :-
    current_prolog_flag(executable, Exe),
    file_directory_name(Exe, Dir).
```

13.3 State initialization

The `initialization/1` and `initialization/2` directive may be used to register goals to be executed at various points in the life cycle of an executable. Alternatively, one may consider *lazy initialization* which typically follows the pattern below. Single threaded code can avoid using `with_mutex/2`.

```
:- dynamic x_done/0.
:- volatile x_done/0.

x(X) :-
    x_done,
    !,
    use_x(X).
x(X) :-
    with_mutex(x, create_x),
    use_x(X).

create_x :-
    x_done,
    !.
create_x :-
    <create x>
    asserta(x_done).
```

13.4 Using program resources

A *resource* is 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`) that acts as a *virtual file system* for the SWI-Prolog I/O predicates (see `open/4`, `register_iri_scheme/3`).

A resource has a *name*. The *source* data of a resource is a file. Resources are declared by adding clauses to the predicate `resource/2` or `resource/3`. Resources can be accessed from Prolog as files that start with `res://` or they can be opened using `open_resource/3`.

13.4.1 Resources as files

As of SWI-Prolog 7.7.13, resources that are compiled into the program can be accessed using the normal file handling predicates. Currently the following predicates transparently handle resources as read-only files:

- `open/3`, `open/4`
- `access_file/2`
- `exists_file/1`
- `exists_directory/1`
- `time_file/2`
- `size_file/2`

In addition, `open_shared_object/3`, underlying `use_foreign_library/1` handles *shared objects* or DLLs by copying them to a temporary file and opening this file. If the OS allows for it, the copied file is deleted immediately, otherwise it is deleted on program termination.

With the ability to open resources as if they were files we can use them for many tasks without changing the source code as required when using `open_resource/2`. Below we describe a typical scenario.

- Related resources are placed in one or more directories. Consider a web application where we have several directories holding icons. Add clauses to `file_search_path/2` that makes all icons accessible using the term `icon`(*file*).

- Add a clause as below before creating the state. This causes all icons to be become available as `res://app/icon/`*file*.

```
resource(app/icon, icon(.)).
```

- Add a clause to `file_search_path/2` that make the icons available from the resource data. For example using the code below.

```
:- asserta(user:file_search_path(icon, 'res://app/icon').
```

13.4.2 Access resources using open_resource

Before the system had the ability to open resources as files, resources were opened using the predicates `open_resource/2` or `open_resource/3`. These predicates provide somewhat better dynamic control over resources depending on whether the code is running from files or from a saved state. The main disadvantage is that having a separate open call requires rewriting code to make it work with resources rather than files.

open_resource(+*Name, -Stream***)**
open_resource(+*Name, -Stream, +Options***)**

Opens the resource specified by *Name*. If successful, *Stream* is unified with an input stream that provides access to the resource. The stream can be tuned using the *Options*, which is a subset of the options provided by `open/4`.

type(*Type***)**
encoding(*Encoding***)**
bom(*Bool***)**

Options that determine the binary/text type, encoding for text streams and whether or not the content should be checked for a BOM marker. The options have the same meaning as the corresponding options for `open/4`.

The predicate `open_resource/3` first checks `resource/2`. When successful it will open the returned resource source file. Otherwise it will look in the program's 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 affecting the running environment, while the runtime system quickly accesses the system resources.

13.4.3 Declaring resources

resource(*:Name, +FileSpec***)**
resource(*:Name, +FileSpec, +Options***)**

These predicates are defined as dynamic predicates in the module `user`. Clauses for them 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. *FileSpec* is a file specification that may exploit `file_search_path/2` (see `absolute_file_name/2`).

Often, 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.

If *FileSpec* points at a directory, the content of the directory is recursively added below *Name*. If *FileSpec* a term of the form `Alias`(*Name*), all directories that match this specification are enumerated and their content is added to the resource database. If an file appears in multiple results of this search path only the first file is added. Note that this is consistent with the normal behaviour where `absolute_file_name/3` returns the first match. The *Options* can be used to control what is saved from a directory.

include(+*Patterns*)
Only include a file from a directory if it matches at least one of the members of *Patterns*.

exclude(+*Patterns*)
Excludes a file from a directory if it matches at least one of the members of *Patterns*.

13.4.4 Managing resource files

As of version 7.7.13, SWI-Prolog resource files are zip(1) files. Prolog creates and accesses its resource files using the minizip project. The resource files may be examined and modified using any tool that can process zip files.

13.5 Debugging and updating deployed systems

SWI-Prolog provides several facilities to debug and update running (server) applications. The core to these facilities are:

- Hot-swap recompilation (section 4.3.2 and the library `hotswap`) allow, with some limitation, making modifications to running services. This includes adding debugging and logging statements.
- To make this useful some form of interaction is required. This can be implemented using signal handlers (Unix), specific HTTP services, generic HTTP services (e.g., SWISH) or networked interaction using the library `prolog_server` that allow interaction using netcat (`nc`) or `telnet`.

13.6 Protecting your code

Prolog in general, but SWI-Prolog in particular is an transparent environment. Prolog's "code is data" point of view makes this natural as it simplifies development and debugging. Some users though want or need to protect their code against copying or reverse engineering.

There are three ways to distribute code: as source, as `.qlf` file and in a saved state. Both QLF files and saved states contain the code as *virtual machine code*. QLF files capture the predicates and directives, while saved state capture the current state of the program. From the viewpoint of protecting code there is no significant difference.

There are two aspects to protection. One is to make sure the attacker has no access to the code in any format and the other is to provide access to a non-human-readable version of the code. The second approach is known as code obfuscation. Code obfuscation typically remove layout and comments and rename all internal identifiers. If an attacker gets access to the SWI-Prolog virtual machine code this can be *decompiled*. The decompiled code does not include layout information variable names and comments. Other identifiers, notably predicate and module names are maintained. This provides some protection against understanding the source as Prolog code without meaningful variable names and comments is generally hard to follow.

For further protecting the code, there are several scenarios.

- If the user has unrestricted access to the file system on which the application is installed the user can always access the state or QLF file. This data can be loaded into a compatible emulator and be *decompiled*.

- If the user can run arbitrary Prolog code or shell commands the state can be protected by embedding it as a string in the executable deny read access to the executable. This requires a small C program that includes the string and uses `PL_set_resource_db_mem()` to register the string as the resource database. See `PL_set_resource_db_mem()` for details. This protection should be combined with the `protect_static_code` described below.
- Some extra protection can be provided using the Prolog flag `protect_static_code`, which disables decompilation of *static* predicates. Note that most Prolog implementations cannot decompile static code. Various SWI-Prolog tools depend on this ability though. Examples are `list_undefined/0`, `autoload/0`, `show_coverage/1`, etc.

13.6.1 Obfuscating code in saved states

If the option `obfuscate`(*true*) is used with `qsave_program/2`, certain atoms in the saved state are renamed. The renaming is performed by library `obfuscate`. The current implementation is rather conservative, renaming atoms that are used only to define the functor that names a predicate. This is a safe operation, provided the application does not create new references to renamed predicates by reading additional source code or constructing the atom that names the predicate dynamically in some other way such as using `atom_concat/3`. Predicates that are called this way must be declared using `public/1`.

Note that more aggressive renaming is possible, but this requires more detailed analysis of the various roles played by some atom. Helpful and descriptive predicate names tend to be unique and are thus subject to this transformation. More general names tend to collide with other roles of the same atom and thus prevent renaming.

13.7 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 provides a flexible mechanism for locating runtime files.

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 explicitly or because trapping unknown predicates is disabled (see `unknown/2` and `current_prolog_flag/2`), library predicates must be loaded explicitly.
- Using libraries reduces the footprint of applications that don't need them.

The documentation of the library has 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 library(aggregate): Aggregation operators on backtrackable predicates

Compatibility Quintus, SICStus 4. The `forall/2` is a SWI-Prolog built-in and `term_variables/3` is a SWI-Prolog built-in with **different semantics**.

To be done
- Analysing the aggregation template and compiling a predicate for the list aggregation can be done at compile time.
- `aggregate_all/3` can be rewritten to run in constant space using non-backtrackable assignment on a term.

This library provides aggregating operators over the solutions of a predicate. The operations are a generalisation of the `bagof/3`, `setof/3` and `findall/3` built-in predicates. Aggregations that can be computed incrementally avoid `findall/3` and run in constant memory. The defined aggregation operations are counting, computing the sum, minimum, maximum, a bag of solutions and a set of solutions. We first give a simple example, computing the country with the smallest area:

```
smallest_country(Name, Area) :-
    aggregate(min(A, N), country(N, A), min(Area, Name)).
```

There are four aggregation predicates (`aggregate/3`, `aggregate/4`, `aggregate_all/3` and `aggregate/4`), distinguished on two properties.

aggregate vs. aggregate_all The aggregate predicates use `setof/3` (`aggregate/4`) or `bagof/3` (`aggregate/3`), dealing with existential qualified variables (`Var^Goal`) and providing multiple solutions for the remaining free variables in *Goal*. The `aggregate_all/3` predicate uses `findall/3`, implicitly qualifying all free variables and providing exactly one solution, while `aggregate_all/4` uses `sort/2` over solutions that Discriminator (see below) generated using `findall/3`.

The Discriminator argument The versions with 4 arguments deduplicate redundant solutions of Goal. Solutions for which both the template variables and Discriminator are identical will be treated as one solution. For example, if we wish to compute the total population of all countries, and for some reason `country(belgium, 11000000)` may succeed twice, we can use the following to avoid counting the population of Belgium twice:

```
    aggregate(sum(P), Name, country(Name, P), Total)
```

All aggregation predicates support the following operators below in Template. In addition, they allow for an arbitrary named compound term, where each of the arguments is a term from the list below. For example, the term `r(min(X), max(X))` computes both the minimum and maximum binding for X.

count
Count number of solutions. Same as `sum(1)`.

sum(*Expr*)
Sum of *Expr* for all solutions.

min(*Expr*)
Minimum of *Expr* for all solutions.

min(*Expr, Witness*)
A term `min(Min, Witness)`, where Min is the minimal version of *Expr* over all solutions, and *Witness* is any other template applied to solutions that produced Min. If multiple solutions provide the same minimum, *Witness* corresponds to the first solution.

max(*Expr*)
Maximum of *Expr* for all solutions.

max(*Expr, Witness*)
As `min(Expr, Witness)`, but producing the maximum result.

set(*X*)
An ordered set with all solutions for *X*.

bag(*X*)
A list of all solutions for *X*.

Acknowledgements

The development of this library was sponsored by SecuritEase, `http://www.securitease.com`

aggregate(*+Template, :Goal, -Result*) *[nondet]*

Aggregate bindings in *Goal* according to *Template*. The `aggregate/3` version performs `bagof/3` on *Goal*.

aggregate(*+Template, +Discriminator, :Goal, -Result*) *[nondet]*

Aggregate bindings in *Goal* according to *Template*. The `aggregate/4` version performs `setof/3` on *Goal*.

aggregate_all(*+Template, :Goal, -Result*) *[semidet]*

Aggregate bindings in *Goal* according to *Template*. The `aggregate_all/3` version performs `findall/3` on *Goal*. Note that this predicate fails if *Template* contains one or more of `min(X)`, `max(X)`, `min(X,Witness)` or `max(X,Witness)` and *Goal* has no solutions, i.e., the minimum and maximum of an empty set is undefined.

The *Template* values `count`, `sum(X)`, `max(X)`, `min(X)`, `max(X,W)` and `min(X,W)` are processed incrementally rather than using `findall/3` and run in constant memory.

aggregate_all(*+Template, +Discriminator, :Goal, -Result*) *[semidet]*

Aggregate bindings in *Goal* according to *Template*. The `aggregate_all/4` version performs `findall/3` followed by `sort/2` on *Goal*. See `aggregate_all/3` to understand why this predicate can fail.

foreach(*:Generator, :Goal*)

True when the conjunction of *instances* of *Goal* created from solutions for *Generator* is true. Except for term copying, this could be implemented as below.

```
foreach(Generator, Goal) :-
    findall(Goal, Generator, Goals),
    maplist(call, Goals).
```

The actual implementation uses `findall/3` on a template created from the variables *shared* between *Generator* and *Goal*. Subsequently, it uses every instance of this template to instantiate *Goal*, call *Goal* and undo *only* the instantiation of the template and *not* other instantiations created by running *Goal*. Here is an example:

```
?- foreach(between(1,4,X), dif(X,Y)), Y = 5.
Y = 5.
?- foreach(between(1,4,X), dif(X,Y)), Y = 3.
false.
```

The predicate `foreach/2` is mostly used if *Goal* performs backtrackable destructive assignment on terms. Attributed variables (underlying constraints) are an example. Another example of a backtrackable data structure is in `library(hashtable)`. If we care only about the side effects (I/O, dynamic database, etc.) or the truth value of *Goal*, `forall/2` is a faster and simpler alternative. If *Goal* instantiates its arguments it is will often fail as the argument cannot be instantiated to multiple values. It is possible to incrementally *grow* an argument:

```
?- foreach(between(1,4,X), member(X, L)).
L = [1,2,3,4|_].
```

Note that SWI-Prolog up to version 8.3.4 created copies of *Goal* using `copy_term/2` for each iteration, this makes the current implementation unable to properly handle compound terms (in *Goal*'s arguments) that share variables with the *Generator*. As a workaround you can define a goal that does not use compound terms, like in this example:

```
mem(E,L) :-  % mem/2 hides the compound argument from foreach/2
   member(r(E),L).

?- foreach(  between(1,5,N), mem(N,L)).
```

free_variables(*:Generator, +Template, +VarList0, -VarList*) *[det]*

Find free variables in bagof/setof template. In order to handle variables properly, we have to find all the universally quantified variables in the *Generator*. All variables as yet unbound are universally quantified, unless

1. they occur in the template
2. they are bound by X^P, `setof/3`, or `bagof/3`

`free_variables(Generator, Template, OldList, NewList)` finds this set using OldList as an accumulator.

author
- Richard O'Keefe
- Jan Wielemaker (made some SWI-Prolog enhancements)

license Public domain (from DEC10 library).

To be done
- Distinguish between control-structures and data terms.
- Exploit our built-in `term_variables/2` at some places?

sandbox:safe_meta(*+Goal, -Called*) *[semidet,multifile]*

Declare the aggregate meta-calls safe. This cannot be proven due to the manipulations of the argument *Goal*.

A.2 library(ansi_term): Print decorated text to ANSI consoles

See also http://en.wikipedia.org/wiki/ANSI_escape_code

This library allows for exploiting the color and attribute facilities of most modern terminals using ANSI escape sequences. This library provides the following:

- `ansi_format/3` allows writing messages to the terminal with ansi attributes.
- It defines the hook `prolog:message_line_element/2`, which provides ansi attributes for `print_message/2`.

ansi_format(*+ClassOrAttributes, +Format, +Args*) *[det]*

Format text with ANSI attributes. This predicate behaves as `format/2` using *Format* and *Args*, but if the `current_output` is a terminal, it adds ANSI escape sequences according to Attributes. For example, to print a text in bold cyan, do

```
?- ansi_format([bold,fg(cyan)], 'Hello ~w', [world]).
```

Attributes is either a single attribute, a list thereof or a term that is mapped to concrete attributes based on the current theme (see `prolog:console_color/2`). The attribute names are derived from the ANSI specification. See the source for `sgr_code/2` for details. Some commonly used attributes are:

bold

underline

`fg`*(Color)* , `bg`*(Color)* , `hfg`*(Color)* , `hbg`*(Color)*
For `fg(Color)` and `bg(Color)`, the colour name can be '#RGB' or '#RRGGBB'

`fg8`*(Spec)* , `bg8`*(Spec)*
8-bit color specification. *Spec* is a colour name, `h(Color)` or an integer 0..255.

`fg`*(R, G, B)* , `bg`*(R, G, B)*
24-bit (direct color) specification. The components are integers in the range 0..255.

Defined color constants are below. `default` can be used to access the default color of the terminal.

- black, red, green, yellow, blue, magenta, cyan, white

ANSI sequences are sent if and only if

- The `current_output` has the property `tty(true)` (see `stream_property/2`).
- The Prolog flag `color_term` is `true`.

prolog:console_color(*+Term, -AnsiAttributes*) *[semidet,multifile]*

Hook that allows for mapping abstract terms to concrete ANSI attributes. This hook is used by *theme* files to adjust the rendering based on user preferences and context. Defaults are defined in the file `boot/messages.pl`.

See also `library(theme/dark)` for an example implementation and the *Term* values used by the system messages.

prolog:message_line_element(*+Stream, +Term*) *[semidet,multifile]*

Hook implementation that deals with `ansi(+Attr, +Fmt, +Args)` in message specifications.

ansi_get_color(*+Which, -RGB*) *[semidet]*

Obtain the *RGB* color for an ANSI color parameter. *Which* is either a color alias or an integer ANSI color id. Defined aliases are `foreground` and `background`. This predicate sends a request to the console (`user_output`) and reads the reply. This assumes an `xterm` compatible terminal.

Arguments

RGB	is a term `rgb(Red,Green,Blue)`. The color components are integers in the range 0..65535.

A.3 library(apply): Apply predicates on a list

See also

- `apply_macros.pl` provides compile-time expansion for part of this library.
- `http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm`
- Unit test code in `src/Tests/library/test_apply.pl`

To be done Add `include/4`, `include/5`, `exclude/4`, `exclude/5`

This module defines meta-predicates that apply a predicate on all members of a list.

All predicates support partial application in the Goal argument. This means that these calls are identical:

```
?- maplist(=, [foo, foo], [X, Y]).
?- maplist(=(foo), [X, Y]).
```

include(*:Goal, +List1, ?List2*) *[det]*

Filter elements for which *Goal* succeeds. True if *List2* contains those elements Xi of *List1* for which `call(Goal, Xi)` succeeds.

See also `exclude/3`, `partition/4`, `convlist/3`.
Compatibility Older versions of SWI-Prolog had `sublist/3` with the same arguments and semantics.

exclude(*:Goal, +List1, ?List2*) *[det]*

Filter elements for which *Goal* fails. True if *List2* contains those elements Xi of *List1* for which `call(Goal, Xi)` fails.

See also `include/3`, `partition/4`

partition(*:Pred, +List, ?Included, ?Excluded*) *[det]*

Filter elements of *List* according to *Pred*. True if *Included* contains all elements for which `call(Pred, X)` succeeds and *Excluded* contains the remaining elements.

See also `include/3`, `exclude/3`, `partition/5`.

partition(*:Pred, +List, ?Less, ?Equal, ?Greater*) *[semidet]*

Filter *List* according to *Pred* in three sets. For each element Xi of *List*, its destination is determined by `call(Pred, Xi, Place)`, where Place must be unified to one of <, = or >. *Pred* must be deterministic.

See also `partition/4`

maplist(*:Goal, ?List1*)
maplist(*:Goal, ?List1, ?List2*)
maplist(*:Goal, ?List1, ?List2, ?List3*)
maplist(*:Goal, ?List1, ?List2, ?List3, ?List4*)

True if *Goal* is successfully applied on all matching elements of the list. The maplist family of predicates is defined as:

```
maplist(G, [X_11, ..., X_1n],
           [X_21, ..., X_2n],
           ...,
           [X_m1, ..., X_mn]) :-
   call(G, X_11, ..., X_m1),
   call(G, X_12, ..., X_m2),
   ...
   call(G, X_1n, ..., X_mn).
```

This family of predicates is deterministic iff *Goal* is deterministic and *List1* is a proper list, i.e., a list that ends in `[]`.

convlist(*:Goal, +ListIn, -ListOut*) *[det]*

Similar to `maplist/3`, but elements for which `call(Goal, ElemIn, _)` fails are omitted from *ListOut*. For example (using `library(yall)`):

```
?- convlist([X,Y]>>(integer(X), Y is X^2),
            [3, 5, foo, 2], L).
L = [9, 25, 4].
```

Compatibility Also appears in YAP `library(maplist)` and SICStus `library(lists)`.

foldl(*:Goal, +List, +V0, -V*)
foldl(*:Goal, +List1, +List2, +V0, -V*)
foldl(*:Goal, +List1, +List2, +List3, +V0, -V*)
foldl(*:Goal, +List1, +List2, +List3, +List4, +V0, -V*)

Fold an ensemble of *m* (0 <= *m* <= 4) lists of length *n* head-to-tail ("fold-left"), using columns of *m* list elements as arguments for *Goal*. The `foldl` family of predicates is defined as follows, with *V0* an initial value and *V* the final value of the folding operation:

```
foldl(G, [X_11, ..., X_1n],
         [X_21, ..., X_2n],
         ...,
         [X_m1, ..., X_mn], V0, V) :-
   call(G, X_11, ..., X_m1, V0, V1),
   call(G, X_12, ..., X_m2, V1, V2),
   ...
   call(G, X_1n, ..., X_mn, V<n-1>, V).
```

No implementation for a corresponding `foldr` is given. A `foldr` implementation would consist in first calling `reverse/2` on each of the *m* input lists, then applying the appropriate `foldl`. This is actually more efficient than using a properly programmed-out recursive algorithm that cannot be tail-call optimized.

scanl(*:Goal, +List, +V0, -Values*)
scanl(*:Goal, +List1, +List2, +V0, -Values*)
scanl(*:Goal, +List1, +List2, +List3, +V0, -Values*)
scanl(*:Goal, +List1, +List2, +List3, +List4, +V0, -Values*)

Scan an ensemble of *m* (0 <= *m* <= 4) lists of length *n* head-to-tail (”scan-left”), using columns of *m* list elements as arguments for *Goal*. The `scanl` family of predicates is defined as follows, with *V0* an initial value and *V* the final value of the scanning operation:

```
scanl(G, [X_11, ..., X_1n],
         [X_21, ..., X_2n],
         ...,
         [X_m1, ..., X_mn], V0, [V0, V1, ..., Vn] ) :-
   call(G, X_11, ..., X_m1, V0, V1),
   call(G, X_12, ..., X_m2, V1, V2),
   ...
   call(G, X_1n, ..., X_mn, V<n-1>, Vn).
```

`scanl` behaves like a `foldl` that collects the sequence of values taken on by the *Vx* accumulator into a list.

A.4 library(assoc): Association lists

Authors: *Richard A. O'Keefe, L.Damas, V.S.Costa and Markus Triska*

A.4.1 Introduction

An *association list* as implemented by this library is a collection of unique *keys* that are associated to *values*. Keys must 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.

This library uses **AVL trees** to implement association lists. This means that

- inserting a key
- changing an association
- fetching a single element

are all *O(*`log(N)`*)* *worst-case* (and expected) time operations, where *N* denotes the number of elements in the association list.

The logarithmic overhead is often acceptable in practice. Notable advantages of association lists over several other methods are:

- `library(assoc)` is written entirely in Prolog, making it portable to other systems
- the interface predicates fit the declarative nature of Prolog, avoiding destructive updates to terms
- AVL trees scale very predictably and can be used to represent sparse arrays efficiently.

A.4.2 Creating association lists

An association list is *created* with one of the following predicates:

empty_assoc(*?Assoc*) *[semidet]*
Is true if *Assoc* is the empty association list.

list_to_assoc(*+Pairs, -Assoc*) *[det]*
Create an association from a list *Pairs* of Key-Value pairs. List must not contain duplicate keys.

Errors `domain_error(unique_key_pairs, List)` if List contains duplicate keys

ord_list_to_assoc(*+Pairs, -Assoc*) *[det]*
Assoc is created from an ordered list *Pairs* of Key-Value pairs. The pairs must occur in strictly ascending order of their keys.

Errors `domain_error(key_ordered_pairs, List)` if pairs are not ordered.

A.4.3 Querying association lists

An association list can be *queried* with:

get_assoc(*+Key, +Assoc, -Value*) *[semidet]*
True if *Key-Value* is an association in *Assoc*.

Errors `type_error(assoc, Assoc)` if *Assoc* is not an association list.

get_assoc(*+Key, +Assoc0, ?Val0, ?Assoc, ?Val*) *[semidet]*
True if *Key-Val0* is in *Assoc0* and *Key-Val* is in *Assoc*.

max_assoc(*+Assoc, -Key, -Value*) *[semidet]*
True if *Key-Value* is in *Assoc* and *Key* is the largest key.

min_assoc(*+Assoc, -Key, -Value*) *[semidet]*
True if *Key-Value* is in assoc and *Key* is the smallest key.

gen_assoc(*?Key, +Assoc, ?Value*) *[nondet]*
True if *Key-Value* is an association in *Assoc*. Enumerates keys in ascending order on backtracking.

See also `get_assoc/3`.

A.4.4 Modifying association lists

Elements of an association list can be changed and inserted with:

put_assoc(+*Key, +Assoc0, +Value, -Assoc*) *[det]*
Assoc is *Assoc0*, except that *Key* is associated with *Value*. This can be used to insert and change associations.

del_assoc(+*Key, +Assoc0, ?Value, -Assoc*) *[semidet]*
True if *Key*-*Value* is in *Assoc0*. *Assoc* is *Assoc0* with *Key*-*Value* removed.

del_min_assoc(+*Assoc0, ?Key, ?Val, -Assoc*) *[semidet]*
True if *Key*-Value is in *Assoc0* and *Key* is the smallest key. *Assoc* is *Assoc0* with *Key*-Value removed. Warning: This will succeed with *no* bindings for *Key* or *Val* if *Assoc0* is empty.

del_max_assoc(+*Assoc0, ?Key, ?Val, -Assoc*) *[semidet]*
True if *Key*-Value is in *Assoc0* and *Key* is the greatest key. *Assoc* is *Assoc0* with *Key*-Value removed. Warning: This will succeed with *no* bindings for *Key* or *Val* if *Assoc0* is empty.

A.4.5 Conversion predicates

Conversion of (parts of) an association list to *lists* is possible with:

assoc_to_list(+*Assoc, -Pairs*) *[det]*
Translate *Assoc* to a list *Pairs* of Key-Value pairs. The keys in *Pairs* are sorted in ascending order.

assoc_to_keys(+*Assoc, -Keys*) *[det]*
True if *Keys* is the list of keys in *Assoc*. The keys are sorted in ascending order.

assoc_to_values(+*Assoc, -Values*) *[det]*
True if *Values* is the list of values in *Assoc*. *Values* are ordered in ascending order of the key to which they were associated. *Values* may contain duplicates.

A.4.6 Reasoning about association lists and their elements

Further inspection predicates of an association list and its elements are:

is_assoc(+*Assoc*) *[semidet]*
True if *Assoc* is an association list. This predicate checks that the structure is valid, elements are in order, and tree is balanced to the extent guaranteed by AVL trees. I.e., branches of each subtree differ in depth by at most 1.

map_assoc(*:Pred, +Assoc*) *[semidet]*
True if *Pred*(Value) is true for all values in *Assoc*.

map_assoc(*:Pred, +Assoc0, ?Assoc*) *[semidet]*
Map corresponding values. True if *Assoc* is *Assoc0* with *Pred* applied to all corresponding pairs of of values.

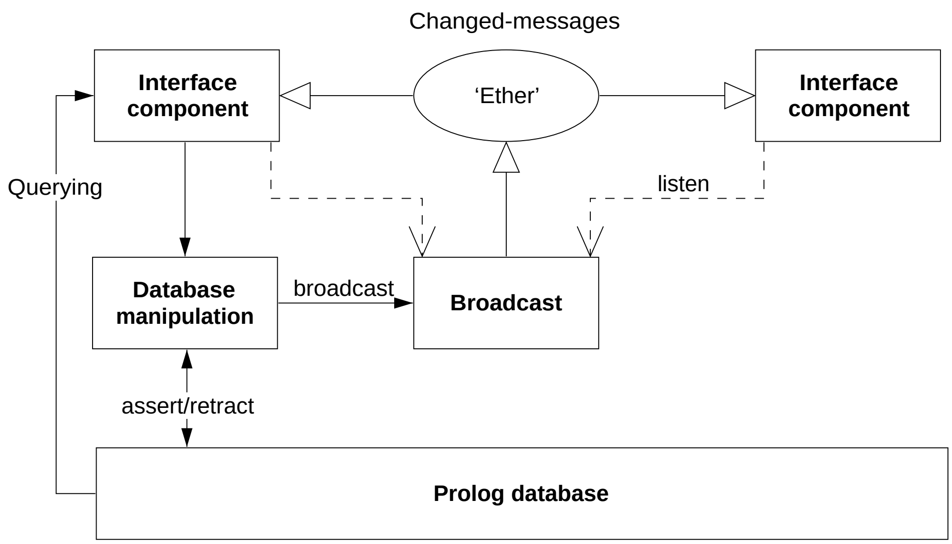

Figure A.1: Information-flow using broadcasting service

A.5 library(broadcast): Broadcast and receive event notifications

The `broadcast` library was invented to realise GUI applications consisting of stand-alone components that use the Prolog database for storing the application data. Figure A.1 illustrates the flow of information using this design

The broadcasting service provides two services. Using the 'shout' service, an unknown number of agents may listen to the message and act. The broadcaster is not (directly) aware of the implications. Using the 'request' service, listening agents are asked for an answer one-by-one and the broadcaster is allowed to reject answers using normal Prolog failure.

Shouting is often used to inform about changes made to a common database. Other messages can be "save yourself" or "show this".

Requesting is used to get information while the broadcaster is not aware who might be able to answer the question. For example "who is showing X?".

broadcast(+*Term*)
: Broadcast *Term*. There are no limitations to *Term*, though being a global service, it is good practice to use a descriptive and unique principal functor. All associated goals are started and regardless of their success or failure, `broadcast/1` always succeeds. Exceptions are passed.

broadcast_request(+*Term*)
: Unlike `broadcast/1`, this predicate stops if an associated goal succeeds. Backtracking causes it to try other listeners. A broadcast request is used to fetch information without knowing the identity of the agent providing it. C.f. "Is there someone who knows the age of John?" could be asked using

```
        ...,
        broadcast_request(age_of('John', Age)),
```

If there is an agent (*listener*) that registered an 'age-of' service and knows about the age of 'John' this question will be answered.

listen(*+Template, :Goal*)

Register a *listen* channel. Whenever a term unifying *Template* is broadcasted, call *Goal*. The following example traps all broadcasted messages as a variable unifies to any message. It is commonly used to debug usage of the library.

```
?- listen(Term, (writeln(Term),fail)).
?- broadcast(hello(world)).
hello(world)
true.
```

listen(*+Listener, +Template, :Goal*)

Declare *Listener* as the owner of the channel. Unlike a channel opened using `listen/2`, channels that have an owner can terminate the channel. This is commonly used if an object is listening to broadcast messages. In the example below we define a 'name-item' displaying the name of an identifier represented by the predicate `name_of/2`.

```
:- pce_begin_class(name_item, text_item).

variable(id,    any,    get, "Id visualised").

initialise(NI, Id:any) :->
        name_of(Id, Name),
        send_super(NI, initialise, name, Name,
                   message(NI, set_name, @arg1)),
        send(NI, slot, id, Id),
        listen(NI, name_of(Id, Name),
               send(NI, selection, Name)).

unlink(NI) :->
        unlisten(NI),
        send_super(NI, unlink).

set_name(NI, Name:name) :->
        get(NI, id, Id),
        retractall(name_of(Id, _)),
        assert(name_of(Id, Name)),
        broadcast(name_of(Id, Name)).

:- pce_end_class.
```

unlisten(*+Listener*)

Deregister all entries created with `listen/3` whose *Listener* unify.

unlisten(*+Listener, +Template*)

Deregister all entries created with `listen/3` whose *Listener* and *Template* unify.

unlisten(*+Listener, +Template, :Goal*)
Deregister all entries created with `listen/3` whose *Listener*, *Template* and *Goal* unify.

listening(*?Listener, ?Template, ?Goal*)
Examine the current listeners. This predicate is useful for debugging purposes.

A.6 library(charsio): I/O on Lists of Character Codes

Compatibility The naming of this library is not in line with the ISO standard. We believe that the SWI-Prolog native predicates form a more elegant alternative for this library.

This module emulates the Quintus/SICStus library `charsio.pl` for reading and writing from/to lists of character codes. Most of these predicates are straight calls into similar SWI-Prolog primitives. Some can even be replaced by ISO standard predicates.

format_to_chars(*+Format, +Args, -Codes*) *[det]*
Use `format/2` to write to a list of character codes.

format_to_chars(*+Format, +Args, -Codes, ?Tail*) *[det]*
Use `format/2` to write to a difference list of character codes.

write_to_chars(*+Term, -Codes*)
Write a term to a code list. True when *Codes* is a list of character codes written by `write/1` on *Term*.

write_to_chars(*+Term, -Codes, ?Tail*)
Write a term to a code list. *Codes**Tail* is a difference list of character codes produced by `write/1` on *Term*.

atom_to_chars(*+Atom, -Codes*) *[det]*
Convert *Atom* into a list of character codes.

deprecated Use ISO `atom_codes/2`.

atom_to_chars(*+Atom, -Codes, ?Tail*) *[det]*
Convert *Atom* into a difference list of character codes.

number_to_chars(*+Number, -Codes*) *[det]*
Convert Atom into a list of character codes.

deprecated Use ISO `number_codes/2`.

number_to_chars(*+Number, -Codes, ?Tail*) *[det]*
Convert *Number* into a difference list of character codes.

read_from_chars(*+Codes, -Term*) *[det]*
Read *Codes* into *Term*.

Compatibility The SWI-Prolog version does not require *Codes* to end in a full-stop.

read_term_from_chars(+*Codes, -Term, +Options*) *[det]*
Read *Codes* into *Term*. *Options* are processed by `read_term/3`.

Compatibility sicstus

open_chars_stream(+*Codes, -Stream*) *[det]*
Open *Codes* as an input stream.

See also `open_string/2`.

with_output_to_chars(*:Goal, -Codes*) *[det]*
Run *Goal* as with `once/1`. Output written to `current_output` is collected in *Codes*.

with_output_to_chars(*:Goal, -Codes, ?Tail*) *[det]*
Run *Goal* as with `once/1`. Output written to `current_output` is collected in *Codes**Tail*.

with_output_to_chars(*:Goal, -Stream, -Codes, ?Tail*) *[det]*
Same as `with_output_to_chars/3` using an explicit stream. The difference list *Codes**Tail* contains the character codes that *Goal* has written to *Stream*.

A.7 library(check): Consistency checking

See also
- `gxref/0` provides a graphical cross referencer
- PceEmacs performs real time consistency checks while you edit
- `library(prolog_xref)` implements 'offline' cross-referencing
- `library(prolog_codewalk)` implements 'online' analysis

This library provides some consistency checks for the loaded Prolog program. The predicate `make/0` runs `list_undefined/0` to find undefined predicates in 'user' modules.

check *[det]*
Run all consistency checks defined by `checker/2`. Checks enabled by default are:

- `list_undefined/0` reports undefined predicates
- `list_trivial_fails/0` reports calls for which there is no matching clause.
- `list_redefined/0` reports predicates that have a local definition and a global definition. Note that these are **not** errors.
- `list_autoload/0` lists predicates that will be defined at runtime using the autoloader.

list_undefined *[det]*
list_undefined(+*Options*) *[det]*
Report undefined predicates. This predicate finds undefined predicates by decompiling and analyzing the body of all clauses. *Options*:

module_class(+*Classes*)
Process modules of the given *Classes*. The default for classes is `[user]`. For example, to include the libraries into the examination, use `[user,library]`.

See also
- gxref/0 provides a graphical cross-referencer.
- make/0 calls list_undefined/0

list_autoload *[det]*

Report predicates that may be auto-loaded. These are predicates that are not defined, but will be loaded on demand if referenced.

See also autoload/0
To be done This predicate uses an older mechanism for finding undefined predicates. Should be synchronized with list undefined.

list_redefined

Lists predicates that are defined in the global module user as well as in a normal module; that is, predicates for which the local definition overrules the global default definition.

list_cross_module_calls *[det]*

List calls from one module to another using Module:Goal where the callee is not defined exported, public or multifile, i.e., where the callee should be considered *private*.

list_void_declarations *[det]*

List predicates that have declared attributes, but no clauses.

list_trivial_fails *[det]*
list_trivial_fails(+*Options*) *[det]*

List goals that trivially fail because there is no matching clause. *Options*:

module_class(+*Classes*)
Process modules of the given *Classes*. The default for classes is [user]. For example, to include the libraries into the examination, use [user,library].

trivial_fail_goal(*:Goal*) *[multifile]*

Multifile hook that tells list_trivial_fails/0 to accept *Goal* as valid.

list_strings *[det]*
list_strings(+*Options*) *[det]*

List strings that appear in clauses. This predicate is used to find portability issues for changing the Prolog flag double_quotes from codes to string, creating packed string objects. Warnings may be suppressed using the following multifile hooks:

- string_predicate/1 to stop checking certain predicates
- valid_string_goal/1 to tell the checker that a goal is safe.

See also Prolog flag double_quotes.

list_rationals *[det]*
list_rationals(+*Options*) *[det]*

List rational numbers that appear in clauses. This predicate is used to find portability issues for changing the Prolog flag rational_syntax to natural, creating rational numbers from <integer>/<nonneg>. *Options*:

module_class(+*Classes*)
Determines the modules classes processed. By default only user code is processed. See `prolog_program_clause/2`.

arithmetic(+*Bool*)
If `true` (default `false`) also warn on rationals appearing in arithmetic expressions.

See also Prolog flag `rational_syntax` and `prefer_rationals`.

list_format_errors *[det]*
list_format_errors(+*Options*) *[det]*
List argument errors for `format/2`,3.

string_predicate(:*PredicateIndicator*) *[multifile]*
Multifile hook to disable `list_strings/0` on the given predicate. This is typically used for facts that store strings.

valid_string_goal(+*Goal*) *[semidet,multifile]*
Multifile hook that qualifies *Goal* as valid for `list_strings/0`. For example, `format("Hello world~n")` is considered proper use of string constants.

checker(:*Goal*, +*Message:text*) *[nondet,multifile]*
Register code validation routines. Each clause defines a *Goal* which performs a consistency check executed by `check/0`. *Message* is a short description of the check. For example, assuming the `my_checks` module defines a predicate `list_format_mistakes/0`:

```
:- multifile check:checker/2.
check:checker(my_checks:list_format_mistakes,
              "errors with format/2 arguments").
```

The predicate is dynamic, so you can disable checks with `retract/1`. For example, to stop reporting redefined predicates:

```
retract(check:checker(list_redefined,_)).
```

A.8 library(clpb): CLP(B): Constraint Logic Programming over Boolean Variables

author Markus Triska

A.8.1 Introduction

This library provides CLP(B), Constraint Logic Programming over Boolean variables. It can be used to model and solve combinatorial problems such as verification, allocation and covering tasks.

CLP(B) is an instance of the general CLP(*X*) scheme (section 8), extending logic programming with reasoning over specialised domains.

The implementation is based on reduced and ordered Binary Decision Diagrams (BDDs).

Benchmarks and usage examples of this library are available from: **https://www.metalevel.at/clpb/**

We recommend the following references for citing this library in scientific publications:

```
@inproceedings{Triska2016,
  author    = "Markus Triska",
  title     = "The {Boolean} Constraint Solver of {SWI-Prolog}:
               System Description",
  booktitle = "FLOPS",
  series    = "LNCS",
  volume    = 9613,
  year      = 2016,
  pages     = "45--61"
}

@article{Triska2018,
  title = "Boolean constraints in {SWI-Prolog}:
           A comprehensive system description",
  journal = "Science of Computer Programming",
  volume = "164",
  pages = "98 - 115",
  year = "2018",
  note = "Special issue of selected papers from FLOPS 2016",
  issn = "0167-6423",
  doi = "https://doi.org/10.1016/j.scico.2018.02.001",
  url = "http://www.sciencedirect.com/science/article/pii/S0167642318300273",
  author = "Markus Triska",
  keywords = "CLP(B), Boolean unification, Decision diagrams, BDD"
}
```

These papers are available from https://www.metalevel.at/swiclpb.pdf and https://www.metalevel.at/boolean.pdf respectively.

A.8.2 Boolean expressions

A *Boolean expression* is one of:

`0`	false
`1`	true
variable	unknown truth value
atom	universally quantified variable
~ *Expr*	logical NOT
Expr + *Expr*	logical OR
Expr * *Expr*	logical AND
Expr # *Expr*	exclusive OR
Var ^ *Expr*	existential quantification
Expr =:= *Expr*	equality
Expr =\= *Expr*	disequality (same as #)
Expr =< *Expr*	less or equal (implication)
Expr >= *Expr*	greater or equal
Expr < *Expr*	less than
Expr > *Expr*	greater than
`card(Is,Exprs)`	cardinality constraint (*see below*)
`+(Exprs)`	n-fold disjunction (*see below*)
`*(Exprs)`	n-fold conjunction (*see below*)

where *Expr* again denotes a Boolean expression.

The Boolean expression `card(Is,Exprs)` is true iff the number of true expressions in the list *Exprs* is a member of the list *Is* of integers and integer ranges of the form `From-To`. For example, to state that precisely two of the three variables *X*, *Y* and *Z* are `true`, you can use `sat(card([2],[X,Y,Z]))`.

`+(Exprs)` and `*(Exprs)` denote, respectively, the disjunction and conjunction of all elements in the list *Exprs* of Boolean expressions.

Atoms denote parametric values that are universally quantified. All universal quantifiers appear implicitly in front of the entire expression. In residual goals, universally quantified variables always appear on the right-hand side of equations. Therefore, they can be used to express functional dependencies on input variables.

A.8.3 Interface predicates

The most frequently used CLP(B) predicates are:

sat(+*Expr*)
True iff the Boolean expression *Expr* is satisfiable.

taut(+*Expr*, -*T*)
If *Expr* is a tautology with respect to the posted constraints, succeeds with *T* = **1**. If *Expr* cannot be satisfied, succeeds with *T* = **0**. Otherwise, it fails.

labeling(+*Vs*)
Assigns truth values to the variables *Vs* such that all constraints are satisfied.

The unification of a CLP(B) variable *X* with a term *T* is equivalent to posting the constraint `sat(X=:=T)`.

A.8.4 Examples

Here is an example session with a few queries and their answers:

```
?- use_module(library(clpb)).
true.

?- sat(X*Y).
X = Y, Y = 1.

?- sat(X * ~X).
false.

?- taut(X * ~X, T).
T = 0,
sat(X=:=X).

?- sat(X^Y^(X+Y)).
sat(X=:=X),
sat(Y=:=Y).

?- sat(X*Y + X*Z), labeling([X,Y,Z]).
X = Z, Z = 1, Y = 0 ;
X = Y, Y = 1, Z = 0 ;
X = Y, Y = Z, Z = 1.

?- sat(X =< Y), sat(Y =< Z), taut(X =< Z, T).
T = 1,
sat(X=:=X*Y),
sat(Y=:=Y*Z).

?- sat(1#X#a#b).
sat(X=:=a#b).
```

The pending residual goals constrain remaining variables to Boolean expressions and are declaratively equivalent to the original query. The last example illustrates that when applicable, remaining variables are expressed as functions of universally quantified variables.

A.8.5 Obtaining BDDs

By default, CLP(B) residual goals appear in (approximately) algebraic normal form (ANF). This projection is often computationally expensive. We can set the Prolog flag `clpb_residuals` to the value `bdd` to see the BDD representation of all constraints. This results in faster projection to residual goals, and is also useful for learning more about BDDs. For example:

```
?- set_prolog_flag(clpb_residuals, bdd).
true.
```

```
?- sat(X#Y).
node(3)- (v(X, 0)->node(2);node(1)),
node(1)- (v(Y, 1)->true;false),
node(2)- (v(Y, 1)->false;true).
```

Note that this representation cannot be pasted back on the toplevel, and its details are subject to change. Use `copy_term/3` to obtain such answers as Prolog terms.

The variable order of the BDD is determined by the order in which the variables first appear in constraints. To obtain different orders, we can for example use:

```
?- sat(+[1,Y,X]), sat(X#Y).
node(3)- (v(Y, 0)->node(2);node(1)),
node(1)- (v(X, 1)->true;false),
node(2)- (v(X, 1)->false;true).
```

A.8.6 Enabling monotonic CLP(B)

In the default execution mode, CLP(B) constraints are *not* monotonic. This means that *adding* constraints can yield new solutions. For example:

```
?-          sat(X=:=1), X = 1+0.
false.

?- X = 1+0, sat(X=:=1), X = 1+0.
X = 1+0.
```

This behaviour is highly problematic from a logical point of view, and it may render **declarative debugging** techniques inapplicable.

Set the flag `clpb_monotonic` to `true` to make CLP(B) **monotonic**. If this mode is enabled, then you must wrap CLP(B) variables with the functor `v/1`. For example:

```
?- set_prolog_flag(clpb_monotonic, true).
true.

?- sat(v(X)=:=1#1).
X = 0.
```

A.8.7 Example: Pigeons

In this example, we are attempting to place *I* pigeons into *J* holes in such a way that each hole contains at most one pigeon. One interesting property of this task is that it can be formulated using only *cardinality constraints* (`card/2`). Another interesting aspect is that this task has no short resolution refutations in general.

In the following, we use **Prolog DCG notation** to describe a list *Cs* of CLP(B) constraints that must all be satisfied.

```
:- use_module(library(clpb)).
:- use_module(library(clpfd)).

pigeon(I, J, Rows, Cs) :-
        length(Rows, I), length(Row, J),
        maplist(same_length(Row), Rows),
        transpose(Rows, TRows),
        phrase((all_cards(Rows,[1]),all_cards(TRows,[0,1])), Cs).

all_cards([], _) --> [].
all_cards([Ls|Lss], Cs) --> [card(Cs,Ls)], all_cards(Lss, Cs).
```

Example queries:

```
?- pigeon(9, 8, Rows, Cs), sat(*(Cs)).
false.

?- pigeon(2, 3, Rows, Cs), sat(*(Cs)),
   append(Rows, Vs), labeling(Vs),
   maplist(portray_clause, Rows).
[0, 0, 1].
[0, 1, 0].
etc.
```

A.8.8 Example: Boolean circuit

Consider a Boolean circuit that express the Boolean function XOR with 4 NAND gates. We can model such a circuit with CLP(B) constraints as follows:

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

nand_gate(X, Y, Z) :- sat(Z =:= ~(X*Y)).

xor(X, Y, Z) :-
        nand_gate(X, Y, T1),
        nand_gate(X, T1, T2),
        nand_gate(Y, T1, T3),
        nand_gate(T2, T3, Z).
```

Using universally quantified variables, we can show that the circuit does compute XOR as intended:

```
?- xor(x, y, Z).
sat(Z=:=x#y).
```

A.8.9 Acknowledgments

The interface predicates of this library follow the example of **SICStus Prolog**.

Use SICStus Prolog for higher performance in many cases.

A.8.10 CLP(B) predicate index

In the following, each CLP(B) predicate is described in more detail.

We recommend the following link to refer to this manual:

http://eu.swi-prolog.org/man/clpb.html

sat(+*Expr*) *[semidet]*

True iff *Expr* is a satisfiable Boolean expression.

taut(+*Expr*, -*T*) *[semidet]*

Tautology check. Succeeds with $T = 0$ if the Boolean expression *Expr* cannot be satisfied, and with $T = 1$ if *Expr* is always true with respect to the current constraints. Fails otherwise.

labeling(+*Vs*) *[multi]*

Enumerate concrete solutions. Assigns truth values to the Boolean variables *Vs* such that all stated constraints are satisfied.

sat_count(+*Expr*, -*Count*) *[det]*

Count the number of admissible assignments. *Count* is the number of different assignments of truth values to the variables in the Boolean expression *Expr*, such that *Expr* is true and all posted constraints are satisfiable.

A common form of invocation is `sat_count(+[1|Vs], Count)`: This counts the number of admissible assignments to *Vs* without imposing any further constraints.

Examples:

```
?- sat(A =< B), Vs = [A,B], sat_count(+[1|Vs], Count).
Vs = [A, B],
Count = 3,
sat(A=:=A*B).

?- length(Vs, 120),
   sat_count(+Vs, CountOr),
   sat_count(*(Vs), CountAnd).
Vs = [...],
CountOr = 1329227995784915872903807060280344575,
CountAnd = 1.
```

weighted_maximum(+*Weights*, +*Vs*, -*Maximum*) *[multi]*

Enumerate weighted optima over admissible assignments. Maximize a linear objective function over Boolean variables *Vs* with integer coefficients *Weights*. This predicate assigns 0 and 1 to the variables in *Vs* such that all stated constraints are satisfied, and *Maximum* is the maximum of `sum(Weight_i*V_i)` over all admissible assignments. On backtracking, all admissible assignments that attain the optimum are generated.

This predicate can also be used to *minimize* a linear Boolean program, since negative integers can appear in *Weights*.

Example:

```
?- sat(A#B), weighted_maximum([1,2,1], [A,B,C], Maximum).
A = 0, B = 1, C = 1, Maximum = 3.
```

random_labeling(+*Seed*, +*Vs*) *[det]*

Select a single random solution. An admissible assignment of truth values to the Boolean variables in *Vs* is chosen in such a way that each admissible assignment is equally likely. *Seed* is an integer, used as the initial seed for the random number generator.

A.9 library(clpfd): CLP(FD): Constraint Logic Programming over Finite Domains

author Markus Triska

Development of this library has moved to SICStus Prolog.
Please see **CLP(Z)** for more information.

A.9.1 Introduction

This library provides CLP(FD): Constraint Logic Programming over Finite Domains. This is an instance of the general CLP(*X*) scheme (section 8), extending logic programming with reasoning over specialised domains. CLP(FD) lets us reason about **integers** in a way that honors the relational nature of Prolog.

Read **The Power of Prolog** to understand how this library is meant to be used in practice.

There are two major use cases of CLP(FD) constraints:

1. **declarative integer arithmetic** (section A.9.3)
2. solving **combinatorial problems** such as planning, scheduling and allocation tasks.

The predicates of this library can be classified as:

- *arithmetic* constraints like `#=/2`, `#>/2` and `#\=/2` (section A.9.17)
- the *membership* constraints `in/2` and `ins/2` (section A.9.17)
- the *enumeration* predicates `indomain/1`, `label/1` and `labeling/2` (section A.9.17)
- *combinatorial* constraints like `all_distinct/1` and `global_cardinality/2` (section A.9.17)
- *reification* predicates such as `#<==>/2` (section A.9.17)
- *reflection* predicates such as `fd_dom/2` (section A.9.17)

In most cases, *arithmetic constraints* (section A.9.2) are the only predicates you will ever need from this library. When reasoning over integers, simply replace low-level arithmetic predicates like `(is)/2` and `(>)/2` by the corresponding CLP(FD) constraints like `#=/2` and `#>/2` to honor and preserve declarative properties of your programs. For satisfactory performance, arithmetic constraints are implicitly rewritten at compilation time so that low-level fallback predicates are automatically used whenever possible.

Almost all Prolog programs also reason about integers. Therefore, it is highly advisable that you make CLP(FD) constraints available in all your programs. One way to do this is to put the following directive in your `<config>/init.pl` initialisation file:

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

All example programs that appear in the CLP(FD) documentation assume that you have done this.

Important concepts and principles of this library are illustrated by means of usage examples that are available in a public git repository: **github.com/triska/clpfd**

If you are used to the complicated operational considerations that low-level arithmetic primitives necessitate, then moving to CLP(FD) constraints may, due to their power and convenience, at first feel to you excessive and almost like cheating. It *isn't*. Constraints are an integral part of all popular Prolog systems, and they are designed to help you eliminate and avoid the use of low-level and less general primitives by providing declarative alternatives that are meant to be used instead.

When teaching Prolog, CLP(FD) constraints should be introduced *before* explaining low-level arithmetic predicates and their procedural idiosyncrasies. This is because constraints are easy to explain, understand and use due to their purely relational nature. In contrast, the modedness and directionality of low-level arithmetic primitives are impure limitations that are better deferred to more advanced lectures.

We recommend the following reference (PDF: metalevel.at/swiclpfd.pdf) for citing this library in scientific publications:

```
@inproceedings{Triska12,
  author    = {Markus Triska},
  title     = {The Finite Domain Constraint Solver of {SWI-Prolog}},
  booktitle = {FLOPS},
  series    = {LNCS},
  volume    = {7294},
  year      = {2012},
  pages     = {307-316}
}
```

More information about CLP(FD) constraints and their implementation is contained in: **metalevel.at/drt.pdf**

The best way to discuss applying, improving and extending CLP(FD) constraints is to use the dedicated `clpfd` tag on stackoverflow.com. Several of the world's foremost CLP(FD) experts regularly participate in these discussions and will help you for free on this platform.

A.9.2 Arithmetic constraints

In modern Prolog systems, **arithmetic constraints** subsume and supersede low-level predicates over integers. The main advantage of arithmetic constraints is that they are true *relations* and can be used in all directions. For most programs, arithmetic constraints are the only predicates you will ever need from this library.

The most important arithmetic constraint is `#=/2`, which subsumes both `(is)/2` and `(=:=)/2` over integers. Use `#=/2` to make your programs more general. See declarative integer arithmetic (section A.9.3).

In total, the arithmetic constraints are:

Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #\= Expr2	Expr1 is not equal to Expr2
Expr1 #>= Expr2	Expr1 is greater than or equal to Expr2
Expr1 #=< Expr2	Expr1 is less than or equal to Expr2
Expr1 #> Expr2	Expr1 is greater than Expr2
Expr1 #< Expr2	Expr1 is less than Expr2

Expr1 and *Expr2* denote **arithmetic expressions**, which are:

integer	Given value
variable	Unknown integer
?(*variable*)	Unknown integer
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr ^ Expr	Exponentiation
`min(Expr,Expr)`	Minimum of two expressions
`max(Expr,Expr)`	Maximum of two expressions
Expr `mod` Expr	Modulo induced by floored division
Expr `rem` Expr	Modulo induced by truncated division
`abs(Expr)`	Absolute value
Expr // Expr	Truncated integer division
Expr div Expr	Floored integer division

where *Expr* again denotes an arithmetic expression.

The bitwise operations `(\)/1`, `(/\)/2`, `(\/)/2`, `(>>)/2`, `(<<)/2`, `lsb/1`, `msb/1`, `popcount/1` and `(xor)/2` are also supported.

A.9.3 Declarative integer arithmetic

The *arithmetic constraints* (section A.9.2) `#=/2`, `#>/2` etc. are meant to be used *instead* of the primitives `(is)/2`, `(=:=)/2`, `(>)/2` etc. over integers. Almost all Prolog programs also reason about integers. Therefore, it is recommended that you put the following directive in your `<config>/init.pl` initialisation file to make CLP(FD) constraints available in all your programs:

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

Throughout the following, it is assumed that you have done this.

The most basic use of CLP(FD) constraints is *evaluation* of arithmetic expressions involving integers. For example:

```
?- X #= 1+2.
X = 3.
```

This could in principle also be achieved with the lower-level predicate `(is)/2`. However, an important advantage of arithmetic constraints is their purely relational nature: Constraints can be used in *all directions*, also if one or more of their arguments are only partially instantiated. For example:

```
?- 3 #= Y+2.
Y = 1.
```

This relational nature makes CLP(FD) constraints easy to explain and use, and well suited for beginners and experienced Prolog programmers alike. In contrast, when using low-level integer arithmetic, we get:

```
?- 3 is Y+2.
ERROR: is/2: Arguments are not sufficiently instantiated

?- 3 =:= Y+2.
ERROR: =:=/2: Arguments are not sufficiently instantiated
```

Due to the necessary operational considerations, the use of these low-level arithmetic predicates is considerably harder to understand and should therefore be deferred to more advanced lectures.

For supported expressions, CLP(FD) constraints are drop-in replacements of these low-level arithmetic predicates, often yielding more general programs. See `n_factorial/2` (section A.9.4) for an example.

This library uses `goal_expansion/2` to automatically rewrite constraints at compilation time so that low-level arithmetic predicates are *automatically* used whenever possible. For example, the predicate:

```
positive_integer(N) :- N #>= 1.
```

is executed as if it were written as:

```
positive_integer(N) :-
        (   integer(N)
        ->  N >= 1
        ;   N #>= 1
        ).
```

This illustrates why the performance of CLP(FD) constraints is almost always completely satisfactory when they are used in modes that can be handled by low-level arithmetic. To disable the automatic rewriting, set the Prolog flag `clpfd_goal_expansion` to `false`.

If you are used to the complicated operational considerations that low-level arithmetic primitives necessitate, then moving to CLP(FD) constraints may, due to their power and convenience, at first feel to you excessive and almost like cheating. It *isn't*. Constraints are an integral part of all popular Prolog systems, and they are designed to help you eliminate and avoid the use of low-level and less general primitives by providing declarative alternatives that are meant to be used instead.

A.9.4 Example: Factorial relation

We illustrate the benefit of using `#=/2` for more generality with a simple example.

Consider first a rather conventional definition of `n_factorial/2`, relating each natural number *N* to its factorial *F*:

```
n_factorial(0, 1).
n_factorial(N, F) :-
        N #> 0,
        N1 #= N - 1,
        n_factorial(N1, F1),
        F #= N * F1.
```

This program uses CLP(FD) constraints *instead* of low-level arithmetic throughout, and everything that *would have worked* with low-level arithmetic *also* works with CLP(FD) constraints, retaining roughly the same performance. For example:

```
?- n_factorial(47, F).
F = 258623241511168180642964355153611979969197632389120000000000 ;
false.
```

Now the point: Due to the increased flexibility and generality of CLP(FD) constraints, we are free to *reorder* the goals as follows:

```
n_factorial(0, 1).
n_factorial(N, F) :-
        N #> 0,
        N1 #= N - 1,
        F #= N * F1,
        n_factorial(N1, F1).
```

In this concrete case, *termination* properties of the predicate are improved. For example, the following queries now both terminate:

```
?- n_factorial(N, 1).
N = 0 ;
N = 1 ;
false.

?- n_factorial(N, 3).
false.
```

To make the predicate terminate if *any* argument is instantiated, add the (implied) constraint `F #\= 0` before the recursive call. Otherwise, the query `n_factorial(N, 0)` is the only non-terminating case of this kind.

The value of CLP(FD) constraints does *not* lie in completely freeing us from *all* procedural phenomena. For example, the two programs do not even have the same *termination properties* in all cases.

Instead, the primary benefit of CLP(FD) constraints is that they allow you to try different execution orders and apply **declarative debugging** techniques *at all*! Reordering goals (and clauses) can significantly impact the performance of Prolog programs, and you are free to try different variants if you use declarative approaches. Moreover, since all CLP(FD) constraints *always terminate*, placing them earlier can at most *improve*, never worsen, the termination properties of your programs. An additional benefit of CLP(FD) constraints is that they eliminate the complexity of introducing `(is)/2` and `(=:=)/2` to beginners, since *both* predicates are subsumed by `#=/2` when reasoning over integers.

In the case above, the clauses are mutually exclusive *if* the first argument is sufficiently instantiated. To make the predicate deterministic in such cases while retaining its generality, you can use `zcompare/3` to *reify* a comparison, making the different cases distinguishable by pattern matching. For example, in this concrete case and others like it, you can use `zcompare(Comp, 0, N)` to obtain as *Comp* the symbolic outcome (<, =, >) of 0 compared to N.

A.9.5 Combinatorial constraints

In addition to subsuming and replacing low-level arithmetic predicates, CLP(FD) constraints are often used to solve combinatorial problems such as planning, scheduling and allocation tasks. Among the most frequently used **combinatorial constraints** are `all_distinct/1`, `global_cardinality/2` and `cumulative/2`. This library also provides several other constraints like `disjoint2/1` and `automaton/8`, which are useful in more specialized applications.

A.9.6 Domains

Each CLP(FD) variable has an associated set of admissible integers, which we call the variable's **domain**. Initially, the domain of each CLP(FD) variable is the set of *all* integers. CLP(FD) constraints like `#=/2`, `#>/2` and `#\=/2` can at most reduce, and never extend, the domains of their arguments. The constraints `in/2` and `ins/2` let us explicitly state domains of CLP(FD) variables. The process of determining and adjusting domains of variables is called constraint **propagation**, and it is performed automatically by this library. When the domain of a variable contains only one element, then the variable is automatically unified to that element.

Domains are taken into account when further constraints are stated, and by enumeration predicates like `labeling/2`.

A.9.7 Example: Sudoku

As another example, consider *Sudoku*: It is a popular puzzle over integers that can be easily solved with CLP(FD) constraints.

```
sudoku(Rows) :-
        length(Rows, 9), maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        blocks(Ds, Es, Fs),
        blocks(Gs, Hs, Is).
```

```
blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
        blocks(Ns1, Ns2, Ns3).

problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).
```

Sample query:

```
?- problem(1, Rows), sudoku(Rows), maplist(portray_clause, Rows).
[9, 8, 7, 6, 5, 4, 3, 2, 1].
[2, 4, 6, 1, 7, 3, 9, 8, 5].
[3, 5, 1, 9, 2, 8, 7, 4, 6].
[1, 2, 8, 5, 3, 7, 6, 9, 4].
[6, 3, 4, 8, 9, 2, 1, 5, 7].
[7, 9, 5, 4, 6, 1, 8, 3, 2].
[5, 1, 9, 2, 8, 6, 4, 7, 3].
[4, 7, 2, 3, 1, 9, 5, 6, 8].
[8, 6, 3, 7, 4, 5, 2, 1, 9].
Rows = [[9, 8, 7, 6, 5, 4, 3, 2|...], ... , [...|...]].
```

In this concrete case, the constraint solver is strong enough to find the unique solution without any search. For the general case, see search (section A.9.9).

A.9.8 Residual goals

Here is an example session with a few queries and their answers:

```
?- X #> 3.
X in 4..sup.

?- X #\= 20.
X in inf..19\/21..sup.

?- 2*X #= 10.
X = 5.
```

```
?- X*X #= 144.
X in -12\/12.

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

?- X #= Y #<==> B, X in 0..3, Y in 4..5.
B = 0,
X in 0..3,
Y in 4..5.
```

The answers emitted by the toplevel are called *residual programs*, and the goals that comprise each answer are called **residual goals**. In each case above, and as for all pure programs, the residual program is declaratively equivalent to the original query. From the residual goals, it is clear that the constraint solver has deduced additional domain restrictions in many cases.

To inspect residual goals, it is best to let the toplevel display them for us. Wrap the call of your predicate into `call_residue_vars/2` to make sure that all constrained variables are displayed. To make the constraints a variable is involved in available as a Prolog term for further reasoning within your program, use `copy_term/3`. For example:

```
?- X #= Y + Z, X in 0..5, copy_term([X,Y,Z], [X,Y,Z], Gs).
Gs = [clpfd: (X in 0..5), clpfd: (Y+Z#=X)],
X in 0..5,
Y+Z#=X.
```

This library also provides *reflection* predicates (like `fd_dom/2`, `fd_size/2` etc.) with which we can inspect a variable's current domain. These predicates can be useful if you want to implement your own labeling strategies.

A.9.9 Core relations and search

Using CLP(FD) constraints to solve combinatorial tasks typically consists of two phases:

1. **Modeling**. In this phase, all relevant constraints are stated.
2. **Search**. In this phase, *enumeration predicates* are used to search for concrete solutions.

It is good practice to keep the modeling part, via a dedicated predicate called the **core relation**, separate from the actual search for solutions. This lets us observe termination and determinism properties of the core relation in isolation from the search, and more easily try different search strategies.

As an example of a constraint satisfaction problem, consider the cryptoarithmetic puzzle SEND + MORE = MONEY, where different letters denote distinct integers between 0 and 9. It can be modeled in CLP(FD) as follows:

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
        Vars = [S,E,N,D,M,O,R,Y],
```

```
        Vars ins 0..9,
        all_different(Vars),
                  S*1000 + E*100 + N*10 + D +
                  M*1000 + O*100 + R*10 + E #=
        M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0.
```

Notice that we are *not* using `labeling/2` in this predicate, so that we can first execute and observe the modeling part in isolation. Sample query and its result (actual variables replaced for readability):

```
?- puzzle(As+Bs=Cs).
As = [9, A2, A3, A4],
Bs = [1, 0, B3, A2],
Cs = [1, 0, A3, A2, C5],
A2 in 4..7,
all_different([9, A2, A3, A4, 1, 0, B3, C5]),
91*A2+A4+10*B3#=90*A3+C5,
A3 in 5..8,
A4 in 2..8,
B3 in 2..8,
C5 in 2..8.
```

From this answer, we see that this core relation *terminates* and is in fact *deterministic*. Moreover, we see from the residual goals that the constraint solver has deduced more stringent bounds for all variables. Such observations are only possible if modeling and search parts are cleanly separated.

Labeling can then be used to search for solutions in a separate predicate or goal:

```
?- puzzle(As+Bs=Cs), label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
false.
```

In this case, it suffices to label a subset of variables to find the puzzle's unique solution, since the constraint solver is strong enough to reduce the domains of remaining variables to singleton sets. In general though, it is necessary to label all variables to obtain ground solutions.

A.9.10 Example: Eight queens puzzle

We illustrate the concepts of the preceding sections by means of the so-called *eight queens puzzle*. The task is to place 8 queens on an 8x8 chessboard such that none of the queens is under attack. This means that no two queens share the same row, column or diagonal.

To express this puzzle via CLP(FD) constraints, we must first pick a suitable representation. Since CLP(FD) constraints reason over *integers*, we must find a way to map the positions of queens to integers. Several such mappings are conceivable, and it is not immediately obvious which we should use. On top of that, different constraints can be used to express the desired relations. For such reasons, *modeling* combinatorial problems via CLP(FD) constraints often necessitates some creativity and has been described as more of an art than a science.

In our concrete case, we observe that there must be exactly one queen per column. The following representation therefore suggests itself: We are looking for 8 integers, one for each column, where each integer denotes the *row* of the queen that is placed in the respective column, and which are subject to certain constraints.

In fact, let us now generalize the task to the so-called *N queens puzzle*, which is obtained by replacing 8 by *N* everywhere it occurs in the above description. We implement the above considerations in the **core relation** `n_queens/2`, where the first argument is the number of queens (which is identical to the number of rows and columns of the generalized chessboard), and the second argument is a list of *N* integers that represents a solution in the form described above.

```
n_queens(N, Qs) :-
        length(Qs, N),
        Qs ins 1..N,
        safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :- safe_queens(Qs, Q, 1), safe_queens(Qs).

safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
        Q0 #\= Q,
        abs(Q0 - Q) #\= D0,
        D1 #= D0 + 1,
        safe_queens(Qs, Q0, D1).
```

Note that all these predicates can be used in *all directions*: We can use them to *find* solutions, *test* solutions and *complete* partially instantiated solutions.

The original task can be readily solved with the following query:

```
?- n_queens(8, Qs), label(Qs).
Qs = [1, 5, 8, 6, 3, 7, 2, 4] .
```

Using suitable labeling strategies, we can easily find solutions with 80 queens and more:

```
?- n_queens(80, Qs), labeling([ff], Qs).
Qs = [1, 3, 5, 44, 42, 4, 50, 7, 68|...] .

?- time((n_queens(90, Qs), labeling([ff], Qs))).
% 5,904,401 inferences, 0.722 CPU in 0.737 seconds (98% CPU)
Qs = [1, 3, 5, 50, 42, 4, 49, 7, 59|...] .
```

Experimenting with different search strategies is easy because we have separated the core relation from the actual search.

A.9.11 Optimisation

We can use `labeling/2` to minimize or maximize the value of a CLP(FD) expression, and generate solutions in increasing or decreasing order of the value. See the labeling options `min(Expr)` and `max(Expr)`, respectively.

Again, to easily try different labeling options in connection with optimisation, we recommend to introduce a dedicated predicate for posting constraints, and to use `labeling/2` in a separate goal. This way, we can observe properties of the core relation in isolation, and try different labeling options without recompiling our code.

If necessary, we can use `once/1` to commit to the first optimal solution. However, it is often very valuable to see alternative solutions that are *also* optimal, so that we can choose among optimal solutions by other criteria. For the sake of **purity** and completeness, we recommend to avoid `once/1` and other constructs that lead to impurities in CLP(FD) programs.

Related to optimisation with CLP(FD) constraints are `library(simplex)` and CLP(Q) which reason about *linear* constraints over rational numbers.

A.9.12 Reification

The constraints `in/2`, `#=/2`, `#\=/2`, `#</2`, `#>/2`, `#=</2`, and `#>=/2` can be *reified*, which means reflecting their truth values into Boolean values represented by the integers 0 and 1. Let P and Q denote reifiable constraints or Boolean variables, then:

#\ Q	True iff Q is false
P #\/ Q	True iff either P or Q
P #/\ Q	True iff both P and Q
P #\ Q	True iff either P or Q, but not both
P #<==> Q	True iff P and Q are equivalent
P #==> Q	True iff P implies Q
P #<== Q	True iff Q implies P

The constraints of this table are reifiable as well.

When reasoning over Boolean variables, also consider using CLP(B) constraints as provided by `library(clpb)`.

A.9.13 Enabling monotonic CLP(FD)

In the default execution mode, CLP(FD) constraints still exhibit some non-relational properties. For example, *adding* constraints can yield new solutions:

```
?-          X #= 2, X = 1+1.
false.

?- X = 1+1, X #= 2, X = 1+1.
X = 1+1.
```

This behaviour is highly problematic from a logical point of view, and it may render declarative debugging techniques inapplicable.

Set the Prolog flag `clpfd_monotonic` to `true` to make CLP(FD) **monotonic**: This means that *adding* new constraints *cannot* yield new solutions. When this flag is `true`, we must wrap variables that occur in arithmetic expressions with the functor `(?)/1` or `(#)/1`. For example:

```
?- set_prolog_flag(clpfd_monotonic, true).
true.

?- #(X) #= #(Y) + #(Z).
#(Y)+ #(Z)#= #(X).

?-          X #= 2, X = 1+1.
ERROR: Arguments are not sufficiently instantiated
```

The wrapper can be omitted for variables that are already constrained to integers.

A.9.14 Custom constraints

We can define custom constraints. The mechanism to do this is not yet finalised, and we welcome suggestions and descriptions of use cases that are important to you.

As an example of how it can be done currently, let us define a new custom constraint `oneground(X,Y,Z)`, where Z shall be 1 if at least one of X and Y is instantiated:

```
:- multifile clpfd:run_propagator/2.

oneground(X, Y, Z) :-
        clpfd:make_propagator(oneground(X, Y, Z), Prop),
        clpfd:init_propagator(X, Prop),
        clpfd:init_propagator(Y, Prop),
        clpfd:trigger_once(Prop).

clpfd:run_propagator(oneground(X, Y, Z), MState) :-
        (   integer(X) -> clpfd:kill(MState), Z = 1
        ;   integer(Y) -> clpfd:kill(MState), Z = 1
        ;   true
        ).
```

First, `clpfd:make_propagator/2` is used to transform a user-defined representation of the new constraint to an internal form. With `clpfd:init_propagator/2`, this internal form is then attached to X and Y. From now on, the propagator will be invoked whenever the domains of X or Y are changed. Then, `clpfd:trigger_once/1` is used to give the propagator its first chance for propagation even though the variables' domains have not yet changed. Finally, `clpfd:run_propagator/2` is extended to define the actual propagator. As explained, this predicate is automatically called by the constraint solver. The first argument is the user-defined representation of the constraint as used in `clpfd:make_propagator/2`, and the second argument is a mutable state that can be used to prevent further invocations of the propagator when the constraint has become entailed, by using `clpfd:kill/1`. An example of using the new constraint:

```
?- oneground(X, Y, Z), Y = 5.
Y = 5,
Z = 1,
X in inf..sup.
```

A.9.15 Applications

CLP(FD) applications that we find particularly impressive and worth studying include:

- Michael Hendricks uses CLP(FD) constraints for flexible reasoning about *dates* and *times* in the `julian` package.
- Julien Cumin uses CLP(FD) constraints for integer arithmetic in `Brachylog`.

A.9.16 Acknowledgments

This library gives you a glimpse of what **SICStus Prolog** can do. The API is intentionally mostly compatible with that of SICStus Prolog, so that you can easily switch to a much more feature-rich and much faster CLP(FD) system when you need it. I thank Mats Carlsson, the designer and main implementor of SICStus Prolog, for his elegant example. I first encountered his system as part of the excellent **GUPU** teaching environment by Ulrich Neumerkel. Ulrich was also the first and most determined tester of the present system, filing hundreds of comments and suggestions for improvement. Tom Schrijvers has contributed several constraint libraries to SWI-Prolog, and I learned a lot from his coding style and implementation examples. Bart Demoen was a driving force behind the implementation of attributed variables in SWI-Prolog, and this library could not even have started without his prior work and contributions. Thank you all!

A.9.17 CLP(FD) predicate index

In the following, each CLP(FD) predicate is described in more detail.

We recommend the following link to refer to this manual:
`http://eu.swi-prolog.org/man/clpfd.html`

Arithmetic constraints

Arithmetic constraints are the most basic use of CLP(FD). Every time you use `(is)/2` or one of the low-level arithmetic comparisons (`(<)/2`, `(>)/2` etc.) over integers, consider using CLP(FD) constraints *instead*. This can at most *increase* the generality of your programs. See declarative integer arithmetic (section A.9.3).

?X **#=** *?Y*
: The arithmetic expression *X* equals *Y*. This is the most important arithmetic constraint (section A.9.2), subsuming and replacing both `(is)/2` *and* `(=:=)/2` over integers. See declarative integer arithmetic (section A.9.3).

?X **#\=** *?Y*
: The arithmetic expressions *X* and *Y* evaluate to distinct integers. When reasoning over integers,

replace `(=\=)/2` by `#\=/2` to obtain more general relations. See declarative integer arithmetic (section A.9.3).

?X **#>=** *?Y*
: Same as *Y* `#=<` *X*. When reasoning over integers, replace `(>=)/2` by `#>=/2` to obtain more general relations. See declarative integer arithmetic (section A.9.3).

?X **#=<** *?Y*
: The arithmetic expression *X* is less than or equal to *Y*. When reasoning over integers, replace `(=<)/2` by `#=</2` to obtain more general relations. See declarative integer arithmetic (section A.9.3).

?X **#>** *?Y*
: Same as *Y* `#<` *X*. When reasoning over integers, replace `(>)/2` by `#>/2` to obtain more general relations See declarative integer arithmetic (section A.9.3).

?X **#<** *?Y*
: The arithmetic expression *X* is less than *Y*. When reasoning over integers, replace `(<)/2` by `#</2` to obtain more general relations. See declarative integer arithmetic (section A.9.3).

In addition to its regular use in tasks that require it, this constraint can also be useful to eliminate uninteresting symmetries from a problem. For example, all possible matches between pairs built from four players in total:

```
?- Vs = [A,B,C,D], Vs ins 1..4,
        all_different(Vs),
        A #< B, C #< D, A #< C,
   findall(pair(A,B)-pair(C,D), label(Vs), Ms).
Ms = [ pair(1, 2)-pair(3, 4),
       pair(1, 3)-pair(2, 4),
       pair(1, 4)-pair(2, 3)].
```

Membership constraints

If you are using CLP(FD) to model and solve combinatorial tasks, then you typically need to specify the admissible domains of variables. The *membership constraints* `in/2` and `ins/2` are useful in such cases.

?Var **in** *+Domain*
: *Var* is an element of *Domain*. *Domain* is one of:

Integer
: Singleton set consisting only of *Integer*.

Lower **..** *Upper*
: All integers *I* such that *Lower* =< *I* =< *Upper*. *Lower* must be an integer or the atom **inf**, which denotes negative infinity. *Upper* must be an integer or the atom **sup**, which denotes positive infinity.

Domain1 **\/** *Domain2*
: The union of *Domain1* and *Domain2*.

+*Vars* **ins** +*Domain*
: The variables in the list *Vars* are elements of *Domain*. See `in/2` for the syntax of *Domain*.

Enumeration predicates

When modeling combinatorial tasks, the actual search for solutions is typically performed by *enumeration predicates* like `labeling/2`. See the the section about *core relations* and search for more information.

indomain(*?Var*)
: Bind *Var* to all feasible values of its domain on backtracking. The domain of *Var* must be finite.

label(+*Vars*)
: Equivalent to `labeling([], Vars)`. See `labeling/2`.

labeling(+*Options*, +*Vars*)
: Assign a value to each variable in *Vars*. Labeling means systematically trying out values for the finite domain variables *Vars* until all of them are ground. The domain of each variable in *Vars* must be finite. *Options* is a list of options that let you exhibit some control over the search process. Several categories of options exist:

  The variable selection strategy lets you specify which variable of *Vars* is labeled next and is one of:

  leftmost
  : Label the variables in the order they occur in *Vars*. This is the default.

  ff
  : *First fail*. Label the leftmost variable with smallest domain next, in order to detect infeasibility early. This is often a good strategy.

  ffc
  : Of the variables with smallest domains, the leftmost one participating in most constraints is labeled next.

  min
  : Label the leftmost variable whose lower bound is the lowest next.

  max
  : Label the leftmost variable whose upper bound is the highest next.

  The value order is one of:

  up
  : Try the elements of the chosen variable's domain in ascending order. This is the default.

  down
  : Try the domain elements in descending order.

  The branching strategy is one of:

  step
  : For each variable X, a choice is made between X = V and X #\= V, where V is determined by the value ordering options. This is the default.

enum
: For each variable X, a choice is made between X = V_1, X = V_2 etc., for all values V_i of the domain of X. The order is determined by the value ordering options.

bisect
: For each variable X, a choice is made between X `#=<` M and X `#>` M, where M is the midpoint of the domain of X.

At most one option of each category can be specified, and an option must not occur repeatedly.

The order of solutions can be influenced with:

- `min(Expr)`
- `max(Expr)`

This generates solutions in ascending/descending order with respect to the evaluation of the arithmetic expression Expr. Labeling *Vars* must make Expr ground. If several such options are specified, they are interpreted from left to right, e.g.:

```
?- [X,Y] ins 10..20, labeling([max(X),min(Y)],[X,Y]).
```

This generates solutions in descending order of X, and for each binding of X, solutions are generated in ascending order of Y. To obtain the incomplete behaviour that other systems exhibit with "`maximize(Expr)`" and "`minimize(Expr)`", use `once/1`, e.g.:

```
once(labeling([max(Expr)], Vars))
```

Labeling is always complete, always terminates, and yields no redundant solutions. See core relations and search (section A.9.9) for usage advice.

Global constraints

A *global constraint* expresses a relation that involves many variables at once. The most frequently used global constraints of this library are the combinatorial constraints `all_distinct/1`, `global_cardinality/2` and `cumulative/2`.

all_distinct(+*Vars*)
: True iff *Vars* are pairwise distinct. For example, `all_distinct/1` can detect that not all variables can assume distinct values given the following domains:

```
?- maplist(in, Vs,
           [1\/3..4, 1..2\/4, 1..2\/4, 1..3, 1..3, 1..6]),
   all_distinct(Vs).
false.
```

all_different(+*Vars*)
: Like `all_distinct/1`, but with weaker propagation. Consider using `all_distinct/1` instead, since `all_distinct/1` is typically acceptably efficient and propagates much more strongly.

sum(+*Vars*, +*Rel*, ?*Expr***)**

The sum of elements of the list *Vars* is in relation *Rel* to *Expr*. *Rel* is one of #=, #\=, #<, #>, #=< or #>=. For example:

```
?- [A,B,C] ins 0..sup, sum([A,B,C], #=, 100).
A in 0..100,
A+B+C#=100,
B in 0..100,
C in 0..100.
```

scalar_product(+*Cs*, +*Vs*, +*Rel*, ?*Expr***)**

True iff the scalar product of *Cs* and *Vs* is in relation *Rel* to *Expr*. *Cs* is a list of integers, *Vs* is a list of variables and integers. *Rel* is #=, #\=, #<, #>, #=< or #>=.

lex_chain(+*Lists***)**

Lists are lexicographically non-decreasing.

tuples_in(+*Tuples*, +*Relation***)**

True iff all *Tuples* are elements of *Relation*. Each element of the list *Tuples* is a list of integers or finite domain variables. *Relation* is a list of lists of integers. Arbitrary finite relations, such as compatibility tables, can be modeled in this way. For example, if 1 is compatible with 2 and 5, and 4 is compatible with 0 and 3:

```
?- tuples_in([[X,Y]], [[1,2],[1,5],[4,0],[4,3]]), X = 4.
X = 4,
Y in 0\/3.
```

As another example, consider a train schedule represented as a list of quadruples, denoting departure and arrival places and times for each train. In the following program, Ps is a feasible journey of length 3 from A to D via trains that are part of the given schedule.

```
trains([[1,2,0,1],
        [2,3,4,5],
        [2,3,0,1],
        [3,4,5,6],
        [3,4,2,3],
        [3,4,8,9]]).

threepath(A, D, Ps) :-
        Ps = [[A,B,_T0,T1],[B,C,T2,T3],[C,D,T4,_T5]],
        T2 #> T1,
        T4 #> T3,
        trains(Ts),
        tuples_in(Ps, Ts).
```

In this example, the unique solution is found without labeling:

```
?- threepath(1, 4, Ps).
Ps = [[1, 2, 0, 1], [2, 3, 4, 5], [3, 4, 8, 9]].
```

serialized(*+Starts, +Durations*)

Describes a set of non-overlapping tasks. *Starts* = [S_1,...,S_n], is a list of variables or integers, *Durations* = [D_1,...,D_n] is a list of non-negative integers. Constrains *Starts* and *Durations* to denote a set of non-overlapping tasks, i.e.: S_i + D_i =< S_j or S_j + D_j =< S_i for all 1 =< i < j =< n. Example:

```
?- length(Vs, 3),
   Vs ins 0..3,
   serialized(Vs, [1,2,3]),
   label(Vs).
Vs = [0, 1, 3] ;
Vs = [2, 0, 3] ;
false.
```

See also Dorndorf et al. 2000, ”Constraint Propagation Techniques for the Disjunctive Scheduling Problem”

element(*?N, +Vs, ?V*)

The *N*-th element of the list of finite domain variables *Vs* is *V*. Analogous to `nth1/3`.

global_cardinality(*+Vs, +Pairs*)

Global Cardinality constraint. Equivalent to `global_cardinality(Vs, Pairs, [])`. See `global_cardinality/3`.

Example:

```
?- Vs = [_,_,_], global_cardinality(Vs, [1-2,3-_]), label(Vs).
Vs = [1, 1, 3] ;
Vs = [1, 3, 1] ;
Vs = [3, 1, 1].
```

global_cardinality(*+Vs, +Pairs, +Options*)

Global Cardinality constraint. *Vs* is a list of finite domain variables, *Pairs* is a list of Key-Num pairs, where Key is an integer and Num is a finite domain variable. The constraint holds iff each V in *Vs* is equal to some key, and for each Key-Num pair in *Pairs*, the number of occurrences of Key in *Vs* is Num. *Options* is a list of options. Supported options are:

consistency(*value*)

A weaker form of consistency is used.

cost(*Cost, Matrix*)

Matrix is a list of rows, one for each variable, in the order they occur in *Vs*. Each of these rows is a list of integers, one for each key, in the order these keys occur in *Pairs*. When variable v_i is assigned the value of key k_j, then the associated cost is *Matrix*_{ij}. *Cost* is the sum of all costs.

circuit(+*Vs***)**
True iff the list *Vs* of finite domain variables induces a Hamiltonian circuit. The k-th element of *Vs* denotes the successor of node k. Node indexing starts with 1. Examples:

```
?- length(Vs, _), circuit(Vs), label(Vs).
Vs = [] ;
Vs = [1] ;
Vs = [2, 1] ;
Vs = [2, 3, 1] ;
Vs = [3, 1, 2] ;
Vs = [2, 3, 4, 1] .
```

cumulative(+*Tasks***)**
Equivalent to `cumulative(Tasks, [limit(1)])`. See `cumulative/2`.

cumulative(+*Tasks, +Options***)**
Schedule with a limited resource. *Tasks* is a list of tasks, each of the form `task(S_i, D_i, E_i, C_i, T_i)`. S_i denotes the start time, D_i the positive duration, E_i the end time, C_i the non-negative resource consumption, and T_i the task identifier. Each of these arguments must be a finite domain variable with bounded domain, or an integer. The constraint holds iff at each time slot during the start and end of each task, the total resource consumption of all tasks running at that time does not exceed the global resource limit. *Options* is a list of options. Currently, the only supported option is:

limit(*L***)**
The integer *L* is the global resource limit. Default is 1.

For example, given the following predicate that relates three tasks of durations 2 and 3 to a list containing their starting times:

```
tasks_starts(Tasks, [S1,S2,S3]) :-
        Tasks = [task(S1,3,_,1,_),
                 task(S2,2,_,1,_),
                 task(S3,2,_,1,_)].
```

We can use `cumulative/2` as follows, and obtain a schedule:

```
?- tasks_starts(Tasks, Starts), Starts ins 0..10,
   cumulative(Tasks, [limit(2)]), label(Starts).
Tasks = [task(0, 3, 3, 1, _G36), task(0, 2, 2, 1, _G45), ...],
Starts = [0, 0, 2] .
```

disjoint2(+*Rectangles***)**
True iff *Rectangles* are not overlapping. *Rectangles* is a list of terms of the form F(X_i, W_i, Y_i, H_i), where F is any functor, and the arguments are finite domain variables or integers that denote, respectively, the X coordinate, width, Y coordinate and height of each rectangle.

automaton(*+Vs, +Nodes, +Arcs*)

Describes a list of finite domain variables with a finite automaton. Equivalent to `automaton(Vs, _, Vs, Nodes, Arcs, [], [], _)`, a common use case of `automaton/8`. In the following example, a list of binary finite domain variables is constrained to contain at least two consecutive ones:

```
two_consecutive_ones(Vs) :-
        automaton(Vs, [source(a),sink(c)],
                  [arc(a,0,a), arc(a,1,b),
                   arc(b,0,a), arc(b,1,c),
                   arc(c,0,c), arc(c,1,c)]).
```

Example query:

```
?- length(Vs, 3), two_consecutive_ones(Vs), label(Vs).
Vs = [0, 1, 1] ;
Vs = [1, 1, 0] ;
Vs = [1, 1, 1].
```

automaton(*+Sequence, ?Template, +Signature, +Nodes, +Arcs, +Counters, +Initials, ?Finals*)

Describes a list of finite domain variables with a finite automaton. True iff the finite automaton induced by *Nodes* and *Arcs* (extended with *Counters*) accepts *Signature*. *Sequence* is a list of terms, all of the same shape. Additional constraints must link *Sequence* to *Signature*, if necessary. *Nodes* is a list of `source(Node)` and `sink(Node)` terms. *Arcs* is a list of `arc(Node,Integer,Node)` and `arc(Node,Integer,Node,Exprs)` terms that denote the automaton's transitions. Each node is represented by an arbitrary term. Transitions that are not mentioned go to an implicit failure node. *Exprs* is a list of arithmetic expressions, of the same length as *Counters*. In each expression, variables occurring in *Counters* symbolically refer to previous counter values, and variables occurring in *Template* refer to the current element of *Sequence*. When a transition containing arithmetic expressions is taken, each counter is updated according to the result of the corresponding expression. When a transition without arithmetic expressions is taken, all counters remain unchanged. *Counters* is a list of variables. *Initials* is a list of finite domain variables or integers denoting, in the same order, the initial value of each counter. These values are related to *Finals* according to the arithmetic expressions of the taken transitions.

The following example is taken from Beldiceanu, Carlsson, Debruyne and Petit: "Reformulation of Global Constraints Based on Constraints Checkers", Constraints 10(4), pp 339-362 (2005). It relates a sequence of integers and finite domain variables to its number of inflexions, which are switches between strictly ascending and strictly descending subsequences:

```
sequence_inflexions(Vs, N) :-
        variables_signature(Vs, Sigs),
        automaton(Sigs, _, Sigs,
                  [source(s),sink(i),sink(j),sink(s)],
                  [arc(s,0,s), arc(s,1,j), arc(s,2,i),
                   arc(i,0,i), arc(i,1,j,[C+1]), arc(i,2,i),
```

```
                     arc(j,0,j), arc(j,1,j),
                     arc(j,2,i,[C+1])],
                    [C], [0], [N]).

variables_signature([], []).
variables_signature([V|Vs], Sigs) :-
        variables_signature_(Vs, V, Sigs).

variables_signature_([], _, []).
variables_signature_([V|Vs], Prev, [S|Sigs]) :-
        V #= Prev #<==> S #= 0,
        Prev #< V #<==> S #= 1,
        Prev #> V #<==> S #= 2,
        variables_signature_(Vs, V, Sigs).
```

Example queries:

```
?- sequence_inflexions([1,2,3,3,2,1,3,0], N).
N = 3.

?- length(Ls, 5), Ls ins 0..1,
   sequence_inflexions(Ls, 3), label(Ls).
Ls = [0, 1, 0, 1, 0] ;
Ls = [1, 0, 1, 0, 1].
```

chain(+*Zs*, +*Relation*)
Zs form a chain with respect to *Relation*. *Zs* is a list of finite domain variables that are a chain with respect to the partial order *Relation*, in the order they appear in the list. *Relation* must be #=, #=<, #>=, #< or #>. For example:

```
?- chain([X,Y,Z], #>=).
X#>=Y,
Y#>=Z.
```

Reification predicates

Many CLP(FD) constraints can be *reified*. This means that their truth value is itself turned into a CLP(FD) variable, so that we can explicitly reason about whether a constraint holds or not. See reification (section A.9.12).

+*Q*
Q does *not* hold. See reification (section A.9.12).

For example, to obtain the complement of a domain:

```
?- #\ X in -3..0\/10..80.
X in inf.. -4\/1..9\/81..sup.
```

?P **#<==>** *?Q*

P and *Q* are equivalent. See reification (section A.9.12).

For example:

```
?- X #= 4 #<==> B, X #\= 4.
B = 0,
X in inf..3\/5..sup.
```

The following example uses reified constraints to relate a list of finite domain variables to the number of occurrences of a given value:

```
vs_n_num(Vs, N, Num) :-
        maplist(eq_b(N), Vs, Bs),
        sum(Bs, #=, Num).

eq_b(X, Y, B) :- X #= Y #<==> B.
```

Sample queries and their results:

```
?- Vs = [X,Y,Z], Vs ins 0..1, vs_n_num(Vs, 4, Num).
Vs = [X, Y, Z],
Num = 0,
X in 0..1,
Y in 0..1,
Z in 0..1.

?- vs_n_num([X,Y,Z], 2, 3).
X = 2,
Y = 2,
Z = 2.
```

?P **#==>** *?Q*

P implies *Q*. See reification (section A.9.12).

?P **#<==** *?Q*

Q implies *P*. See reification (section A.9.12).

?P **#/** *?Q*

P and *Q* hold. See reification (section A.9.12).

?P **#\\/** *?Q*

P or *Q* holds. See reification (section A.9.12).

For example, the sum of natural numbers below 1000 that are multiples of 3 or 5:

```
?- findall(N, (N mod 3 #= 0 #\/ N mod 5 #= 0, N in 0..999,
               indomain(N)),
```

```
             Ns),
    sum(Ns, #=, Sum).
Ns = [0, 3, 5, 6, 9, 10, 12, 15, 18|...],
Sum = 233168.
```

?P **#** *?Q*
Either *P* holds or *Q* holds, but not both. See reification (section A.9.12).

zcompare(*?Order, ?A, ?B*)
Analogous to `compare/3`, with finite domain variables *A* and *B*.

Think of `zcompare/3` as *reifying* an arithmetic comparison of two integers. This means that we can explicitly reason about the different cases *within* our programs. As in `compare/3`, the atoms <, > and = denote the different cases of the trichotomy. In contrast to `compare/3` though, `zcompare/3` works correctly for *all modes*, also if only a subset of the arguments is instantiated. This allows you to make several predicates over integers deterministic while preserving their generality and completeness. For example:

```
n_factorial(N, F) :-
        zcompare(C, N, 0),
        n_factorial_(C, N, F).

n_factorial_(=, _, 1).
n_factorial_(>, N, F) :-
        F #= F0*N,
        N1 #= N - 1,
        n_factorial(N1, F0).
```

This version of `n_factorial/2` is deterministic if the first argument is instantiated, because argument indexing can distinguish the different clauses that reflect the possible and admissible outcomes of a comparison of *N* against 0. Example:

```
?- n_factorial(30, F).
F = 265252859812191058636308480000000.
```

Since there is no clause for <, the predicate automatically *fails* if *N* is less than 0. The predicate can still be used in all directions, including the most general query:

```
?- n_factorial(N, F).
N = 0,
F = 1 ;
N = F, F = 1 ;
N = F, F = 2 .
```

In this case, all clauses are tried on backtracking, and `zcompare/3` ensures that the respective ordering between N and 0 holds in each case.

The truth value of a comparison can also be reified with (#<==>)/2 in combination with one of the *arithmetic constraints* (section A.9.2). See reification (section A.9.12). However, `zcompare/3` lets you more conveniently distinguish the cases.

Reflection predicates

Reflection predicates let us obtain, in a well-defined way, information that is normally internal to this library. In addition to the predicates explained below, also take a look at `call_residue_vars/2` and `copy_term/3` to reason about CLP(FD) constraints that arise in programs. This can be useful in program analyzers and declarative debuggers.

fd_var(+*Var*)
True iff *Var* is a CLP(FD) variable.

fd_inf(+*Var*, *-Inf*)
Inf is the infimum of the current domain of *Var*.

fd_sup(+*Var*, *-Sup*)
Sup is the supremum of the current domain of *Var*.

fd_size(+*Var*, *-Size*)
Reflect the current size of a domain. *Size* is the number of elements of the current domain of *Var*, or the atom **sup** if the domain is unbounded.

fd_dom(+*Var*, *-Dom*)
Dom is the current domain (see `in/2`) of *Var*. This predicate is useful if you want to reason about domains. It is *not* needed if you only want to display remaining domains; instead, separate your model from the search part and let the toplevel display this information via residual goals.

For example, to implement a custom labeling strategy, you may need to inspect the current domain of a finite domain variable. With the following code, you can convert a *finite* domain to a list of integers:

```
dom_integers(D, Is) :- phrase(dom_integers_(D), Is).

dom_integers_(I)       --> { integer(I) }, [I].
dom_integers_(L..U)    --> { numlist(L, U, Is) }, Is.
dom_integers_(D1\/D2) --> dom_integers_(D1), dom_integers_(D2).
```

Example:

```
?- X in 1..5, X #\= 4, fd_dom(X, D), dom_integers(D, Is).
D = 1..3\/5,
Is = [1,2,3,5],
X in 1..3\/5.
```

fd_degree(+*Var*, *-Degree*) *[det]*
Degree is the number of constraints currently attached to *Var*.

FD set predicaes

These predicates allow operating directly on the internal representation of CLP(FD) domains. In this context, such an internal domain representation is called an **FD set**.

Note that the exact term representation of FD sets is unspecified and will vary across CLP(FD) implementations or even different versions of the same implementation. FD set terms should be manipulated **only** using the predicates in this section. The behavior of other operations on FD set terms is undefined. In particular, you should **not** construct or deconstruct FD sets by unification, and you **cannot** reliably compare FD sets using unification or generic term equality/comparison predicates.

?Var **in_set** *+Set*
: *Var* is an element of the FD set *Set*.

fd_set(*?Var, -Set***)** *[det]*
: *Set* is the FD set representation of the current domain of *Var*.

is_fdset(*@Set***)** *[semidet]*
: *Set* is currently bound to a valid FD set.

empty_fdset(*-Set***)** *[det]*
: *Set* is the empty FD set.

fdset_parts(*?Set, ?Min, ?Max, ?Rest***)** *[semidet]*
: *Set* is a non-empty FD set representing the domain *Min..Max* `\/` *Rest*, where *Min..Max* is a non-empty interval (see `fdset_interval/3`) and *Rest* is another FD set (possibly empty).

  If *Max* is **sup**, then *Rest* is the empty FD set. Otherwise, if *Rest* is non-empty, all elements of *Rest* are greater than *Max*+1.

  This predicate should only be called with either *Set* or all other arguments being ground.

empty_interval(*+Min, +Max***)** *[semidet]*
: *Min..Max* is an empty interval. *Min* and *Max* are integers or one of the atoms **inf** or **sup**.

fdset_interval(*?Interval, ?Min, ?Max***)** *[semidet]*
: *Interval* is a non-empty FD set consisting of the single interval *Min..Max*. *Min* is an integer or the atom **inf** to denote negative infinity. *Max* is an integer or the atom **sup** to denote positive infinity.

  Either *Interval* or *Min* and *Max* must be ground.

fdset_singleton(*?Set, ?Elt***)** *[semidet]*
: *Set* is the FD set containing the single integer *Elt*.

  Either *Set* or *Elt* must be ground.

fdset_min(*+Set, -Min***)** *[semidet]*
: *Min* is the lower bound (infimum) of the non-empty FD set *Set*. *Min* is an integer or the atom **inf** if *Set* has no lower bound.

fdset_max(*+Set, -Max***)** *[semidet]*
: *Max* is the upper bound (supremum) of the non-empty FD set *Set*. *Max* is an integer or the atom **sup** if *Set* has no upper bound.

fdset_size(*+Set, -Size*) *[det]*
Size is the number of elements of the FD set *Set*, or the atom **sup** if *Set* is infinite.

list_to_fdset(*+List, -Set*) *[det]*
Set is an FD set containing all elements of *List*, which must be a list of integers.

fdset_to_list(*+Set, -List*) *[det]*
List is a list containing all elements of the finite FD set *Set*, in ascending order.

range_to_fdset(*+Domain, -Set*) *[det]*
Set is an FD set equivalent to the domain *Domain*. *Domain* uses the same syntax as accepted by (in)/2.

fdset_to_range(*+Set, -Domain*) *[det]*
Domain is a domain equivalent to the FD set *Set*. *Domain* is returned in the same format as by `fd_dom/2`.

fdset_add_element(*+Set1, +Elt, -Set2*) *[det]*
Set2 is the same FD set as *Set1*, but with the integer *Elt* added. If *Elt* is already in *Set1*, the set is returned unchanged.

fdset_del_element(*+Set1, +Elt, -Set2*) *[det]*
Set2 is the same FD set as *Set1*, but with the integer *Elt* removed. If *Elt* is not in *Set1*, the set returned unchanged.

fdset_disjoint(*+Set1, +Set2*) *[semidet]*
The FD sets *Set1* and *Set2* have no elements in common.

fdset_intersect(*+Set1, +Set2*) *[semidet]*
The FD sets *Set1* and *Set2* have at least one element in common.

fdset_intersection(*+Set1, +Set2, -Intersection*) *[det]*
Intersection is an FD set (possibly empty) of all elements that the FD sets *Set1* and *Set2* have in common.

fdset_member(*?Elt, +Set*) *[nondet]*
The integer *Elt* is a member of the FD set *Set*. If *Elt* is unbound, *Set* must be finite and all elements are enumerated on backtracking.

fdset_eq(*+Set1, +Set2*) *[semidet]*
True if the FD sets *Set1* and *Set2* are equal, i. e. contain exactly the same elements. This is not necessarily the same as unification or a term equality check, because some FD sets have multiple possible term representations.

fdset_subset(*+Set1, +Set2*) *[semidet]*
The FD set *Set1* is a (non-strict) subset of *Set2*, i. e. every element of *Set1* is also in *Set2*.

fdset_subtract(*+Set1, +Set2, -Difference*) *[det]*
The FD set *Difference* is *Set1* with all elements of *Set2* removed, i. e. the set difference of *Set1* and *Set2*.

fdset_union(+*Set1, +Set2, -Union*) *[det]*
The FD set *Union* is the union of FD sets *Set1* and *Set2*.

fdset_union(+*Sets, -Union*) *[det]*
The FD set *Union* is the n-ary union of all FD sets in the list *Sets*. If *Sets* is empty, *Union* is the empty FD set.

fdset_complement(+*Set, -Complement*) *[det]*
The FD set *Complement* is the complement of the FD set *Set*. Equivalent to `fdset_subtract(inf..sup, Set, Complement)`.

A.9.18 Closing and opening words about CLP(FD)

CLP(FD) constraints are one of the main reasons why logic programming approaches are picked over other paradigms for solving many tasks of high practical relevance. The usefulness of CLP(FD) constraints for scheduling, allocation and combinatorial optimization tasks is well-known both in academia and industry.

With this library, we take the applicability of CLP(FD) constraints one step further, following the road that visionary systems like SICStus Prolog have already clearly outlined: This library is designed to completely subsume and *replace* low-level predicates over integers, which were in the past repeatedly found to be a major stumbling block when introducing logic programming to beginners.

Embrace the change and new opportunities that this paradigm allows! Use CLP(FD) constraints in your programs. The use of CLP(FD) constraints instead of low-level arithmetic is also a good indicator to judge the quality of any introductory Prolog text.

A.10 library(clpqr): Constraint Logic Programming over Rationals and Reals

Author: *Christian Holzbaur*, ported to SWI-Prolog by *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.[1] 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 explicitly before using it:

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

or

[1] http://www.ai.univie.ac.at/cgi-bin/tr-online?number+95-09

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

A.10.1 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 equating 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 been replaced by *NewVars*. This operation does not change the constraint store. E.g. in

⟨*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
	|	+⟨*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

```
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.

A.10.2 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.10.3 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:

$A = B * C$	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)
$X = pow(Y, Z)$ $X = exp(Y, Z)$ $X = Y \text{ ^ } Z$	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

- *Unification with a variable*

```
{X = Y}
X = Y
```

- *Unification with a number*

```
{X = 5.0}
X = 5.0
```

A.10.4 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.10.5 Status and known problems

The clpq and clpr libraries are 'orphaned', i.e., they currently have no maintainer.

- *Top-level output*
  The top-level output may contain variables not present in the original query:

```
?- {X+Y>=1}.
{Y=1-X+_G2160, _G2160>=0}.

?-
```

Nonetheless, for linear constraints this kind of answer means unconditional satisfiability.

- *Dumping constraints*
  The first argument of `dump/3` has to be a list of free variables at call-time:

```
?- {X=1},dump([X],[Y],L).
ERROR: Unhandled exception: Unknown message:
       instantiation_error(dump([1],[_G11],_G6),1)
?-
```

A.11 library(csv): Process CSV (Comma-Separated Values) data

See also RFC 4180

To be done
- Implement immediate assert of the data to avoid possible stack overflows.
- Writing creates an intermediate code-list, possibly overflowing resources. This waits for pure output!

This library parses and generates CSV data. CSV data is represented in Prolog as a list of rows. Each row is a compound term, where all rows have the same name and arity.

csv_read_file(*+File, -Rows*) *[det]*
csv_read_file(*+File, -Rows, +Options*) *[det]*

Read a CSV file into a list of rows. Each row is a Prolog term with the same arity. *Options* is handed to `csv//2`. Remaining options are processed by `phrase_from_file/3`. The default separator depends on the file name extension and is `\t` for `.tsv` files and `,` otherwise.

Suppose we want to create a predicate `table/6` from a CSV file that we know contains 6 fields per record. This can be done using the code below. Without the option `arity(6)`, this would generate a predicate table/N, where N is the number of fields per record in the data.

```
?- csv_read_file(File, Rows, [functor(table), arity(6)]),
   maplist(assert, Rows).
```

csv_read_stream(*+Stream, -Rows, +Options*) *[det]*

Read CSV data from *Stream*. See also `csv_read_row/3`.

csv(*?Rows*) **//** *[det]*
csv(*?Rows, +Options*) **//** *[det]*

Prolog DCG to 'read/write' CSV data. *Options*:

separator(*+Code*)
The comma-separator. Must be a character code. Default is (of course) the comma. Character codes can be specified using the 0' notion. E.g., using `separator(0';)` parses a semicolon separated file.

ignore_quotes(*+Boolean*)
If `true` (default false), threat double quotes as a normal character.

strip(*+Boolean*)
: If `true` (default `false`), strip leading and trailing blank space. RFC4180 says that blank space is part of the data.

skip_header(*+CommentLead*)
: Skip leading lines that start with *CommentLead*. There is no standard for comments in CSV files, but some CSV files have a header where each line starts with `#`. After skipping comment lines this option causes `csv//2` to skip empty lines. Note that an empty line may not contain white space characters (space or tab) as these may provide valid data.

convert(*+Boolean*)
: If `true` (default), use `name/2` on the field data. This translates the field into a number if possible.

case(*+Action*)
: If `down`, downcase atomic values. If `up`, upcase them and if `preserve` (default), do not change the case.

functor(*+Atom*)
: Functor to use for creating row terms. Default is `row`.

arity(*?Arity*)
: Number of fields in each row. This predicate raises a `domain_error(row_arity(Expected), Found)` if a row is found with different arity.

match_arity(*+Boolean*)
: If `false` (default `true`), do not reject CSV files where lines provide a varying number of fields (columns). This can be a work-around to use some incorrect CSV files.

csv_read_file_row(*+File, -Row, +Options*) *[nondet]*
: True when *Row* is a row in *File*. First unifies *Row* with the first row in *File*. Backtracking yields the second, ... row. This interface is an alternative to `csv_read_file/3` that avoids loading all rows in memory. Note that this interface does not guarantee that all rows in *File* have the same arity.

In addition to the options of `csv_read_file/3`, this predicate processes the option:

line(*-Line*)
: *Line* is unified with the 1-based line-number from which *Row* is read. Note that *Line* is not the physical line, but rather the *logical* record number.

csv_read_row(*+Stream, -Row, +CompiledOptions*) *[det]*
: Read the next CSV record from *Stream* and unify the result with *Row*. *CompiledOptions* is created from options defined for `csv//2` using `csv_options/2`. *Row* is unified with `end_of_file` upon reaching the end of the input.

csv_options(*-Compiled, +Options*) *[det]*
: *Compiled* is the compiled representation of the CSV processing options as they may be passed into `csv//2`, etc. This predicate is used in combination with `csv_read_row/3` to avoid repeated processing of the options.

csv_write_file(+*File*, +*Data*) *[det]*
csv_write_file(+*File*, +*Data*, +*Options*) *[det]*

Write a list of Prolog terms to a CSV file. *Options* are given to `csv//2`. Remaining options are given to `open/4`. The default separator depends on the file name extension and is `\t` for `.tsv` files and `,` otherwise.

csv_write_stream(+*Stream*, +*Data*, +*Options*) *[det]*

Write the rows in *Data* to *Stream*. This is similar to `csv_write_file/3`, but can deal with data that is produced incrementally. The example below saves all answers from the predicate `data/3` to File.

```
save_data(File) :-
   setup_call_cleanup(
       open(File, write, Out),
       forall(data(C1,C2,C3),
              csv_write_stream(Out, [row(C1,C2,C3)], [])),
       close(Out)),
```

A.12 library(dcg/basics): Various general DCG utilities

To be done This is just a starting point. We need a comprehensive set of generally useful DCG primitives.

This library provides various commonly used DCG primitives acting on list of character **codes**. Character classification is based on `code_type/2`.

This module started its life as `library(http/dcg_basics)` to support the HTTP protocol. Since then, it was increasingly used in code that has no relation to HTTP and therefore this library was moved to the core library.

string_without(+*EndCodes*, -*Codes*) `//` *[det]*

Take as many codes from the input until the next character code appears in the list *EndCodes*. The terminating code itself is left on the input. Typical use is to read upto a defined delimiter such as a newline or other reserved character. For example:

```
    ...,
    string_without("\n", RestOfLine)
```

Arguments

EndCodes	is a list of character codes.

See also `string//1`.

string(-*Codes*) `//` *[nondet]*

Take as few as possible tokens from the input, taking one more each time on backtracking. This code is normally followed by a test for a delimiter. For example:

```
upto_colon(Atom) -->
        string(Codes), ":", !,
        { atom_codes(Atom, Codes) }.
```

See also `string_without//2`.

blanks // *[det]*
Skip zero or more white-space characters.

blank // *[semidet]*
Take next `space` character from input. Space characters include newline.

See also `white//0`

nonblanks(*-Codes*) **//** *[det]*
Take all `graph` characters

nonblank(*-Code*) **//** *[semidet]*
Code is the next non-blank (`graph`) character.

blanks_to_nl // *[semidet]*
Take a sequence of `blank//0` codes if blanks are followed by a newline or end of the input.

whites // *[det]*
Skip white space *inside* a line.

See also `blanks//0` also skips newlines.

white // *[semidet]*
Take next `white` character from input. White characters do *not* include newline.

alpha_to_lower(*?C*) **//** *[semidet]*
Read a letter (class `alpha`) and return it as a lowercase letter. If *C* is instantiated and the DCG list is already bound, *C* must be `lower` and matches both a lower and uppercase letter. If the output list is unbound, its first element is bound to *C*. For example:

```
?- alpha_to_lower(0'a, `AB`, R).
R = [66].
?- alpha_to_lower(C, `AB`, R).
C = 97, R = [66].
?- alpha_to_lower(0'a, L, R).
L = [97|R].
```

digits(*?Chars*) **//** *[det]*
digit(*?Char*) **//** *[det]*
integer(*?Integer*) **//** *[det]*
Number processing. The predicate `digits//1` matches a possibly empty set of digits, `digit//1` processes a single digit and integer processes an optional sign followed by a non-empty sequence of digits into an integer.

float(*?Float*) **//** *[det]*

Process a floating point number. The actual conversion is controlled by `number_codes/2`.

number(*+Number*) **//** *[det]*
number(*-Number*) **//** *[semidet]*

Generate extract a number. Handles both integers and floating point numbers.

xinteger(*+Integer*) **//** *[det]*
xinteger(*-Integer*) **//** *[semidet]*

Generate or extract an integer from a sequence of hexadecimal digits. Hexadecimal characters include both uppercase (A-F) and lowercase (a-f) letters. The value may be preceded by a sign (+/-)

xdigit(*-Weight*) **//** *[semidet]*

True if the next code is a hexdecimal digit with *Weight*. *Weight* is between 0 and 15. Hexadecimal characters include both uppercase (A-F) and lowercase (a-f) letters.

xdigits(*-WeightList*) **//** *[det]*

List of weights of a sequence of hexadecimal codes. *WeightList* may be empty. Hexadecimal characters include both uppercase (A-F) and lowercase (a-f) letters.

eol

Matches end-of-line. Matching `\r\n`, `\n` or end of input (`eos//0`).

eos

Matches end-of-input. The implementation behaves as the following portable implementation:

```
eos --> call(eos_).
eos_([], []).
```

To be done This is a difficult concept and violates the *context free* property of DCGs. Explain the exact problems.

remainder(*-List*) **//**

Unify *List* with the remainder of the input.

prolog_var_name(*-Name:atom*) **//** *[semidet]*

Matches a Prolog variable name. Primarily intended to deal with quasi quotations that embed Prolog variables.

atom(*++Atom*) **//** *[det]*

Generate codes of *Atom*. Current implementation uses `write/1`, dealing with any Prolog term. *Atom* must be ground though.

A.13 library(dcg/high_order): High order grammar operations

This library provides facilities comparable `maplist/3`, `ignore/1` and `foreach/2` for DCGs.

STATUS: This library is experimental. The interface and implementation may change based on feedback. Please send feedback to the mailinglist or the issue page of the `swipl-devel.git` repository.

sequence(*:Element, ?List*) **//** *[nondet]*

Match or generate a sequence of *Element*. This predicate is deterministic if *List* is fully instantiated and *Element* is deterministic. When parsing, this predicate is *gready* and does not prune choice points. For example:

```
?- phrase(sequence(digit, Digits), `123a`, L).
Digits = "123",
L = [97] ;
Digits = [49, 50],
L = [51, 97] ;
...
```

sequence(*:Element, :Sep, ?List*) **//** *[nondet]*

Match or generate a sequence of *Element* where each pair of elements is separated by *Sep*. When *parsing*, a matched *Sep commits*. The final element is *not* committed. More formally, it matches the following sequence:

```
Element?, (Sep,Element)*
```

See also `sequence//5`.

sequence(*:Start, :Element, :Sep, :End, ?List*) **//** *[semidet]*

Match or generate a sequence of *Element* enclosed by *Start* end *End*, where each pair of elements is separated by *Sep*. More formally, it matches the following sequence:

```
Start, Element?, (Sep,Element)*, End
```

The example below matches a Prolog list of integers:

```
?- phrase(sequence(("[",blanks),
                   number, (",",blanks),
                   (blanks,"]"), L),
                   `[1, 2, 3 ] a`, Tail).
L = [1, 2, 3],
Tail = [32, 97].
```

optional(*:Match, :Default*) **//** *[det]*

Perform an optional match, executing *Default* if *Match* is not matched. This is comparable to `ignore/1`. Both *Match* and *Default* are DCG body terms. *Default* is typically used to instantiate the output variables of *Match*, but may also be used to match a default representation. Using `[]` for *Default* succeeds without any additional actions if *Match* fails. For example:

```
?- phrase(optional(number(X), {X=0}), `23`, Tail).
X = 23,
Tail = [].
?- phrase(optional(number(X), {X=0}), `aap`, Tail).
```

```
X = 0,
Tail = `aap`.
```

foreach(*:Generator, :Element*) **//** *[det]*
foreach(*:Generator, :Element, :Sep*) **//** *[det]*

Generate a list from the solutions of *Generator*. This predicate collects all solutions of *Generator*, applies *Element* for each solution and *Sep between* each pair of solutions. For example:

```
?- phrase(foreach(between(1,5,X), number(X), ", "), L).
L = "1, 2, 3, 4, 5".
```

A.14 library(debug): Print debug messages and test assertions

author Jan Wielemaker

This library is a replacement for `format/3` for printing debug messages. Messages are assigned a *topic*. By dynamically enabling or disabling topics the user can select desired messages. Debug statements are removed when the code is compiled for optimization.

See manual for details. With XPCE, you can use the call below to start a graphical monitoring tool.

```
?- prolog_ide(debug_monitor).
```

Using the predicate `assertion/1` you can make assumptions about your program explicit, trapping the debugger if the condition does not hold.

debugging(*+Topic*) *[semidet]*
debugging(*-Topic*) *[nondet]*
debugging(*?Topic, ?Bool*) *[nondet]*

Examine debug topics. The form `debugging(+Topic)` may be used to perform more complex debugging tasks. A typical usage skeleton is:

```
        (   debugging(mytopic)
        ->  <perform debugging actions>
        ;   true
        ),
        ...
```

The other two calls are intended to examine existing and enabled debugging tokens and are typically not used in user programs.

debug(*+Topic*) *[det]*
nodebug(*+Topic*) *[det]*

Add/remove a topic from being printed. `nodebug(_)` removes all topics. Gives a warning if the topic is not defined unless it is used from a directive. The latter allows placing debug topics at the start of a (load-)file without warnings.

For `debug/1`, *Topic* can be a term *Topic* > Out, where Out is either a stream or stream-alias or a filename (atom). This redirects debug information on this topic to the given output.

list_debug_topics *[det]*
List currently known debug topics and their setting.

debug_message_context(+*What*) *[det]*
Specify additional context for debug messages.

> **deprecated** New code should use the Prolog flag `message_context`. This predicates adds or deletes topics from this list.

debug(+*Topic*, +*Format*, :*Args*) *[det]*
Format a message if debug topic is enabled. Similar to `format/3` to `user_error`, but only prints if *Topic* is activated through `debug/1`. *Args* is a meta-argument to deal with goal for the @-command. Output is first handed to the hook `prolog:debug_print_hook/3`. If this fails, *Format*+*Args* is translated to text using the message-translation (see `print_message/2`) for the term `debug(Format, Args)` and then printed to every matching destination (controlled by `debug/1`) using `print_message_lines/3`.

The message is preceded by '% ' and terminated with a newline.

> **See also** `format/3`.

prolog:debug_print_hook(+*Topic*, +*Format*, +*Args*) *[semidet,multifile]*
Hook called by `debug/3`. This hook is used by the graphical frontend that can be activated using `prolog_ide/1`:

```
?- prolog_ide(debug_monitor).
```

assertion(:*Goal*) *[det]*
Acts similar to C `assert()` macro. It has no effect if *Goal* succeeds. If *Goal* fails or throws an exception, the following steps are taken:

- call `prolog:assertion_failed/2`. If `prolog:assertion_failed/2` fails, then:
  - If this is an interactive toplevel thread, print a message, the stack-trace, and finally trap the debugger.
  - Otherwise, throw `error(assertion_error(Reason, G),_)` where Reason is one of `fail` or the exception raised.

prolog:assertion_failed(+*Reason*, +*Goal*) *[semidet,multifile]*
This hook is called if the *Goal* of `assertion/1` fails. *Reason* is unified with either `fail` if *Goal* simply failed or an exception call otherwise. If this hook fails, the default behaviour is activated. If the hooks throws an exception it will be propagated into the caller of `assertion/1`.

A.15 library(dicts): Dict utilities

This library defines utilities that operate on lists of dicts, notably to make lists of dicts consistent by adding missing keys, converting between lists of compounds and lists of dicts, joining and slicing lists of dicts.

dicts_same_tag(*+List, -Tag*) *[semidet]*
True when *List* is a list of dicts that all have the tag *Tag*.

dict_size(*+Dict, -KeyCount*) *[det]*
True when *KeyCount* is the number of keys in *Dict*.

dict_keys(*+Dict, -Keys*) *[det]*
True when *Keys* is an ordered set of the keys appearing in *Dict*.

dicts_same_keys(*+List, -Keys*) *[semidet]*
True if *List* is a list of dicts that all have the same keys and *Keys* is an ordered set of these keys.

dicts_to_same_keys(*+DictsIn, :OnEmpty, -DictsOut*)
DictsOut is a copy of *DictsIn*, where each dict contains all keys appearing in all dicts of *DictsIn*. Values for keys that are added to a dict are produced by calling *OnEmpty* as below. The predicate `dict_fill/4` provides an implementation that fills all new cells with a predefined value.

```
call(:OnEmpty, +Key, +Dict, -Value)
```

dict_fill(*+ValueIn, +Key, +Dict, -Value*) *[det]*
Implementation for the `dicts_to_same_keys/3` *OnEmpty* closure that fills new cells with a copy of *ValueIn*. Note that `copy_term/2` does not really copy ground terms. Below are two examples. Note that when filling empty cells with a variable, each empty cell is bound to a new variable.

```
?- dicts_to_same_keys([r{x:1}, r{y:2}], dict_fill(null), L).
L = [r{x:1, y:null}, r{x:null, y:2}].
?- dicts_to_same_keys([r{x:1}, r{y:2}], dict_fill(_), L).
L = [r{x:1, y:_G2005}, r{x:_G2036, y:2}].
```

Use `dict_no_fill/3` to raise an error if a dict is missing a key.

dicts_join(*+Key, +DictsIn, -Dicts*) *[semidet]*
Join dicts in *Dicts* that have the same value for *Key*, provided they do not have conflicting values on other keys. For example:

```
?- dicts_join(x, [r{x:1, y:2}, r{x:1, z:3}, r{x:2,y:4}], L).
L = [r{x:1, y:2, z:3}, r{x:2, y:4}].
```

Errors `existence_error(key, Key, Dict)` if a dict in Dicts1 or Dicts2 does not contain *Key*.

dicts_join(*+Key, +Dicts1, +Dicts2, -Dicts*) *[semidet]*

Join two lists of dicts (*Dicts1* and *Dicts2*) on *Key*. Each pair D1-D2 from *Dicts1* and *Dicts2* that have the same (==) value for *Key* creates a new dict D with the union of the keys from D1 and D2, provided D1 and D2 to not have conflicting values for some key. For example:

```
?- DL1 = [r{x:1,y:1},r{x:2,y:4}],
   DL2 = [r{x:1,z:2},r{x:3,z:4}],
   dicts_join(x, DL1, DL2, DL).
   DL = [r{x:1, y:1, z:2}, r{x:2, y:4}, r{x:3, z:4}].
```

Errors `existence_error(key, Key, Dict)` if a dict in *Dicts1* or *Dicts2* does not contain *Key*.

dicts_slice(*+Keys, +DictsIn, -DictsOut*) *[det]*

DictsOut is a list of Dicts only containing values for *Keys*.

dicts_to_compounds(*?Dicts, +Keys, :OnEmpty, ?Compounds*) *[semidet]*

True when *Dicts* and *Compounds* are lists of the same length and each element of *Compounds* is a compound term whose arguments represent the values associated with the corresponding keys in *Keys*. When converting from dict to row, *OnEmpty* is used to compute missing values. The functor for the compound is the same as the tag of the pair. When converting from dict to row and the dict has no tag, the functor `row` is used. For example:

```
?- Dicts = [_{x:1}, _{x:2, y:3}],
   dicts_to_compounds(Dicts, [x], dict_fill(null), Compounds).
Compounds = [row(1), row(2)].
?- Dicts = [_{x:1}, _{x:2, y:3}],
   dicts_to_compounds(Dicts, [x,y], dict_fill(null), Compounds).
Compounds = [row(1, null), row(2, 3)].
?- Compounds = [point(1,1), point(2,4)],
   dicts_to_compounds(Dicts, [x,y], dict_fill(null), Compounds).
Dicts = [point{x:1, y:1}, point{x:2, y:4}].
```

When converting from *Dicts* to *Compounds Keys* may be computed by `dicts_same_keys/2`.

A.16 library(error): Error generating support

author
- Jan Wielemaker
- Richard O'Keefe
- Ulrich Neumerkel

See also
- `library(debug)` and `library(prolog_stack)`.
- `print_message/2` is used to print (uncaught) error terms.

This module provides predicates to simplify error generation and checking. It's implementation is based on a discussion on the SWI-Prolog mailinglist on best practices in error handling. The utility predicate `must_be/2` provides simple run-time type validation. The *_error predicates are simple wrappers around `throw/1` to simplify throwing the most common ISO error terms.

type_error(+*ValidType*, +*Culprit*)
Tell the user that *Culprit* is not of the expected *ValidType*. This error is closely related to `domain_error/2` because the notion of types is not really set in stone in Prolog. We introduce the difference using a simple example.

Suppose an argument must be a non-negative integer. If the actual argument is not an integer, this is a *type_error*. If it is a negative integer, it is a *domain_error*.

Typical borderline cases are predicates accepting a compound term, e.g., `point(X,Y)`. One could argue that the basic type is a compound-term and any other compound term is a domain error. Most Prolog programmers consider each compound as a type and would consider a compound that is not `point(_,_)` a *type_error*.

domain_error(+*ValidDomain*, +*Culprit*)
The argument is of the proper type, but has a value that is outside the supported values. See `type_error/2` for a more elaborate discussion of the distinction between type- and domain-errors.

existence_error(+*ObjectType*, +*Culprit*)
Culprit is of the correct type and correct domain, but there is no existing (external) resource of type *ObjectType* that is represented by it.

existence_error(+*ObjectType*, +*Culprit*, +*Set*)
Culprit is of the correct type and correct domain, but there is no existing (external) resource of type *ObjectType* that is represented by it in the provided set. The thrown exception term carries a formal term structured as follows: `existence_error(ObjectType, Culprit, Set)`

Compatibility This error is outside the ISO Standard.

permission_error(+*Operation*, +*PermissionType*, +*Culprit*)
It is not allowed to perform *Operation* on (whatever is represented by) *Culprit* that is of the given *PermissionType* (in fact, the ISO Standard is confusing and vague about these terms' meaning).

instantiation_error(+*FormalSubTerm*)
An argument is under-instantiated. I.e. it is not acceptable as it is, but if some variables are bound to appropriate values it would be acceptable.

	Arguments
FormalSubTerm	is the term that needs (further) instantiation. Unfortunately, the ISO error does not allow for passing this term along with the error, but we pass it to this predicate for documentation purposes and to allow for future enhancement.

uninstantiation_error(+*Culprit*)
An argument is over-instantiated. This error is used for output arguments whose value cannot

be known upfront. For example, the goal `open(File, read, input)` cannot succeed because the system will allocate a new unique stream handle that will never unify with `input`.

representation_error(+*Flag*)
A representation error indicates a limitation of the implementation. SWI-Prolog has no such limits that are not covered by other errors, but an example of a representation error in another Prolog implementation could be an attempt to create a term with an arity higher than supported by the system.

syntax_error(+*Culprit*)
A text has invalid syntax. The error is described by *Culprit*. According to the ISO Standard, *Culprit* should be an implementation-dependent atom.

To be done Deal with proper description of the location of the error. For short texts, we allow for Type(Text), meaning Text is not a valid Type. E.g. `syntax_error(number('1a'))` means that `1a` is not a valid number.

resource_error(+*Resource*)
A goal cannot be completed due to lack of resources. According to the ISO Standard, *Resource* should be an implementation-dependent atom.

must_be(+*Type*, @*Term*) *[det]*
True if *Term* satisfies the type constraints for *Type*. Defined types are `atom`, `atomic`, `between`, `boolean`, `callable`, `chars`, `codes`, `text`, `compound`, `constant`, `float`, `integer`, `nonneg`, `positive_integer`, `negative_integer`, `nonvar`, `number`, `oneof`, `list`, `list_or_partial_list`, `symbol`, `var`, `rational`, `encoding`, `dict` and `string`.

Most of these types are defined by an arity-1 built-in predicate of the same name. Below is a brief definition of the other types.

acyclic	Acyclic term (tree); see `acyclic_term/1`
any	any term
`between(FloatL,FloatU)`	Number [FloatL..FloatU]
`between(IntL,IntU)`	Integer [IntL..IntU]
boolean	One of `true` or `false`
char	Atom of length 1
chars	Proper list of 1-character atoms
code	Representation Unicode code point
codes	Proper list of Unicode character codes
constant	Same as `atomic`
cyclic	Cyclic term (rational tree); see `cyclic_term/1`
dict	A dictionary term; see `is_dict/1`
encoding	Valid name for a character encoding; see `current_encoding/1`
list	A (non-open) list; see `is_list/1`
negative_integer	Integer < 0
nonneg	Integer >= 0
`oneof(L)`	Ground term that is member of L
pair	Key-Value pair
positive_integer	Integer > 0
proper_list	Same as list
`list(Type)`	Proper list with elements of *Type*
list_or_partial_list	A list or an open list (ending in a variable); see `is_list_or_partial_list/1`
stream	A stream name or valid stream handle; see `is_stream/1`
symbol	Same as `atom`
text	One of `atom`, `string`, `chars` or `codes`
type	*Term* is a valid type specification

Note: The Windows version can only represent Unicode code points up to 2^16-1. Higher values cause a representation error on most text handling predicates.

throws instantiation_error if *Term* is insufficiently instantiated and `type_error(Type, Term)` if *Term* is not of *Type*.

is_of_type(*+Type, @Term*) *[semidet]*
True if *Term* satisfies *Type*.

has_type(*+Type, @Term*) *[semidet,multifile]*
True if *Term* satisfies *Type*.

current_encoding(*?Name*) *[nondet]*
True if *Name* is the name of a supported encoding. See encoding option of e.g., `open/4`.

current_type(*?Type, @Var, -Body*) *[nondet]*
True when *Type* is a currently defined type and *Var* satisfies *Type* of the body term *Body* succeeds.

A.17 library(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 guarantee 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 advised to avoid the use of reset_gensym or at least to reset only the keys used by your program to avoid unexpected side effects on other components.

A.18 library(increval): Incremental dynamic predicate modification

Compatibility XSB

This module emulates the XSB module `increval`. This module serves two goals: (1) provide alternatives for the dynamic clause manipulation predicates that propagate into the incremental tables and (2) query the dynamically maintained *Incremental Depency Graph* (IDG).

The change propagation for incremental dynamic predicates. SWI-Prolog relies in `prolog_listen/2` to forward any change to dynamic predicates to the table IDG and `incr_assert/1` and friends thus simply call the corresponding database update.

is_incremental_subgoal(*?SubGoal*) *[nondet]*
: This predicate non-deterministically unifies Subgoal with incrementally tabled subgoals that are currently table entries.

incr_directly_depends(*:Goal1, :Goal2*) *[nondet]*
: True if *Goal1* depends on *Goal2* in the IDG.

  Compatibility : In XSB, at least one of Goal 1 or Goal 2 must be bound. This implementation may be used with both arguments unbound.

incr_trans_depends(*:Goal1, Goal2*) *[nondet]*
: True for each pair in the transitive closure of `incr_directly_depends(G1, G2)`.

incr_invalid_subgoals(*-List*) *[det]*
List is a sorted list (set) of the incremental subgoals that are currently invalid.

incr_is_invalid(*:Subgoal*) *[semidet]*
True when *Subgoal*'s table is marked as invalid.

incr_invalidate_calls(*:Goal*) *[det]*
Invalidate all tables for subgoals of *Goal* as well as tables that are affected by these.

incr_invalidate_call(*:Goal*) *[det]*
This is the XSB name, but the manual says `incr_invalidate_calls/1` and the comment with the code suggests this is misnamed.

deprecated Use `incr_invalidate_calls/1`.

incr_table_update
Updated all invalid tables

incr_propagate_calls(*:Answer*) *[det]*
Activate the monotonic answer propagation similarly to when a new fact is asserted for a monotonic dynamic predicate. The *Answer* term must match a monotonic dynamic predicate.

A.19 library(intercept): Intercept and signal interface

This library allows for creating an execution context (goal) which defines how calls to `send_signal/1` are handled. This library is typically used to fetch values from the context or process results depending on the context.

For example, assume we parse a (large) file using a grammar (see `phrase_from_file/3`) that has some sort of *record* structure. What should we do with the recognised records? We can return them in a list, but if the input is large this is a huge overhead if the records are to be asserted or written to a file. Using this interface we can use

```
document -->
    record(Record),
    !,
    { send_signal(record(Record)) },
    document.
document -->
    [].
```

Given the above, we can assert all records into the database using the following query:

```
    ...,
    intercept(phrase_from_file(File, document),
              record(Record),
              assertz(Record)).
```

Or, we can collect all records in a list using `intercept_all/4`:

```
...,
intercept_all(Record,
              phrase_from_file(File, document), record(Record),
              Records).
```

intercept(*:Goal, ?Ball, :Handler*)

Run *Goal* as `call/1`. If somewhere during the execution of *Goal* `send_signal/1` is called with a *Signal* that unifies with *Ball*, run *Handler* and continue the execution.

This predicate is related to `catch/3`, but rather than aborting the execution of *Goal* and running *Handler* it continues the execution of *Goal*. This construct is also related to *delimited continuations* (see `reset/3` and `shift/1`). It only covers one (common) use case for delimited continuations, but does so with a simpler interface, at lower overhead and without suffering from poor interaction with the cut.

Note that *Ball* and *Handler* are *copied* before calling the (copy) of *Handler* to avoid instantiation of *Ball* and/or *Handler* which can make a subsequent signal fail.

See also `intercept/4`, `reset/3`, `catch/4`, `broadcast_request/1`.
Compatibility Ciao

intercept(*:Goal, ?Ball, :Handler, +Arg*)

Similar to `intercept/3`, but the copy of *Handler* is called as `call(Copy,Arg)`, which allows passing large context arguments or arguments subject to unification or *destructive assignment*. For example:

```
?- intercept(send_signal(x), X, Y=X).
true.

?- intercept(send_signal(x), X, =(X), Y).
Y = x.
```

intercept_all(*+Template, :Goal, ?Ball, -List*)

True when *List* contains all instances of *Template* that have been sent using `send_signal/1` where the argument unifies with *Ball*. Note that backtracking in *Goal* resets the *List*. For example, given

```
enum(I, Max) :- I =< Max, !, send_signal(emit(I)),
                I2 is I+1, enum(I2, Max).
enum(_, _).
```

Consider the following queries

```
?- intercept_all(I, enum(1,6), emit(I), List).
List = [1, 2, 3, 4, 5, 6].

?- intercept_all(I, (between(1,3,Max),enum(1,Max)),
                 emit(I), List).
Max = 1, List = [1] ;
Max = 2, List = [1, 2] ;
Max = 3, List = [1, 2, 3].
```

See also nb_intercept_all/4

nb_intercept_all(+*Template, :Goal, ?Ball, -List***)**

As intercept_all/4, but backtracing inside *Goal* does not reset *List*. Consider this program and the subsequent queries

```
enum_b(F, T) :- forall(between(F, T, I), send_signal(emit(I))).
```

```
?- intercept_all(I, enum_b(1, 6), emit(I), List).
List = [].

?- nb_intercept_all(I, enum_b(1, 6), emit(I), List).
List = [1, 2, 3, 4, 5, 6].
```

send_signal(+*Signal***)**

If this predicate is called from a sub-goal of intercept/3, execute the associated *Handler* of the intercept/3 environment.

Errors unintercepted_signal(Signal) if there is no matching intercept environment.

send_silent_signal(+*Signal***)**

As send_signal/1, but succeed silently if there is no matching intercept environment.

A.20 library(iostream): Utilities to deal with streams

See also library(archive), library(process), library(zlib), library(http/http_stream)

This library contains utilities that deal with streams, notably originating from non-built-in sources such as URLs, archives, windows, processes, etc.

The predicate open_any/5 acts as a *broker* between applications that can process data from a stream and libraries that can create streams from diverse sources. Without this predicate, processing data inevitably follows the pattern below. As *call_some_open_variation* can be anything, this blocks us from writing predicates such as load_xml(From, DOM) that can operate on arbitrary input sources.

```
setup_call_cleanup(
    call_some_open_variation(Spec, In),
    process(In),
    close(In)).
```

Libraries that can open streams can install the hook `iostream:open_hook/6` to make their functionality available through `open_any/5`.

open_any(*+Specification, +Mode, -Stream, -Close, +Options*)
Establish a stream from *Specification* that should be closed using *Close*, which can either be called or passed to `close_any/1`. *Options* processed:

encoding(*Enc*)
Set stream to encoding *Enc*.

Without loaded plugins, the `open_any/5` processes the following values for *Specification*. If no rule matches, `open_any/5` processes *Specification* as `file(Specification)`.

Stream
A plain stream handle. Possisible post-processing options such as encoding are applied. *Close* does *not* close the stream, but resets other side-effects such as the encoding.

stream(*Stream*)
Same as a plain *Stream*.

FileURL
If *Specification* is of the form =file://...=, the pointed to file is opened using `open/4`. Requires `library(uri)` to be installed.

file(*Path*)
Explicitly open the file *Path*. *Path* can be an *Path*(File) term as accepted by `absolute_file_name/3`.

string(*String*)
Open a Prolog string, atom, list of characters or codes as an *input* stream.

The typical usage scenario is given in the code below, where <process> processes the input.

```
setup_call_cleanup(
    open_any(Spec, read, In, Close, Options),
    <process>(In),
    Close).
```

Currently, the following libraries extend this predicate:

library(*http/http_open*)
Adds support for URLs using the `http` and `https` schemes.

close_any(+*Goal*)
Execute the *Close* closure returned by `open_any/5`. The closure can also be called directly. Using `close_any/1` can be considered better style and enhances tractability of the source code.

open_hook(+*Spec, +Mode, -Stream, -Close, +Options0, -Options*) *[semidet,multifile]*
Open *Spec* in *Mode*, producing *Stream*.

Arguments	
Close	is unified to a goal that must be called to undo the side-effects of the action, e.g., typically the term `close(Stream)`
Options0	are the options passed to `open_any/5`
Options	are passed to the post processing filters that may be installed by `open_any/5`.

A.21 library(listing): List programs and pretty print clauses

To be done
- More settings, support *Coding Guidelines for Prolog* and make the suggestions there the default.
- Provide persistent user customization

This module implements listing code from the internal representation in a human readable format.

- `listing/0` lists a module.
- `listing/1` lists a predicate or matching clause
- `listing/2` lists a predicate or matching clause with options
- `portray_clause/2` pretty-prints a clause-term

Layout can be customized using `library(settings)`. The effective settings can be listed using `list_settings/1` as illustrated below. Settings can be changed using `set_setting/2`.

```
?- list_settings(listing).
========================================================================
Name                      Value (*=modified) Comment
========================================================================
listing:body_indentation  4               Indentation used goals in the body
listing:tab_distance      0               Distance between tab-stops.
...
```

listing
Lists all predicates defined in the calling module. Imported predicates are not listed. To list the content of the module `mymodule`, use one of the calls below.

```
?- mymodule:listing.
?- listing(mymodule:_).
```

listing(*:What*) *[det]*
listing(*:What, +Options*) *[det]*

List matching clauses. *What* is either a plain specification or a list of specifications. Plain specifications are:

- Predicate indicator (Name/Arity or Name//Arity) Lists the indicated predicate. This also outputs relevant *declarations*, such as `multifile/1` or `dynamic/1`.
- A *Head* term. In this case, only clauses whose head unify with *Head* are listed. This is illustrated in the query below that only lists the first clause of `append/3`.

```
?- listing(append([], _, _)).
lists:append([], L, L).
```

The following options are defined:

variable_names(*+How*)
One of `source` (default) or `generated`. If `source`, for each clause that is associated to a source location the system tries to restore the original variable names. This may fail if macro expansion is not reversible or the term cannot be read due to different operator declarations. In that case variable names are generated.

source(*+Bool*)
If `true` (default `false`), extract the lines from the source files that produced the clauses, i.e., list the original source text rather than the *decompiled* clauses. Each set of contiguous clauses is preceded by a comment that indicates the file and line of origin. Clauses that cannot be related to source code are decompiled where the comment indicates the decompiled state. This is notably practical for collecting the state of *multifile* predicates. For example:

```
?- listing(file_search_path, [source(true)]).
```

portray_clause(*+Clause*) *[det]*
portray_clause(*+Out:stream, +Clause*) *[det]*
portray_clause(*+Out:stream, +Clause, +Options*) *[det]*

Portray '*Clause*' on the current output stream. Layout of the clause is to our best standards. Deals with control structures and calls via meta-call predicates as determined using the predicate property meta_predicate. If *Clause* contains attributed variables, these are treated as normal variables.

Variable names are by default generated using `numbervars/4` using the option `singletons(true)`. This names the variables *A*, *B*, ... and the singletons _. Variables can be named explicitly by binding them to a term `'$VAR'(Name)`, where *Name* is an atom denoting a valid variable name (see the option `numbervars(true)` from `write_term/2`) as well as by using the `variable_names(Bindings)` option from `write_term/2`.

Options processed in addition to `write_term/2` options:

variable_names(*+Bindings*)
See above and `write_term/2`.

indent(*+Columns*)
Left margin used for the clause. Default `0`.

module(*+Module*)
Module used to determine whether a goal resolves to a meta predicate. Default `user`.

A.22 library(lists): List Manipulation

Compatibility Virtually every Prolog system has `library(lists)`, but the set of provided predicates is diverse. There is a fair agreement on the semantics of most of these predicates, although error handling may vary.

This library provides commonly accepted basic predicates for list manipulation in the Prolog community. Some additional list manipulations are built-in. See e.g., `memberchk/2`, `length/2`.

The implementation of this library is copied from many places. These include: "The Craft of Prolog", the DEC-10 Prolog library (LISTRO.PL) and the YAP lists library. Some predicates are reimplemented based on their specification by Quintus and SICStus.

member(*?Elem, ?List*)
True if *Elem* is a member of *List*. The SWI-Prolog definition differs from the classical one. Our definition avoids unpacking each list element twice and provides determinism on the last element. E.g. this is deterministic:

```
member(X, [One]).
```

author Gertjan van Noord

append(*?List1, ?List2, ?List1AndList2*)
List1AndList2 is the concatenation of *List1* and *List2*

append(*+ListOfLists, ?List*)
Concatenate a list of lists. Is true if *ListOfLists* is a list of lists, and *List* is the concatenation of these lists.

Arguments	
ListOfLists	must be a list of *possibly* partial lists

prefix(*?Part, ?Whole*)
True iff *Part* is a leading substring of *Whole*. This is the same as `append(Part, _, Whole)`.

select(*?Elem, ?List1, ?List2*)
Is true when *List1*, with *Elem* removed, results in *List2*. This implementation is determinsitic if the last element of *List1* has been selected.

selectchk(*+Elem, +List, -Rest*) *[semidet]*
Semi-deterministic removal of first element in *List* that unifies with *Elem*.

select(*?X, ?XList, ?Y, ?YList*) *[nondet]*

Select from two lists at the same position. True if *XList* is unifiable with *YList* apart a single element at the same position that is unified with *X* in *XList* and with *Y* in *YList*. A typical use for this predicate is to *replace* an element, as shown in the example below. All possible substitutions are performed on backtracking.

```
?- select(b, [a,b,c,b], 2, X).
X = [a, 2, c, b] ;
X = [a, b, c, 2] ;
false.
```

See also `selectchk/4` provides a semidet version.

selectchk(*?X, ?XList, ?Y, ?YList*) *[semidet]*

Semi-deterministic version of `select/4`.

nextto(*?X, ?Y, ?List*)

True if *Y* directly follows *X* in *List*.

delete(*+List1, @Elem, -List2*) *[det]*

Delete matching elements from a list. True when *List2* is a list with all elements from *List1* except for those that unify with *Elem*. Matching *Elem* with elements of *List1* is uses `\+ Elem \= H`, which implies that *Elem* is not changed.

See also `select/3`, `subtract/3`.
deprecated There are too many ways in which one might want to delete elements from a list to justify the name. Think of matching (= vs. ==), delete first/all, be deterministic or not.

nth0(*?Index, ?List, ?Elem*)

True when *Elem* is the *Index*'th element of *List*. Counting starts at 0.

Errors `type_error(integer, Index)` if *Index* is not an integer or unbound.
See also `nth1/3`.

nth1(*?Index, ?List, ?Elem*)

Is true when *Elem* is the *Index*'th element of *List*. Counting starts at 1.

See also `nth0/3`.

nth0(*?N, ?List, ?Elem, ?Rest*) *[det]*

Select/insert element at index. True when *Elem* is the *N*'th (0-based) element of *List* and *Rest* is the remainder (as in by `select/3`) of *List*. For example:

```
?- nth0(I, [a,b,c], E, R).
I = 0, E = a, R = [b, c] ;
I = 1, E = b, R = [a, c] ;
I = 2, E = c, R = [a, b] ;
false.
```

```
?- nth0(1, L, a1, [a,b]).
L = [a, a1, b].
```

nth1(*?N, ?List, ?Elem, ?Rest*) *[det]*

As `nth0/4`, but counting starts at 1.

last(*?List, ?Last*)

Succeeds when *Last* is the last element of *List*. This predicate is `semidet` if *List* is a list and `multi` if *List* is a partial list.

> **Compatibility** There is no de-facto standard for the argument order of `last/2`. Be careful when porting code or use `append(_, [Last], List)` as a portable alternative.

proper_length(*@List, -Length*) *[semidet]*

True when *Length* is the number of elements in the proper list *List*. This is equivalent to

```
proper_length(List, Length) :-
      is_list(List),
      length(List, Length).
```

same_length(*?List1, ?List2*)

Is true when *List1* and *List2* are lists with the same number of elements. The predicate is deterministic if at least one of the arguments is a proper list. It is non-deterministic if both arguments are partial lists.

> **See also** `length/2`

reverse(*?List1, ?List2*)

Is true when the elements of *List2* are in reverse order compared to *List1*.

permutation(*?Xs, ?Ys*) *[nondet]*

True when *Xs* is a permutation of *Ys*. This can solve for *Ys* given *Xs* or *Xs* given *Ys*, or even enumerate *Xs* and *Ys* together. The predicate `permutation/2` is primarily intended to generate permutations. Note that a list of length N has N! permutations, and unbounded permutation generation becomes prohibitively expensive, even for rather short lists (10! = 3,628,800).

If both *Xs* and *Ys* are provided and both lists have equal length the order is `|`*Xs*`|`^2. Simply testing whether *Xs* is a permutation of *Ys* can be achieved in order log(`|`*Xs*`|`) using `msort/2` as illustrated below with the `semidet` predicate `is_permutation/2`:

```
is_permutation(Xs, Ys) :-
  msort(Xs, Sorted),
  msort(Ys, Sorted).
```

The example below illustrates that *Xs* and *Ys* being proper lists is not a sufficient condition to use the above replacement.

```
?- permutation([1,2], [X,Y]).
X = 1, Y = 2 ;
X = 2, Y = 1 ;
false.
```

Errors `type_error(list, Arg)` if either argument is not a proper or partial list.

flatten(*+NestedList, -FlatList*) *[det]*

Is true if *FlatList* is a non-nested version of *NestedList*. Note that empty lists are removed. In standard Prolog, this implies that the atom '`[]`' is removed too. In SWI7, `[]` is distinct from '`[]`'.

Ending up needing `flatten/2` often indicates, like `append/3` for appending two lists, a bad design. Efficient code that generates lists from generated small lists must use difference lists, often possible through grammar rules for optimal readability.

See also `append/2`

clumped(*+Items, -Pairs*)

Pairs is a list of `Item-Count` pairs that represents the *run length encoding* of *Items*. For example:

```
?- clumped([a,a,b,a,a,a,a,c,c,c], R).
R = [a-2, b-1, a-4, c-3].
```

Compatibility SICStus

max_member(*-Max, +List*) *[semidet]*

True when *Max* is the largest member in the standard order of terms. Fails if *List* is empty.

See also
- `compare/3`
- `max_list/2` for the maximum of a list of numbers.

min_member(*-Min, +List*) *[semidet]*

True when *Min* is the smallest member in the standard order of terms. Fails if *List* is empty.

See also
- `compare/3`
- `min_list/2` for the minimum of a list of numbers.

max_member(*:Pred, -Max, +List*) *[semidet]*

True when *Max* is the largest member according to *Pred*, which must be a 2-argument callable that behaves like (`@=<`)/2. Fails if *List* is empty. The following call is equivalent to `max_member/2`:

```
?- max_member(@=<, X, [6,1,8,4]).
X = 8.
```

See also `max_list/2` for the maximum of a list of numbers.

min_member(*:Pred, -Min, +List*) *[semidet]*

True when *Min* is the smallest member according to *Pred*, which must be a 2-argument callable that behaves like (@=<)/2. Fails if *List* is empty. The following call is equivalent to `max_member/2`:

```
?- min_member(@=<, X, [6,1,8,4]).
X = 1.
```

See also `min_list/2` for the minimum of a list of numbers.

sum_list(*+List, -Sum*) *[det]*

Sum is the result of adding all numbers in *List*.

max_list(*+List:list(number), -Max:number*) *[semidet]*

True if *Max* is the largest number in *List*. Fails if *List* is empty.

See also `max_member/2`.

min_list(*+List:list(number), -Min:number*) *[semidet]*

True if *Min* is the smallest number in *List*. Fails if *List* is empty.

See also `min_member/2`.

numlist(*+Low, +High, -List*) *[semidet]*

List is a list [*Low*, *Low*+1, ... *High*]. Fails if *High* < *Low*.

Errors
- `type_error(integer, Low)`
- `type_error(integer, High)`

is_set(*@Set*) *[semidet]*

True if *Set* is a proper list without duplicates. Equivalence is based on `==/2`. The implementation uses `sort/2`, which implies that the complexity is N*`log(N)` and the predicate may cause a resource-error. There are no other error conditions.

list_to_set(*+List, ?Set*) *[det]*

True when *Set* has the same elements as *List* in the same order. The left-most copy of duplicate elements is retained. *List* may contain variables. Elements *E1* and *E2* are considered duplicates iff *E1* == *E2* holds. The complexity of the implementation is N*`log(N)`.

Errors *List* is type-checked.

See also `sort/2` can be used to create an ordered set. Many set operations on ordered sets are order N rather than order N**2. The `list_to_set/2` predicate is more expensive than `sort/2` because it involves, two sorts and a linear scan.

Compatibility Up to version 6.3.11, `list_to_set/2` had complexity N**2 and equality was tested using `=/2`.

intersection(*+Set1, +Set2, -Set3*) *[det]*

True if *Set3* unifies with the intersection of *Set1* and *Set2*. The complexity of this predicate is |*Set1*|*|*Set2*|. A *set* is defined to be an unordered list without duplicates. Elements are considered duplicates if they can be unified.

See also `ord_intersection/3`.

union(*+Set1, +Set2, -Set3*) *[det]*

True if *Set3* unifies with the union of the lists *Set1* and *Set2*. The complexity of this predicate is |*Set1*|*|*Set2*|. A *set* is defined to be an unordered list without duplicates. Elements are considered duplicates if they can be unified.

See also `ord_union/3`

subset(*+SubSet, +Set*) *[semidet]*

True if all elements of *SubSet* belong to *Set* as well. Membership test is based on `memberchk/2`. The complexity is |*SubSet*|*|*Set*|. A *set* is defined to be an unordered list without duplicates. Elements are considered duplicates if they can be unified.

See also `ord_subset/2`.

subtract(*+Set, +Delete, -Result*) *[det]*

Delete all elements in *Delete* from *Set*. Deletion is based on unification using `memberchk/2`. The complexity is |*Delete*|*|*Set*|. A *set* is defined to be an unordered list without duplicates. Elements are considered duplicates if they can be unified.

See also `ord_subtract/3`.

A.23 library(main): Provide entry point for scripts

See also

- `library(prolog_stack)` to force backtraces in case of an uncaught exception.
- XPCE users should have a look at `library(pce_main)`, which starts the GUI and processes events until all windows have gone.

This library is intended for supporting PrologScript on Unix using the `#!` magic sequence for scripts using commandline options. The entry point `main/0` calls the user-supplied predicate `main/1` passing a list of commandline options. Below is a simle `echo` implementation in Prolog.

```
#!/usr/bin/env swipl

:- initialization(main, main).

main(Argv) :-
    echo(Argv).

echo([]) :- nl.
echo([Last]) :- !,
    write(Last), nl.
```

```
echo([H|T]) :-
    write(H), write(' '),
    echo(T).
```

main

Call `main/1` using the passed command-line arguments. Before calling `main/1` this predicate installs a signal handler for `SIGINT` (Control-C) that terminates the process with status 1.

When `main/0` is called interactively it simply calls `main/1` with the arguments. This allows for debugging scripts as follows:

```
$ swipl -l script.pl -- arg ...
?- gspy(suspect/1).                % setup debugging
?- main.                           % run program
```

argv_options(*:Argv, -Positional, -Options*) *[det]*

Parse command line arguments. This predicate acts in one of two modes.

- If the calling module defines `opt_type/3`, full featured parsing with long and short options, type conversion and help is provided.
- If `opt_type/3` is not defined, only unguided transformation using long options is supported. See `argv_untyped_options/3` for details.

When **guided**, three predicates are called in the calling module. `opt_type/3` **must** be defined, the others need not. Note that these three predicates *may* be defined as *multifile* to allow multiple modules contributing to the provided commandline options. Defining them as *discontiguous* allows for creating blocks that describe a group of related options.

opt_type(*Opt, Name, Type*)

Defines *Opt* to add an option *Name*(Value), where Value statisfies *Type*. *Opt* does not include the leading `-`. A single character implies a short option, multiple a long option. Long options use _ as *word separator*, user options may use either _ or `-`. *Type* is one of:

A **|** *B*

Disjunctive type.

boolean(*Default*)

boolean

Boolean options are special. They do not take a value except for when using the long `--opt=value` notation. This explicit value specification converts `true`, `True`, `TRUE`, `on`, `On`, `ON`, `1` and the obvious false equivalents to Prolog `true` or `false`. If the option is specified, Default is used. If `--no-opt` or `--noopt` is used, the inverse of Default is used.

integer
Argument is converted to an integer

float
Argument is converted to a float. User may specify an integer

nonneg
As `integer`. Requires value >= 0.

natural
As `integer`. Requires value >= 1.

number
Any number (integer, float, rational).

between(*Low, High*)
If both one of *Low* and *High* is a float, convert as `float`, else convert as `integer`. Then check the range.

atom
No conversion

oneof(*List*)
As `atom`, but requires the value to be a member of *List* (*enum* type).

string
Convert to a SWI-Prolog string

file
Convert to a file name in Prolog canonical notation using `prolog_to_os_filename/2`.

file(*Access*)
As `file`, and check access using `access_file/2`. A value `-` is not checked for access, assuming the application handles this as standard input or output.

term
Parse option value to a Prolog term.

term(*+Options*)
As `term`, but passes *Options* to `term_string/3`. If the option `variable_names(Bindings)` is given the option value is set to the *pair* `Term-Bindings`.

opt_help(*Name, HelpString*)
Help string used by `argv_usage/1`.

opt_meta(*Name, Meta*)
If a typed argument is required this defines the placeholder in the help message. The default is the uppercase version of the type *functor name*. This produces the `FILE` in e.g. `-f FILE`.

By default, `-h`, `-?` and `--help` are bound to help. If `opt_type(Opt, help, boolean)` is true for some *Opt*, the default help binding and help message are disabled and the normal user rules apply. In particular, the user should also provide a rule for `opt_help(help, String)`.

argv_options(*:Argv, -Positional, -Options, +ParseOptions*) *[det]*
As `argv_options/3` in **guided** mode, Currently this version allows parsing argument op-

tions throwing an exception rather than calling `halt/1` by passing an empty list to *ParseOptions*. *ParseOptions*:

on_error(+*Goal*)
If *Goal* is `halt(Code)`, exit with Code. Other goals are currently not supported.

options_after_arguments(+*Boolean*)
If `false` (default `true`), stop parsing after the first positional argument, returning options that follow this argument as positional arguments. E.g, `-x file -y` results in positional arguments `[file, '-y']`

argv_usage(*:Level*) *[det]*
Use `print_message/2` to print a usage message at *Level*. To print the message as plain text indefault color, use `debug`. Other meaningful options are `informational` or `warning`. The help page consists of four sections, two of which are optional:

1. The **header** is created from `opt_help(help(header), String)`. It is optional.
2. The **usage** is added by default. The part behind `Usage: <command>` is by default `[options]` and can be overruled using `opt_help(help(usage), String)`.
3. The actual option descriptions. The options are presented in the order they are defined in `opt_type/3`. Subsequent options for the same *destination* (option name) are joined with the first.
4. The *footer_* is created from `opt_help(help(footer), String)`. It is optional.

The help provided by `help(header)`, `help(usage)` and `help(footer)` are either a simple string or a list of elements as defined by `print_message_lines/3`. In the latter case, the construct `\Callable` can be used to call a DCG rule in the module from which the user calls `argv_options/3`. For example, we can add a bold title using

```
opt_help(help(header), [ansi(bold, '~w', ['My title'])]).
```

cli_parse_debug_options(+*OptionsIn*, -*Options*) *[det]*
Parse certain commandline options for debugging and development purposes. *Options* processed are below. Note that the option argument is an atom such that these options may be activated as e.g., `--debug='http(_)'`.

debug(*Topic*)
Call `debug(Topic)`. See `debug/1` and `debug/3`.

spy(*Predicate*)
Place a spy-point on *Predicate*.

gspy(*Predicate*)
As spy using the graphical debugger. See `tspy/1`.

interactive(*true*)
Start the Prolog toplevel after `main/1` completes.

cli_enable_development_system
Re-enable the development environment. Currently re-enables xpce if this was loaded, but not initialised and causes the interactive toplevel to be re-enabled.

This predicate may be called from `main/1` to enter the Prolog toplevel rather than terminating the application after `main/1` completes.

A.24 library(nb_set): 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 data structures is not supported by a large number of Prolog implementations, but 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 outlined 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 empty 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, binding *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., an *ordered list*).

A.25 library(www_browser): 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 version of Windows 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 browser 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.26 library(occurs): Finding and counting sub-terms

See also `library(terms)` provides similar predicates and is probably more wide-spread than this library.

This is a SWI-Prolog implementation of the corresponding Quintus library, based on the generalised `arg/3` predicate of SWI-Prolog.

contains_term(+*Sub*, +*Term*) *[semidet]*
Succeeds if *Sub* is contained in *Term* (=, deterministically)

contains_var(+*Sub*, +*Term*) *[det]*
Succeeds if *Sub* is contained in *Term* (==, deterministically)

free_of_term(+*Sub*, +*Term*)
Succeeds of *Sub* does not unify to any subterm of *Term*

free_of_var(+*Sub*, +*Term*)
Succeeds of *Sub* is not equal (==) to any subterm of *Term*

occurrences_of_term(+*SubTerm*, +*Term*, ?*Count*)
Count the number of SubTerms in *Term*

occurrences_of_var(+*SubTerm*, +*Term*, ?*Count*)
Count the number of SubTerms in *Term*

sub_term(*-Sub, +Term*)
Generates (on backtracking) all subterms of *Term*.

sub_var(*-Sub, +Term*)
Generates (on backtracking) all subterms (==) of *Term*.

sub_term_shared_variables(*+Sub, +Term, -Vars*) *[det]*
If *Sub* is a sub term of *Term*, *Vars* is bound to the list of variables in *Sub* that also appear outside *Sub* in *Term*. Note that if *Sub* appears twice in *Term*, its variables are all considered shared.

An example use-case is refactoring a large clause body by introducing intermediate predicates. This predicate can be used to find the arguments that must be passed to the new predicate.

A.27 library(option): Option list processing

See also
- `library(record)`
- Option processing capabilities may be declared using the directive `predicate_options/3`.

To be done We should consider putting many options in an assoc or record with appropriate preprocessing to achieve better performance.

The `library(option)` provides some utilities for processing option lists. Option lists are commonly used as an alternative for many arguments. Examples of built-in predicates are `open/4` and `write_term/3`. Naming the arguments results in more readable code, and the list nature makes it easy to extend the list of options accepted by a predicate. Option lists come in two styles, both of which are handled by this library.

Name(Value) This is the preferred style.

Name = Value This is often used, but deprecated.

Processing options inside time-critical code (loops) can cause serious overhead. One possibility is to define a record using `library(record)` and initialise this using make_<record>`/2`. In addition to providing good performance, this also provides type-checking and central declaration of defaults.

```
:- record atts(width:integer=100, shape:oneof([box,circle])=box).

process(Data, Options) :-
        make_atts(Options, Attributes),
        action(Data, Attributes).

action(Data, Attributes) :-
        atts_shape(Attributes, Shape),
        ...
```

Options typically have exactly one argument. The library does support options with 0 or more than one argument with the following restrictions:

- The predicate `option/3` and `select_option/4`, involving default are meaningless. They perform an `arg(1, Option, Default)`, causing failure without arguments and filling only the first option-argument otherwise.

- `meta_options/3` can only qualify options with exactly one argument.

option(*?Option, +OptionList, +Default*) *[semidet]*

Get an *Option* from *OptionList*. *OptionList* can use the Name=Value as well as the Name(Value) convention.

Arguments

Option	Term of the form Name(?Value).

option(*?Option, +OptionList*) *[semidet]*

Get an *Option* from *OptionList*. *OptionList* can use the Name=Value as well as the Name(Value) convention. Fails silently if the option does not appear in *OptionList*.

Arguments

Option	Term of the form Name(?Value).

select_option(*?Option, +Options, -RestOptions*) *[semidet]*

Get and remove *Option* from an option list. As `option/2`, removing the matching option from *Options* and unifying the remaining options with *RestOptions*.

select_option(*?Option, +Options, -RestOptions, +Default*) *[det]*

Get and remove *Option* with default value. As `select_option/3`, but if *Option* is not in *Options*, its value is unified with *Default* and *RestOptions* with *Options*.

merge_options(*+New, +Old, -Merged*) *[det]*

Merge two option lists. *Merged* is a sorted list of options using the canonical format Name(Value) holding all options from *New* and *Old*, after removing conflicting options from *Old*.

Multi-values options (e.g., `proxy(Host, Port)`) are allowed, where both option-name and arity define the identity of the option.

meta_options(*+IsMeta, :Options0, -Options*) *[det]*

Perform meta-expansion on options that are module-sensitive. Whether an option name is module-sensitive is determined by calling `call(IsMeta, Name)`. Here is an example:

```
        meta_options(is_meta, OptionsIn, Options),
        ...

is_meta(callback).
```

Meta-options must have exactly one argument. This argument will be qualified.

To be done Should be integrated with declarations from `predicate_options/3`.

dict_options(*?Dict, ?Options*) *[det]*

Convert between an option list and a dictionary. One of the arguments must be instantiated. If the option list is created, it is created in canonical form, i.e., using Option(Value) with the *Options* sorted in the standard order of terms. Note that the conversion is not always possible due to different constraints and conversion may thus lead to (type) errors.

- *Dict* keys can be integers. This is not allowed in canonical option lists.
- *Options* can hold multiple options with the same key. This is not allowed in dicts. This predicate removes all but the first option on the same key.
- *Options* can have more than one value (`name(V1,V2)`). This is not allowed in dicts.

Also note that most system predicates and predicates using this library for processing the option argument can both work with classical Prolog options and dicts objects.

A.28 library(optparse): command line parsing

author Marcus Uneson
version 0.20 (2011-04-27)
To be done : validation? e.g, numbers; file path existence; one-out-of-a-set-of-atoms

This module helps in building a command-line interface to an application. In particular, it provides functions that take an option specification and a list of atoms, probably given to the program on the command line, and return a parsed representation (a list of the customary Key(Val) by default; or optionally, a list of Func(Key, Val) terms in the style of `current_prolog_flag/2`). It can also synthesize a simple help text from the options specification.

The terminology in the following is partly borrowed from python, see http://docs.python.org/library/optparse.html#terminology . Very briefly, *arguments* is what you provide on the command line and for many prologs show up as a list of atoms `Args` in `current_prolog_flag(argv, Args)`. For a typical prolog incantation, they can be divided into

- *runtime arguments*, which controls the prolog runtime; conventionally, they are ended by '–';
- *options*, which are key-value pairs (with a boolean value possibly implicit) intended to control your program in one way or another; and
- *positional arguments*, which is what remains after all runtime arguments and options have been removed (with implicit arguments – true/false for booleans – filled in).

Positional arguments are in particular used for mandatory arguments without which your program won't work and for which there are no sensible defaults (e.g,, input file names). Options, by contrast, offer flexibility by letting you change a default setting. Options are optional not only by etymology: this library has no notion of mandatory or required options (see the python docs for other rationales than laziness).

The command-line arguments enter your program as a list of atoms, but the programs perhaps expects booleans, integers, floats or even prolog terms. You tell the parser so by providing an *options specification*. This is just a list of individual option specifications. One of those, in turn, is a list of ground prolog terms in the customary Name(Value) format. The following terms are recognized (any others raise error).

opt(*Key*)
: *Key* is what the option later will be accessed by, just like for `current_prolog_flag(Key, Value)`. This term is mandatory (an error is thrown if missing).

shortflags(*ListOfFlags*)
ListOfFlags denotes any single-dashed, single letter args specifying the current option (`-s` , `-K`, etc). Uppercase letters must be quoted. Usually *ListOfFlags* will be a singleton list, but sometimes aliased flags may be convenient.

longflags(*ListOfFlags*)
ListOfFlags denotes any double-dashed arguments specifying the current option (`--verbose`, `--no-debug`, etc). They are basically a more readable alternative to short flags, except

1. long flags can be specified as `--flag value` or `--flag=value` (but not as `--flagvalue`); short flags as `-f val` or `-fval` (but not `-f=val`)
2. boolean long flags can be specified as `--bool-flag` or `--bool-flag=true` or `--bool-flag true`; and they can be negated as `--no-bool-flag` or `--bool-flag=false` or `--bool-flag false`.

Except that shortflags must be single characters, the distinction between long and short is in calling convention, not in namespaces. Thus, if you have `shortflags([v])`, you can use it as `-v2` or `-v 2` or `--v=2` or `--v 2` (but not `-v=2` or `--v2`).

Shortflags and longflags both default to `[]`. It can be useful to have flagless options – see example below.

meta(*Meta*)
Meta is optional and only relevant for the synthesized usage message and is the name (an atom) of the metasyntactic variable (possibly) appearing in it together with type and default value (e.g, `x:integer=3`, `interest:float=0.11`). It may be useful to have named variables (`x`, `interest`) in case you wish to mention them again in the help text. If not given the `Meta:` part is suppressed – see example below.

type(*Type*)
Type is one of `boolean, atom, integer, float, term`. The corresponding argument will be parsed appropriately. This term is optional; if not given, defaults to `term`.

default(*Default*)
Default value. This term is optional; if not given, or if given the special value '_', an uninstantiated variable is created (and any type declaration is ignored).

help(*Help*)
Help is (usually) an atom of text describing the option in the help text. This term is optional (but obviously strongly recommended for all options which have flags).

Long lines are subject to basic word wrapping – split on white space, reindent, rejoin. However, you can get more control by supplying the line breaking yourself: rather than a single line of text, you can provide a list of lines (as atoms). If you do, they will be joined with the appropriate indent but otherwise left untouched (see the option `mode` in the example below).

Absence of mandatory option specs or the presence of more than one for a particular option throws an error, as do unknown or incompatible types.

As a concrete example from a fictive application, suppose we want the following options to be read from the command line (long `flag(s)`, short `flag(s)`, meta:type=default, help)

```
--mode                    -m     atom=SCAN        data gathering mode,
                                                  one of
                                                   SCAN: do this
                                                   READ: do that
                                                   MAKE: make numbers
                                                   WAIT: do nothing
--rebuild-cache           -r     boolean=true     rebuild cache in
                                                  each iteration
--heisenberg-threshold    -t,-h  float=0.1        heisenberg threshold
--depths, --iters         -i,-d  K:integer=3      stop after K
                                                  iterations
--distances                      term=[1,2,3,5]   initial prolog term
--output-file             -o     FILE:atom=_      write output to FILE
--label                   -l     atom=REPORT      report label
--verbosity               -v     V:integer=2      verbosity level,
                                                  1 <= V <= 3
```

We may also have some configuration parameters which we currently think not needs to be controlled from the command line, say `path('/some/file/path')`.

This interface is described by the following options specification (order between the specifications of a particular option is irrelevant).

```
ExampleOptsSpec =
    [ [opt(mode    ), type(atom), default('SCAN'),
        shortflags([m]),   longflags(['mode'] ),
        help([ 'data gathering mode, one of'
             , '  SCAN: do this'
             , '  READ: do that'
             , '  MAKE: fabricate some numbers'
             , '  WAIT: don''t do anything'])]

    , [opt(cache), type(boolean), default(true),
        shortflags([r]),   longflags(['rebuild-cache']),
        help('rebuild cache in each iteration')]

    , [opt(threshold), type(float), default(0.1),
        shortflags([t,h]),  longflags(['heisenberg-threshold']),
        help('heisenberg threshold')]

    , [opt(depth), meta('K'), type(integer), default(3),
        shortflags([i,d]),longflags([depths,iters]),
        help('stop after K iterations')]

    , [opt(distances), default([1,2,3,5]),
        longflags([distances]),
        help('initial prolog term')]
```

```
    , [opt(outfile), meta('FILE'), type(atom),
        shortflags([o]),  longflags(['output-file']),
        help('write output to FILE')]

    , [opt(label), type(atom), default('REPORT'),
        shortflags([l]), longflags([label]),
        help('report label')]

    , [opt(verbose),  meta('V'), type(integer), default(2),
        shortflags([v]),  longflags([verbosity]),
        help('verbosity level, 1 <= V <= 3')]

    , [opt(path), default('/some/file/path/')]
    ].
```

The help text above was accessed by `opt_help(ExamplesOptsSpec, HelpText)`. The options appear in the same order as in the OptsSpec.

Given `ExampleOptsSpec`, a command line (somewhat syntactically inconsistent, in order to demonstrate different calling conventions) may look as follows

```
ExampleArgs = [ '-d5'
              , '--heisenberg-threshold', '0.14'
              , '--distances=[1,1,2,3,5,8]'
              , '--iters', '7'
              , '-ooutput.txt'
              , '--rebuild-cache', 'true'
              , 'input.txt'
              , '--verbosity=2'
              ].
```

`opt_parse(ExampleOptsSpec, ExampleArgs, Opts, PositionalArgs)` would then succeed with

```
Opts =    [ mode('SCAN')
          , label('REPORT')
          , path('/some/file/path')
          , threshold(0.14)
          , distances([1,1,2,3,5,8])
          , depth(7)
          , outfile('output.txt')
          , cache(true)
          , verbose(2)
          ],
PositionalArgs = ['input.txt'].
```

Note that `path('/some/file/path')` showing up in Opts has a default value (of the implicit type 'term'), but no corresponding flags in OptsSpec. Thus it can't be set from the command line. The rest of your program doesn't need to know that, of course. This provides an alternative to the common practice of asserting such hard-coded parameters under a single predicate (for instance `setting(path, '/some/file/path')`), with the advantage that you may seamlessly upgrade them to command-line options, should you one day find this a good idea. Just add an appropriate flag or two and a line of help text. Similarly, suppressing an option in a cluttered interface amounts to commenting out the flags.

`opt_parse/5` allows more control through an additional argument list as shown in the example below.

```
?- opt_parse(ExampleOptsSpec, ExampleArgs,  Opts, PositionalArgs,
             [ output_functor(appl_config)
             ]).

Opts =     [ appl_config(verbose, 2),
           , appl_config(label, 'REPORT')
           ...
           ]
```

This representation may be preferable with the empty-flag configuration parameter style above (perhaps with asserting `appl_config/2`).

A.28.1 Notes and tips

- In the example we were mostly explicit about the types. Since the default is `term`, which subsumes `integer, float, atom`, it may be possible to get away cheaper (e.g., by only giving booleans). However, it is recommended practice to always specify types: parsing becomes more reliable and error messages will be easier to interpret.

- Note that `-sbar` is taken to mean `-s bar`, not `-s -b -a -r`, that is, there is no clustering of flags.

- `-s=foo` is disallowed. The rationale is that although some command-line parsers will silently interpret this as `-s =foo`, this is very seldom what you want. To have an option argument start with '=' (very un-recommended), say so explicitly.

- The example specifies the option `depth` twice: once as `-d5` and once as `--iters 7`. The default when encountering duplicated flags is to `keeplast` (this behaviour can be controlled, by ParseOption duplicated_flags).

- The order of the options returned by the parsing functions is the same as given on the command line, with non-overridden defaults prepended and duplicates removed as in previous item. You should not rely on this, however.

- Unknown flags (not appearing in OptsSpec) will throw errors. This is usually a Good Thing. Sometimes, however, you may wish to pass along flags to an external program (say, one called by `shell/2`), and it means duplicated effort and a maintenance headache to have to specify all possible flags for the external program explicitly (if it even can be done). On the other hand, simply taking all unknown flags as valid makes error checking much less efficient and identification of positional arguments uncertain. A better solution is to collect all arguments intended for passing along to an indirectly called program as a single argument, probably as an atom (if you don't need to inspect them first) or as a prolog term (if you do).

opt_arguments(+*OptsSpec, -Opts, -PositionalArgs***)** *[det]*

Extract commandline options according to a specification. Convenience predicate, assuming that command-line arguments can be accessed by `current_prolog_flag/2` (as in swi-prolog). For other access mechanisms and/or more control, get the args and pass them as a list of atoms to `opt_parse/4` or `opt_parse/5` instead.

Opts is a list of parsed options in the form Key(Value). Dashed args not in *OptsSpec* are not permitted and will raise error (see tip on how to pass unknown flags in the module description). *PositionalArgs* are the remaining non-dashed args after each flag has taken its argument (filling in `true` or `false` for booleans). There are no restrictions on non-dashed arguments and they may go anywhere (although it is good practice to put them last). Any leading arguments for the runtime (up to and including '–') are discarded.

opt_parse(+*OptsSpec, +ApplArgs, -Opts, -PositionalArgs***)** *[det]*

Equivalent to `opt_parse(OptsSpec, ApplArgs, Opts, PositionalArgs, [])`.

opt_parse(+*OptsSpec, +ApplArgs, -Opts, -PositionalArgs, +ParseOptions***)** *[det]*

Parse the arguments Args (as list of atoms) according to *OptsSpec*. Any runtime arguments (typically terminated by '–') are assumed to be removed already.

Opts is a list of parsed options in the form Key(Value), or (with the option `functor(Func)` given) in the form Func(Key, Value). Dashed args not in *OptsSpec* are not permitted and will raise error (see tip on how to pass unknown flags in the module description). *PositionalArgs* are the remaining non-dashed args after each flag has taken its argument (filling in `true` or `false` for booleans). There are no restrictions on non-dashed arguments and they may go anywhere (although it is good practice to put them last). *ParseOptions* are

output_functor(*Func***)**

Set the functor *Func* of the returned options *Func*(Key,Value). Default is the special value 'OPTION' (upper-case), which makes the returned options have form Key(Value).

duplicated_flags(*Keep***)**

Controls how to handle options given more than once on the commad line. *Keep* is one of `keepfirst, keeplast, keepall` with the obvious meaning. Default is `keeplast`.

allow_empty_flag_spec(*Bool***)**

If true (default), a flag specification is not required (it is allowed that both shortflags and longflags be either `[]` or absent). Flagless options cannot be manipulated from the command line and will not show up in the generated help. This is useful when you have (also) general configuration parameters in your *OptsSpec*, especially if you think they one day might need to be controlled externally. See example in the module overview. `allow_empty_flag_spec(false)` gives the more customary behaviour of raising error on empty flags.

opt_help(*+OptsSpec, -Help:atom*) *[det]*
True when *Help* is a help string synthesized from *OptsSpec*.

parse_type(*+Type, +Codes:list(code), -Result*) *[semidet,multifile]*
Hook to parse option text *Codes* to an object of type *Type*.

A.29 library(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 `library(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 `library(oset)`, an older ordered set library distributed with SWI-Prolog. New applications are advised to use `library(ordsets)`.

Some of these predicates match directly to corresponding list operations. It is advised to use the versions from this library to make clear you are operating on ordered sets. An exception is `member/2`. See `ord_memberchk/2`.

The ordsets library is based on the standard order of terms. This implies it can handle all Prolog terms, including variables. Note however, that the ordering is not stable if a term inside the set is further instantiated. Also note that variable ordering changes if variables in the set are unified with each other or a variable in the set is unified with a variable that is 'older' than the newest variable in the set. In practice, this implies that it is allowed to use `member(X, OrdSet)` on an ordered set that holds variables only if X is a fresh variable. In other cases one should cease using it as an ordset because the order it relies on may have been changed.

is_ordset(*@Term*) *[semidet]*
True if *Term* is an ordered set. All predicates in this library expect ordered sets as input arguments. Failing to fullfil this assumption results in undefined behaviour. Typically, ordered sets are created by predicates from this library, `sort/2` or `setof/3`.

ord_empty(*?List*) *[semidet]*
True when *List* is the empty ordered set. Simply unifies list with the empty list. Not part of Quintus.

ord_seteq(*+Set1, +Set2*) *[semidet]*
True if *Set1* and *Set2* have the same elements. As both are canonical sorted lists, this is the same as `==/2`.

Compatibility sicstus

list_to_ord_set(*+List, -OrdSet*) *[det]*
Transform a list into an ordered set. This is the same as sorting the list.

ord_intersect(*+Set1, +Set2*) *[semidet]*
True if both ordered sets have a non-empty intersection.

ord_disjoint(+*Set1*, +*Set2*) *[semidet]*

True if *Set1* and *Set2* have no common elements. This is the negation of `ord_intersect/2`.

ord_intersect(+*Set1*, +*Set2*, -*Intersection*)

Intersection holds the common elements of *Set1* and *Set2*.

> **deprecated** Use `ord_intersection/3`

ord_intersection(+*PowerSet*, -*Intersection*)

Intersection of a powerset. True when *Intersection* is an ordered set holding all elements common to all sets in *PowerSet*.

> **Compatibility** sicstus

ord_intersection(+*Set1*, +*Set2*, -*Intersection*) *[det]*

Intersection holds the common elements of *Set1* and *Set2*. Uses `ord_disjoint/2` if *Intersection* is bound to `[]` on entry.

ord_intersection(+*Set1*, +*Set2*, ?*Intersection*, ?*Difference*) *[det]*

Intersection and difference between two ordered sets. *Intersection* is the intersection between *Set1* and *Set2*, while *Difference* is defined by `ord_subtract(Set2, Set1, Difference)`.

> **See also** `ord_intersection/3` and `ord_subtract/3`.

ord_add_element(+*Set1*, +*Element*, ?*Set2*) *[det]*

Insert an element into the set. This is the same as `ord_union(Set1, [Element], Set2)`.

ord_del_element(+*Set*, +*Element*, -*NewSet*) *[det]*

Delete an element from an ordered set. This is the same as `ord_subtract(Set, [Element], NewSet)`.

ord_selectchk(+*Item*, ?*Set1*, ?*Set2*) *[semidet]*

Selectchk/3, specialised for ordered sets. Is true when `select(Item, Set1, Set2)` and *Set1*, *Set2* are both sorted lists without duplicates. This implementation is only expected to work for *Item* ground and either *Set1* or *Set2* ground. The "chk" suffix is meant to remind you of `memberchk/2`, which also expects its first argument to be ground. `ord_selectchk(X, S, T) => ord_memberchk(X, S) & \+ ord_memberchk(X, T)`.

> **author** Richard O'Keefe

ord_memberchk(+*Element*, +*OrdSet*) *[semidet]*

True if *Element* is a member of *OrdSet*, compared using ==. Note that *enumerating* elements of an ordered set can be done using `member/2`.

Some Prolog implementations also provide `ord_member/2`, with the same semantics as `ord_memberchk/2`. We believe that having a semidet `ord_member/2` is unacceptably inconsistent with the *_chk convention. Portable code should use `ord_memberchk/2` or `member/2`.

author Richard O'Keefe

ord_subset(*+Sub, +Super*) *[semidet]*
Is true if all elements of *Sub* are in *Super*

ord_subtract(*+InOSet, +NotInOSet, -Diff*) *[det]*
Diff is the set holding all elements of *InOSet* that are not in *NotInOSet*.

ord_union(*+SetOfSets, -Union*) *[det]*
True if *Union* is the union of all elements in the superset *SetOfSets*. Each member of *SetOfSets* must be an ordered set, the sets need not be ordered in any way.

author Copied from YAP, probably originally by Richard O'Keefe.

ord_union(*+Set1, +Set2, -Union*) *[det]*
Union is the union of *Set1* and *Set2*

ord_union(*+Set1, +Set2, -Union, -New*) *[det]*
True iff `ord_union(Set1, Set2, Union)` and `ord_subtract(Set2, Set1, New)`.

ord_symdiff(*+Set1, +Set2, ?Difference*) *[det]*
Is true when *Difference* is the symmetric difference of *Set1* and *Set2*. I.e., *Difference* contains all elements that are not in the intersection of *Set1* and *Set2*. The semantics is the same as the sequence below (but the actual implementation requires only a single scan).

```
      ord_union(Set1, Set2, Union),
      ord_intersection(Set1, Set2, Intersection),
      ord_subtract(Union, Intersection, Difference).
```

For example:

```
?- ord_symdiff([1,2], [2,3], X).
X = [1,3].
```

A.30 library(pairs): Operations on key-value lists

author Jan Wielemaker
See also `keysort/2`, `library(assoc)`

This module implements common operations on Key-Value lists, also known as *Pairs*. Pairs have great practical value, especially due to `keysort/2` and the library `assoc.pl`.

This library is based on discussion in the SWI-Prolog mailinglist, including specifications from Quintus and a library proposal by Richard O'Keefe.

pairs_keys_values(*?Pairs, ?Keys, ?Values*) *[det]*
True if *Keys* holds the keys of *Pairs* and *Values* the values.

Deterministic if any argument is instantiated to a finite list and the others are either free or finite lists. All three lists are in the same order.

See also `pairs_values/2` and `pairs_keys/2`.

pairs_values(*+Pairs, -Values*) *[det]*

Remove the keys from a list of Key-Value pairs. Same as `pairs_keys_values(Pairs, _, Values)`

pairs_keys(*+Pairs, -Keys*) *[det]*

Remove the values from a list of Key-Value pairs. Same as `pairs_keys_values(Pairs, Keys, _)`

group_pairs_by_key(*+Pairs, -Joined:list(Key-Values)*) *[det]*

Group values with equivalent (`==/2`) consecutive keys. For example:

```
?- group_pairs_by_key([a-2, a-1, b-4, a-3], X).

X = [a-[2,1], b-[4], a-[3]]
```

Sorting the list of pairs before grouping can be used to group *all* values associated with a key. For example, finding all values associated with the largest key:

```
?- sort(1, @>=, [a-1, b-2, c-3, a-4, a-5, c-6], Ps),
   group_pairs_by_key(Ps, [K-Vs|_]).
K = c,
Vs = [3, 6].
```

In this example, sorting by key only (first argument of `sort/4` is 1) ensures that the order of the values in the original list of pairs is maintained.

Arguments

Pairs	*Key*-Value list
Joined	List of *Key*-Group, where Group is the list of *Values* associated with equivalent consecutive Keys in the same order as they appear in *Pairs*.

transpose_pairs(*+Pairs, -Transposed*) *[det]*

Swap Key-Value to Value-Key. The resulting list is sorted using `keysort/2` on the new key.

map_list_to_pairs(*:Function, +List, -Keyed*)

Create a Key-Value list by mapping each element of *List*. For example, if we have a list of lists we can create a list of Length-*List* using

```
        map_list_to_pairs(length, ListOfLists, Pairs),
```

A.31 library(persistency): Provide persistent dynamic predicates

To be done

- Provide type safety while loading

- Thread safety must now be provided at the user-level. Can we provide generic thread safety? Basically, this means that we must wrap all exported predicates. That might better be done outside this library.
- Transaction management?
- Should assert_<name> only assert if the database does not contain a variant?
- Since we have `prolog_listen/2`, we could use direct `assert/1` and `retract/1` and use the system hooks to deal with the updates.

This module provides simple persistent storage for one or more dynamic predicates. A database is always associated with a module. A module that wishes to maintain a database must declare the terms that can be placed in the database using the directive `persistent/1`.

The `persistent/1` expands each declaration into four predicates:

- `name(Arg, ...)`
- `assert_name(Arg, ...)`
- `retract_name(Arg, ...)`
- `retractall_name(Arg, ...)`

As mentioned, a database can only be accessed from within a single module. This limitation is on purpose, forcing the user to provide a proper API for accessing the shared persistent data.

This module requires the same thread-synchronization as the normal Prolog database. This implies that if each individual assert or retract takes the database from one consistent state to the next, no additional locking is required. If more than one elementary database operation is required to get from one consistent state to the next, both updating and querying the database must be locked using `with_mutex/2`.

Below is a simple example, where adding a user does not need locking as it is a single *assert*, while modifying a user requires both a retract and assert and thus needs to be locked.

```
:- module(user_db,
          [ attach_user_db/1,           % +File
            current_user_role/2,        % ?User, ?Role
            add_user/2,                 % +User, +Role
            set_user_role/2             % +User, +Role
          ]).
:- use_module(library(persistency)).

:- persistent
        user_role(name:atom, role:oneof([user,administrator])).

attach_user_db(File) :-
        db_attach(File, []).

%%      current_user_role(+Name, -Role) is semidet.

current_user_role(Name, Role) :-
        with_mutex(user_db, user_role(Name, Role)).

add_user(Name, Role) :-
```

```
        assert_user_role(Name, Role).

set_user_role(Name, Role) :-
        user_role(Name, Role), !.
set_user_role(Name, Role) :-
        with_mutex(user_db,
                   (  retractall_user_role(Name, _),
                      assert_user_role(Name, Role))).
```

persistent +*Spec*

Declare dynamic database terms. Declarations appear in a directive and have the following format:

```
:- persistent
        <callable>,
        <callable>,
        ...
```

Each specification is a callable term, following the conventions of `library(record)`, where each argument is of the form

```
name:type
```

Types are defined by `library(error)`.

current_persistent_predicate(*:PI*) *[nondet]*

True if *PI* is a predicate that provides access to the persistent database DB.

db_attach(*:File, +Options*)

Use *File* as persistent database for the calling module. The calling module must defined `persistent/1` to declare the database terms. Defined options:

sync(*+Sync*)

One of `close` (close journal after write), `flush` (default, flush journal after write) or `none` (handle as fully buffered stream).

If *File* is already attached this operation may change the `sync` behaviour.

db_attached(*:File*) *[semidet]*

True if the context module attached to the persistent database *File*.

db_assert(*:Term*) *[det]*

Assert *Term* into the database and record it for persistency. Note that if the on-disk file has been modified it is first reloaded.

db_detach *[det]*
Detach persistency from the calling module and delete all persistent clauses from the Prolog database. Note that the file is not affected. After this operation another file may be attached, providing it satisfies the same persistency declaration.

db_retractall(:*Term***)** *[det]*
Retract all matching facts and do the same in the database. If *Term* is unbound, `persistent/1` from the calling module is used as generator.

db_retract(:*Term***)** *[nondet]*
Retract terms from the database one-by-one.

db_sync(:*What***)**
Synchronise database with the associated file. *What* is one of:

reload
Database is reloaded from file if the file was modified since loaded.

update
As `reload`, but use incremental loading if possible. This allows for two processes to examine the same database file, where one writes the database and the other periodycally calls `db_sync(update)` to follow the modified data.

gc
Database was re-written, deleting all retractall statements. This is the same as `gc(50)`.

gc(*Percentage***)**
GC DB if the number of deleted terms is greater than the given percentage of the total number of terms.

gc(*always***)**
GC DB without checking the percentage.

close
Database stream was closed

detach
Remove all registered persistency for the calling module

nop
No-operation performed

With unbound *What*, `db_sync/1` reloads the database if it was modified on disk, gc it if it is dirty and close it if it is opened.

db_sync_all(+*What***)**
Sync all registered databases.

A.32 library(pio): Pure I/O

This library provides pure list-based I/O processing for Prolog, where the communication to the actual I/O device is performed transparently through coroutining. This module itself is just an interface to the actual implementation modules.

A.32.1 library(pure_input): Pure Input from files and streams

To be done Provide support for alternative input readers, e.g. reading terms, tokens, etc.

This module is part of `pio.pl`, dealing with *pure input*: processing input streams from the outside world using pure predicates, notably grammar rules (DCG). Using pure predicates makes non-deterministic processing of input much simpler.

Pure input uses attributed variables to read input from the external source into a list *on demand*. The overhead of lazy reading is more than compensated for by using block reads based on `read_pending_codes/3`.

Ulrich Neumerkel came up with the idea to use coroutining for creating a *lazy list*. His implementation repositioned the file to deal with re-reading that can be necessary on backtracking. The current implementation uses destructive assignment together with more low-level attribute handling to realise pure input on any (buffered) stream.

phrase_from_file(*:Grammar, +File*) *[nondet]*

Process the content of *File* using the DCG rule *Grammar*. The space usage of this mechanism depends on the length of the not committed part of *Grammar*. Committed parts of the temporary list are reclaimed by the garbage collector, while the list is extended on demand due to unification of the attributed tail variable. Below is an example that counts the number of times a string appears in a file. The library dcg/basics provides `string//1` matching an arbitrary string and `remainder//1` which matches the remainder of the input without parsing.

```
:- use_module(library(dcg/basics)).

file_contains(File, Pattern) :-
        phrase_from_file(match(Pattern), File).

match(Pattern) -->
        string(_),
        string(Pattern),
        remainder(_).

match_count(File, Pattern, Count) :-
        aggregate_all(count, file_contains(File, Pattern), Count).
```

This can be called as (note that the pattern must be a string (code list)):

```
?- match_count('pure_input.pl', `file`, Count).
```

phrase_from_file(*:Grammar, +File, +Options*) *[nondet]*

As `phrase_from_file/2`, providing additional *Options*. *Options* are passed to `open/4`.

phrase_from_stream(*:Grammar, +Stream*)

Run Grammer against the character codes on *Stream*. *Stream* must be buffered.

syntax_error(+*Error*) **//**

Throw the syntax error *Error* at the current location of the input. This predicate is designed to be called from the handler of `phrase_from_file/3`.

throws `error(syntax_error(Error), Location)`

lazy_list_location(-*Location*) **//** *[det]*

Determine current (error) location in a lazy list. True when *Location* is an (error) location term that represents the current location in the DCG list.

Arguments	
Location	is a term `file(Name, Line, LinePos, CharNo)` or `stream(Stream, Line, LinePos, CharNo)` if no file is associated to the stream RestLazyList. Finally, if the Lazy list is fully materialized (ends in `[]`), *Location* is unified with `end_of_file-CharCount`.

See also `lazy_list_character_count//1` only provides the character count.

lazy_list_character_count(-*CharCount*) **//**

True when *CharCount* is the current character count in the Lazy list. The character count is computed by finding the distance to the next frozen tail of the lazy list. *CharCount* is one of:

- An integer
- A term end_of_file-Count

See also `lazy_list_location//1` provides full details of the location for error reporting.

stream_to_lazy_list(+*Stream*, -*List*) *[det]*

Create a lazy list representing the character codes in *Stream*. *List* is a partial list ending in an attributed variable. Unifying this variable reads the next block of data. The block is stored with the attribute value such that there is no need to re-read it.

Compatibility Unlike the previous version of this predicate this version does not require a repositionable stream. It does require a buffer size of at least the maximum number of bytes of a multi-byte sequence (6).

A.33 library(portray_text): Portray text

SWI-Prolog has the special string data type. However, in Prolog, text may be represented more traditionally as a list of character-codes, i.e. (small) integers (in SWI-Prolog specifically, those are Unicode code points). This results in output like the following (here using the backquote notation which maps text to a list of codes):

```
?- writeln(`hello`).
[104, 101, 108, 108, 111]

?- atom_codes("hello",X).
X = [104,101,108,108,111].
```

Unless you know the Unicode tables by heart, this is pretty unpleasant for debugging. Loading `library(portray_text)` makes the toplevel and debugger consider certain lists of integers as text and print them as text. This is called ”portraying”. Of course, interpretation is imperfect as there is no way to tell in general whether `[65,66]` should written as `‘AB‘` or as `[65,66]`. Therefore it is important that the user be aware of the fact that this conversion is enabled. This is why this library must be loaded explicitly.

To be able to copy the printed representation and paste it back, printed text is enclosed in *back quotes* if `current_prolog_flag/2` for the flag `back_quotes` is `codes` (the default), and enclosed in *double quotes* otherwise. Certain control characters are printed out in backslash-escaped form.

The default heuristic only considers list of codes as text if the codes are all from the set of 7-bit ASCII without most of the control characters. A code is classified as text by `text_code/1`, which in turn calls `is_text_code/1`. Define `portray_text:is_text_code/1` to succeed on additional codes for more flexibility (by default, that predicate succeeds nowhere). For example:

```
?- maplist([C,R]>>(portray_text:text_code(C)->R=y;R=n),
           ‘G\u00e9n\u00e9rateur‘,Results).
Results = [y,n,y,n,y,y,y,y,y,y].
```

Now make `is_text_code/1` accept anything:

```
?- [user].
|: portray_text:is_text_code(_).
|: ^D
% user://3 compiled 0.00 sec, 1 clauses
true.
```

Then:

```
?- maplist([C,R]>>(portray_text:text_code(C)->R=y;R=n),
           ‘G\u00e9n\u00e9rateur‘,Results).
Results = [y,y,y,y,y,y,y,y,y,y].
```

portray_text(*+OnOff:boolean*) *[det]*

Switch portraying on or off. If `true`, consider lists of integers as list of Unicode code points and print them as corresponding text inside quotes: `‘text‘` or `"text"`. Quoting depends on the value of `current_prolog_flag/2` `back_quotes`. Same as

```
?- set_portray_text(enabled, true).
```

set_portray_text(*+Key, +Value*) *[det]*
set_portray_text(*+Key, ?Old, +New*) *[det]*

Set options for portraying. Defined Keys are:

enabled
Enable/disable portray text

min_length
Only consider for conversion lists of integers that have a length of at least *Value*. Default is 3.

ellipsis
When converting a list that is longer than *Value*, elide the output at the start using ellipsis, leaving only *Value* number of non-elided characters: `'...end'`

is_text_code(+*Code:nonneg*) *[semidet,multifile]*
Multifile hook that can be used to extend the set of character codes that is recognised as likely text. By default, `is_text_code/1` fails everywhere and internally, only non-control ASCII characters (32-126) and the the control codes (9,10,13) are accepted.

To be done we might be able to use the current locale to include the appropriate code page. (Does that really make sense?)

A.34 library(predicate_options): Declare option-processing of predicates

Discussions with Jeff Schultz helped shaping this library

A.34.1 The strength and weakness of predicate options

Many ISO predicates accept options, e.g., `open/4`, `write_term/3`. Options offer an attractive alternative to proliferation into many predicates and using high-arity predicates. Properly defined and used, they also form a mechanism for extending the API of both system and application predicates without breaking portability. I.e., previously fixed behaviour can be replaced by dynamic behaviour controlled by an option where the default is the previously defined fixed behaviour. The alternative to using options is to add an additional argument and maintain the previous definition. While a series of predicates with increasing arity is adequate for a small number of additional parameters, the untyped positional argument handling of Prolog quickly makes this unmanageable.

The ISO standard uses the extensibility offered by options by allowing implementations to extend the set of accepted options. While options form a perfect solution to maintain backward portability in a linear development model, it is not well equipped to deal with concurrent branches because

1. There is no API to find which options are supported in a particular implementation.

2. While the portability problem caused by a missing predicate in Prolog *A* can easily be solved by implementing this predicate, it is much harder to add processing of an additional option to an already existing predicate.

Different Prolog implementations can be seen as concurrent development branches of the Prolog language. Different sets of supported options pose a serious portability issue. Using an option *O* that establishes the desired behaviour on system *A* leads (on most systems) to an error or system *B*. Porting may require several actions:

- Drop *O* (if the option is not vital, such as the layout options to `write_term/3`)

- Replace *O* by *O2* (i.e., a differently named option doing the same)
- Something else (cannot be ported; requires a totally different approach, etc.)

Predicates that process options are particularly a problem when writing a compatibility layer to run programs developed for System *A* on System *B* because complete emulation is often hard, may cause a serious slowdown and is often not needed because the application-to-be-ported only uses options that are shared by all target Prolog implementations. Unfortunately, the consequences of a partial emulation cannot be assessed by tools.

A.34.2 Options as arguments or environment?

We distinguish two views on options. One is to see them as additional parameters that require strict existence, type and domain-checking and the other is to consider them 'locally scoped environment variables'. Most systems adopt the first option. SWI-Prolog adopts the second: it silently ignores options that are not supported but does type and domain checking of option-values. The 'environment' view is commonly used in applications to create predicates supporting more options using the skeleton below. This way of programming requires that *pred1* and *pred2* do not interpret the same option differently. In cases where this is not true, the options must be distributed by *some_pred*. We have been using this programming style for many years and in practice it turns out that the need for active distribution of options is rare. I.e., options either have distinct names or multiple predicates implement the same option but this has the desired effect. An example of the latter is the `encoding` option, which typically needs to be applied consistently.

```
some_pred(..., Options) :-
      pred1(..., Options),
      pred2(..., Options).
```

As stated before, options provide a readable alternative to high-arity predicates and offer a robust mechanism to evolve the API, but at the cost of some runtime overhead and weaker consistency checking, both at compiletime and runtime. From our experience, the 'environment' approach is productive, but the consequence is that mistyped options are silently ignored. The option infrastructure described in this section tries to remedy these problems.

A.34.3 Improving on the current situation

Whether we see options as arguments or locally scoped environment variables, the most obvious way to improve on the current situation is to provide reflective support for options: discover that an argument is an option-list and find what options are supported. Reflective access to options can be used by the compiler and development environment as well as by the runtime system to warn or throw errors.

Options as types

An obvious approach to deal with options is to define the different possible option values as a type and type the argument that processes the option as list(<option_type>), as illustrated below. Considering options as types fully covers the case where we consider options as additional parameters.

```
:- type open_option ---> type(stream_type) |
                         alias(atom) | ... .
:- pred open(source_sink, open_mode, stream, list(open_option)).
```

There are three reasons for considering a different approach:

- There is no consensus about types in the Prolog world, neither about what types should look like, nor whether or not they are desirable. It is not likely that this debate will be resolved shortly.
- Considering options as types does not support the 'environment' view, which we consider the most productive.
- Even when using types, we need reflective access to what options are provided in order to be able to write compile or runtime conditional code.

Reflective access to options

From the above, we conclude that we require reflective access to find out whether an option is supported and valid for a particular predicate. Possible option values must be described by types. Due to lack of a type system, we use `library(error)` to describe allowed option values. Predicate options are declared using `predicate_options/3`:

predicate_options(*:PI, +Arg, +Options*) *[det]*

Declare that the predicate *PI* processes options on *Arg*. *Options* is a list of options processed. Each element is one of:

- Option(ModeAndType) *PI* processes Option. The option-value must comply to ModeAndType. Mode is one of + or - and Type is a type as accepted by `must_be/2`.
- pass_to(:*PI*,*Arg*) The option-list is passed to the indicated predicate.

Below is an example that processes the option `header(boolean)` and passes all options to `open/4`:

```
:- predicate_options(write_xml_file/3, 3,
                     [ header(boolean),
                       pass_to(open/4, 4)
                     ]).

write_xml_file(File, XMLTerm, Options) :-
    open(File, write, Out, Options),
    (   option(header(true), Options, true)
    ->  write_xml_header(Out)
    ;   true
    ),
    ...
```

This predicate may only be used as a *directive* and is processed by `expand_term/2`. Option processing can be specified at runtime using `assert_predicate_options/3`, which is intended to support program analysis.

assert_predicate_options(*:PI, +Arg, +Options, ?New*) *[semidet]*
As predicate_options(:*PI*, +*Arg*, +*Options*). *New* is a boolean indicating whether the declarations have changed. If *New* is provided and `false`, the predicate becomes semidet and fails without modifications if modifications are required.

The predicates below realise the support for compile and runtime checking for supported options.

current_predicate_option(*:PI, ?Arg, ?Option*) *[nondet]*
True when *Arg* of *PI* processes *Option*. For example, the following is true:

```
?- current_predicate_option(open/4, 4, type(text)).
true.
```

This predicate is intended to support conditional compilation using `if/1` ... `endif/0`. The predicate `current_predicate_options/3` can be used to access the full capabilities of a predicate.

check_predicate_option(*:PI, +Arg, +Option*) *[det]*
Verify predicate options at runtime. Similar to `current_predicate_option/3`, but intended to support runtime checking.

Errors
- `existence_error(option, OptionName)` if the option is not supported by *PI*.
- `type_error(Type, Value)` if the option is supported but the value does not match the option type. See `must_be/2`.

The predicates below can be used in a development environment to inform the user about supported options. PceEmacs uses this for colouring option names and values.

current_option_arg(*:PI, ?Arg*) *[nondet]*
True when *Arg* of *PI* processes predicate options. Which options are processed can be accessed using `current_predicate_option/3`.

current_predicate_options(*:PI, ?Arg, ?Options*) *[nondet]*
True when *Options* is the current active option declaration for *PI* on *Arg*. See `predicate_options/3` for the argument descriptions. If *PI* is ground and refers to an undefined predicate, the autoloader is used to obtain a definition of the predicate.

The library can execute a complete check of your program using `check_predicate_options/0`:

check_predicate_options *[det]*
Analyse loaded program for erroneous options. This predicate decompiles the current program and searches for calls to predicates that process options. For each option list, it validates whether the provided options are supported and validates the argument type. This predicate performs partial dataflow analysis to track option-lists inside a clause.

See also `derive_predicate_options/0` can be used to derive declarations for predicates that pass options. This predicate should normally be called before `check_predicate_options/0`.

The library offers predicates that may be used to create declarations for your application. These predicates are designed to cooperate with the module system.

derive_predicate_options *[det]*
Derive new predicate option declarations. This predicate analyses the loaded program to find clauses that process options using one of the predicates from `library(option)` or passes options to other predicates that are known to process options. The process is repeated until no new declarations are retrieved.

See also `autoload/0` may be used to complete the loaded program.

retractall_predicate_options *[det]*
Remove all dynamically (derived) predicate options.

derived_predicate_options(*:PI, ?Arg, ?Options*) *[nondet]*
Derive option arguments using static analysis. True when *Options* is the current *derived* active option declaration for *PI* on *Arg*.

derived_predicate_options(*+Module*) *[det]*
Derive predicate option declarations for a module. The derived options are printed to the `current_output` stream.

A.35 library(prolog_debug): User level debugging tools

This library provides tools to control the Prolog debuggers. Traditionally this code was built-in. Because these tools are only required in (interactive) debugging sessions they have been moved into the library.

prolog:debug_control_hook(*+Action*) *[multifile]*
Allow user-hooks in the Prolog debugger interaction. See the calls below for the provided hooks. We use a single predicate with action argument to avoid an uncontrolled poliferation of hooks.

spy(*:Spec*) *[det]*
nospy(*:Spec*) *[det]*
nospyall *[det]*
Set/clear spy-points. A successfully set or cleared spy-point is reported using `print_message/2`, level `informational`, with one of the following terms, where *Spec* is of the form M:Head.

- `spy(Spec)`
- `nospy(Spec)`

See also `spy/1` and `nospy/1` call the hook `prolog:debug_control_hook/1` to allow for alternative specifications of the thing to debug.

debugging *[det]*
Report current status of the debugger.

trap(*+Exception*) *[det]*
notrap(*+Exception*) *[det]*
Install a trap on `error(Formal, Context)` exceptions that unify with *Exception*. The tracer is started when a matching exception is raised. This predicate enables *debug mode* using `debug/0` to get more context about the exception. Even with debug mode disabled exceptions are still trapped and thus one may call `nodebug/0` to run in normal mode after installing a trap.

The predicate `notrap/1` removes matching (unifying) traps.

See also
- `gtrap/1` to trap using the graphical debugger.
- *Edit exceptions* menu in PceEmacs and the graphical debugger that provide a graphical frontend to trap exceptions.

exception_hook(*+ExIn, -ExOut, +Frame, +Catcher*) *[failure]*
Trap exceptions and consider whether or not to start the tracer.

A.36 library(prolog_jiti): Just In Time Indexing (JITI) utilities

To be done Use `print_message/2` and dynamically figure out the column width.

This module provides utilities to examine just-in-time indexes created by the system and can help diagnosing space and performance issues.

jiti_list *[det]*
jiti_list(*:Spec*) *[det]*
List the JITI (Just In Time Indexes) of selected predicates. The predicate `jiti_list/0` list all just-in-time indexed predicates. The predicate `jiti_list/1` takes one of the patterns below. All parts except for Name can be variables. The last pattern takes an arbitrary number of arguments.

- Module:Head
- Module:Name/Arity
- Module:Name

The columns use the following notation:

- The *Indexed* column describes the `argument(s)` indexed:
  - A plain integer refers to a 1-based argument number
  - *A*+*B* is a multi-argument index on the arguments *A* and *B*.

  - *A/B* is a deep-index on sub-argument *B* of argument *A*.
- The *Buckets* specifies the number of buckets of the hash table
- The *Speedup* specifies the selectivity of the index
- The *Flags* describes additional properties, currently:
  - `L` denotes that the index contains multiple compound terms with the same name/arity that may be used to create deep indexes. The deep indexes themselves are created as just-in-time indexes.

A.37 library(prolog_pack): A package manager for Prolog

See also Installed packages can be inspected using `?- doc_browser.`
To be done
- Version logic
- Find and resolve conflicts
- Upgrade git packages
- Validate git packages
- Test packages: run tests from directory 'test'.

The `library(prolog_pack)` provides the SWI-Prolog package manager. This library lets you inspect installed packages, install packages, remove packages, etc. It is complemented by the built-in `attach_packs/0` that makes installed packages available as libraries.

pack_list_installed *[det]*
List currently installed packages. Unlike `pack_list/1`, only locally installed packages are displayed and no connection is made to the internet.

See also Use `pack_list/1` to find packages.

pack_info(+*Pack*)
Print more detailed information about *Pack*.

pack_search(+*Query*) *[det]*
pack_list(+*Query*) *[det]*
Query package server and installed packages and display results. *Query* is matches case-insensitively against the name and title of known and installed packages. For each matching package, a single line is displayed that provides:

- Installation status
  - **p**: package, not installed
  - **i**: installed package; up-to-date with public version
  - **U**: installed package; can be upgraded
  - **A**: installed package; newer than publically available
  - **l**: installed package; not on server
- Name@Version
- Name@Version(ServerVersion)

- Title

Hint: `?- pack_list('').` lists all packages.

The predicates `pack_list/1` and `pack_search/1` are synonyms. Both contact the package server at http://www.swi-prolog.org to find available packages.

> **See also** `pack_list_installed/0` to list installed packages without contacting the server.

pack_install(+*Spec:atom*) *[det]*

Install a package. *Spec* is one of

- Archive file name
- HTTP URL of an archive file name. This URL may contain a star (*) for the version. In this case pack_install asks for the directory content and selects the latest version.
- GIT URL (not well supported yet)
- A local directory name given as `file://` URL or `'.'`
- A package name. This queries the package repository at http://www.swi-prolog.org

After resolving the type of package, `pack_install/2` is used to do the actual installation.

pack_install(+*Name*, +*Options*) *[det]*

Install package *Name*. Processes the options below. Default options as would be used by `pack_install/1` are used to complete the provided *Options*.

url(+*URL*)
: Source for downloading the package

package_directory(+*Dir*)
: Directory into which to install the package.

global(+*Boolean*)
: If `true`, install in the XDG common application data path, making the pack accessible to everyone. If `false`, install in the XDG user application data path, making the pack accessible for the current user only. If the option is absent, use the first existing and writable directory. If that doesn't exist find locations where it can be created and prompt the user to do so.

interactive(+*Boolean*)
: Use default answer without asking the user if there is a default action.

silent(+*Boolean*)
: If `true` (default false), suppress informational progress messages.

upgrade(+*Boolean*)
: If `true` (default `false`), upgrade package if it is already installed.

rebuild(*Condition*)
: Rebuild the foreign components. *Condition* is one of `if_absent` (default, do nothing if the directory with foreign resources exists), `make` (run `make`) or `true` (run 'make distclean' followed by the default configure and build steps).

git(*+Boolean*)
If `true` (default `false` unless *URL* ends with =.git=), assume the URL is a GIT repository.

link(*+Boolean*)
Can be used if the installation source is a local directory and the file system supports symbolic links. In this case the system adds the current directory to the pack registration using a symbolic link and performs the local installation steps.

Non-interactive installation can be established using the option `interactive(false)`. It is adviced to install from a particular *trusted* URL instead of the plain pack name for unattented operation.

pack_url_file(*+URL, -File*) *[det]*
True if *File* is a unique id for the referenced pack and version. Normally, that is simply the base name, but GitHub archives destroy this picture. Needed by the pack manager.

ssl_verify(*+SSL, +ProblemCert, +AllCerts, +FirstCert, +Error*)
Currently we accept all certificates. We organise our own security using SHA1 signatures, so we do not care about the source of the data.

pack_rebuild(*+Pack*) *[det]*
Rebuilt possible foreign components of *Pack*.

pack_rebuild *[det]*
Rebuild foreign components of all packages.

environment(*-Name, -Value*) *[nondet,multifile]*
Hook to define the environment for building packs. This Multifile hook extends the process environment for building foreign extensions. A value provided by this hook overrules defaults provided by `def_environment/2`. In addition to changing the environment, this may be used to pass additional values to the environment, as in:

```
prolog_pack:environment('USER', User) :-
    getenv('USER', User).
```

Arguments

Name	is an atom denoting a valid variable name
Value	is either an atom or number representing the value of the variable.

pack_upgrade(*+Pack*) *[semidet]*
Try to upgrade the package *Pack*.

To be done Update dependencies when updating a pack from git?

pack_remove(*+Name*) *[det]*
Remove the indicated package.

pack_property(*?Pack, ?Property*) *[nondet]*
True when *Property* is a property of an installed *Pack*. This interface is intended for programs that wish to interact with the package manager. Defined properties are:

directory(*Directory*)
: *Directory* into which the package is installed

version(*Version*)
: Installed version

title(*Title*)
: Full title of the package

author(*Author*)
: Registered author

download(*URL*)
: Official download *URL*

readme(*File*)
: Package `README` file (if present)

todo(*File*)
: Package `TODO` file (if present)

atom_version(*?Atom, ?Version*)
: Translate between atomic version representation and term representation. The term representation is a list of version components as integers and can be compared using `@>`

pack_attach(*+Dir, +Options*) *[det]*
: Attach a single package in *Dir*. The *Dir* is expected to contain the file `pack.pl` and a `prolog` directory. *Options* processed:

duplicate(*+Action*)
: What to do if the same package is already installed in a different directory. *Action* is one of

warning
: Warn and ignore the package

keep
: Silently ignore the package

replace
: Unregister the existing and insert the new package

search(*+Where*)
: Determines the order of searching package library directories. Default is `last`, alternative is `first`.

See also `attach_packs/2` to attach multiple packs from a directory.

A.38 library(prolog_xref): Prolog cross-referencer data collection

See also Where this library analyses *source text*, `library(prolog_codewalk)` may be used to analyse *loaded code*. The `library(check)` exploits `library(prolog_codewalk)` to report on e.g., undefined predicates.

bug `meta_predicate/1` declarations take the module into consideration. Predicates that are both available as meta-predicate and normal (in different modules) are handled as meta-predicate in all places.

This library collects information on defined and used objects in Prolog source files. 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.

The library is exploited by two graphical tools in the SWI-Prolog environment: the XPCE front-end started by `gxref/0`, and `library(prolog_colour)`, which exploits this library for its syntax highlighting.

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`). Input handling is done by the `library(prolog_source)`, which may be hooked to process any source 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 by *Source*.

prolog:called_by(+*Goal*, +*Module*, +*Context*, -*Called***) *[semidet,multifile]*

True when *Called* is a list of callable terms called from *Goal*, handled by the predicate *Module*:*Goal* and executed in the context of the module *Context*. Elements of *Called* may be qualified. If not, they are called in the context of the module *Context*.

prolog:called_by(+*Goal*, -*ListOfCalled***) *[multifile]*

If this succeeds, the cross-referencer assumes *Goal* may call any of the goals in *ListOfCalled*. If this call fails, default meta-goal analysis is used to determine additional called goals.

deprecated New code should use `prolog:called_by/4`

prolog:meta_goal(+*Goal*, -*Pattern***) *[multifile]*

Define meta-predicates. See the examples in this file for details.

prolog:hook(*Goal***)** *[multifile]*

True if *Goal* is a hook that is called spontaneously (e.g., from foreign code).

xref_source(+*Source***)** *[det]*

xref_source(+*Source*, +*Options***)** *[det]*

Generate the cross-reference data for *Source* if not already done and the source is not modified. Checking for modifications is only done for files. *Options* processed:

silent(+*Boolean***)**

If `true` (default `false`), emit warning messages.

module(+*Module***)**

Define the initial context module to work in.

register_called(+*Which***)**

Determines which calls are registerd. *Which* is one of `all`, `non_iso` or `non_built_in`.

comments(+*CommentHandling*)
How to handle comments. If `store`, comments are stored into the database as if the file was compiled. If `collect`, comments are entered to the xref database and made available through `xref_mode/2` and `xref_comment/4`. If `ignore`, comments are simply ignored. Default is to `collect` comments.

process_include(+*Boolean*)
Process the content of included files (default is `true`).

Arguments	
Source	File specification or XPCE buffer

xref_clean(+*Source*) *[det]*
Reset the database for the given source.

xref_current_source(*?Source*)
Check what sources have been analysed.

xref_done(+*Source, -Time*) *[det]*
Cross-reference executed at *Time*

xref_called(*?Source, ?Called, ?By*) *[nondet]*
xref_called(*?Source, ?Called, ?By, ?Cond*) *[nondet]*
xref_called(*?Source, ?Called, ?By, ?Cond, ?Line*) *[nondet]*
True when *By* is called from *Called* in *Source*. Note that `xref_called/3` and `xref_called/4` use `distinct/2` to return only distinct `Called-By` pairs. The `xref_called/5` version may return duplicate `Called-By` if *Called* is called from multiple clauses in *By*, but at most one call per clause.

Arguments	
By	is a head term or one of the reserved terms `'<directive>'(Line)` or `'<public>'(Line)`, indicating the call is from an (often `initialization/1`) directive or there is a `public/1` directive that claims the predicate is called from in some untractable way.
Cond	is the (accumulated) condition as defined by `:- if(Cond)` under which the calling code is compiled.
Line	is the *start line* of the calling clause.

xref_defined(*?Source, +Goal, ?How*) *[nondet]*
Test if *Goal* is accessible in *Source*. If this is the case, *How* specifies the reason why the predicate is accessible. Note that this predicate does not deal with built-in or global predicates, just locally defined and imported ones. *How* is one of of the terms below. Location is one of Line (an integer) or File:Line if the definition comes from an included (using `:- include(File)`) directive.

- `dynamic(Location)`
- `thread_local(Location)`
- `multifile(Location)`
- `public(Location)`

- `local(Location)`
- `foreign(Location)`
- `constraint(Location)`
- `imported(From)`

xref_definition_line(*+How, -Line*)
If the 3th argument of xref_defined contains line info, return this in *Line*.

xref_exported(*?Source, ?Head*) *[nondet]*
True when *Source* exports *Head*.

xref_module(*?Source, ?Module*) *[nondet]*
True if *Module* is defined in *Source*.

xref_uses_file(*?Source, ?Spec, ?Path*) *[nondet]*
True when *Source* tries to load a file using *Spec*.

Arguments

Spec	is a specification for `absolute_file_name/3`
Path	is either an absolute file name of the target file or the atom <not_found>.

xref_op(*?Source, Op*) *[nondet]*
Give the operators active inside the module. This is intended to setup the environment for incremental parsing of a term from the source-file.

Arguments

Op	Term of the form `op(Priority, Type, Name)`

xref_prolog_flag(*?Source, ?Flag, ?Value, ?Line*) *[nondet]*
True when *Flag* is set to *Value* at *Line* in *Source*. This is intended to support incremental parsing of a term from the source-file.

xref_comment(*?Source, ?Title, ?Comment*) *[nondet]*
Is true when *Source* has a section comment with *Title* and *Comment*

xref_comment(*?Source, ?Head, ?Summary, ?Comment*) *[nondet]*
Is true when *Head* in *Source* has the given PlDoc comment.

xref_mode(*?Source, ?Mode, ?Det*) *[nondet]*
Is true when *Source* provides a predicate with *Mode* and determinism.

xref_option(*?Source, ?Option*) *[nondet]*
True when *Source* was processed using *Option*. Options are defined with `xref_source/2`.

xref_meta(*+Source, +Head, -Called*) *[semidet]*
True when *Head* calls *Called* in *Source*.

Arguments

Called	is a list of called terms, terms of the form Term+Extra or terms of the form `//`(Term).

xref_meta(*+Head, -Called*) *[semidet]*
xref_meta_src(*+Head, -Called, +Src*) *[semidet]*

True when *Called* is a list of terms called from *Head*. Each element in *Called* can be of the form Term+Int, which means that Term must be extended with Int additional arguments. The variant `xref_meta/3` first queries the local context.

deprecated New code should use `xref_meta/3`.
To be done
- Split predifined in several categories. E.g., the ISO predicates cannot be redefined.
- Rely on the meta_predicate property for many predicates.

xref_hook(*?Callable*)

Definition of known hooks. Hooks that can be called in any module are unqualified. Other hooks are qualified with the module where they are called.

xref_public_list(*+Spec, +Source, +Options*) *[semidet]*

Find meta-information about File. This predicate reads all terms upto the first term that is not a directive. It uses the module and meta_predicate directives to assemble the information in *Options*. *Options* processed:

path(*-Path*)
Path is the full path name of the referenced file.

module(*-Module*)
Module is the module defines in *Spec*.

exports(*-Exports*)
Exports is a list of predicate indicators and operators collected from the `module/2` term and reexport declarations.

public − *Public*
Public declarations of the file.

meta(*-Meta*)
Meta is a list of heads as they appear in `meta_predicate/1` declarations.

silent(*+Boolean*)
Do not print any messages or raise exceptions on errors.

The information collected by this predicate is cached. The cached data is considered valid as long as the modification time of the file does not change.

Arguments	
Source	is the file from which *Spec* is referenced.

xref_public_list(*+File, -Path, -Export, +Src*) *[semidet]*
xref_public_list(*+File, -Path, -Module, -Export, -Meta, +Src*) *[semidet]*
xref_public_list(*+File, -Path, -Module, -Export, -Public, -Meta, +Src*) *[semidet]*

Find meta-information about *File*. This predicate reads all terms upto the first term that is not a directive. It uses the module and meta_predicate directives to assemble the information described below.

These predicates fail if *File* is not a module-file.

	Arguments
Path	is the canonical path to *File*
Module	is the module defined in *Path*
Export	is a list of predicate indicators.
Meta	is a list of heads as they appear in `meta_predicate/1` declarations.
Src	is the place from which *File* is referenced.

deprecated New code should use `xref_public_list/3`, which unifies all variations using an option list.

xref_source_file(+*Spec*, -*File*, +*Src*) *[semidet]*
xref_source_file(+*Spec*, -*File*, +*Src*, +*Options*) *[semidet]*
Find named source file from *Spec*, relative to *Src*.

A.39 library(quasi_quotations): Define Quasi Quotation syntax

author Jan Wielemaker. Introduction of Quasi Quotation was suggested by Michael Hendricks.
See also Why it's nice to be quoted: quasiquoting for haskell

Inspired by Haskell, SWI-Prolog support *quasi quotation*. Quasi quotation allows for embedding (long) strings using the syntax of an external language (e.g., HTML, SQL) in Prolog text and syntax-aware embedding of Prolog variables in this syntax. At the same time, quasi quotation provides an alternative to represent long strings and atoms in Prolog.

The basic form of a quasi quotation is defined below. Here, *Syntax* is an arbitrary Prolog term that must parse into a *callable* (atom or compound) term and Quotation is an arbitrary sequence of characters, not including the sequence `|}`. If this sequence needs to be embedded, it must be escaped according to the rules of the target language or the 'quoter' must provide an escaping mechanism.

```
{|Syntax||Quotation|}
```

While reading a Prolog term, and if the Prolog flag `quasi_quotes` is set to `true` (which is the case if this library is loaded), the parser collects quasi quotations. After reading the final full stop, the parser makes the call below. Here, *SyntaxName* is the functor name of *Syntax* above and *SyntaxArgs* is a list holding the arguments, i.e., `Syntax =.. [SyntaxName|SyntaxArgs]`. Splitting the syntax into its name and arguments is done to make the quasi quotation parser a predicate with a consistent arity 4, regardless of the number of additional arguments.

```
call(+SyntaxName, +Content, +SyntaxArgs, +VariableNames, -Result)
```

The arguments are defined as

- *SyntaxName* is the principal functor of the quasi quotation syntax. This must be declared using `quasi_quotation_syntax/1` and there must be a predicate SyntaxName/4.
- *Content* is an opaque term that carries the content of the quasi quoted material and position information about the source code. It is passed to `with_quasi_quote_input/3`.

- *SyntaxArgs* carries the additional arguments of the *Syntax*. These are commonly used to make the parameter passing between the clause and the quasi quotation explicit. For example:

```
...,
{|html(Name, Address)||
 <tr><td>Name<td>Address</tr>
 |}
```

- *VariableNames* is the complete variable dictionary of the clause as it is made available throug `read_term/3` with the option `variable_names`. It is a list of terms `Name = Var`.
- *Result* is a variable that must be unified to resulting term. Typically, this term is structured Prolog tree that carries a (partial) representation of the abstract syntax tree with embedded variables that pass the Prolog parameters. This term is normally either passed to a predicate that serializes the abstract syntax tree, or a predicate that processes the result in Prolog. For example, HTML is commonly embedded for writing HTML documents (see `library(http/html_write)`). Examples of languages that may be embedded for processing in Prolog are SPARQL, RuleML or regular expressions.

The file `library(http/html_quasiquotations)` provides the, suprisingly simple, quasi quotation parser for HTML.

with_quasi_quotation_input(+*Content*, -*Stream*, :*Goal***)** *[det]*

Process the quasi-quoted *Content* using *Stream* parsed by *Goal*. *Stream* is a temporary stream with the following properties:

- Its initial *position* represents the position of the start of the quoted material.
- It is a text stream, using `utf8` *encoding*.
- It allows for repositioning
- It will be closed after *Goal* completes.

	Arguments
Goal	is executed as `once(Goal)`. *Goal* must succeed. Failure or exceptions from *Goal* are interpreted as syntax errors.

See also `phrase_from_quasi_quotation/2` can be used to process a quotation using a grammar.

phrase_from_quasi_quotation(:*Grammar*, +*Content***)** *[det]*

Process the quasi quotation using the DCG *Grammar*. Failure of the grammar is interpreted as a syntax error.

See also `with_quasi_quotation_input/3` for processing quotations from stream.

quasi_quotation_syntax(:*SyntaxName***)** *[det]*

Declare the predicate *SyntaxName*/4 to implement the the quasi quote syntax *SyntaxName*. Normally used as a directive.

quasi_quotation_syntax_error(*+Error*)
Report `syntax_error(Error)` using the current location in the quasi quoted input parser.

throws `error(syntax_error(Error), Position)`

A.40 library(random): Random numbers

author R.A. O'Keefe, V.S. Costa, L. Damas, Jan Wielemaker
See also Built-in function `random/1`: A is `random(10)`

This library is derived from the DEC10 library random. Later, the core random generator was moved to C. The current version uses the SWI-Prolog arithmetic functions to realise this library. These functions are based on the GMP library.

random(*-R:float*) *[det]*
Binds *R* to a new random float in the *open* interval (0.0,1.0).

See also
- `setrand/1`, `getrand/1` may be used to fetch/set the state.
- In SWI-Prolog, `random/1` is implemented by the function `random_float/0`.

random_between(*+L:int, +U:int, -R:int*) *[semidet]*
Binds *R* to a random integer in [*L*,*U*] (i.e., including both *L* and *U*). Fails silently if *U*<*L*.

random(*+L:int, +U:int, -R:int*) *[det]*
random(*+L:float, +U:float, -R:float*) *[det]*
Generate a random integer or float in a range. If *L* and *U* are both integers, *R* is a random integer in the half open interval [*L*,*U*). If *L* and *U* are both floats, *R* is a float in the open interval (*L*,*U*).

deprecated Please use `random/1` for generating a random float and `random_between/3` for generating a random integer. Note that `random_between/3` includes the upper bound, while this predicate excludes it.

setrand(*+State*) *[det]*
getrand(*-State*) *[det]*
Query/set the state of the random generator. This is intended for restarting the generator at a known state only. The predicate `setrand/1` accepts an opaque term returned by `getrand/1`. This term may be asserted, written and read. The application may not make other assumptions about this term.

For compatibility reasons with older versions of this library, `setrand/1` also accepts a term `rand(A,B,C)`, where A, B and C are integers in the range 1..30,000. This argument is used to seed the random generator. Deprecated.

Errors `existence_error(random_state, _)` is raised if the underlying infrastructure cannot fetch the random state. This is currently the case if SWI-Prolog is not compiled with the GMP library.
See also `set_random/1` and `random_property/1` provide the SWI-Prolog native implementation.

maybe *[semidet]*

Succeed/fail with equal probability (variant of `maybe/1`).

maybe(*+P*) *[semidet]*

Succeed with probability *P*, fail with probability 1-*P*

maybe(*+K, +N*) *[semidet]*

Succeed with probability *K/N* (variant of `maybe/1`)

random_perm2(*?A, ?B, ?X, ?Y*) *[semidet]*

Does *X*=*A*,*Y*=*B* or *X*=*B*,*Y*=*A* with equal probability.

random_member(*-X, +List:list*) *[semidet]*

X is a random member of *List*. Equivalent to random_between(1, |*List*|), followed by `nth1/3`. Fails of *List* is the empty list.

Compatibility Quintus and SICStus libraries.

random_select(*-X, +List, -Rest*) *[semidet]*
random_select(*+X, -List, +Rest*) *[det]*

Randomly select or insert an element. Either *List* or *Rest* must be a list. Fails if *List* is the empty list.

Compatibility Quintus and SICStus libraries.

random_subseq(*+List, -Subseq, -Complement*) *[det]*
random_subseq(*-List, +Subseq, +Complement*) *[semidet]*

Selects a random subsequence *Subseq* of *List*, with *Complement* containing all elements of *List* that were not selected. Each element of *List* is included with equal probability in either *Subseq* or *Complement*.

`random_subseq/3` may also be called with *Subseq* and *Complement* bound and *List* unbound, which will recreate *List* by randomly interleaving *Subseq* and *Complement*. This mode may fail randomly, matching SICStus behavior. The failure probability corresponds to the probability of the "forward" mode selecting a *Subseq*/*Complement* combination with different lengths.

Compatibility SICStus 4

randset(*+K:int, +N:int, -S:list(int)*) *[det]*

S is a sorted list of *K* unique random integers in the range 1..*N*. The implementation uses different techniques depending on the ratio *K/N*. For small *K/N* it generates a set of *K* random numbers, removes the duplicates and adds more numbers until |*S*| is *K*. For a large *K/N* it enumerates 1..*N* and decides randomly to include the number or not. For example:

```
?- randset(5, 5, S).
S = [1, 2, 3, 4, 5].          (always)
?- randset(5, 20, S).
S = [2, 7, 10, 19, 20].
```

See also `randseq/3`.

randseq(*+K:int, +N:int, -List:list(int)*) *[det]*

S is a list of *K* unique random integers in the range 1..*N*. The order is random. Defined as

```
randseq(K, N, List) :-
      randset(K, N, Set),
      random_permutation(Set, List).
```

See also `randset/3`.

random_permutation(*+List, -Permutation*) *[det]*
random_permutation(*-List, +Permutation*) *[det]*

Permutation is a random permutation of *List*. This is intended to process the elements of *List* in random order. The predicate is symmetric.

Errors instantiation_error, `type_error(list, _)`.

random_numlist(*+P, +L, +U, -List*) *[det]*

Unify *List* with an ascending list of integers between *L* and *U* (inclusive). Each integer in the range *L*..*U* is included with probability *P*.

Compatibility SICStus 4

A.41 library(readutil): Read utilities

See also
- `library(pure_input)` allows for processing files with DCGs.
- `library(lazy_lists)` for creating lazy lists from input.

This library provides some commonly used reading predicates. As these predicates have proven to be time-critical in some applications we moved them to C. For compatibility as well as to reduce system dependency, we link the foreign code at runtime and fallback to the Prolog implementation if the shared object cannot be found.

read_line_to_codes(*+Stream, -Line:codes*) *[det]*

Read the next line of input from *Stream*. Unify content of the lines as a list of character codes with *Line 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 a trailing newline character.

read_line_to_codes(*+Stream, -Line, ?Tail*) *[det]*

Difference-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 reading a block of lines up to 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_line_to_string(+*Stream*, -*String*) *[det]*

Read the next line from *Stream* into *String*. *String* does not contain the line terminator. *String* is unified with the *atom* `end_of_file` if the end of the file is reached.

> **See also** `read_string/5` can be used to read lines with separated records without creating intermediate strings.

read_stream_to_codes(+*Stream*, -*Codes*) *[det]*

read_stream_to_codes(+*Stream*, -*Codes*, ?*Tail*) *[det]*

Read input from *Stream* to a list of character codes. The version `read_stream_to_codes/3` creates a difference-list.

read_file_to_codes(+*Spec*, -*Codes*, +*Options*) *[det]*

Read the file *Spec* into a list of *Codes*. *Options* is split into options for `absolute_file_name/3` and `open/4`. In addition, the following option is provided:

tail(*?Tail*)
Read the data into a *difference list Codes**Tail*.

> **See also** `phrase_from_file/3` and `read_file_to_string/3`.

read_file_to_string(+*Spec*, -*String*, +*Options*) *[det]*

Read the file *Spec* into a the string *String*. *Options* is split into options for `absolute_file_name/3` and `open/4`.

> **See also** `phrase_from_file/3` and `read_file_to_codes/3`.

read_file_to_terms(+*Spec*, -*Terms*, +*Options*) *[det]*

Read the file *Spec* into a list of terms. *Options* is split over `absolute_file_name/3`, `open/4` and `read_term/3`. In addition, the following option is processed:

tail(*?Tail*)
If present, *Terms**Tail* forms a *difference list*.

Note that the *output* options of `read_term/3`, such as `variable_names` or `subterm_positions` will cause `read_file_to_terms/3` to fail if *Spec* contains multiple terms because the values for the different terms will not unify.

A.42 library(record): Access named fields in a term

The library `record` provides named access to fields in a record represented as a compound term such as `point(X, Y)`. The Prolog world knows various approaches to solve this problem, unfortunately with no consensus. The approach taken by this library is proposed by Richard O'Keefe on the SWI-Prolog mailinglist.

The approach automates a technique commonly described in Prolog text-books, where access and modification predicates are defined for the record type. Such predicates are subject to normal import/export as well as analysis by cross-referencers. Given the simple nature of the access predicates, an optimizing compiler can easily inline them for optimal performance.

A record is defined using the directive `record/1`. We introduce the library with a short example:

```
:- record point(x:integer=0, y:integer=0).

        ...,
        default_point(Point),
        point_x(Point, X),
        set_x_of_point(10, Point, Point1),

        make_point([y(20)], YPoint),
```

The principal functor and arity of the term used defines the name and arity of the compound used as records. Each argument is described using a term of the format below.

⟨*name*⟩[:⟨*type*⟩][=⟨*default*⟩]

In this definition, ⟨*name*⟩ is an atom defining the name of the argument, ⟨*type*⟩ is an optional type specification as defined by `must_be/2` from library `error`, and ⟨*default*⟩ is the default initial value. The ⟨*type*⟩ defaults to `any`. If no default value is specified the default is an unbound variable.

A record declaration creates a set of predicates through *term-expansion*. We describe these predicates below. In this description, ⟨*constructor*⟩ refers to the name of the record ('point' in the example above) and ⟨*name*⟩ to the name of an argument (field).

- *default_⟨constructor⟩(-Record)*
  Create a new record where all fields have their default values. This is the same as make_⟨*constructor*⟩([], Record).

- *make_⟨constructor⟩(+Fields, -Record)*
  Create a new record where specified fields have the specified values and remaining fields have their default value. Each field is specified as a term ⟨*name*⟩(⟨*value*⟩). See example in the introduction.

- *make_⟨constructor⟩(+Fields, -Record, -RestFields)*
  Same as make_⟨*constructor*⟩/2, but named fields that do not appear in *Record* are returned in *RestFields*. This predicate is motivated by option-list processing. See library `option`.

- *⟨constructor⟩_⟨name⟩(Record, Value)*
  Unify *Value* with argument in *Record* named ⟨*name*⟩.[2]

[2] Note this is not called 'get_' as it performs unification and can perfectly well instantiate the argument.

- *⟨constructor⟩_data(?Name, +Record, ?Value)*
  True when *Value* is the value for the field named *Name* in *Record*. This predicate does not perform type-checking.

- *set_⟨name⟩_of_⟨constructor⟩(+Value, +OldRecord, -NewRecord)*
  Replace the value for ⟨*name*⟩ in *OldRecord* by *Value* and unify the result with *NewRecord*.

- *set_⟨name⟩_of_⟨constructor⟩(+Value, !Record)*
  Destructively replace the argument ⟨*name*⟩ in *Record* by *Value* based on `setarg/3`. Use with care.

- *nb_set_⟨name⟩_of_⟨constructor⟩(+Value, !Record)*
  As above, but using non-backtrackable assignment based on `nb_setarg/3`. Use with *extreme* care.

- *set_⟨constructor⟩_fields(+Fields, +Record0, -Record)*
  Set multiple fields using the same syntax as make_⟨*constructor*⟩/2, but starting with *Record0* rather than the default record.

- *set_⟨constructor⟩_fields(+Fields, +Record0, -Record, -RestFields)*
  Similar to set_⟨*constructor*⟩`_fields/4`, but fields not defined by ⟨*constructor*⟩ are returned in *RestFields*.

- *set_⟨constructor⟩_field(+Field, +Record0, -Record)*
  Set a single field specified as a term ⟨*name*⟩(⟨*value*⟩).

record(+*Spec*)
: The construct `:- record Spec, ...` is used to define access to named fields in a compound. It is subject to term-expansion (see `expand_term/2`) and cannot be called as a predicate. See section A.42 for details.

A.43 library(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 if 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 it does not already exist.

registry_set_key(+*Path*, +*Name*, +*Value***)**
Associate a named value to this key. Creates (a path to) the key if it 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 like `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 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 includes the above predicates, it is given as an example:

```
shell_register_prolog(Ext) :-
        current_prolog_flag(argv, [Me|_]),
        atomic_list_concat(['"', 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.44 library(settings): Setting management

author Jan Wielemaker

See also `library(config)` distributed with XPCE provides an alternative aimed at graphical applications.

This library allows management of configuration settings for Prolog applications. Applications define settings in one or multiple files using the directive `setting/4` as illustrated below:

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

:- setting(version, atom,   '1.0', 'Current version').
:- setting(timeout, number,    20, 'Timeout in seconds').
```

The directive is subject to `term_expansion/2`, which guarantees proper synchronisation of the database if source-files are reloaded. This implies it is **not** possible to call `setting/4` as a predicate.

Settings are local to a module. This implies they are defined in a two-level namespace. Managing settings per module greatly simplifies assembling large applications from multiple modules that configuration through settings. This settings management library ensures proper access, loading and saving of settings.

setting(*:Name, +Type, +Default, +Comment*) *[det]*

Define a setting. *Name* denotes the name of the setting, *Type* its type. *Default* is the value before it is modified. *Default* can refer to environment variables and can use arithmetic expressions as defined by `eval_default/4`.

If a second declaration for a setting is encountered, it is ignored if *Type* and *Default* are the same. Otherwise a permission_error is raised.

Arguments

Name	*Name* of the setting (an atom)
Type	*Type* for setting. One of `any` or a type defined by `must_be/2`.
Default	*Default* value for the setting.
Comment	Atom containing a (short) descriptive note.

setting(*:Name, ?Value*) *[nondet]*

True when *Name* is a currently defined setting with *Value*. Note that `setting(Name, Value)` only enumerates the settings of the current module. All settings can be enumerated using `setting(Module:Name, Value)`. This predicate is `det` if *Name* is ground.

Errors `existence_error(setting, Name)`

env(*+Name:atom, -Value:number*) *[det]*
env(*+Name:atom, +Default:number, -Value:number*) *[det]*

Evaluate environment variables on behalf of arithmetic expressions.

set_setting(*:Name, +Value*) *[det]*

Change a setting. Performs existence and type-checking for the setting. If the effective value of the setting is changed it broadcasts the event below.

```
settings(changed(Module:Name, Old, New))
```

Note that modified settings are **not** automatically persistent. The application should call `save_settings/0` to persist the changes.

Errors
- `existence_error(setting, Name)`
- `type_error(Type, Value)`

restore_setting(*:Name*) *[det]*
Restore the value of setting *Name* to its default. Broadcast a change like `set_setting/2` if the current value is not the default.

set_setting_default(*:Name, +Default*) *[det]*
Change the default for a setting. The effect is the same as `set_setting/2`, but the new value is considered the default when saving and restoring a setting. It is intended to change application defaults in a particular context.

load_settings(*File*) *[det]*
load_settings(*File, +Options*) *[det]*
Load local settings from *File*. Succeeds if *File* does not exist, setting the default save-file to *File*. *Options* are:

undefined(*+Action*)
Define how to handle settings that are not defined. When `error`, an error is printed and the setting is ignored. when `load`, the setting is loaded anyway, waiting for a definition.

If possibly changed settings need to be persistent, the application must call `save_settings/0` as part of its shutdown. In simple cases calling `at_halt(save_settings)` is sufficient.

save_settings *[semidet]*
save_settings(*+File*) *[semidet]*
Save modified settings to *File*. Fails silently if the settings file cannot be written. The `save_settings/0` only attempts to save the settings file if some setting was modified using `set_setting/2`.

Errors `context_error(settings, no_default_file)` for `save_settings/0` if no default location is known.

current_setting(*?Setting*) *[nondet]*
True if *Setting* is a currently defined setting

setting_property(*+Setting, +Property*) *[det]*
setting_property(*?Setting, ?Property*) *[nondet]*
Query currently defined settings. *Property* is one of

comment(*-Atom*)

type(*-Type*)
Type of the setting.

default(*-Default*)
Default value. If this is an expression, it is evaluated.

source(*-File:-Line*)
Location where the setting is defined.

list_settings *[det]*
list_settings(*+Module*) *[det]*
List settings to `current_output`. The second form only lists settings on the matching module.

To be done Compute the required column widths

convert_setting_text(*+Type, +Text, -Value*)
Converts from textual form to Prolog *Value*. Used to convert values obtained from the environment. Public to provide support in user-interfaces to this library.

Errors `type_error(Type, Value)`

A.45 library(strings): String utilities

See also
- `format/3` can format to a string as well. The `library(lynx/format)` provides primitive to wrap long strings.
- The core system provides many additional string processing predicates.

To be done There are probably many other high level string predicates that belong in this library. For example, predicates similar to the functions in https://docs.python.org/3/library/textwrap.html

This module provides string handling utilities, currently notably for dealing with multi-line strings and *interpolation*. The library provides a couple of primitives as well definitions for the `string` *quasi quotation* syntax. The latter allows for constructing both single line and multi-line long strings based on template interpolation. Below is a simple example using the quasi quotation syntax.

```
test(To) :-
    write({|string(To)||
           | Dear {To},
           |
           | I'm happy to announce a string interpolation quasi quoter.
           |}.
```

Warning
The general purpose string interpolation implemented by this library should **not** be used to create strings for a formal language such as HTML, JavaScript, SQL, etc. because the result will be subject to **injection attacks**, providing a serious **security risc**. The core idea of quasi quotation is to know about the target language and interpolate Prolog data into the template **while respecting the syntax of the target language**, notable to **escape certain characters where needed**. See also `library(http/html_write)` and `library(http/js_write)` which define quasi quotation rules for HTML and JavaScript.

string(*+Content, +Args, +Binding, -DOM*)

Implements the quasi quotation syntax `string`. If the first character of the content is a newline (i.e., there is a newline *immediately* after the `||` token) this first uses `dedent_lines/3` to the remove common white space prefix from the lines. This is called with the option `chars("\s\t|")`, i.e., also removing `|` characters and `tab(8)`.

If the quasi quotation syntax carries arguments (e.g., `string(To)`), the string is compiled into a function that produces the result of interpolating the arguments into the template. See user functions on dict objects. If there are no arguments, the result is simply the final string.

See also
- `interpolate_string/4` for the interpolation syntax.
- Section for examples and discussion.

To be done Specify tab width and allow for {@Goal} templates.

interpolate_string(*:In, -Out, +Map, +Options*)

Establish a string from a template by replacing patterns. Supported patterns are:

{*Name***}**
If *Map* contains `Name=Value`, insert *Value* using `write/1`. If *Name* does not appear in *Map*, raise an existence error. *Name* must satisfy the rules for a Prolog variable.

{*Name,Default***}**
As above, but if *Name* does not appear in *Map*, use *Value*

{*@(Goal)***}**
Insert the output (to `current_output`) of *Goal* here. For safety reasons only accepted if *Options* contains `goals(true)`

string_lines(*?String, ?Lines*) *[det]*

True when *String* represents *Lines*. This follows the normal text convention that a line is defined as a possible empty string followed by a newline character ("`\n`"). E.g.

```
?- string_lines("a\nb\n", L).
L = ["a", "b"].
?- string_lines(S, ["a", "b"]).
S = "a\nb\n".
```

This predicate is a true *relation* if both arguments are in canonical form, i.e. all text is represented as strings and the first argument ends with a newline. The implementation tolerates non-canonical input: other types than strings are accepted and *String* does not need to end with a newline.

See also `split_string/4`. Using `split_string(String, "\n", "", Lines)` on a string that ends in a newline adds an additional empty string compared to `string_lines/2`.

dedent_lines(*+In, -Out, +Options*)

Remove shared indentation for all lines in a string. Lines are separated by "`\n`" – conversion to and from external forms (such as "`\r\n`") are typically done by the I/O predicates. A final "`\n`" is preserved.

Options:

tab(*N*)
Assume tabs at columns of with *N*. When omitted, tabs are taken literally and only exact matches are removed.

chars(*CodesOrString*)
Characters to remove. This can notably be used to remove additional characters such as `*` or '`|`'. Default is `" \t"`.

indent_lines(*+Prefix, +In, -Out*) *[det]*
Add *Prefix* to the beginning of lines in *In*. Lines are separated by "`\n`" – conversion to and from external forms (such as "`\r\n`") are typically done by the I/O predicates. Lines that consist entirely of whitespace are left as-is.

indent_lines(*:Filter, +Prefix, +In, -Out*) *[det]*
Similar to `indent_lines/3`, but only adds *Prefix* to lines for which `call(Filter, Line)` succeeds.

A.46 library(simplex): Solve linear programming problems

author Markus Triska

A.46.1 Introduction

A **linear programming problem** or simply **linear program** (LP) consists of:

- a set of *linear* **constraints**
- a set of **variables**
- 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. As one basic example, consider 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.

Solving an LP or integer linear program (ILP) with this library typically comprises 4 stages:

1. an initial state is generated with `gen_state/1`
2. all relevant constraints are added with `constraint/3`
3. `maximize/3` or `minimize/3` are used to obtain a *solved state* that represents an optimum solution
4. `variable_value/3` and `objective/2` are used on the solved state to obtain variable values and the objective function at the optimum.

The most frequently used predicates are thus:

gen_state(*-State*)
: Generates an initial state corresponding to an empty linear program.

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.

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*. \arg{*S*} is unified with an internal representation of the solved instance.

minimize(*+Objective, +S0, -S*)
: Analogous to `maximize/3`.

variable_value(*+State, +Variable, -Value*)
: *Value* is unified with the value obtained for *Variable*. *State* must correspond to a solved instance.

objective(*+State, -Objective*)
: Unifies *Objective* with the result of the objective function at the obtained extremum. *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.46.2 Delayed column generation

Delayed column generation means that more constraint columns are added to an existing LP. The following predicates are frequently used when this method is applied:

constraint(*+Name, +Constraint, +S0, -S*)
: Like `constraint/3`, and attaches the name *Name* (an atom or compound term) to the new constraint.

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.

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.

An example application of *delayed column generation* to solve a *bin packing* task is available from: **metalevel.at/various/colgen/**

A.46.3 Solving LPs with special structure

The following predicates allow you to solve specific kinds of LPs more efficiently:

transportation(+*Supplies, +Demands, +Costs, -Transport*)
Solves a transportation problem. *Supplies* and *Demands* must be lists of non-negative integers. Their respective sums must be equal. *Costs* is a list of lists representing the cost matrix, where an entry (*i*,*j*) denotes the integer 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*.

assignment(+*Cost, -Assignment*)
Solves a linear assignment problem. *Cost* is a list of lists representing the quadratic cost matrix, where element (i,j) denotes the integer 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.

A.46.4 Examples

We include a few examples for solving LPs with this library.

Example 1

This is the "radiation therapy" example, taken from *Introduction to Operations Research* by Hillier and Lieberman.

Prolog DCG notation is used to *implicitly* thread the state through posting the constraints:

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

radiation(S) :-
        gen_state(S0),
        post_constraints(S0, S1),
        minimize([0.4*x1, 0.5*x2], S1, S).

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).
```

An example query:

```
?- radiation(S), variable_value(S, x1, Val1),
                 variable_value(S, x2, Val2).
Val1 = 15 rdiv 2,
Val2 = 9 rdiv 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.

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

knapsack(S) :-
        knapsack_constraints(S0),
        maximize([7*x(1), 4*x(2)], S0, S).

knapsack_constraints(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).
```

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_constraints(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.

Example 3

We are given:

- 3 coins each worth 1 unit
- 20 coins each worth 5 units and
- 10 coins each worth 20 units.

The task is to find a *minimal* number of these coins that amount to 111 units in total. We introduce variables `c(1)`, `c(5)` and `c(20)` denoting how many coins to take of the respective type:

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

coins(S) :-
        gen_state(S0),
        coins(S0, S).

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)]).
```

An example query:

```
?- coins(S), variable_value(S, c(1), C1),
             variable_value(S, c(5), C5),
             variable_value(S, c(20), C20).

C1 = 1,
C5 = 2,
C20 = 5.
```

A.47 library(solution_sequences): Modify solution sequences

See also

- all solution predicates `findall/3`, `bagof/3` and `setof/3`.
- `library(aggregate)`

The meta predicates of this library modify the sequence of solutions of a goal. The modifications and the predicate names are based on the classical database operations DISTINCT, LIMIT, OFFSET, ORDER BY and GROUP BY.

These predicates were introduced in the context of the SWISH Prolog browser-based shell, which can represent the solutions to a predicate as a table. Notably wrapping a goal in `distinct/1` avoids duplicates in the result table and using `order_by/2` produces a nicely ordered table.

However, the predicates from this library can also be used to stay longer within the clean paradigm where non-deterministic predicates are composed from simpler non-deterministic predicates by means of conjunction and disjunction. While evaluating a conjunction, we might want to eliminate duplicates of the first part of the conjunction. Below we give both the classical solution for solving variations of `(a(X), b(X))` and the ones using this library side-by-side.

Avoid duplicates of earlier steps

`setof(X, a(X), Xs),`
`member(X, Xs),`
`b(X).`
	`distinct(a(X)),`
	`b(X)`

Note that the `distinct/1` based solution returns the first result of `distinct(a(X))` immediately after `a/1` produces a result, while the `setof/3` based solution will first compute all results of `a/1`.

Only try b(X) only for the top-10 a(X)

`setof(X, a(X), Xs),`
`reverse(Xs, Desc),`
`first_max_n(10, Desc, Limit),`
`member(X, Limit),`
`b(X)`
	`limit(10, order_by([desc(X)], a(X))),`
	`b(X)`

Here we see power of composing primitives from this library and staying within the paradigm of pure non-deterministic relational predicates.

distinct(*:Goal*)
distinct(*?Witness, :Goal*)

True if *Goal* is true and no previous solution of *Goal* bound *Witness* to the same value. As previous answers need to be copied, equivalence testing is based on *term variance* (`=@=/2`). The variant `distinct/1` is equivalent to `distinct(Goal,Goal)`.

If the answers are ground terms, the predicate behaves as the code below, but answers are returned as soon as they become available rather than first computing the complete answer set.

```
distinct(Goal) :-
    findall(Goal, Goal, List),
    list_to_set(List, Set),
    member(Goal, Set).
```

reduced(*:Goal*)
reduced(*?Witness, :Goal, +Options*)

Similar to `distinct/1`, but does not guarantee unique results in return for using a limited amount of memory. Both `distinct/1` and `reduced/1` create a table that block duplicate results. For `distinct/1`, this table may get arbitrary large. In contrast, `reduced/1` discards the table and starts a new one of the table size exceeds a specified limit. This filter is useful for reducing the number of answers when processing large or infinite long tail distributions. *Options*:

size_limit(+*Integer***)**
Max number of elements kept in the table. Default is 10,000.

limit(+*Count, :Goal***)**
Limit the number of solutions. True if *Goal* is true, returning at most *Count* solutions. Solutions are returned as soon as they become available.

	Arguments
Count	is either `infinite`, making this predicate equivalent to `call/1` or an integer. If *Count* < *1* this predicate fails immediately.

offset(+*Count, :Goal***)**
Ignore the first *Count* solutions. True if *Goal* is true and produces more than *Count* solutions. This predicate computes and ignores the first *Count* solutions.

call_nth(*:Goal, ?Nth***)**
True when *Goal* succeeded for the *Nth* time. If *Nth* is bound on entry, the predicate succeeds deterministically if there are at least *Nth* solutions for *Goal*.

order_by(+*Spec, :Goal***)**
Order solutions according to *Spec*. *Spec* is a list of terms, where each element is one of. The ordering of solutions of *Goal* that only differ in variables that are *not* shared with *Spec* is not changed.

asc(*Term***)**
Order solution according to ascending *Term*

desc(*Term***)**
Order solution according to descending *Term*

This predicate is based on `findall/3` and (thus) variables in answers are *copied*.

group_by(+*By, +Template, :Goal, -Bag***)** *[nondet]*
Group bindings of *Template* that have the same value for *By*. This predicate is almost the same as `bagof/3`, but instead of specifying the existential variables we specify the free variables. It is provided for consistency and complete coverage of the common database vocabulary.

A.48 library(tables): XSB interface to tables

This module provides an XSB compatible library to access tables as created by tabling (see `table/1`). The aim of this library is first of all compatibility with XSB. This library contains some old and internal XSB predicates that are marked deprecated.

t_not(*:Goal*)
Tabled negation.

deprecated This is a synonym to `tnot/1`.

tfindall(*+Template, :Goal, -Answers*)
This predicate emerged in XSB in an attempt to provide a safer alternative to `findall/3`. This doesn't really work in XSB and the SWI-Prolog emulation is a simple call to `findall/3`. Note that *Goal* may not be a variant of an *incomplete* table.

deprecated Use `findall/3`

set_pil_on
set_pil_off
Dummy predicates for XSB compatibility.

deprecated These predicates have no effect.

get_call(*:CallTerm, -Trie, -Return*) *[semidet]*
True when *Trie* is an answer trie for a variant of *CallTerm*. *Return* is a term ret/N with N variables that share with variables in *CallTerm*. The *Trie* contains zero or more instances of the *Return* term. See also `get_calls/3`.

get_calls(*:CallTerm, -Trie, -Return*) *[nondet]*
True when *Trie* is an answer trie for a variant that unifies with *CallTerm* and Skeleton is the answer skeleton. See `get_call/3` for details.

get_returns(*+ATrie, -Return*) *[nondet]*
True when *Return* is an answer template for the AnswerTrie.

Arguments

Return	is a term `ret(...)`. See `get_calls/3`.

get_returns(*+AnswerTrie, -Return, -NodeID*) *[nondet]*
True when *Return* is an answer template for the *AnswerTrie* and the answer is represented by the trie node *NodeID*.

Arguments

Return	is a term `ret(...)`. See `get_calls/3`.

get_returns_and_tvs(*+AnswerTrie, -Return, -TruthValue*) *[nondet]*
Identical to `get_returns/2`, but also obtains the truth value of a given answer, setting *TruthValue* to `t` if the answer is unconditional and to `u` if it is conditional. If a conditional answer has multiple delay lists, this predicate will succeed only once, so that using this predicate may be more efficient than `get_residual/2` (although less informative)

get_returns_and_dls(*+AnswerTrie, -Return, :DelayLists*) *[nondet]*
True when *Return* appears in *AnswerTrie* with the given *DelayLists*. *DelayLists* is a list of lists, where the inner lists expresses a conjunctive condition and and outer list a disjunction.

get_residual(*:CallTerm, -DelayList*) *[nondet]*

True if *CallTerm* appears in a table and has *DelayList*. SWI-Prolog's representation for a delay is a body term, more specifically a disjunction of conjunctions. The XSB representation is non-deterministic and uses a list to represent the conjunction.

The delay condition is a disjunction of conjunctions and is represented as such in the native SWI-Prolog interface as a nested term of ;/2 and ,/2, using `true` if the answer is unconditional. This XSB predicate returns the associated conjunctions non-deterministically as a list.

See also `call_residual_program/2` from `library(wfs)`.

get_returns_for_call(*:CallTerm, -AnswerTerm*) *[nondet]*

True if *AnswerTerm* appears in the tables for the *variant CallTerm*.

abolish_table_pred(*:CallTermOrPI*)

Invalidates all tabled subgoals for the predicate denoted by the predicate or term indicator Pred.

> **To be done** If Pred has a subgoal that contains a conditional answer, the default behavior will be to transitively abolish any tabled predicates with subgoals having answers that depend on any conditional answers of S.

abolish_table_call(*+Head*) *[det]*
abolish_table_call(*+Head, +Options*) *[det]*

Same as `abolish_table_subgoals/1`. See also `abolish_table_pred/1`.

> **deprecated** Use abolish_table_subgoals/[1,2].

abolish_table_subgoals(*:Head, +Options*)

Behaves as `abolish_table_subgoals/1`, but allows the default `table_gc_action` to be over-ridden with a flag, which can be either `abolish_tables_transitively` or `abolish_tables_singly`.

> **Compatibility** *Options* is compatible with XSB, but does not follow the ISO option handling conventions.

A.49 library(terms): Term manipulation

> **Compatibility** YAP, SICStus, Quintus. Not all versions of this library define exactly the same set of predicates, but defined predicates are compatible.

Compatibility library for term manipulation predicates. Most predicates in this library are provided as SWI-Prolog built-ins.

term_size(*@Term, -Size*) *[det]*

True if *Size* is the size in *cells* occupied by *Term* on the global (term) stack. A *cell* is 4 bytes on 32-bit machines and 8 bytes on 64-bit machines. The calculation does take *sharing* into account. For example:

```
?- A = a(1,2,3), term_size(A,S).
S = 4.
?- A = a(1,2,3), term_size(a(A,A),S).
S = 7.
?- term_size(a(a(1,2,3), a(1,2,3)), S).
S = 11.
```

Note that small objects such as atoms and small integers have a size 0. Space is allocated for floats, large integers, strings and compound terms.

variant(@*Term1*, @*Term2***) *[semidet]*

Same as SWI-Prolog `Term1 =@= Term2`.

subsumes_chk(@*Generic*, @*Specific***)

True if *Generic* can be made equivalent to *Specific* without changing *Specific*.

deprecated Replace by `subsumes_term/2`.

subsumes(+*Generic*, @*Specific***)

True if *Generic* is unified to *Specific* without changing *Specific*.

deprecated It turns out that calls to this predicate almost always should have used `subsumes_term/2`. Also the name is misleading. In case this is really needed, one is adviced to follow `subsumes_term/2` with an explicit unification.

term_subsumer(+*Special1*, +*Special2*, -*General***) *[det]*

General is the most specific term that is a generalisation of *Special1* and *Special2*. The implementation can handle cyclic terms.

author Inspired by LOGIC.PRO by Stephen Muggleton
Compatibility SICStus

term_factorized(+*Term*, -*Skeleton*, -*Substiution***)

Is true when *Skeleton* is *Term* where all subterms that appear multiple times are replaced by a variable and Substitution is a list of Var=Value that provides the subterm at the location Var. I.e., After unifying all substitutions in Substiutions, *Term* == *Skeleton*. *Term* may be cyclic. For example:

```
?- X = a(X), term_factorized(b(X,X), Y, S).
Y = b(_G255, _G255),
S = [_G255=a(_G255)].
```

mapargs(:*Goal*, ?*Term1*, ?*Term2***)

Term1 and *Term2* have the same functor (name/arity) and for each matching pair of arguments `call(Goal, A1, A2)` is true.

mapsubterms(*:Goal, +Term1, -Term2*) *[det]*
mapsubterms_var(*:Goal, +Term1, -Term2*) *[det]*

Recursively map sub terms of *Term1* into subterms of *Term2* for every pair for which `call(Goal, ST1, ST2)` succeeds. Procedurably, the mapping for each (sub) term pair `T1/T2` is defined as:

- If *T1* is a variable
  - `mapsubterms/3` unifies *T2* with *T1*.
  - `mapsubterms_var/3` treats variables as other terms.
- If `call(Goal, T1, T2)` succeeds we are done. Note that the mapping does not continue in *T2*. If this is desired, *Goal* must call `mapsubterms/3` explicitly as part of its conversion.
- If *T1* is a dict, map all values, i.e., the *tag* and *keys* are left untouched.
- If *T1* is a list, map all elements, i.e., the list structure is left untouched.
- If *T1* is a compound, use `same_functor/3` to instantiate *T2* and recurse over the term arguments left to right.
- Otherwise *T2* is unified with *T1*.

Both predicates are implemented using `foldsubterms/5`.

foldsubterms(*:Goal3, +Term1, +State0, -State*) *[semidet]*
foldsubterms(*:Goal4, +Term1, ?Term2, +State0, -State*) *[semidet]*

The predicate `foldsubterms/5` calls `call(Goal4, SubTerm1, SubTerm2, StateIn, StateOut)` for each subterm, including variables, in *Term1*. If this call fails, *StateIn* and *StateOut* are the same. This predicate may be used to map subterms in a term while collecting state about the mapped subterms. The `foldsubterms/4` variant does not map the term.

same_functor(*?Term1, ?Term2*) *[semidet]*
same_functor(*?Term1, ?Term2, -Arity*) *[semidet]*
same_functor(*?Term1, ?Term2, ?Name, ?Arity*) *[semidet]*

True when *Term1* and *Term2* are terms that have the same functor (*Name/Arity*). The arguments must be sufficiently instantiated, which means either *Term1* or *Term2* must be bound or both *Name* and *Arity* must be bound.

If *Arity* is 0, *Term1* and *Term2* are unified with *Name* for compatibility.

Compatibility SICStus

A.50 library(thread): High level thread primitives

author Jan Wielemaker

This module defines simple to use predicates for running goals concurrently. Where the core multi-threaded API is targeted at communicating long-living threads, the predicates here are defined to run goals concurrently without having to deal with thread creation and maintenance explicitely.

Note that these predicates run goals concurrently and therefore these goals need to be thread-safe. As the predicates in this module also abort branches of the computation that are no longer needed, predicates that have side-effect must act properly. In a nutshell, this has the following consequences:

- Nice clean Prolog code without side-effects (but with cut) works fine.
- Side-effects are bad news. If you really need assert to store intermediate results, use the `thread_local/1` declaration. This also guarantees cleanup of left-over clauses if the thread is cancelled. For other side-effects, make sure to use `call_cleanup/2` to undo them should the thread be cancelled.
- Global variables are ok as they are thread-local and destroyed on thread cancellation. Note however that global variables in the calling thread are **not** available in the threads that are created. You have to pass the value as an argument and initialise the variable in the new thread.
- Thread-cancellation uses `thread_signal/2`. Using this code with long-blocking foreign predicates may result in long delays, even if another thread asks for cancellation.

concurrent(*+N, :Goals, +Options*) *[semidet]*

Run *Goals* in parallel using *N* threads. This call blocks until all work has been done. The *Goals* must be independent. They should not communicate using shared variables or any form of global data. All *Goals* must be thread-safe.

Execution succeeds if all goals have succeeded. If one goal fails or throws an exception, other workers are abandoned as soon as possible and the entire computation fails or re-throws the exception. Note that if multiple goals fail or raise an error it is not defined which error or failure is reported.

On successful completion, variable bindings are returned. Note however that threads have independent stacks and therefore the goal is copied to the worker thread and the result is copied back to the caller of `concurrent/3`.

Choosing the right number of threads is not always obvious. Here are some scenarios:

- If the goals are CPU intensive and normally all succeeding, typically the number of CPUs is the optimal number of threads. Less does not use all CPUs, more wastes time in context switches and also uses more memory.
- If the tasks are I/O bound the number of threads is typically higher than the number of CPUs.
- If one or more of the goals may fail or produce an error, using a higher number of threads may find this earlier.

Arguments

N	Number of worker-threads to create. Using 1, no threads are created. If *N* is larger than the number of *Goals* we create exactly as many threads as there are *Goals*.
Goals	List of callable terms.
Options	Passed to `thread_create/3` for creating the workers. Only options changing the stack-sizes can be used. In particular, do not pass the detached or alias options.

See also In many cases, `concurrent_maplist/2` and friends is easier to program and is tractable to program analysis.

concurrent_forall(:*Generate, :Action***)** *[semidet]*
concurrent_forall(:*Generate, :Action, +Options***)** *[semidet]*

True when *Action* is true for all solutions of *Generate*. This has the same semantics as `forall/2`, but the *Action* goals are executed in multiple threads. Notable a failing *Action* or a *Action* throwing an exception signals the calling thread which in turn aborts all workers and fails or re-throws the generated error. *Options*:

threads(+*Count***)**
Number of threads to use. The default is determined by the Prolog flag `cpu_count`.

To be done Ideally we would grow the set of workers dynamically, similar to dynamic scheduling of HTTP worker threads. This would avoid creating threads that are never used if *Generate* is too slow or does not provide enough answers and would further raise the number of threads if *Action* is I/O bound rather than CPU bound.

concurrent_and(:*Generator, :Test***)**
concurrent_and(:*Generator, :Test, +Options***)**

Concurrent version of `(Generator,Test)`. This predicate creates a thread providing solutions for *Generator* that are handed to a pool of threads that run *Test* for the different instantiations provided by *Generator* concurrently. The predicate is logically equivalent to a simple conjunction except for two aspects: (1) terms are *copied* from *Generator* to the test *Test* threads while answers are copied back to the calling thread and (2) answers may be produced out of order.

If the evaluation of some *Test* raises an exception, `concurrent_and/2`,3 is terminated with this exception. If the caller commits after a given answer or raises an exception while `concurrent_and/2`,3 is active with pending choice points, all involved resources are reclaimed.

Options:

threads(+*Count***)**
Create a worker pool holding *Count* threads. The default is the Prolog flag `cpu_count`.

This predicate was proposed by Jan Burse as `balance((Generator,Test))`.

concurrent_maplist(:*Goal, +List***)** *[semidet]*
concurrent_maplist(:*Goal, +List1, +List2***)** *[semidet]*
concurrent_maplist(:*Goal, +List1, +List2, +List3***)** *[semidet]*

Concurrent version of `maplist/2`. This predicate uses `concurrent/3`, using multiple *worker* threads. The number of threads is the minimum of the list length and the number of cores available. The number of cores is determined using the prolog flag `cpu_count`. If this flag is absent or 1 or *List* has less than two elements, this predicate calls the corresponding maplist/N version using a wrapper based on `once/1`. Note that all goals are executed as if wrapped in `once/1` and therefore these predicates are *semidet*.

Note that the the overhead of this predicate is considerable and therefore *Goal* must be fairly expensive before one reaches a speedup.

first_solution(-*X, :Goals, +Options***)** *[semidet]*

Try alternative solvers concurrently, returning the first answer. In a typical scenario, solving any of the goals in *Goals* is satisfactory for the application to continue. As soon as one of the tried alternatives is successful, all the others are killed and `first_solution/3` succeeds.

For example, if it is unclear whether it is better to search a graph breadth-first or depth-first we can use:

```
search_graph(Grap, Path) :-
        first_solution(Path, [ breadth_first(Graph, Path),
                               depth_first(Graph, Path)
                             ],
                       []).
```

Options include thread stack-sizes passed to thread_create, as well as the options `on_fail` and `on_error` that specify what to do if a solver fails or triggers an error. By default execution of all solvers is terminated and the result is returned. Sometimes one may wish to continue. One such scenario is if one of the solvers may run out of resources or one of the solvers is known to be incomplete.

on_fail(*Action***)**
If `stop` (default), terminate all threads and stop with the failure. If `continue`, keep waiting.

on_error(*Action***)**
As above, re-throwing the error if an error appears.

bug `first_solution/3` cannot deal with non-determinism. There is no obvious way to fit non-determinism into it. If multiple solutions are needed wrap the solvers in `findall/3`.

call_in_thread(*+Thread, :Goal***)** *[semidet]*
Run *Goal* as an interrupt in the context of *Thread*. This is based on `thread_signal/2`. If waiting times out, we inject a `stop(Reason)` exception into *Goal*. Interrupts can be nested, i.e., it is allowed to run a `call_in_thread/2` while the target thread is processing such an interrupt.

This predicate is primarily intended for debugging and inspection tasks.

A.51 library(thread_pool): Resource bounded thread management

See also `http_handler/3` and `http_spawn/2`.

The module `library(thread_pool)` manages threads in pools. A pool defines properties of its member threads and the maximum number of threads that can coexist in the pool. The call `thread_create_in_pool/4` allocates a thread in the pool, just like `thread_create/3`. If the pool is fully allocated it can be asked to wait or raise an error.

The library has been designed to deal with server applications that receive a variety of requests, such as HTTP servers. Simply starting a thread for each request is a bit too simple minded for such servers:

- Creating many CPU intensive threads often leads to a slow-down rather than a speedup.

- Creating many memory intensive threads may exhaust resources
- Tasks that require little CPU and memory but take long waiting for external resources can run many threads.

Using this library, one can define a pool for each set of tasks with comparable characteristics and create threads in this pool. Unlike the worker-pool model, threads are not started immediately. Depending on the design, both approaches can be attractive.

The library is implemented by means of a manager thread with the fixed thread id `__thread_pool_manager`. All state is maintained in this manager thread, which receives and processes requests to create and destroy pools, create threads in a pool and handle messages from terminated threads. Thread pools are *not* saved in a saved state and must therefore be recreated using the `initialization/1` directive or otherwise during startup of the application.

thread_pool_create(+*Pool, +Size, +Options*) *[det]*

Create a pool of threads. A pool of threads is a declaration for creating threads with shared properties (stack sizes) and a limited number of threads. Threads are created using `thread_create_in_pool/4`. If all threads in the pool are in use, the behaviour depends on the `wait` option of `thread_create_in_pool/4` and the `backlog` option described below. *Options* are passed to `thread_create/3`, except for

backlog(+*MaxBackLog*)

Maximum number of requests that can be suspended. Default is `infinite`. Otherwise it must be a non-negative integer. Using `backlog(0)` will never delay thread creation for this pool.

The pooling mechanism does *not* interact with the `detached` state of a thread. Threads can be created both `detached` and normal and must be joined using `thread_join/2` if they are not detached.

thread_pool_destroy(+*Name*) *[det]*

Destroy the thread pool named *Name*.

Errors `existence_error(thread_pool, Name)`.

current_thread_pool(*?Name*) *[nondet]*

True if *Name* refers to a defined thread pool.

thread_pool_property(*?Name, ?Property*) *[nondet]*

True if *Property* is a property of thread pool *Name*. Defined properties are:

options(*Options*)

Thread creation options for this pool

free(*Size*)

Number of free slots on this pool

size(*Size*)

Total number of slots on this pool

members(*ListOfIDs*)
: *ListOfIDs* is the list or threads running in this pool

running(*Running*)
: Number of running threads in this pool

backlog(*Size*)
: Number of delayed thread creations on this pool

thread_create_in_pool(*+Pool, :Goal, -Id, +Options*) *[det]*

Create a thread in *Pool*. *Options* overrule default thread creation options associated to the pool. In addition, the following option is defined:

wait(*+Boolean*)
: If `true` (default) and the pool is full, wait until a member of the pool completes. If `false`, throw a resource_error.

Errors
- `resource_error(threads_in_pool(Pool))` is raised if wait is `false` or the backlog limit has been reached.
- `existence_error(thread_pool, Pool)` if *Pool* does not exist.

worker_exitted(*+PoolName, +WorkerId, :AtExit*)

It is possible that '__thread_pool_manager' no longer exists while closing down the process because the manager was killed before the worker.

To be done Find a way to discover that we are terminating Prolog.

create_pool(*+PoolName*) *[semidet,multifile]*

Hook to create a thread pool lazily. The hook is called if `thread_create_in_pool/4` discovers that the thread pool does not exist. If the hook succeeds, `thread_create_in_pool/4` retries creating the thread. For example, we can use the following declaration to create threads in the pool `media`, which holds a maximum of 20 threads.

```
:- multifile thread_pool:create_pool/1.

thread_pool:create_pool(media) :-
    thread_pool_create(media, 20, []).
```

A.52 library(ugraphs): Graph manipulation library

author
- R.A.O'Keefe
- Vitor Santos Costa
- Jan Wielemaker

license BSD-2 or Artistic 2.0

The S-representation of a graph is a list of (vertex-neighbours) pairs, where the pairs are in standard order (as produced by keysort) and the neighbours of each vertex are also in standard order (as produced by sort). This form is convenient for many calculations.

A new UGraph from raw data can be created using `vertices_edges_to_ugraph/3`. Adapted to support some of the functionality of the SICStus ugraphs library by Vitor Santos Costa. Ported from YAP 5.0.1 to SWI-Prolog by Jan Wielemaker.

vertices(+*Graph*, -*Vertices*)
Unify *Vertices* with all vertices appearing in *Graph*. Example:

```
?- vertices([1-[3,5],2-[4],3-[],4-[5],5-[]], L).
L = [1, 2, 3, 4, 5]
```

vertices_edges_to_ugraph(+*Vertices*, +*Edges*, -*UGraph*) *[det]*
Create a *UGraph* from *Vertices* and edges. 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-[]]
```

add_vertices(+*Graph*, +*Vertices*, -*NewGraph*)
Unify *NewGraph* with a new graph obtained by adding the list of *Vertices* to *Graph*. Example:

```
?- add_vertices([1-[3,5],2-[]], [0,1,2,9], NG).
NG = [0-[], 1-[3,5], 2-[], 9-[]]
```

del_vertices(+*Graph*, +*Vertices*, -*NewGraph*) *[det]*
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([1-[3,5],2-[4],3-[],4-[5],5-[],6-[],7-[2,6],8-[]],
                [2,1],
                NL).
NL = [3-[],4-[5],5-[],6-[],7-[6],8-[]]
```

Compatibility Upto 5.6.48 the argument order was (+*Vertices*, +*Graph*, -*NewGraph*). Both YAP and SWI-Prolog have changed the argument order for compatibility with recent SICStus as well as consistency with `del_edges/3`.

add_edges(*+Graph, +Edges, -NewGraph*)

Unify *NewGraph* with a new graph obtained by adding the list of *Edges* to *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-[]]
```

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]]
```

del_edges(*+Graph, +Edges, -NewGraph*)

Unify *NewGraph* with a new graph obtained by removing the list of *Edges* from *Graph*. Notice that no vertices are deleted. 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-[]]
```

edges(*+Graph, -Edges*)

Unify *Edges* with all edges appearing in *Graph*. Example:

```
?- edges([1-[3,5],2-[4],3-[],4-[5],5-[]], L).
L = [1-3, 1-5, 2-4, 4-5]
```

transitive_closure(*+Graph, -Closure*)

Generate the graph *Closure* as the transitive closure of *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]]
```

transpose_ugraph(*Graph, NewGraph*) *[det]*

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|*log(|V|)). 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-[]]
```

Compatibility This predicate used to be known as `transpose/2`. Following SICStus 4, we reserve `transpose/2` for matrix transposition and renamed ugraph transposition to `transpose_ugraph/2`.

compose(*+LeftGraph, +RightGraph, -NewGraph*)
Compose *NewGraph* 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-[]]
```

top_sort(*+Graph, -Sorted*) *[semidet]*
top_sort(*+Graph, -Sorted, ?Tail*) *[semidet]*
Sorted is a topological sorted list of nodes in *Graph*. 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]
```

The predicate `top_sort/3` is a difference list version of `top_sort/2`.

neighbors(*+Vertex, +Graph, -Neigbours*) *[det]*
neighbours(*+Vertex, +Graph, -Neigbours*) *[det]*
Neigbours is a sorted list of the neighbours of *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]
```

connect_ugraph(*+UGraphIn, -Start, -UGraphOut*) *[det]*
Adds *Start* as an additional vertex that is connected to all vertices in *UGraphIn*. This can be used to create an topological sort for a not connected graph. *Start* is before any vertex in *UGraphIn* in the standard order of terms. No vertex in *UGraphIn* can be a variable.

Can be used to order a not-connected graph as follows:

```
top_sort_unconnected(Graph, Vertices) :-
    (   top_sort(Graph, Vertices)
    ->  true
    ;   connect_ugraph(Graph, Start, Connected),
        top_sort(Connected, Ordered0),
        Ordered0 = [Start|Vertices]
    ).
```

complement(*+UGraphIn, -UGraphOut*)
UGraphOut is a ugraph with an edge between all vertices that are *not* connected in *UGraphIn* and all edges from *UGraphIn* removed. 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]]
```

To be done Simple two-step algorithm. You could be smarter, I suppose.

reachable(+*Vertex*, +*UGraph*, -*Vertices***)**
True when *Vertices* is an ordered set of vertices reachable in *UGraph*, including *Vertex*. Example:

```
?- reachable(1,[1-[3,5],2-[4],3-[],4-[5],5-[]],V).
V = [1, 3, 5]
```

A.53 library(url): Analysing and constructing URL

author
- Jan Wielemaker
- Lukas Faulstich

deprecated New code should use `library(uri)`, provided by the `clib` package.

This library deals with the analysis and construction of a URL, Universal Resource Locator. 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.

global_url(+*URL*, +*Base*, -*Global***)** *[det]*
Translate a possibly relative *URL* into an absolute one.

Errors `syntax_error(illegal_url)` if *URL* is not legal.

is_absolute_url(+*URL***)**
True if *URL* is an absolute *URL*. That is, a *URL* that starts with a protocol identifier.

http_location(?*Parts*, ?*Location***)**
Construct or analyze an HTTP location. This is 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/<version>
```

Arguments	
Location	Atom or list of character codes.

parse_url(*?URL, ?Attributes*) *[det]*

Construct or analyse a *URL*. *URL* is an atom holding a *URL* or a variable. *Attributes* 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 \arg{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 HTTP GET method. 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 all of this for an HTTP *URL*.

```
?- parse_url('http://www.xyz.org/hello?msg=Hello+World%21#x',
       P).

P = [ protocol(http),
      host('www.xyz.org'),
      fragment(x),
      search([ msg = 'Hello World!'
             ]),
      path('/hello')
    ]
```

By instantiating the parts-list this predicate can be used to create a *URL*.

parse_url(*+URL, +BaseURL, -Attributes*) *[det]*

Similar to `parse_url/2` for relative URLs. If *URL* is relative, it is resolved using the absolute *URL BaseURL*.

www_form_encode(*+Value, -XWWWFormEncoded*) *[det]*
www_form_encode(*-Value, +XWWWFormEncoded*) *[det]*

En/decode to/from application/x-www-form-encoded. Encoding encodes all characters except RFC 3986 *unreserved* (ASCII `alnum` (see `code_type/2`)), and one of "-._~" using percent encoding. Newline is mapped to `%OD%OA`. When decoding, newlines appear as a single newline (10) character.

Note that a space is encoded as `%20` instead of `+`. Decoding decodes both to a space.

deprecated Use `uri_encoded/3` for new code.

set_url_encoding(*?Old, +New*) *[semidet]*

Query and set the encoding for URLs. The default is `utf8`. The only other defined value is `iso_latin_1`.

To be done Having a global flag is highly inconvenient, but a work-around for old sites using ISO Latin 1 encoding.

url_iri(*+Encoded, -Decoded*) *[det]*
url_iri(*-Encoded, +Decoded*) *[det]*

Convert between a URL, encoding in US-ASCII and an IRI. An IRI is a fully expanded Unicode string. Unicode strings are first encoded into UTF-8, after which %-encoding takes place.

parse_url_search(*?Spec, ?Fields:list(Name=Value)*) *[det]*

Construct or analyze an HTTP search specification. This deals with form data using the MIME-type `application/x-www-form-urlencoded` as used in HTTP GET requests.

file_name_to_url(*+File, -URL*) *[det]*
file_name_to_url(*-File, +URL*) *[semidet]*

Translate between a filename and a file:`//` *URL*.

To be done Current implementation does not deal with paths that need special encoding.

A.54 library(varnumbers): Utilities for numbered terms

See also `numbervars/4`, `=@=/2` (`variant/2`).

Compatibility This library was introduced by Quintus and available in many related implementations, although not with exactly the same set of predicates.

This library provides the inverse functionality of the built-in `numbervars/3`. Note that this library suffers from the known issues that '$VAR'(X) is a normal Prolog term and, -unlike the built-in numbervars-, the inverse predicates do *not* process cyclic terms. The following predicate is true for any acyclic term that contains no '$VAR'(X), `integer(X)` terms and no constraint variables:

```
always_true(X) :-
      copy_term(X, X2),
      numbervars(X),
      varnumbers(X, Copy),
      Copy =@= X2.
```

numbervars(+*Term*) *[det]*
Number variables in *Term* using $VAR(N). Equivalent to `numbervars(Term, 0, _)`.

See also `numbervars/3`, `numbervars/4`

varnumbers(+*Term, -Copy*) *[det]*
Inverse of `numbervars/1`. Equivalent to `varnumbers(Term, 0, Copy)`.

varnumbers(+*Term, +Start, -Copy*) *[det]*
Inverse of `numbervars/3`. True when *Copy* is a copy of *Term* with all variables numbered >= *Start* consistently replaced by fresh variables. Variables in *Term* are *shared* with *Copy* rather than replaced by fresh variables.

Errors `domain_error(acyclic_term, Term)` if *Term* is cyclic.
Compatibility Quintus, SICStus. Not in YAP version of this library

max_var_number(+*Term, +Start, -Max*) *[det]*
True when *Max* is the max of *Start* and the highest numbered $VAR(N) term.

author Vitor Santos Costa
Compatibility YAP

varnumbers_names(+*Term, -Copy, -VariableNames*) *[det]*
If *Term* is a term with numbered and named variables using the reserved term '$VAR'(X), *Copy* is a copy of *Term* where each '$VAR'(X) is consistently replaced by a fresh variable and Bindings is a list `X = Var`, relating the *X* terms with the variable it is mapped to.

See also `numbervars/3`, `varnumbers/3`, `read_term/3` using the `variable_names` option.

A.55 library(yall): Lambda expressions

author Paulo Moura and Jan Wielemaker
To be done Extend optimization support

Prolog realizes *high-order* programming with meta-calling. The core predicate of this is `call/1`, which simply calls its argument. This can be used to define higher-order predicates such as `ignore/1` or `forall/2`. The call/N construct calls a *closure* with N-1 *additional arguments*. This is used to define higher-order predicates such as the `maplist/2`-5 family or `foldl/4`-7.

The *closure* concept used here is somewhat different from the closure concept from functional programming. The latter is a function that is always evaluated in the context that existed at function creation time. Here, a closure is a term of arity *0 =< L =< K*. The term's functor is the name of a predicate of arity *K* and the term's *L* arguments (where *L* could be 0) correspond to *L* leftmost arguments of said predicate, bound to parameter values. For example, a closure involving `atom_concat/3` might be the term `atom_concat(prefix)`. In order of increasing *L*, one would have increasingly more complete closures that could be passed to `call/3`, all giving the same result:

```
call(atom_concat,prefix,suffix,R).
call(atom_concat(prefix),suffix,R).
call(atom_concat(prefix,suffix),R).
call(atom_concat(prefix,suffix,R)).
```

The problem with higher order predicates based on call/N is that the additional arguments are always added to the end of the closure's argument list. This often requires defining trivial helper predicates to get the argument order right. For example, if you want to add a common postfix to a list of atoms you need to apply `atom_concat(In,Postfix,Out)`, but `maplist(atom_concat(Postfix),ListIn,ListOut)` calls `atom_concat(Postfix,In,Out)`. This is where `library(yall)` comes in, where the module name, *yall*, stands for *Yet Another Lambda Library*.

The library allows us to write a lambda expression that *wraps around* the (possibly complex) goal to call:

```
?- maplist([In,Out]>>atom_concat(In,'_p',Out), [a,b], ListOut).
ListOut = [a_p, b_p].
```

A bracy list `{...}` specifies which variables are *shared* between the wrapped goal and the surrounding context. This allows us to write the code below. Without the `{Postfix}` a fresh variable would be passed to `atom_concat/3`.

```
add_postfix(Postfix, ListIn, ListOut) :-
    maplist({Postfix}/[In,Out]>>atom_concat(In,Postfix,Out),
            ListIn, ListOut).
```

This introduces the second application area of lambda expressions: the ability to confine variables to the called goal's context. This features shines when combined with `bagof/3` or `setof/3` where one normally has to list those variables whose bindings one is *not* interested in using the `Var^Goal` construct (marking *Var* as existentially quantified and confining it to the called goal's context). Lambda expressions allow you to do the converse: specify the variables which one *is* interested in. These variables are common to the context of the called goal and the surrounding context.

Lambda expressions use the syntax below

```
{...}/[...]>>Goal.
```

The `{...}` optional part is used for lambda-free variables (the ones shared between contexts). The order of variables doesn't matter, hence the `{...}` set notation.

The `[...]` optional part lists lambda parameters. Here, order of variables matters, hence the list notation.

As `/` and `>>` are standard infix operators, no new operators are added by this library. An advantage of this syntax is that we can simply unify a lambda expression with `{Free}/[Parameters]>>Lambda` to access each of its components. Spaces in the lambda expression are not a problem although the goal may need to be written between '()'s. Goals that are qualified by a module prefix also need to be wrapped inside parentheses.

Combined with `library(apply_macros)`, `library(yall)` allows writing one-liners for many list operations that have the same performance as hand-written code.

This module implements Logtalk's lambda expressions syntax.

The development of this module was sponsored by Kyndi, Inc.

+Parameters **>>** *+Lambda*
>>(*+Parameters, +Lambda, ?A1*)
>>(*+Parameters, +Lambda, ?A1, ?A2*)
>>(*+Parameters, +Lambda, ?A1, ?A2, ?A3*)
>>(*+Parameters, +Lambda, ?A1, ?A2, ?A3, ?A4*)
>>(*+Parameters, +Lambda, ?A1, ?A2, ?A3, ?A4, ?A5*)
>>(*+Parameters, +Lambda, ?A1, ?A2, ?A3, ?A4, ?A5, ?A6*)
>>(*+Parameters, +Lambda, ?A1, ?A2, ?A3, ?A4, ?A5, ?A6, ?A7*)

Calls a copy of *Lambda*. This is similar to `call(Lambda,A1,...)`, but arguments are reordered according to the list *Parameters*:

- The first `length(Parameters)` arguments from *A1*, ... are unified with (a copy of) *Parameters*, which *may* share them with variables in *Lambda*.
- Possible excess arguments are passed by position.

Arguments

Parameters	is either a plain list of parameters or a term `{Free}/List`. *Free* represents variables that are shared between the context and the *Lambda* term. This is needed for compiling *Lambda* expressions.

+Free **/** *:Lambda*
/(*+Free, :Lambda, ?A1*)
/(*+Free, :Lambda, ?A1, ?A2*)
/(*+Free, :Lambda, ?A1, ?A2, ?A3*)
/(*+Free, :Lambda, ?A1, ?A2, ?A3, ?A4*)
/(*+Free, :Lambda, ?A1, ?A2, ?A3, ?A4, ?A5*)
/(*+Free, :Lambda, ?A1, ?A2, ?A3, ?A4, ?A5, ?A6*)
/(*+Free, :Lambda, ?A1, ?A2, ?A3, ?A4, ?A5, ?A6, ?A7*)

Shorthand for `Free/[]>>Lambda`. This is the same as applying call/N on *Lambda*, except that only variables appearing in *Free* are bound by the call. For example

```
p(1,a).
p(2,b).
```

```
?- {X}/p(X,Y).
X = 1;
X = 2.
```

This can in particularly be combined with `bagof/3` and `setof/3` to *select* particular variables to be concerned rather than using existential quantification (`^/2`) to *exclude* variables. For example, the two calls below are equivalent.

```
setof(X, Y^p(X,Y), Xs)
setof(X, {X}/p(X,_), Xs)
```

is_lambda(@*Term***)** *[semidet]*

True if *Term* is a valid Lambda expression.

lambda_calls(+*LambdaExpression, -Goal***)** *[det]*
lambda_calls(+*LambdaExpression, +ExtraArgs, -Goal***)** *[det]*

Goal is the goal called if call/N is applied to *LambdaExpression*, where *ExtraArgs* are the additional arguments to call/N. *ExtraArgs* can be an integer or a list of concrete arguments. This predicate is used for cross-referencing and code highlighting.

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*) *[det]*

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_current_choice(*-Choice*) *[semidet]*

Unify *Choice* with an integer provided a reference to the last choice point. Fails if the current environment has no choice points. See also `prolog_choice_attribute/3`.

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 the calling context, 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]

[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.

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.

predicate_indicator
: Similar to `goal`, but only returning the [⟨*module*⟩:]⟨*name*⟩/⟨*arity*⟩ term describing the term, not the actual arguments. It avoids creating an illegal term as `goal` and is used by the library `prolog_stack`.

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.

skipped
: *Value* is `true` if this frame was skipped in the debugger.

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 or belongs to a compound meta-call (e.g., call((a,b))).

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 3rd argument or obtained using `prolog_current_choice/1`. *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).

pc
Requests the program counter to which the choice point refers. Only applicable for in-clause choice points.

clause
Request the clause that will be tried if this choice point is activated. Only applicable for choice points of type `clause`.

This predicate is used for the graphical debugger to show the choice point stack.

deterministic(*-Boolean*)
Unifies its argument with `true` if no choice point exists that is more recent than the entry of the clause in which it appears. There are few realistic situations for using this predicate. It is used by the `prolog/0` top level to check whether Prolog should prompt the user for alternatives. Similar results can be achieved in a more portable fashion using `call_cleanup/2`.

B.2 Ancestral cuts

prolog_cut_to(*+Choice*)
Prunes all choice points created since *Choice*. Can be used together with `prolog_current_choice/1` to implement *ancestral* cuts. This predicate is in the hackers corner because it should not be used in normal Prolog code. It may be used to create new high level control structures, particularly for compatibility purposes.

Note that in the current implementation, the pruned choice points and environment frames are *not* reclaimed. As a consequence, where predicates that are deterministic due to clause indexing, normal cuts or `(if->then;else)` and and tail recursive run in bounded local stack space, predicates using `prolog_cut_to/1` will run out of stack.

B.3 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 that 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(*PC*)
: Normal entry through the redo port of the 4-port debugger. The `redo` port signals resuming a predicate to generate alternative solutions. If *PC* is 0 (zero), clause indexing has found another clause that will be tried next. Otherwise, *PC* is the program counter in the current clause where execution continues. This implies we are dealing with an in-clause choice point left by, e.g., `;/2`. Note that non-determinism in foreign predicates are also handled using an in-clause choice point.

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 alternatives. 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* must be unified with a term that specifies how execution must continue. The following actions are defined:

abort
: Abort execution. See `abort/0`.

continue
: Continue (i.e., *creep* in the command line debugger).

fail
: Make the current goal fail.

ignore
: Step over the current goal without executing it.

nodebug
: Continue execution in normal nodebugging mode. See `nodebug/0`.

retry
Retry the current frame.

retry(*Frame*)
Retry the given frame. This must be a parent of the current frame.

skip
Skip over the current goal (i.e., *skip* in the command line debugger).

up
Skip to the parent goal (i.e., *up* in the command line debugger).

Together with the predicates described in section 4.39 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 to 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_frame(*-Frame*)
Indicate *Frame* as a skipped frame and set the 'skip level' (see `prolog_skip_level/2` to the recursion depth of *Frame*. The effect of the skipped flag is that a redo on a child of this frame is handled differently. First, a `redo` trace is called for the child, where the skip level is set to `redo_in_skip`. Next, the skip level is set to skip level of the skipped frame.

prolog_skip_level(*-Old, +New*)
Unify *Old* with the old value of 'skip level' and then set this level according to *New*. *New* is an integer, the atom `very_deep` (meaning don't skip) or the atom `skip_in_redo` (see `prolog_skip_frame/1`). The 'skip level' is a setting of each Prolog thread that disables the debugger on all recursion levels deeper than the level of the variable. See also `prolog_skip_frame/1`.

B.4 Breakpoint and watchpoint handling

SWI-Prolog support *breakpoints*. Breakpoints can be manipulated with the library `prolog_breakpoints`. Setting a breakpoint replaces a virtual machine instruction with the `D_BREAK` instruction. If the virtual machine executes a `D_BREAK`, it performs a callback to decide on the action to perform. This section describes this callback, called `prolog:break_hook/6`.

prolog:break_hook(*+Clause, +PC, +FR, +BFR, +Expression, -Action*) *[hook,semidet]*
Experimental This hook is called if the virtual machine executes a `D_BREAK`, set using

set_breakpoint/4. *Clause* and *PC* identify the breakpoint. *FR* and *BFR* provide the environment frame and current choicepoint. *Expression* identifies the action that is interrupted, and is one of the following:

call(*Goal*)
: The instruction will call *Goal*. This is generated for nearly all instructions. Note that *Goal* is semantically equivalent to the compiled body term, but might differ syntactically. This is notably the case when arithmetic expressions are compiled in optimized mode (see `optimise`). In particular, the arguments of arithmetic expressions have already been evaluated. Thus, *A* is 3**B*, where *B* equals 3 results in a term `call(A is 9)` if the clause was compiled with optimization enabled.

!
: The instruction will call the cut. Because the semantics of metacalling the cut differs from executing the cut in its original context we do not wrap the cut in `call/1`.

:-
: The breakpoint is on the *neck* instruction, i.e., after performing the head unifications.

exit
: The breakpoint is on the *exit* instruction, i.e., at the end of the clause. Note that the exit instruction may not be reached due to last-call optimisation.

unify_exit
: The breakpoint is on the completion of an in-lined unification while the system is not in debug mode. If the system is in debug mode, inlined unification is returned as call(Var=Term).[2]

If `prolog:break_hook/6` succeeds, it must unify *Action* with a value that describes how execution must continue. Possible values for *Action* are:

continue
: Just continue as if no breakpoint was present.

debug
: Continue in *debug mode*. See `debug/0`.

trace
: Continue in *trace mode*. See `trace/0`.

call(*Goal*)
: Execute *Goal* instead of the goal that would be executed. *Goal* is executed as `call/1`, preserving (non-)determinism and exceptions.

If this hook throws an exception, the exception is propagated normally. If this hook is not defined or fails, the default action is executed. This implies that, if the thread is in debug mode, the tracer will be enabled (`trace`) and otherwise the breakpoint is ignored (`continue`).

This hook allows for injecting various debugging scenarios into the executable without recompiling. The hook can access variables of the calling context using the frame inspection predicates. Here are some examples.

[2] This hack will disappear if we find a good solution for applying D_BREAK to inlined unification. Only option might be to place the break on both the unification start and end instructions.

- Create *conditional* breakpoints by imposing conditions before deciding the return `trace`.
- Watch variables at a specific point in the execution. Note that binding of these variables can be monitored using *attributed variables*, see section 8.1.
- Dynamically add *assertions* on variables using `assertion/1`.
- Wrap the *Goal* into a meta-call that traces progress of the *Goal*.

B.5 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 built-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`, `none` if the exception is not caught or `'C'` if the exception is caught in C calling Prolog using the flag `PL_Q_CATCH_EXCEPTION`.

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 a 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. For example, imagine an application performing an unwanted division by zero while all other errors are expected and handled. We can force the debugger using the hook definition below. Run the program in debug mode (see `debug/0`) to preserve as much as possible of the error context.

```
user:prolog_exception_hook(
          error(evaluation_error(zero_divisor), _),
          _, _, _) :-
        trace, fail.
```

B.6 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. 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 predicate indicator ([module]:⟨*name*⟩/⟨*arity*⟩). If the predicate fails, Prolog will generate an `existence_error` exception. The hook is intended to implement alternatives to the built-in autoloader, such as autoloading code from a database. Do *not* use this hook to suppress existence errors on predicates. See also `unknown` and section 2.14.

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`. See also `nb_current/2`.

B.7 Prolog events

Version 8.1.9 introduces a uniform mechanism to listen to events that happen in the Prolog engine. It replaces and generalises `prolog_event_hook/1`, a hook that was introduced to support the graphical debugger. The current implementation deals with debug, thread and dynamic database events. We expect this mechanism to deal with more hooks in the future.

prolog_listen(+*Channel, :Closure*)
prolog_listen(+*Channel, :Closure, +Options*)

Call *Closure* if an event that matches *Channel* happens inside Prolog. Possible choice points are pruned as by `once/1`. Possible failure is ignored, but exceptions are propagated into the environment. Multiple closures can be associated with the same channel. Execution of the list of closures may be terminated by an exception. Options:

as(+*Location*)

Location is one of `first` (default) or `last` and determines whether the new handler is expected as first or last.

name(+*Atom*)
Give the handler a name. A new registration using the same name replaces the existing handler rather than adding a new handler. Names are local to the *Channel*, i.e., different channels can use the same name.

Defined channels are described below. The *Channel* argument is the name of the term listed below. The arguments are added as additional arguments to the given *Closure*.

abort
Called by `abort/0`.

erase(*DbRef*)
Called on an erased recorded database reference or clause. Note that a retracted clauses is not immediately removed. Clauses are reclaimed by `garbage_collect_clauses/0`, which is normally executed automatially in the `gc` thread. This specific channel is used by `clause_info/5` to reclaim source layout of reclaimed clauses. User applications should typically use the *PredicateIndicator* channel.

break(*Action, ClauseRef, PCOffset*)
Traps events related to Prolog break points. See library `prolog_breakpoints`

frame_finished(*FrameRef*)
Indicates that a stack frame that has been examined using `prolog_current_frame/1`, `prolog_frame_attribute/3` and friends has been deleted. Used by the source level debugger to avoid that the stack view references non-existing frames.

thread_exit(*Thread*)
Globally registered channel that is called by any thread just before the thread is terminated.

thread_start(*Thread*)
Globally registered channel that is called by any thread after the thread initialization and before running the thread's goal.

this_thread_exit
Thread local version of the `thread_exit` channel that is also used by the `at_exit`(*Closure*) option of `thread_create/3`.

PredicateIndicator(*Action, Context*)
Track changes to a predicate. This notably allows tracking modifications to dynamic predicates. The channel also allows tracking changes to *monotonic* tables (section 7.8). Both monotonic and incremental tabling use this to track changes to `incremental` and `monotonic` dynamic predicates. Below is an example illustrating events from changing a dynamic predicate.

```
:- dynamic p/1.
:- prolog_listen(p/1, updated(p/1)).

updated(Pred, Action, Context) :-
    format('Updated ~p: ~p ~p~n', [Pred, Action, Context]).
```

```
?- assert(p(a)).
Updated p/1: assertz <clause>(0x55db261709d0)
?- retractall(p(_)).
```

```
Updated p/1: retractall start(user:p(_12294))
Updated p/1: retract <clause>(0x55db261719c0)
Updated p/1: retractall end(user:p(_12294))
```

asserta

assertz
A new clauses has been added as first (last) for the given predicate. *Context* is a clause reference. The hook is called after the clause has been added. If the hook fails the clause is removed.

retract
A clause was retracted from the given predicate using either `retract/1`, `erase/1` or `retractall/1`. *Context* is a clause reference. The hook is called before the clause is removed. If the hook fails, the clause is not removed.

retractall
The begining and end of `retractall/1` is indicated with the *Action* `retractall`. The context argument is `start`(*Head*) or `end`(*Head*).

rollback(*Action*)
Issued when rolling back (discarding) a transaction. *Action* is the local action being reverted and is one of `asserta`, `assertz` or `retract`. Context is the involved clause. See `transaction/1` and `snapshot/1`.

new_answer
A new answer was added to a tabled predicate. The context is the answer term. Currently implemented for *monotonic* tabling only. Future versions may also implement this for normal tabling. See section 7.8.2.

prolog_unlisten(+*Channel, :Closure*)
Remove matching closures registered with `prolog_listen/3`.

B.8 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 A.21 for editing hooks and section 4.11 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 the debugger. For example, XPCE uses these hooks to allow for spying methods rather than 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 `nospy/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 succeeds, 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.9 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`.

This can be used to load from unusual places as well as dealing with Prolog code that is not represented as a Prolog source text (for example some binary representation). For example, library `http/http_load` loads Prolog directly from an HTTP server. See also `prolog:open_source_hook/3`, which merely allows for changing how a physical file is opened.

prolog:open_source_hook(*+Path, -Stream, +Options*)
This hooks is called by the compiler to overrule the default `open/3` call `open`(*Path, read, Stream*). *Options* provide the options as provided to `load_files/2`. If the hook succeeds compilation continues by loading from the returned (input) stream. This hook is particularly suited to support running the code to a preprocessor. See also `prolog_load_file/2`.

prolog:comment_hook(+*Comments, +Pos, +Term*)

This hook allows for processing comments encountered by the compiler. If this hook is defined, the compiler calls `read_term/2` with the option `comments`(*Comments*). If the list of comments returned by `read_term/2` is not empty it calls this comment hook with the following arguments.

- *Comments* is the non-empty list of comments. Each comment is a pair *Position-String*, where *String* is a string object (see section 5.2) that contains the comment *including* delimiters. Consecutive line comments are returned as a single comment.
- *Pos* is a stream-position term that describes the starting position of *Term*
- *Term* is the term read.

This hook is exploited by the documentation system. See `stream_position_data/3`. See also `read_term/3`.

C Compatibility with other Prolog dialects

This chapter explains issues for writing portable Prolog programs. It was started after discussion with Vitor Santos Costa, the leading developer of YAP Prolog[1] YAP and SWI-Prolog have expressed the ambition to enhance the portability beyond the trivial Prolog examples, including complex libraries involving foreign code.

Although it is our aim to enhance compatibility, we are still faced with many incompatibilities between the dialects. As a first step both YAP and SWI will provide some instruments that help developing portable code. A first release of these tools appeared in SWI-Prolog 5.6.43. Some of the facilities are implemented in the base system, others in the library `dialect.pl`.

- The Prolog flag `dialect` is an unambiguous and fast way to find out which Prolog dialect executes your program. It has the value `swi` for SWI-Prolog and `yap` on YAP.
- The Prolog flag `version_data` is bound to a term `swi`(*Major, Minor, Patch, Extra*)
- Conditional compilation using `:- if(Condition) ...:- endif` is supported. See section 4.3.1.
- The predicate `expects_dialect/1` allows for specifying for which Prolog system the code was written.
- The predicates `exists_source/1` and `source_exports/2` can be used to query the library content. The `require/1` directive can be used to get access to predicates without knowing their location.
- The module predicates `use_module/1`, `use_module/2` have been extended with a notion for 'import-except' and 'import-as'. This is particularly useful together with `reexport/1` and `reexport/2` to compose modules from other modules and mapping names.
- Foreign code can expect `__SWI_PROLOG__` when compiled for SWI-Prolog and `__YAP_PROLOG__` when compiled on YAP.

:- expects_dialect(+*Dialect*)

This directive states that the code following the directive is written for the given Prolog *Dialect*. See also `dialect`. The declaration holds until the end of the file in which it appears. The current dialect is available using `prolog_load_context/2`.

The exact behaviour of this predicate is still subject to discussion. Of course, if *Dialect* matches the running dialect the directive has no effect. Otherwise we check for the existence of `library`(*dialect/Dialect*) and load it if the file is found. Currently, this file has this functionality:

[1] http://yap.sourceforge.net/

- Define system predicates of the requested dialect we do not have.
- Apply `goal_expansion/2` rules that map conflicting predicates to versions emulating the requested dialect. These expansion rules reside in the dialect compatibility module, but are applied if prolog_load_context(dialect, Dialect) is active.
- Modify the search path for library directories, putting libraries compatible with the target dialect before the native libraries.
- Setup support for the default filename extension of the dialect.

source_exports(+*Spec*, +*Export***)**
Is true if source *Spec* exports *Export*, a predicate indicator. Fails without error otherwise.

C.1 Some considerations for writing portable code

The traditional way to write portable code is to define custom predicates for all potentially non-portable code and define these separately for all Prolog dialects one wishes to support. Here are some considerations.

- Probably the best reason for this is that it allows to define minimal semantics required by the application for the portability predicates. Such functionality can often be mapped efficiently to the target dialect. Contrary, if code was written for dialect X, the defined semantics are those of dialect X. Emulating all extreme cases and full error handling compatibility may be tedious and result in a much slower implementation than needed. Take for example `call_cleanup/2`. The SICStus definition is fundamentally different from the SWI definition, but 99% of the applications just want to make calls like below to guarantee *StreamIn* is closed, even if `process/1` misbehaves.

```
        call_cleanup(process(StreamIn), close(In))
```

- As a drawback, the code becomes full of *my_call_cleanup*, etc. and every potential portability conflict needs to be abstracted. It is hard for people who have to maintain such code later to grasp the exact semantics of the *my_** predicates and applications that combine multiple libraries using this compatibility approach are likely to encounter conflicts between the portability layers. A good start is not to use *my_**, but a prefix derived from the library or application name or names that explain the intended semantics more precisely.

- Another problem is that most code is initially not written with portability in mind. Instead, ports are requested by users or arise from the desire to switch Prolog dialect. Typically, we want to achieve compatibility with the new Prolog dialect with minimal changes, often keeping compatibility with the original dialect(s). This problem is well known from the C/Unix world and we advise anyone to study the philosophy of GNU autoconf, from which we will illustrate some highlights below.

The GNU autoconf suite, known to most people as `configure`, was an answer to the frustrating life of Unix/C programmers when Unix dialects were about as abundant and poorly standardised as Prolog dialects today. Writing a portable C program can only be achieved using cpp, the C preprocessor. The C preprocessor performs two tasks: macro expansion and conditional compilation. Prolog realises macro expansion through `term_expansion/2` and `goal_expansion/2`. Conditional compilation is achieved using `:- if(Condition)` as explained in section 4.3.1. The situation appears similar.

The important lesson learned from GNU autoconf is that the *last* resort for conditional compilation to achieve portability is to switch on the platform or dialect. Instead, GNU autoconf allows you to write tests for specific properties of the platform. Most of these are whether or not some function or file is available. Then there are some standard tests for difficult-to-write-portable situations and finally there is a framework that allows you to write arbitrary C programs and check whether they can be compiled and/or whether they show the intended behaviour. Using a separate `configure` program is needed in C, as you cannot perform C compilation step or run C programs from the C preprocessor. In most Prolog environments we do not need this distinction as the compiler is integrated into the runtime environment and Prolog has excellent reflexion capabilities.

We must learn from the distinction to test for features instead of platform (dialect), as this makes the platform-specific code robust for future changes of the dialect. Suppose we need `compare/3` as defined in this manual. The `compare/3` predicate is not part of the ISO standard, but many systems support it and it is not unlikely it will become ISO standard or the intended dialect will start supporting it. GNU autoconf strongly advises to test for the availability:

```
:- if(\+current_predicate(_, compare(_,_,_))).
compare(<, Term1, Term2) :-
        Term1 @< Term2, !.
compare(>, Term1, Term2) :-
        Term1 @> Term2, !.
compare(=, Term1, Term2) :-
        Term1 == Term2.
:- endif.
```

This code is **much** more robust against changes to the intended dialect and, possibly at least as important, will provide compatibility with dialects you didn't even consider porting to right now.

In a more challenging case, the target Prolog has `compare/3`, but the semantics are different. What to do? One option is to write a `my_compare/3` and change all occurrences in the code. Alternatively you can rename calls using `goal_expansion/2` like below. This construct will not only deal with Prolog dialects lacking `compare/3` as well as those that only implement it for numeric comparison or have changed the argument order. Of course, writing rock-solid code would require a complete test-suite, but this example will probably cover all Prolog dialects that allow for conditional compilation, have core ISO facilities and provide `goal_expansion/2`, the things we claim a Prolog dialect should have to start writing portable code for it.

```
:- if(\+catch(compare(<,a,b), _, fail)).
compare_standard_order(<, Term1, Term2) :-
        Term1 @< Term2, !.
compare_standard_order(>, Term1, Term2) :-
        Term1 @> Term2, !.
compare_standard_order(=, Term1, Term2) :-
        Term1 == Term2.

goal_expansion(compare(Order, Term1, Term2),
```

```
                    compare_standard_order(Order, Term1, Term2)).
:- endif.
```

C.2 Notes on specific dialects

The level of maturity of the various dialect emulation implementations varies enormously. All of them have been developed to realise portability for one or more, often large, programs. This section provides some notes on emulating a particular dialect.

C.2.1 Notes on specific dialects

XSB Prolog compatibility emerged from a project to integrate XSB's advanced tabling support in SWI-Prolog (see section 7). This project has been made possible by Kyndi.[2] The XSB dialect implementation has been created to share as much as possible of the XSB test suite as well as some larger programs to evaluate both tabling implementations. The dialect emulation was extended to support Pharos.[3].

Emulating XSB is relatively complicated due to the large distance from the Quintus descendant Prolog systems. Notably XSB's name based module system is hard to map on SWI-Prolog's predicate based module system. As a result, only non-modular projects or projects with basic usage of modules are supported. For the development of new projects that require modules more advanced module support we suggest using Logtalk.

Loading XSB source files

SWI-Prolog's emulation of XSB depends on the XSB preferred file name extension `.P`. This extension is used by `dialect/xsb/source` to initiate a two phase loading process based on `term_expansion/2` of the virtual term `begin_of_file`.

1. In the first phase the file is read with XSB compatible operator declarations and all directives (:- Term) are extracted. The directives are used to determine that the file defines a module (iff the file contains an `export/1` directive) and construct a SWI-Prolog compatible module declaration. As XSB has a two phase compiler where SWI has a single phase compiler, this is also used to move some directives to the start of the file.

2. The second phase loads the file as normal.

To load a project in both XSB and SWI-Prolog it is advised to make sure all source files use the `.P` file name extension. Next, write a SWI-Prolog loader in a `.pl` file that contains e.g.,

```
:- use_module(library(dialect/xsb/source)).

:- [main_file].
```

[2]This project was initiated by Benjamin Grosof and carried out in cooperation with Theresa Swift, David S. Warren and Fabrizio Riguzzi.

[3]Pharos was used to evaluate *incremental tabling* (section 7.7), a protect with Edward Schwatz and Cory Cohen from CMU

It is also possible to put the able `use_module/1` directive in your personal initialization file (see section 2.2), after which XSB files can be loaded as normal SWI-Prolog files using

```
% swipl file.P
```

XSB code may depend on the `gpp` preprocessor. We do not provide `gpp`. It is however possible to send XSB source files through `gpp` by loading `library/dialect/xsb/gpp`. This require `gpp` to be accessible through the environment variable `PATH` or the `file_search_path/2` alias `path`. We refer to the `gpp` library for details.

C.2.2 The XSB import directive

The XSB import directive takes the form as below.

```
:- import p/1, q/2, ... from <lib>.
```

This import directive is resolved as follows:

- If the referenced library is found as a local file, it is loaded and the requested predicates are imported.
- Otherwise, the referenced library is searched for in the `dialect/xsb` directory of the SWI-Prolog library. If found, the predicates are imported from this library.
- The referenced predicates are searched for in SWI-Prolog built-in predicates and the SWI-Prolog library. If found, they are made available if necessary.

D Glossary of Terms

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 (= 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 `asserta/1` and `assertz/1`.

atom
: Textual constant. Used as name for *compound* terms, to represent constants or text.

backtracking
: Search 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 (`:-`).

choice point
: A *choice point* represents a choice in the search for a *solution*. Choice points are created if multiple clauses match a *query* or using disjunction (`;/2`). On *backtracking*, the execution state of the most recent *choice point* is restored and search continues with the next alternative (i.e., next clause or second branch of `;/2`).

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 as "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` and *meta-predicate*.

dcg
: Abbreviation for *Definite Clause Grammar*.

det [determinism]
: Short for *deterministic*.

determinism
: How many solutions a *goal* can provide. Values are 'nondet' (zero to infinite), 'multi' (one to infinite), 'det' (exactly one) and 'semidet' (zero or one).

deterministic
: A *predicate* is *deterministic* if it succeeds exactly one time without leaving a *choice point*.

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 have failed if it could not be *proven*.

float

Computer's crippled representation of a real number. Represented as 'IEEE double'.

foreign

Computer code expressed in languages other 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* either succeeds, in which case the *variables* in the *compound* terms have a *binding*, or it *fails* if Prolog fails to prove it.

hashing

Indexing technique used for quick lookup.

head

Part of a *clause* before the *neck* operator (`:-`). 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 6.

indexing

Indexing is a technique used to quickly select candidate *clauses* of a *predicate* for a specific *goal*. In most Prolog systems, indexing is done (only) 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 to quickly select all *clauses* where the first argument may *unify* with the first argument of the *goal*. SWI-Prolog supports just-in-time and multi-argument indexing. See section 2.18.

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.

instantiation [of an argument]

To what extend a term is bound to a value. Typical levels are 'unbound' (a *variable*), 'ground' (term without variables) or 'partially bound' (term with embedded variables).

meta-predicate
A *predicate* that reasons about other *predicates*, either by calling them, (re)defining them or querying *properties*.

mode [declaration]
Declaration of an argument *instantiation* pattern for a *predicate*, often accompanied with a *determinism*.

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*.

multi [determinism]
A *predicate* is said to have *determinism* multi if it generates at *least* one answer.

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*.

nondet
Short for *non deterministic*.

non deterministic
A *non deterministic* predicate is a predicate that mail fail or succeed any number of times.

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 uses *indexing* to select candidate *clauses* and then tries these *clauses* one-by-one. See also *backtracking*.

predicate indicator
Term of the form Name/Arity (traditional) or Name//Arity (ISO DCG proposal), where Name is an atom and Arity a non-negative integer. It acts as an *indicator* (or reference) to a predicate or *DCG* rule.

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*.

semidet
Shorthand for

semi deterministic
.

semi deterministic
A *predicate* that is *semi deterministic* either fails or succeeds exactly once without a *choice point*. See also *deterministic*.

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 = B, 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 2.16.1.

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 5.2, 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.
false.

?- (A = b; true; A = c).
A = b ;
true ;
A = c .
```

See also *unify*.

E SWI-Prolog License Conditions and Tools

As of version 7.4.0[1], the SWI-Prolog source code is distributed under the Simplified BSD license:

This, unfortunately, **does not mean you can any version of SWI-Prolog under the above license**. The SWI-Prolog core may be linked to libraries that are more restrictive and in addition your code may have loaded extension packages that have more restrictive conditions. In particular, the core is by default linked to libgmp, distributed under the Lesser GNU Public license.

The above implies you need to configure and recompile the system without these components. For this we provide options to the `configure` script:

```
./configure --without-gpl
./configure --without-lgpl
```

[1]Actually pre-release 7.3.33

The GNU MP Bignum Library provides unbounded integers, rational numbers and some cryptographical functionality. As libgmp is provided under the Lesser GNU Public license it may legally be combined with proprietary software as long as libgmp is *dynamically linked* (default) and the end user can replace the libgmp shared object and use your application with their (possibly modified) version of libgmp. In practice this leads to problems if the application is not accessible (e.g., embedded in closed hardware) or you want to avoid customers to peek around in the process memory as they can easily do so by adding a backdoor to the modified LGPL component. Note that such a protection is in general not possible anyway if the customer has unrestricted access to the machine on which the application runs.

E.1 Contributing to the SWI-Prolog project

To reach maximal coherence we will, as a rule of thumb, only accept new code that has the Simplified BSD license and existing code with a *permissive* license such as MIT, Apache, BSD-3, etc. In exceptional cases we may accept code with GPL or LGPL conditions. Such code must be tagged using a `license/1` directive (Prolog) or a call to `PL_license()` for foreign code and, if they are part of the core, the code must be excluded using the `--without-gpl` or `--without-lgpl` option.

E.2 Software support to keep track of license conditions

Given the above, it is possible that SWI-Prolog packages and extensions rely on the GPL, LGPL or other licenses. The predicates below allow for registering license requirements for Prolog files and foreign modules. The predicate `license/0` reports which components from the currently configured system are distributed under non-permissive open source licenses and therefore may need to be replaced to suit your requirements.

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. Likewise for for LGPL components.

license(*+LicenseId, +Component*)

Register the fact that *Component* is distributed under a license identified by *LicenseId*. Known license identifiers can be listed using `known_licenses/0`. A new license can be registered as a known language using a declaration like below. The second argument defines the *category* if the license, which is one of `gpl`, `lgpl`, `permissive` or `proprietary`.

```
:- multifile license:license/3.

license:license(mylicense, permissive,
                [ comment('My personal license'),
                  url('http://www.mine.org/license.html')
                ]).

:- license(mylicense).
```

license(+*LicenseId*)
Intended 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()`.

known_licenses
List all licenses *known* to the system. This does not imply the system contains code covered by the listed licenses. See `license/2`.

E.3 License conditions inherited from used code

E.3.1 Cryptographic routines

Cryptographic routines are used in `variant_sha1/2` and `crypt`. These routines are provided under the following conditions:

```
Copyright (c) 2002, Dr Brian Gladman, Worcester, UK.   All rights reserved.

LICENSE TERMS

The free distribution and use of this software in both source and binary
form is allowed (with or without changes) provided that:

  1. distributions of this source code include the above copyright
     notice, this list of conditions and the following disclaimer;

  2. distributions in binary form include the above copyright
     notice, this list of conditions and the following disclaimer
     in the documentation and/or other associated materials;

  3. the copyright holder's name is not used to endorse products
     built using this software without specific written permission.

ALTERNATIVELY, provided that this notice is retained in full, this product
may be distributed under the terms of the GNU General Public License (GPL),
in which case the provisions of the GPL apply INSTEAD OF those given above.

DISCLAIMER

This software is provided 'as is' with no explicit or implied warranties
in respect of its properties, including, but not limited to, correctness
and/or fitness for purpose.
```

Summary

F

F.1 Predicates

The predicate summary is used by the Prolog predicate `apropos/1` to suggest predicates from a keyword.

@/2	Call using calling context
`!`/0	Cut (discard choicepoints)
`$`/0	Discard choicepoints and demand deterministic success
`$`/1	Verify goal succeeds deterministically
`,`/2	Conjunction of goals
`->`/2	If-then-else
`*->`/2	Soft-cut
`.`/2	Consult. Also functional notation
`:<`/2	Select keys from a dict
`;`/2	Disjunction of two goals
`<`/2	Arithmetic smaller
=/2	True when arguments are unified
`=..`/2	"Univ." Term to list conversion
`=:=`/2	Arithmetic equality
`=<`/2	Arithmetic smaller or equal
`==`/2	Test for strict equality
`=@=`/2	Test for structural equality (variant)
`=\=`/2	Arithmetic not equal
`>`/2	Arithmetic larger
`>=`/2	Arithmetic larger or equal
`>:<`/2	Partial dict unification
?=/2	Test of terms can be compared now
`@<`/2	Standard order smaller
`@=<`/2	Standard order smaller or equal
`@>`/2	Standard order larger
`@>=`/2	Standard order larger or equal
`\+`/1	Negation by failure. Same as `not/1`
`\=`/2	True if arguments cannot be unified
`\==`/2	True if arguments are not strictly equal
`\=@=`/2	Not structural identical
`^`/2	Existential quantification (`bagof/3`, `setof/3`)
`|`/2	Disjunction in DCGs. Same as `;/2`
`{}`/1	DCG escape; constraints

abolish/1	Remove predicate definition from the database
abolish/2	Remove predicate definition from the database
abolish_all_tables/0	Abolish computed tables
abolish_module_tables/1	Abolish all tables in a module
abolish_monotonic_tables/0	Abolish all monotonic tables
abolish_nonincremental_tables/0	Abolish non-auttomatic tables
abolish_nonincremental_tables/1	Abolish non-auttomatic tables
abolish_private_tables/0	Abolish tables of this thread
abolish_shared_tables/0	Abolish tables shared between threads
abolish_table_subgoals/1	Abolish tables for a goal
abort/0	Abort execution, return to top level
absolute_file_name/2	Get absolute path name
absolute_file_name/3	Get absolute path name with options
answer_count_restraint/0	Undefined answer due to `max_answers`
access_file/2	Check access permissions of a file
acyclic_term/1	Test term for cycles
add_import_module/3	Add module to the auto-import list
add_nb_set/2	Add term to a non-backtrackable set
add_nb_set/3	Add term to a non-backtrackable set
append/1	Append to a file
apple_current_locale_identifier/1	Get Apple locale info
apply/2	Call goal with additional arguments
apropos/1	`online_help` Search manual
arg/3	Access argument of a term
assoc_to_list/2	Convert association tree to list
assert/1	Add a clause to the database
assert/2	Add a clause to the database, give reference
asserta/1	Add a clause to the database (first)
asserta/2	Add a clause to the database (first)
assertion/1	Make assertions about your program
assertz/1	Add a clause to the database (last)
assertz/2	Add a clause to the database (last)
attach_console/0	Attach I/O console to thread
attach_packs/0	Attach add-ons
attach_packs/1	Attach add-ons from directory
attach_packs/2	Attach add-ons from directory
attribute_goals/3	Project attributes to goals
attr_unify_hook/2	Attributed variable unification hook
attr_portray_hook/2	Attributed variable print hook
attvar/1	Type test for attributed variable
at_end_of_stream/0	Test for end of file on input
at_end_of_stream/1	Test for end of file on stream
at_halt/1	Register goal to run at `halt/1`
atom/1	Type check for an atom
atom_chars/2	Convert between atom and list of characters
atom_codes/2	Convert between atom and list of characters codes
atom_concat/3	Contatenate two atoms

atom_length/2	Determine length of an atom
atom_number/2	Convert between atom and number
atom_prefix/2	Test for start of atom
atom_string/2	Conversion between atom and string
atom_to_term/3	Convert between atom and term
atomic/1	Type check for primitive
atomic_concat/3	Concatenate two atomic values to an atom
atomic_list_concat/2	Append a list of atomics
atomic_list_concat/3	Append a list of atomics with separator
atomics_to_string/2	Concatenate list of inputs to a string
atomics_to_string/3	Concatenate list of inputs to a string
autoload/1	Declare a file for autoloading
autoload/2	Declare a file for autoloading specific predicates
autoload_all/0	Autoload all predicates now
autoload_path/1	Add directories for autoloading
b_getval/2	Fetch backtrackable global variable
b_set_dict/3	Destructive assignment on a dict
b_setval/2	Assign backtrackable global variable
bagof/3	Find all solutions to a goal
between/3	Integer range checking/generating
blob/2	Type check for a blob
bounded_number/3	Number between bounds
break/0	Start interactive top level
break_hook/6	(hook) Debugger hook
byte_count/2	Byte-position in a stream
call/1	Call a goal
call/[2..]	Call with additional arguments
call_cleanup/3	Guard a goal with a cleaup-handler
call_cleanup/2	Guard a goal with a cleaup-handler
call_dcg/3	As `phrase/3` without type checking
call_delays/2	Get the condition associated with an answer
call_residue_vars/2	Find residual attributed variables
call_residual_program/2	Get residual program associated with an answer
call_shared_object_function/2	UNIX: Call C-function in shared (.so) file
call_with_depth_limit/3	Prove goal with bounded depth
call_with_inference_limit/3	Prove goal in limited inferences
callable/1	Test for atom or compound term
cancel_halt/1	Cancel `halt/0` from an `at_halt/1` hook
catch/3	Call goal, watching for exceptions
char_code/2	Convert between character and character code
char_conversion/2	Provide mapping of input characters
char_type/2	Classify characters
character_count/2	Get character index on a stream
chdir/1	Compatibility: change working directory
chr_constraint/1	CHR Constraint declaration
chr_show_store/1	List suspended CHR constraints
chr_trace/0	Start CHR tracer

chr_type/1	CHR Type declaration
chr_notrace/0	Stop CHR tracer
chr_leash/1	Define CHR leashed ports
chr_option/2	Specify CHR compilation options
clause/2	Get clauses of a predicate
clause/3	Get clauses of a predicate
clause_property/2	Get properties of a clause
close/1	Close stream
close/2	Close stream (forced)
close_dde_conversation/1	Win32: Close DDE channel
close_shared_object/1	UNIX: Close shared library (.so file)
collation_key/2	Sort key for locale dependent ordering
comment_hook/3	(hook) handle comments in sources
compare/3	Compare, using a predicate to determine the order
compile_aux_clauses/1	Compile predicates for `goal_expansion/2`
compile_predicates/1	Compile dynamic code to static
compiling/0	Is this a compilation run?
compound/1	Test for compound term
compound_name_arity/3	Name and arity of a compound term
compound_name_arguments/3	Name and arguments of a compound term
code_type/2	Classify a character-code
consult/1	Read (compile) a Prolog source file
context_module/1	Get context module of current goal
convert_time/8	Break time stamp into fields
convert_time/2	Convert time stamp to string
copy_stream_data/2	Copy all data from stream to stream
copy_stream_data/3	Copy n bytes from stream to stream
copy_predicate_clauses/2	Copy clauses between predicates
copy_term/2	Make a copy of a term
copy_term/3	Copy a term and obtain attribute-goals
copy_term/4	Copy part of the variables in a term
copy_term_nat/2	Make a copy of a term without attributes
copy_term_nat/4	Copy part of the variables in a term
create_prolog_flag/3	Create a new Prolog flag
current_arithmetic_function/1	Examine evaluable functions
current_atom/1	Examine existing atoms
current_blob/2	Examine typed blobs
current_char_conversion/2	Query input character mapping
current_engine/1	Enumerate known engines
current_flag/1	Examine existing flags
current_foreign_library/2	`shlib` Examine loaded shared libraries (.so files)
current_format_predicate/2	Enumerate user-defined format codes
current_functor/2	Examine existing name/arity pairs
current_input/1	Get current input stream
current_key/1	Examine existing database keys
current_locale/1	Get the current locale
current_module/1	Examine existing modules

current_op/3	Examine current operator declarations
current_output/1	Get the current output stream
current_predicate/1	Examine existing predicates (ISO)
current_predicate/2	Examine existing predicates
current_signal/3	Current software signal mapping
current_stream/3	Examine open streams
current_table/2	Find answer table for a variant
current_transaction/1	Detect encapsulating transactions
current_trie/1	Enumerate known tries
cyclic_term/1	Test term for cycles
day_of_the_week/2	Determine ordinal-day from date
date_time_stamp/2	Convert date structure to time-stamp
date_time_value/3	Extract info from a date structure
dcg_translate_rule/2	Source translation of DCG rules
dcg_translate_rule/4	Source translation of DCG rules
dde_current_connection/2	Win32: Examine open DDE connections
dde_current_service/2	Win32: Examine DDE services provided
dde_execute/2	Win32: Execute command on DDE server
dde_register_service/2	Win32: Become a DDE server
dde_request/3	Win32: Make a DDE request
dde_poke/3	Win32: POKE operation on DDE server
dde_unregister_service/1	Win32: Terminate a DDE service
debug/0	Test for debugging mode
debug/1	Select topic for debugging
debug/3	Print debugging message on topic
debug_control_hook/1	(hook) Extend `spy/1`, etc.
debugging/0	Show debugger status
debugging/1	Test where we are debugging topic
default_module/2	Query module inheritance
del_attr/2	Delete attribute from variable
del_attrs/1	Delete all attributes from variable
del_dict/4	Delete Key-Value pair from a dict
delays_residual_program/2	Get the residual program for an answer
delete_directory/1	Remove a folder from the file system
delete_file/1	Remove a file from the file system
delete_import_module/2	Remove module from import list
det/1	Declare predicates as deterministic
deterministic/1	Test deterministicy of current clause
dif/2	Constrain two terms to be different
directory_files/2	Get entries of a directory/folder
discontiguous/1	Indicate distributed definition of a predicate
divmod/4	Compute quotient and remainder of two integers
downcase_atom/2	Convert atom to lower-case
duplicate_term/2	Create a copy of a term
dwim_match/2	Atoms match in “Do What I Mean” sense
dwim_match/3	Atoms match in “Do What I Mean” sense
dwim_predicate/2	Find predicate in “Do What I Mean” sense

dynamic/1	Indicate predicate definition may change
dynamic/2	Indicate predicate definition may change
edit/0	Edit current script- or associated file
edit/1	Edit a file, predicate, module (extensible)
elif/1	Part of conditional compilation (directive)
else/0	Part of conditional compilation (directive)
empty_assoc/1	Create/test empty association tree
empty_nb_set/1	Test/create an empty non-backtrackable set
encoding/1	Define encoding inside a source file
endif/0	End of conditional compilation (directive)
engine_create/3	Create an interactor
engine_create/4	Create an interactor
engine_destroy/1	Destroy an interactor
engine_fetch/1	Get term from caller
engine_next/2	Ask interactor for next term
engine_next_reified/2	Ask interactor for next term
engine_post/2	Send term to an interactor
engine_post/3	Send term to an interactor and wait for reply
engine_self/1	Get handle to running interactor
engine_yield/1	Make term available to caller
ensure_loaded/1	Consult a file if that has not yet been done
erase/1	Erase a database record or clause
exception/3	(hook) Handle runtime exceptions
exists_directory/1	Check existence of directory
exists_file/1	Check existence of file
exists_source/1	Check existence of a Prolog source
exists_source/2	Check existence of a Prolog source
expand_answer/2	Expand answer of query
expand_file_name/2	Wildcard expansion of file names
expand_file_search_path/2	Wildcard expansion of file paths
expand_goal/2	Compiler: expand goal in clause-body
expand_goal/4	Compiler: expand goal in clause-body
expand_query/4	Expanded entered query
expand_term/2	Compiler: expand read term into clause(s)
expand_term/4	Compiler: expand read term into clause(s)
expects_dialect/1	For which Prolog dialect is this code written?
explain/1	`explain` Explain argument
explain/2	`explain` 2nd argument is explanation of first
export/1	Export a predicate from a module
fail/0	Always false
false/0	Always false
fast_term_serialized/2	Fast term (de-)serialization
fast_read/2	Read binary term serialization
fast_write/2	Write binary term serialization
current_prolog_flag/2	Get system configuration parameters
file_base_name/2	Get file part of path
file_directory_name/2	Get directory part of path

file_name_extension/3	Add, remove or test file extensions
file_search_path/2	Define path-aliases for locating files
find_chr_constraint/1	Returns a constraint from the store
findall/3	Find all solutions to a goal
findall/4	Difference list version of `findall/3`
findnsols/4	Find first *N* solutions
findnsols/5	Difference list version of `findnsols/4`
fill_buffer/1	Fill the input buffer of a stream
flag/3	Simple global variable system
float/1	Type check for a floating point number
float_class/2	Classify (special) floats
float_parts/4	Get mantissa and exponent of a float
flush_output/0	Output pending characters on current stream
flush_output/1	Output pending characters on specified stream
forall/2	Prove goal for all solutions of another goal
format/1	Formatted output
format/2	Formatted output with arguments
format/3	Formatted output on a stream
format_time/3	C strftime() like date/time formatter
format_time/4	date/time formatter with explicit locale
format_predicate/2	Program `format/[1,2]`
term_attvars/2	Find attributed variables in a term
term_variables/2	Find unbound variables in a term
term_variables/3	Find unbound variables in a term
text_to_string/2	Convert arbitrary text to a string
freeze/2	Delay execution until variable is bound
frozen/2	Query delayed goals on var
functor/3	Get name and arity of a term or construct a term
functor/4	Get name and arity of a term or construct a term
garbage_collect/0	Invoke the garbage collector
garbage_collect_atoms/0	Invoke the atom garbage collector
garbage_collect_clauses/0	Invoke clause garbage collector
gen_assoc/3	Enumerate members of association tree
gen_nb_set/2	Generate members of non-backtrackable set
gensym/2	Generate unique atoms from a base
get/1	Read first non-blank character
get/2	Read first non-blank character from a stream
get_assoc/3	Fetch key from association tree
get_assoc/5	Fetch key from association tree
get_attr/3	Fetch named attribute from a variable
get_attrs/2	Fetch all attributes of a variable
get_byte/1	Read next byte (ISO)
get_byte/2	Read next byte from a stream (ISO)
get_char/1	Read next character as an atom (ISO)
get_char/2	Read next character from a stream (ISO)
get_code/1	Read next character (ISO)
get_code/2	Read next character from a stream (ISO)

get_dict/3	Get the value associated to a key from a dict
get_dict/5	Replace existing value in a dict
get_flag/2	Get value of a flag
get_single_char/1	Read next character from the terminal
get_string_code/3	Get character code at index in string
get_time/1	Get current time
get0/1	Read next character
get0/2	Read next character from a stream
getenv/2	Get shell environment variable
goal_expansion/2	Hook for macro-expanding goals
goal_expansion/4	Hook for macro-expanding goals
ground/1	Verify term holds no unbound variables
gdebug/0	Debug using graphical tracer
gspy/1	Spy using graphical tracer
gtrace/0	Trace using graphical tracer
guitracer/0	Install hooks for the graphical debugger
gxref/0	Cross-reference loaded program
halt/0	Exit from Prolog
halt/1	Exit from Prolog with status
term_hash/2	Hash-value of ground term
term_hash/4	Hash-value of term with depth limit
help/0	Give help on help
help/1	Give help on predicates and show parts of manual
help_hook/1	(hook) User-hook in the help-system
if/1	Start conditional compilation (directive)
ignore/1	Call the argument, but always succeed
import/1	Import a predicate from a module
import_module/2	Query import modules
in_pce_thread/1	Run goal in XPCE thread
in_pce_thread_sync/1	Run goal in XPCE thread
include/1	Include a file with declarations
initialization/1	Initialization directive
initialization/2	Initialization directive
initialize/0	Run program initialization
instance/2	Fetch clause or record from reference
integer/1	Type check for integer
interactor/0	Start new thread with console and top level
is/2	Evaluate arithmetic expression
is_absolute_file_name/1	True if arg defines an absolute path
is_assoc/1	Verify association list
is_dict/1	Type check for a dict
is_dict/2	Type check for a dict in a class
is_engine/1	Type check for an engine handle
is_list/1	Type check for a list
is_most_general_term/1	Type check for general term
is_stream/1	Type check for a stream handle
is_trie/1	Type check for a trie handle

is_thread/1	Type check for an thread handle
join_threads/0	Join all terminated threads interactively
keysort/2	Sort, using a key
known_licenses/0	Print known licenses
last/2	Last element of a list
leash/1	Change ports visited by the tracer
length/2	Length of a list
library_directory/1	(hook) Directories holding Prolog libraries
license/0	Evaluate licenses of loaded modules
license/1	Define license for current file
license/2	Define license for named module
line_count/2	Line number on stream
line_position/2	Character position in line on stream
list_debug_topics/0	List registered topics for debugging
list_to_assoc/2	Create association tree from list
list_to_set/2	Remove duplicates from a list
list_strings/0	Help porting to version 7
load_files/1	Load source files
load_files/2	Load source files with options
load_foreign_library/1	`shlib` Load shared library (.so file)
load_foreign_library/2	`shlib` Load shared library (.so file)
locale_create/3	Create a new locale object
locale_destroy/1	Destroy a locale object
locale_property/2	Query properties of locale objects
locale_sort/2	Language dependent sort of atoms
make/0	Reconsult all changed source files
make_directory/1	Create a folder on the file system
make_library_index/1	Create autoload file INDEX.pl
malloc_property/1	Property of the allocator
make_library_index/2	Create selective autoload file INDEX.pl
map_assoc/2	Map association tree
map_assoc/3	Map association tree
dict_create/3	Create a dict from data
dict_pairs/3	Convert between dict and list of pairs
max_assoc/3	Highest key in association tree
memberchk/2	Deterministic `member/2`
message_hook/3	Intercept `print_message/2`
message_line_element/2	(hook) Intercept `print_message_lines/3`
message_property/2	(hook) Define display of a message
message_queue_create/1	Create queue for thread communication
message_queue_create/2	Create queue for thread communication
message_queue_destroy/1	Destroy queue for thread communication
message_queue_property/2	Query message queue properties
message_queue_set/2	Set a message queue property
message_to_string/2	Translate message-term to string
meta_predicate/1	Declare access to other predicates
min_assoc/3	Lowest key in association tree

module/1	Query/set current type-in module
module/2	Declare a module
module/3	Declare a module with language options
module_property/2	Find properties of a module
module_transparent/1	Indicate module based meta-predicate
msort/2	Sort, do not remove duplicates
multifile/1	Indicate distributed definition of predicate
mutex_create/1	Create a thread-synchronisation device
mutex_create/2	Create a thread-synchronisation device
mutex_destroy/1	Destroy a mutex
mutex_lock/1	Become owner of a mutex
mutex_property/2	Query mutex properties
mutex_statistics/0	Print statistics on mutex usage
mutex_trylock/1	Become owner of a mutex (non-blocking)
mutex_unlock/1	Release ownership of mutex
mutex_unlock_all/0	Release ownership of all mutexes
name/2	Convert between atom and list of character codes
nb_current/2	Enumerate non-backtrackable global variables
nb_delete/1	Delete a non-backtrackable global variable
nb_getval/2	Fetch non-backtrackable global variable
nb_link_dict/3	Non-backtrackable assignment to dict
nb_linkarg/3	Non-backtrackable assignment to term
nb_linkval/2	Assign non-backtrackable global variable
nb_set_to_list/2	Convert non-backtrackable set to list
nb_set_dict/3	Non-backtrackable assignment to dict
nb_setarg/3	Non-backtrackable assignment to term
nb_setval/2	Assign non-backtrackable global variable
nl/0	Generate a newline
nl/1	Generate a newline on a stream
nodebug/0	Disable debugging
nodebug/1	Disable debug-topic
noguitracer/0	Disable the graphical debugger
nonground/2	Term is not ground due to witness
nonvar/1	Type check for bound term
nonterminal/1	Set predicate property
noprofile/1	Hide (meta-) predicate for the profiler
noprotocol/0	Disable logging of user interaction
normalize_space/2	Normalize white space
nospy/1	Remove spy point
nospyall/0	Remove all spy points
not/1	Negation by failure (argument not provable). Same as `\+/1`
not_exists/1	Tabled negation for non-ground or non-tabled goals
notrace/0	Stop tracing
notrace/1	Do not debug argument goal
nth_clause/3	N-th clause of a predicate
nth_integer_root_and_remainder/4	Integer root and remainder
number/1	Type check for integer or float

number_chars/2	Convert between number and one-char atoms
number_codes/2	Convert between number and character codes
number_string/2	Convert between number and string
numbervars/3	Number unbound variables of a term
numbervars/4	Number unbound variables of a term
on_signal/3	Handle a software signal
once/1	Call a goal deterministically
op/3	Declare an operator
open/3	Open a file (creating a stream)
open/4	Open a file (creating a stream)
open_dde_conversation/3	Win32: Open DDE channel
open_null_stream/1	Open a stream to discard output
open_resource/3	Open a program resource as a stream
open_shared_object/2	UNIX: Open shared library (.so file)
open_shared_object/3	UNIX: Open shared library (.so file)
open_source_hook/3	(hook) Open a source file
open_string/2	Open a string as a stream
ord_list_to_assoc/2	Convert ordered list to assoc
parse_time/2	Parse text to a time-stamp
parse_time/3	Parse text to a time-stamp
pce_dispatch/1	Run XPCE GUI in separate thread
pce_call/1	Run goal in XPCE GUI thread
peek_byte/1	Read byte without removing
peek_byte/2	Read byte without removing
peek_char/1	Read character without removing
peek_char/2	Read character without removing
peek_code/1	Read character-code without removing
peek_code/2	Read character-code without removing
peek_string/3	Read a string without removing
phrase/2	Activate grammar-rule set
phrase/3	Activate grammar-rule set (returning rest)
phrase_from_quasi_quotation/2	Parse quasi quotation with DCG
please/3	Query/change environment parameters
plus/3	Logical integer addition
portray/1	(hook) Modify behaviour of `print/1`
predicate_property/2	Query predicate attributes
predsort/3	Sort, using a predicate to determine the order
print/1	Print a term
print/2	Print a term on a stream
print_message/2	Print message from (exception) term
print_message_lines/3	Print message to stream
profile/1	Obtain execution statistics
profile/2	Obtain execution statistics
profile_count/3	Obtain profile results on a predicate
profiler/2	Obtain/change status of the profiler
prolog/0	Run interactive top level
prolog_alert_signal/2	Query/set unblock signal

prolog_choice_attribute/3	Examine the choice point stack
prolog_current_choice/1	Reference to most recent choice point
prolog_current_frame/1	Reference to goal's environment stack
prolog_cut_to/1	Realise global cuts
prolog_edit:locate/2	Locate targets for `edit/1`
prolog_edit:locate/3	Locate targets for `edit/1`
prolog_edit:edit_source/1	Call editor for `edit/1`
prolog_edit:edit_command/2	Specify editor activation
prolog_edit:load/0	Load `edit/1` extensions
prolog_exception_hook/4	Rewrite exceptions
prolog_file_type/2	Define meaning of file extension
prolog_frame_attribute/3	Obtain information on a goal environment
prolog_ide/1	Program access to the development environment
prolog_list_goal/1	(hook) Intercept tracer 'L' command
prolog_listen/2	Listen to Prolog events
prolog_listen/3	Listen to Prolog events
prolog_load_context/2	Context information for directives
prolog_load_file/2	(hook) Program `load_files/2`
prolog_skip_level/2	Indicate deepest recursion to trace
prolog_skip_frame/1	Perform 'skip' on a frame
prolog_stack_property/2	Query properties of the stacks
prolog_to_os_filename/2	Convert between Prolog and OS filenames
prolog_trace_interception/4	`user` Intercept the Prolog tracer
prolog_unlisten/2	Stop listening to Prolog events
project_attributes/2	Project constraints to query variables
prompt1/1	Change prompt for 1 line
prompt/2	Change the prompt used by `read/1`
protocol/1	Make a log of the user interaction
protocola/1	Append log of the user interaction to file
protocolling/1	On what file is user interaction logged
public/1	Declaration that a predicate may be called
put/1	Write a character
put/2	Write a character on a stream
put_assoc/4	Add Key-Value to association tree
put_attr/3	Put attribute on a variable
put_attrs/2	Set/replace all attributes on a variable
put_byte/1	Write a byte
put_byte/2	Write a byte on a stream
put_char/1	Write a character
put_char/2	Write a character on a stream
put_code/1	Write a character-code
put_code/2	Write a character-code on a stream
put_dict/3	Add/replace multiple keys in a dict
put_dict/4	Add/replace a single key in a dict
qcompile/1	Compile source to Quick Load File
qcompile/2	Compile source to Quick Load File
qsave_program/1	Create runtime application

qsave_program/2	Create runtime application
quasi_quotation_syntax/1	Declare quasi quotation syntax
quasi_quotation_syntax_error/1	Raise syntax error
radial_restraint/0	Tabbling radial restraint was violated
random_property/1	Query properties of random generation
rational/1	Type check for a rational number
rational/3	Decompose a rational
read/1	Read Prolog term
read/2	Read Prolog term from stream
read_clause/3	Read clause from stream
read_link/3	Read a symbolic link
read_pending_codes/3	Fetch buffered input from a stream
read_pending_chars/3	Fetch buffered input from a stream
read_string/3	Read a number of characters into a string
read_string/5	Read string upto a delimiter
read_term/2	Read term with options
read_term/3	Read term with options from stream
read_term_from_atom/3	Read term with options from atom
read_term_with_history/2	Read term with command line history
recorda/2	Record term in the database (first)
recorda/3	Record term in the database (first)
recorded/2	Obtain term from the database
recorded/3	Obtain term from the database
recordz/2	Record term in the database (last)
recordz/3	Record term in the database (last)
redefine_system_predicate/1	Abolish system definition
reexport/1	Load files and re-export the imported predicates
reexport/2	Load predicates from a file and re-export it
reload_foreign_libraries/0	Reload DLLs/shared objects
reload_library_index/0	Force reloading the autoload index
rename_file/2	Change name of file
repeat/0	Succeed, leaving infinite backtrack points
require/1	This file requires these predicates
reset/3	Wrapper for delimited continuations
reset_gensym/1	Reset a gensym key
reset_gensym/0	Reset all gensym keys
reset_profiler/0	Clear statistics obtained by the profiler
resource/2	Declare a program resource
resource/3	Declare a program resource
retract/1	Remove clause from the database
retractall/1	Remove unifying clauses from the database
same_file/2	Succeeds if arguments refer to same file
same_term/2	Test terms to be at the same address
see/1	Change the current input stream
seeing/1	Query the current input stream
seek/4	Modify the current position in a stream
seen/0	Close the current input stream

select_dict/2	Select matching attributes from a dict
select_dict/3	Select matching attributes from a dict
set_end_of_stream/1	Set physical end of an open file
set_flag/2	Set value of a flag
set_input/1	Set current input stream from a stream
set_locale/1	Set the default local
set_malloc/1	Set memory allocator property
set_module/1	Set properties of a module
set_output/1	Set current output stream from a stream
set_prolog_IO/3	Prepare streams for interactive session
set_prolog_flag/2	Define a system feature
set_prolog_gc_thread/1	Control the gc thread
set_prolog_stack/2	Modify stack characteristics
set_random/1	Control random number generation
set_stream/2	Set stream attribute
set_stream_position/2	Seek stream to position
set_system_IO/3	Rebind stdin/stderr/stdout
setup_call_cleanup/3	Undo side-effects safely
setup_call_catcher_cleanup/4	Undo side-effects safely
setarg/3	Destructive assignment on term
setenv/2	Set shell environment variable
setlocale/3	Set/query C-library regional information
setof/3	Find all unique solutions to a goal
shell/1	Execute OS command
shell/2	Execute OS command
shift/1	Shift control to the closest `reset/3`
shift_for_copy/1	Shift control to the closest `reset/3`
show_profile/1	Show results of the profiler
sig_atomic/1	Run goal without handling signals
size_abstract_term/3	Abstract a term (tabling support)
size_file/2	Get size of a file in characters
size_nb_set/2	Determine size of non-backtrackable set
skip/1	Skip to character in current input
skip/2	Skip to character on stream
sleep/1	Suspend execution for specified time
snapshot/1	Run goal in isolation
sort/2	Sort elements in a list
sort/4	Sort elements in a list
source_exports/2	Check whether source exports a predicate
source_file/1	Examine currently loaded source files
source_file/2	Obtain source file of predicate
source_file_property/2	Information about loaded files
source_location/2	Location of last read term
split_string/4	Break a string into substrings
spy/1	Force tracer on specified predicate
stamp_date_time/3	Convert time-stamp to date structure
statistics/0	Show execution statistics

statistics/2	Obtain collected statistics
stream_pair/3	Create/examine a bi-directional stream
stream_position_data/3	Access fields from stream position
stream_property/2	Get stream properties
string/1	Type check for string
string_concat/3	`atom_concat/3` for strings
string_length/2	Determine length of a string
string_chars/2	Conversion between string and list of characters
string_codes/2	Conversion between string and list of character codes
string_code/3	Get or find a character code in a string
string_lower/2	Case conversion to lower case
string_upper/2	Case conversion to upper case
string_predicate/1	(hook) Predicate contains strings
strip_module/3	Extract context module and term
style_check/1	Change level of warnings
sub_atom/5	Take a substring from an atom
sub_atom_icasechk/3	Case insensitive substring match
sub_string/5	Take a substring from a string
subsumes_term/2	One-sided unification test
succ/2	Logical integer successor relation
swritef/2	Formatted write on a string
swritef/3	Formatted write on a string
tab/1	Output number of spaces
tab/2	Output number of spaces on a stream
table/1	Declare predicate to be tabled
tabled_call/1	Helper for `not_exists/1`
tdebug/0	Switch all threads into debug mode
tdebug/1	Switch a thread into debug mode
tell/1	Change current output stream
telling/1	Query current output stream
term_expansion/2	(hook) Convert term before compilation
term_expansion/4	(hook) Convert term before compilation
term_singletons/2	Find singleton variables in a term
term_string/2	Read/write a term from/to a string
term_string/3	Read/write a term from/to a string
term_subsumer/3	Most specific generalization of two terms
term_to_atom/2	Convert between term and atom
thread_affinity/3	Query and control the *affinity* mask
thread_alias/1	Set the alias name of a thread
thread_at_exit/1	Register goal to be called at exit
thread_create/2	Create a new Prolog task
thread_create/3	Create a new Prolog task
thread_detach/1	Make thread cleanup after completion
thread_exit/1	Terminate Prolog task with value
thread_get_message/1	Wait for message
thread_get_message/2	Wait for message in a queue
thread_get_message/3	Wait for message in a queue

thread_idle/2	Reduce footprint while waiting
thread_initialization/1	Run action at start of thread
thread_join/1	Wait for Prolog task-completion
thread_join/2	Wait for Prolog task-completion
thread_local/1	Declare thread-specific clauses for a predicate
thread_message_hook/3	Thread local `message_hook/3`
thread_peek_message/1	Test for message
thread_peek_message/2	Test for message in a queue
thread_property/2	Examine Prolog threads
thread_self/1	Get identifier of current thread
thread_send_message/2	Send message to another thread
thread_send_message/3	Send message to another thread
thread_setconcurrency/2	Number of active threads
thread_signal/2	Execute goal in another thread
thread_statistics/3	Get statistics of another thread
thread_update/2	Update a module and signal waiters
thread_wait/2	Wait for a goal to become true
threads/0	List running threads
throw/1	Raise an exception (see `catch/3`)
time/1	Determine time needed to execute goal
time_file/2	Get last modification time of file
tmp_file/2	Create a temporary filename
tmp_file_stream/3	Create a temporary file and open it
tnodebug/0	Switch off debug mode in all threads
tnodebug/1	Switch off debug mode in a thread
tnot/1	Tabled negation
told/0	Close current output
tprofile/1	Profile a thread for some period
trace/0	Start the tracer
trace/1	Set trace point on predicate
trace/2	Set/Clear trace point on ports
tracing/0	Query status of the tracer
transaction/1	Run goal in a transaction
transaction/2	Run goal in a transaction
transaction/3	Run goal in a transaction
transaction_updates/1	Updates to be committed in a transaction
trie_delete/3	Remove term from trie
trie_destroy/1	Destroy a trie
trie_gen/3	Get all terms from a trie
trie_gen_compiled/2	Get all terms from a trie
trie_gen_compiled/3	Get all terms from a trie
trie_insert/2	Insert term into a trie
trie_insert/3	Insert term into a trie
trie_insert/4	Insert term into a trie
trie_lookup/3	Lookup a term in a trie
trie_new/1	Create a trie
trie_property/2	Examine a trie's properties

trie_update/3	Update associated value in trie
trie_term/2	Get term from a trie by handle
trim_heap/0	Release unused malloc() resources
trim_stacks/0	Release unused stack resources
tripwire/2	(hook) Handle a tabling tripwire event
true/0	Succeed
tspy/1	Set spy point and enable debugging in all threads
tspy/2	Set spy point and enable debugging in a thread
tty_get_capability/3	Get terminal parameter
tty_goto/2	Goto position on screen
tty_put/2	Write control string to terminal
tty_size/2	Get row/column size of the terminal
ttyflush/0	Flush output on terminal
undefined/0	Well Founded Semantics: true nor false
undo/1	Schedule goal for backtracking
unify_with_occurs_check/2	Logically sound unification
unifiable/3	Determining binding required for unification
unknown/2	Trap undefined predicates
unload_file/1	Unload a source file
unload_foreign_library/1	`shlib` Detach shared library (.so file)
unload_foreign_library/2	`shlib` Detach shared library (.so file)
unsetenv/1	Delete shell environment variable
untable/1	Remove tabling instrumentation
upcase_atom/2	Convert atom to upper-case
use_foreign_library/1	Load DLL/shared object (directive)
use_foreign_library/2	Load DLL/shared object (directive)
use_module/1	Import a module
use_module/2	Import predicates from a module
valid_string_goal/1	(hook) Goal handles strings
var/1	Type check for unbound variable
var_number/2	Check that var is numbered by numbervars
var_property/2	Variable properties during macro expansion
variant_sha1/2	Term-hash for term-variants
variant_hash/2	Term-hash for term-variants
version/0	Print system banner message
version/1	Add messages to the system banner
visible/1	Ports that are visible in the tracer
volatile/1	Predicates that are not saved
wait_for_input/3	Wait for input with optional timeout
when/2	Execute goal when condition becomes true
wildcard_match/2	POSIX style glob pattern matching
wildcard_match/3	POSIX style glob pattern matching
win_add_dll_directory/1	Add directory to DLL search path
win_add_dll_directory/2	Add directory to DLL search path
win_remove_dll_directory/1	Remove directory from DLL search path
win_exec/2	Win32: spawn Windows task
win_has_menu/0	Win32: true if console menu is available

win_folder/2	Win32: get special folder by CSIDL
win_insert_menu/2	swipl-win.exe: add menu
win_insert_menu_item/4	swipl-win.exe: add item to menu
win_process_modules/1	Win32 get .exe and .dll files of the process
win_shell/2	Win32: open document through Shell
win_shell/3	Win32: open document through Shell
win_registry_get_value/3	Win32: get registry value
win_get_user_preferred_ui_languages/2	Win32: get language preferences
win_window_color/2	Win32: change colors of console window
win_window_pos/1	Win32: change size and position of window
window_title/2	Win32: change title of window
with_mutex/2	Run goal while holding mutex
with_output_to/2	Write to strings and more
with_quasi_quotation_input/3	Parse quasi quotation from stream
with_tty_raw/1	Run goal with terminal in raw mode
working_directory/2	Query/change CWD
write/1	Write term
write/2	Write term to stream
writeln/1	Write term, followed by a newline
writeln/2	Write term, followed by a newline to a stream
write_canonical/1	Write a term with quotes, ignore operators
write_canonical/2	Write a term with quotes, ignore operators on a stream
write_length/3	Dermine #characters to output a term
write_term/2	Write term with options
write_term/3	Write term with options to stream
writef/1	Formatted write
writef/2	Formatted write on stream
writeq/1	Write term, insert quotes
writeq/2	Write term, insert quotes on stream

F.2 Library predicates

F.2.1 library(aggregate)

aggregate/3	Aggregate bindings in Goal according to Template.
aggregate/4	Aggregate bindings in Goal according to Template.
aggregate_all/3	Aggregate bindings in Goal according to Template.
aggregate_all/4	Aggregate bindings in Goal according to Template.
foreach/2	True when the conjunction of _instances_ of Goal created from solutions for Generator is true.
free_variables/4	Find free variables in bagof/setof template.

F.2.2 library(ansi_term)

ansi_format/3	Format text with ANSI attributes.
ansi_get_color/2	Obtain the RGB color for an ANSI color parameter.
console_color/2	Hook that allows for mapping abstract terms to concrete ANSI attributes.

F.2.3 library(apply)

convlist/3	Similar to maplist/3, but elements for which call(Goal, ElemIn, _) fails are omitted from ListOut.
exclude/3	Filter elements for which Goal fails.
foldl/4	Fold an ensemble of _m_ (0 <= _m_ <= 4) lists of length _n_ head-to-tail ("fold-left"), using columns of _m_
foldl/5	Fold an ensemble of _m_ (0 <= _m_ <= 4) lists of length _n_ head-to-tail ("fold-left"), using columns of _m_
foldl/6	Fold an ensemble of _m_ (0 <= _m_ <= 4) lists of length _n_ head-to-tail ("fold-left"), using columns of _m_
foldl/7	Fold an ensemble of _m_ (0 <= _m_ <= 4) lists of length _n_ head-to-tail ("fold-left"), using columns of _m_
include/3	Filter elements for which Goal succeeds.
maplist/2	True if Goal is successfully applied on all matching elements of the list.
maplist/3	True if Goal is successfully applied on all matching elements of the list.
maplist/4	True if Goal is successfully applied on all matching elements of the list.
maplist/5	True if Goal is successfully applied on all matching elements of the list.
partition/4	Filter elements of List according to Pred.
partition/5	Filter List according to Pred in three sets.
scanl/4	Scan an ensemble of _m_ (0 <= _m_ <= 4) lists of length _n_ head-to-tail ("scan-left"), using columns of _m
scanl/5	Scan an ensemble of _m_ (0 <= _m_ <= 4) lists of length _n_ head-to-tail ("scan-left"), using columns of _m
scanl/6	Scan an ensemble of _m_ (0 <= _m_ <= 4) lists of length _n_ head-to-tail ("scan-left"), using columns of _m
scanl/7	Scan an ensemble of _m_ (0 <= _m_ <= 4) lists of length _n_ head-to-tail ("scan-left"), using columns of _m

F.2.4 library(assoc)

assoc_to_list/2	Translate assoc into a pairs list
assoc_to_keys/2	Translate assoc into a key list
assoc_to_values/2	Translate assoc into a value list
empty_assoc/1	Test/create an empty assoc
gen_assoc/3	Non-deterministic enumeration of assoc
get_assoc/3	Get associated value
get_assoc/5	Get and replace associated value

list_to_assoc/2	Translate pair list to assoc
map_assoc/2	Test assoc values
map_assoc/3	Map assoc values
max_assoc/3	Max key-value of an assoc
min_assoc/3	Min key-value of an assoc
ord_list_to_assoc/2	Translate ordered list into an assoc
put_assoc/4	Add association to an assoc

F.2.5 library(broadcast)

broadcast/1	Send event notification
broadcast_request/1	Request all agents
listen/2	Listen to event notifications
listen/3	Listen to event notifications
unlisten/1	Stop listening to event notifications
unlisten/2	Stop listening to event notifications
unlisten/3	Stop listening to event notifications
listening/3	Who is listening to event notifications?

F.2.6 library(charsio)

atom_to_chars/2	Convert Atom into a list of character codes.
atom_to_chars/3	Convert Atom into a difference list of character codes.
format_to_chars/3	Use format/2 to write to a list of character codes.
format_to_chars/4	Use format/2 to write to a difference list of character codes.
number_to_chars/2	Convert Atom into a list of character codes.
number_to_chars/3	Convert Number into a difference list of character codes.
open_chars_stream/2	Open Codes as an input stream.
read_from_chars/2	Read Codes into Term.
read_term_from_chars/3	Read Codes into Term.
with_output_to_chars/2	Run Goal as with once/1.
with_output_to_chars/3	Run Goal as with once/1.
with_output_to_chars/4	Same as with_output_to_chars/3 using an explicit stream.
write_to_chars/2	Write a term to a code list.
write_to_chars/3	Write a term to a code list.

F.2.7 library(check)

check/0	Run all consistency checks defined by checker/2.
checker/2	Register code validation routines.
list_autoload/0	Report predicates that may be auto-loaded.
list_cross_module_calls/0	List calls from one module to another using Module:Goal where the callee is not defined exp
list_format_errors/0	List argument errors for format/2,3.
list_format_errors/1	List argument errors for format/2,3.
list_rationals/0	List rational numbers that appear in clauses.
list_rationals/1	List rational numbers that appear in clauses.

list_redefined/0	Lists predicates that are defined in the global module =user= as well as in a normal module; t
list_strings/0	List strings that appear in clauses.
list_strings/1	List strings that appear in clauses.
list_trivial_fails/0	List goals that trivially fail because there is no matching clause.
list_trivial_fails/1	List goals that trivially fail because there is no matching clause.
list_undefined/0	Report undefined predicates.
list_undefined/1	Report undefined predicates.
list_void_declarations/0	List predicates that have declared attributes, but no clauses.
string_predicate/1	Multifile hook to disable list_strings/0 on the given predicate.
trivial_fail_goal/1	Multifile hook that tells list_trivial_fails/0 to accept Goal as valid.
valid_string_goal/1	Multifile hook that qualifies Goal as valid for list_strings/0.

F.2.8 library(clpb)

labeling/1	Enumerate concrete solutions.
random_labeling/2	Select a single random solution.
sat/1	True iff Expr is a satisfiable Boolean expression.
sat_count/2	Count the number of admissible assignments.
taut/2	Tautology check.
weighted_maximum/3	Enumerate weighted optima over admissible assignments.

F.2.9 library(clpfd)

#/\/2	P and Q hold.
#</2	The arithmetic expression X is less than Y.
#<==/2	Q implies P.
#<==>/2	P and Q are equivalent.
#=/2	The arithmetic expression X equals Y.
#=</2	The arithmetic expression X is less than or equal to Y.
#==>/2	P implies Q.
#>/2	Same as Y #< X.
#>=/2	Same as Y #=< X.
#\/1	Q does _not_ hold.
#\/2	Either P holds or Q holds, but not both.
#\//2	P or Q holds.
#\=/2	The arithmetic expressions X and Y evaluate to distinct integers.
all_different/1	Like all_distinct/1, but with weaker propagation.
all_distinct/1	True iff Vars are pairwise distinct.
automaton/3	Describes a list of finite domain variables with a finite automaton.
automaton/8	Describes a list of finite domain variables with a finite automaton.
chain/2	Zs form a chain with respect to Relation.
circuit/1	True iff the list Vs of finite domain variables induces a Hamiltonian circuit.
cumulative/1	Equivalent to cumulative(Tasks, [limit(1)]).
cumulative/2	Schedule with a limited resource.
disjoint2/1	True iff Rectangles are not overlapping.
element/3	The N-th element of the list of finite domain variables Vs is V.

empty_fdset/1	Set is the empty FD set.
empty_interval/2	Min..Max is an empty interval.
fd_degree/2	Degree is the number of constraints currently attached to Var.
fd_dom/2	Dom is the current domain (see in/2) of Var.
fd_inf/2	Inf is the infimum of the current domain of Var.
fd_set/2	Set is the FD set representation of the current domain of Var.
fd_size/2	Reflect the current size of a domain.
fd_sup/2	Sup is the supremum of the current domain of Var.
fd_var/1	True iff Var is a CLP(FD) variable.
fdset_add_element/3	Set2 is the same FD set as Set1, but with the integer Elt added.
fdset_complement/2	The FD set Complement is the complement of the FD set Set.
fdset_del_element/3	Set2 is the same FD set as Set1, but with the integer Elt removed.
fdset_disjoint/2	The FD sets Set1 and Set2 have no elements in common.
fdset_eq/2	True if the FD sets Set1 and Set2 are equal, i.
fdset_intersect/2	The FD sets Set1 and Set2 have at least one element in common.
fdset_intersection/3	Intersection is an FD set (possibly empty) of all elements that the FD sets Set1 and Set2 have in c
fdset_interval/3	Interval is a non-empty FD set consisting of the single interval Min..Max.
fdset_max/2	Max is the upper bound (supremum) of the non-empty FD set Set.
fdset_member/2	The integer Elt is a member of the FD set Set.
fdset_min/2	Min is the lower bound (infimum) of the non-empty FD set Set.
fdset_parts/4	Set is a non-empty FD set representing the domain Min..Max \/ Rest, where Min..Max is a non-
fdset_singleton/2	Set is the FD set containing the single integer Elt.
fdset_size/2	Size is the number of elements of the FD set Set, or the atom *sup* if Set is infinite.
fdset_subset/2	The FD set Set1 is a (non-strict) subset of Set2, i.
fdset_subtract/3	The FD set Difference is Set1 with all elements of Set2 removed, i.
fdset_to_list/2	List is a list containing all elements of the finite FD set Set, in ascending order.
fdset_to_range/2	Domain is a domain equivalent to the FD set Set.
fdset_union/2	The FD set Union is the n-ary union of all FD sets in the list Sets.
fdset_union/3	The FD set Union is the union of FD sets Set1 and Set2.
global_cardinality/2	Global Cardinality constraint.
global_cardinality/3	Global Cardinality constraint.
in/2	Var is an element of Domain.
in_set/2	Var is an element of the FD set Set.
indomain/1	Bind Var to all feasible values of its domain on backtracking.
ins/2	The variables in the list Vars are elements of Domain.
is_fdset/1	Set is currently bound to a valid FD set.
label/1	Equivalent to labeling([], Vars).
labeling/2	Assign a value to each variable in Vars.
lex_chain/1	Lists are lexicographically non-decreasing.
list_to_fdset/2	Set is an FD set containing all elements of List, which must be a list of integers.
range_to_fdset/2	Set is an FD set equivalent to the domain Domain.
scalar_product/4	True iff the scalar product of Cs and Vs is in relation Rel to Expr.
serialized/2	Describes a set of non-overlapping tasks.
sum/3	The sum of elements of the list Vars is in relation Rel to Expr.
tuples_in/2	True iff all Tuples are elements of Relation.
zcompare/3	Analogous to compare/3, with finite domain variables A and B.

F.2.10 library(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

F.2.11 library(csv)

csv_options/2	Compiled is the compiled representation of the CSV processing options as they may be passed into
csv_read_file/2	Read a CSV file into a list of rows.
csv_read_file/3	Read a CSV file into a list of rows.
csv_read_file_row/3	True when Row is a row in File.
csv_read_row/3	Read the next CSV record from Stream and unify the result with Row.
csv_read_stream/3	Read CSV data from Stream.
csv_write_file/2	Write a list of Prolog terms to a CSV file.
csv_write_file/3	Write a list of Prolog terms to a CSV file.
csv_write_stream/3	Write the rows in Data to Stream.
csv//1	Prolog DCG to 'read/write' CSV data.
csv//2	Prolog DCG to 'read/write' CSV data.

F.2.12 library(dcgbasics)

alpha_to_lower//1	Read a letter (class =alpha=) and return it as a lowercase letter.
atom//1	Generate codes of Atom.
blank//0	Take next =space= character from input.
blanks//0	Skip zero or more white-space characters.
blanks_to_nl//0	Take a sequence of blank//0 codes if blanks are followed by a newline or end of the input.
digit//1	Number processing.
digits//1	Number processing.
eol//0	Matches end-of-line.
eos//0	Matches end-of-input.
float//1	Process a floating point number.
integer//1	Number processing.
nonblank//1	Code is the next non-blank (=graph=) character.
nonblanks//1	Take all =graph= characters.
number//1	Generate extract a number.
prolog_var_name//1	Matches a Prolog variable name.
remainder//1	Unify List with the remainder of the input.
string//1	Take as few as possible tokens from the input, taking one more each time on backtracking.
string_without//2	Take as many codes from the input until the next character code appears in the list EndCodes.

white//0	Take next =white= character from input.
whites//0	Skip white space _inside_ a line.
xdigit//1	True if the next code is a hexdecimal digit with Weight.
xdigits//1	List of weights of a sequence of hexadecimal codes.
xinteger//1	Generate or extract an integer from a sequence of hexadecimal digits.

F.2.13 library(dcghighorder)

foreach//2	Generate a list from the solutions of Generator.
foreach//3	Generate a list from the solutions of Generator.
optional//2	Perform an optional match, executing Default if Match is not matched.
sequence//2	Match or generate a sequence of Element.
sequence//3	Match or generate a sequence of Element where each pair of elements is separated by Sep.
sequence//5	Match or generate a sequence of Element enclosed by Start end End, where each pair of elements is separa

F.2.14 library(debug)

assertion/1	Acts similar to C assert() macro.
assertion_failed/2	This hook is called if the Goal of assertion/1 fails.
debug/1	Add/remove a topic from being printed.
debug/3	Format a message if debug topic is enabled.
debug_message_context/1	Specify additional context for debug messages.
debug_print_hook/3	Hook called by debug/3.
debugging/1	Examine debug topics.
debugging/2	Examine debug topics.
list_debug_topics/0	List currently known debug topics and their setting.
nodebug/1	Add/remove a topic from being printed.

F.2.15 library(dicts)

dict_fill/4	Implementation for the dicts_to_same_keys/3 ‘OnEmpty‘ closure that fills new cells with a copy
dict_keys/2	True when Keys is an ordered set of the keys appearing in Dict.
dict_size/2	True when KeyCount is the number of keys in Dict.
dicts_join/3	Join dicts in Dicts that have the same value for Key, provided they do not have conflicting value
dicts_join/4	Join two lists of dicts (Dicts1 and Dicts2) on Key.
dicts_same_keys/2	True if List is a list of dicts that all have the same keys and Keys is an ordered set of these keys.
dicts_same_tag/2	True when List is a list of dicts that all have the tag Tag.
dicts_slice/3	DictsOut is a list of Dicts only containing values for Keys.
dicts_to_compounds/4	True when Dicts and Compounds are lists of the same length and each element of Compounds i
dicts_to_same_keys/3	DictsOut is a copy of DictsIn, where each dict contains all keys appearing in all dicts of DictsIn

F.2.16 library(error)

current_encoding/1	True if Name is the name of a supported encoding.
current_type/3	True when Type is a currently defined type and Var satisfies Type of the body term Body succe
domain_error/2	The argument is of the proper type, but has a value that is outside the supported values.

existence_error/2	Culprit is of the correct type and correct domain, but there is no existing (external) resource of t
existence_error/3	Culprit is of the correct type and correct domain, but there is no existing (external) resource of t
has_type/2	True if Term satisfies Type.
instantiation_error/1	An argument is under-instantiated.
is_of_type/2	True if Term satisfies Type.
must_be/2	True if Term satisfies the type constraints for Type.
permission_error/3	It is not allowed to perform Operation on (whatever is represented by) Culprit that is of the give
representation_error/1	A representation error indicates a limitation of the implementation.
resource_error/1	A goal cannot be completed due to lack of resources.
syntax_error/1	A text has invalid syntax.
type_error/2	Tell the user that Culprit is not of the expected ValidType.
uninstantiation_error/1	An argument is over-instantiated.

F.2.17 library(explain)

explain/1	Give an explanation on Term.
explain/2	True when Explanation is an explanation of Term.

F.2.18 library(help)

apropos/1	Print objects from the manual whose name or summary match with Query.
help/0	Show help for What.
help/1	Show help for What.
show_html_hook/1	Hook called to display the extracted HTML document.

F.2.19 library(increval)

F.2.20 library(summaries.d/increval.tex)

F.2.21 library(intercept)

F.2.22 library(summaries.d/intercept.tex)

F.2.23 library(iostream)

F.2.24 library(summaries.d/iostream.tex)

F.2.25 library(listing)

listing/0	Lists all predicates defined in the calling module.
listing/1	List matching clauses.
listing/2	List matching clauses.
portray_clause/1	Portray 'Clause' on the current output stream.
portray_clause/2	Portray 'Clause' on the current output stream.
portray_clause/3	Portray 'Clause' on the current output stream.

F.2.26 library(lists)

append/2	Concatenate a list of lists.
append/3	List1AndList2 is the concatenation of List1 and List2.
clumped/2	Pairs is a list of ‘Item-Count‘ pairs that represents the _run length encoding_ of Items.
delete/3	Delete matching elements from a list.
flatten/2	Is true if FlatList is a non-nested version of NestedList.
intersection/3	True if Set3 unifies with the intersection of Set1 and Set2.
is_set/1	True if Set is a proper list without duplicates.
last/2	Succeeds when Last is the last element of List.
list_to_set/2	True when Set has the same elements as List in the same order.
max_list/2	True if Max is the largest number in List.
max_member/2	True when Max is the largest member in the standard order of terms.
max_member/3	True when Max is the largest member according to Pred, which must be a 2-argument callable that bel
member/2	True if Elem is a member of List.
min_list/2	True if Min is the smallest number in List.
min_member/2	True when Min is the smallest member in the standard order of terms.
min_member/3	True when Min is the smallest member according to Pred, which must be a 2-argument callable that be
nextto/3	True if Y directly follows X in List.
nth0/3	True when Elem is the Index’th element of List.
nth0/4	Select/insert element at index.
nth1/3	Is true when Elem is the Index’th element of List.
nth1/4	As nth0/4, but counting starts at 1.
numlist/3	List is a list [Low, Low+1, ... High].
permutation/2	True when Xs is a permutation of Ys.
prefix/2	True iff Part is a leading substring of Whole.
proper_length/2	True when Length is the number of elements in the proper list List.
reverse/2	Is true when the elements of List2 are in reverse order compared to List1.
same_length/2	Is true when List1 and List2 are lists with the same number of elements.
select/3	Is true when List1, with Elem removed, results in List2.
select/4	Select from two lists at the same position.
selectchk/3	Semi-deterministic removal of first element in List that unifies with Elem.
selectchk/4	Semi-deterministic version of select/4.
subset/2	True if all elements of SubSet belong to Set as well.
subtract/3	Delete all elements in Delete from Set.
sum_list/2	Sum is the result of adding all numbers in List.
union/3	True if Set3 unifies with the union of the lists Set1 and Set2.

F.2.27 library(main)

argv_options/3	Parse command line arguments.
argv_options/4	As argv_options/3 in __guided__ mode, Currently this version allows parsing argume
argv_usage/1	Use print_message/2 to print a usage message at Level.
cli_enable_development_system/0	Re-enable the development environment.
cli_parse_debug_options/2	Parse certain commandline options for debugging and development purposes.
main/0	Call main/1 using the passed command-line arguments.

F.2.28 library(occurs)

contains_term/2	Succeeds if Sub is contained in Term (=, deterministically).
contains_var/2	Succeeds if Sub is contained in Term (==, deterministically).
free_of_term/2	Succeeds of Sub does not unify to any subterm of Term.
free_of_var/2	Succeeds of Sub is not equal (==) to any subterm of Term.
occurrences_of_term/3	Count the number of SubTerms in Term.
occurrences_of_var/3	Count the number of SubTerms in Term.
sub_term/2	Generates (on backtracking) all subterms of Term.
sub_term_shared_variables/3	If Sub is a sub term of Term, Vars is bound to the list of variables in Sub that also appear
sub_var/2	Generates (on backtracking) all subterms (==) of Term.

F.2.29 library(option)

dict_options/2	Convert between an option list and a dictionary.
merge_options/3	Merge two option lists.
meta_options/3	Perform meta-expansion on options that are module-sensitive.
option/2	Get an Option from OptionList.
option/3	Get an Option from OptionList.
select_option/3	Get and remove Option from an option list.
select_option/4	Get and remove Option with default value.

F.2.30 library(optparse)

opt_arguments/3	Extract commandline options according to a specification.
opt_help/2	True when Help is a help string synthesized from OptsSpec.
opt_parse/4	Equivalent to opt_parse(OptsSpec, ApplArgs, Opts, PositionalArgs, `[]`).
opt_parse/5	Parse the arguments Args (as list of atoms) according to OptsSpec.
parse_type/3	Hook to parse option text Codes to an object of type Type.

F.2.31 library(ordsets)

is_ordset/1	True if Term is an ordered set.
list_to_ord_set/2	Transform a list into an ordered set.
ord_add_element/3	Insert an element into the set.
ord_del_element/3	Delete an element from an ordered set.
ord_disjoint/2	True if Set1 and Set2 have no common elements.
ord_empty/1	True when List is the empty ordered set.
ord_intersect/2	True if both ordered sets have a non-empty intersection.
ord_intersect/3	Intersection holds the common elements of Set1 and Set2.
ord_intersection/2	Intersection of a powerset.
ord_intersection/3	Intersection holds the common elements of Set1 and Set2.
ord_intersection/4	Intersection and difference between two ordered sets.
ord_memberchk/2	True if Element is a member of OrdSet, compared using ==.
ord_selectchk/3	Selectchk/3, specialised for ordered sets.
ord_seteq/2	True if Set1 and Set2 have the same elements.
ord_subset/2	Is true if all elements of Sub are in Super.
ord_subtract/3	Diff is the set holding all elements of InOSet that are not in NotInOSet.

ord_symdiff/3	Is true when Difference is the symmetric difference of Set1 and Set2.
ord_union/2	True if Union is the union of all elements in the superset SetOfSets.
ord_union/3	Union is the union of Set1 and Set2.
ord_union/4	True iff ord_union(Set1, Set2, Union) and ord_subtract(Set2, Set1, New).

F.2.32 library(persistency)

current_persistent_predicate/1	True if PI is a predicate that provides access to the persistent database DB.
db_assert/1	Assert Term into the database and record it for persistency.
db_attach/2	Use File as persistent database for the calling module.
db_attached/1	True if the context module attached to the persistent database File.
db_detach/0	Detach persistency from the calling module and delete all persistent clauses from the Pr
db_retract/1	Retract terms from the database one-by-one.
db_retractall/1	Retract all matching facts and do the same in the database.
db_sync/1	Synchronise database with the associated file.
db_sync_all/1	Sync all registered databases.
persistent/1	Declare dynamic database terms.

F.2.33 library(portraytext)

is_text_code/1	Multifile hook that can be used to extend the set of character codes that is recognised as likely text.
portray_text/1	Switch portraying on or off.
set_portray_text/2	Set options for portraying.
set_portray_text/3	Set options for portraying.

F.2.34 library(predicate_options)

assert_predicate_options/4	As predicate_options(:PI, +Arg, +Options).
check_predicate_option/3	Verify predicate options at runtime.
check_predicate_options/0	Analyse loaded program for erroneous options.
current_option_arg/2	True when Arg of PI processes predicate options.
current_predicate_option/3	True when Arg of PI processes Option.
current_predicate_options/3	True when Options is the current active option declaration for PI on Arg.
derive_predicate_options/0	Derive new predicate option declarations.
derived_predicate_options/1	Derive predicate option declarations for a module.
derived_predicate_options/3	Derive option arguments using static analysis.
predicate_options/3	Declare that the predicate PI processes options on Arg.
retractall_predicate_options/0	Remove all dynamically (derived) predicate options.

F.2.35 library(prologdebug)

debugging/0	Report current status of the debugger.
exception_hook/4	Trap exceptions and consider whether or not to start the tracer.
nospy/1	Set/clear spy-points.
nospyall/0	Set/clear spy-points.
notrap/1	Install a trap on error(Formal, Context) exceptions that unify with Exception.

spy/1	Set/clear spy-points.
trap/1	Install a trap on error(Formal, Context) exceptions that unify with Exception.

F.2.36 library(prologjiti)

jiti_list/0	List the JITI (Just In Time Indexes) of selected predicates.
jiti_list/1	List the JITI (Just In Time Indexes) of selected predicates.

F.2.37 library(prologpack)

atom_version/2	Translate between atomic version representation and term representation.
environment/2	Hook to define the environment for building packs.
pack_attach/2	Attach a single package in Dir.
pack_info/1	Print more detailed information about Pack.
pack_install/1	Install a package.
pack_install/2	Install package Name.
pack_list/1	Query package server and installed packages and display results.
pack_list_installed/0	List currently installed packages.
pack_property/2	True when Property is a property of an installed Pack.
pack_rebuild/0	Rebuild foreign components of all packages.
pack_rebuild/1	Rebuilt possible foreign components of Pack.
pack_remove/1	Remove the indicated package.
pack_search/1	Query package server and installed packages and display results.
pack_upgrade/1	Try to upgrade the package Pack.
pack_url_file/2	True if File is a unique id for the referenced pack and version.
ssl_verify/5	Currently we accept all certificates.

F.2.38 library(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

F.2.39 library(pairs)

group_pairs_by_key/2	Group values with equivalent (==/2) consecutive keys.
map_list_to_pairs/3	Create a Key-Value list by mapping each element of List.
pairs_keys/2	Remove the values from a list of Key-Value pairs.
pairs_keys_values/3	True if Keys holds the keys of Pairs and Values the values.
pairs_values/2	Remove the keys from a list of Key-Value pairs.

transpose_pairs/2	Swap Key-Value to Value-Key.

F.2.40 library(pio)

library(pure_input)

phrase_from_file/2	Process the content of File using the DCG rule Grammar.
phrase_from_file/3	As phrase_from_file/2, providing additional Options.
phrase_from_stream/2	Run Grammer against the character codes on Stream.
stream_to_lazy_list/2	Create a lazy list representing the character codes in Stream.
lazy_list_character_count//1	True when CharCount is the current character count in the Lazy list.
lazy_list_location//1	Determine current (error) location in a lazy list.
syntax_error//1	Throw the syntax error Error at the current location of the input.

F.2.41 library(random)

getrand/1	Query/set the state of the random generator.
maybe/0	Succeed/fail with equal probability (variant of maybe/1).
maybe/1	Succeed with probability P, fail with probability 1-P.
maybe/2	Succeed with probability K/N (variant of maybe/1).
random/1	Binds R to a new random float in the _open_ interval (0.0,1.0).
random/3	Generate a random integer or float in a range.
random_between/3	Binds R to a random integer in [L,U] (i.e., including both L and U).
random_member/2	X is a random member of List.
random_numlist/4	Unify List with an ascending list of integers between L and U (inclusive).
random_perm2/4	Does X=A,Y=B or X=B,Y=A with equal probability.
random_permutation/2	Permutation is a random permutation of List.
random_select/3	Randomly select or insert an element.
random_subseq/3	Selects a random subsequence Subseq of List, with Complement containing all elements of Lis
randseq/3	S is a list of K unique random integers in the range 1..N.
randset/3	S is a sorted list of K unique random integers in the range 1..N.
setrand/1	Query/set the state of the random generator.

F.2.42 library(readutil)

read_file_to_codes/3	Read the file Spec into a list of Codes.
read_file_to_string/3	Read the file Spec into a the string String.
read_file_to_terms/3	Read the file Spec into a list of terms.
read_line_to_codes/2	Read the next line of input from Stream.
read_line_to_codes/3	Difference-list version to read an input line to a list of character codes.
read_line_to_string/2	Read the next line from Stream into String.
read_stream_to_codes/2	Read input from Stream to a list of character codes.
read_stream_to_codes/3	Read input from Stream to a list of character codes.

F.2.43 library(record)

record/1	Define named fields in a term

F.2.44 library(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

F.2.45 library(settings)

F.2.46 library(simplex)

assignment/2	Solve assignment problem
constraint/3	Add linear constraint to state
constraint/4	Add named linear constraint to state
constraint_add/4	Extend a named constraint
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

F.2.47 library(terms)

F.2.48 library(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_ugraph/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

F.2.49 library(url)

file_name_to_url/2	Translate between a filename and a file:// URL.
global_url/3	Translate a possibly relative URL into an absolute one.
http_location/2	Construct or analyze an HTTP location.
is_absolute_url/1	True if URL is an absolute URL.
parse_url/2	Construct or analyse a URL.
parse_url/3	Similar to parse_url/2 for relative URLs.
parse_url_search/2	Construct or analyze an HTTP search specification.
set_url_encoding/2	Query and set the encoding for URLs.
url_iri/2	Convert between a URL, encoding in US-ASCII and an IRI.
www_form_encode/2	En/decode to/from application/x-www-form-encoded.

F.2.50 library(www_browser)

www_open_url/1	Open a web-page in a browser

F.2.51 library(solution_sequences)

call_nth/2	True when Goal succeeded for the Nth time.
distinct/1	True if Goal is true and no previous solution of Goal bound Witness to the same value.
distinct/2	True if Goal is true and no previous solution of Goal bound Witness to the same value.
group_by/4	Group bindings of Template that have the same value for By.
limit/2	Limit the number of solutions.
offset/2	Ignore the first Count solutions.
order_by/2	Order solutions according to Spec.
reduced/1	Similar to distinct/1, but does not guarantee unique results in return for using a limited amount of memory.
reduced/3	Similar to distinct/1, but does not guarantee unique results in return for using a limited amount of memory.

F.2.52 library(thread)

call_in_thread/2	Run Goal as an interrupt in the context of Thread.
concurrent/3	Run Goals in parallel using N threads.
concurrent_and/2	Concurrent version of ‘(Generator,Test)‘.
concurrent_and/3	Concurrent version of ‘(Generator,Test)‘.
concurrent_forall/2	True when Action is true for all solutions of Generate.
concurrent_forall/3	True when Action is true for all solutions of Generate.
concurrent_maplist/2	Concurrent version of maplist/2.
concurrent_maplist/3	Concurrent version of maplist/2.
concurrent_maplist/4	Concurrent version of maplist/2.

first_solution/3	Try alternative solvers concurrently, returning the first answer.

F.2.53 library(thread_pool)

create_pool/1	Hook to create a thread pool lazily.
current_thread_pool/1	True if Name refers to a defined thread pool.
thread_create_in_pool/4	Create a thread in Pool.
thread_pool_create/3	Create a pool of threads.
thread_pool_destroy/1	Destroy the thread pool named Name.
thread_pool_property/2	True if Property is a property of thread pool Name.
worker_exitted/3	It is possible that '__thread_pool_manager' no longer exists while closing down the process bec

F.2.54 library(varnumbers)

max_var_number/3	True when Max is the max of Start and the highest numbered $VAR(N) term.
numbervars/1	Number variables in Term using $VAR(N).
varnumbers/2	Inverse of numbervars/1.
varnumbers/3	Inverse of numbervars/3.
varnumbers_names/3	If Term is a term with numbered and named variables using the reserved term '$VAR'(X), Copy

F.2.55 library(yall)

//2	Shorthand for 'Free/[]>>Lambda'.
//3	Shorthand for 'Free/[]>>Lambda'.
//4	Shorthand for 'Free/[]>>Lambda'.
//5	Shorthand for 'Free/[]>>Lambda'.
//6	Shorthand for 'Free/[]>>Lambda'.
//7	Shorthand for 'Free/[]>>Lambda'.
//8	Shorthand for 'Free/[]>>Lambda'.
//9	Shorthand for 'Free/[]>>Lambda'.
>>/2	Calls a copy of Lambda.
>>/3	Calls a copy of Lambda.
>>/4	Calls a copy of Lambda.
>>/5	Calls a copy of Lambda.
>>/6	Calls a copy of Lambda.
>>/7	Calls a copy of Lambda.
>>/8	Calls a copy of Lambda.
>>/9	Calls a copy of Lambda.
is_lambda/1	True if Term is a valid Lambda expression.
lambda_calls/2	Goal is the goal called if call/N is applied to LambdaExpression, where ExtraArgs are the additional ar
lambda_calls/3	Goal is the goal called if call/N is applied to LambdaExpression, where ExtraArgs are the additional ar

F.3 Arithmetic Functions

*/2	Multiplication
**/2	Power function
+/1	Unary plus (No-op)
+/2	Addition
-/1	Unary minus
-/2	Subtraction
//2	Division
///2	Integer division
/\/2	Bitwise and
<</2	Bitwise left shift
>>/2	Bitwise right shift
./2	List of one character: character code
\/1	Bitwise negation
\//2	Bitwise or
^/2	Power function
abs/1	Absolute value
acos/1	Inverse (arc) cosine
acosh/1	Inverse hyperbolic cosine
asin/1	Inverse (arc) sine
asinh/1	Inverse (arc) sine
atan/1	Inverse hyperbolic sine
atan/2	Rectangular to polar conversion
atanh/1	Inverse hyperbolic tangent
atan2/2	Rectangular to polar conversion
ceil/1	Smallest integer larger than arg
ceiling/1	Smallest integer larger than arg
cos/1	Cosine
cosh/1	Hyperbolic cosine
copysign/2	Apply sign of N2 to N1
cputime/0	Get CPU time
denominator/1	Denominator of a rational number (N/D)
div/2	Integer division
e/0	Mathematical constant
erf/1	Gauss error function
erfc/1	Complementary error function
epsilon/0	Floating point precision
eval/1	Evaluate term as expression
exp/1	Exponent (base e)
float/1	Explicitly convert to float
float_fractional_part/1	Fractional part of a float
float_integer_part/1	Integer part of a float
floor/1	Largest integer below argument
gcd/2	Greatest common divisor
getbit/2	Get bit at index from large integer

inf/0	Positive infinity
integer/1	Round to nearest integer
lgamma/1	Log of gamma function
log/1	Natural logarithm
log10/1	10 base logarithm
lcm/2	Least Common Multiple
lsb/1	Least significant bit
max/2	Maximum of two numbers
min/2	Minimum of two numbers
msb/1	Most significant bit
mod/2	Remainder of division
nan/0	Not a Number (NaN)
nexttoward/2	Next float in some direction
numerator/1	Numerator of a rational number (N/D)
powm/3	Integer exponent and modulo
random/1	Generate random number
random_float/0	Generate random number
rational/1	Convert to rational number
rationalize/1	Convert to rational number
rdiv/2	Ration number division
rem/2	Remainder of division
round/1	Round to nearest integer
roundtoward/2	Float arithmetic with specified rounding
truncate/1	Truncate float to integer
pi/0	Mathematical constant
popcount/1	Count 1s in a bitvector
sign/1	Extract sign of value
sin/1	Sine
sinh/1	Hyperbolic sine
sqrt/1	Square root
tan/1	Tangent
tanh/1	Hyperbolic tangent
xor/2	Bitwise exclusive or

F.4 Operators

$	1	fx	Bind top-level variable
^	200	xfy	Existential qualification
^	200	xfy	Arithmetic function
mod	300	xfx	Arithmetic function
*	400	yfx	Arithmetic function
/	400	yfx	Arithmetic function
//	400	yfx	Arithmetic function
<<	400	yfx	Arithmetic function
>>	400	yfx	Arithmetic function
xor	400	yfx	Arithmetic function
+	500	fx	Arithmetic function
-	500	fx	Arithmetic function
?	500	fx	XPCE: obtainer
\	500	fx	Arithmetic function
+	500	yfx	Arithmetic function
-	500	yfx	Arithmetic function
/\	500	yfx	Arithmetic function
\/	500	yfx	Arithmetic function
:	600	xfy	module:term separator
<	700	xfx	Predicate
=	700	xfx	Predicate
=..	700	xfx	Predicate
=:=	700	xfx	Predicate
<	700	xfx	Predicate
==	700	xfx	Predicate
=@=	700	xfx	Predicate
=\=	700	xfx	Predicate
>	700	xfx	Predicate
>=	700	xfx	Predicate
@<	700	xfx	Predicate
@=<	700	xfx	Predicate
@>	700	xfx	Predicate
@>=	700	xfx	Predicate
as	700	xfx	Predicate
is	700	xfx	Predicate
\=	700	xfx	Predicate
\==	700	xfx	Predicate
=@=	700	xfx	Predicate
not	900	fy	Predicate
\+	900	fy	Predicate
,	1000	xfy	Predicate
->	1050	xfy	Predicate
*->	1050	xfy	Predicate
;	1100	xfy	Predicate

`|`	1105	xfy	DCG disjunction
discontiguous	1150	fx	Directive
dynamic	1150	fx	Directive
module_transparent	1150	fx	Directive
meta_predicate	1150	fx	Head
multifile	1150	fx	Directive
thread_local	1150	fx	Directive
volatile	1150	fx	Directive
initialization	1150	fx	Directive
`:-`	1200	fx	Introduces a directive
`?-`	1200	fx	Introduces a directive
`-->`	1200	xfx	DCGrammar: rewrite
`:-`	1200	xfx	head `:-` body. separator

Bibliography

[Bowen *et al.*, 1983] D. L. Bowen, L. M. Byrd, and WF. Clocksin. A portable Prolog compiler. In L. M. Pereira, editor, *Proceedings of the Logic 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.

[Byrd, 1980] L. Byrd. Understanding the control flow of Prolog programs. *Logic Programming Workshop*, 1980.

[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.

[Desouter *et al.*, 2015] Benoit Desouter, Marko van Dooren, and Tom Schrijvers. Tabling as a library with delimited control. *TPLP*, 15(4-5):419–433, 2015.

[Frühwirth,] T. Frühwirth. Thom Fruehwirth's constraint handling rules website. http://www.constraint-handling-rules.org.

[Frühwirth, 2009] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[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.

[Grosof & Swift, 2013] Benjamin Nathan Grosof and Terrance Swift. Radial Restraint: A semantically clean approach to bounded rationality for logic programs. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013.

[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, 1992] Christian Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In *PLILP*, volume 631, pages 260–268. Springer-Verlag, 1992. LNCS 631.

[Kernighan & Ritchie, 1978] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Neumerkel, 1993] Ulrich Neumerkel. The binary WAM, a simplified Prolog engine. Technical report, Technische Universität Wien, 1993. http://www.complang.tuwien.ac.at/ulrich/papers/PDF/binwam-nov93.pdf.

[O'Keefe, 1990] R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Massachussetts, 1990.

[Pereira, 1986] F. Pereira. *C-Prolog User's Manual*. EdCaad, University of Edinburgh, 1986.

[Qui, 1997] AI International ltd., Berkhamsted, UK. *Quintus Prolog, User Guide and Reference Manual*, 1997.

[Sagonas & Swift, 1998] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Program. Lang. Syst.*, 20(3):586–634, 1998.

[Sagonas *et al.*, 2000] Konstantinos Sagonas, Terrance Swift, and David S. Warren. An abstract machine for efficiently computing queries to well-founded models. *The Journal of Logic Programming*, 45(1):1 – 41, 2000.

[Schimpf, 2002] Joachim Schimpf. Logical loops. In PeterJ. Stuckey, editor, *Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 224–238. Springer Berlin Heidelberg, 2002.

[Schrijvers *et al.*, 2013] Tom Schrijvers, Bart Demoen, Benoit Desouter, and Jan Wielemaker. Delimited continuations for Prolog. *TPLP*, 13(4-5):533–546, 2013.

[Sterling & Shapiro, 1986] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 1986.

[Swift, 2014] Terrance Swift. Incremental tabling in support of knowledge representation and reasoning. *TPLP*, 14(4-5):553–567, 2014.

[Tarau, 2011] Paul Tarau. Coordination and concurrency in multi-engine Prolog. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6721 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2011.

[Triska, 2016] Markus Triska. The Boolean constraint solver of SWI-Prolog: System description. In *FLOPS*, volume 9613 of *LNCS*, pages 45–61, 2016. https://www.metalevel.at/swiclpb.pdf.

[Wielemaker, 2013] Jan Wielemaker. Extending the logical update view with transaction support. *CoRR*, abs/1301.7669, 2013.

[Zhou, 2010] Neng-Fa Zhou. Declarative loops and list comprehensions for Prolog. http://www.sci.brooklyn.cuny.edu/ zhou/papers/loops.pdf, Jan 2010.

Index