# Modular SB-Prolog Manual

Brian Paxton

15th June 1992

## 1 Introduction

Although this document is described as a manual, it can be more accurately described as a brief overview of the facilities available in Modular SB-Prolog. Since the system is based on SB-Prolog 3.1, and is the result of work on my final year project as an undergraduate at Edinburgh University, a lot of information is already available in the following two documents :

[1] "SB—Prolog Manual", Saumya K Debray, Department of Computer Science, University of Arizona, 1988

[2] "The Implementation of a Modular Prolog System Based on Standard ML Modules", Brian Paxton, Fourth Year Report, Departments of Computer Science and Artificial Intelligence, University of Edinburgh, May 30, 1992

Other work on the modules system can be found in :

[3] "A Calculus for the Construction of Modular Prolog Programs", D T Sannella and L A Wallen, The Journal of Logic Programming, volume 12, nos 1 and 2, January 1992, pp 147-178

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

With this abundance of information already available, this document simply overviews the facilities available in Modular SB-Prolog and highlights the changes made to the original system. For an introduction to the basic features of and motivation for the modules system, see [3]. For a more detailed discussion of SB-Prolog the user is encouraged to look at [1]. However, due to complexities arising with the extra-logical predicates like assert/1 and compound/3, the user will find [2] an invaluable introduction to the techniques used by Modular SB-Prolog.

### 1.1 Terminology

The following is a summary of the terminology used in this document. This terminology is used because the modules system introduces module constructs called *structures* and *func-*

*tors* which are terms already used in the Prolog community. Hopefully this terminology prevents any such contradictions from becoming problematic.

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*).

Throughout this document "SB-Prolog" refers to the original (standard) version of SB-Prolog, and "Modular SB-Prolog" refers to the new system.

## 2 Syntax

It is sufficient to simply state in this section that Modular SB-Prolog has much the same syntax as SB-Prolog (and hence C-Prolog) and so is largely standard. A fuller description is available [1].

I include in Figure 1 a list of the operators used by Modular SB-Prolog for reference purposes. The list is that of SB-Prolog with the addition of operators for the module constructs.

The syntax for the module constructs themselves is described below. This syntax is only acceptable to consult/1.

PROGRAMS *prog*
  *prog* ::= *dec*
SIGNATURE BINDINGS *sigb*
  *sigb* ::= *atid* = *sigexpr*
FUNCTOR BINDINGS funb
  *funb* ::= *atid*(*plist*) [/ *sigexpr*] = *strexpr*
  *plist* ::= $atid_1/sigexpr_1$, ... , $atid_N/sigexpr_N$
    [`sharing` $patheq_1$ `and` ... `and` $patheq_M$]
  *patheq* ::= $id_1$ = ... = $id_N$
STRUCTURE BINDINGS *strb*
  *strb* ::= *atid* [/ *sigexpr*] = *strexpr*
SIGNATURE BINDINGS *sigexpr*
  *sigexpr* ::= *atid*
    `sig` *spec* `end`
  *spec* ::= `pred` $atid_1/nat_1$ `and` ... `and` $atid_N/nat_N$.
    `fun` $atid_1/nat_1$ `and` ... `and` $atid_N/nat_N$.
    `structure` $specstrb_1$ `and` ... `and` $specstrb_N$.
    `sharing` $patheq_1$ `and` ... `and` $patheq_N$.

| Precedence | Type | Operator |
|---|---|---|
| 1200 | xfx | [':-','-->'] |
| 1200 | fx | ':-' |
| 1198 | xfx | '::-' |
| 1150 | fy | mode |
| 1100 | xfy | ';' |
| 1050 | xfy | '->' |
| 1000 | xfy | ',' |
| 900 | fy | [not,'\-',spy,nospy] |
| 760 | fx | [inherit,pred,fun,functor,signature,structure] |
| 750 | xfx | sharing |
| 730 | xfy | and |
| 700 | xfy | '=' |
| 700 | xfx | [is,'=..','?=','\=','==','\==','@<','@>','@=<','@>=', '=:=','=\=','<','>','=<','>='] |
| 690 | fx | [struct,sig] |
| 500 | yfx | ['+','-','/\','\/'] |
| 500 | fx | ['+','-','\'] |
| 450 | xfy | ---> |
| 400 | yfx | ['*','/','//','<<','>>'] |
| 300 | xfx | mod |
| 200 | xfy | '^' |
| 100 | xfy | ':' |

Figure 1: The Operators of Modular SB-Prolog

*spec spec'*

*specstrb* ::= *atid*/*sigexpr*

STRUCTURE EXPRESSIONS *strexpr*

*strexpr* ::= *id*

`struct` *dec* `end`

*strexpr*/*sigexpr*

*atid*($strexpr_1$, ... , $strexpr_N$)

DECLARATIONS *dec*

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

`fun` $atid_1/nat_1$ `and` ... `and` $atid_N/nat_N$.

`fun` *atid* [/ *nat*] = *id*.

`pred` $atid_1/nat_1$ `and` ... `and` $atid_N/nat_N$.

`structure` *strb*.

`inherit` *atid*.

`signature` *sigb*.

`functor` *funb*.

*dec dec'*

MODULAR PROLOG IDENTIFIERS id

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

# 3 Command Line Options

In general, the interpreter is started as follows:

sbprolog -$option_1$ -$option_2$ ... -$option_n$ *bc_file*

The following is a list of the options recognised by Modular SB-Prolog :

**T** Generates a trace at entry to each called routine.

**t** Enables tracing of certain events (not of much use).

**d** Produces a disassembled dump of *bc_file* into a file named 'dump.pil' and exits.

**n** Adds machine addresses when producing trace and dump.

**s** Maintains information for the builtin *statistics*/0. Default: off.

**m *size*** Allocates *size* words (4 bytes) of space to the local stack and heap together. Default: 100000.

**p *size*** Allocates *size* words of space to the program area. Default: 100000.

**b *size*** Allocates *size* words of space to the trail stack. Default: $m/5$, where $m$ is the amount of space allocated to the local stack and heap together. This parameter, if specified, must follow the -m parameter.

# 4 Predicates Available

All the following predicates are available in the pervasive (system) signature. Most are identical to their SB-Prolog equivalents so descriptions are intentionally sparse to save space.

The SB-Prolog manual discusses some predicates which are prefixed with a $. These predicates are no longer recommended for use as many are not correctly integrated with the modules system. The user is encouraged to use only the predicates listed in this manual.

Before beginning this section it is important to point out a feature of Modular SB-Prolog. Every structure (module) in the modules environment is given a unique internal name called a 'tag' (this is simply an integer). Some of the predicates listed below require an extra argument which is the tag of the structure in which they are to be executed in order to perform their task correctly. For predicates that require this extra argument, Modular Prolog adds the argument automatically. This processing is performed by the command line, by consult/1, call/2 and by the assert/1 family and the retract/1 family of predicates. This does mean that a program is listed with this additional argument even if it did not exist in the original program.

For example, name/2 requires an extra argument, so the call

```
name(Atom,[116,101,115,116])
```

is converted to

```
name(Atom,[116,101,115,116],X)
```

by Modular SB-Prolog where X is the tag of the structure inside which the call is made.

Note that call/1 does not peform this processing, so the call

```
call(name(Atom,[116,101,115,116]))
```

fails as name/2 actually doesn't exist, name/3 is the only executable form of the predicate.

The structure tag can actually be inserted by the user as follows

```
current_structure(Tag)
name(Atom,[116,101,115,116],Tag)
```

The tag need not be the tag of the current structure, but in most cases this is what is required. However, if the user required to perform a name/2 with respect to a structure whose tag is Y (i.e. create an atom which belongs to structure Y) then the call is simply

```
name(Atom,[116,101,115,116],Y)
```

There are in fact two special structure tags used internally by the system. The top level structure (called 'root') has the structure tag 0, and the system (pervasive) structure

| Actual Predicate | Transformed To |
|---|---|
| current_structure/1 | current_structure/1 |
| name/2 | name/3 |
| read/1 | read/2 |
| read/2 (Some cases) | read/3 |
| compound/3 | compound/4 |
| bldstr/3 | bldstr/4 |
| =../2 | =../3 |
| structure/2 | structure/3 |

Figure 2: Predicates requiring a structure-tag argument (Handled automatically)

has a tag 'perv'. If 'perv' is passed to any of these special predicates then the action is performed with respect to the pervasive structure. This is of little use, but is mentioned for completeness. It is generally the case that if a structure tag passed to a predicate is not a valid one, an error is displayed and the call fails.

The predicates that require this additional structure tag argument are indicated individually later. Figure 2 lists these predicates for reference purposes. Some of these predicates are also described as being "Sensitive to fun X = Y declarations". This refers to the ability to declare equality over functions and the effect of the equalities on these predicates (this is described in [2]). Any predicates that are "Sensitive to fun X = Y declarations" can correctly handle these function equalities. The reader is encouraged to read the sections in [2] which discuss these predicates.

## 4.1 Consulting Files

```
consult(File)
consult(File,Opts)
consult(File,Opts,Preds)
```

Consults a Modular Prolog file File, using the options given in the list Opts. A list of all the predicates loaded is given in Preds. The options currently supported are :

- `v` Verbose mode. Displays the names of structures, signatures and functors as they are processed. The output is similar to that of Standard ML.
- `t` Trace mode. Sets up a trace point on every predicate loaded.

If no list of options is given, the default is [v].

As the consult progresses, warning messages and error messages are displayed on screen. Warnings are not serious and do not affect the consult. Errors are serious, and the call to consult fails. In most such cases, consult will fail indicating the error that it encountered. If consulting using verbose mode, the error can be traced quite easily as the name of a construct is only ever displayed on screen after the processing for that construct is complete.

If a call to consult fails with an error, it is important to point out that the module constructs loaded before the erroneous one will not be fully built. This is because although the constructs will be defined, any code contained within those constructs will not have been added to the database (this is done at the end of the consult in one go). The database is only guaranteed to be consistent if the consult is successful.

## 4.2 Input and Output

`writename(Term)` Writes Term out to the current output stream. No home structure name is displayed, and if the term is a compound term, only the name of the term is printed (no arguments are printed).

`writeqname(Term)` As writename/1, but names are quoted where necessary to make the output acceptable to read/1.

`put(Char)` The character whose ASCII code is Char is output to the current output stream.

`nl` Newline

`tab(Num)` Num spaces are output to the current output stream.

`tell(File)` File becomes the current output stream.

`tell(File,Num)` File becomes the current output stream. If Num is 1 then File is appended, otherwise File is overwritten.

`telling(File)` File is the current output stream.

`told` Closes the current output stream.

`get(Char)` Char is the ASCII code of the next non-blank printable character from the current input stream. If the end of file is reached -1 is returned, but the stream is not closed.

`get0(Char)` Char is the ASCII code of the next character from the current input stream. If the end of file is reached -1 is returned, but the stream is not closed.

`see(File)` File becomes the current input stream.

`seeing(File)` File is the current input stream.

`seen` Closes the current input stream.

`write(Term)` Writes Term out to the current output stream. Current operator declarations are considered and all terms are printed along with their home structure name (no structure name is displayed for terms in the root or pervasive structures).

`writeq(Term)` Writes Term out to the current output stream. No operators are considered and names are quoted where necessary (so input is acceptable to read/1).

`display(Term)` As write/1, but always writes out to the 'user' stream (the terminal).

`print(Term)` In this version of Prolog, this is simply another name for write/1.

`print_al(N,Term)` If Term is a number or an atom, this prints Term left-aligned to a width N (padding extra spaces to the right). If Term is longer than N then Term is printed with no padding. If Term is an atom, no structure information is displayed, only the atom itself.

`print_ar(N,Term)` As print_al/2, but right-aligned.

`errmsg(Term)` As write/1, but always writes to the 'stderr' stream (the standard error stream).

`read(Term)` The internal form of read/1 is read/2. (Processed automatically).

`read(Term,Vars)` The internal form of this is read/3.

`read(Term,Tag)` Read a term, terminated by a period ('.') from the current input stream. Sensitive to 'fun X = Y' declarations. Processing performed with respect to the structure whose tag is Tag (usually the current structure). For details, see [1].

`read(Term,Vars,Tag)` Read a term, terminated by a period ('.') from the current input stream. Vars is a list of variables encountered in Term, and is a list of *VariableName* = *ActualVariable* pairs. Sensitive to 'fun X = Y' declarations. Processing performed with respect to the structure whose tag is Tag (usually the current structure).

`portray_term(Term)` As write/1 but variables are displayed as 'V*n*' instead of '*_internaladdress*'.

`portray_clause(Clause)` Writes Term to the standard output stream, in clause format (the same form displayed by listing/0). Variables in Clause are displayed as 'V*n*'.

### 4.3 Term Comparison

`X = Y` The terms X and Y are unified.

`X ?= Y` The terms X and Y can unify (but unification is not actually performed).

`X \= Y` The terms X and Y cannot unify.

`X == Y` The terms X and Y are identical. For example 'X == Y' fails (unless they have been explicitly unified earlier).

`X \== Y` The terms X and Y are not identical.

`X =:= Y` The arithmetic expressions X and Y are equal when evaulated.

`X =\= Y` The arithmetic expressions X and Y are not equal when evaulated.

`X < Y` The arithmetic expression X is less than expression Y when evaluated.

`X =< Y` The arithmetic expression X is not greater than expression Y when evaluated.

`X >= Y` The arithmetic expression X is not less than expression Y when evaluated.

`X > Y` The arithmetic expression X is greater than expression Y when evaluated.

The rest of these comparison terms use the SB-Prolog standard order of terms. The order is as follows (in ascending order) : variables, numbers, atoms, complex terms (ordered by arity, then name). However, since the modules system identifies names local to a structure by internally prefixing the name with a structure tag, the ordering of atoms and compound terms may seem rather convoluted, but necessary.

`X @=< Y` The term X is not greater than Y.

`X @< Y` The term X is less than Y.

`X @> Y` The term X is greater than Y.

`X @>= Y` The term X is not less than Y.

`compare(Op,X,Y)` The result of comparing X and Y is Op, which may be '>', '<' or '='. Op need not be bound at the time of call.

### 4.4 Arithmetic

`X is Y` The number X is the result of evaluating the arithmetic expression Y.

`eval(X,Y)` This is equivalent to Y is X.

`floatc(Num,Mant,Exp)` Here, $Num = Mant \times 2^{Exp}, 0 \leq Mant \leq 1$

`exp(X,Y)` Here, $Y = e^X$ $(X = log_e(Y))$.

`square(X,Y)` Here, $Y = X^2$ $(X = \sqrt{Y})$.

`sin(X,Y)` Here, $Y = sin(X)$ $(X = arcsin(Y))$, $\frac{-\pi}{2} \leq Y \leq \frac{\pi}{2}$.

`floor(X,Y)` Integer to float conversion (and vice versa). The integer Y is the largest integer not greater than the absolute value of float X.

The usual arithmetic operators can be used inside arithmetic expressions. These are listed as follows :

1. `X + Y` addition.
2. `X - Y` subtraction.
3. `X * Y` multiplication.
4. `X / Y` division.
5. `X // Y` integer division.
6. `X mod Y` integer modulo.
7. `-X` unary minus.

8. `X /\ Y` integer bitwise conjunction.

9. `X \/ Y` integer bitwise disjunction.

10. `X << Y` integer bitwise shift left by Y places.

11. `X >> Y` integer bitwise shift right by Y places.

12. `\` integer bitwise negation.

13. `sqrt(X)` square root.

14. `square(X)` square.

15. `integer(X)` convert to integer.

16. `float(X)` convert to float.

17. `exp(X)` exponential.

18. `ln(X)` natural logarithm.

19. `sin(X)` sine.

20. `arcsin(X)` inverse sine.

The expressions 13 through 20 are actually converted into separate calls to floor/2, floatc/2, exp/2, square/2 and sin/2 automatically by the system. For example, the call

```
X is 1 + sqrt(Y).
```

becomes the conjunction

```
square(U,Y),
X is 1 + U.
```

## 4.5 Term Handling

`atom(Term)` Term is an atom (which here means any 0-arity compound term).

`atomic(Term)` Term is an atom (which here means any 0-arity compound term) or a number.

`integer(Term)` Term is an integer.

`number(Term)` Term is an integer or a float.

`arg(N,Term,Item)` Item is the $N^{th}$ argument of compound term Term. Arguments are numbered from 1 and the call fails if N is out of range.

`compound_term(Term)` Term is an compound term.

`arity(Term,Arity)` Arity is the arity of compound term Term.

`real(Term)` Term is a floating point number.

`float(Term)` Term is a floating point number.

`is_buffer(Term)` Term is a buffer (see later).

`function(Term)` Term is a function application.

`predicate(Term)` Term is a predicate application.

`dismantle_name(Atom,Name,Tag)` The name (printable part) of Atom is Name and its home structure tag is Tag. Suitable combinations of arguments can be used to build and decompose atoms. For example, to move an atom Atom to a structure whose tag is New, use the following :

```
dismantle_name(Atom,Name,_),
dismantle_name(NewAtom,Name,New)
```

`var(Term)` Term is an unbound variable.

`nonvar(Term)` Term is not an unbound variable.

`gennum(Number)` Successive calls return unique integers.

`gensym(Atom,Term)` Uses gennum/1 to generate a distinct integer which is concatenated to the atom Atom to give Term. For example :

```
gensym(test,X)
```

would return 'test124' (if the current gennum/1 value was 124).

`X =.. Y` The internal form of =../2 is =../3. (Processed automatically).

`'=..'(X,[Name|List],Tag)` The compound term X has name Name and arguments given in List. Sensitive to 'fun X = Y' declarations. Processing performed with respect to the structure whose tag is Tag (usually the current structure).

`name(Atom,List)` The internal form of name/2 is name/3. (Processed automatically).

`name(Atom,List,Tag)` Atom is an atom or a number which corresponds to the list of ASCII codes List. Sensitive to 'fun X = Y' declarations. Processing performed with respect to the structure whose tag is Tag (usually the current structure).

`compound(Term,Name,Arity)` The internal form of compound/3 in compound/4. (Processed automatically).

`compound(Term,Name,Arity)` The compound term Term has name Name and arity Arity. Sensitive to 'fun X = Y' declarations. Processing performed with respect to the structure whose tag is Tag (usually the current structure). This is the new name of the SB-Prolog predicate functor/3.

**bldstr(Name,Arity,Term)** The internal form of bldstr/3 is bldstr/4. (Processed automatically).

**bldstr(Name,Arity,Term,Tag)** Given Name and Arity, a compound term *Name/Arity* is built. Sensitive to 'fun X = Y' declarations. Processing performed with respect to the structure whose tag is Tag (usually the current structure).

### 4.6 Database

NOTES :

1. Some of the database predicates used here can take an optional argument which identifies a structure inside which that operation is to be performed. For example, the call :

   ```
   assert(data(1),X)
   ```

   (where X is the tag of a structure 'test') would assert the clause data(1) into the structure 'test'.

   However, any database manipulation predicate which can take an optional structure tag argument has a restriction imposed on the clause type it can take as an argument (this is discussed in detail in [2]). If an operation is to be performed in a remote structure, the clause must be moved to that structure before the operation can go ahead. This moving operation cannot accept any clauses which contain 'Outer-structure references'. This basically means that if the clause has to be moved, it can only contain terms belonging to the same structure (or the pervasive structure). For example :

   ```
   assert(test(one),X)              % Okay
   assert(test(x:one),X)            % Illegal
   assert(x:test(x:one),X)          % Okay
   assert(x:test(user),X)           % Okay
   ```

   This last is legal as user/0 is a pervasive function. The predicates which can generate this error are the assert family, retract/2, retractall/2 and call/2.

2. When asserting a clause for a predicate into the database an index is set up on the first argument of the predicate (a kind of optimization). This index can be changed by using a call to index/3. For example, to set up an index on the second argument of the predicate test/3 instead of the first use the declaration :

   ```
   :- index(test,3,2).
   ```

   This index is respected by all assert predicates.

3. Arguments to any predicate to assert or retract a clause cannot take as an argument a function application of a function declared during a consult. In other words, any function declared using a ‘fun X’ or ‘fun X = Y’ declaration is not a valid argument to assert or retract and the call fails with an error.

Adding clauses to the database :

`assert(Clause)` Add clause Clause to the end of the clauses for the corresponding predicate.

`assert(Clause,Tag)` Add clause Clause to the end of the clauses for the corresponding predicate inside the structure whose tag is Tag.

`assert(Clause,Ref)` Add clause Clause to the end of the clauses for the corresponding predicate. Ref is bound to the database reference of the clause.

`assert(Clause,Ref,Tag)` Add clause Clause to the end of the clauses for the corresponding predicate inside the structure whose tag is Tag. Ref is bound to the database reference of the clause.

`assert(Clause,AZ,N,Ref)` Add clause Clause into the database using indexing on the $N_{th}$ argument (if N is 0 then no index is created). If AZ equals 0 then the clause is added to the beginning of the clauses for the corresponding predicate. If AZ equals 1 then the clause is added to the end. Ref is bound to the database reference of the clause.

`asserta(Clause)` As assert/1, but add to the end of the clauses for the corresponding predicate. Due to a restriction imposed on asserta/1,2,3, no indexing is placed on arguments to clauses asserted using asserta/1,2,3.

`asserta(Clause,Tag)` As assert/2, but add to the beginning of the clauses for the corresponding predicate.

`asserta(Clause,Ref)` As assert/2, but add to the beginning of the clauses for the corresponding predicate.

`asserta(Clause,Ref,Tag)` As assert/3, but add to the beginning of the clauses for the corresponding predicate.

`assertz(Clause)` As assert/1.

`assertz(Clause,Tag)` As assert/2.

`assertz(Clause,Ref)` As assert/2.

`assertz(Clause,Ref,Tag)` As assert/3.

`asserti(Clause,N)` As assert/1, but an index is placed on the $N^{th}$ argument of the clause.

`asserti(Clause,N,Tag)` As assert/2, but an index is placed on the $N^{th}$ argument of the clause.

`assert_union(Head1,Head2)` Head1 and Head2 are predicate applications. A final clause is added to the code for the predicate of Head1 :

```
Head1 :- Head2.
```

This is a very low level assert and the arguments given in Head1 and Head2 are ignored - it is only the predicate name and arity of each argument that is important.

Removing clauses from the database :

`retract(Clause)` Remove the first clause which matches Clause from the database.

`retract(Clause,Tag)` Remove the first clause which matches Clause inside the structure whose tag is Tag. Cannot accept outer-structure references.

`retractall(Head)` Remove all clauses from the database whose head matches Head.

`retractall(Head,Tag)` Remove all clauses whose head matches Head from the structure whose tag is Tag. Cannot accept outer-structure references.

`abolish(Head)` Head is a predicate application and all clauses for that predicate are removed.

`abolish(Name,Arity)` All clauses for the predicate Name/Arity are removed.

Accessing clauses from the database :

`clause(Head,Body)` Given Head, the database is searched for clauses matching Head and the body of that clause is unified with Body. Other matches are generated through backtracking.

`clause(Head,Body,Ref)` As clause/2, but the database reference of the matching clause is also returned in Ref. Can be called with Head set or Ref set.

`listing(Pred)` Pred is a *name/arity* pair or a list of pairs. The predicates named are listed out.

`listing` Lists out all the currently defined signatures, structures and functors. The format is clear, but not in a form suitable for a future consult/1.

`list_module(Tag)` Lists the contents of the structure whose tag is Tag.

Internal database predicates :

`globalset(Head)` Head should be a predicate application with one argument (of the form *Name*(*Argument*)) where the argument is either a variable, number or atom. This predicate has a similar effect to the call

```
retract(Name(_)), assert(Name(Argument)).
```

If *argument* is a variable, the effect of this call is to make the variable 'global' and accessible through a call to *name*/1.

`recorda(Key,Term,Ref)` Term is recorded as the first item for key Key, and has the database reference Ref. If Key is a compound term, then only the name of that term is significant.

`recordz(Key,Term,Ref)` Term is recorded as the last item for key Key, and has the database reference Ref.

`recorded(Key,Term,Ref)` Given a key Key, Term and Ref are successively unified with items stored under that key.

`erase(Ref)` Erase an entry whose database reference is Ref.

`instance(Ref,Term)` Given a database reference Ref, Term is the entry recorded there.

Accessing state of the database :

`current_predicate(Name,Term)` Generates, through backtracking, all currently defined predicates. A predicate test/2 would return Name as 'test' and Term as test(_,_).

`current_function(Name,Term)` As current_predicate/2, but for functions only.

`current_atom(Atom)` As current_predicate/2, but for atoms only.

`predicate_property(Term,Type)` Given a term Term corresponding to a predicate application, Type is bound to the type of that predicate ('interpreted' or 'compiled'). Fails if the predicate is not defined.

`pervasive(Term)` Term is a *name/arity* pair corresponding to a predicate or function constant. Succeeds if that constant is a pervasive function or predicate.

`pervasive0(Term)` Term is a function or predicate application and the call succeeds if that term corresponds to a pervasive function or predicate.

`pervasive_function(Term)` As pervasive/1 but succeeds for pervasive functions only.

`pervasive_function0(Term)` As pervasive0/1 but succeeds for pervasive functions only.

`pervasive_predicate(Term)` As pervasive/1 but succeeds for pervasive predicates only.

`pervasive_predicate0(Term)` As pervasive0/1 but succeeds for pervasive predicates only.

`functor_name(Name)` Generates, through backtracking, the names of all currently defined functors.

`structure_name(Name)` Generates, through backtracking, the names of all currently defined structures.

`signature_name(Name)` Generates, through backtracking, the names of all currently defined signatures.

`symtype(Term,Type)` Term is a function or predicate application and Type is bound to 0 if Term corresponds to a function, 1 for an interpreted predicate, 2 for a compiled predicate and 3 for a buffer.

## 4.7 Sets

`setof(Pattern,Call,Set)` Set is the ordered set of patterns (Pattern) such that Call is provable. Succeeds if Set is empty. As in Standard Prolog use ^/2 to existentially quantify variables in the call, as follows :

```
setof(X, X^member(X,List), Set)
```

`bagof(Pattern,Call,List)` As setof/3, but List is unordered and may contain duplicates.

`findall(Pattern,Call,List)` As bagof/3, but any variables in Call which do not occur in Pattern are treated as local and alternatives are not returned for different bindings of these variables.

## 4.8 List Utilities

`length(List,Len)` Len is the length of List.

`append(List1,List2,Result)` Classic list concatenation.

`member(Item,List)` Classic list membership.

`sort(List,Sorted)` List is sorted into the list Sorted in the standard order. Duplicates are removed.

`keysort(List,Sorted)` List is a list of *Key–Value* pairs. The list is sorted by the keys into list Sorted.

`reverse(List,Reverse)` Reverse is the reverse of List.

`merge(List1,List2,Merged)` Merged is the concatenation of List1 and List2, with duplicates removed.

`absmember(Item,List)` As member/2, but uses ==/2 for comparisons and not unification.

`absmerge(List1,List2,Merged)` As merge/3, but duplication checks are performed using ==/2 and not unification.

`closetail(List)` If the tail of List is a variable, the variable is bound to the empty list (and so 'closes' the tail of the list). For example :

```
| ?- X = [a,b,c|Y],closetail(X).

X = [a,b,c]
Y = []
```

`nthmember(Item,List,N)` Item is the $N^{th}$ member of List. Various combinations of unbound arguments can be used.

### 4.9 Making Calls

`(X,Y)` Conjunction

`(X;Y)` Disjunction

`(X -> Y)` If X then Y else fail.

`(X -> Y ; Z)` If X then Y else Z.

`not Goal` If Goal succeeds, then fail, else succeed.

`\+` Goal As not/1.

`call(Goal)` Execute Goal.

`call(Goal)` Execute Goal inside the structure whose tag is Tag. Goal must not contain any 'Outer-structure references' (discussed earlier under assert/2).

### 4.10 Structure Handling

`current_structure(Tag)` The internal form of current_structure/1 is current_structure/2. (Processed automatically).

`current_structure(Tag,Current)` Tag is the tag of the current structure. The second argument is added automatically by the system and is the current structure tag. This predicate is therefore defined simply as :

```
current_structure(X,X).
```

`structure(Tag,Name)` As structure/3 with respect to the top level (root) structure. (Processed automatically).

`structure(Tag,Name,WRT)` Internal form of structure/3 is structure/4.

`structure(Tag,Name,WRT,Current)` Tag is the structure tag of the structure whose name is Name with respect to the structure whose tag is WRT. Current must be the tag of the current structure.

### 4.11 Operating System Access

`cputime(Time)` Time is the elapsed time since Prolog was started (in milli-seconds).

`syscall(Number,Args,Result)` Executes the Unix system call number Number, with arguments Args, giving the result Result. For details, see [1].

`system(Command)` Invokes a shell to execute Command. For example :

```
system(ls).
system('cat testfile | more').
```

### 4.12 Other Stuff

`statistics` Displays information on memory usage.

`statistics(Keyword,Info)` Given Keyword returns statistical information in Info as follows :

| Keyword | Info |
|---|---|
| runtime | [Prolog cputime, time since last statistics/2 call] |
| memory | [total memory,0] |
| core | [total memory,0] |
| program | [program space used, program space free] |
| heap | [program space used, program space free] |
| global_stack | [global stack used, global stack free] |
| loca_stack | [local stack used, local stack free] |
| trail | [trail stack used, trail stack free] |
| garbage_collection | [0,0] |
| stack_shifts | [0,0] |

`pred_undefined(Term)` Given a predicate application, succeeds if that predicate is undefined.

`break` Suspends execution and enters a break level. To terminate break level, press CTRL-D and program will continue execution. break/0 has the same effect as pressing CTRL-C during execution.

`abort` Aborts execution and goes back to top level.

`repeat` Always succeeds, even on backtracking.

`loaded_mods(Name)` Returns the names, through backtracking, of all the system library modules currently loaded. (Nothing to do with the new module system.)

`defined_mods(Name,Exportlist)` Returns the names and export lists, through backtracking, of all the system library modules currently available. (Nothing to do with the new module system.)

`load(File)` Load a file containing compiled Prolog code.

`!` Discard all choice points made since parent goal was called.

`fail` Always fails.

`true` Always succeeds.

`halt` Terminate Prolog session.

`getclauses(File,Clauselist)` Used by the compiler and the old version of consult to load in all the clauses for a given file. Clauselist is a list of entries of the following form:

pred(*name,arity,unused,unused,clause list*)

where *clause list* is a list of the following entries :

fact(*clause head*)

or

rule(*clause head,clause body*)

This predicate cannot be used to load clauses inside Modular Prolog constructs.

`getclauses(File,Clauselist,Predlist)` As getclauses/2, but also returns Predlist, a list of the predicates loaded (given as *name/arity* pairs).

`attach(Item,List)` Given an open ended list List (a list of the form [$element_1$, $element_2$, ..., $element_N$ |Variable]), Item is added into the tail of the list. For example :

```
| ?- X = [a,b|_], attach(c,X).

X = [a,b,c|_1396136]
yes
```

`expand_term(Term1,Term2)` Term1 is expanded to form Term2. In Modular Prolog, this simply means that if Term1 is a definite clause grammar rule, then it is converted to clause form in Term2. If it is not a grammar rule, Term1 = Term2.

### 4.13 Tracing Facilities

In order to trace a predicate in Modular SB-Prolog, the predicate concerned has to have a trace point or a spy point set on it — no other predicates will be traced. In order to make setting trace points easier, the 't' option in consult/2,3 can be used to automatically set trace points on all the predicates that are loaded during that consult.

Tracing works in much the same way as any other tracing package and traces the predicate on call, successful exit and failure. The following commands are available :

| | |
|---|---|
| c newline | creep |
| a | abort |
| b | break (enter a break level) |
| f | fail |
| h | help |
| l | leap (to next spy point) |
| n | nodebug |
| q | quasi-skip (as skip, but stops at spy points) |
| s | skip (do not trace the current goal) |
| r | retry (go back to Call port of current goal) |
| e | end the Prolog session |

Tracing is described in more detail in [1] and has not been changed in any way.

`debug` Enable debugging mode.

`nodebug` Disable debugging mode.

`trace(Pred)` Pred is a *name/arity* pair (or a list of such pairs) and trace points are set on those predicates.

`untrace(Pred)` As trace/1, but removes trace points.

`spy(Pred)` As trace/1, but sets spy points.

`nospy(Pred)` As trace/1, but removes spy points.

`trace` This is a very low level tracer which simply displays, at the Warren Abstract Machine level, calls to every predicate (including pervasive ones). Not of much use and prints out a large amount of information.

`untrace` Disables the tracing set by trace/0.

`debugging` Displays information about the status of debug mode and currently set spy and trace points.

`tracepreds(List)` Returns a list of all predicates which currently have trace points set. List is a list of *name/arity* pairs.

`spypreds(List)` As tracepreds/1, but for spy points.

### 4.14 Definite Clause Grammars

`dcg(Rule,Clause)` Converts a DCG rule Rule, to a Prolog clause Clause.

`phrase(Start,List)` The normal method of calling a grammar. Start is the start symbol of the grammar (a non-terminal) and List is the sentence to be parsed.

`phrase(Start,In,Out)` Calls a grammar. Start is the grammar start symbol, In is the sentence to be parsed (a list), and Out is what is left over after parsing is complete.

`'C'(L1,Terminal,L2)` Used by DCGs and is defined 'C'([X|S],X,S).

### 4.15 Library Predicates

`access(File,Mode)` Calls the operating system to access a file. For example,

```
access(test,0).
```

tests to see if the file 'test' exists.

`access(File,Mode,Success)` As access/2 but Success is the exit level returned by the operating system (normally 0).

`call_ref(Call,Ref)` Calls the predicate whose database reference is Ref, using Call as the call.

`call_ref(Call,Ref,Tr)` As call_ref/2 and Tr is used to handle 'trust' optimization. See [1] for details.

`errno(Number)` The last error which occured during a call to an operating system function.

`flags(Number,Value)` Can be used to set low level tracing, etc. but is not of much use to the user.

`index(Name,Arity,N)` Usually the compiler or the assert family creates a predicate which has an index on the first argument. A call to index/3 will set an index on the $N^{th}$ argument of the predicate *Name/Arity*, and all clauses created after this will respect this declaration.

`mode(Mode)` Mode declarations for use by the compiler. See [1] for details.

`mode(Name,Arity,Mode)` Mode declarations for use by the compiler. See [1] for details.

`nodynload(Name,Arity)` Disables the dynamic loading of predicate *Name/Arity*. Since the dynamic loader in Modular SB-Prolog only loads pervasive predicates (a necessary restriction), the argument must correspond to a pervasive predicate (otherwise the declaration will do nothing).

`op(Prec,Type,Name)` Sets up an operator whose name is Name, with type Type and precedence Prec. Name may be a list of names. Operator types are fx, fy, xf, yf, xfx, xfy or yfx.

`save(File)` Dumps Prolog's memory into the file File. The program space, heap, stacks and internal registers are all saved. The state can be restored using restore/1.

`restore(File)` Restores Prolog's memory from the file File, which was previously saved using save/1.

`exists(File)` Succeeds if File exists in the current directory.

### 4.16 The Profiler

(This is identical to the SB-Prolog profiler so see [1] for details.)

The profiler allows the user to set time points and count points on predicates — a time point times the length of time a call to a predicate takes to execute and a count point counts the number of calls to a predicate.

`count(Pred)` Pred is a *name/arity* pair, or a list of pairs, and count points are set on each predicate in the list.

`time(Pred)` As count/1, but sets time points.

`nocount(Pred)` As count/1, but removes count points.

`notime(Pred)` As count/1, but removes time points.

`profiling` Displays the status of profiling, and the predicates with time or count points set on them.

`prof_reset(Pred)` As count/1, but simply resets the counter and timer values on the predicates to 0.

`resetcount(Pred)` As count/1, but simply resets the counter values on the predicates to 0.

`resettime(Pred)` As count/1, but simply resets the timer values on the predicates to 0.

`profile` Disables profiling.

`noprofile` Enables profiling.

`timepreds(Preds)` Returns a list of predicates which have time points set on them. Preds is a list of *name/arity* pairs.

`countpreds(Preds)` As timepreds/1 but for count points.

`prof_stats` Displays time and counting information on all profiled predicates.

`prof_stats(Reset)` As prof_stats/0 but if Reset equals 1, then all count and time points are reset to 0.

### 4.17 Handling Buffers

Buffers are introduced in [1] but are used only for low level system predicates. I shall not discuss the following predicates in any detail, but mention them only for completeness.

`alloc_perm(Size,Name)` A buffer of size Size (in bytes) is allocated in the program space (and so is permanent).

`alloc_heap(Size,Name)` A buffer of size Size (in bytes) is allocated on the heap (and so is deallocated on backtracking).

`trimbuff(Type,Name,Length)` The buffer Name is trimmed to be Length bytes long (if possible). Type is 0 for a program space buffer, 1 for a heap buffer.

`substring(Dir,NumBytes,Const,Locin,Buff,Locout)` ???

`subnumber(Dir,NumBytes,NumCon,Locin,Buff,Locout)` ???

`subdelim(Dir,Delim,Const,Locin,Buff,Locout)` ???

`conlength(Term,Length)` Given a term Term, its length is returned in Length. Term may be an atom, a buffer or an integer. Note that if the argument is an atom, the length returned is the length of the atom plus the structure tag information, which takes up a minimum of 3 bytes. To remove structure tag information, use dismantle_name/3 as follows :

```
                dismantle_name(Atom,Newatom,_),
                conlength(Newatom,Length).
```

`hashval(Arg,Size,Hashval)` Hashes values ???

# 5 System Functions

The following is a list of the pervasive functions listed in the pervasive signature. It is included here for reference only.

Definite Clause Grammars :

| | |
|---|---|
| '- ->'/2 | '{}'/0 |
| '{}'/1 | |

Operators (used by op/3) :

| | |
|---|---|
| fx/0 | xfx/0 |
| fy/0 | xfy/0 |
| yfx/0 | xf/0 |
| yf/0 | |

Files :

| | |
|---|---|
| stderr/0 | end_of_file/0 |
| user/0 | |

Statistics (used by statistics/2) :

| | |
|---|---|
| runtime/0 | core/0 |
| memory/0 | stack_shifts/0 |
| heap/0 | program/0 |
| global_stack/0 | local_stack/0 |
| trail/0 | garbage_collection/0 |

Predicate types (used by predicate_property/2) :

| | |
|---|---|
| interpreted/0 | compiled/0 |

Compare (used by compare/3) :

| | |
|---|---|
| '<'/0 | '>'/0 |
| '='/0 | |

Mathematical (used by is/2) :

| | |
|---|---|
| '/'/2 | '-'/2 |
| '<<'/2 | '+'/2 |
| '-'/1 | '+'/1 |
| '//'/2 | '\/'/2 |
| '>>'/2 | '/\'/2 |
| '*'/2 | '\'/1 |
| mod/2 | sqrt/1 |
| square/1 | arcsin/1 |
| integer/1 | float/1 |
| exp/1 | ln/1 |
| sin/1 | |

Compile/consult/etc arguments :

| | |
|---|---|
| '++'/0 | t/0 |
| '?'/0 | v/0 |
| nv/0 | a/0 |
| c/0 | d/0 |
| e/0 | s/0 |
| '+'/0 | '-'/0 |

Others :

| | |
|---|---|
| ':'/2 | '/'/2 |
| perv/0 | sharing/2 |
| inherit/1 | pred/1 |
| fun/1 | functor/1 |
| structure/1 | signature/1 |
| struct/1 | sig/1 |
| and/2 | '[]'/0 |
| '.'/2 | '::-'/2 |
| ':-'/1 | ':-'/2 |

## 6 The Compiler

I will not discuss the compiler here, but refer to [1] instead. The compiler remains is exactly the same form as SB-Prolog and so is used in exactly the same way.

If a Modular Prolog user wishes to run a compiled SB-Prolog program, there are problems. The compiler currently compiles only standard SB-Prolog code and cannot accept Modular SB-Prolog code. Not only that, if the resulting compiled code is loaded into Modular SB-Prolog it is loaded into the pervasive structure. Since the pervasive structure has a well-defined signature, the code that has been loaded will actually be hidden inside the pervasive structure and cannot be accessed directly from the command line. A simple hack is available to run these compiled programs and is as follows : Instead of typing calls at the command line as normal, type the following :

```
$read(X), call(X).
```

and then type the call you wish to make. This works because $read/1 is an internal system read routine which ignores all module information and does not process input like other versions of read.

# 7 Known Bugs

The following is a list of the known bugs in Modular SB-Prolog. Since SB-Prolog is itself a buggy system, most of these bugs are inherited from the original system and are not a consequence of the modules system.

1. Decompilation of clauses (for listing/0,1, retract/1,2, etc.)

   If we assert the following clauses into the database :

   ```
   test(a(X,Y),Y).
   test(a(X,Y),X).        *
   test(a(X,Y,Z),Y).
   test(a(X,Y,Z),Z).
   ```

   The only clause that can be decompiled is the second one (marked *). The program executes correctly, but cannot be decompiled. Decompilation is required when listing clauses, retracting clauses or using clause/2,3.

   (For those that are interested, this is because when a clause is asserted into the database using assert/1, the clause is compiled into two different forms - one for execution, and one for decompilation. The code used for execution seems fine, but there is (at least one) bug in the code for decompilation).

2. Certain clauses displayed using portray_clause/1 do not display very well. For example :

   ```
   | ?- portray_clause((test :- one,(two;three))).
   test :-
       one,
       (two ;
   three
       ).
   yes
   ```

3. If a number is passed to currsym/2 instead of a buffer, a function application or a predicate application, the Prolog system is terminated (instead of simply failing the call).

4. When moving clauses to remote structure in order to perform a call/2, assert/2 or retract/2, if the predicate in the head of the clause is a pervasive one, then all other items in the clause must also be pervasive. For example :

```
| ?- structure(X,a),
     assert(atom(one),X).
*** Error: Cannot move clause - contains references to substructures
no
```

5. The profiler predicates notime/1 and nocount/1 are buggy, and currently do not remove time or count points on predicates. (The call succeeds, but nothing happens).

6. The final bug seems to be a bug in the compiler, so is not of immediate concern to Modular SB-Prolog. This bug is difficult to trace, but generally, problems arise when using clauses of the form :

   ```
   test(......) :-
         !,
         calla -> callb ; callc.
   ```

   This clause will execute correctly when consulted (asserted) into the database, but will not execute if compiled. This can be very difficult to spot, but is easily rectified by placing parenthesis around the conditional :

   ```
   test(......) :-
         !,
         ( calla -> callb ; callc ).
   ```

## 8 Differences from SB-Prolog

Some of the major differences between SB-Prolog and Modular SB-Prolog are given here for reference.

- The SB-Prolog predicate functor/3 is now called compound/3.
- There are no extension table facilities in Modular SB-Prolog.
- The dynamic loader cannot be used for user predicates, only system (pervasive) ones.
- The macro expander is not used by the new version of consult/1, but remains intact for use by the old version (now called $oldconsult/1) and the compiler.

## 9 Errata to SB-Prolog Manual

The SB-Prolog manual distributed with version 3.1 of the system is actually for SB-Prolog version 3.0. This means that several items are out of date. These are as follows:

- Sections 4.3 and parts of 5.2 discuss the comparison of integers and floats and a technique whereby numbers are considered equal if they are ‘close enough’. This was introduced to get round the problems of finite precision floating point numbers. This feature has been removed, so these sections should be ignored.

- Section 5.2 does not mention the introduction of the arithmetic functions sqrt/1, square/1, integer/1, etc. which have been added to the system. The discussion given in this manual is consistent with version 3.1 of SB-Prolog.

# Index