Index

Bibliography

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

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

[Bowen & Byrd, 1983] D.L. Bowen and L.M. Byrd. A portable Prolog compiler. In L.M. Pereira, editor, *Proceedings of the Login Programming Workshop 1983*, Lisabon, Portugal, 1983. Universidade nova de Lisboa.

[Bratko, 1986] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading Massachusetts, 1986.

[Clocksin & Melish, 1981] W.F. Clocksin and C.S. Melish. *Programming in Prolog*. Springer-Verlag, New York, 1981.

[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.

[OKeefe, 1990] R.A. OKeefe. *The Craft of Prolog*. The MIT Press, Massachussetts, 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, Massachusetts, 1986.

[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, N.Y., 1983. Major revision March 1984.

`tab/2`	Output number of spaces on a stream
`tan/1`	Arithmetic: tangent
`tell/1`	Change current output stream
`telling/1`	Query current output stream
`term_expansion/2`	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
`told/0`	Close current output
`trace/0`	Start the tracer
`tracing/0`	Query status of the tracer
`trim_stacks/0`	Release unused memory resources
`true/0`	Succeed
`tty_fold/2`	Make terminal fold long lines in output
`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
`unsetenv/1`	Delete Unix 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`	Set ports that are visible in the tracer
`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
`writeq/1`	Write term, insert quotes
`writeq/2`	Write term, insert quotes on stream
`xor/2`	Arithmetic: exclusive or
`|/2`	Disjunction of goals

`recorded/3`	Obtain term from the database
`recordz/2`	Record term in the database (last)
`recordz/3`	Record term in the database (last)
`rename_file/2`	Change name of Unix file
`repeat/0`	Succeed, leaving infinite backtrackpoints
`reset_profiler/0`	Clear statistics obtained by the profiler
`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_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_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
`setenv/2`	Set Unix 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 Unix subshell
`shell/1`	Execute Unix command
`shell/2`	Execute Unix command
`show_profile/1`	Show results of the profiler
`sin/1`	Arithmetic: sine
`size_file/2`	Get size of a file in characters
`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
`spy/1`	Force tracer on specified predicate
`sqrt/1`	Arithmetic: square root
`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

`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`	Negation by failure (not provable)
`notrace/0`	Stop tracing
`nth0/3`	N-th element of a list (0-based)
`nth1/3`	N-th element of a list (1-based)
`number/1`	Type check for integer or float
`numbervars/4`	Enumerate unbound variables of a term
`once/1`	Call a goal deterministicaly
`op/3`	Declare an operator
`open/3`	Open a file (creating a stream)
`open_null_stream/1`	Open a stream to discard output
`pi/0`	Arithmetic: mathematical constant
`please/3`	Query/change environment parameters
`plus/3`	Logical integer addition
`portray/1`	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_skip_level/2`	Indicate deepest recursion to trace
`prolog_trace_interception/3`	Intercept the Prolog tracer
`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
`random/1`	Arithmetic: generate random number
`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_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

`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 Unix environment variable
`ground/1`	Verify term holds no unbound variables
`halt/0`	Exit from Prolog
`help/0`	Give help on help
`help/1`	Give help on predicates and show parts of manual
`history_depth/1`	Number of remembered queries
`read_history/6`	Read using history substitution
`ignore/1`	Call the argument, but always succeed
`import/1`	Import a predicate from a module
`index/1`	Change clause indexing
`int_to_atom/2`	Convert from integer to atom
`int_to_atom/3`	Convert from integer to atom (non-decimal)
`integer/1`	Arithmetic: round to nearest integer
`integer/1`	Type check for integer
`intersection/3`	Set intersection
`is/2`	Evaluate arithmetic expression
`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`	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
`log/1`	Arithmetic: natural logarithm
`log10/1`	Arithmetic: 10 base logarithm
`make/0`	Reconsult all changed source files
`maplist/3`	Transform all elements of a list
`max/2`	Arithmetic: Maximum of two numbers
`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
`min/2`	Arithmetic: Minimum of two numbers
`mod/2`	Arithmetic: remainder of division
`module/2`	Declare a module
`module_transparent/1`	Indicate module based meta predicate
`msort/2`	Sort, do not remove duplicates
`multifile/1`	Indicate distributed definition of predicate
`name/2`	Convert between atom and list of ASCII characters
`nl/0`	Generate a newline

`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
`debug/0`	Test for debugging mode
`debugging/0`	Show debugger status
`delete/3`	Delete all matching members from a list
`delete_file/1`	Unlink a file from the Unix file system
`discontiguous/1`	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`	Indicate predicate definition may change
`e/0`	Arithmetic: mathematical constant
`ed/0`	Edit last edited predicate
`ed/1`	Edit a predicate
`edit/0`	Edit last edited file
`edit/1`	Edit a file
`ensure_loaded/1`	Consult a file if that has not yet been done
`erase/1`	Erase a database record or clause
`exception/3`	Handle runtime exceptions
`exists_directory/1`	Check existence of Unix directory
`exists_file/1`	Check existence of Unix file
`exp/1`	Arithmetic: exponent (base e)
`expand_file_name/2`	Wildcard expansion of file names
`export/1`	Export a predicate from a module
`fail/0`	Always false
`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
`floor/1`	Arithmetic: largest integer below argument
`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`	Produce formatted output
`format/2`	Produce 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

`abolish/2`	Remove predicate definition from the database
`abort/0`	Abort execution, return to top level
`abs/1`	Arithmetic: absolute value
`absolute_file_name/2`	Get absolute Unix path name
`access_file/2`	Check access permissions of a file
`acos/1`	Arithmetic: inverse (arc) cosine
`append/1`	Append to a file
`append/3`	Concatenate lists
`apply/2`	Call goal with additional arguments
`apropos/1`	Show related predicates and manual sections
`arithmetic_function/1`	Register an evaluable function
`arg/3`	Access argument of a term
`asin/1`	Arithmetic: inverse (arc) sine
`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)
`atan/1`	Arithmetic: inverse (arc) tangent
`atan/2`	Arithmetic: rectangular to polar conversion
`atom/1`	Type check for an atom
`atom_length/2`	Determine length of an atom
`atom_to_term/3`	Convert between atom and term
`atomic/1`	Type check for primitive
`bagof/3`	Find all solutions to a goal
`between/3`	Integer range checking/generating
`break/0`	Start interactive toplevel
`call/1`	Call a goal
`ceil/1`	Arithmetic: smallest integer larger than argument
`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
`close/1`	Close stream
`compiling/0`	Is this a compilation run?
`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
`cos/1`	Arithmetic: cosine
`cputime/0`	Arithmetic: get CPU time
`current_atom/1`	Examine existing atoms
`current_arithmetic_function/1`	Examine evaluable functions
`current_flag/1`	Examine existing flags
`current_functor/2`	Examine existing name/arity pairs
`current_input/1`	Get current input stream
`current_key/1`	Examine existing database keys

Appendix B

Predicate Summary

`!/0`	Cut. Discard choicepoints
`*/2`	Arithmetic: multiplication
`+/2`	Arithmetic: addition
`,/2`	Conjuction of goals
`-/1`	Arithmetic: unary minus
`-/2`	Arithmetic: subtraction
`->/2`	If-then-else
`./2`	List operator. Also consult
`///2`	Arithmetic: Integer division
`//2`	Arithmetic: division
`/\/2`	Arithmetic: bitwise and
`;/2`	Disjunction of goals
`</2`	Arithmetic smaller
`<</2`	Arithmetic: bitwise left shift
`=../2`	Univ. Term to list conversion
`=/2`	Unification
`=:=/2`	Arithmetic equal
`=</2`	Arithmetic smaller or equal
`==/2`	Identical
`=@=/2`	Structural identical
`=\=/2`	Arithmetic not equal
`>/2`	Arithmetic larger
`>=/2`	Arithmetic larger or equal
`>>/2`	Arithmetic: bitwise right shift
`@</2`	Standard order smaller
`@=</2`	Standard order smaller or equal
`@>/2`	Standard order larger
`@>=/2`	Standard order larger or equal
`\/1`	Bitwise negation
`\//2`	Arithmetic: bitwise or
`\+/1`	Negation by failure (not provable)
`\=/2`	Not unifyable
`\==/2`	Not identical
`\=@=/2`	Not structural identical
`^/2`	Existential quantification (bagof/3, setof/3)

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.

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!
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).

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 3.34 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

option to include dbx debugging information. Then load them into SWI-Prolog. Now obtain the name of the current symbol table and the process id of Prolog. Then start dbx (or dbxtool) using

```
sun% dbx[tool] <symbol file> <pid>
```

Should this be done often then the following foreign predicate definition might help:

```
pl_dbx()
{ char *symbolfile = PL_query(PL_QUERY_SYMBOLFILE);
  char cmd[256];

  sprintf(cmd, "dbxtool %s %d &", symbolfile, getpid());
  if ( system(cmd) == 0 )
    PL_succeed;
  else
    PL_fail;
}
```

Register this predicate as `dbx/0` using the following call in your initialisation function:

```
PL_register_foreign("dbx", 0, pl_dbx, 0);
```

5.7.4 Name Conflicts in C modules

In the current version of the system all public C functions of SWI-Prolog are in the symbol table. This can lead to name clashes with foreign code. Someday I should write a program to strip all these symbols from the symbol table (why does Unix not have that?). For now I can only suggest to give your function another name. You can do this using the C preprocessor. If –for example– your foreign package uses a function warning(), which happens to exist in SWI-Prolog as well, the following macro should fix the problem.

```
#define warning warning_
```

5.7.5 Compatibility of the Foreign Interface

As far as I' aware of, there is no standard for foreign language interfaces in Prolog. The SWI-Prolog interface is no attempt to propose such a standard. It is (in part) tailored to the possibilities of the SWI-Prolog machinery. BIM-Prolog has a similar interface to analyse and construct terms. The major difference is that they have garbage collection and calls are made available to lock and unlock terms for garbage collection. I built a similar interface to Edinburgh C-Prolog (although less clean). This at least tells us that the interface can work for various forms of the WAM as well as a structure sharing Prolog.

As no standard exists nor emerges, users of the foreign language interface should carefully design the interface if the C-code should be portable to other Prolog implementation. The best advice to give is to define a small interface layer around the C-application and interface this to Prolog.

Compiling and Loading Foreign Code

```
sun% cc -O -c lowercase.c
sun% pl
/staff/jan/.plrc consulted, 0.166667 seconds, 2256 bytes.
Welcome to SWI-Prolog (version 1.6.0, May 1992)
Copyright (c) 1990, University of Amsterdam

1 ?- load_foreign(lowercase, init_lowercase).
foreign file(s) lowercase loaded, 0.016667 seconds, 464 bytes.

Yes
2 ?- lowercase('Hello World!', L).

L = 'hello world!'

Yes
```

5.7 Notes on Using Foreign Code

5.7.1 Garbage Collection and Foreign Code

Currently no interface between foreign code and the garbage collector has been defined. The garbage collector is disabled during execution of foreign code. Future versions might define such an interface. This probably will introduce incompatible changes to the current interface definition.

5.7.2 Memory Allocation

SWI-Prolog's memory allocation is based on the malloc() library routines. Foreign applications can savely use malloc(), realloc() and free(). Memory allocation using brk() or sbrk() is not allowed as these calls conflict with malloc().

5.7.3 Debugging Foreign Code

NOTE: this section is highly machine dependent. The tricks described here are tested on SUN-3 and SUN-4. They might work on other BSD variants of Unix.

Debugging incrementally loaded executables is a bit more difficult than debugging normal executables. The oldest way of debugging (putting print statements in your code at critical points) of course still works. 'Post-crash' debugging however is not possible. For adb/dbx to work they need (besides the core) the text segment and the symbol table. The symbol table lives somewhere on /tmp (called '`/tmp/pl_ld_...._.`', where '...' is the process id and '.' is an additional number to make sure the temporary file is unique. The text segment lives partly in the core (the incremental loaded bit) and partly in the SWI-Prolog executable.

The only way to debug foreign language code using a debugger is by starting the debugger on the running core image. Dbx(1) can do this. First compile the source files to be debugged with the '`-g`'

5.6 Example of Using the Foreign Interface

Below is an example showing all stages of the declaration of a foreign predicate that transforms atoms possibly holding uppercase letters into an atom only holding lower case letters. Figure 5.4 shows the C-source file.

C-Source file (lowercase.c)

```
/*  Include file depends on local installation */
#include "/usr/local/lib/pl/library/SWI-prolog.h"
#include <ctype.h>

long
pl_lowercase(u, l)
term u, l;
{ char *copy;
  char *s, *q;
  atomic la;

  if ( PL_type(u) != PL_ATOM )
    return PL_warning("lowercase/2: instantiation fault");
  s = PL_atom_value(PL_atomic(u));
  copy = (char *) malloc(strlen(s)+1);

  for( q=copy; *s; q++, s++)
    *q = (isupper(*s) ? tolower(*s) : *s);
  *q = '\0';

  la = PL_new_atom(copy);
  free(copy);

  return PL_unify_atomic(l, la);
}

init_lowercase()
{ PL_register_foreign("lowercase", 2, pl_lowercase, 0);
}
```

Figure 5.4: Lowercase source file

Registering Foreign Predicates

bool **PL_register_foreign(***name, arity, function, [...option...] 0***)**

Register a C-function to implement a Prolog predicate. After this call returns successfully a predicate with name *name* (a char *) and arity *arity* (a C int) is created. When called in Prolog, Prolog will call *function*. *[...option...]* forms a 0-terminated list of options for the installation. These are:

`PL_FA_NOTRACE`	Predicate cannot be seen in the tracer
`PL_FA_TRANSPARENT`	Predicate is module transparent
`PL_FA_NONDETERMINISTIC`	Predicate is non-deterministic. This attribute is currently ignored, but will probably be used in future versions.

void **PL_fatal_error(***format, a1, ...***)**
Print a message like PL_warning(), but starting with '`FATAL ERROR: `' and then exits Prolog.

Environment Control from Foreign Code

bool **PL_action(***int, C_type***)**
Perform some action on the Prolog system. *int* describes the action, *C_type* provides the argument if necessary. The actions are listed in table 5.1.

`PL_ACTION_TRACE`	Start Prolog tracer
`PL_ACTION_DEBUG`	Switch on Prolog debug mode
`PL_ACTION_BACKTRACE`	Print backtrace on current output stream
`PL_ACTION_HALT`	Halt Prolog execution. This action should be called rather than Unix exit() to give Prolog the opportunity to clean up. This call does not return.
`PL_ACTION_ABORT`	Generate a Prolog abort. This call does not return.
`PL_ACTION_BREAK`	Create a standard Prolog break environment. Returns after the user types control-D.
`PL_ACTION_SYMBOLFILE`	The argument (a char *) is considered to be hold the symbolfile for further incremental loading. Should be called by user applications that perform incremental loading as well and want to inform Prolog of the new symbol table.

Table 5.1: PL_action() options

Querying Prolog

C_type **PL_query(***int***)**
Obtain status information on the Prolog system. The actual argument type depends on the information required. *int* describes what information is wanted. The options are given in table 5.2.

`PL_QUERY_ARGC`	Return an integer holding the number of arguments given to Prolog from Unix.
`PL_QUERY_ARGV`	Return a char ** holding the argument vector given to Prolog from Unix.
`PL_QUERY_SYMBOLFILE`	Return a char * holding the current symbol file of the running process.
`PL_QUERY_ORGSYMBOLFILE`	Return the initial symbol file (before loading) of Prolog. By setting the symbol file to this value no name clashes can occur with previously loaded foreign files (but no symbols can be shared with earlier loaded modules as well).

Table 5.2: PL_query() options

Foreign Code and Modules

Modules are identified via a unique handle. The following functions are available to query and manipulate modules.

module **PL_context()**
Return the module identifier of the context module of the currently active foreign predicate.

term **PL_strip_module(***term, module **)
Utility function. If *term* is a term, possibly holding the module construct *module:rest* this function will return *rest* and fill *module ** with *module*. For further nested module constructs the inner most module is returned via *module **. If *term* is not a module construct *term* will simply be returned. If *module ** is `NULL` it will be set to the context module. Otherwise it will be left untouched. The following example shows how to obtain the plain term and module if the default module is the user module:

```
{ module m = PL_new_module(PL_new_atom("user"));

  if ( (term = PL_strip_module(term, &m)) == NULL )
    return PL_warning("Illegal module specification");

  ...
```

atomic **PL_module_name(***module***)**
Return the name of *module* as an atom.

module **PL_new_module(***atomic***)**
Find an existing or create a new module with name specified by the atom *atomic*.

Catching Unix Signals

SWI-Prolog catches the Unix signals SIGINT, SIGFPE and SIGSEGV. To avoid problems with foreign code attempting to catch these signals foreign code should call PL_signal() to install signal handlers rather than the Unix library function signal(). SWI-Prolog will always handle SIGINT itself. SIGFPE and SIGSEGV are passed to the foreign code handlers if Prolog did not expect that signal.

void **(*PL_signal(***sig, func***))()**
This function should be used to install signal handlers rather than the Unix library function signal(). It ensures consistent signal handling between SWI-Prolog and the foreign code and reinstalls signal handlers if a state created with `save_program/1` is restarted.

Errors and warnings

Two standard functions are available to print standard Prolog errors to the standard error stream.

bool **PL_warning(***format, a1, ...***)**
Print an error message starting with '`[WARNING: `', followed by the output from *format*, followed by a '`]`' and a newline. Then start the tracer. *format* and the arguments are the same as for printf(2). No more than 10 arguments can be provided.