bool **PL_call(***term, module***)**
Call term just like the Prolog predicate `once/1`. *Term* is called in the specified module, or in the context module if module = NULL. Returns `TRUE` if the call succeeds, `FALSE` otherwise. Figure 5.3 shows an example to obtain the number of defined atoms. All checks are omitted to improve readability.

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 5.3 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.

```
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);               /* call it */

                            /* 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 5.3: Calling Prolog

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

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

functor **PL_new_functor(***atomic, 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.

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

bool **PL_unify_atomic(***term, atomic***)**
Unify a term with an atomic value.

bool **PL_unify_functor(***term, functor***)**
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 5.3.

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.

```
pl_display(t)
term t;
{ functor 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 5.2: Foreign definition of `display/1`

atomic **PL_atomic(***term***)**
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***)**
Transforms an integer from Prolog internal representation into a C long.

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

*char ** **PL_atom_value(***atomic***)**
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 **PL_functor(***term***)**
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(t1, t2)
term t1, t2;
{ if ( PL_type(t1) != PL_TERM || PL_type(t2) != PL_TERM )
    PL_fail;
  if ( PL_functor(t1) == PL_functor(t2) )
    PL_succeed;
  PL_fail;
}
```

atomic **PL_functor_name(***functor***)**
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***)**
Return a C integer representing the arity of *functor*.

term **PL_arg(***term, 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 5.2 shows a definition of `display/1` to illustrate the described functions.

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

foreign_t
my_function(a0, a1, handle)
term a0, 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 5.1: Skeleton for non-deterministic foreign functions

Analysing Terms via the Foreign Interface

Each argument of a foreign function (except for the control argument) is of type *term*. 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*)
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.

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 5.1 shows a skeleton for a non-deterministic foreign predicate definition.

5.4 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 Prolog data types used in the interface.

term 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 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 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.

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

5.5 The Foreign Include File

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

- Communicating about modules
- Printing standard Prolog warning/error messages
- Global actions on Prolog (halt, break, abort, etc.)
- Querying the status of Prolog

A C-file to be included normally defines a number of functions that implement foreign language Prolog predicates, private support functions and an installation function. The user should compile this file into a '`.o`' file using 'cc -c file ...', after which Prolog can be asked to load the file using the simplified `load_foreign/2` predicate or the more flexible `load_foreign/5` predicate. Prolog will call the Unix loader ld(1) to create an executable. It will then determine the actual size of the executable, allocate memory for it and call the loader again to create an executable that can be loaded in the allocated memory. After the executable is loaded the entry point of the new executable is called. This function should register all defined foreign predicates with Prolog.

5.3 Loading Foreign Modules

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. To simplify maintenance of packages in heterogeneous networks the system first tries whether the object file is available from a subdirectory whose name depends on the system used. The names of the subdirectories is shown below.

Directory	Machine
`sun4`	Sparc Station 1 and SUN-4
`sun3`	SUN-3
`hpux`	HP9000 running HPUX

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:

```
ld -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.

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 via 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 Portability of the Foreign Interface

The foreign language interface 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). See the introduction section for a list of systems to which the foreign interface is available.

5.2 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. 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
- Calling Prolog from C

```
:- 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`

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.

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 show the two cases in which this simple transformation does not work.

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

```
?- 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_1, consistent),
   world_1:unkown(_, fail).
```

These calls create a module called `world_1` and make the call 'world_1: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_5:export(solve(_,_)).          % exports solve/2 from world_5
?- world_3:import(world_5:solve(_,_)).  % and import it to world_3
```

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 *+Name/+Arity, ...*
: *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

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:

```
%       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 ??.

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

4.5 Using the Module System

The current structure of the module system has been designed with some specific organisations for large programs in mind. Many large programs define a basic library layer on top of which the actual program itself is defined. The module *user*, acting as the default module for all other modules of the program can be used to distribute these definitions over all program module without introducing the need to import this common layer each time explicitely. It can also be used to redefine built-in predicates if this is required to maintain compatibility to some other Prolog implementation. Typically, the loadfile of a large application looks like this:

```
:- use_module(compatibility).   % load XYZ prolog compatibility

:- use_module(                  % load generic parts
        [ error                 % errors and warnings
        , goodies               % general goodies (library extensions)
        , debug                 % application specific debugging
        , virtual_machine       % virtual machine of application
        , ...                   % more generic stuff
        ]).

:- ensure_loaded(
        [ ...                   % the application itself
        ]).
```

The 'use_module' declarations will import the public predicates from the generic modules into the `user` module. The 'ensure_loaded' directive loads the modules that constitute the actual application. It is assumed these modules import predicates from each other using `use_module/[1,2]` as far as necessary.

In combination with the object-oriented schema described below it is possible to define a neat modular architecture. The generic code defines general utilities and the message passing predicates (`invoke/3` in the example below). The application modules define classes that communicate using the message passint predicates.

4.5.1 Object Oriented Programming

Another typical way to use the module system is for defining classes within an object oriented paradigm. The class structure and the methods of a class can be defined in a module and the explicit module-boundary overruling describes in section ?? can by used by the message passing code to invoke the behaviour. An outline of this mechanism is given below.

4.4 Importing Predicates into a Module

As explained before, in the predicate based approach adjapted by SWI-Prolog, each module has it's own predicate space. In SWI-Prolog, a module initially is completely empty. Predicates can be added to a module by loading a module file as demonstrated in the previous section, using assert or by *importing* them from another module.

Two mechanisms for importing predicates explicitely from another module exist. The `use_module/[1,2]` predicates load a module file and import (part of the) public predicates of the file. The `import/1` predicate imports any predicate from any module.

use_module(+*File*)
: Load the file(s) specified with *File* just like `ensure_loaded/1`. The files should all be module files. All exported predicates from the loaded files are imported into the context module. The difference between this predicate and `ensure_loaded/1` becomes apparent if the file is already loaded into another module. In this case `ensure_loaded/1` does nothing; use_module will import all public predicates of the module into the current context module.

use_module(+*File*, +*ImportList*)
: Load the file specified with *File* (only one file is accepted). *File* should be a module file. *ImportList* is a list of name/arity pairs specifying the predicates that should be imported from the loaded module. If a predicate is specified that is not exported from the loaded module a warning will be printed. The predicate will nevertheless be imported to simplify debugging.

import(+*Head*)
: Import predicate *Head* into the current context module. *Head* should specify the source module using the <*module*>:<*term*> construct. Note that predicates are normally imported using one of the directives `use_module/[1,2]`. `import/1` is meant for handling imports into dynamically created modules.

It would be rather inconvient to have to import each predicate refered to by the module, including the system predicates. For this reason each module is assigned a *default* module. All predicates in the default module are available without extra declarations. Their definition however can be overruled in the local module. This schedule is implemented by the exception handling mechanism of SWI-Prolog: if an undefined predicate exception is raised for a predicate in some module, the exception handler first tries to import the predicate from the module's default module. On success, normal execution is resumed.

4.4.1 Reserved Modules

SWI-Prolog contains two special modules. The first one is the module `system`. This module contains all built-in predicates described in this manual. Module `system` has no default module assigned to it. The second special module is the module `user`. This module forms the initial working space of the user. Initially it is empty. The default module of module `user` is `system`, making all built-in predicate definitions available as defaults. Built-in predicates thus can be overruled by defining them in module `user` before they are used.

All other modules default to module `user`. This implies they can use all predicates imported into `user` without explicitely importing them.

In this case we would like maplist to call `extend_atom/3` in the module `extend`. A name based module system will do this correctly. It will tag the atom `extend_atom` with the module and maplist will use this to construct the tagged term `extend_atom/3`. A name based module however, will not only tag the atoms that will eventually be used to refer to a predicate, but *all* atoms that are not declared public. So, with a name based module system also data is local to the module. This introduces another serious problem:

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

action(Object, sleep, Arg) :- ....
action(Object, awake, Arg) :- ....

:- module(process, [awake_process/2]).

awake_process(Process, Arg) :-
        action(Process, awake, Arg).
```

This code uses a simple object-oriented implementation technique were atoms are used as method selectors. Using a name based module system, this code will not work, unless we declare the selectors puclic atoms in all modules that use them. Predicate based module systems do not require particular precautions for handling this case.

It appears we have to choose either to have local data, or to have trouble with meta-predicates. Probably it is best to choose for the predicate based approach as novice users will not often write generic meta-predicates that have to be used across multiple modules, but are likely to write programs that pass data around across modules. Experienced Prolog programmers should be able to deal with the complexities of meta-predicates in a predicate based module system.

4.3 Defining a Module

Modules normally are created by loading a *module file*. A module file is a file holding a `module/2` directive as its first term. The `module/2` directive declares the name and the public (i.e. externally visible) predicates of the module. The rest of the file is loaded into the module. Below is an example of a module file, defining `reverse/2`.

```
:- module(reverse, [reverse/2]).

reverse(List1, List2) :-
        rev(List1, [], List2).

rev([], List, List).
rev([Head|List1], List2, List3) :-
        rev(List1, [Head|List2], List3).
```

Chapter 4

Using Modules

4.1 Why Using Modules?

In traditional Prolog systems the predicate space was flat. This approach is not very suitable for the development of large applications, certainly not if these applications are developed by more than one programmer. In many cases, the definition of a Prolog predicate requires sub-predicates that are intented only to complete the definition of the main predicate. With a flat and global predicate space these support predicates will be visible from the entire program.

For this reason, it is desirable that each source module has it's own predicate space. A module consists of a declaration for it's name, it's *public* predicates and the predicates themselves. This approach allow the programmer to use short (local) names for support predicates without worrying about name conflicts with the support predicates of other modules. The module declaration also makes explicit which predicates are meant for public usage and which for private purposes. Finally, using the module information, cross reference programs can indicate possible problems much better.

4.2 Name-based versus Predicate-based Modules

Two approaches to realise a module system are commonly used in Prolog and other languages. The first one is the *name based* module system. In these systems, each atom read is tagged (normally prefixed) with the module name, with the exception of those atoms that are defined *public*. In the second approach, each module actually implements its own predicate space.

A critical problem with using modules in Prolog is introduced by the meta-predicates that transform between Prolog data and Prolog predicates. Consider the case where we write:

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

add_extension(Extension, Plain, Extended) :-
        maplist(extend_atom(Extension), Plain, Extended).

extend_atom(Extension, Plain, Extended) :-
        concat(Plain, Extension, Extended).
```

sleep(+*Time*)

Suspend execution *Time* seconds. *Time* is either a floating point number or an integer. Granularity is dependent on the system's timer granularity (1/60 seconds on most Unix systems). A negative time causes the timer to return immediately. As Unix is a multi-tasking environment we can only ensure execution is suspended for *at least Time* seconds.

```
loop :-
        generator,
            trim_stacks,
            potentionally_expensive_operation,
        stop_condition, !.
```

The prolog top level loop is written this way, reclaiming memory resources after every user query.

3.38 Miscellaneous

dwim_match(*+Atom1, +Atom2*)
Succeeds if *Atom1* matches *Atom2* in 'Do What I Mean' sence. Both *Atom1* and *Atom2* may also be integers or floats. The two atoms match if:

- They are identical
- They differ by one character (spy ≡ spu)
- One character is insterted/deleted (debug ≡ deug)
- Two characters are transposed (trace ≡ tarce)
- 'Sub-words' are glued differently (existsfile ≡ existsFile ≡ exists_file)
- Two adjacent sub words are transposed (existsFile ≡ fileExists)

dwim_match(*+Atom1, +Atom2, -Difference*)
Equivalent to `dwim_match/2`, but unifies *Difference* with an atom identifying the the difference between *Atom1* and *Atom2*. The return values are (in the same order as above): `equal`, `mismatched_char`, `inserted_char`, `transposed_char`, `separated` and `transposed_word`.

wildcard_match(*+Pattern, +String*)
Succeeds if *String* matches the wildcart pattern *Pattern*. *Pattern* is very simular the the Unix csh pattern matcher. The patterns are given below:

`?`	Matches one arbitrary character
`*`	Matches any number of arbitrary characters
`[...]`	Matches one of the characters specified at `...` `<char>-<char>` indicates a range.
`{...}`	Matches any of the patterns of the comma separated list between the brackets.

Example:

```
?- wildcard_match('[a-z]*.{pro,pl}[%~]', 'a_hello.pl%').

Yes.
```

gensym(*+Base, -Unique*)
Generate a unique atom from base *Base* and unify it with *Unique*. *Base* should be an atom. The first call will return `<base>1`, the next `<base>2`, etc. Note that this is no warant that the atom is unique in the system.[13]

[13] BUG: I plan to supply a real `gensym/2` which does give this warrant for future versions.