Chapter 6

Generating Runtime Applications

This chapter describes the features of SWI-Prolog for delivering applications that can run without the development version of the system installed.

A SWI-Prolog built application consists of at least two parts: the emulator and the compiled application. The latter is in the same format as a SWI-Prolog boot-file and SWI-Prolog pre-compiled (QLF) file. This format is fast loadable and abstracted just far enough to be machine independent. This implies an application delivered in binary format can run on any computer for which an emulator is available without modification.

qsave_program(*+File, +ListOfOptions*)
Saves the current state of the program to the file 'File'. The result is an executable shell-script, that will start the emulator. *ListOfOptions* is a list of *Key = Value* pairs. The available keys are described in table 6.1.

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
autoload		bool	If true, run `autoload/0` first
map		atom	File containing info on dump

Table 6.1: Key = Value pairs for `qsave_program/2`

The `/bin/sh` script contains the following data:

1. The **shell script header** starts as:

```
#!/bin/sh
#SAVE-VERSION=<num>
#PROLOG-VERSION=<num>
exec ${SWIPL-/path-to-emulator} -x $0 "$@"
```

2. The **settings** section contains the default values for the various command line options.
3. The **predicates** section contains all predicates from the currently running system. Clauses of predicates defined as *volatile* (see `volatile/1`) are **not** saved. Neither are foreign predicates (see also below).
4. The **record** section contains the database records saved with `recorda/3` and friends. The current version saves records using directives.
5. The **flag** section contains the global flags saved using the `flag/3` predicate. Flags are saved as directives.
6. The **feature** section contains all features that do not originate from the emulator itself. See `set_feature/1`.
7. The **import** section contains the imports as far as they are not handled by the auto-import system. That is, an import is stored if the module is `user` or the module `user` contains a different definition as the one imported in the target module.'

Before writing the data to file, `qsave_program/2` will run `autoload/0` to all required autoloading the system can discover. See `autoload/0`.

Provided the application does not require any of the Prolog libraries to be loaded at runtime, the only file from the SWI-Prolog development environment required is the emulator itself. The emulator may be built in two flavours. The default is the *development* emulator. The *runtime* emulator is similar, but lacks the tracer. The stand-alone program `chpl` may be used to change the default path to the emulator.

qsave_program(*+File***)**
Equivalent to `qsave_program(File, [])`.

autoload
Check the current Prolog program for predicates that are refered 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 will not depend on one of the libraries. `Autoload/0` will find all *direct* references to predicates. It does not find predicates referenced via meta-predicates. The predicate `log/2` is defined in the library(quintus) to provide a quintus compatible means to compute the natural logarithm of a number. The following program will behave correctly if its state is executed in an environment where the library(quintus) is not available:

```
logtable(From, To) :-
        From > To, !.
logtable(From, To) :-
        log(From, Value),
        format('~d~t~8|~2f~n', [From, Value]),
        F is From + 1,
        logtable(F, To).
```

However, the following implementation refers to `log/2` through the meta-predicate `maplist/2`. Autoload will not be able to find the reference. This problem may be fixed either by loading the module libtary(quintus) explicitely or use `require/1` to tell the system that the predicate `log/2` is required by this module.

```
logtable(From, To) :-
        findall(X, between(From, To, X), Xlist),
        maplist(log, Xlist, SineList),
        write_table(Xlist, SineList).

write_table([], []).
write_table([I|IT], [V|VT]) :-
        format('~d~t~8|~2f~n', [I, V]),
        write_table(IT, VT).
```

volatile *+Name/Arity, ...*
Declare that the clauses of specified predicates should **not** be saved to the program. The volatile declaration is normally used to avoid that the clauses of dynamic predicates that represent data for the current session is saved in the state file.

6.1 Limitations of qsave_program

There are three areas that require special attention when using `qsave_program/[1,2]`.

- If the program is an embedded Prolog application or uses the foreign language interface, care has to be taken to restore the appropriate foreign context. See section 6.2 for details.
- If the program uses directives (`:- goal.` lines) that perform other actions then setting predicate attributes (dynamic, volatile, etc.) or loading files (consult, etc.), the directive may need to be prefixed with `initialization/1`.
- 'Database references as returned by `clause/3`, `recorded/3`, etc. are not preserved and may thus not be part of the database when saved.

6.2 Runtimes and Foreign Code

Some applications may need to use the foreign language interface. Object code is by definition machine-dependent and thus cannot be part of the saved program file.

To complicate the matter even furtherm there are various ways of loading foreign code:

- *Using the library(shlib) predicates*
  This is the preferred way of dealing with foreign code. It loads quickly and ensures an acceptable level of indepedence between the versions of the emulator and the foreign code loaded. It works on Unix machines supporting shared libraries and library functions to load them. Most modern Unixes satisfy this constraint.[1]. It also works on the Win32 platform: Windows-NT, '95 and Windows 3.1 running win32s.
- *Static linking*
  This mechanism works on all machines, but generally requires the same C-compiler and linker to be used for the external code as is used to build SWI-Prolog itself. This mechanism is the preferred way if shared libraries are not supported.

[1] Linux with the old a.out format does **not**

- *Using* `load_foreign/2,5]`
  Basically only works on Unix system supporting the a.out format executables. This mechanism is slow and non-portable. It should be avoided whenever possible.

To make a runtime executable that can run on multiple platforms one must make runtime checks to find the correct way of linking. Suppose we have a source-file `myextension` defining the installation function `install`.

If this file is compiled to a shared library, `load_foreign_library/1` will load this library and call the installation function to initialise the foreign code. If it is loaded as a static extension, define install() as the predicate `install/0`:

```
static foreign_t
pl_install()
{ install();

  PL_succeed;
}

PL_extension PL_extensions [] =
{
/*{ "name",     arity,  function,       PL_FA_<flags> },*/

  { "install",  0,      pl_install,     0 },
  { NULL,       0,      NULL,           0 }     /* terminating line */
};
```

Now, use the following Prolog code to load the foreign library:

```
load_foreign_extensions :-
        current_predicate(install, install), !, % static loaded
        install.
load_foreign_extensions :-                      % shared library
        load_foreign_library(foreign(myextension)).

:- initialization load_foreign_extensions.
```

The path alias `foreign` is defined by `file_search_path/2`. By default it searches the directories <home>/lib/<arch> and <home>/lib. The application can specify additional rules for `file_search_path/2`.

6.3 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` command-line argument is the preferred way to locate runtime files. The first step is to define an alias for the toplevel directory of your application. We will call this directory `gnatdir` in our examples.

A good place for storing data associated with SWI-Prolog runtime systems is below the emulator's home-directory. `swi` is a predefined alias for this directory. The following is a useful default definition for the search path.

```
user:file_search_path(gnatdir, swi(gnat)).
```

The application should locate all files using absolute_file_name. Suppose gnatdir contains a file `config.pl` to define local configuration. Then use the code below to load this file:

```
configure_gnat :-
        (   absolute_file_name(gnatdir('config.pl'), ConfigFile)
        ->  consult(ConfigFile)
        ;   format(user_error, 'gnat: Cannot locate config.pl~n'),
            halt(1)
        ).
```

6.4 Using chpl for Configuration Information

6.4.1 Changing the emulator of a runtime application

The program chpl, may be used to manipulate the header of a SWI-Prolog bootfile or state created with `qsave_program/[1,2]`.

It will be used most commonly by the installer of a SWI-Prolog runtime application to specify the path to the emulator. If the end-user decided to install the SWI-Prolog runtime environment in

```
/usr/local/lib/rt-pl-2.1.4
```

the gnat application can be told to use this emulator using:

```
% /usr/local/lib/rtpl-2.1.4/bin/chpl -e /usr/local/lib/rt-pl-2.1.4/bin/pl gnat
```

Now, `gnat` may be installed in any public or private directory for binaries.

6.4.2 Passing a path to the application

Suppose the system administrator has installed the SWI-Prolog runtime environment in `/usr/local/lib/rt-pl-2.1.4`. A user wants to install `gnat`, but gnat will look for its configuration in `/usr/local/lib/rt-pl-2.1.4/gnat` where the user cannot write.

The user decides to install the gnat runtime files in `/users/bob/lib/gnat`. For one-time usage, the user may decide to start gnat using the command:

```
% gnat -p gnatdir=/users/bob/lib/gnat
```

For a more widely used executable, this is not very comfortable. The user may decide to edit the shell-script part of gnat. Upto the line holding

```
# End Header
```

gnat is a simple `/bin/sh` script. After this line, the file is binary and may contain long lines. Most editors are not capable of editing such files.[2] Instead of editing the file directly, the program `chpl` may be used to extract and replace the header of `gnat`. The following editing sequence will work with any editor capable of editing ascii files.

```
% chpl -x gnat > gnat.hdr
% emacs gnat.hdr
% chpl -h gnat.hdr gnat
```

The header may be changed to the following to install gnat properly:

```
#!/bin/sh
# SWI-Prolog version: 2.1.4
# SWI-Prolog save-version: 25
exec ${SWIPL-/usr/local/lib/rt-pl-2.1.4/bin/pl} -x $0 \
     -p gnatdir=/users/bob/lib/gnat "$@"
```

6.5 The Runtime Environment

6.5.1 The Runtime Emulator

The sources may be used to built two versions of the emulator. By default, the *development* emulator is built. This emulator contains all features for interactive development of Prolog applications. If the system is configured using `--enable-runtime`, `make` will create a *runtime* version of the emulator. This emulator is equivalent to the development version, except for the following features:

- *No input editing*
  The GNU library -lreadline that provides EMACS compatible editing of input lines will never be linked to the system.
- *No tracer*
  The tracer and all its options are removed, making the system a little bit faster too.
- *No profiler*
  `profile/3` and friends are not supported. This saves some space and provides a bit better performance.
- *No interrupt*
  Keyboard interrupt (Control-C normally) is not rebound and will normally terminate the application.
- *feature(runtime, true) succeeds*
  This may be used to verify your application is running in the runtime environment rather than the development environment.

[2] If you use GNU-Emacs, make sure `require-final-newline` is set to `nil`

- `clause/[2,3]` *do not work on static predicates*
  This feature inhibits listing your program. It is only a very limited protection however.

The following fragment is an example for building the runtime environment in `$HOME/lib/rt-pl-2.1.4`. If possible, the shared-library interface should be configured to ensure it can serve a large number of applications.

```
% cd pl-2.1.4
% mkdir runtime
% cd runtime
% ../src/configure --enable-runtime --enable-shared --prefix=$HOME
% make
% make rt-install
```

The runtime directory contains the components listed below. This directory may be tar'ed and shipped with your application.

`README.RT`	Info on the runtime environment
`bin/pl`	The emulator itself
`bin/chpl`	The utility to change the runtime
`man/chpl.1`	Manual page for chpl
`man/pl.1`	Manual page for pl
`swipl`	pointer to the home directory (.)
`lib/`	directory for shared libraries
`lib/<arch>/`	machine-specific shared libraries

Appendix A

Hackers corner

This appendix describes a number of predicates which enable the Prolog user to inspect the Prolog environment and manipulate (or even redefine) the debugger. They can be used as entry points for experiments with debugging tools for Prolog. The predicates described here should be handled with some care as it is easy to corrupt the consistency of the Prolog system by misusing them.

A.1 Examining the Environment Stack

prolog_current_frame(-*Frame***)**
Unify *Frame* with an integer providing a reference to the parent of the current local stack frame. A pointer to the current local frame cannot be provided as the predicate succeeds deterministically and therefore its frame is destroyed immediately after succeeding.

prolog_frame_attribute(*+Frame, +Key,* -*Value***)**
Obtain information about the local stack frame *Frame*. *Frame* is a frame reference as obtained through `prolog_current_frame/1`, `prolog_trace_interception/3` or this predicate. The key values are described in table A.1.

A.2 Intercepting the Tracer

prolog_trace_interception(*+Port, +Frame,* -*Action***)**
Dynamic predicate, normally not defined. This predicate is called from the SWI-Prolog debugger just before it would show a port. If this predicate succeeds the debugger assumes the trace action has been taken care of and continues execution as described by *Action*. Otherwise the normal Prolog debugger actions are performed.

Port is one of `call`, `redo`, `exit`, `fail` or `unify`. *Frame* is an integer reference to the current local stack frame. *Action* should be unified with one of the atoms `continue` (just continue execution), `retry` (retry the current goal) or `fail` (force the current goal to fail). Leaving it a variable is identical to `continue`.

Together with the predicates described in section 3.36 and the other predicates of this chapter this predicate enables the Prolog user to define a complete new debugger in Prolog. Besides this it enables the Prolog programmer monitor the execution of a program. The example shown

Key	Value
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 '1' if *Frame* still is a candidate for backtracking. '0' otherwise.
goal	*Value* is unified with the goal associated with *Frame*. If the definition module of the active predicate is not *user* the goal is represented as *module:goal*. Do not instantiate variables in this goal unless you *know* what you are doing!
clause	*Value* is unified with a reference to the currently running clause. Fails if the current goal is associated with a foreign (C) defined predicate. See also `nth_clause/3` and `clause_property/2`.
level	*Value* is unified with the recursion level of *Frame*. The top level frame is at level '0'.
parent	*Value* is unified with an integer reference to the parent local stack frame of *Frame*. Fails if *Frame* is the top frame.
context_module	*Value* is unified with the name of the context module of the environment.
top	*Value* is unified with '1' if *Frame* is the top Prolog goal from a recursive call back from the foreign language. '0' otherwise.

Table A.1: Key values of `prolog_current_frame/1`

in figure A.1 records all goals trapped by the tracer in the database. To trace the execution of 'go' this way the following query should be given:

```
?- trace, go, notrace.
```

```
prolog_trace_interception(Port, Frame, continue) :-
        prolog_frame_attribute(Frame, goal, Goal),
        prolog_frame_attribute(Frame, level, Level),
        recordz(trace, trace(Port, Level, Goal)).
```

Figure A.1: Record a trace in the database

prolog_skip_level(-*Old*, +*New*)

Unify *Old* with the old value of 'skip level' and than set this level according to *New*. New is an integer, or the special atom `very_deep` (meaning don't skip). The 'skip level' is a global variable of the Prolog system that disables the debugger on all recursion levels deeper than the level of the variable. Used to implement the trace options 'skip' (sets skip level to the level of the frame) and 'up' (sets skip level to the level of the parent frame (i.e. the level of this frame minus 1).

A.3 Exception Handling

A start has been made to make exception handling available to the Prolog user. On exceptions a dynamic and multifile defined predicate `exception/3` is called. If this user defined predicate succeeds Prolog assumes the exception has been taken care of. Otherwise the system default exception handler is called.

exception(*+Exception, +Context, -Action*)

Dynamic predicate, normally not defined. Called by the Prolog system on run-time exceptions. Currently `exception/3` is only used for trapping undefined predicates. Future versions might handle signal handling, floating exceptions and other runtime errors via this mechanism. The values for *Exception* are described below.

undefined_predicate

If *Exception* is `undefined_predicate` *Context* is instantiated to a term *Name/Arity*. *Name* refers to the name and *Arity* to the arity of the undefined predicate. If the definition module of the predicate is not *user*, *Context* will be of the form *Module:Name/Arity*. If the predicate fails Prolog will print the default error warning and start the tracer. If the predicate succeeds it should instantiate the last argument either to the atom `fail` to tell Prolog to fail the predicate or the atom `retry` to tell Prolog to retry the predicate. This only makes sense if the exception handler has defined the predicate. Otherwise it will lead to a loop.

warning

If prolog wants to give a warning while reading a file, it will first raise the exception *warning*. The context argument is a term of the form *warning(Path, LineNo, Message)*, where *Path* is the absolute filename of the file prolog is reading; *LineNo* is an extimate of the line number where the error occurred and *Message* is a Prolog string indicating the message. The *Action* argument is ignored. The error is supposed to be presented to the user if the exception handler succeeds. Otherwise the standard Prolog warning message is printed.

This exception is used by the library `emacs_interface`, that integrates error handling with GNU-Emacs.

Appendix B

Summary

B.1 Predicates

`! /0`	Cut (discard choicepoints)
`! /1`	Cut block. See `block/3`
`, /2{xfy 1000}`	Conjuction of goals
`-> /2{xfy 1050}`	If-then-else
`. /2`	Consult. Also list constructor
`; /2{xfy 1100}`	Disjunction of goals. Same as `| /2`
`< /2{xfx 700}`	Arithmetic smaller
`= /2{xfx 700}`	Unification
`=.. /2{xfx 700}`	"Univ." Term to list conversion
`=:= /2{xfx 700}`	Arithmetic equal
`=< /2{xfx 700}`	Arithmetic smaller or equal
`== /2{xfx 700}`	Identical
`=@= /2{xfx 700}`	Structural identical
`=\= /2{xfx 700}`	Arithmetic not equal
`> /2{xfx 700}`	Arithmetic larger
`>= /2{xfx 700}`	Arithmetic larger or equal
`@< /2{xfx 700}`	Standard order smaller
`@=< /2{xfx 700}`	Standard order smaller or equal
`@> /2{xfx 700}`	Standard order larger
`@>= /2{xfx 700}`	Standard order larger or equal
`\+ /1{ fy 900}`	Negation by failure (argument not provable). Same as `not/1`
`\= /2{xfx 700}`	Not unifyable
`\== /2{xfx 700}`	Not identical
`\=@= /2{xfx 700}`	Not structural identical
`^ /2{xfy 200}`	Existential quantification (`bagof/3`, `setof/3`)
`| /2{xfy 1100}`	Disjunction of goals. Same as `; /2`
`abolish/2`	Remove predicate definition from the database
`abort/0`	Abort execution, return to top level
`absolute_file_name/2`	Get absolute path name
`absolute_file_name/3`	Get absolute path name with options
`access_file/2`	Check access permissions of a file

append/1	Append to a file
append/3	Concatenate lists
apply/2	Call goal with additional arguments
apropos/1	library(online_help): Show related predicates and manual sections
arg/3	Access argument of a term
arithmetic_function/1	Register an evaluable function
assert/1	Add a clause to the database
assert/2	Add a clause to the database, give reference
asserta/1	Add a clause to the database (first)
asserta/2	Add a clause to the database (first)
assertz/1	Add a clause to the database (last)
assertz/2	Add a clause to the database (last)
at_halt/1	Register goal to run at halt/1
at_initialization/1	Register goal to run at start-up
atom/1	Type check for an atom
atom_char/2	Convert between atom and ASCII value
atom_chars/2	Convert between atom and list of ASCII values
atom_length/2	Determine length of an atom
atom_to_term/3	Convert between atom and term
atomic/1	Type check for primitive
autoload/0	Autoload all predicates now
bagof/3	Find all solutions to a goal
between/3	Integer range checking/generating
block/3	Start a block ('catch'/'throw')
break/0	Start interactive toplevel
call/1	Call a goal
call/[2..]	Call with aditional arguments
call_dll_function/2	Win32: Call function in dynamic link library (.dll file)
call_shared_object_function/2	UNIX: Call C-function in shared (.so) file
character_count/2	Get character index on a stream
chdir/1	Change working directory
checklist/2	Invoke goal on all members of a list
clause/2	Get clauses of a predicate
clause/3	Get clauses of a predicate
clause_property/2	Get properties of a clause
close/1	Close stream
close_dde_conversation/1	Win32: Close DDE channel
close_dll/1	Win32: Close dynamic link library (.dll file)
close_shared_object/1	UNIX: Close shared library (.so file)
compare/3	Compare, using a predicate to determine the order
compiling/0	Is this a compilation run?
compound/1	Test for compound term
concat/3	Append two atoms
concat_atom/2	Append a list of atoms
consult/1	Read (compile) a Prolog source file
context_module/1	Get context module of current goal
convert_time/8	Convert time stamp
copy_term/2	Make a copy of a term
current_arithmetic_function/1	Examine evaluable functions

`current_atom/1`	Examine existing atoms
`current_flag/1`	Examine existing flags
`current_foreign_library/2`	library(`shlib`): Examine loaded shared libraries (.so files)
`current_functor/2`	Examine existing name/arity pairs
`current_input/1`	Get current input stream
`current_key/1`	Examine existing database keys
`current_module/1`	Examine existing modules
`current_module/2`	Examine existing modules
`current_op/3`	Examine current operator declaractions
`current_output/1`	Get the current output stream
`current_predicate/2`	Examine existing predicates
`current_stream/3`	Examine open streams
`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_unregister_service/1`	Win32: Terminate a DDE service
`debug/0`	Test for debugging mode
`debugging/0`	Show debugger status
`default_module/2`	Get the default modules of a module
`delete/3`	Delete all matching members from a list
`delete_file/1`	Remove a file from the file system
`discontiguous/1{fx 1150}`	Indicate distributed definition of a predicate
`display/1`	Write a term, ignore operators
`display/2`	Write a term, ignore operators on a stream
`displayq/1`	Write a term with quotes, ignore operators
`displayq/2`	Write a term with quotes, ignore operators on a stream
`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{fx 1150}`	Indicate predicate definition may change
`ed/0`	Edit last edited predicate
`ed/1`	Edit a predicate
`edit/0`	Edit last edited file
`edit/1`	Edit a file
`edit_source/1`	(hook) Intercept editing
`ensure_loaded/1`	Consult a file if that has not yet been done
`erase/1`	Erase a database record or clause
`exception/3`	(hook) Handle runtime exceptions
`exists_directory/1`	Check existence of directory
`exists_file/1`	Check existence of file
`exit/2`	Exit from named block. See block/3
`expand_file_name/2`	Wildcard expansion of file names
`expand_file_search_path/2`	Wildcard expansion of file paths
`expand_term/2`	Compiler: expand read term into clause(s)
`explain/1`	library(`explain`): Explain argument
`explain/2`	library(`explain`): 2nd argument is expalanation of first
`export/1`	Export a predicate from a module

`export_list/2`	List of public predicates of a module
`fail/0`	Always false
`fail/1`	Immediately fail named block. See block/3
`feature/2`	Get system configuration parameters
`file_base_name/2`	Get file part of path
`file_directory_name/2`	Get directory part of path
`file_search_path/2`	Define path-aliases for locating files
`fileerrors/2`	Do/Don't warn on file errors
`findall/3`	Find all solutions to a goal
`flag/3`	Simple global variable system
`flatten/2`	Transform nested list into flat list
`float/1`	Type check for a floating point number
`flush/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
`foreign_file/1`	Examine loaded foreign files
`format/1`	Formatted output
`format/2`	Formatted output with arguments
`format/3`	Formatted output on a stream
`format_predicate/2`	Program `format/[1,2]`
`free_variables/2`	Find unbound variables in a term
`functor/3`	Get name and arity of a term or construct a term
`garbage_collect/0`	Invoke the garbage collector
`gensym/2`	Generate unique atoms from a base
`get/1`	Read first non-blank character
`get/2`	Read first non-blank character from a stream
`get0/1`	Read next character
`get0/2`	Read next character from a stream
`get_single_char/1`	Read next character from the terminal
`get_time/1`	Get current time
`getenv/2`	Get shell environment variable
`ground/1`	Verify term holds no unbound variables
`halt/0`	Exit from Prolog
`halt/1`	Exit from Prolog with status
`hash_term/2`	Hash-value of ground term
`help/0`	Give help on help
`help/1`	Give help on predicates and show parts of manual
`history_depth/1`	Number of remembered queries
`ignore/1`	Call the argument, but always succeed
`import/1`	Import a predicate from a module
`index/1`	Change clause indexing
`initialization/1`	Initialization directive
`int_to_atom/2`	Convert from integer to atom
`int_to_atom/3`	Convert from integer to atom (non-decimal)
`integer/1`	Type check for integer
`intersection/3`	Set intersection
`is/2{xfx 700}`	Evaluate arithmetic expression

`is_absolute_file_name/1`	True if arg defines an absolute path
`is_list/1`	Type check for a list
`is_set/1`	Type check for a set
`keysort/2`	Sort, using a key
`last/2`	Last element of a list
`leash/1`	Change ports visited by the tracer
`length/2`	Length of a list
`library_directory/1`	(hook) Directories holding Prolog libraries
`limit_stack/2`	Limit stack expansion
`line_count/2`	Line number on stream
`line_position/2`	Character position in line on stream
`list_to_set/2`	Remove duplicates
`listing/0`	List program in current module
`listing/1`	List predicate
`load_foreign/2`	Load foreign (C) module
`load_foreign/5`	Load foreign (C) module
`load_foreign_library/1`	library(`shlib`): Load shared library (.so file)
`load_foreign_library/2`	library(`shlib`): Load shared library (.so file)
`make/0`	Reconsult all changed source files
`make_fat_filemap/1`	Win32: Create file containing non-FAT filenames
`make_library_index/1`	Create autoload file INDEX.pl
`maplist/3`	Transform all elements of a list
`member/2`	Element is member of a list
`memberchk/2`	Deterministic `member/2`
`merge/3`	Merge two sorted lists
`merge_set/3`	Merge two sorted sets
`module/1`	Query/set current type-in module
`module/2`	Declare a module
`module_transparent/1{fx 1150}`	Indicate module based meta predicate
`msort/2`	Sort, do not remove duplicates
`multifile/1{fx 1150}`	Indicate distributed definition of predicate
`name/2`	Convert between atom and list of ASCII characters
`nl/0`	Generate a newline
`nl/1`	Generate a newline on a stream
`nodebug/0`	Disable debugging
`nonvar/1`	Type check for bound term
`noprotocol/0`	Disable logging of user interaction
`nospy/1`	Remove spy point
`nospyall/0`	Remove all spy points
`not/1{fy 900}`	Negation by failure (argument not provable). Same as `\+ /1`
`notrace/0`	Stop tracing
`nth0/3`	N-th element of a list (0-based)
`nth1/3`	N-th element of a list (1-based)
`nth_clause/3`	N-th clause of a predicate
`number/1`	Type check for integer or float
`number_chars/2`	Convert between number and atom
`numbervars/4`	Enumerate unbound variables of a term using a given base

`once/1`	Call a goal deterministicaly
`op/3`	Declare an operator
`open/3`	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_shared_object/2`	UNIX: Open shared library (.so file)
`phrase/2`	Activate grammar-rule set
`phrase/3`	Activate grammar-rule set (returning rest)
`please/3`	Query/change environment parameters
`plus/3`	Logical integer addition
`portray/1`	(hook) Modify behaviour of `print/1`
`portray_clause/1`	Pretty print a clause
`predicate_property/2`	Query predicate attributes
`predsort/3`	Sort, using a predicate to determine the order
`preprocessor/2`	Install a preprocessor before the compiler
`print/1`	Print a term
`print/2`	Print a term on a stream
`profile/3`	Obtain execution statistics
`profile_count/3`	Obtain profile results on a predicate
`profiler/2`	Obtain/change status of the profiler
`prolog/0`	Run interactive toplevel
`prolog_current_frame/1`	Reference to goal's environment stack
`prolog_frame_attribute/3`	Obtain information on a goal environment
`prolog_load_context/2`	Context information for directives
`prolog_skip_level/2`	Indicate deepest recursion to trace
`prolog_to_os_filename/2`	Convert between Prolog and OS filenames
`prolog_trace_interception/3`	library(`user`): Intercept the Prolog tracer
`prompt1/1`	Change prompt for 1 line
`prompt/2`	Change the prompt used by read/1
`proper_list/1`	Type check for list
`protocol/1`	Make a log of the user interaction
`protocola/1`	Append log of the user interaction to file
`protocolling/1`	On what file is user interaction logged
`put/1`	Write a character
`put/2`	Write a character on a stream
`qcompile/1`	Compile source to Quick Load File
`qload/1`	Load Quick Load File as consult/1
`qsave_program/1`	Create runtime application
`qsave_program/2`	Create runtime application
`read/1`	Read Prolog term
`read/2`	Read Prolog term from stream
`read_clause/1`	Read clause
`read_clause/2`	Read clause from stream
`read_history/6`	Read using history substitution
`read_link/3`	Read a symbolic link
`read_variables/2`	Read clause including variable names
`read_variables/3`	Read clause including variable names from stream
`recorda/2`	Record term in the database (first)

recorda/3	Record term in the database (first)
recorded/2	Obtain term from the database
recorded/3	Obtain term from the database
recordz/2	Record term in the database (last)
recordz/3	Record term in the database (last)
redefine_system_predicate/1	Abolish system definition
rename_file/2	Change name of file
repeat/0	Succeed, leaving infinite backtrackpoints
require/1	This file requires these predicates
reset_profiler/0	Clear statistics obtained by the profiler
restore/1	Restore saved-state (save/1, save_program/1)
retract/1	Remove clause from the database
retractall/1	Remove unifying clauses from the database
reverse/2	Inverse the order of the elements in a list
same_file/2	Succeeds if arguments refer to same file
save/1	Save program including current goal
save/2	Save program including current goal
save_program/1	Save the current program on a file
save_program/2	Save the current program on a file
see/1	Change the current input stream
seeing/1	Query the current input stream
seen/0	Close the current input stream
select/3	Select element of a list
set_feature/2	Define a system feature
set_input/1	Set current input stream from a stream
set_output/1	Set current output stream from a stream
set_tty/2	Set 'tty' stream
setarg/3	Destructive assignment on term
setenv/2	Set shell environment variable
setof/3	Find all unique solutions to a goal
sformat/2	Format on a string
sformat/3	Format on a string
shell/0	Execute interactive subshell
shell/1	Execute OS command
shell/2	Execute OS command
show_profile/1	Show results of the profiler
size_file/2	Get size of a file in characters
skip/1	Skip to character in current input
skip/2	Skip to character on stream
sleep/1	Suspend execution for specified time
sort/2	Sort elements in a list
source_file/1	Examine currently loaded source files
source_file/2	Obtain source file of predicate
source_location/2	Location of last read term
spy/1	Force tracer on specified predicate
stack_parameter/4	Some systems: Query/Set runtime stack parameter
statistics/0	Show execution statistics
statistics/2	Obtain collected statistics
stream_position/3	Get/seek to position in file
string/1	Type check for string

`string_length/2`	Determine length of a string
`string_to_atom/2`	Conversion between string and atom
`string_to_list/2`	Conversion between string and list of ASCII
`style_check/1`	Change level of warnings
`sublist/3`	Determine elements that meet condition
`subset/2`	Generate/check subset relation
`substring/4`	Get part of a string
`subtract/3`	Delete elements that do not meet condition
`succ/2`	Logical integer successor relation
`swritef/2`	Formatted write on a string
`swritef/3`	Formatted write on a string
`tab/1`	Output number of spaces
`tab/2`	Output number of spaces on a stream
`tell/1`	Change current output stream
`telling/1`	Query current output stream
`term_expansion/2`	(hook) Convert term before compilation
`term_to_atom/2`	Convert between term and atom
`time/1`	Determine time needed to execute goal
`time_file/2`	Get last modification time of file
`tmp_file/2`	Create a temporary filename
`told/0`	Close current output
`trace/0`	Start the tracer
`trace/1`	Set trace-point on predicate
`trace/2`	Set/Clear trace-point on ports
`tracing/0`	Query status of the tracer
`trim_stacks/0`	Release unused memory resources
`true/0`	Succeed
`tty_get_capability/3`	Get terminal parameter
`tty_goto/2`	Goto position on screen
`tty_put/2`	Write control string to terminal
`ttyflush/0`	Flush output on terminal
`union/3`	Union of two sets
`unknown/2`	Trap undefined predicates
`unload_foreign_library/1`	library(`shlib`): Detach shared library (.so file)
`unsetenv/1`	Delete shell environment variable
`use_module/1`	Import a module
`use_module/2`	Import predicates from a module
`var/1`	Type check for unbound variable
`visible/1`	Ports that are visible in the tracer
`volatile/1{fx 1150}`	Predicates that are not saved
`wait_for_input/3`	Wait for input with optional timeout
`wildcard_match/2`	Csh(1) style wildcard match
`write/1`	Write term
`write/2`	Write term to stream
`write_ln/1`	Write term, followed by a newline
`writef/1`	Formatted write
`writef/2`	Formatted write on stream

`writeq/1`	Write term, insert quotes
`writeq/2`	Write term, insert quotes on stream

B.2 Artihmetic Functions

`*  /2{yfx 400}`	Multiplication
`+  /2{yfx 500}`	Addition
`-  /1{ fx 500}`	Unary minus
`-  /2{yfx 500}`	Subtraction
`/  /2{yfx 400}`	Division
`// /2{yfx 400}`	Integer division
`/\ /2{yfx 500}`	Bitwise and
`<< /2{yfx 400}`	Bitwise left shift
`>> /2{yfx 400}`	Bitwise right shift
`"C"/1` (list of one character)	Character code
`\  /1{ fx 500}`	Bitwise negation
`\/ /2{yfx 500}`	Bitwise or
`^  /2{xfy 200}`	Power function
`abs/1`	Absolute value
`acos/1`	Inverse (arc) cosine
`asin/1`	Inverse (arc) sine
`atan/1`	Inverse (arc) tangent
`atan/2`	Rectangular to polar conversion
`ceil/1`	Smallest integer larger than argument
`cos/1`	Cosine
`cputime/0`	Get CPU time
`e/0`	Mathematical constant
`exp/1`	Exponent (base e)
`floor/1`	Largest integer below argument
`integer/1`	Round to nearest integer
`log/1`	Natural logarithm
`log10/1`	10 base logarithm
`max/2`	Maximum of two numbers
`min/2`	Minimum of two numbers
`mod/2{xfx 300}`	Remainder of division
`random/1`	Generate random number
`pi/0`	Mathematical constant
`sin/1`	Sine
`sqrt/1`	Square root
`tan/1`	Tangent
`xor/2{yfx 400}`	Bitwise exclusive or

B.3 Operators

1	`fx`	`$`	Bind toplevel variable
200	`xfy`	`^`	(1) Predicate
200	`xfy`	`^`	(2) Arithmetic function
300	`xfx`	`mod`	Arithmetic function

400	yfx	*	Arithmetic function
400	yfx	/	Arithmetic function
400	yfx	//	Arithmetic function
400	yfx	<<	Arithmetic function
400	yfx	>>	Arithmetic function
400	yfx	xor	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
700	xfx	is	Predicate
700	xfx	\=	Predicate
700	xfx	\==	Predicate
700	xfx	\=@=	Predicate
900	fy	not	Predicate
900	fy	\+	Predicate
1000	xfy	,	Predicate
1050	xfy	->	Predicate
1100	xfy	;	Predicate
1100	xfy	|	Predicate
1150	fx	discontiguous	Predicate
1150	fx	dynamic	Predicate
1150	fx	module_transparent	Predicate
1150	fx	multifile	Predicate
1150	fx	volatile	Predicate
1150	fx	initialization	Predicate
1200	fx	:-	Introduces a directive
1200	fx	?-	Introduces a directive
1200	xfx	-->	DCGrammar: rewrite
1200	xfx	:-	head :- body. separator

Bibliography

[Anjewierden & Wielemaker, 1989] A. Anjewierden and J. Wielemaker. Extensible objects. ESPRIT Project 1098 Technical Report UvA-C1-TR-006a, University of Amsterdam, March 1989.

[BIM, 1989] *BIM Prolog release 2.4*. Everberg, Belgium, 1989.

[Bowen & Byrd, 1983] D. L. Bowen and L. M. Byrd. A portable Prolog compiler. In L. M. Pereira, ed., *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 MA, 1986.

[Clocksin & Mellish, 1981] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1981. [4th ed. 1994]

[Kernighan & Ritchie, 1978] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs NJ, 1978.

[O'Keefe, 1985] R. A. O'Keefe. *This Here Isn't Even Fine*, 1985. Also available from SWI, University of Amsterdam.

[O'Keefe, 1990] R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge MA, 1990.

[Pereira, 1986] F. Pereira. *C-Prolog User's Manual*, 1986.

[Qui, 1987] *Quintus Prolog, User Guide and Reference Manual*. Mountain View CA, 1987.

[Sterling & Shapiro, 1986] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge MA, 1986. [2nd ed. 1994]

[Warren, 1983] D. H. D. Warren. The runtime environment for a Prolog compiler using a copy algorithm. Technical Report 83/052, SUNY and Stone Brook, New York, 1983. Major revision March 1984.