University of Edinburgh

Departments of Computer Science and

Artificial Intelligence

The Implementation of a Modular Prolog System Based on Standard ML Modules

4th Year Project Report

Brian Paxton

May 30, 1992

Abstract

This report describes an implementation of the Standard ML modules system for Prolog, originally presented by Sannella and Wallen in [SW92]. The modules system itself allows hierarchically structured programs to be constructed from parameterised components and provides a form of structural data abstraction. Also discussed, in some detail, is the interaction of the extra-logical facilities of Prolog with the modules system.

Chapter 1

Introduction

The task of this project was to implement a modules system for Prolog. The main result of the work is a working implementation of the modules system proposed by Sannella and Wallen in their paper [SW92]. This document discusses the problems involved in designing and implementing the modules system and the decisions made throughout the duration of the work. The modules system is based on the modularisation facilities of Standard ML and provides facilities to hierarchically structure programs as well as providing a form of structural data abstraction. For a more detailed discussion of the module system, see [SW92] or for a discussion of the modules system in Standard ML see [Har89].

This project aims to implement the modules system on top of an existing Prolog system. Many existing modules systems have been proposed for Prolog, but these lack flexibility and features such as data abstraction. The modules system I have implemented here addresses these issues, and I have gone on to describe how existing Prolog extra-logical predicates, such as assert/1, can be modified to operate in the new environment. Many other module systems, many implemented via a simple pre-processor which converts modular programs to Standard Prolog ones, do not discuss issues such as the run-time effect of extra-logical predicates, resulting in incomplete and inflexible systems.

The report is structured into several chapters. The first is an introductory overview to the modules system itself, which is followed by a chapter giving the formal syntax of the language and a discussion of the differences between the system implemented and the system proposed in [SW92]. After this, I briefly overview the nature of the project and discuss the two versions of the system I have implemented. Concluding chapters discuss issues such as using the new Modular Prolog and possible ideas for future Modular Prolog work. Throughout this report, the reader is assumed to have a knowledge of Standard Prolog.

1.1 Terminology

The terminology I will be using throughout this report is given below. Prolog has much terminology associated with it, much of which is contradictory (see appendix B), but hopefully the terminology I shall use highlights the nature of the modules system, by emphasizing the difference between predicates and functions, without straying too far from Standard Prolog terminology.

The need to summarize terminology here is highlighted by the modules system itself, which introduces module constructs called structures and functors. These names are used by the Prolog community already, and so I will try to avoid contradiction where possible.

A *Predicate Constant* consists of a *Predicate Symbol* and an *Arity*.

A *Function Constant* consists of a *Function Symbol* and an *Arity*.

A *Function Application* consists of a *Function Constant* and a sequence of *Terms*

A *Predicate Application* consists of a *Predicate Constant* and a sequence of *Terms*

A *Compound Term* is a *Function Application* or *Predicate Application*.

An *Atom* is a zero arity *Function Application*.

A *Number* is an *Integer* or a *Float*.

A *Term* is anything expressible in the language
(*Variable, Atom, Number, Compound Term or List*).

A Term is *Atomic* if it is an *Atom* or a *Number* (but not a *Variable*).

If the terms *predicate* or *function* appear by themselves it is an abbreviated form of *predicate constant* or *function constant* respectively.

Chapter 2

The Module System

The modules system implemented here was proposed by Sannella and Wallen for several important reasons. Firstly, although efficient implementations of Prolog are available, Standard Prolog lacks the ability to create large programs from independent modules with well-defined interfaces. The user continually has to check that predicate names are unique throughout the program as name clashes can be fatal. This is not an easy task when programs can be thousands of lines long and can be spread over several files. Secondly, the addition of facilities for data abstraction has been shown in the context of other languages to be a useful tool for writing complex programs. With data abstraction, the internal representation of a data structure can be updated with little or no alteration to existing programs — a definite advantage in larger systems. Thirdly, the addition of a modules system like the one discussed here allows a better programming style and formal development strategy to be used when constructing Prolog programs from formal specifications. Work on formal program development can be found in [RK92] where program development using this modules system is discussed.

The modules system itself consists of three types of component : structures, signatures and functors. Structures are the basic building blocks of the system, signatures define the interfaces to structures and functors are parameterised structures. I begin by introducing structures.

The following is an example of a structure 'stack1' :

```
structure stack1 =
    struct
        fun item/2 and empty/0.
        newstack(empty).
        pop(item(X,Stack),Stack,X).
        push(Stack,X,item(X,Stack)).
        isempty(empty).
    end.
```

Normally, the body of a structure begins with a sequence of declarations, followed by the actual program code for the structure. In the above example, the only declaration is `fun item/2 and empty/2` which introduces the language to be used by the code in the rest of the structure. Here `fun item/2 and empty/2` declares a function 'item' of arity 2 and a function 'empty' of arity 0. These are the only functions that may be used by the program code in the structure — using any others produces a warning message.

In addition to the declaration of functions, the modules system also allows declaration of predicates. If the line `pred test/1` were inserted into the above structure, a predicate 'test' (of arity 1) would be created, but will have no code associated with it. In this case, any calls to that predicate will simply fail. The declaration of predicates cannot be used to declare a predicate which has clauses listed later in the structure.

Signatures define the contents of a structure that are externally visible. Any constant that exists inside a structure but not in its corresponding signature is hidden. This provides a method for hiding details of actual code and to allow the creation of abstract data types.

If we extend the above example to give structure 'stack1' a signature, we would get something like :

```
signature stacksig =
    sig
        pred pop/3 and push/3 and newstack/1 and isempty/1.
    end.

structure stack2/stacksig =
    struct
        fun item/2 and empty/0.
        newstack(empty).
        pop(item(X,Stack),Stack,X).
        push(Stack,X,item(X,Stack)).
        isempty(empty).
    end.
```

The structure 'stack2' matches the signature 'stacksig' as every constant referred to in 'stacksig' appears in 'stack2'. Note however, that there is no mention of the functions 'item/2' and 'empty/0' in 'stacksig'. This is because we want the structure to provide a set of stack operations without allowing the user to get direct access to the representation of stacks. Omitting the function declarations in the signature effectively hides the functions from the outside world.

In the earlier example, no signature was given for the structure 'stack1'. This is a special case and means that any constant, predicate or function contained in the structure is visible from outside the structure.

A point worth noting here is that a constant cannot be declared as both a predicate and a function. To clarify, if a function `fun test/2` were declared in a structure, it would be illegal to go on and declare `pred test/2` or to simply give program clauses for the predicate 'test/2'. Conversely, a constant declared as a predicate, cannot be redeclared as a function later. It is possible however, to have declarations like `fun test/2` and `pred test/1` as 'test/1' and 'test/2' are considered unique constants.

Taking the idea of a function further, we can declare functions as 'fun X = Y'. This is shown in the following example :

```
structure uses_stacks
    struct
        fun item/2.
        fun newitem/2 = stack1:item.
        stack_non_empty(newitem(_,_)).
        ....
    end.
```

Here we have declared a new function 'newitem/2' in terms of the function 'stack1:item/2'. These two functions are now considered identical for unification purposes. These declarations allow functions in different structures to be compared simply by unification. However, more importantly, these declarations allow one data structure to be defined in terms of other data structures, which may exist inside other structures. This allows a simple form of datatyping in Prolog.

The line `fun newitem/2 = stack1:item` can actually be simplified to `fun newitem = stack1:item` as there is only one function symbol 'item' inside structure 'stack1' so no confusion can arise. Equality can be defined over any function previously declared (as long as the function is not hidden), so the declaration `fun newitem = item` is equally valid, and sets up an equality between 'uses_stacks:newitem/2' and 'uses_stacks:item/2'.

After defining basic structures, it is then straightforward to define others in terms of the basic ones. The simplest of these declarations is as follows :

```
structure renamedstack = stack2.
```

which in fact is equivalent to :

```
structure renamedstack = stack2/stacksig.
structure renamedstack/stacksig = stack2.
```

The other method for combining structures is to use functors. When declaring a functor, a list of arguments is given, along with the signature of each. For example :

```
functor utils(x/stacksig) =
    struct
```

```
            structure s = x.
            ismember(X,Stack) :-
                s:pop(Stack,_,X).
            ismember(X,Stack) :-
                s:pop(Stack,NewStack,_),
                ismember(X,NewStack).
        end.

    structure memberstack = utils(stack2).
```

There are a few concepts to note here. Firstly, the line **structure s = x** shows how a hierarchy of structures can be built. The structure 's' is a substructure of the parent structure. These are legal declarations inside normal structures too. Secondly, in order to use a predicate or a function in a substructure, the predicate or function name must be qualified. This means specifying a pathname to the 'home' module of the constant concerned. In the above example, `s:pop(Stack,_,X)` is an example of this. In general, structures can be nested to any depth, so paths can be of any length, and have the general form $module_1$: ... $module_N$:*name*.

A functor by itself cannot be used as a program section, but the results of applying a functor to parameter structures (a process called functor application) can be used to build new structures. The structure 'memberstack' is built in this way above. Parameters can be any structure which matches the corresponding signature in the functor heading.

A last concept to grasp in this introductory overview is the concept of sharing. Consider the following program :

```
    structure stack3/stacksig =
        struct
            newstack([]).
            pop([X|Stack],Stack,X).
            push(Stack,X,[X|Stack]).
            isempty([]).
        end.
    functor moreutils(x/stacksig) =
        struct
            structure stack = x.
            haslength(Stack,0) :-
                stack:isempty(Stack).
            haslength(Stack,Len) :-
                stack:pop(Stack,Nstack,_),
                haslength(Nstack,Part),
```

```
            Len is Part + 1.
    end.
structure one = utils(stack2).
structure two = moreutils(stack3).
functor example(x/sig1, y/sig2) =
    struct
        structure a = x.
        structure b = y.
        test :-
            a:stack:newstack(X),
            b:stack:isempty(X).

    end.
structure final = example(one,two).
```

However, there is a problem. The structures 'final:a:stack' and 'final:b:stack' both define an abstract data type which matches 'stacksig', but there is no reason why the actual implementation and internal representation of the data type is the same. In other words, we cannot expect objects created in these separate structures to unify when we do not know anything about them (a call to the predicate final:test/0 in the above program will fail). It is for this reason we introduce sharing constraints. This statement, placed after the argument list in the functor head, ensures that certain substructures of the arguments are in fact the same structure by forbidding application of the functor to inappropriate parameter structures. This ensures that the representation used by the abstract data type is consistent, and compatibility is guaranteed.

Here is the new version of the functor :

```
functor example(x/sig1, y/sig2 sharing x:stack = y:stack) =
    struct
        ....
    end.
```

This ensures that 'x:stack' and 'y:stack' inside the functor are the same structure. Once this constraint is in place, we find that any pair of structures accepted by the functor example will ensure that example:test/0 succeeds. If we adapted the declaration of structures 'one' and 'two' to

```
structure one = utils(stack2).
structure two = moreutils(stack2).
```

or to

```
structure one = utils(stack3).
structure two = moreutils(stack3).
```

we find that the program is acceptable and a call to example:test/0 succeeds. This facility makes certain classes of buggy program illegal, and acts as a debugging aid for the programmer.

This concludes the basic overview of the modules system. Examples of larger programs are given in Appendix C. Sannella and Wallen do a similar overview in more detail in their paper [SW92], and Read and Kazmierczak also use Modular Prolog in their work [RK92]. However, it is vital to point out that although the modules system introduced here is basically the same system as described in the other two papers, the syntax has changed slightly in one or two respects. A full list of syntax changes is given in the next chapter.

Chapter 3

Syntax

The following is the full syntax used by Modular Prolog in this paper. Major changes from that given in [SW92] are marked with *.

PROGRAMS *prog*

$prog ::= dec$

SIGNATURE BINDINGS *sigb*

$sigb ::= atid = sigexpr$

FUNCTOR BINDINGS funb

$funb ::= atid(plist) = strexpr$

$plist ::= atid_1/sigexpr_1, \ldots , atid_N/sigexpr_N$

$\quad\quad [\texttt{sharing}\ patheq_1\ \texttt{and}\ \ldots\ \texttt{and}\ patheq_M]$

$patheq ::= id_1 = \ldots = id_N$

STRUCTURE BINDINGS *strb*

$strb ::= atid = strexpr$

SIGNATURE BINDINGS *sigexpr*

$sigexpr ::= atid$

`sig` *spec* `end`

$spec ::=$ `pred` *atid/nat.*

`fun` *atid/nat.*

`structure` $specstrb_1$ `and` ... `and` $specstrb_N$. *

`sharing` $patheq_1$ `and` ... `and` $patheq_N$. *

spec spec'

$specstrb ::= atid/sigexpr$

STRUCTURE EXPRESSIONS *strexpr*

$strexpr ::= id$

`struct` *dec* `end`

strexpr/sigexpr

$atid(strexpr_1,\ \ldots\ ,\ strexpr_N)$

DECLARATIONS *dec*

dec ::= $atid(term_1,\ldots,term_N)$ [:- $atid_1(term_{11},\ldots),\ldots,atid_M(term_{M1},\ldots)$].

fun $atid/nat$.

fun $atid/nat = id$.

pred $atid/nat$. *

structure *strb*. *

signature *sigb*. *

functor *funb*. *

dec dec'

MODULAR PROLOG IDENTIFIERS id

id ::= *atid* | *atid*:*id*

Asterisks only show the major changes and these are discussed later.

3.1 Derived Forms

The functor binding

atid(*plist*) / *sigexpr* = *strexpr*

is equivalent to

atid(*plist*) = *strexpr* / *sigexpr*

The structure binding

atid / *sigexpr* = *strexpr*

is equivalent to

atid = *strexpr* / *sigexpr*

The declaration

inherit *atid*.

is equivalent to

structure *atid* = *atid*.

The specification and declaration

pred $atid1_1/nat_1$ **and** ... **and** $atid_N/nat_N$.

is equivalent to

pred $atid_1/nat_1$.

...

pred $atid_N/nat_N$.

The specification and declaration

$\texttt{fun}\ atid_1/nat_1\ \texttt{and}\ \ldots\ \texttt{and}\ atid_N/nat_N.$

is equivalent to

$\texttt{fun}\ atid_1/nat_1.$

...

$\texttt{fun}\ atid_N/nat_N.$

The declaration

$\texttt{fun}\ atid = id.$

is equivalent to

$\texttt{fun}\ atid/n = id.$ (Provided id unambiguously refers to a function constant with arity n).

3.2 Syntax Differences

There are differences between this syntax and that given in [SW92]. The major changes are justified in this section.

One of the more obvious changes to the syntax is that a colon (:) is now used to build module paths, and the slash (/) to signify the arity or signature of an object. This is swapping the roles of these characters introduced in [SW92]. The reason is simple. The Prolog community has long been using a '/' to specify predicate arities (for example, append/3), so it is natural to retain that role. Its role is now extended to specifying the signature of structures as well (for example, 'structure1/signature1', which means 'structure1' must match 'signature1'), allowing the colon to be used solely for the purpose of modular path identification. This change is minor, but makes transition from Standard Prolog to Modular Prolog easier.

When specifying sharing constraints of structures inside a signature, the old syntax was :

$\texttt{structure}\ specstrb_1\ \texttt{and}\ \ldots\ \texttt{and}\ specstrb_N$

$[\texttt{sharing}\ patheq_1\ \texttt{and}\ \ldots\ \texttt{and}\ patheq_N].$

This has now been changed to the form given earlier, where the sharing specification is considered separate from the structure specification :

$\texttt{structure}\ specstrb_1\ \texttt{and}\ \ldots\ \texttt{and}\ specstrb_N.$

$\texttt{sharing}\ patheq_1\ \texttt{and}\ \ldots\ \texttt{and}\ patheq_N.$

This small change was made simply to bring the syntax more up to date with the current syntax of Standard ML. The change makes no difference to the semantics of the language (except that the sharing specification must occur after the structure specification) and actually makes the syntax of the language simpler.

The syntax of declarations of structures, functors and signatures have been altered as well. The form of these declarations was originally :

> **structure** $strb1_1$ **and** .... **and** $strb_N$.
>
> **signature** $sigb1_1$ **and** .... **and** $sigb_N$.
>
> **functor** $funb1_1$ **and** .... **and** $funb_N$.
>
> where $N \geq 1$.

but this has now been simplified to the restricted case where N = 1. This change reduces the complexity of the syntax, but does not reduce the expressive power of the language. This will now be justified.

The token 'and' is used in Standard ML to allow definitions of more than one item to be defined together. In the case of functions, it is used so that mutually recursive functions can be created. For example :

```
fun x(arg) = y(arg)          (* An ML program. *)
and y(arg) = x(arg) ;        (* Silly example. *)
```

However, when extended to module constructs, recursively defined constructs are disallowed as one of the restrictions of the modules system is that constructs have to be defined before they are used (see [SW92], section 2.8). Therefore, the following program is illegal

```
structure one =
    struct
       structure t = two.     % Recursive.
       ....
    end
and two =
    struct
       structure o = one.     % Recursive.
       ....
    end.
```

Since recursively defined constructs are illegal, any two structures defined as a sequence separated by 'and's is equivalent to a sequence of separate structure declarations, as long as the same order in maintained. Therefore, the change to the syntax does not alter the expressive power of the module syntax.

The syntax of declarations has been extended to include `pred atid/nat`. This is because the module language insists that all predicates and functions are declared before they are used.

This means that in certain cases it is of convenience to declare a predicate which contains no clauses, and so this declaration creates an 'empty' predicate in the database. However, more importantly, this means that the declaration before use constraint is not violated. This declaration also ensures that the run-time system does not generate 'Unknown Predicate' errors when an attempt is made to call it (calls simply fail). Database programs are the major class of programs which can use this facility to their convenience. Take the following example :

```
structure database =
    struct
        pred data/1.
        add(X) :-
            assert(data(X)).
        list :-
            data(X),
            write(X), nl,
            fail.
        list.
    end.
```

Without the predicate declaration `pred data/1`, the programmer would have to insert a clause `data(_) :- fail` in order to prevent 'Unknown predicate' errors if 'database:list/0' was called when no data had been added and to ensure that the program was a valid one.

None of these changes in syntax reduce the power of the modules system, but attempt to make the transition from Standard Prolog to Modular Prolog simpler (by retaining some Standard Prolog conventions) and attempt to bring the modules system more up to date with current versions of the Standard ML modules.

3.3 Further Changes

This section covers the changes made to the proposed system that do not fall into the category of syntax changes.

The first change, relating to the semantics of the language, has to do with the treatment of declarations like :

```
structure a = b/bsig.
```

In the old semantics, this declaration created a new structure 'a', completely distinct from structure 'b'. This is not really intuitive, since we are effectively defining an equality between structures. It seems to make more sense for structure 'a' to simply be another name for structure 'b'.

A good example of where the original semantics would give undesirable results is given in the following program :

```
signature datasig = ... .
structure data/datasig =
    struct
        ....
        data(1).
        data(2).
        list_data :-
            data(X), write(X), nl, fail.
    end.
structure test =
    struct
        structure testdata = data/datasig.
        another_list_data :-
            data(X), write(X), nl, fail.
    end.
```

If we assert new data into the structure 'data', we find that test:another_list_data/0 and data:list_data/0 output different data. If the line `structure testdata = data/datasig` were replaced by `structure testdata = data`, we would find the outputs were the same. The addition of a signature constraint should not affect the results so dramatically. The semantics were therefore changed in this respect and remain consistent with the semantics of Standard ML.

Abstractions were introduced in [SW92] but are not implemented in this Modules System as they are of questionable value in Prolog. In Standard ML, abstractions are used to hide actual datatypes inside a structure. i.e. compare the following :

```
                signature SIG =
                  sig
                    type t;
                    val x:int -> t
                  end;

structure S:SIG =           |    abstraction S:SIG =
  struct                    |      struct
    type t = int;           |        type t = int;
    fun x a = a             |        fun x a = a
  end;                      |      end;
```

```
                           |
- S.x(3);                  |    - S.x(3);
> val it = 3 : int;        |    > val it = - : S.t
- S.x(3):int;              |    - S.x(3):int;
> val it = 3 : int;        |    Error: expression and constraint
                           |        don't agree (tycon mismatch)
                           |      expression: S.t
                           |      constraint: int
                           |      in expression:
                           |        x (3)
```

The abstraction limits all information available about a structure to that specified in the signature. Therefore, the second example produces an error because the datatype 's.t' is known outside 's' and the datatype 'int' is known, but the information that 's.t' is in fact equal to 'int' is hidden inside the abstraction.

This has no real meaning in Prolog since there is no strong notion of types in Prolog and it is sufficient to simply hide predicate and functions. [SW92] tried to approximate ML abstractions in Prolog but results were not really satisfactory.

In the original paper, 'open' declarations were proposed. If an `open Y` declaration was made inside a structure X, the signature of structure Y would be merged with X's signature, allowing X to use the predicate and functions of structure Y without explicit qualification. However, this is simply a programming convenience and can be avoided by simply performing an `inherit Y` and qualifying all references to Y. It is for this reason that 'open' declarations were not implemented in my version of the modules system.

In Standard Prolog there are two forms of consulting : consult and reconsult. If a file is loaded that contains clauses for a predicate already defined in the database, consult will merge the two sets of clauses, but reconsult will erase the old existing clauses before loading the new ones. To clarify, imagine we have a file called 'test' which contains the following :

```
data(one).
data(two).
```

If we consult the file twice we get the following in memory

```
data(one).
data(two).
data(one).
data(two).
```

whereas if we reconsult the file twice we get

```
data(one).
data(two).
```

The technique used when reloading modular programs is in fact the same technique as that used by Standard ML. In Standard ML, if a structure, signature or functor is reloaded, it supersedes the existing definition, but the existing definition is not removed. For example, suppose the structure 'test' is declared and several other structures use that version of the declaration. If a newer version of 'test' is loaded, it replaces the old version, and any new structures using the structure 'test' use the new version. However, the existing structures that used the old version of 'test', continue to use the old version, not the new. This idea is used in Modular Prolog : a module construct is totally defined in terms of the most recent version of other constructs.

In this respect, the new modular version of consult is something in between a Standard Prolog consult and reconsult. In the new consult, old versions of predicates defined at the top level are merged with any new clauses that may be loaded. However, reloading a structure generates a completely new and distinct structure, so any predicates that are reloaded inside that structure are actually distinct predicates from the ones that were previously loaded. In this respect, the database is merged (nothing is ever removed).

Notice that a modular version of consult which actually tries to merge clauses for predicates declared inside structures is in fact impossible to implement. Take the following program, which corresponds to a complete file :

```
structure a =
    struct
        ... some predicates ...
    end.

structure b = a.

structure a =
    struct
        ... some more predicates ...
    end.
```

If the file was consulted twice, and we wanted each predicate in the file to have its clauses merged, how would the consult routine know that when we read in the first version of structure 'a' for the second time, that that structure was the same as the first structure 'a' loaded previously, and not the last structure 'a' loaded ? This is why a merging consult is impossible.

The modules language discussed here introduces a powerful program construction language, with basic datatyping through the use of function and predicate declarations. However, the original semantics enforced the constraint that all predicates and functions had to be declared before they were used. This is not ideal in a Prolog environment. To illustrate this point, consider Prolog atoms which are regarded as zero-arity functions. In many cases an atom is used as a function, but in equally many other cases, it is not. Atoms are used to name files and structures, and are used as text to be displayed. It is not practical for the user to declare every piece of text that is contained within a program, nor is it practical to declare arguments to all the following predicates — structure/2, consult/1, abolish/2, see/1, tell/1, listing/1, display/1, print/1 — all of which take atoms in some form as arguments, but none can really be described as a function.

To make the modules language more usable in this respect, I have relaxed this declaration before use constraint, and now accept all predicates and functions, even if they have not been declared, displaying only a warning message instead or rejecting undeclared input. Checks are still made on qualified predicates and functions (predicates and functions not in the current structure) as these are still subject to hiding constraints. Undeclared predicates and functions are not considered during signature matching, so any constant which must match a signature specification must still be explicitly declared.

These changes make the modules system much easier to use, and in fact it was personal experience constructing my own Modular Prolog programs that suggested the change.

As mentioned in the overview of the modules system, all predicates and functions now have to be distinct — a *name/arity* pair inside a structure must define a predicate or a function, but not both. This approach provides a cleaner programming style, without greatly reducing flexibility. This decision and its consequences are discussed later.

This concludes the list of changes made to the modules system. For the changes made to the semantic equations given in [SW92] to compensate for the changes listed in the previous two sections, see the appendix A.

Chapter 4

Overview of Project

Having overviewed the modules system, I now discuss the implementation of it. The modules system in this project is a simple extension to Standard Prolog, so is implemented on top of an existing version of Prolog.

Firstly a brief overview of the underlying Prolog system is required. The Prolog system I use is SB-Prolog, an interpreter and compiler package based on the Warren Abstract Machine[1]. The Warren Abstract Machine (hereafter referred to as the WAM) is an abstract Prolog instruction set suitable for software, firmware or hardware implementation. The WAM can be regarded as the instruction set of the ideal “Prolog Machine” and is introduced in a paper by D.H.D.Warren [War83].

The Prolog system itself can be divided into two levels — a C level and a Prolog level. The C level consists of the WAM emulator and a small library of very low level system predicates including input and output predicates. The rest of the system is written in Prolog itself, so user programs and the runtime system coexist at the Prolog level throughout. This architecture is shown in Figure 4.1.

SB-Prolog itself has its own notion of modules, but these are designed for a special purpose. The Prolog level system is divided into a set of library files, and only a few of these are loaded at Prolog boot time — the rest are loaded only when needed. The system knows which library file to load when a predicate is required, as it is given a list of predicates each library file exports. If a predicate is required and is undefined, a check is made to see if it appears in any library file's export list. If it does, that file is loaded and the erroneous call is retried. These are SB-Prolog modules — the components are files and the modules system is only used to drive the dynamic loader. There is therefore no problem installing a new user-level modules facility on top of this — both can coexist without conflict.

The implementation of Modular Prolog proceeded in two phases. The first was a simplified

[1] This project does not alter the compiler in any way, only the interpreter, so the compiler package is ignored throughout this discussion.

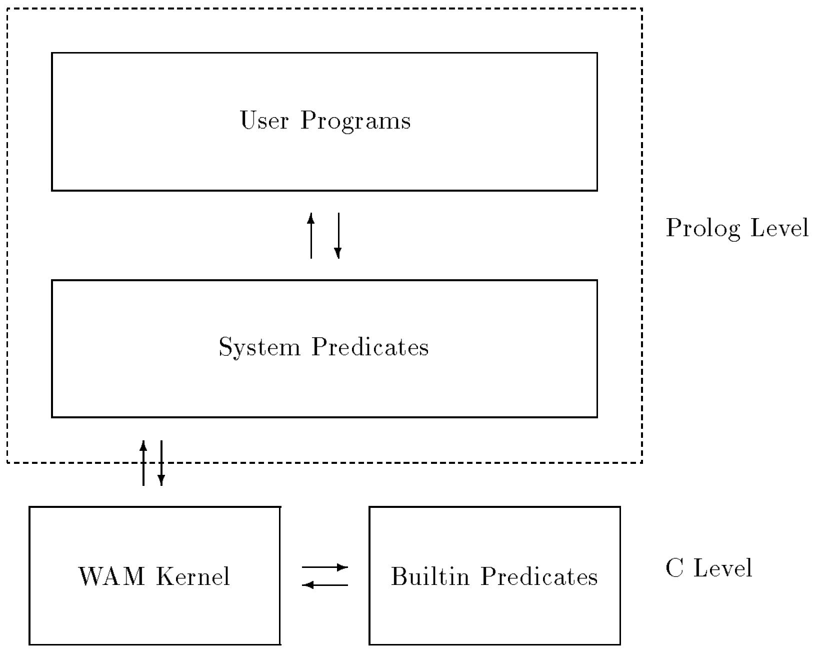

Figure 4.1: The Structure of SB-Prolog

version of the system, built as a test-bed and runs simply as a pre-processor system ; this is described in chapter 5. The second was a full version of Modular Prolog ; this is described in chapter 6.

Chapter 5

The Preprocessor Version

Early work on this project produced a very simple pre-processor version of the system which read in a Modular Prolog file and converted it to a Standard Prolog program in memory. A simple interface allowed command line calls to be made.

I will not discuss the implementation of this version of the system in any depth, to leave more space for a discussion of the major work on the full system. This chapter is only intended to highlight problems with a simple preprocessing version of the system, which provided the ideas and direction for the work on the full implementation.

This system basically implemented the semantic equations given in [SW92], and included only a few of the changes made to the modules environment discussed in sections 3.2 and 3.3. For example, there was no reloading of structures, and structure and sharing specifications were still on one line.

In the preprocessing version a function was built which mapped every unique predicate and function in the Modular Prolog program to a unique internal name. This mapping was used to convert the original program to an equivalent Standard Prolog program. The internal names had the form *'functionN'* or *'predicateN'*, where N is a unique number. These names have no meaning to the user at all and are only machine readable. The resulting programs, when translated into Standard Prolog, looked something like the following :

```
predicate9(function7).
predicate39(V0,V1) :-
    predicate13(V1,V0).
predicate39(V0,V1) :-
    predicate14(V1,V2),
    predicate39(V0,V2).
predicate39(V0,V1) :-
    predicate15(V1,V2),
    predicate39(V3,V2).
```

```
predicate10(function8(V0,V1,V2)).
predicate13(function8(V0,V1,V2),V0).
predicate12(V0,V1,V2,function8(V0,V1,V2)).
predicate11(function7).
```

The actual subset of Modular Prolog this system could handle was much smaller than that of the full system described in the rest of this report. This was because system predicates like name/2, =../2, functor/2, read/1, assert/1, retract/1, see/1, write/1 and listing/0 were not integrated with the modules system, and produced inconsistent results. This had the effect of reducing the usability of this system enormously (in fact, almost completely!).

A couple of the problems encountered in this system are discussed here. Firstly, consider the following example :

```
structure one =
  struct
    fun a/0.
    fun a/2.
    test(X) :-
        X =.. [a,1,2].
  end.
```

To pick one particular scenario, the function one:a/0 is stored internally as 'function23'/0 and the function one:a/2 is stored internally as 'function24'/2. The call to =../2 should create the internal form of one:a(1,2). However, since =../2 does not interact with the modules environment, =../2 will build the function 'function23'(1,2) which is not the internal form of one:a(1,2).

Other problems are associated with input and output. Write/1 always prints the internal form of a predicate or function constant, and so its output is not human-readable. Similarly, read/1 can only accept input in internal form.

This system was slow, cumbersome, memory-intensive and could only support very small Modular Programs (up to about a page long) before the system ran into memory problems. This, together with the lack of integration between the system predicates and the modules environment made the system impractical to use.

However, programs which did not use any of the extra-logical predicates of Prolog worked correctly and the system allowed me to experiment with these simple modular programs. The work on this version of the system also allowed adequate time to establish a close understanding of the semantics of the modules system (given in [SW92]), an area I was unfamiliar with when I began this work. This work was also valuable in that its foundations were later reused in the version of consult/1 in the full Modular Prolog system.

Chapter 6

The Full Modular Prolog

This chapter describes a full implementation of Modular Prolog. This system was successfully implemented on top of SB-Prolog, and is fully functional. There were several major issues involved in the design and construction of this system. These were the representation of the module environment, identifying objects local to structures, the treatment of predicates and functions (and the distinction between them) and the integration of the extra-logical predicates of Standard Prolog into the modules system. These are discussed in this chapter.

6.1 Representation of Module Environment

The paper by Sannella and Wallen [SW92] gives the detailed semantic equations which describe the semantics of the modules environment. The implementation of consult/1 follows this semantics closely and the actual data structures used in the semantics are almost identical to those implemented. For example, Sannella and Wallen state that a signature is denoted by a triple $< substrs, preds, funs >$ where :

$substrs$: substructure names $\rightarrow$ “internal” structure names
$preds$: predicate names $\rightarrow$ “internal” predicate names
$funs$: function names $\rightarrow$ “internal” functions names

This particular data structure is stored in consult/1 in the form :

sig([$substructure \rightarrow internalname$, ...],
[$predicate \rightarrow internalname$, ...],
[$function \rightarrow internalname$, ...])

The list of all signatures, structures and functors currently defined (called “environments”) are simply lists, which are eventually asserted into the database. Even the architecture of the consult/1 program itself clearly shows the areas of the program which implement specific semantic equations. This similarity between the semantics and the implementation makes the task of proving the program is correct easier — important for a first working version.

However, there are disadvantages with this technique as well. Firstly, the denotational semantics used to specify the system is elegant and compact, but does not lead directly to an efficient implementation. An efficient implementation may stray far from the style introduced by the semantics, and that implementation may be hard to find. Secondly, the data structures themselves are very large and unmanageable. I introduced some basic data compression techniques into the consult/1 program to reduce environment size, and to use the Prolog database more effectively. Again, this affects efficiency (time efficiency is increased considerably as much of the data is removed to the database, reducing the amount of data that is carried internally in arguments ; space efficiency is not altered at all, as I simply add data to the database during processing instead of after processing has finished). Lastly, the environments given in the semantics only store the top level module constructs — the others are only implicitly given as parts of the top level ones. This architecture makes the processing of sub-structures for operations like listing/0 slightly tricky, and shows the need for a better data structure.

However, criticisms aside, the data structures are represented clearly in the same form as in the semantics. The module constructs defined are eventually stored in clause form in the database and define the predicates $module_structure/5, $module_signature/5 and $module_functor/10. The clauses are stored in reverse order in the database simply so that a call to any of these predicates returns the module construct most recently defined.

The result of a call to consult/1 is in two parts. The first is the environment which is built into the predicates discussed above. The environment actually stores more than this however, as declared functions ('fun X' or 'fun X = Y') are stored in clauses for the predicate $declared_function/1, and 'fun X = Y' declarations are stored in clauses for the predicate $mapped_function/4. This data is used by the extra-logical predicates. The second part is the actual program code. This is a set of Standard Prolog clauses which is equivalent to the Modular Prolog program. This shows the close resemblance between the pre-processor version of the program and the full Modular Prolog.

There are two structures declared automatically at Prolog boot time — the 'pervasive' and the 'root' structures. The pervasive structure is the system structure and its signature contains predicates like assert/1 and write/1 and functions like '[ ]'/0 and user/0. The root structure is the top level structure in which command line queries are evaluated.

The modules environment itself can only be modified through the use of consult/1. Extra-logical predicates like assert/1 have been modified to allow the user to alter the contents of the individual structures, but structures, signatures and functors can only be declared through consult/1. The environment itself is an ever-growing structure, and there is no way in which the user can "retract" a structure.

6.2 Identifying Objects

Consider the following program :

```
structure a =
    struct
        fun a/1.
    end.
structure b =
    struct
        fun a/1.
    end.
```

The function a/1 is defined twice but in different structures. Since we wish to distinguish between the two, we need to internally represent all objects as a (*structure tag* x *name* x *arity*) triple, instead of the simpler Standard Prolog method of representing them as a (*name* x *arity*) pair. If we set up a function (*structure tag* x *name* x *arity*) → *object tag*, we can uniquely identify each object in the system by an object tag (it was this kind of technique that was used in the pre-processor version of the system).

The modules system itself creates an extra problem as it allows definitions like 'fun X = Y'. The basic trouble is that 'fun X = Y' sets up a many-to-one function from actual name (X) to unification name (Y). The language must support this facility of unification over functions with different names. However, although vital, this facility is not very commonly used and is generally never more than a couple of levels deep[1], so a complex representation strategy is not necessary.

Standard SB-Prolog itself implements a system where a (*name* x *arity*) pair maps uniquely to a single entry in the program symbol table. This means that even constants with the same name do not map to the same symbol table entry if they have different arities. The representation makes unification easy — two objects can only ever unify if they correspond to the same symbol table entry. This, of course, makes unification of items a simple pointer comparison.

The task of implementation therefore includes extending a (*name* x *arity*) pair to a (*structure tag* x *name* x *arity*) triple which maps into the symbol table. The result must ensure that X and Y still unify if we declare 'fun X = Y', must be straightforward to implement and must not hamper the speed of Prolog too much. There are several possibilities :

1. The first technique involves modifying the SB-Prolog symbol table entries so that each object is not only represented by name and arity, but also by home structure tag. This would not affect much of the Prolog run-time system and would reduce implementation details from the Prolog level to the C level.

[1] With the declarations 'fun X = Y' and 'fun Y = Z', Y is one level deep and Z is two levels deep.

In order to allow the handling of `fun X = Y` declarations, each symbol table entry would also require a new field `unifies_with` which is a pointer to the symbol table entry for the function it is been declared equal to. The unification algorithm for the system would be modified to use the `unifies_with` pointer.

However, although this would lead to a fast and economical system which shows off the module system nicely (by identifying each object using a structure tag as well), it requires changes at the C level which removes all chances of porting the system to other versions of Prolog. Also, care must be taken when handling functions declared as follows :

```
fun a/2.
fun c = a.
fun b = a.
```

If we set the `unifies_with` pointer of c/2 to point to a/2 and the `unifies_with` pointer of b/2 to point to a/2, then we must encode the unification algorithm in such a way as to ensure that c/2 and b/2 unify, even they they are only indirectly connected through `unifies_with` pointers to a/2.

2. Another possibility is to store each object as the single name 'tag1_name1$$tag2_name2', where 'tag1' and 'tag2' are structure tags.

   For example :

   $$\underbrace{1\_test}_{1}\$\$\underbrace{202\_test2}_{2}$$

   (1) This part used for printing, etc.

   (2) This part used for unification.

   This technique represents the home structure tag of the object, and the `unifies_with` field explicitly in the text of the name. The arity of course will be present in the symbol table entry anyway. If the name of the object is to be displayed, *name1* is extracted from the name and displayed. If the object is to be unified with another, *tag2* and *name2* will be used. Any functions not involved in `fun X = Y` declarations are stored in the simpler form 'tag_name', with no need to include a unification part.

   However, there are many disadvantages with this method. It is very cumbersome to implement and not very elegant. More importantly however, unification will no longer be a simple pointer comparison, but will involve comparing characters of the name, to extract the unification information. Since Prolog relies on unification, the performance of Prolog will fall by at least an order of magnitude, which is far from desirable.

   This technique also suffers from the same problem as in the first technique above when given function declarations of the form `fun c = a` and `fun b = a` — the unification algorithm has to take great care when unifying the functions c/2 and b/2.

3. On a similar vein as above, each object could be stored in the form '*tag__name*', but here the structure tag and object name correspond to the unification name of the object — each function is stored in its unification form only. For example, a function is declared `fun one/2 = two`. When the function one/2 is created during a consult, or by some extra-logical predicate, the function that is actually built is two/2. Therefore if the function one/2 is ever decomposed or printed, the name 'two' is printed instead of 'one'.

   Note that this technique does not suffer from the same problem as the above two techniques when given declarations of the form `fun c = a` and `fun b = a` simply because a/2, b/2 and c/2 will all be stored internally as a/2.

   This method ensures that unification is guaranteed over functions with different names (as they are all actually stored internally as the same function) without altering the Prolog unification algorithm and is relatively straightforward to implement.

   However there are a few obvious problems. The major one is simply that under certain circumstances it is tricky, if not impossible to get back exactly what you typed, as a function may not be stored as you expect. This means that printing and decomposition can only produce the 'best' solution it can, which is not always what is required.

4. Lastly, a system similar to the preprocessor system could be scaled up to the full system. Here every structure, predicate and function is stored as a unique internal name. A function would be created to map each unique modular name to its internal name. This of course means that we need to store function tables from modular name to internal name for use by all the extra-logical predicates. These functions are large, would be slow in evaluation and consume a lot of memory. Also, every extra-logical predicate would have to be heavily modified and performance of each would fall. In other words, this is totally impractical.

These factors were all taken into account, along with one overriding factor — if the changes to the Prolog system only alter the Prolog code for the language and not the underlying C code, then the system can be more easily ported to any other existing Prolog system. Since items 1 and 2 above require alteration to the low-level C code, and 3 and 4 do not, the choice finally implemented was 3. A full discussion of this method is given later.

In Modular Prolog however, the pervasive structure is treated slightly differently. Instead of tagging each symbol in the structure, we simply leave all symbols untagged. This is how pervasive terms are recognised — if there is no tag, then the symbol is pervasive. This has several advantages. The first is that the original compiled version of the Prolog level system need not be recompiled, but more importantly, the pervasive structure is used as a kind of trash can for functions which belong to no other structure in Modular Prolog. For example, if a function is created which contains no structure tag, it is hidden inside the pervasive structure by the pervasive signature, so remains out of the user area of memory and safe. Modular Prolog makes full use of the fact that untagged functions are hidden and many of the extra-logical predicates use this 'trash can' facility.

6.3 Predicates and Functions

One aspect of this modules system that is emphasized deliberately is the distinction between predicates and functions. From a logical point of view, to be valid, a clause has to have the following form :

$$\begin{aligned}&\mathit{predicateapplication} \text{ :-}\\&\quad \mathit{predicateapplication}_1 \ ,\\&\quad \ldots. \ ,\\&\quad \mathit{predicateapplication}_N .\end{aligned}$$

where the arguments to each predicate application do not contain any further predicate applications.

However, most Prolog implementations do not have such a rigorous syntax and allow functions and predicate applications to be placed anywhere. In SB-Prolog for example, predicate and function applications can be used in any way you like, with a distinction made between them only if a call is attempted to one. For example, if you pass a function application to call/1 it will fail with an 'Unknown predicate' error, whereas is you pass call/1 a predicate application, you will never generate this error. This is the only difference between them and in fact, both predicates and functions are stored in exactly the same way.

As mentioned earlier, I have chosen to impose the constraint that a constant cannot be both a predicate and a function. This is justified here :

We have module constructs that require functions and predicates to be explicitly declared, so do we enforce a syntax as rigorous as the logical one, or do we follow the Prolog tradition of allowing the programmer "to do what he/she likes"? There were several possibilities to choose from :

Given a constant p :

1. p can't be used as a function and a predicate simultaneously (here, 'function' and 'predicate' have the logical meanings — function applications can be used as arguments to predicate applications, predicate applications cannot). If p is a function it can only be used as a function. If p is a predicate it can only be used as a predicate.

   This implies that the constraint that arguments to predicate applications must be function applications has to be relaxed (as you have to pass a predicate application to call/1 for example).

2. p can be used as both a predicate and a function. This means that you could enforce the constraint that an argument to a predicate application must not be another predicate

application. If you want to assert a clause for the predicate p into the database, for example, you have to declare p as both a function (so it can be passed to assert/1) and a predicate (so you can call it).

3. As 2, but ‘clever’.

   ‘Clever’ means that arguments to predicates like assert/1 can be predicate applications, but no others can. Further, if the user defines something like :

```
update(X,Y) :-
        retract(X),
        assert(Y).
```

   the system will work out that the arguments to update/2 can be predicate applications and use this new information when checking programs. This implies that predicates like update/2 must be declared before any calls to that predicate are processed (or the system will not recognize the new constraints).

The version I have implemented in my system is the first. Once a constant is declared as a function, it is always a function (in the module in which it is declared). The second version is rather clumsy and the third requires a lot of processing and requires knowledge of system predicates which can accept predicate application arguments (this is system dependent knowledge to some extent).

Relaxing the constraint that arguments to predicate applications cannot be further predicate applications is important. First, it allows free use of assert/1 and retract/1, as well as user defined predicates like update/2 (given above), but also makes the translation stage during a consult simpler — if you know that a compound term is a function application or a predicate application, but not both, it is a simple task to ensure that the correct internal form of the compound term is used. Another added advantage is that if consult/1 knows at consult time whether a compound term refers to a predicate or a function, it can do a certain amount of program checking. For example, we can check that arguments to call/1 should always be predicate applications and not function applications (which will always fail anyway). This extension, if implemented, would be able to check for certain common runtime errors in large programs. (This is not implemented in the current version of the system).

After making this decision I attempted to implement an environment in the Prolog system which continued to emphasize the distinction between predicates and functions. I first of all removed system predicates like structure/1 which refer to both function and predicate applications and replaced them by function/1 and predicate/1. These predicates have the same basic semantics as structure/1 but have been extended — function/1 succeeds only if the argument is bound to a function application and predicate/1 only succeeds if the argument is bound to a predicate application.

In SB-Prolog, if you define a function, say 'fun test/0', it is regarded as a function by SB-Prolog. However, if you perform an 'assert(test)', then SB-Prolog regards test/0 as a predicate. This is somewhat counterintuitive; if you have declared an object as a function, then you expect it to remain a function. To prevent these minor irregularities, I have now placed constraints on the use of the assert and retract family of system predicates — arguments to these predicates can no longer be function applications of functions declared during a consult. In other words :

```
| ?- consult(user).
[Opening user]
fun test/0.
Updating database ...
[Closing user]
yes
| ?- assert(test).
*** Error : Cannot assert test/0 it has been declared as a function
no
```

Note that if test had not been defined by a `fun test/0` declaration, the following would happen :

```
| ?- function(test).
yes
| ?- assert(test).
yes
| ?- predicate(test).
yes
| ?- function(test).
no
```

The first call to `function(test)` succeeds as test/0 is created as a function by SB-Prolog as a side-effect of parsing. Any unknown compound term that is created (by parsing input or by some extra-logical predicate) is regarded as a function by SB-Prolog.

This restriction on the assert family is to stop unexpected results arising when a function declared as a function by a `fun X` or `fun X = Y` declaration suddenly becomes a predicate.

6.4 Extra Logical Predicates

One of the main aims of this project was to experiment with the extra-logical facilities in Prolog, and to find a suitable method of integrating them with the modules environment. This is a challenging problem because it is at this point we discover whether the choice of internal representation of modular constants is the correct one, and more importantly, by integrating the extra-logical predicates with the modules environment we turn the Modular Prolog system into a full, usable system, rather than a toy one (as the pre-processing version was).

The next few sections describe various types of extra-logical predicate and how they interact with the module environment. I begin by describing term manipulation predicates such as =../2, functor/3 and name/2, then predicates that have been extended to operate in remote structures, such as call/1, assert/1 and retract/1 and then discuss input and output predicates such as read/1 and write/1. I complete this discussion by overviewing other predicates that are available but do not warrant detailed discussion.

6.4.1 Term Manipulation

There are several "term manipulation" predicates within Standard Prolog, predominantly =../2, functor/3 and name/2. Since I have settled on an internal representation of functions where functions are actually stored as their unification form, these system predicates have to reflect this.

It is important to point out at this point that although programs are rigorously checked by consult/1 when they are loaded to ensure that all functions that are used are declared, no such checks are performed by the other extra-logical predicates. If a call is made to an extra-logical predicate to create a function, that function will be created regardless of whether it has been declared in advance or not. This freedom is important when using Prolog to its full power.

I begin this discussion by defining some terminology. If we have a declaration

```
fun X = Y.
```

we call Y the *unification form* of X and call X the *print form* of Y. We can then define functions from print form to unification form and from unification form to print form, which can be used by the above term manipulation predicates for extracting the correct forms of a function. However, there are problems to overcome when building these functions. Since we can declare functions such as the following

```
fun X.
fun Y = X.
fun Z = Y.
```

we have to ensure that the function from print form to unification form selects the unification name to be the transitive closure of the declarations. Here, for example, the unification form of Z is X. This example also shows a problem with the function from unification form to print form — there may be more than one equally plausible result. For example, the print form of X could be X itself, or Y, or Z. To select the 'best' solution out of the possibilities I use the following rule: 'always select the most recently declared print form for a given unification form out of those declared in the current structure'. So, for example, the print form of X is Z in the above example. Whenever a term is created, the function from print form to unification form is applied to find the correct form to use, and whenever a term is decomposed, the function from unification form to print form is used.

This is exemplified by the following example :

```
structure one =
  struct
    fun a/2.
    fun b = a.
    ....
  end.
structure two =
  struct
    fun c = one:a.
    ....
  end.
structure three =
  struct
    ....
  end.
```

If either of the functions one:a/2, one:b/2 or two:c/2 is created, the unification form is one:a/2 and so one:a/2 is the actual function created. However, if a term built using one:a/2 is decomposed, the result depends on the structure in which the decomposition is made. If the term is decomposed inside structure 'one', the most recently declared print form of one:a/2 is returned (which is one:b/2). If the term is decomposed inside structure 'two', two:c/2 is returned instead. If no valid 'fun X = Y' declarations apply, as in structure 'three', then one:a/2 is simply returned.

The extra-logical term manipulation predicates use these ideas. Take for example the predicate functor/3. (Due to the unfortunate name of this predicate, which has modular connotations, it has been renamed compound/3 in Modular Prolog). In Standard Prolog, a call to functor/3 of the form

```
| ?- functor(Function,Name,Arity)
```

has the declarative reading 'the function application Function has name Name and arity Arity'. In Modular Prolog, this has the same meaning, but the implementation is more complex. For example, if we make the call

```
| ?- compound(test(1,2),Name,Arity)
```

we must remind ourselves that test/2 is stored internally as its unification form. In order to retrieve the name of test/2 again from the unification form, we must use the function from unification form to print form to get the correct form to perform the compound/3 operation on (note that this is not guaranteed to get back test/2 itself). After performing the operation we must consider the fact that the resulting name is itself a function test/0, and so we must apply the function from print form to unification form for that function, in order to return the correct result. This processing is performed completely automatically by the Prolog system and the user, for most purposes, only requires a passing knowledge of these ideas to ensure that these predicates are used correctly.

This seems somewhat complex, but since 'fun X = Y' declarations are not that common, these checks are very often trivial. The predicates name/2 and =../2 follow a similar reasoning.

Considering a larger example, take the following program :

```
structure three =
  struct
    fun aa/2 and bb/0.
    fun bb/2 = aa.
    fun cc/2 = bb.
    test :-
      X =.. [bb,1,2],
      X =.. [bb,1,2].
    test(X) :-
      Y =.. [bb,1,2],
      Y =.. [X,1,2].
  end.
structure four =
  struct
     inherit three.
     fun ff/0.
     fun ff/2 = three:bb.
     test :-
       X =.. [ff,1,2],
       X =.. [ff,1,2].
     test(X) :-
       Y =.. [ff,1,2],
       Y =.. [X,1,2].
   end.
```

Briefly summarizing the results, we find that a call to three:test/0 succeeds, but a call to three:test(X) unifies X to three:cc/0 (and not three:bb/0) as the unification form of three:bb/2 is three:aa/2 and the print form of three:aa/2 is three:cc/2. A call to four:test/0 will succeed and a call to four:test(X) binds X to four:ff/0 as the unification form of four:ff/2 is three:aa/2 and the print form of three:aa/2 is four:ff/2 (as we are now in the structure 'four').

From an implementation point of view, there is a difficulty here — each of the predicates mentioned above require knowledge of the structure in which they are used in order to work out the results of the function from unification form to print form correctly. This is implemented by extending each of the above predicates to an equivalent form with an extra argument, which corresponds to the current structure tag. Since this is rather awkward for the user to handle, the Prolog system automatically places the extra argument into the calls to the predicates that require it. Any program clauses given to consult/1, call/1, assert/1, retract/1 and the command line undergo these automatic transformations so that the user does not need to consider them. This does mean however, that a program may look different from the original form typed in (the system predicate listing/0 will always list the transformed version), but it is a necessary addition. (There are other predicates in the system which insist on extra structure tag arguments too — these are mentioned later.)

This is the major additional complexity created by my representation of functions. However, as mentioned earlier, 'fun X = Y' declarations are rare, and the predicates that manipulate them even rarer. Other representations I have considered suffer from other problems, and so my method seems satisfactory.

There is further complexity involved in the term manipulation predicates — the pervasive structure. Since the pervasive signature is implicitly imported into every structure and since its contents can be used without qualification, the term manipulation predicates must recognise when a pervasive compound term is built or decomposed and take the necessary action. For example, consider the call :

```
X =.. [assert,test]
```

This would build the predicate application 'assert(test)'. However, the function assert/0 used in the argument is not pervasive, but assert/1 is. The code for =../2 must spot this and build the result appropriately (i.e. ensure that assert/1 is untagged which identifies it as pervasive) so that a call to assert can be made.

The complexity of these predicates is a direct result of the modules system. However, this discussion is required only for implementation purposes — all the above processing is performed automatically by the system.

6.4.2 Remote Structure Operations

There are several system predicates that can be naturally extended to operate on structures other than the one in which the call was made. These are the database manipulation predicates assert/1 and retract/1 and the predicate call/1. Extensions to these predicates were proposed by Sannella and Wallen in [SW92]..

Each of the above predicates now accepts a structure tag as an extra argument and the operation they perform is done with respect to the structure indicated by that tag. Say the structure tag of a structure 'a:b' was X, the call

```
assert(data(fred),X)
```

is equivalent to the call

```
assert(a:b:data(a:b:fred))
```

This extension adds considerable power to the database predicates and is extremely useful when constructing database applications and the like. From an implementation point of view, this is a simple operation. If a remote structure tag is given, the clause is physically moved to the new structure before it is asserted into the database. However, an important restriction is imposed on the type of clause which is acceptable to these predicates — all the predicates and functions contained within that clause must belong to the same structure. In more technical terms, the structure tags of every predicate and function within the clause must be equal. This means, for example, that a call like the following is illegal

```
assert(data(names:fred),X).
```

and the call fails displaying a warning message. This is because the predicate data/1 does not belong to the same structure as the function names:fred/0. The function 'names:fred' is classed as an *outer-structure reference* as the function refers to a structure other than the source structure (which is the home structure of the predicate in the head of the clause, i.e. data/1).

There are several reasons for imposing this restriction :

- The programmer is encouraged not to perform dirty tricks using this form of assert. Imposing this restriction hopefully results in a better programming style. The database should used sparingly anyway, as it reduces program clarity greatly.
- To illustrate one potential problem if these outer-structure references were legal, take the following example :

```
structure c =
    struct
        data(one).
    end.

structure a =
```

```
        struct
            structure b = c.
            ....
        end.
```

Consider the command line query :

```
| ?- structure(X,a), assert(b:data(two),X).
```

This would assert a second clause into structure 'c'. (The call `structure(X,a)` returns the structure tag of 'a'). However, the command line parses all input with respect to the top level structure. Since there is no structure 'b' defined at the top level, the command line cannot dereference the modular path 'b:data' and therefore cannot accept this call. This is a difficult problem to overcome.

- Another problem is a simple one, as follows. Consider the following program :

```
structure a =
    struct
        data(one).
    end.

structure b =
    struct
        more_data(one).
    end.
```

If the following call was attempted

```
| ?- structure(X,a), assert(data(b:two),X).
```

assert could not assert the clause into the database, as structure 'a' has no substructure 'b' which the data 'b:two'/0 could refer to. It is a trivial task to check this, but time consuming.

- There is yet another problem, exemplified by the following example :

```
structure one =
    struct
        structure two/sig1 =
            struct
```

```
                test.
            end.
        structure three = two/sig1.
        example :-
            structure(Tag,four),
            assert(test(three:data),Tag).
    end.

structure four =
    struct
        structure two =
            struct
                test.
            end.
        structure three =
            struct
                test.
            end.
    end.
```

(Note that 'one:two' and 'one:three' will share the same structure tag while 'four:two' and 'four:three' will have different structure tags.)

Consider a call to the predicate one:example/0.

Remembering that test/1 and three:data/0 are stored internally as tagged terms, the basic task of the call to assert is to 'decompile' the clause to determine the structures in which its constituent parts belong and 'recompile' the same clause in structure 'four'. When we check the structure tags of the components of the clause in the call to assert, we find that test/1 belongs to structure 'one' and three:data/0 belongs to structure 'one:three'. So far we have built the clause

```
one:test(one:three:data)
```

from its internal (tagged) representation. We now move the clause to structure 'four' which simply involves replacing 'one' by 'four' in the above clause and asserting

```
four:test(four:three:data)
```

into the database to complete the task.

However, the task is not that simple. Structure ‘one:two’ and structure ‘one:three’ share the same structure tag, so we could equally well select substructure ‘two’ instead of substructure ‘three’ in the decompilation stage and attempt to assert

```
four:test(four:two:data)
```

into the database instead. This would give a completely different result, with no clear method of distinguishing the best solution.

If the restriction suggested above were imposed, the system would not encounter such problems as every predicate and function in the clause would belong to the same structure.

This kind of problem cannot be resolved easily. Notice however, that if we did not change the semantics of a `structure X = Y/Sig` declaration (discussed in section 3.3) and retained the version in [SW92] where the new structure had a different tag to the old, this problem would not appear (though it would still appear if the program read `structure X = Y` without the signature constraint). This is an example of the intricacies involved in the design of the module environment.

These are the major reasons why the restriction described above is enforced.

It is important to clarify the use of assert/2 as this point. Take the following example :

```
structure a =
    struct
        data(one).
    end.
structure b =
    struct
        another_predicate.
    end.
```

The following is a series of calls to assert/2 along with a description of its action and why. Throughout, the variable `Tag` is the structure tag of structure ‘a’.

1. `assert(data(two),Tag)` Asserts the clause `a:data(a:one)` into the database.

2. `assert(b:data(b:two),Tag)` Asserts the clause `a:data(a:one)` into the database. The clause is legal, as all predicates and functions in it belong to the same structure, so the clause is simply moved to structure ‘a’ and asserted.

3. `assert(data(b:two),Tag)` Fails and displays an error because the clause passed to it contains a function (b:two/0) which does not belong to the same structure as the predicate (data/1).

The discussion relating to retract/1 and call/1 is identical and shows that the restriction imposed is there for good reason and not simply to make implementation easier.

6.4.3 Input/Output

The next major set of extra-logical predicates available in Prolog are the input and output predicates. I discuss these in this section. To simplify the discussion, I am only concentrating on the predicates write/1, writename/1 and read/1.

The system predicate write/1 is the general term printing routine for Prolog. The new version works in exactly the same way as before, but when write/1 prints out a function or predicate application, it finds the name of the structure to which the term belongs and prints out the pathname of that structure instead of simply the structure tag. Any term in the root structure or pervasive structure is printed without a module name or tag. For example :

```
| ?- write(test(1,2)).    % Test/2 is in the root structure.
test(1,2)
yes
| ?- write(assert(test)). % Assert/1 is pervasive.
assert(test)
yes
| ?- write(a:test(1,2)).  % Assuming structure 'a' has been declared.
a:test(1,2)
yes
```

When two names correspond to a single structure, as in the following program :

```
structure one =
    struct
        fun blah/0.
        ...
    end.
structure two = one.
```

write/1 will display the name which corresponds to the last structure declared. For example

```
| ?- write(one:blah).
two:blah
```

It makes no difference that the structure you typed may not be the same structure which is printed, simply because you have declared the structures to be equal anyway. The choice of name to print is not important and the last is always printed to be consistent.

Since each structure has a signature which identifies which predicates and functions are available for use by external structures, it might be reasonable to modify write/1 to display only terms that are not hidden by a signature. However, there a number of reasons for not doing this:

- The output of write/1 would depend on the structure in which the call was made. For example, if the function X were hidden inside structure Y by Y's signature, and an attempt were made to print X outside Y, then nothing (or some special token) would be printed as X is hidden. However, if an attempt were made to print X inside Y, X is not actually hidden, so can be printed. This means that write/1 would have to be extended in much the same way as the term manipulation predicates above so that the current structure tag could be passed to it.
- Every time a term was to be printed, its tag would have to be found, the signature for the structure corresponding to that tag would have to be generated, then checks made to see if the term was in fact hidden. This would reduce the speed of write/1 enormously.
- Finally, if write/1 were to print only visible terms, then the programmer would have to specify in advance which functions or predicates he/she will want to print at runtime. This is very awkward and the programmer would lose the flexibility offered by write/1 when debugging programs for example, making the system difficult to use.

Write/1, for the reasons outlined above, retains its ability to print any term which is passed to it, regardless of whether that term is hidden or not.

Another term printing predicate is writename/1. This predicate is actually a simple form of write/1, and can only print numbers, variables and atoms. If it is passed a compound term as an argument, it will only display the name of the term, not the arguments. I have re-written this predicate so that it makes no attempt to print module paths, and simply throws away any structure tag information passed in the argument. The reason for this is simple and came to light when constructing example modular programs for the system. Almost every program will output data to the screen in some form or other, and more importantly, many programs print friendly prompts and text messages to make the program more user friendly. In these cases it is undesirable to see text such as 'Please type your name' displayed as 'database:search_tree_type0:get_data:Please type your name' or '134__Please type your name' simply because the text occurs in a deeply nested structure. Writename/1 is therefore used to display non-compound terms with no module path information to provide a better user interface.

The last major input/output predicate discussed here is read/1. Read/1 shares many of the problems that the term manipulation predicates do, as well as having some of its own. For example, one of the tasks the read/1 routine has to do is check that any function applications typed are not involved in 'fun X = Y' declarations. If they are, the appropriate unification form has to be returned.

When read/1 accepts input in Modular Prolog, no checks are made to ensure that a function typed has been previously declared — any input is accepted. The only restrictions imposed are that the term must be syntactically correct and any modular paths within the term must exist with respect to the current structure. For example :

```
| ?- read(X).  % read/2 has not been previously declared.
test(1,2).

X = test(1,2)
yes
| ?- read(X).
a:b:test(1,2).
*** Error: Unknown structure a : b during read
```

One problem encountered with read/1 is the fact that its input has to be read in with respect to the current structure. If the user types an atom, say 'fred', the read/1 routine must tag that atom with the tag of the current structure. Similarly, if the user types 'one:fred', the read/1 routine must dereference the module path with respect to the current structure, to get the correct result. Read/1 is therefore one of the class of predicates that require an extra current structure tag argument to work correctly.

Operators are another problem for read/1. Should operators have effect only inside the structure in which the operator declaration was made? For example, consider the program :

```
structure one =
    struct
        fun infix/2.
        :- op(100,xfx,infix).  % Create infix operator.
        ...
    end.
```

If the operator declaration had a local effect (was only effective within the structure in which it was declared), the function infix/2 could only be read in infix form within the structure 'one'. Similarly, the function infix/2 would only be output (using write/1) in infix form within the structure 'one'. This leaves both read/1 and write/1 dependent on the structure in which they were called, which is awkward to manage. All operator declarations are therefore considered global — if any function is declared in a local structure as a prefix, postfix or infix operator, then that function will be printed out or read in using that operator declaration, regardless of where the input/output operation is performed. To clarify, consider the following :

```
| ?- consult(user).
structure a =
    struct
        fun test/2.
        :- op(500,xfx,test).
    end.
```

```
yes
| ?- read(X).
a a:test b.

X = a a:test b
yes
| ?- read(X).
a test b.

*** syntax error ***    % The function test/2 in the root
a <<here>> test b       % structure is not infix.
```

Operators are tricky to handle in this version of Modular Prolog and technical difficulties mean that operators cannot be used by files which are to be loaded by consult/1. This is awkward, but there are two reasons for this. Firstly, consult operates by loading a file into memory in one operation, then processing that file after the read is complete. This means that any calls to op/3 to define operators inside the file are not actually processed until after the loading is complete — to late to be of use in the file itself. This means, for example, that the following program will not be acceptable to the parser :

```
structure test =
    struct
        fun operator/2.
        :- op(500,xft,operator).
        test :-
            write(a operator b), nl.
    end.
```

If the parser executed any op/3 declarations as the file was read in, this problem would be solved. However, the parser itself cannot execute the op/3 declaration as it is read in, simply because the function operator/2 has not yet be created, and more importantly, the structure 'test' has not yet been created or assigned a structure tag. To get round this problem, the parser and the consult/1 code would have to be combined, which is tricky.

The second problem occurs for a similar reason. To simplify the processing of files, the entire file is read into the pervasive structure, then clauses are moved to the correct structures as they are processed. This means that any operators defined in the top level structure cannot be used by a consulted file as the file is not read into the top level structure, but the pervasive structure instead.

These problems mean that operators cannot be used by consult/1 (which is an undesirable restriction), but the work involved in rectifying the problem is considerable. The current architecture is clear and well-structured and any deviation from this would lose this clarity.

6.4.4 Other Bits

There are many other system predicates which have been modified to work in the Modular Prolog environment. I will not discuss any more in detail, as none of them introduces any new concepts, but I will list some of them to give a flavour of what the new Modular Prolog has to offer.

- assert(Clause,Structuretag)
  retract(Clause,Structuretag)
  retractall(Clause,Structuretag)

  These database manipulation predicates now accept a second argument which is a structure tag, and can operate on remote structures as well as their own.

- consult(File)
  consult(File,Options)
  consult(File,Options,Preds)

  Consult has three forms, with one, two or three arguments, which perform the same tasks as the old consult in Standard SB-Prolog. The first argument is the filename to use, the second is a list of options ('v' for verbose loading and 't' sets up trace points on all the predicates loaded during the consult), and the third argument returns a list of all the predicates loaded into the database.

- listing
  listing(Predicate)
  list_module(Structuretag)

  listing/0 now lists out all the top level structures, signatures and functors, and list_module/1 can be used to selectively list certain modules. listing/1 works as in Standard Prolog, and lists the clauses for the predicate given to it as an argument (in the form *name/arity*). Note that listing/0 attempts to list the contents of the database as it was read in, but the listed output is not detailed enough to be used as input for a future consult.

- pervasive(Function_or_predicate)
  pervasive_function(Function)
  pervasive_predicate(Predicate)

  Allows the user to obtain the functions and predicates in the pervasives signature.

- signature_name(Name)
  functor_name(Name)
  structure_name(Name)

Allows the user to obtain the names of the module constructs currently loaded which were declared at the top level.

- structure(Structuretag,Name)
  structure(Structuretag,Name,Withrespectto)
  current_structure(Structuretag)

  Allows the user to find the structure tag of any structure. All three are given an extra structure tag argument automatically by the system in order to work correctly. This means that the definition of current_structure/1 reduces to a simple `current_structure(X,X)`. Structure/2 is a special case of structure/3 where results are obtained with respect to the root structure.

  As an example of these predicates in operation, consider the following:

```
structure a =
    struct
        test(X) :-
            current_structure(X).
    end.
structure b =
    struct
        structure b = a.
        test1(X) :-
            structure(X,b).
        test2(X) :-
            current_structure(Y),
            structure(X,b,Y).
    end.
```

  Say the structure tag of structure 'a' is 45 and the tag of 'b' is 46. A call to a:test/1 returns 45. A call to b:test1/1 finds the structure tag of 'b' with respect to the root structure and so returns 46. A call to b:test2/1 finds the structure tag of 'b' with respect to structure 'b' and so returns 45.

  These predicates were proposed in [SW92] and are vital to allow other extra-logical predicates like assert/2 to operate in remote structures.

- dismantle_name(Term,Name,Structuretag)

  Here, if Atom is an atom then Name is bound to the name part of Atom (Name has no tag) and Tag is bound to the structure tag of Atom. Alternatively, if Name is bound (to an untagged atom) and Structuretag is bound, then Atom is bound to the atom whose name is Name and whose structure tag is Structuretag. This can be used to move atoms between structures. For example, the following call moves an atom X to a structure whose tag is New :

```
dismantle_name(X,Name,_),       % Remove the old tag.
dismantle_name(NewX,Name,New).  % Add the new tag.
```

In addition to these predicates, facilities for tracing are also available, as well as other less important features.

6.5 Performance and Portability

In this section is briefly discuss the performance of Modular Prolog in comparison to Standard Prolog in both memory consumption and execution time and then go on to discuss the portability of the system.

When testing the performance of Modular Prolog, I used the Eight Queens program given in appendix C. A version of this program was created which would run under Standard SB-Prolog, with no modules, as a comparison. This is also given in appendix C. To time the operation of consult/1, I loaded a large (non-modular) benchmark program using both versions of Prolog. Figure 6.1 shows the results of my experiments. Execution times were calculated on ten consecutive runs of the depth first search solution to the eight queens problem. Memory consumption was calculated as accurately as possible but the SB-Prolog dynamic loader makes the task very complex. The memory consumption figure for Standard Prolog is accurate, but the Modular Prolog figure is only an estimate.

	Memory Consumption (bytes)	Execution Time (seconds)	Consult Time (seconds)
Standard Prolog	570	43	91
Modular Prolog	2000	43	189

Figure 6.1: Performance of Modular and Standard Prolog

As is clearly shown from the results, execution time is not hindered in any way by the modules system. This result is not surprising as consult/1 simply translates the Modular Prolog program into Standard Prolog and so imposes little or no overhead in runtime performance. However, if a program which made extensive use of the extra-logical predicates was tried, then a small reduction in performance may be observed.

Memory performance is poor in the modules system. Again, this is no surprise. As well as storing the program code, the modules system has to store the module environment data. Another problem occurs because the new modular consult loads the file initially into the pervasive structure and then moves the program to the correct structures during processing (this was mentioned in section 6.4.3). However, this moving operation is actually a copy and so there are, in most cases, two copies of each predicate and function name in the symbol table, one untagged (pervasive) version and one tagged (local) one. This is an important memory problem.

However, memory usage aside, Modular Prolog is not hampered in any way by the modules environment and executes as fast as Standard SB-Prolog, only slowed by the dramatic increase in processing necessary to perform a consult (as can be seen in the table).

There are several aspects of the modules system that must be considered when discussing the portability of the system to other versions of Prolog.

The representation of predicate and function symbols in the modules environment was chosen partly because it is easily portable to any version of Prolog. The modules environment itself is simply a series of Prolog clauses corresponding to each modular construct declared, and so again this can be directly implemented in any other version of Prolog.

However, there are parts of the module system which cannot be so easily ported. The most obvious is the contents of the pervasive signature and the treatment of the pervasive structure. Different versions of Prolog have different sets of system predicates available to the user and so the contents of the pervasive signature and the nature of the pervasive structure will change.

The interaction of the extra-logical predicates with the modules system is the only other major consideration. There are many extra-logical predicates which cannot be used with the modules system at all, or have special cases which do not appear in Standard Prolog. These predicates often require rewriting to operate correctly. In order to convert all the extra-logical predicates available in a version of Standard Prolog to their modules system equivalents, they have to be considered one-by-one and changed appropriately, retaining compatibility with other parts of the original system that may use them. This is what I did with SB-Prolog and is probably the most time-consuming part of the process of implementation.

The system is, on the whole, written in a way that makes it easily portable and could, with a little work, be implemented on any other Prolog system.

Chapter 7

Using Modular Prolog

Modular Prolog at first glance has the same look as Standard Prolog, and provides much the same facilities. However, due to the changes made to the extra-logical predicates of SB-Prolog, many Standard Prolog programs cannot be executed without modification. Having said this, many existing programs will be compatible with the modules system and can still be used.

There are other interesting points to note about the operation of Modular Prolog. Firstly, module paths specified by colons are only acceptable to the system predicates read/1 and consult/1 and nowhere else. Since these predicates convert any module paths into structure tags and build the correct internal names for predicates and functions, module paths do not actually exist at all at runtime, and can be regarded at a consult time concept which is replaced by structure tags for execution. Any clauses containing colons (:) which cannot be processed as valid module paths are regarded as illegal and are disallowed.

Secondly, the command line in Prolog normally displays a list of all variables in the call along with their bindings if the call returned successfully. The new Prolog displays results in a similar way, but any items in the result which are hidden inside substructures (because they are hidden by a signature), are displayed as '...' instead. This emphasizes the use of abstract data types in the new Prolog. It should be noted however, that since the signature of a structure is built by consult/1, and never changes, any undeclared functions created by calls to the extra-logical predicates are also hidden. This should be kept in mind. As an example of the command line in operation, here we make a call to the structure 'stack2' defined in chapter 2 of this report. The result shows the hiding operation in action :

```
| ?- stack2:newstack(X),
     stack2:push(X,1,Y),
     stack2:pop(Y,Z,Popped),
     stack2:isempty(Z).

X = ...
```

```
Y = ...
Z = ...
Popped = 1
yes
```

Lastly, the user has to keep in mind structure tags when performing unification. A common mistake would be to accept input from the keyboard in one structure and compare the input with data listed in the database inside another. This of course will never succeed as the data in these structures are mutually disjoint (as they are represented by different structure tags). The best way to avoid this is to make sure all data is read in inside the structure that contains the data.

This is highlighted by the following example :

```
structure data =
    struct
        fun fred/0.
        acceptable(fred).
        read_in(X) :-
            read(X).
    end.
structure test =
    struct
        inherit data.
        test1 :-
            read(X),
            data:acceptable(X).
        test2 :-
            data:read_in(X),
            data:acceptable(X).
    end.
```

If 'fred' is typed in response to the input needed for test:test1/0 and test:test2/0, we find that test:test1/0 fails but test:test2/0 succeeds.

I expect that mistakes like this will be frequent.

This confusion may be less easy to detect when comparing terms in the root structure with terms in the pervasives structure. Both are displayed by write/1 in the same way (no module name or tag is given), and so the user may miss the fact that they belong to different structures. The user should keep in mind which constants are regarded as pervasive, which actually includes functions such as a/0 and memory/0 which are not obviously pervasive.[1]

[1] a/0 is actually a compiler flag and memory/0 is an option for obtaining system data via statistics/2.

There are many eccentricities introduced when extra-logical predicates interact with the modules environment. These are too numerous to mention, and are generally obvious when considered closely. The following example illustrates one of these :

```
structure b =
    struct
        ...
    end.
structure a =
    struct
        structure b = a.
        ...
    end.
structure a =
    struct
        ...
    end.
```

If you attempted to print an atom that originated in the old structure 'a:b', you get 'a:b:*atom*' printed. Since structure 'a' has been redefined, and contains no substructure 'b', there seems to be an inconsistency. This result can be spotted when using other extra-logical predicates too, and is because the module environment is a continually expanding data structure, which never contracts — copies exist of all previously defined structures, signatures and functors, even those which can no longer be used. It is only if a stale structure tag is presented to the system that such apparent inconsistencies arise.

Chapter 8

Future

There are many areas in which this project could be extended in the future, so I have devoted this chapter to listing some of them. Although the modifications and extensions suggested here vary immensely in complexity, they are not presented in any particular order.

The first suggestion is a simple measure to allow better structure tag validation. Just now, structure tags are simply integers and so it is easy for a rogue program to generate a random tag and use it to assert, retract or call predicates in a random remote structure. A more secure system would set up structure tags as a new distinct datatype and disallow operations such as the arithmetic ones to be performed on them. This is a very simple runtime safety measure.

One optimization that could be introduced is briefly described in Sannella and Wallen's paper [SW92]. Each time a functor is applied, the code within its body is duplicated in the database. This seems to be a waste, and techniques could be introduced to overcome this problem. However, as Sannella and Wallen point out, the number of functor applications is likely to be small and alternative techniques are often undesirable (for example, using explicit `calls`). David MacQueen describes an algorithm for Standard ML modules in his work which allows all structures produced by functor application on the same functor to share one copy of the functor code (see [Mac88]). A similar idea could be done in Modular Prolog.

The data structures used for storing the module environment are currently a simple implementation of those used by the semantic equations in [SW92]. This is not the most efficient representation (in time and space) and an improvement could be found which is more flexible and less memory and processor intensive.

The current version of listing/0 attempts to produce an output which looks like the file as it was read in. However, the output is not in a format that is acceptable to consult/1 for reading. A better version of listing/0 would output a listing in a form suitable for consult/1 so that the program could be re-read at a later date. This would require more information to be stored, and more processing to be done on the module environment data. Two of the major faults in the current version of listing/0 are as follows (there are many other minor ones as well).

Firstly, listing/0 only displays top level structures and no substructures. If it were to display substructures in a naive implementation, large chunks of code would be repeated. For example :

```
structure a =
    struct
        ....
    end.
structure b = a.
```

would print the code in '....' twice for structures a and b which means that if a listing of the program were read in, structure 'a' and 'b' will no longer be the same structure, but distinct ones. Secondly, the order in which the output is displayed bares no relation to the dependency order which exists in the original modular program (each construct has to be declared before it is used). This would require the use of planning algorithms to construct display orders. There are many other problems which make implementation of such a system very difficult, but there are too many to discuss here.

Another suggestion made by Sannella and Wallen in their paper was the addition of a type checking system for Prolog. There are a number of suggestions for type checking systems, such as the one by Mycroft and O'Keefe [MO83] and the one by Horiuchi and Kanamori [HK87] These could be incorporated into the modular system without much trouble. However, there are decisions to be made about how the type checking system responds to Prolog's extra-logical predicates. Should the system be a consult time checker only or should individually asserted clauses be type-checked as well? At a basic level however, installation of a simple type checking system is a relatively straightforward task.

SB-Prolog is an interpreter and compiler system, and so far I have only discussed the modules system interacting with interpreted Prolog programs (I have only used the compiler for system development purposes). Extensions could be made to the compiler to enable it to accept modular programs allowing programs to gain performance by compilation. However, we need to decide how to handle separately compiled sections of modular code as structure tags have to be unique with respect to the runtime environment and cannot be generated at compile time. This means a full compilation from program text to Warren Abstract Machine instructions may not be possible. Also, module environments need to be included in the compiled files, so later programs can use the constructs that they define. These are the kind of problems this extension would have to overcome.

When new structures are allowed to replace old ones, we have to consider what happens to structures that can no longer be accessed and are redundant (this happens regularly if the same files are continually consulted). In Standard ML, the garbage collector removes these structures from memory and reclaims their space. A similar garbage collector could be designed for Modular Prolog which tidies up the program areas of memory when necessary[1]. Space savings from carrying out these garbage collections can be considerable.

The modules system itself resembles an object-oriented programming environment if we consider a structure as an object and the clauses within it as the operations that can be performed on that object. If we had facilities to dynamically create and destroy structures, we could develop a Prolog-style object-oriented programming environment.

Dynamic binding of modular paths is another future possibility. By dynamic binding of modular names, I mean the ability to form pieces of code like the following :

```
structure one =
  struct
     structure a = alpha.
     structure b = beta.
     test(Data) :-
          select_a_or_b(X),
          X:get_data(Data).
     ...
   end.
```

where the structures in path names need only be bound at run time, not statically at consult time. This can add considerable flexibility to the system.

Note that the addition of dynamic modular paths in the head of a clause is of little use and can lead to undefined results. For example,

```
testpredicate(X:test) :-
        write(X),nl,
        X:predicate(X:test).
```

could be used to get the home structure of the argument, print out that name and finally call predicate/1 inside that structure. For this to operate correctly, the head of the clause would have to bind X to the textual name of the structure instead of a simple tag. However, there can be many possible values of X if declarations like

```
structure one = two.
```

are used. Similar problems are encountered with calls to =/2. For example, the call

```
X = Y:test
```

suffers from exactly the same problems as the predicate testpredicate/1 above where there could be many equally valid results. I believe that the problems encountered in the implementation of

[1]The current version of SB-Prolog has no garbage collector at all for the program space and the garbage collector for the heap is buggy and out of action!

allowing unbound paths in the head of a clause outweighs the rather fewer advantages. I only suggest allowing dynamic module paths in the body of clauses.

However, implementation of this extension is very difficult and involves modification of the Warren Abstract Machine. For example, consider the simple query

```
?- X = test, process(X:fred).
```

The value of X is not known until the call to process/1 is about to be made. However, in order for the call to continue, 'test:fred' has to be dereferenced before the call is made. There is no interaction here with anything at the Prolog level (if the call was made via call/1, we could do some pre-processing), so all processing has to be done at the Warren Abstract Machine level. This area is beyond the scope of this project, so is left only as an introduction to the complexities of dynamic binding in the modules environment.

This concludes an overview of some future ideas for the modules system and highlights some of the problems and shortfalls of the system.

Chapter 9

Related Work

There are numerous other modules systems that have been proposed for Prolog, and many have been implemented in systems available today. However, most commercial systems have very simplistic modules systems. For example, in Quintus Prolog [Qui], a widely used system, modules are flat and do not address function modularity. When a module is defined it is specified by a name and a public predicate list (a list of the predicates defined inside the module which can be accessed from other modules). The modules interact by specifying which other modules they are to import predicates from. If a module M imports a module N, then module M can call any predicate P which is given in module N's public predicate list, without the need to qualify[1] predicates. This does mean that the list of predicates defined in a module has to be distinct from the public list of any module it imports. However, Quintus allows dirty programming and by explicitly qualifying predicates one can override the importing rules. One interesting aspect of Quintus Prolog which could be considered for future versions of my modules system, is that the database manipulation predicates can only operate on predicates within the current module unless a remote predicate has been defined as a 'dynamic' predicate in its home module.

LPA Prolog [Log] has a similar system to Quintus, but allows hierarchical modules. Modules are defined by their name, their export list, and an additional import list (predicates only). Both systems are typical of many of the modules systems in commercial Prolog systems. Even MProlog [Log85], whose name is an acronym for Modular PROgramming in LOGic, is a flat modules system, with commands to import or export individual predicates or entire modules only.

Dietrich [Die89] proposed a preprocessor based modules system for Prolog which could easily be placed on top of any existing Prolog system. The modules system itself is in much the same style to Quintus Prolog (modules are flat, and are explicitly imported if needed), but includes extensions for function modularity so the implementation of abstract data types is

[1] Given a predicate *predicate*, its qualified form is *module*:*predicate*, where *module* is the 'home' module for that predicate.

possible. Dietrich addresses the issue of extra-logical predicates in his paper but claims that a pre-processor based system is enough to resolve any problems that may occur at run time. His argument is flawed however, failing to take into account calls like :

```
:- read(X), assert(X).
```

Dietrich also suggests that instead of creating one system module implicitly imported to all modules, there could be several system modules which have to be explicitly imported if needed. Example modules could be 'arithmetic' or 'lists' for handling specific tasks. This idea is used in the Gödel language by Hill and Lloyd [HL91], regarded as the declarative successor to Prolog. But again Gödel has a very simplistic module system. It does provide however, modularisation facilities for both predicates and functions, allowing abstract data types to be created.

Chen [Che87] bases his module system on second-order logic because he claims that since Prolog itself is based on first-order logic, it is natural to base a modules system on second-order logic. Take the following simplistic example of a module :

```
mod_test(Printout,Comp) = {
    Printout(A,B) :-
        Comp(A,B),
        write(correct), nl.
    Printout(A,B) :-
        write(incorrect), nl.
    }
```

'Printout' and 'Comp' can be regarded as predicate variables for the module 'mod_test'. When the module is to be used, you 'call' it with predicate arguments (actual predicates or predicate variables), and a module instance is created for the duration of that call. For example, consider the call :

```
:- mod_test(Printout,<=), Printout(3,5).
correct
```

This builds a module which defines a predicate Printout, parameterised using the predicate <=/2. Note that Printout is actually a variable, and is assigned a value at run time. A call is made to the predicate bound to Printout and after the call is complete, the module created for the call is destroyed when the binding of Printout is lost.

Complex compound modules can be built in this system and this leads to a powerful, hierarchical modules system. However, the author does not address the issues of extra-logical predicates, claiming "no higher-order logic is sufficient for characterizing the semantics of call/1". This does reduce the usefulness of the system.

I complete this overview by considering the modules system proposed in a draft document by the International Organisation for Standardisation [Pro]. They propose a completely flat modules system, with no concept of function modularisation. The programmer can completely bypass the modules system by explicitly qualifying predicate names, giving a rather 'dirty' system. This, of course, bears remarkable similarity to the Quintus Prolog and other major Prolog systems available today.

It is interesting to note that out of the modules system discussed above, only Chen [Che87] introduces the concept of parameterised modules. In my system, functors are parameterised modules and are used frequently to build up larger programs from smaller components. Chen's module system allows parameterised modules which accept predicate arguments only but has no concept of modules parameterised by other modules.

Sannella and Wallen discuss other modules systems in their paper [SW92], including work by Miller [Mil89] and Goguen and Meseguer [GM84], bit I will not repeat their discussion here.

Chapter 10

Conclusion

This report has described the implementation and refinement of the modules system for Prolog proposed by Sannella and Wallen [SW92]. They base their work on the modularisation facilities of Standard ML and describe an extended language which allows construction of Prolog programs by instantiation of parameterised components and provides facilities for data abstraction.

In particular, I have extended this work to discuss the interaction of the module environment with Prolog's extra-logical facilities, an area often neglected when other module systems are considered.

The Modular Prolog system developed throughout this project sits on top of an existing Prolog system (SB-Prolog) and my work shows that this modules system does not require alteration of the basic Prolog interpreter (the Warren Abstract Machine). The only changes to the code of SB-Prolog that were in fact required were to the Prolog level routines which handle the extra-logical facilities of Prolog.

However, a word must be said about the suitability of Prolog for such a modules system. In a language like Standard ML, where there are no facilities to dynamically alter programs or to build new datatypes, then the modules system is suitable. However, the modules system itself implies a reasonably static environment, which is not the nature of Prolog at all. In Prolog, the programmer requires the ability to make full use of the database and the ability to create terms in any form, at any time. This does not interact nicely with the modules system.

The module system also introduces 'fun X = Y' declarations to enable functions with different names to unify. Since unification is an integral part of Prolog, surely additions such as this are altering the underlying language? The programming methodology used with modules is different to that used in Standard Prolog. Standard Prolog programmers are used to an environment without declarations and without a strong distinction between predicates and functions — flexibility is often the key word for Prolog. The modules system introduces both these constraints, so is a radically new approach for Prolog programmers. The need to include a relaxed approach to declaration before use, allowing undeclared predicates and functions to be accepted, shows to some extent how the modules system is too restrictive for Prolog.

Having said this, Prolog is in need of a good modules system. This modules system is powerful and flexible, and provides much needed facilities such as data abstraction. But is it flexible enough for the Prolog programming community?

In sum, I have implemented a powerful hierarchical modules system for Prolog which adds module constructs and data abstraction facilities needed to allow Prolog to become a more convenient language for large projects.

Bibliography

[Bra86] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

[Che87] Weidong Chen. A Theory of Modules based on Second-Order Logic. In *Proceedings of Symposium on Logic Programming*, pages 24–34, 1987.

[Deb88] Saumya K Debray. *SB–Prolog Manual.* Department of Computer Science, University of Arizona, 1988.

[Die89] R Dietrich. A Preprocessor Based Module System for Prolog. TAPSOFT, 1989.

[GM84] J Goguen and J Meseguer. Equality, types, modules and generics for logic programming. In *Proceedings on Symposium on Logic Programming*, pages 115–127, 1984.

[GM89] G Gazdar and C Mellish. *Natural Language Processing in Prolog : An Introduction to Computational Linguistics.* Addison-Wesley, 1989.

[Har89] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh, January 1989.

[HK87] Kenji Horiuchi and Tadashi Kanamori. Polymorphic Type Inference on Prolog by Abstract Interpretation. Technical Report TR-263, ICOT Tokyo, 1987.

[HL91] P M Hill and J W Lloyd. The Gödel Report. Technical report, Department of Computer Science, University of Bristol, September 1991.

[Llo87] J W Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[Log] Logic Programming Associates Ltd. *LPA Prolog Professional Manual (Version 1.5).*

[Log85] Logicware. *MPROLOG, Release 2.1*, September 1985.

[Mac88] David MacQueen. An Implementation of Standard ML Modules. Technical report, AT&T Bell Labs, March 1988.

[Mil89] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, Volume 6:79–108, 1989.

[MO83] Alan Mycroft and Richard O'Keefe. A Polymorphic Type System for Prolog. Technical Report No 211, Department of Artificial Intelligence, University of Edinburgh, 1983.

[Pro] Prolog - Part 2, Modules - Working Draft 1.0. International Organisation for Standardisation.

[Pro92] Prolog - Part 1, General Core - Committee Draft 1.0. International Organisation for Standardisation, March 1992.

[Qui] Quintus. *Quintus Prolog Manual.*

[RK92] M G Read and E A Kazmierczak. Formal Program Development in Modular Prolog : A Case Study. Technical Report ECS-LFCS-92-195, University of Edinburgh, January 1992.

[SW92] D T Sannella and L A Wallen. A Calculus for the Construction of Modular Prolog Programs. In *The Journal of Logic Programming*, volume 12, nos 1 and 2, pages 147–178. January 1992.

[War83] D H D Warren. An Abstract Prolog Instruction Set. Technical Report 309, AI Center, SRI International, October 1983.

Appendix A

Changes to the Semantics

The following gives a list of the changes made to the semantic equations given in [SW92]. The semantics used by this system is basically much the same as that proposed, with only minor differences.

First, a list of the semantic changes to compensate for the changes in core syntax is given. There are trivial changes that have been made to the semantics, but these are not included here, only the less trivial cases are given.

- In order to handle the new style of separate `structure` and `sharing` specifications, replace the old $Spec$[[structure ... sharing ...]] with :

  $Spec$[[sharing $patheq_1$ and ... and $patheq_n$]] $sig\ \psi =$

  $$identify(Patheq[[patheq_1]]sig \cup \ldots \cup Patheq[[patheq_n]]sig, sig)$$

- To handle the new `pred Name/Arity` declarations, add the new equation :

  Dec[[pred $atid/nat$.]] $< substrs, preds, funs > \rho\ \psi\ \pi =$

  let tag be an unused internal predicate name in

  $< substrs, preds \cup \{< atid, nat > \mapsto tag\}, funs >, \rho, \psi, \pi, \emptyset >$

  $error$ if $< atid, nat > \in dom(preds)$

The following is the list of the semantic equation changes that do not fall into the category of syntax changes.

- A subtle change to fit : $structure$ x $signature \longrightarrow structure$. This function no longer creates a unique tag for the structure produced as a result of the fitting operation, as we want structures defined by statements like `structure one = two/signature1` to share the same structure tag.

  $fit(< tag, < substrs, preds, funs >>, < substrs', preds', funs' >) =$

$< tag, < substrs\lceil dom(substrs'), preds\lceil dom(preds'), funs\lceil dom(funs') >>$

$error$ ... as before ...

- In order to ensure that we can redefine a structure more than once, and use only the most recently declared, we change the following :

  In $addsubstrs$: $(atid$ x $structure) - set$ x $signature \rightarrow signature$, change last line of function which was :

  $$< substrs \cup substrs', preds \cup preds', funs \cup funs' >$$

  to

  $$< substrs[substrs'], preds[preds'], fun[funs'] >$$

  so that new structure declarations replace the old ones.

  In the definition of $Strb[[atid = strexpr]]$ the error case

  $$error \text{ if } atid \in dom(\rho)$$

  should be removed to allow structures to be redefined.

  And finally, in the definition of Dec[[structure ...]] the section that reads :

  $$\rho \cup \{ \ atid_1 \mapsto sig_1, \ \ldots \}$$

  should be replaced by

  $$\rho \ \ [atid_1 \mapsto sig_1, \ \ldots]$$

  so that when the structure environment ρ is updated, new versions of a structure replace old ones.

  Similar changes are necessary for the declaration of signatures and functors, but these are not listed here.

- In order to ensure that predicate and functions are unique, the error lines in $Spec[[fun \ldots]]$, $Spec[[pred \ldots]]$, $Dec[[atid(\ldots) \text{ :- } predappl_1, \ldots]]$, $Dec[[fun \ldots]]$, $Dec[[pred \ldots]]$ and $Dec[[atid(\ldots)]]$ need to be extended to ensure that any constant defined is not already a function or a predicate. The additional error case is :

  $error$ if $< atid, nat > \in dom(preds)$ or if $< atid, nat > \in dom(funs)$

This concludes the list of changes to the semantic equations that I propose.

Appendix B

Notes on Prolog Terminology

It is interesting to briefly discuss the terminology used in this report. In the general Prolog literature, the terminology used for describing Prolog depends very much on the author and the intended audience and in fact much of the terminology used is completely contradictory.

For example, in [SW92] the following terminology is used :

Predicate Constant = predicate symbol and arity

Function Constant = function symbol and arity

Term = function constant and sequence of arguments (terms)

Atom = predicate constant and sequence of arguments (terms)

This is consistent with the terminology used by Lloyd [Llo87], which is standard in the logic community.

However, taking the Prolog programmer's view of Prolog, we see very different terminology, briefly outlined below :

Functor = symbol and arity

Compound term = functor and arguments (not including atoms)

Atom = zero arity functor

Structure = compound term

Atomic = an atom or a number but not a variable

Term = anything (a variable, an atom, a number, a compound term or a list)

This is taken from the SB-Prolog manual [Deb88], which is consistent with Prolog texts such as [Bra86]. Notice that the main inconsistencies appear in the definitions of *atom* and *term* and that the latter terminology does not address the distinction between functions and predicates, a notion emphasized by the former. These inconsistencies were a major problem I encountered when introducing the terminology I use throughout this document. No standard set of terminology exists for Prolog.

A draft copy of the International Standard for Prolog [Pro92] introduces a knot of terminology, spanning several pages. To give an example, the document uses terms such as *predicate*, *predicate indicator*, *predicate name*, *predication* and *procedure*, all referring to predicates and their specification. This seems somewhat unnecessary and overly complex.

I have tried to keep a centre line in this argument, and so have introduced the terminology given in the introduction to this report, which I hope is self explanatory.

Appendix C

Example Programs

This appendix gives the code for several Modular Prolog programs and is included to back up the introduction to the modules system given in chapter 2 and ideas developed in later chapters.

C.1 Eight Queens Problem I

The first program is adapted from a program in “Prolog Programming for Artificial Intelligence” by I. Bratko [Bra86] and highlights how the modules system can be used for simple AI applications.

The program solves the Eight Queens problem by depth-first or breadth-first search. The basic problem is to place eight queens on an eight by eight chess board such that no queen can attack any other.

```
signature searchsig =
    sig
        pred solve/2.   % To start search.
    end.

signature problemsig =
    sig
        pred s/2 and    % Calculate successor states.
             goal/1.    % Defines a valid goal state.
    end.

functor dfs(p/problemsig)/searchsig =
    struct
        structure x = p.
        solve(Node,Solution) :-
```

```
            df([],Node,Solution).
        df(Path,Node,[Node|Path]) :-
            x:goal(Node).
        df(Path,Node,Sol) :-
            x:s(Node,Node1),
            not member(Node1,Path), % No Cycles!
            df([Node|Path],Node1,Sol).
    end.

functor bfs(p/problemsig)/searchsig =
    struct
        structure x = p.
        solve(Node,Solution) :-
            bf([[Node]],Solution).
        bf([[Node|Path]|_],[Node|Path]) :-
            x:goal(Node).
        bf([[N|Path]|Paths],Solution) :-
            bagof([M,N|Path],
                  (x:s(N,M),not member(M,[N|Path])),
                  Newpaths), % Newpaths = acyclic extensions of [N|Path]
            append(Paths,Newpaths,Paths1), !,
            bf(Paths1,Solution);
            bf(Paths,Solution).   % Case that N has no successors.
    end.

structure eightqueens/problemsig =
    struct
        goal([_,_,_,_,_,_,_,_]).
        s(Queens,[Queen|Queens]) :-
            member(Queen,[1,2,3,4,5,6,7,8]),
            not member(Queen, Queens),
            safe([Queen|Queens]).
        safe([]).
        safe([Queen|Others]) :-
            safe(Others),
            noattack(Queen,Others,1).
        noattack(_,[],_).
        noattack(Y,[Y1|Ylist],Xdist) :-
```

```
            Y1 - Y =\= Xdist,
            Y - Y1 =\= Xdist,
            Dist1 is Xdist + 1,
            noattack(Y, Ylist, Dist1).
    end.

structure eightbfs = bfs(eightqueens).
structure eightdfs = dfs(eightqueens).
```

To solve the problem by depth first search use :

```
eightdfs:solve([],Solution)
```

or by breadth first search use :

```
eightbfs:solve([],Solution)
```

The solution is a list of the nodes which form the path taken to solve the problem. The node at the head of the list is actually the final solution, and has the form :

$$[position_1, position_2, \ldots, position_8]$$

which means place one queen at (1,$position_1$), the next at (2,$position_2$), etc.

C.2 Eight Queens Problem II

This program is not a Modular Prolog program but runs under Standard Prolog. It was used to compare the memory and execution performance of Modular and Standard Prolog.

```
solvedfs(Node,Solution) :-
    df([],Node,Solution).

df(Path,Node,[Node|Path]) :-
    goal(Node).
df(Path,Node,Sol) :-
    s(Node,Node1),
    not member(Node1,Path), % No Cycles!
    df([Node|Path],Node1,Sol).

solvedfs(Node,Solution) :-
    bf([[Node]],Solution).
```

```
bf([[Node|Path]|_],[Node|Path]) :-
    goal(Node).
bf([[N|Path]|Paths],Solution) :-
    bagof([M,N|Path],
          (s(N,M),not member(M,[N|Path])),
          Newpaths), % Newpaths = acyclic extensions of [N|Path]
    append(Paths,Newpaths,Paths1), !,
    bf(Paths1,Solution);
    bf(Paths,Solution).   % Case that N has no successors.

goal([_,_,_,_,_,_,_,_]).

s(Queens,[Queen|Queens]) :-
    member(Queen,[1,2,3,4,5,6,7,8]),
    not member(Queen, Queens),
    safe([Queen|Queens]).

safe([]).
safe([Queen|Others]) :-
    safe(Others),
    noattack(Queen,Others,1).

noattack(_,[],_).
noattack(Y,[Y1|Ylist],Xdist) :-
    Y1 - Y =\= Xdist,
    Y - Y1 =\= Xdist,
    Dist1 is Xdist + 1,
    noattack(Y, Ylist, Dist1).
```

C.3 Database Management

The following is an example of a database management program. Although it is a rather small and very artificial example, it shows many of the facilities available under the modules system. However, the program is not a good example of good modular programming style and methods used here are not ideal — it has been written simply to show as many modular and extra-logical facilities in action as is possible in one program. The other programs given in this appendix give a better indication of good programming technique.

Some of the facilities shown are :

- Asserting into remote structures.
- Calling predicates in remote structures.
- The use of term manipulation predicates.
- Input and output predicates.
- Moving atoms to remote structures (using dismantle_name/3).

```
signature dataopsig =
    sig
        fun record/2.
        pred add_record/0 and get_record/1.
    end.

signature searchsig =
    sig
        pred ismatch/3.
    end.

structure database =
    struct
        pred record/2.
    end.

structure dataoperations/dataopsig =
    struct
        fun record/2.
        add_record :-
            writename('Type name'),nl,
            get_atom(A),
            writename('Type search keys (terminate with end).'), nl,
            get_keys(List),
            B =.. [data|List],
            structure(Tag,database),
            assert(record(A,B),Tag).
        get_record(record(X,Y)) :-
            var(X), var(Y),
            structure(Tag,database),
            call(record(X,Y),Tag).
```

```
                % Note the use of var/1 to guarantee that no
                % outer structure references occur during call/2.
        get_atom(X) :-
            repeat,
            writename('> '),
            read(X),
            (atom(X) -> true ;
                (writename('Data must be an atom, please re-type'),
                 nl,fail)).
        get_keys(List) :-
            get_atom(X),
            (X == end -> List = [] ;
                (get_keys(Rest),
                 List = [X|Rest])).
    end.

structure search1/searchsig =
    struct
        fun item = dataoperations:record.
        ismatch(X,item(A,B),A) :-
            B =.. [_|Rest],    % _ would actually be database:data/0
            member(X,Rest).
    end.

structure search2/searchsig =
    struct
        fun item = dataoperations:record.
        ismatch(X,item(A,B),A) :-
            match_args(1,B,X).
        match_args(Arg,B,X) :-
            arg(Arg,B,Item),
            (Item = X -> true ;
                        (Narg is Arg + 1,
                         match_args(Narg,B,X))).
    end.

functor time(x/searchsig) =
    struct
```

```
        structure search = x.
        inherit dataoperations.
        timesearch(SF) :-
            cputime(X),
            get_results(SF),
            cputime(Y),
            Diff is Y - X,
            writename('Time taken is '),
            writename(Diff), nl.
        get_results(SF) :-
            dataoperations:get_record(X),
            search:ismatch(SF,X,A),
            tab(8),writename('Found '),
            writename(A),
            tab(5),
            write([A]),
            nl,fail.
        get_results(_).
    end.

functor menu(x/dataopsig,y/searchsig,z/searchsig) =
    struct
        structure dataops = x.
        structure search1 = time(y).
        structure search2 = time(z).
        menu :-
            nl,
            writename('Select option required'), nl,
            writename(' 1 - Add new record'), nl,
            writename(' 2 - Timed search 1'), nl,
            writename(' 3 - Timed search 2'), nl,
            writename(' 4 - Quit'), nl,
            writename('{Terminate all input with a period (.)}'), nl,
            repeat,
            writename('> '),
            read(X),
            valid_choice(X),
            menu.
```

```
        valid_choice(1) :-
            dataops:add_record.
        valid_choice(2) :-
            get_search_key(Key),
            search1:timesearch(Key).
        valid_choice(3) :-
            get_search_key(Key),
            search2:timesearch(Key).
        valid_choice(4) :-
            abort.
        valid_choice(_) :-
            writename('Invalid choice - reselect'), nl, fail.
        get_search_key(Key) :-
            repeat,
            writename('Type search key > '),
            read(X),
            (atom(X) -> (structure(Tag,database),
                         dismantle_name(X,Name,_),
                         dismantle_name(Key,Name,Tag)) ;
                        (writename('Search key must be an atom'),
                         nl,fail)).
                  % Note here that we have to move the atom typed from
                  % the current structure to the structure 'database'
                  % as the original data was moved to 'database' and
                  % the tags must match for the atoms to be equal.
    end.

structure program = menu(dataoperations,search1,search2).
```

The system is started using `program:menu`.

C.4 Active Chart Parser

The following is an active chart parser which can handle a small subset of English. The program is adapted from the active chart parser given in "Natural Language Processing in Prolog : An Introduction to Computational Linguistics" by G. Gazdar and C. Mellish (see [GM89]).

```
signature agenda_sig =
    sig
```

```
        pred add_to_agenda/4 and remove_from_agenda/4 and
             erase_agenda/0 and agenda_data/4.
    end.

structure agenda/agenda_sig =
    struct
        pred agenda_data/4.
        add_to_agenda(V0,V1,C,Needed) :-
            agenda_data(V0,V1,C,Needed) ->
                true ;
                asserta(agenda_data(V0,V1,C,Needed)).
        remove_from_agenda(V0,V1,C,Needed) :-
            retract(agenda_data(V0,V1,C,Needed)).
        erase_agenda :-
            retractall(agenda_data(_,_,_,_)).
    end.

signature chart_sig =
    sig
        pred add_chart_edge/4 and erase_chart/0 and chart_edge/4.
    end.

structure chart/chart_sig =
    struct
        pred chart_edge/4.
        add_chart_edge(V0,V1,C,Needed) :-
            chart_edge(V0,V1,C,Needed) ->
                true ;
                asserta(chart_edge(V0,V1,C,Needed)).
        erase_chart :-
            retractall(chart_edge(_,_,_,_)).
    end.

signature grammar_sig =
    sig
        pred initial/1 and rule/2 and lex/2 and tag/1.
    end.
```

```
structure grammar1/grammar_sig =
    struct
        tag(X) :-
            current_structure(X).
        initial(s).
        rule(s,[np,vp]).
        rule(np,[det,nb]).
        rule(nb,[n]).
        rule(nb,[n,rel]).
        rule(rel,[wh,vp]).
        rule(vp,[iv]).
        rule(vp,[tv,np]).
        rule(vp,[dv,np,pp]).
        rule(vp,[sv,s]).
        rule(pp,[p,np]).
        lex(np,[kim]).
        lex(np,[sandy]).
        lex(np,[lee]).
        lex(np,[bread]).
        lex(det,[a]).
        lex(det,[the]).
        lex(det,[her]).
        lex(n,[consumer]).
        lex(n,[duck]).
        lex(n,[man]).
        lex(n,[woman]).
        lex(wh,[who]).
        lex(wh,[that]).
        lex(p,[to]).
        lex(iv,[died]).
        lex(iv,[ate]).
        lex(tv,[ate]).
        lex(tv,[saw]).
        lex(tv,[gave]).
        lex(dv,[gave]).
        lex(dv,[handed]).
        lex(sv,[knew]).
    end.
```

```
structure utils =
    struct
      % For each X do Y.
        foreach(X,Y) :-
            call(X),
            do(Y),
            fail.
        foreach(_,_).
        do(Y) :-
            call(Y),!.
      % Read in a sentence, terminated by a '.' and tag
      % each word with Tag.
        get_sentence(Wordlist,Tag) :-
            get0(Char),
            getrest(Char,Wordlist,Tag).
        getrest(46,[],_) :- !.
        getrest(32,Wordlist,Tag) :- !,
            get_sentence(Wordlist,Tag).
        getrest(Letter,[Word|Wordlist],Tag) :-
            getletters(Letter,Letters,Nextchar),
            name(Word,Letters,Tag),           % Note this use of name/2.
            getrest(Nextchar,Wordlist,Tag).
        getletters(46,[],46) :- !.
        getletters(32,[],32) :- !.
        getletters(Let,[Let|Letters],Nextchar) :-
            get0(Char),
            getletters(Char,Letters,Nextchar).
    end.

functor abs_parser(x/grammar_sig,y/agenda_sig,z/chart_sig) =
    struct
        inherit utils.
        structure grammar = x.
        structure agenda = y.
        structure chart = z.

        start :-
```

```
    agenda:erase_agenda,
    chart:erase_chart,
    grammar:tag(Tag),
    utils:get_sentence(Sentence,Tag),
    grammar:initial(Symbol),
    start_agenda(Sentence,0),
    start_active(Symbol,0),
    expand_edges,
    chart:chart_edge(0,_,Symbol,[]).
    start_agenda([],_).

start_agenda([Word|Words],V0) :-
    V1 is V0 + 1,
    utils:foreach(grammar:lex(Category,[Word]),
            agenda:add_to_agenda(V0,V1,Category,[])),
    start_agenda(Words,V1).

start_active(Category,Vertex) :-
    utils:foreach(grammar:rule(Category,Categories),
            agenda:add_to_agenda(Vertex,Vertex,Category,Categories)).

expand_edges :-
    agenda:remove_from_agenda(V0,V1,Category,Needed) ,!,
    extend_edges(V0,V1,Category,Needed).
expand_edges.

extend_edges(V0,V1,Category,Needed) :-
    chart:chart_edge(V0,V1,Category,Needed),
    expand_edges.
extend_edges(V0,V1,Category,Needed) :-
    chart:add_chart_edge(V0,V1,Category,Needed),
    new_edges(V0,V1,Category,Needed),
    expand_edges.

new_edges(V1,V2,Category1,[]) :-
              % Inactive edges
    utils:foreach(chart:chart_edge(V0,V1,Category2,
                                   [Category|Categories]),
```

```
                    agenda:add_to_agenda(V0,V2,Category2,Categories)).
        new_edges(V1,V2,Category1,[Category2|Categories]) :-
                      % Active edges
            start_active(Category2,V2),
            utils:foreach(chart:chart_edge(V2,V3,Category2,[]),
                    agenda:add_to_agenda(V1,V3,Category1,Categories)).
    end.

structure final = abs_parser(grammar1,agenda,chart).
```

The program is started using `final:start` and expects the user to type a sentence terminated by a period ('.').