```
%       Define class point

:- module(point, []).           % class point, no exports

%        name           type,           default access
%                                       value

variable(x,             integer,        0,      both).
variable(y,             integer,        0,      both).

%         method name   predicate name  arguments

behaviour(mirror,       mirror,         []).

mirror(P) :-
        fetch(P, x, X),
        fetch(P, y, Y),
        store(P, y, X),
        store(P, x, Y).
```

The predicates `fetch/3` and `store/3` are predicates that change instance variables of instances. The figure below indicates how message passing can easily be implemented:

```
%       invoke(+Instance, +Selector, ?ArgumentList)
%       send a message to an instance

invoke(I, S, Args) :-
        class_of_instance(I, Class),
        Class:behaviour(S, P, ArgCheck), !,
        convert_arguments(ArgCheck, Args, ConvArgs),
        Goal =.. [P|ConvArgs],
        Class:Goal.
```

The construct *'Module:Goal'* explicitely calls *Goal* in module *Module*. It is discussed in more detail in section 3.7.

4.6 Meta-Predicates in Modules

As indicated in the introduction, the problem with a predicate based module system lies in the difficulty to find the correct predicate from a Prolog term. The predicate 'solution(Solution)' can exist in more than one module, but 'assert(solution(4))' in some module is supposed to refer to the correct version of `solution/1`.

Various approaches are possible to solve this problem. One is to add an extra argument to all predicates (e.g. 'assert(Module, Term)'). Another is to tag the term somehow to indicate which module is desired (e.g. 'assert(Module:Term)'). Both approaches are not very attractive as they make the user responsible for chosing the correct module, inviting unclear programming by asserting in other modules. The predicate `assert/1` is supposed to assert in the module it is called from and should do so without being told explicitely. For this reason, the notion *context module* has been introduced.

4.6.1 Definition and Context Module

Each predicate of the program is assigned a module, called it's *definition module*. The definition module of a predicate is always the module in which the predicate was originally defined. Each active goal in the Prolog system has a *context module* assigned to it.

The context module is used to find predicates from a Prolog term. By default, this module is the definition module of the predicate running the goal. For meta-predicates however, this is the context module of the goal that invoked them. We call this *module_transparent* in SWI-Prolog. In the 'using maplist' example above, the predicate `maplist/3` is declared module_transparent. This implies the context module remains `extend`, the context module of `add_extension/3`. This way `maplist/3` can decide to call extend_atom in module `extend` rather than in it's own definition module.

All built-in predicates that refer to predicates via a Prolog term are declared module_transparent. Below is the code defining maplist.

```
:- module(maplist, maplist/3).

:- module_transparent maplist/3.

%       maplist(+Goal, +List1, ?List2)
%       True if Goal can succesfully be applied to all succesive pairs
%       of elements of List1 and List2.

maplist(_, [], []).
maplist(Goal, [Elem1|Tail1], [Elem2|Tail2]) :-
        apply(Goal, [Elem1, Elem2]),
        maplist(Goal, Tail1, Tail2).
```

4.6.2 Overruling Module Boundaries

The mechanism above is sufficient to create an acceptable module system. There are however cases in which we would like to be able to overrule this schema and explicitely call a predicate in some module or assert explicitly in some module. The first is useful to invoke goals in some module from the user's toplevel or to implement a object-oriented system (see above). The latter is useful to create and modify *dynamic* modules (see section 4.7).

For this purpose, the reserved term `:/2` has been introduced. All built-in predicates that transform a term into a predicate reference will check whether this term is of the form '<Module>:<Term>'. If so, the predicate is searched for in *Module* instead of the goal's context module. The `:/2` operator may be nested, in which case the inner-most module is used.

The special calling construct <Module>:<Goal> pretends *Goal* is called from *Module* instead of the context module. Examples:

```
?- assert(world:done).  % asserts done/0 into module world
?- world:assert(done).  % the same
```

```
?- world:done.          % calls done/0 in module world
```

4.7 Dynamic Modules

Sofar, we discussed modules that were created by loading a module-file. These modules have been introduced on facilitate the development of large applications. The modules are fully defined at load-time of the application and normally will not change during execution. Having the notion of a set of predicates as a self-contained world can be attractive for other purposes as well. For example, assume an application that can reason about multiple worlds. It is attractive to store the data of a particular world in a module, so we extract information from a world simply by invoking goals in this world.

Dynamic modules can easily be created. Any built-in predicate that tries to locate a predicate in a specific module will create this module as a side-effect if it did not yet exist. Example:

```
?- assert(world_a, consistent),
   world_a:unknown(_, fail).
```

These calls create a module called 'world_a' and make the call 'world_a:consistent' succeed. Undefined predicates will not start the tracer or autoloader for this module (see `unknown/2`).

Import and export from dynamically created world is arranged via the predicates `import/1` and `export/1`:

```
?- world_b:export(solve(_,_)).          % exports solve/2 from world_b
?- world_c:import(world_b:solve(_,_)).  % and import it to world_c
```

4.8 Module Handling Predicates

This section gives the predicate definitions for the remaining built-in predicates that handle modules.

:- module(+*Module*, +*PublicList***)**
: This directive can only be used as the first term of a source file. It declares the file to be a *module file*, defining *Module* and exporting the predicates of *PublicList*. *PublicList* is a list of name/arity pairs.

module_transparent +*Preds*
: *Preds* is a comma separated list of name/arity pairs (like `dynamic/1`). Each goal associated with a transparent declared predicate will inherit the *context module* from its parent goal.

context_module(-*Module***)**
: Unify *Module* with the context module of the current goal. `context_module/1` itself is transparent.

export(+*Head***)**
: Add a predicate to the public list of the context module. This implies the predicate will be imported into another module if this module is imported with `use_module/[1,2]`. Note that predicates are normally exported using the directive `module/2`. `export/1` is meant to handle export from dynamically created modules.

export_list(*+Module, ?Exports*)
Unifies *Exports* with a list of terms. Each term has the name and arity of a public predicate of *Module*. The order of the terms in *Exports* is not defined. See also `predicate_property/2`.

4.9 Compatibility of the Module System

The principles behind the module system of SWI-Prolog differ in a number of aspects from the Quintus Prolog module system.

- The SWI-Prolog module system allows the user to redefine system predicates.
- All predicates that are available in the *system* and *user* modules are visible in all other modules as well.
- Quintus has the '`meta_predicate/1`' declaration were SWI-Prolog has the `module_transparent/1` declaration.

The `meta_predicate/1` declaration causes the compiler to tag arguments that pass module sensitive information with the module using the `:/2` operator. This approach has some disadvantages:

- Changing a meta_predicate declaration implies all predicates *calling* the predicate need to be reloaded. This can cause serious consistency problems.
- It does not help for dynamically defined predicates calling module sensitive predicates.
- It slows down the compiler (at least in the SWI-Prolog architecture).
- At least within the SWI-Prolog architecture the run-time overhead is larger than the overhead introduced by the transparent mechanism.

Unfortunately the transparent predicate approach also has some disadvantages. If a predicate A passes module sensitive information to a predicate B, passing the same information to a module sensitive system predicate both A and B should be declared transparent. Using the Quintus approach only A needs to be treated special (i.e. declared with `meta_predicate/1`)[1]. A second problem arises if the body of a transparent predicate uses module sensitive predicates for which it wants to refer to its own module. Suppose we want to define `findall/3` using `assert/1` and `retract/1`[2]. The example in figure 4.1 gives the solution.

The Quintus `meta_predicate/1` directive can in many cases be replaced by the transparent declaration. Figure 4.2 gives a definition of `meta_predicate/1` as available from the 'quintus' library package.

The discussion above about the problems with the transparent mechanism

[1] Although this would make it impossible to call B directly.
[2] The system version uses `recordz/2` and `recorded/3`.

```
:- module(findall, [findall/3]).

:- dynamic
        solution/1.

:- module_transparent
        findall/3,
        store/2.

findall(Var, Goal, Bag) :-
        assert(findall:solution('$mark')),
        store(Var, Goal),
        collect(Bag).

store(Var, Goal) :-
        Goal,                   % refers to context module of
                                % caller of findall/3
        assert(findall:solution(Var)),
        fail.
store(_, _).

collect(Bag) :-
        ...,
```

Figure 4.1: `findall` using modules

```
:- op(1150, fx, (meta_predicate)).

meta_predicate((Head, More)) :- !,
        meta_predicate1(Head),
        meta_predicate(More).
meta_predicate(Head) :-
        meta_predicate1(Head).

meta_predicate1(Head) :-
        Head =.. [Name|Arguments],
        member(Arg, Arguments),
        module_expansion_argument(Arg), !,
        functor(Head, Name, Arity),
        module_transparent(Name/Arity).
meta_predicate1(_).             % just a mode declaration

module_expansion_argument(:).
module_expansion_argument(N) :- integer(N).
```

Figure 4.2: Definition of `meta_predicate/1`

Chapter 5

Foreign Language Interface

SWI-Prolog offers a powerful interface to C [Kernighan & Ritchie, 1978]. The main design objectives of the foreign language interface are flexibility and performance. Most Prolog foreign language interfaces allow the user only to pass primitive data through the interface. The user should normally specify for each argument whether it is an input or output argument as well as the type of the argument. Because type checking and conversion to/from C data types is done by Prolog the actual foreign code is usually short if something simple is wanted. The SWI-Prolog interface does not offer these primitives. Instead Prolog terms in their internal representation are passed via the interface. This allows the user to write 'logical' predicates and pass arbitrary Prolog data over the interface. As a trade-off the user is responsible for type checking and should be careful not to violate consistency rules as Prolog provides access to its internal data structures.

5.1 Overview of the Interface

A special include file called "SWI-Prolog.h" should be included with each C-source file that is to be loaded via the foreign interface. The installation process installs this file in the directory `include` in the SWI-Prolog home directory (?- feature(home, Home).). This C-header file defines various data types, macros and functions that can be used to communicate with SWI-Prolog. Functions and macros can be divided into the following categories:

- Analysing arbitrary Prolog terms
- Constructing new terms or instantiating existing ones
- Returning control information to Prolog
- Registering foreign predicates with Prolog
- Registering variables with the garbage collector
- Calling Prolog from C
- Communicating about modules
- Printing standard Prolog warning/error messages
- Global actions on Prolog (halt, break, abort, etc.)
- Querying the status of Prolog

5.2 Linking Foreign Modules

Foreign modules may be linked to Prolog in three ways. Using *static* linking, the extensions, a small description file and the basic SWI-Prolog object file are linked together to form a new executable. Using *dynamic* linking of object files, the extensions are linked into the running Prolog process. Finally, on some operating systems SWI-Prolog allows for loading dynamic libraries (called *shared libraries* (`.so` files) on Unix systems and *Dynamic Link Libraries* (`.dll` files) on many PC based operating systems.

5.2.1 What linking is provided?

The *static* linking schema can be used on all versions of SWI-Prolog. The `feature/2` predicate may be used to find out what other linking methods are provided for this version.

- *feature(load_foreign, true)*
  If this succeeds the system provides the `load_foreign/2` and `load_foreign/5` predicates that allow for loading object files (`.o` files on Unix machines).
- *feature(open_shared_object, true)*
  If this succeeds the system provides the `open_shared_object/2` and related predicates that allow for handling Unix shared object files based on the Unix library functions dlopen() and friends. See section 5.4.

  Note that, as this feature conflicts with `save/[1,2]` and `save_program/[1,2]`, it is not enabled by default. To enable it, the system should be configured using:

  ```
  % sh configure --enable-dlopen
  ```
- *feature(dll, true)*
  If this succeeds the system provides an interface for loading `.DLL` files by means of `open_dll/2` and friends. See section 5.4.

If either the feature *open_shared_object* or *dll* is true, the library `library(shlib.pl)` provides a common interface for loading foreign files from Prolog.

Note that a specific version of SWI-Prolog may offer multiple of these loading facilities.

5.2.2 What kind of loading should I be using?

All described approaches have their advantages and disadvantages. Static linking is portable and allows for debugging on all platforms. It is relatively cumbersome and the libraries you need to pass to the linker may vary from system to system.

Direct loading of object files is not very portable and generally works only on operating systems using the old BSD Unix a.out format for object and executable files. Also, the text of the loaded executable is in the data-segment of Prolog, disabling text-sharing and protection provided by Unix and some modern PC based operating systems.

Loading shared objects or DLL files provides sharing and protection and is generally the best choice. The problem is that `save/[1,2]` and `save_program/[1,2]` donot cooperate with this mechanism.

Summarising, if you donot need `save/[1,2] or save_program/[1,2]`, loading based on `library(shlib)` is probably the best choice. See section 5.4. Otherwise use loading object files directly for small things and static linking for large applications to exploit sharing and protection.

Development in SWI-Prolog and operating systems are likely to make `shlib` the standard foreign loading schema for SWI-Prolog.

Note that the definition of the foreign predicates is the same, regardless of the linking type used.

5.2.3 Static Linking

Below is an outline of the files structure required for statically linking SWI-Prolog with foreign extensions. `.../pl` refers to the SWI-Prolog home directory (see `feature/2`). <arch> refers to the architecture identifier that may be obtained using `feature/2`.

`.../pl/runtime/<arch>/pl.o`	Main object-file
`.../pl/include/SWI-Prolog.h`	Include file
`.../pl/include/SWI-Stream.h`	Stream I/O include file
`.../pl/include/SWI-Exports`	Export declarations (AIX only)
`.../pl/include/stub.c`	Extension stub

The definition of the foreign predicates is the same as for dynamic linking. Unlike with dynamic linking however, there is no initialisation function. Instead, the file `.../pl/include/stub.c` may be copied to your project and modified to define the foreign extensions. Below is stub.c, modified to link the lowercase example described later in this chapter:

```
/*  Copyright (c) 1991 Jan Wielemaker. All rights reserved.
    jan@swi.psy.uva.nl

    Purpose: Skeleton for extensions
*/

#include <stdio.h>
#include <SWI-Prolog.h>

extern foreign_t pl_lowercase(term, term);

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

  { "lowercase", 2       pl_lowercase,  0 },
  { NULL,        0,      NULL,          0 }     /* terminating line */
};


int
main(int argc, char **argv, char **env)
{ if ( !PL_initialise(argc, argv, env) )
    PL_halt(1);

  PL_halt(PL_toplevel() ? 0 : 1);
}
```

Now, a new executable may be created using the make(1) Makefile fragment below. The correct versions for the variables may be obtained using the `feature/2` predicate:

```
?- feature(c_cc,         CC).
?- feature(c_ldflags,    LDFLAGS).
?- feature(c_staticlibs, STATICLIBS).
?- feature(c_libs,       LIBS).
?- feature(arch,         ARCH).
?- feature(home,         HOME).
```

```
CC=gcc
HOME=/usr/local/lib/pl-2.0.6
CFLAGS=-I$(HOME)/include
LDFLAGS=
STATICLIBS=/usr/lib/libc.a
LIBS=-lreadline -ltermcap -lm
ARCH=i486-linux

PLOBJ=$(HOME)/$(ARCH)/pl.o
OBJ=lowercase.o stub.o

lwpl:   $(PLOBJ) $(OBJ)
        $(CC) -o $@ $(LDFLAGS) $(PLOBJ) $(OBJ) $(STATICLIBS) $(LIBS)
```

The resulting `lwpl` file will load the standard prolog bootstrap file. Normally, your extensions will partly be written in Prolog. In this case you may wish to create a new bootfile that includes the standard Prolog system and your extensions. Write a toplevel loadfile that loads your application and add the following to your Makefile:

```
PLFILES:        load.pl foo.pl bar.pl

lwpl.qlf:       lwpl $(PLFILES)
                ./lwpl -g mytoplevel \
                       -b $(HOME)/boot/init.pl \
                       -c load.pl
```

See also section 2.7 for compiling Prolog sources using the `-c` option.

5.2.4 Dynamic Linking based on `load_foreign/[2,5]`

Portability Note

The incremental linker is highly system dependent. It can easily be ported to machines for which the C linker allows you to link an object file using the symbol table of a (running) executable and use BSD Unix format a.out files. On many Unix systems such an object file can be created using the `-A` option of ld(1). Some operating systems offer primitives for dynamic loading (AIX, MACH). The following call may be used to find out whether load_foreign is functional:

```
?- feature(load_foreign, OK).
```

load_foreign(_+File, +Entry_**)**

Load a foreign file or list of files specified by *File*. The files are searched for similar to `consult/1`. Except that the '`.o`' extension is used rather than '`.pl`'. Thus '`test`' is a valid specification for '`test.o`' in the current directory, '`[test, library(basics)]`' specifies '`test.o`' in the current directory and '`basics.o`' in one of the library directories.

Entry defines the entry point of the resulting executable. The entry point will be called by Prolog to install the foreign predicates.

load_foreign(*+File, +Entry, +Options, +Libraries, +Size*)
The first two arguments are identical to those of `load_foreign/2`. *Options* is (a list of) additional option to be given to the loader. The default command is:[1]

```
$(CC) -N -A <symbolfile> -T <offset> -e <entry> -o <executable>
   <files> -lc
```

The options are inserted just before the files. *Libraries* is (a list of) libraries to be passed to the loader. They are inserted just after the files.

If *Size* is specified Prolog first assumes that the resulting executable will fit in *Size* bytes and do the loading in one pass. If the executable turns out to be larger than *Size* bytes the loading sequence is started again, using the calculated size. To determine the size of an executable specify a size that is too small. Prolog will then print the actual size on the current output stream.

foreign_file(*?File*)
Is true if *File* is the absolute path name of a file loaded as foreign file.

5.3 Dynamic Linking of shared libraries

The interface defined in this section allows the user to load shared libraries (.so files on most Unix systems). This interface is portable to all machines providing the function dlopen(), normally from the library -ldl. To enable it, SWI-Prolog must be configured using

```
% sh configure --enable-dlopen
```

These functions provide the basic interface layer. It is adviced to use the predicates from section 5.4 in your application.

open_shared_object(*+File, -Handle*)
File is the name of a .so file (see your C programmers documentation on how to create a .so file). This file is attached to the current process and *Handle* is unified with a handle to the shared object.

close_shared_object(*+Handle*)
Detach the shared object identified by *Handle*.

call_shared_object_function(*+Handle, +Function*)
Call the named function in the loaded shared library. The function is called without arguments and the return-value is ignored. Normally this function installs foreign language predicates using calls to PL_register_foreign().

5.4 Using the library shlib for .DLL and .so files

This section discusses the functionality of the (autoload) library `shlib.pl`, providing an interface to shared libraries. Currently it supports MS-Windows DLL (`.dll`) libraries and Unix `.so` (shared object) files.

[1] details depend on your operating system.

load_foreign_library(*+Lib* **)**

Equivalent to load_foreign_library(Lib, install).

load_foreign_library(*+Lib, +Entry* **)**

Search for the given foreign library and link it to the current SWI-Prolog instance. The library may be specified with or without the extension. It is searched for using the operating-system defined searching mechanism. If the library can be loaded, the function called *Entry* will be called without arguments. The return value of the function is ignored.

The *Entry* function will normally call PL_register_foreign() to declare functions in the library as foreign predicates.

unload_foreign_library(*+Lib* **)**

If the foreign library defines the function `uninstall`, this function will be called without arguments and its return value is ignored. Next, `abolish/2` is used to remove all known foreign predicates defined in the library. Finally the library itself is detatched from the process.

current_foreign_library(*-Lib, -Predicates* **)**

Query the currently loaded foreign libraries and their predicates. *Predicates* is a list with elements of the form *Module:Head*, indicating the predicates installed with PL_register_foreign().

Figure 5.1 connects a Windows message-box using a foreign function. This example was tested using Windows-NT and Microsoft Visual C++ 2.0.

```
#include <windows.h>
#include <SWI-Prolog.h>

static foreign_t
pl_say_hello(term_t to)
{ if ( PL_is_atom(to) )
  { char *a = PL_atom_value(PL_atomic(to));

    MessageBox(NULL, a, "DLL test", MB_OK|MB_TASKMODAL);

    PL_succeed;
  }

  PL_fail;
}

install_t
install()
{ PL_register_foreign("say_hello", 1, pl_say_hello, 0);
}
```

Figure 5.1: MessageBox() example in Windows-NT

5.5 Interface Data types

The interface functions can be divided into two groups. The first group are functions to obtain data from Prolog or pass data to Prolog. These functions use Prolog internal data types. The second group consists of type conversion functions convert between Prolog internal data and C primitive types. Below is a description of the types used in the interface.

term_t Terms represent arbitrary Prolog data (variables, atoms, integers, floats and compound terms). Terms live either until backtracking takes us back to a point before the term was created or the garbage collector has collected the term.

atomic_t Atomics are Prologs primitive data types (integers, atoms and floats). They can be transformed to C data types (int, char * resp. double). The Prolog representation for an integer is a tagged version of that integer. The mapping between C ints and Prolog integers can only be different over different releases of SWI-Prolog. Atoms are represented by a pointer to a data structure on the Prolog heap. Each such data structure is a unique representation of a string (e.g. to verify that two atoms represent the same string comparing the atoms suffices). The mapping between atoms and string are likely to vary over different sessions of Prolog. Floats are represented as (tagged) pointers to a float living on the global stack. For their life time the same rules apply as for terms. Whether two floats represent the same number can only be discovered by transforming both to C floats and then comparing them. Strings are represented as a tagged pointer to a char * on the global stack or heap. The rules for their lifetime and comparison equal those for floats and terms.

functor_t A functor is the internal representation of a name/arity pair. They are used to find the name and arity of a compound term as well as to construct new compound terms. Like atoms they live for the whole Prolog session and are unique.

predicate_t Handle to a foreign predicate using with PL_call_predicate(). Predicates live forever (although they can loose their definition).

module_t A module is a unique handle to a Prolog module. Modules are used only to call predicates in a specific module.

foreign_t Return type for a C-function implementing a Prolog predicate.

install_t Type for the `install()` and `uninstall()` functions of shared or dynamic link libraries. See section 5.4.

5.6 The Foreign Include File

5.6.1 Argument Passing and Control

If Prolog encounters a foreign predicate at run time it will call a function specified in the predicate definition of the foreign predicate. The arguments (1, ..., arity) pass the Prolog arguments to the goal as Prolog terms. Foreign functions should be declared of type `foreign_t`. Deterministic foreign functions have two alternatives to return control back to Prolog:

void **PL_succeed**
Succeed deterministically. PL_succeed is defined as “return TRUE”.

void **PL_fail**
Fail and start Prolog backtracking. PL_fail is defined as “return FALSE”.

Non-deterministic Foreign Predicates

By default foreign predicates are deterministic. Using the `PL_FA_NONDETERMINISTIC` attribute (see PL_register_foreign()) it is possible to register a predicate as a non-deterministic predicate. Writing non-deterministic foreign predicates is slightly more complicated as the foreign function needs context information for generating the next solution. Note that the same foreign function should be prepared to be simultaneously active in more than one goal. Suppose the `natural_number_below_n/2` is a non-deterministic foreign predicate, backtracking over all natural numbers lower than the first argument. Now consider the following predicate:

```
quotient_below_n(Q, N) :-
        natural_number_below_n(N, N1),
        natural_number_below_n(N, N2),
        Q =:= N1 / N2, !.
```

In this predicate the function `natural_number_below_n/2` simultaneously generates solutions for both its invocations.

Non-deterministic foreign functions should be prepared to handle three different calls from Prolog:

Initial call (`PL_FIRST_CALL`)
: Prolog has just created a frame for the foreign function and asks it to produce the first answer.

Redo call (`PL_REDO`)
: The previous invocation of the foreign function associated with the current goal indicated it was possible to backtrack. The foreign function should produce the next solution.

Terminate call (`PL_CUTTED`)
: The choice point left by the foreign function has been destroyed by a cut. The foreign function is given the opportunity to clean the environment.

Both the context information and the type of call is provided by an argument of type `foreign_t` appended to the argument list for deterministic foreign functions. The macro PL_foreign_control() extracts the type of call from the control argument. The foreign function can pass a context handle using the PL_retry() macro and extract the handle from the extra argument using the PL_foreign_context() macro.

void **PL_retry(***handle***)**
: The foreign function succeeds while leaving a choice point. On backtracking over this goal the foreign function will be called again, but the control argument now indicates it is a 'Redo' call and the macro PL_foreign_context() will return the handle passed via PL_retry(). This handle is a 30 bits signed value (two bits are used for status indication).

int **PL_foreign_control(***control_argument***)**
: Extracts the type of call from the control argument. The return values are described above. Note that the function should be prepared to handle the `PL_CUTTED` case and should be aware that the other arguments are not valid in this case.

long **PL_foreign_context(***control_argument***)**
: Extracts the context from the context argument. In the call type is `PL_FIRST_CALL` the context value is 0L. Otherwise it is the value returned by the last PL_retry() associated with this goal (both if the call type is `PL_REDO` as `PL_CUTTED`).

Note: If a non-deterministic foreign function returns using PL_succeed or PL_fail, Prolog assumes the foreign function has cleaned its environment. **No** call with control argument `PL_CUTTED` will follow.

The code of figure 5.2 shows a skeleton for a non-deterministic foreign predicate definition.

```
typedef struct                  /* define a context structure */
{ ...
} context;

foreign_t
my_function(term_t a0, term_t a1, foreign_t handle)
{ struct context * ctxt;

  switch( PL_foreign_control(handle) )
  { case PL_FIRST_CALL:
        ctxt = (struct context *) malloc(sizeof(struct context));
        ...
        PL_retry(ctxt);
    case PL_REDO:
        ctxt = (struct context *) PL_foreign_context(handle);
        ...
        PL_retry(ctxt);
    case PL_CUTTED:
        free(ctxt);
        PL_succeed;
  }
}
```

Figure 5.2: Skeleton for non-deterministic foreign functions

5.6.2 Analysing Terms via the Foreign Interface

Each argument of a foreign function (except for the control argument) is of type *term_t*. To analyse a term one should first obtain the type of the term. Primitive terms can then be transformed into *atomic* data in internal Prolog representation. This atomic data can be transformed into C-data types. Complex terms are analysed in terms on their functor and arguments. The arguments themselves are terms, allowing the same procedure to be repeated recursively.

int **PL_type(***term_t***)**
Obtain the type of *term*, which should be a term returned by one of the other interface predicates or passed as an argument. The function returns the type of the Prolog term. The type identifiers are listed below.

`PL_VARIABLE`	An unbound variable. The value of term as such is a unique identifier for the variable.
`PL_ATOM`	A Prolog atom.
`PL_STRING`	A Prolog string.
`PL_INTEGER`	A Prolog integer.
`PL_FLOAT`	A Prolog floating point number.
`PL_TERM`	A compound term. Note that a list is a compound term with name '.' and arity 2.

The functions PL_is_<type> allow form an alternative to PL_type. The test "PL_is_var(term)" is equivalent to "PL_type(term) == PL_VARIABLE", but the first is considerably faster.

int **PL_is_var(***term_t***)**
Returns non-zero if *term* is a variable.

int **PL_is_atom(***term_t***)**
Returns non-zero if *term* is an atom.

int **PL_is_string(***term_t***)**
Returns non-zero if *term* is a string.

int **PL_is_int(***term_t***)**
Returns non-zero if *term* is an integer.

int **PL_is_float(***term_t***)**
Returns non-zero if *term* is a float.

int **PL_is_term(***term_t***)**
Returns non-zero if *term* is a compound term.

atomic_t **PL_atomic(***term_t***)**
Return the atomic value of *term* in Prolog internal representation. *Term* should be atomic (e.g. atom, integer, float or string).

long **PL_integer_value(***atomic_t***)**
Transforms an integer from Prolog internal representation into a C long.

double **PL_float_value(***atomic_t***)**
Transforms a float from Prolog internal representation into a C double.

*char ** **PL_atom_value(***atomic_t***)**
Transforms an atom from Prolog internal representation into a 0-terminated C char *. The pointer points directly into the Prolog heap and can assumed to be static. The contents of the character string however should under NO circumstances be modified.

*char ** **PL_string_value(***string***)**
Transform a string from Prolog internal representation into a C char *. The pointer points directly into the Prolog data area. Unlike the pointer returned by PL_atom_value() the C user should copy the value to a private data area if its value should survive the current foreign language call. Like PL_atom_value(), changing the contents of the character string is NOT allowed.

functor_t **PL_functor(***term_t***)**
term should be a complex term. The return value is a unique identifier of the term's name and arity. The following example demonstrates this:

```
pl_same_functor(term_t t1, term_t t2)
{ if ( !PL_is_term(t1) || !PL_is_term(t2) )
    PL_fail;
  if ( PL_functor(t1) == PL_functor(t2) )
    PL_succeed;
  PL_fail;
}
```

atomic_t **PL_functor_name(***functor_t***)**
Return an atom representing the name of *functor*. To get the functor name as char * of a term which is known to be compound:

```
#define functor_name(term) PL_atom_value(PL_functor_name(PL_functor(term)))
```

int **PL_functor_arity(***functor_t***)**
Return a C integer representing the arity of *functor*.

term **PL_arg(***term_t, int***)**
Return the *int*-th argument of *term*. Argument counting starts at 1 and is valid up to and including the arity of *term*. No checks on these boundaries are performed.

Figure 5.3 shows a definition of `display/1` to illustrate the described functions.

5.6.3 Instantiating and Constructing Terms

Terms are instantiated as in Prolog itself by unification. Variables can be unified with atomic data, with a functor and with other terms. New terms are first constructed as a single unbound variable.

term_t **PL_new_term()**
Create a new term. The term is an unbound variable living on the global stack. In the current implementation it lives until Prolog backtracks to before this call. Garbage collection might change this in the future.

atomic_t **PL_new_atom(***char ****)**
Create a Prolog atom from a C char *. The contents of the char * are copied to the Prolog heap.

atomic_t **PL_new_string(***char ****)**
Create a Prolog string, living on the global stack. The contents of the char * are copied into Prolog's data area.

atomic_t **PL_new_integer(***long***)**
Create a Prolog integer from a C long. Note that the integer is truncated to 28 bits. No checks on this are performed.

atomic_t **PL_new_float(***double***)**
Create a Prolog float living on the global stack from *double*.

functor_t **PL_new_functor(***atomic_t, int***)**
Create a Prolog functor identifier form *atomic* (which should be an atom) and int, the arity. *Arity* is valid for any $arity \geq 0$. $Arity = 0$ is used internally, but none of the interface functions use it.

```
pl_display(term_t t)
{ functor_t functor;
  int arity, n;

  switch( PL_type(t) )
  { case PL_VARIABLE:
        printf("_%d", t);
        break;
    case PL_ATOM:
        printf("%s", PL_atom_value(PL_atomic(t)));
        break;
    case PL_STRING:
        printf("\"%s\"", PL_string_value(PL_atomic(t)));
        break;
    case PL_INTEGER:
        printf("%d", PL_integer_value(PL_atomic(t)));
        break;
    case PL_FLOAT:
        printf("%f", PL_float_value(PL_atomic(t)));
        break;
    case PL_TERM:
        functor = PL_functor(t);
        arity = PL_functor_arity(functor);
        printf("%s", PL_atom_value(PL_functor_name(functor)));
        printf("(");
        pl_display(PL_arg(t, 1));
        for( n = 2; n <= arity; n++)
        { printf(", ");
          pl_display(PL_arg(t, n));
        }
        printf(")");
        break;
    default:
        PL_fatal_error("Illegal type in pl_display(): %d",
                                                PL_type(t));
  }

  PL_succeed;
}
```

Figure 5.3: A Foreign definition of `display/1`

int **PL_unify(***term_t, term_t*)
Unify two terms. Return value is `TRUE` or `FALSE`.

int **PL_unify_atomic(***term_t, atomic_t*)
Unify a term with an atomic value.

int **PL_unify_functor(***term_t, functor_t*)
Unify a term with a functor. The unification simply succeeds if *term* is already instantiated to a term with functor *functor*. If *term* is variable it will be instantiated to the most general term of *functor* (e.g. a term with all arguments unbound variables). Otherwise `FALSE` is returned.

If this call succeeds the arguments can be further instantiated by calling PL_arg() and recursively unifying the returned terms.

An example of using these functions is shown in figure 5.4.

5.6.4 Interaction with the garbage collector and stack-shifter

C-functions that manipulate Prolog terms manipulate pointers to the Prolog runtime stacks. Prolog implements two mechanisms for avoiding stack overflow: garbage collection and stack expansion. On machines that allow for it, Prolog will use virtual memory management to detect stack overflow and expand the runtime stacks. On other machines Prolog will reallocate the stacks and update all pointers to them. To do so, Prolog needs to know which variables of active C-functions contain references to Prolog data.

C-variables may be registered with the Prolog engine using the macro PL_lock() and unregistered using PL_unlock(). These macros should be perfectly balanced and any term locked in a C-function needs to be unlocked before control is passed back to Prolog.

A C-variable that is registered must hold valid Prolog data or '0'.

void **PL_lock(***term_t*)
Register the named C-variable with the Prolog engine. The variable should hold a valid Prolog term. The value of the variable may be changed while it remains locked. Each locked variable must be unlocked using PL_unlock(). Locking and unlocking must be balanced.

void **PL_unlock(***term_t*)
Unlock a locked variable. If the named variable is not the most recently locked variable an error message is printed. If a foreign function leaves variables unlocked an error message is issued as well.

When is locking necessary

Stacks shifts or garbage collections may be forced if there is insufficient space on the stacks. Data is pushed on the stacks by most of the term construction functions and if the Prolog engine is called using PL_call() or PL_call_predicate().

By default the Prolog engine ensures the stacks have a minimum of 'min_free' (see `stack_parameter/4`) space when a foreign predicate is activated. If this is insufficient, Prolog checks whether a garbage collection may be performed or the stacks may be enlarged.[2] By default Prolog assumes it is *not* save to shift the stacks or perform a garbage collection when foreign code is active.

[2] When using virtual memory based stacks the system can always enlarge the stacks as it needs not to shift them.

Simple C-defined predicates that do not call Prolog and do not use the term-construction functions intensively do not need to use PL_lock() and PL_unlock(). C-defined predicates that perform call-back on Prolog or use the term construction primitives to (potentially) create large data structures should protect their variables and should defined the PL_FA_GCSAVE attribute (see PL_register_foreign()).

5.6.5 Calling Prolog from C

The Prolog system can be called back from C. This is done by constructing a term with the functions described above and then calling PL_call(). PL_call() executes the goal and returns TRUE or FALSE depending on success or failure of the called predicate. There are no means to backtrack over the Prolog predicate. If alternatives are wanted call `findall/3` and read the result from the third argument.

int **PL_call(***term_t, module_t***)**
Call term just like the Prolog predicate `once/1`. *Term* is called in the specified module, or in the context module if module_t = NULL. Returns `TRUE` if the call succeeds, `FALSE` otherwise. Figure 5.4 shows an example to obtain the number of defined atoms. All checks are omitted to improve readability.

5.6.6 Discarding Data

The Prolog data created during setting up the call and calling Prolog can in most cases be discarded right after the call. See figure 5.4 for an example.

void **PL_mark(***bktrk_buf***)**
Mark the global and trail stacks in *bktrk_buf*.

void **PL_bktrk(***bktrk_buf***)**
Undo all changes in the runtime stacks since a snapshot has been made in buffer using PL_mark(). Changes to the heap are not affected.

It is not necessary to call PL_bktrk() for each PL_mark(). The user should ensure that PL_bktrk() is never called with a buffer that is created after a buffer to which PL_bktrk() has been called. Thus PL_mark(b1) ... PL_mark(b2) ... PL_bktrk(b1) is valid, but it is not allowed to call PL_bktrk(b2) after this sequence.

5.6.7 Calls based on predicate references

In some situations, callback is highly time-critical and needs to be performed often on the same predicate. In these cases, Pl_call() may be too slow as it involves building the gaol using various unification calls and calling PL_call() that will again unpack the call-term. In these cases the interface described in this section may be useful.

predicate_t **PL_predicate(***functor_t, module_t***)**
Return a handle to the given predicate in the given module. If module_t is `NULL`, the current context module will be used. This call always succeeds, possibly creating a handle to an undefined predicate.

```
int
count_atoms()
{ term t;
  int atoms;
  bktrk_buf buf;

  PL_mark(&buf);              /* mark the global stack */

  t = PL_new_term();          /* create `statistics(atoms, Var)' */
  PL_unify_functor(t, PL_new_functor(PL_new_atom("statistics"), 2));
  PL_unify_atomic(PL_arg(t,1), PL_new_atom("atoms"));

  PL_call(t, NULL);           /* call it in current module */

                              /* extract the value from the 2nd arg */
  atoms = PL_integer_value(PL_atomic(PL_arg(t, 2)));

  PL_bktrk(&buf);             /* discard global stack data created */

  return atoms;
}
```

Figure 5.4: Calling Prolog

int **PL_call_predicate(***module_t, int debug, predicate_t, term_t *argv*)
Use a predicate handle to call *predicate* with the given arguments. The vector *argv* must point to a vector of Prolog terms with enough elements for the predicate.

The *module* argument defines the context module of the goal if the called predicate is module transparent. If this argument is specified is `NULL`, this is the current context module. If no context is available it is the `user` module. If the *debug* argument is `FALSE`, the call will be hidden from the Prolog debugger.

The *module* and *debug* arguments are in almost all normal cases `NULL` and `TRUE`. They are needed for some internal cases. PL_call_predicate() is the function containing the actual Prolog Virtual Machine Interpreter, flattening the call-stack as far as possible[3]

The following example realises the same as the example from figure 5.4

Finally, the direct predicate interface provides some utility functions.

int **PL_predicate_arity(***predicate_t*)
Returns the number of arguments required by the predicate. The argument vector provided to PL_call_predicate() should contain at least this number of elements.

atomic_t **PL_predicate_name(***predicate_t*)
Returns an atom identifying the name of the predicate.

functor_t **PL_predicate_functor(***predicate_t*)
Functor description of the term. This is always the same as the first argument of the PL_predicate() call used to find the predicate.

[3] Especially on the SUN SPARC cpu this is critical for good Prolog/C/Prolog calling performance

```
int
count_atoms()
{ term_t t[2];
  int atoms;
  bktrk_buf buf;

  PL_mark(&buf);             /* mark the global stack */

  t[0] = PL_new_term();         /* create `statistics(atoms, Var)' */
  t[1] = PL_new_term();
  PL_unify_atomic(t[0], PL_new_atom("atoms"));
  PL_call_predicate(NULL, TRUE,
                  PL_predicate(PL_functor(PL_new_atom("statistics"), 2)
                               PL_new_module(PL_new_atom("user"))),
                  t);

                             /* extract the value from the 2nd arg */
  atoms = PL_integer_value(PL_atomic(t[1]));

  PL_bktrk(&buf);            /* discard global stack data created */

  return atoms;
}
```

Figure 5.5: Using PL_call_predicate

module_t **PL_predicate_module(***predicate_t***)**
Returns a handle to the module in which the predicate is defined. Note that this is not necessarily the same as the module handle used to locate the predicate with PL_predicate() as the predicate may be imported. As predicate modifications may change the import/export relations, the return value may change between calls.

5.6.8 Foreign Code and Modules

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

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

term_t **PL_strip_module(***term_t, module_t ****)**
Utility function. If *term_t* 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_t **PL_module_name(***module_t***)**
Return the name of *module* as an atom.

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

5.6.9 Catching Signals (Software Interrupts)

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.

5.6.10 Errors and warnings

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

int **PL_warning(***format, a1, ...***)**
Print an error message starting with '`[WARNING: `', followed by the output from *format*, followed by a '`]`' and a newline. Then start the tracer. *format* and the arguments are the same as for printf(2). Always returns FALSE.

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

5.6.11 Environment Control from Foreign Code

int **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.

5.6.12 Querying Prolog

C_type **PL_query(***int***)**
Obtain status information on the Prolog system. The actual argument type depends on the

PL_ACTION_TRACE	Start Prolog tracer
PL_ACTION_DEBUG	Switch on Prolog debug mode
PL_ACTION_BACKTRACE	Print backtrace on current output stream. The argument (an int) is the number of frames printed.
PL_ACTION_HALT	Halt Prolog execution. This action should be called rather than Unix exit() to give Prolog the opportunity to clean up. This call does not return.
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

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

5.6.13 Registering Foreign Predicates

int **PL_register_foreign(***name, arity, function, flags***)**
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*. *flags* forms bitwise or'ed 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.
PL_FA_GCSAVE	The C-implementation properly uses PL_lock() and PL_unlock() to protect variables from the garbage collector and stack-shifter

5.6.14 Foreign Code Hooks

For various specific applications some hooks re provided.

PL_dispatch_hook_t **PL_dispatch_hook(***PL_dispatch_hook_t*)
If this hook is not NULL, this function is called when reading from the terminal. It is supposed to dispatch events when SWI-Prolog is connected to a window environment. It can return two values: `PL_DISPATCH_INPUT` indicates Prolog input is available on file descriptor 0 or `PL_DISPATCH_TIMEOUT` to indicate a timeout. The old hook is returned. The type `PL_dispatch_hook_t` is defined as:

```
typedef int  (*PL_dispatch_hook_t)(void);
```

void **PL_abort_hook(***PL_abort_hook_t*)
Install a hook when `abort/0` is executed. SWI-Prolog `abort/0` is implemented using C setjmp()/longjmp() construct. The hooks are executed in the reverse order of their registration after the longjmp() took place and before the Prolog toplevel is reinvoked. The type `PL_abort_hook_t` is defined as:

```
typedef void (*PL_abort_hook_t)(void);
```

int **PL_abort_unhook(***PL_abort_hook_t*)
Remove a hook installed with PL_abort_hook(). Returns `FALSE` if no such hook is found, `TRUE` otherwise.

void **PL_reinit_hook(***PL_reinit_hook_t*)
Install a hook that is called when a saved program (using `save_program/[1,2]`) is restored. The hooks are called in reverse order. The type `PL_reinit_hook_t` is defined as:

```
typedef void (*PL_reinit_hook_t)(int argc, char **argv);
```

int **PL_reinit_unhook(***PL_reinit_hook_t*)
Remove a hook installed with PL_reinit_hook(). Returns `FALSE` if no such hook is found, `TRUE` otherwise.

5.6.15 Embedding SWI-Prolog in a C-program

As of version 2.1.0, SWI-Prolog may be embedded in a C-program. To reach at a compiled C-program with SWI-Prolog as an embedded application is very similar to creating a statically linked SWI-Prolog executable as described in section 5.2.3.

The file `pl-stub.c` defines SWI-Prologs default main program:

```
int
main(int argc, char **argv, char **env)
{ if ( !PL_initialise(argc, argv, env) )
    PL_halt(1);

  PL_halt(PL_toplevel() ? 0 : 1);
}
```

This may be replaced with your own main C-program. The interface function PL_initialise() **must** be called before any of the other SWI-Prolog foreign language functions described in this chapter. PL_initialise() interprets all the command-line arguments, except for the `-t toplevel` flag that is interpreted by PL_toplevel().

int **PL_initialise(***int argc, char **argv, char **environ*)

Initialises the SWI-Prolog heap and stacks, restores the boot QLF file, loads the system and personal initialisation initialisation files, runs the `at_initialisation/1` hooks and finally runs the `-g goal` hook.

PL_initialise() returns 1 if all initialisation succeeded and 0 otherwise. Various fatal errors may cause PL_initialise to call PL_halt(1), preventing it from returning at all.

int **PL_toplevel()**

Runs the goal of the `-t toplevel` switch (default `prolog/0`) and returns 1 if successfull, 0 otherwise.

void **PL_halt(***int status*)

Cleanup the Prolog environment and calls exit() with the status argument.

5.7 Example of Using the Foreign Interface

Below is an example showing all stages of the declaration of a foreign predicate that transforms atoms possibly holding uppercase letters into an atom only holding lower case letters. Figure 5.6 shows the C-source file.

5.7.1 C-Source file (lowercase.c)

```
/*  Include file depends on local installation */
#include <SWI-prolog.h>
#include <ctype.h>

foreign_t
pl_lowercase(term_t u, term_t l)
{ char *copy;
  char *s, *q;
  atomic_t la;

  if ( !PL_is_atom(u) )
    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);
}

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

Figure 5.6: Lowercase source file

5.7.2 Compiling and Loading Foreign Code

```
sun% cc -I/usr/local/lib/pl/include -O -c lowercase.c
sun% pl
Welcome to SWI-Prolog (version 2.1.3, July 1995)
Copyright (c) 1990-1995, University of Amsterdam. All rights reserved.

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
```

5.8 Notes on Using Foreign Code

5.8.1 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.

5.8.2 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().

5.8.3 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`' 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);
```

5.8.4 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_
```

5.8.5 Compatibility of the Foreign Interface

As far as I am 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.