University of Amsterdam

Dept. of Social Science Informatics (SWI)
Roeterstraat 15, 1018 WB Amsterdam
The Netherlands
Tel. (+31) 20 5256786

SWI-Prolog 2.1

Reference Manual

Updated for version 2.1.4, August 1995

Jan Wielemaker

jan@swi.psy.uva.nl

SWI-Prolog is a Prolog implementation based on a subset of the WAM (Warren Abstract Machine [Warren, 1983]). SWI-Prolog has been designed and implemented such that it can easily be modified for experiments with logic programming and the relation between logic programming and other programming paradigms (such as the object oriented XPCE environment [Anjewierden & Wielemaker, 1989]). SWI-Prolog has a rich set of built-in predicates and reasonable performance, which makes it possible to develop substantial applications in it. The current version offers a module system, garbage collection and an interface to the C language.

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

Contents

Index

Chapter 1

Introduction

1.1 SWI-Prolog

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

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

SWI-Prolog extends the minimal set of instructions described in [Bowen & Byrd, 1983] to improve performance. While extending this set care has been taken to maintain the advantages of decompilation and tracing of compiled code. The extensions include specialised instructions for unification, predicate invocation, some frequently used built-in predicates, arithmetic, and control (`;/2`, `|/2`), if-then (`->/2`) and not (`\+/1`).

This manual does not describe the full syntax and semantics of SWI-Prolog, nor how one should write a program in Prolog. These subjects have been described extensively in the literature. See [Bratko, 1986], [Sterling & Shapiro, 1986], and [Clocksin & Mellish, 1981]. For more advanced Prolog material see [O'Keefe, 1990]. Syntax and standard operator declarations confirm to the 'Edinburgh standard'. Most built in predicates are compatible with those described in [Clocksin & Mellish, 1981]. SWI-Prolog also offers a number of primitive predicates compatible with Quintus Prolog[1], [Qui, 1987] and BIM Prolog[2] [BIM, 1989].

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

1.2 Status

This manual describes version 2.1 of SWI-Prolog. SWI-Prolog has been used now for several years. The application range includes Prolog course material, meta-interpreters, simulation of parallel Prolog, learning systems, natural language processing and two large workbenches for knowledge engineering. Although we experienced rather obvious and critical bugs can remain unnoticed for a remarkable long period, we can assume the basic Prolog system is fairly stable. Bugs can be expected in unfrequently used builtin predicates.

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

1.3 Should you be Using SWI-Prolog?

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

- *SWI-Prolog is not supported*
  Although I usually fix bugs shortly after a bug report arrives, I cannot promise anything. Now that the sources are provided, you can always dig into them yourself.
- *Memory requirements and performance are your first concerns*
  A number of commercial compilers are more keen on memory and performance than SWI-Prolog. I do not wish to offer some of the nice features of the system, nor its portability to compete on raw performance.
- *You need features not offered by SWI-Prolog*
  In this case you may wish to give me suggestions for extensions. If you have great plans, please contact me (you might have to implement them yourself however).

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

- *Nice environment*
  This includes 'Do What I Mean', automatic completion of atom names, history mechanism and a tracer that operates on single key-strokes. Interfaces to standard Unix editors are provided, as well as a facility to maintain programs (see `make/0`).
- *Very fast compiler*
  The compiler handles about 100K bytes per second on a SPARC-II processor.
- *Transparent compiled code*
  SWI-Prolog compiled code can be treated just as interpreted code: you can list it, trace it, assert to or retract from it, etc. This implies you do not have to decide beforehand whether a module should be loaded for debugging or not. Also, performance is much better than the performance of most interpreters.
- *Profiling*
  SWI-Prolog offers tools for performance analysis, which can be very useful to optimise programs. Unless you are very familiar with Prolog and Prolog performance considerations this might be more helpful than a better compiler without these facilities.

- *Flexibility*
  SWI-Prolog allows for easy and flexible integration with C, both Prolog calling C functions as C calling Prolog predicates. SWI-Prolog is provided in source form, which implies SWI-Prolog can be linked in with another package. Command line options and predicates to obtain information from the system and feedback into the system are provided.

- *Integration with PCE*
  SWI-Prolog offers a tight integration to the Object Oriented Package for User Interface Development, called PCE [Anjewierden & Wielemaker, 1989]. PCE is now also available for the X Window System.

1.4 Graphics

SWI-Prolog has no provisions for graphical applications itself. The standard graphical environment to be used with SWI-Prolog is XPCE. XPCE/SWI-Prolog provides an interactive graphical develpment platform that is portable over many Unix/X11 implementations as well as MS-Windows (3.1) and Windows-NT (3.5).

XPCE is a licenced product. A fully functional free demo version for PC/Linux is available from `ftp://swi.psy.uva.nl/pub/xpce/linux`, for more information see URL

`http://www.swi.psy.uva.nl/projects/xpce/home.html`

or contact xpce-request@swi.psy.uva.nl.

1.5 Version 1.5 Release Notes

There are not many changes between version 1.4 and 1.5. The C-sources have been cleaned and comments have been updated. The stack memory management based on using the MMU has been changed to run on a number of System-V Unix systems offering shared memory. Handling dates has been changed. All functions handling dates now return a floating point number, expressing the time in seconds since January 1, 1970. A predicate `convert_time/8` is available to get the year, month, etc. The predicate `time/6` has been deleted. `get_time/1` and `convert_time/8` together do the same.

From version 1.5, the system is distributed in source form, rather than in object form as used with previous releases. This allows users to port SWI-Prolog to new machines, extend and improve the system. If you want your changes to be incorporated in the next release, please indicate all changes using a C-preprocessor flag and send complete source files back to me. Difference listings are of no use, as I generally won't have exactly the same version around.

1.6 Version 1.6 Release Notes

Version 1.6 is completely compatible with version 1.5. Some new features have been added, the system has been ported to various new platforms and there is a provisional interface to GNU-Emacs. This interface will be improved and documented later.

The WAM virtual-machine interpreter has been modified to use GCC-2's support for threaded code.

From version 1.6, the sources are now versioned using the CVS version control system.

1.7 Version 1.7 Release Notes

Version 1.7 intergrates the GNU-readline library, offering powerful history and command-line editing both using EMACS and vi key-bindings.

1.8 Version 1.8 Release Notes

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

1.9 Version 1.9 Release Notes

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

- *Redefinition of system predicates*
  Redefinition of system predicates was allowed silently in older versions. Version 1.9 only allows it if the new definition is headed by a :- `redefine_system_predicate/1` directive.
- *'Answer' reuse*
  The toplevel maintains a table of bindings returned by toplevel goals and allows for reuse of these bindings by prefixing the variables with the $ sign. See section 2.5.
- *Better sourcecode administration*
  Allows for proper updating of multifile predicates and finding the sources of individual clauses.

1.10 Version 2.0 Release Notes

Version 2.0 is first of all a freeze of all the features added to the various 1.9.x releases. Version 2.0.6 for PC has moved from the WATCOM C 32-bit windows extender to Windows-NT and runs under Windows 3.1 using the Win32s NT emulator.

New features offered:

- *32-bit Virtual Machine*
  Removes various limits and improves performance.
- *Inline foreign functions*
  'Simple' foreign predicates no longer build a Prolog stack-frame, but are directly called from the VM. Notably provides a speedup for the test predicates such as `var/1`, etc.
- *Various compatibility improvements*

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

Version 2.0.6 offers a few incompatibilities:

- `retractall/1`
  In previous releases, the definition of `retractall/1` was:

```
retractall(Term) :-
        retract(Term),
        fail.
retractall(_).
```

  As from version 2.0.6, `retractall/1` is implemented as a deterministic foreign predicate compatible with Quintus Prolog. It behaves as:

```
retractall(Head) :-
        retract(Head),
        fail.
retractall(Head) :-
        retract((Head :- _)),
        fail.
retractall(_).
```

  I.e. the definition behaves the same when handling predicates consisting of facts. Clauses with a non-true body will be retracted if their head matches.

- *Foreign interface types*
  All foreign interface types now have names ending in `_t` to lessen the chance for conflicts. `term`, `atomic`, `functor` and `module` have `#define`'s for backward compatibility.

- *PL_register_foreign()*
  The attributes is now a bitwise or of the attribute flags rather than a 0 terminated list. This has no consequences for predicates that have no attributes (99% of them), while predicates with just one attribute will generate a compiler warning, but work properly otherwise. Predicates with more than one attributes must be changed.

- *PL_dispatch_events*
  This pointer is replaced by PL_dispatch_hook(). A function was necessary for the Windows-NT .DLL interface.

1.11 Version 2.1 Release Notes

In addition to several bugfixes, the 2.1 versions provide some new features:

- `setarg/3`
  A new predicate `setarg/3` for extra-logical (destructiveZ) assignment to arguments of terms is provided.

- *Modified* `keysort/2`
  `keysort/2` is now stable with regard to multiple values on the same key. Makes this predicate compatible with SICStus and Quintus.
- *Modified grammar rule expansion*
  DCG translation of free variables now calls `phrase/3`, which has been changed slightly to deal with 'un-parsing'. Modification is probably not complete, but it fixes some problems encountered by Michael Böhlen.
- *Exception handling*
  The top of the runtime stack are automatically dumped on floating point exceptions.
- *Foreign interface*
  Added facilities to allow for embedding SWI-Prolog in C applications.

1.12 Acknowledgements

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

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

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

Chapter 2

Overview

2.1 Starting SWI-Prolog from the Unix Shell

It is advised to install SWI-Prolog as 'pl' in the local binary directory. SWI-Prolog can then be started from the Unix shell by typing '`pl`'. The system will boot from the system's default boot file, perform the necessary initialisations and then enter the interactive top level.

After the necessary system initialisation the system consults (see `consult/1`) the user's initialisation file. This initialisation file should be named '*verb.plrc*'[1] and reside either in the current directory or in the user's home directory. If both exist the initialisation file from the current directory is loaded. The name of the initialisation file can be changed with the '`-f file`' option. After loading the initialisation file SWI-Prolog executes a user initialisation goal. The default goal is a system predicate that prints the banner message. The default can be modified with the '`-g goal`' option. Next the toplevel goal is started. Default is the interactive Prolog loop (see `prolog/0`). The user can overwrite this default with the '`-t toplevel`' option.

2.1.1 Command Line Options

The full set of command line options is given below:

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

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

`-Lsize`
: Give local stack size in K bytes (200 K default). Note that there is no space between the size option and its argument. For machines with dynamic stack allocation this flag sets the maximum value to which the stack is allowed to grow (2 Mbytes default). A maximum is useful to stop buggy programs from claiming all memory resources. `-L0` sets the limit to the highest possible value.

`-Gsize`
: Give global stack size in K bytes (100 K default). For machines with dynamic stack allocation the default is 4 Mbytes. See `-L` for more details.

[1] Windows: '`_plrc`' as files cannot be named '`.plrc`'.

-Tsize
Give trail stack size in K bytes (50 K default). For machines with dynamic stack allocation the default is 4 Mbytes. See `-L` for more details.

-Asize
Give argument stack size in K bytes (5 K default). For machines with dynamic stack allocation the default is 1 Mbytes. See `-L` for more details.

-c file ...
Compile files into an 'intermediate code file'. See section 2.7.

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

-O
Optimised compilation. See `please/3`.

-f file
Use *file* as initialisation file instead of '`.plrc`'. '`-f none`' stops SWI-Prolog from searching for an initialisation file.

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

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

+/-tty
Switches tty control (using ioctl(2)) on (`+tty`) or off (`-tty`). Normally tty control is switched on. This default depends on the installation. You may wish to switch tty control off if Prolog is used from an editor such as Emacs. If switched off `get_single_char/1` and the tracer will wait for a return.

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

-r restorefile
Restore a state created by `save_program/[1,2] or save/[1,2]` using the new-style saved-states. Equivalent to `restore(restorefile)` from Prolog.

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

--
Stops scanning for more arguments, so you can pass arguments for your application after this one.

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

-b initfile ... -c file ...
Boot compilation. `initfile ...` are compiled by the C-written bootstrap compiler,

`file ...` by the normal Prolog compiler. System maintenance only.

`-d level`
Set debug level to *level*. System maintenance only.

2.2 GNU Emacs Interface

A provisional interface to GNU-Emacs has been included since version 1.6 of SWI-Prolog. The interface is based on the freely distributed interface delivered with Quintus Prolog. When running Prolog as an inferior process under GNU-Emacs, there is support for finding predicate definitions, completing atoms, finding the locations of compilation-warnings and many more. For details, see the files `pl/lisp/README` and `pl/lisp/swi-prolog.el`.

2.3 Online Help

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

help
Equivalent to `help(help/1)`.

help(*+What*)
Show specified part of the manual. *What* is one of:

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

Examples

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

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

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

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

explain(*+ToExplain, -Explanation*)
Unify *Explanation* with an explanation for *ToExplain*. Backtracing yields further explanations.

2.4 Query Substitutions

SWI-Prolog offers a query substitution mechanism similar to that of Unix csh (csh(1)), called 'history'. It allows the user to compose new queries from those typed before and remembered by the system. It also allows to correct queries and syntax errors. SWI-Prolog does not offer the Unix csh capabilities to include arguments. This is omitted as it is unclear how the first, second, etc. argument should be defined.[2]

The available history commands are shown in table 2.1. Figure 2.1 gives some examples.

!!.	Repeat last query
!nr.	Repeat query numbered <nr>
!str.	Repeat last query starting with <str>
!?str.	Repeat last query holding <str>
^old^new.	Substitute <old> into <new> of last query
!nr^old^new.	Substitute in query numbered <nr>
!str^old^new.	Substitute in query starting with <str>
!?str^old^new.	Substitute in query holding <str>
h.	Show history list
!h.	Show this list

Table 2.1: History commands

2.4.1 Limitations of the History System

When in top level SWI-Prolog reads the user's queries using history_read/6 rather than read/1. This predicate first reads the current input stream up to a full stop. While doing so it maps all contiguous blank space onto a single space and deletes /* ... */ and % ... <cr> comments. Parts between double quotes (") or single quotes (') are left unaltered. Note that a Prolog full stop consists of a 'non-symbol' character, followed by a period (.), followed by a blank character. 'Symbol' characters are: #$&*+-./:<=>?@^'~. A single quote immediately preceded by a digit (0-9) is considered part of the <digit>'<digit>... (e.g. 2'101; binary number 101) sequence.

After this initial parsing the result is first checked for the special ^old^new. construction. If this fails the string is checked for all occurrences of the !, followed by a !, ?, a digit, a letter or an underscore. These special sequences are analysed and the appropriate substitution from the history list is made.

From the above it follows that it is hard or impossible to correct quotation with single or double quotes, comment delimiters and spacing.

2.5 Reuse of toplevel bindings

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

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

```
/u4/staff/jan/.plrc consulted, 0.066667 seconds, 591 bytes
Welcome to SWI-Prolog (version 2.1.3, July 1995)
Copyright (c) 1990-1995, University of Amsterdam. All rights reserved.

1 ?- append("Hello ", "World", L).

L = [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]

Yes
2 ?- !!, writef('L = %s\n', [L]).
append("Hello ", "World", L), writef('L = %s\n', [L]).
L = Hello World

L = [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]

Yes
3 ?- sublist(integer, [3, f, 3.4], L).

L = [3]

Yes
4 ?- ^integer^number.
sublist(number, [3, f, 3.4], L).

L = [3, 3.400000]

Yes
5 ?- h.
    1   append("Hello ", "World", L).
    2   append("Hello ", "World", L), writef('L = %s\n', [L]).
    3   sublist(integer, [3, f, 3.4], L).
    4   sublist(number, [3, f, 3.4], L).

5 ?- !2^World^Universe.
append("Hello ", "Universe", L), writef('L = %s\n', [L]).
L = Hello Universe

L = [72, 101, 108, 108, 111, 32, 85, 110, 105, 118, 101, 114, 115, 101]

Yes
6 ?- halt.
```

Figure 2.1: Some examples of the history facility

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

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

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

Figure 2.2: Reusing toplevel bindings

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

```
6 ?- X = statistics.

X = statistics

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

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

2.6 Overview of the Debugger

SWI-Prolog has a standard 4-port tracer [Clocksin & Mellish, 1981] with an optional fifth port. This fifth port, called *unify* allows the user to inspect the result after unification of the head. The ports are called *call*, *exit*, *redo*, *fail* and *unify*. The tracer is started by the `trace/0` command, when a spy point is reached and the system is in debugging mode (see `spy/1` and `debug/1`) or when an error is detected at run time. Note that in the interactive toplevel goal `trace/0` means "trace the next query". The tracer shows the port, displaying the port name, the current depth of the recursion and the goal. The goal is printed using the Prolog predicate `print/1` (default), `write/1` or `display/1`. An example using all five ports is shown in figure 2.3.

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

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

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

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

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

Figure 2.3: Example trace

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

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

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

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

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

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

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

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

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

`d` *Display* all
Write goals using the Prolog predicate `display/1`.

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

`f` *Fail* call, redo, exit
Force failure of the current goal

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

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

`i` *Ignore* call, redo, fail
Ignore the current goal, pretending it succeeded.

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

`n` *No debug* all
Continue execution in 'no debug' mode.

`p` *Print* all
Write goals using the Prolog predicate `print/1` (default).

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

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

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

`w` *Write* all
Write goals using the Prolog predicate `write/1`.

The ideal 4 port model as described in many Prolog books [Clocksin & Mellish, 1981] is not visible in many Prolog implementations because code optimisation removes part of the choice- and exit points. Backtrack points are not shown if either the goal succeeded deterministically or its alternatives were removed using the cut. When running in debug mode (`debug/0`) choice points are only destroyed when removed by the cut. In debug mode tail recursion optimisation is switched off.[3]

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

2.7 Compilation

Collections of SWI-Prolog source files can be compiled into an *intermediate code file*. An intermediate code file is a data file from which SWI-Prolog can be started. The command to compile a bundle of source files is:

```
pl [options] [-o output] -c file ...
```

The individual source files may include other files using the standard list notation, `consult/1`, `ensure_loaded/1` and `use_module/[1,2]`. When the `-o file` option is omitted a file named `a.out` is created that holds the intermediate code file.

Intermediate code files start with the BSD Unix magic code `#!` and are executable. This implies they can be started as a command:

```
sun% pl -o my_program -c ...
...
sun% my_program [options]
```

Alternatively, `my_program` can be started with

```
sun% pl -x my_program [options]
```

The following restrictions apply to source files that are to be compiled with '`-c`':

- `term_expansion/2` should not use `assert/1` and or `retract/1` other than for local computational purposes.
- Files can only be included by the standard include directives: [...], `consult/1`, `ensure_loaded/1` and `use_module/[1,2]`. User defined loading predicate invocations will not be compiled.

Directives are executed both when compiling the program and when loading the intermediate code file.

2.8 Environment Control

The current system defines 3 different mechanisms to query and/or set properties of the environment: `please/3`, `flag/3` and `feature/2` as well as a number of special purpose predicates of which `unknown/2`, `fileerrors/2` are examples. The ISO standard defines prolog_flag. It is likely that all these global features will be merged into a single in the future.

please(*+Key, -Old, +New*)

The predicate `please/3`[4] is a solution to avoid large numbers of environment control predicates. Later versions will support other environment control as now provided via the predicates `style_check/2`, `leash/1`, `unknown/2`, the tracer predicates, etc. These predicates are then moved into a library for backwards compatibility. The currently available options are:

[4] The idea comes from BIM_Prolog. The options supported by this predicate are not compatible with those for BIM_Prolog however.

optimise *on/off* (default: *off)*
Switch optimise mode for the compiler `on` or `off` (see also the command line option `-O`). Currently optimised compilation only implies compilation of arithmetic, making it fast, but invisible to the tracer. Later versions might imply various other optimisations such as incorporating a number of basic predicates in the virtual machine (`var/1`, `fail/0`, =/2, etc.) to gain speed at the cost of crippling the debugger. Also source level 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`).

autoload *on/off* (default: *on)*
If `on` autoloading of library functions is enabled. If `off` autoloading is disabled. See section 2.10.

verbose_autoload *on/off* (default: *off)*
If `on` 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 (e.g. where does this predicate come from?).

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

arch (*atom*)
Identifier for the hardware and operating system SWI-Prolog is running on. Used to determine the startup file as well as to select foreign files for the right architecture. See also `load_foreign/5`.

version (*atom*)
The version identifier is an atom consisting of 3 numeric values separated by dots (e.g. `1.9.2`).

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

pipe (*bool*)
If true, tell(pipe(command)), etc. are supported.

load_foreign (*bool*)
If true, `load_foreign/[2,5]` are implemented.

open_shared_object (*bool*)
If true, `open_shared_object/2` and friends are implemented, providing access to shared libraries (.so files). This requires the C-library functions dlopen() and friends as well as the configuration option `--with-dlopen`.

dynamic_stacks (*bool*)
If true, the system provides virtual memory based stack expansion. This makes stack-expansion safe when called from C and while running C-defined functions without these functions taking any precautions. Pure Prolog programs are not affected by this feature as the system will use stack-shifting to expand the stacks at runtime.

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

c_staticlibs (*atom*)
: On some machines, the SWI-Prolog executable is dynamically linked, but requires some libraries to be statically linked.

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

c_ldflags (*atom*)
: Special linker flags passed to link SWI-Prolog.

save (*bool*)
: If true, `save/[1,2]` is implemented.

save_program (*bool*)
: If true, `save_program/[1,2]` is implemented.

readline (*bool*)
: If true, SWI-Prolog is linked with the readline library. This is done by default if you have this library installed on your system.

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

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

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

min_integer (*integer*)
: Minimum integer value.

set_feature(*+Key, +Value*)
: Define a new feature or change its value. Both arguments must be atoms.

2.9 Saved States

2.9.1 Types of Saved States and Portability

On some architectures, SWI-Prolog allows for the creation of saved states. Currently, the C-sources define two alternative implementations for them. The old implementation[5] creates an entirely self-contained executable. It supports only the `save_program/[1,2]` predicates. The new implementation[6] creates a file that is read by the 'base' program. This file contains the data-area of the SWI-Prolog process and optionally the Prolog- and C stacks. This implementation supports `save_program/[1,2]`, `save/[1,2]` and `restore/1`. The state-file depends entirely on *exactly* the Unix program-file used to create it.

2.9.2 Save Predicates

save_program(*+NewProgram, +ListOfOptions*)
: Create a new executable wich will be named *NewProgram*. *ListOfOptions* is a list of *Key* = *Value*

[5] This implementation is based on GNU's unexec() function, creating a new executable from the running process. It is selected using the `O_STORE_PROGRAM` option in the machine-description file.

[6] This implementation is selected by the O_SAVE option in the machine-description file.

pairs that specify the default command line options that will be saved into the new program. If a default is not specified the default compiled into the currently running Prolog executable is used.[7] The available keys are given in table 2.2

Key	Option	Type	Description
local	**-L**	K-bytes	Size (Limit) of local stack
global	**-G**	K-bytes	Size (Limit) of global stack
trail	**-T**	K-bytes	Size (Limit) of trail stack
argument	**-A**	K-bytes	Size (Limit) of argument stack
goal	**-g**	atom	Initialisation goal
toplevel	**-t**	atom	Prolog toplevel goal
init_file	**-f**	atom	Personal initialisation file
tty	**+/−tty**	on/off	Use ioctl(2) calls

Table 2.2: Key = Value pairs for `save_program/2`

As the entire data image of the current process will be saved on the new executable it is desirable to keep this small. Notably the Prolog machine stacks should be kept small. The best way to do this is first to compile the program using the `-c` option. If this is not possible try to find the smallest possible stack sizes to compile the program. On machines with dynamic stack allocation the stacks are **not** written to file and so their size does not matter. Figure 2.4 shows a possible session. Note the use of 'initialise', which is supposed to be a predicate of the application doing time consuming initialisation.

The resulting program can be used for incremental compilation using `-c` or another `save_program/2`.

save_program(*+NewProgram***)**
Equivalent to '`save_program(NewProgram, [])`'.

save(*+State***)**
Writes the current status on the program, Prolog- and C-stacks to the file *State*. When this file is restored, `save/1` succeeds in the restored state and execution continues as illustrated by the following example:

```
1 ?- save(state), format('Hello World~n').
Hello World

Yes
2 ?- halt.
machine% ./state
Hello World

Yes
2 ?- h.
    1   save(state), format('Hello World~n').
2 ?-
```

The `save/1` predicate is normally used for debugging purposes. `save_program/[1,2]` is the preferred way to create a new program.

[7] These are NOT the defaults compiled into an intermediate code file created with pl -c ...