2.11 Syntax Notes

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

- `0'<char>`
  This construct is not accepted by all Prolog systems that claim to have Edinburgh compatible syntax. It describes the ASCII value of `<char>`. To test whether *C* is a lower case character one can use '`between(0'a, 0'z, C)`'.

- `/* ... /* .... */ ... */`
  The `/* ... */` comment statement can be nested. This is useful if some code with `/* ... */` comment statements in it should be commented out.

2.12 System Limits

2.12.1 Limits on Memory Areas

SWI-Prolog has a number of memory areas which are not enlarged at run time, unless you have a version with dynamic stack allocation. The default sizes for these areas should suffice for small applications, but most serious application require larger ones. They all can be modified by command line options. The table below shows these areas. The first column gives the option name to modify the size of the area. This option character should be followed immediately by a number and expresses the number of kilo bytes to use for the area. There are no other limits than the available memory of the machine to the sizes of the areas. The areas are described in table 2.3.

The heap is a memory area to store atoms, clauses, records, flags, etc. This area is dynamically enlarged at runtime on all versions of SWI-Prolog.

2.12.2 Other Limits

Clauses Currently the following limitations apply to clauses. The arity may not be more than 128, the number of links to the 'outside world' (predicates, atoms, (large) integers, etc) may not exceed 512 and the number of variables should be less than 256.[9]

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

Address space SWI-Prolog uses tagged pointers internally. This limits the number of available bits for addressing memory to 29 (512 Mb).

Integers Integers are tagged values. Their value is limited between -2^{26} and $2^{26} - 1$.

Floats Floating point numbers are C-doubles with a 12 bit reduction of the mantisse. For machines using IEEE floating point format, this implies the range is about $\pm 10^{308}$ and the accurracy about 10 digits.

[9]I plan to raise all these limits to 65536, the maximum addressable range by the 16 bits virtual machine codes.

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

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

```
% mkdir ~/lib/prolog
% cd !$
% touch INDEX.pl
%     <create library files>
% pl -g true -t make
 index for library . ... ok.
%
```

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

1. If a 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.

Notes on Automatic Loading

The autoloader is a new feature to SWI-Prolog. Its aim is to simplify program development and program management. Common lisp has a similar feature, but here the user has to specify which library is to be loaded if a specific function is called which is not defined. The advantage of the SWI-Prolog schema is that the user does not have to specify this. The disadvantage however is that the user might be wondering "where the hell this predicate comes from". Only experience can learn whether the functionality of the autoloader is appropriate. Comments are welcome.

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

2.10 Garbage Collection

SWI-Prolog version 1.4 was the first release to support garbage collection. Together with tail-recursion optimisation this guaranties forward chaining programs do not waste indefinite amounts of memory. Previous releases of this manual stressed on using failure-driven loops in those cases that no information needed to be passed to the next iteration via arguments. This to avoid large amounts of garbage. This is no longer stricktly necessary, but it should be noticed that garbage collection is a time consuming activity. Failure driven loops tend to be faster for this reason.[8]

[8]BUG: The garbage collector is deactivated when Prolog is called back from a foreign language predicate. This implies there is no garbage collection within a break environment. More seriously, there is no garbage collection when handling call-backs from —for example— the PCE package.

```
sun% pl -c load
foreign file dbase loaded 0.066667 seconds, 1578 bytes.
setup consulted, 0.500000 seconds, 5091 bytes.
main consulted, 0.333333 seconds, 3352 bytes.
load consulted, 1.000000 seconds, 9867 bytes.
sun% a.out -f none -L10 -G10 -T5
foreign file dbase loaded 0.066667 seconds, 1578 bytes.
Welcome to SWI-Prolog (version 1.6.0, May 1992)
Copyright (c) 1990, University of Amsterdam. All rights reserved.

1 ?- initialise.

Yes
2 ?- save_program(my_program,
        [ local     = 500
        , goal      = go
        , init_file = none
        ]).
Running executable: /usr/local/bin/pl
Saving to my_program; text: 204800 ... data: 357000 ... symbols ... done.
Yes
2 ?- halt.
sun%
```

Figure 2.3: Create a stand-alone executable

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

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

2.9 Automatic loading of libraries

If —at runtime— an undefined predicate is trapped the system will first try to import the predicate from the module's default module. If this fails the *auto loader* is activated. On first activation an index to all library files in all library directories is loaded in core (see `library_directory/1`). If the undefined predicate can be located in the one of the libraries that library file is automatically loaded and the call to the (previously undefined) predicate is resumed. By default this mechanism loads the file silently. The `please/3` option `verbose_autoload` is provided to get verbose loading. The please option `autoload` can be used to enable/disable the entire auto load system.

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

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 2.9.

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?).

2.8 Stand Alone Executables (Saved States)

The introduction of a foreign language interface raised the problem of incorporating the compiled foreign code into SWI-Prolog to create a new stand alone executable.[6] The current also allows one to create applications that do not need any of the SWI-Prolog system files.

save_program(*+NewProgram, +ListOfOptions***)**
Create a new executable wich will be named *NewProgram*. *ListOfOptions* is a list of *Key = Value* 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 2.3 shows a possible session. Note the use of 'initialise', which is supposed to be a predicate of the application doing time consuming initialisation.

[6] Note that this facility has not been ported to all machines. The introduction summarises portability issues.
[7] These are NOT the defaults compiled into an intermediate code file created with pl -c ...

2.6 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.

2.7 Environment Control

please(*+Key, -Old, +New*)
The predicate `please/3`[5] 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:

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,

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

`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 & Melish, 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.[4]

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

```
Yes
2 ?- 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 2.2: Example trace

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`).

`!!.`	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

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`) 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 Overview of the Debugger

SWI-Prolog has a standard 4-port tracer [Clocksin & Melish, 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 2.2.

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.

/ *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

```
/u4/staff/jan/.plrc consulted, 0.066667 seconds, 591 bytes
Welcome to SWI-Prolog (version 1.6.0, May 1992)
Copyright (c) 1990, 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

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.

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.[3]

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

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

[3] 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

`-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 2.6.

`-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 GNU 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.

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 is delivered with 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`.

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.[1] 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 '`.plrc`'[2] 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:

`-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.

`-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.

[1] On the Atari-ST, `pl.ttp` can be installed anywhere. The resource folder `pl` should be below the root directory of a disk and is searched for in the order `cde...ab` (e.i. hard disk first, followed by floppy)

[2] Atari-ST: '`pl.rc`' as files cannot be named '`.plrc`'

Machine	**OS**	**Dynamic Stacks**	**C Interface**	**Saved States**	**Profile**
SUN-3 and 4	SunOs 4	+	+	+	+
HP 9000s300	HP-UX 8.0	–	+	+	+
DEC MIPS	Ultrix 3.1	–	–	+	+
Gould PN	UTX-2.1	+	–	–	+
IBM PS2	AIX 2.0	–	–	+	+
IBM RISC-6000	AIX 3.1	–	+	+	+
VAX	Ultrix	–	+	+	+
ATARI	TOS 1.3	–	–	–	–
PC	LINUX	–	–	–	–

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.

- *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 X-windows.

1.1 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.

1.2 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 threated code.

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

1.3 Portability

The table below shows which machine dependent features have been ported to which architectures. Some of the minus signs could be taken away by spending more effort to the port, others are due to fundamental omisions or bugs in the operating system. Note that the column 'C-interface' implies object code can be linked dynamically from a running Prolog environment. For systems that have a minus here C-code can still be added by relinking SWI-Prolog itself together with the C-extensions to form an extended version of SWI-Prolog.

Status

This manual describes version 1.6.0 of SWI-Prolog. SWI-Prolog has been used now for some years. The application range includes Prolog course material, meta-interpreters, simulation of parallel Prolog, learning systems and a large workbench 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 to be 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.

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 5K bytes per second per MIPS (i.e. 35K bytes per second on SUN-4/110).
- *Transparent compiled code*
  SWI-Prolog compiled code can be treated just as interpreted code: you can list it, trace it, assert from or retract to 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.

Chapter 1

Introduction

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, or (`;/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 & Melish, 1981]. For more advanced Prolog material see [OKeefe, 1990]. Syntax and standard operator declarations confirm to the 'Edinburgh standard'. Most built in predicates are compatible with those described in [Clocksin & Melish, 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

Contents