

The SB-Prolog System, Version 3.0

A User Manual

edited by
Saumya K. Debray

from material by

David Scott Warren
Suzanne Dietrich
SUNY at Stony Brook

Fernando Pereira
SRI International

Department of Computer Science
University of Arizona
Tucson, AZ 85721

September 1988

Contents

1	Introduction	5
2	Getting Started	6
2.1	The Dynamic Loader Search Path	6
2.2	System Directories	7
2.3	Invoking the Simulator	7
2.4	Executing Programs	8
2.4.1	Compiling Programs	8
2.4.2	Loading Byte Code Files	9
2.4.3	Consulting Programs	10
2.5	Execution Directives	11
3	Syntax	11
3.1	Terms	11
3.2	Operators	14
3.3	Clause?	17
3.4	Rule?	17
3.5	Query?	17
4	SB-Prolog: Operational Semantics	18
4.1	Standard Execution Behaviour	18
4.2	Cuts and If-Then-Else	18
4.3	Unification of Floating Point Numbers	19
5	Evaluable Predicates	19
5.1	Input and Output	20
5.1.1	File Handling	20
5.1.2	Term I/O	21
5.1.3	Character I/O	22
5.2	Arithmetic	22
5.3	Convenience	25
5.4	Extra Control	26
5.5	Meta-Logical	26
5.6	Sets	29
5.7	Comparison of Terms	30
5.8	Buffers	31
5.9	Modification of the Program	32

5.10	Internal Database	35
5.11	Information about the State of the Program	36
5.12	Environmental	38
5.13	Global Values	41
5.14	Exotica	42
6	Debugging	44
6.1	High-Level Tracing	44
6.2	Low-Level Tracing	47
7	The Simulator	47
7.1	Invoking the Simulator	47
7.2	Simulator Options	48
7.3	Interrupts	49
8	The Compiler	50
8.1	Invoking the Compiler	50
8.2	Compiler Options	51
8.3	Assembly	52
8.4	Compiler Directives	52
8.4.1	Mode Declarations	52
8.4.2	Indexing Directives	54
9	Libraries	54
10	Macros	55
10.1	Defining Macros	56
10.2	Macro Expander Options	57
11	Extension Tables: Memo Relations	57
12	Definite Clause Grammars	60
13	Profiling Programs	62
14	Other Library Utilities	65
15	CREDITS	66
A	Evaluable Predicates of SB-Prolog	72

B	A Note on Coding for Efficiency	75
B.1	Avoiding Creation of Backtrack Points	75
B.2	Minimizing Data Movement Between Registers	77
B.3	Processing All Arguments of a Term	77
B.4	Testing Unifiability	78
C	Adding Builtins to SB-Prolog	79

List of Figures

1	Structure for the Function <code>.(1,.(2,.(3,[])))</code>	13
2	Structures for the Functions <code>[X L]</code> and <code>[a,b L]</code>	14
3	Extension Table Example	58

List of Tables

1	Operator Priorities	16
2	Inline Predicates of SB-Prolog	20
3	Run Time Statistics Predicates	39
4	Syscall Numbers for Some Unix Systems Calls	41
5	Complementary Tests Recognized by the Compiler	75

Abstract

SB-Prolog is a Prolog system for Unix¹-based systems. The core of the system is an emulator, written in C for portability, of a Prolog virtual machine that is an extension of the Warren Abstract Machine. The remainder of the system, including the translator from Prolog to the virtual machine instructions, is written in Prolog. Parts of this manual, specifically the sections on Prolog syntax and descriptions of some of the builtins, are based on the C-Prolog User Manual by Fernando Pereira.

1 Introduction

SB-Prolog is a Prolog system based on an extension of the Warren Abstract Machine². The WAM simulator is written in C to enhance portability. Prolog source programs can be compiled into *byte code* files, which contain encodings of WAM instructions and are interpreted by the simulator. Programs can also be interpreted via *consult*.

SB-Prolog offers several features that are not found on most Prolog systems currently available. These include: compilation to object files; dynamic loading of predicates; provision for generating executable code on the global stack, which can be later be reclaimed; an *extension table* facility that permits memoization of relations. Other features include full integration between compiled and interpreted code, and a facility for the definition and expansion of macros that is fully compatible with the runtime system.

The system incorporates tail recursion optimization, and performs clause indexing in both compiled and interpreted code. However, there is no garbage collector for the global stack. This may be incorporated into a later version.

One of the few luxuries afforded to a person giving software away for free is the ability to take philosophical stances without hurting his wallet. Based on our faith in the “declarative ideal”, viz. that pure programs with declarative readings are Good, we have attempted to encourage, where possible, a more declarative style of programming. To this end, we have deliberately chosen to not reward programs containing cuts in some situations where more declarative code is possible (see Appendix B). We have also resisted the temptation to make *assert* less expensive. We hope this will help promote a better programming style.

¹Unix is a trademark of AT&T.

²D. H. D. Warren, “An Abstract Prolog Instruction Set”, Tech. Note 309, SRI International, 1983.

2 Getting Started

This section is intended to give a broad overview of the SB-Prolog system, so as to enable the new user to begin using the system with a minimum of delay. Many of the topics touched on here are covered in greater depth in later sections.

2.1 The Dynamic Loader Search Path

In SB-Prolog, it is not necessary for the user to load all the predicates necessary to execute a program. Instead, if an undefined predicate *foo* is encountered during execution, the system searches the user's directories in the order specified by the environment variable `SIMPATH` until it finds a directory containing a file *foo* whose name is that of the undefined predicate. It then dynamically loads and links the file *foo* (which is expected to be a byte code file defining the predicate *foo*), and continues with execution; if no such file can be found, an error message is given and execution fails. This feature makes it unnecessary for the user to have to explicitly link in all the predicates that might be necessary in a program: instead, only those files are loaded which are necessary to have the program execute. This can significantly reduce the memory requirements of programs.

The key to this dynamic search-and-load behaviour is the `SIMPATH` environment variable, which specifies the order in which directories are to be searched. It may be set by adding the following line to the user's `.cshrc` file:

```
setenv SIMPATH path
```

where *path* is a sequence of directory names separated by colons:

$$dir_1:dir_2: \dots : dir_n$$

and dir_i are *full path names* to the respective directories. For example, executing the command

```
setenv SIMPATH .:$HOME/prolog/modlib:$HOME/prolog/lib
```

sets the search order for undefined predicates to the following: first, the directory in which the program is executing is searched; if the appropriate file is not found in this directory, the directories searched are, in order, `~prolog/modlib` and `~prolog/lib`. If the appropriate file is not found in any of these directories, the system gives an error message and execution fails.

The beginning user is advised to include the system directories (listed in the next section) in his `SIMPATH`, in order to be able to access the system libraries (see below).

2.2 System Directories

There are four basic system directories: `cmplib`, `lib`, `modlib` and `sim`. `cmplib` contains the Prolog to byte code translator; `lib` and `modlib` contain library routines. The `src` subdirectory in each of these contains the corresponding Prolog source programs. The directory `sim` contains the simulator, the subdirectory *builtin* contains code for the builtin predicates of the system.

It is recommended that the beginning user include the system directories in his `SIMPATH`, by setting `SIMPATH` to

```
.:SBP/modlib:SBP/lib:SBP/cmplib
```

where *SBP* denotes the path to the root of the SB-Prolog system directories.

2.3 Invoking the Simulator

The simulator is invoked by the command

```
sbprolog bc_file where bc_file
```

is a byte code file resulting from the compilation of a Prolog program. In almost all cases, the user will wish to interact with the SB-Prolog *query evaluator*, in which case *bc_file* will be *\$readloop*, and the command will be

```
sbprolog Path/$readloop
```

where *Path* is the path to the directory containing the command interpreter *\$readloop*. This directory, typically, is `modlib` (see Section 2.2 above).

The command interpreter reads in a query typed in by the user, evaluates it and prints the answer(s), repeating this until it encounters an end-of-file (the standard end-of-file character on the system, e.g. ctrl-D), or the user types in *end_of_file* or *halt*.

The user should ensure that the the directory containing the executable file `sim` (typically, the system directory `sim`: see Section 2.2 above). is included in the shell variable *path*; if not, the full path to the simulator will have to be specified.

In general, the simulator may be invoked with a variety of options, as follows:

`sbprolog -options bc_file`

or

`sbprolog -option1 -option2 ... -optionn bc_file`

The options recognized by the simulator are described in Section 4.2.

When called with a byte code file *bc_file*, the simulator begins execution with the first clause in that file. The first clause in such a file, therefore, should be a clause without any arguments in the head (otherwise, the simulator will attempt to dereference argument pointers in the head that are really pointing into deep space, and usually come to a sad end). If the user is executing a file in this manner rather than using the command interpreter, he should also be careful to include the undefined predicate handler, consisting of the predicates ‘*_Sinterrupt/2*’ and ‘*_Sundefined_pred/1*’, which is normally defined in the files `modlib/src/$init_sys.P` and `modlib/src/$readloop`.

2.4 Executing Programs

There are two ways of executing a program: a source file may be compiled into a byte-code file, which can then be loaded and executed; or, the source file may be interpreted via *consult*. The system supports full integration of compiled and interpreted code, so that some predicates of a program may be compiled, while others may be interpreted. However, the unit of compilation or consulting remains the file. The remainder of this section describes each of these procedures in more detail.

2.4.1 Compiling Programs

The compiler is invoked through the Prolog predicate *compile*. It translates Prolog source programs into byte code that can then be executed on the simulator. The compiler may be invoked as follows:

```
| ?- compile(InFile [, OutFile ] [, OptionsList ]).
```

or

```
| ?- compile(InFile, OutFile, OptionsList, PredList).
```

where optional parameters are enclosed in brackets. *InFile* is the name of the input (i.e. source) file; *OutFile* is the name of the output file (i.e. byte code) file; *OptionsList* is a list of compiler options, and *PredList* is a list of

terms P/N denoting the predicates defined in *InFile*, where P is a predicate name and N its arity.

The input and output file names must be Prolog atoms, i.e. either begin with a lower case letter and consist only of letters, digits, dollar signs and underscores; or, be enclosed within single quotes. If the output file name is not specified, it defaults to *InFile.out*. The list of options, if specified, is a Prolog list, i.e. a term of the form

$$[option_1, option_2, \dots, option_n].$$

If left unspecified, it defaults to the empty list `[]`. *PredList*, if specified, is usually given as an uninstantiated variable; its principal use is for setting trace points on the predicates in the file (see Sections 6 and 8). Notice that *PredList* can only appear in *compile/4*.

A list of compiler options appears in Section 8.2.

2.4.2 Loading Byte Code Files

Byte code files may be loaded into the simulator using the predicate *load*:

$$| \text{?- load}(\textit{ByteCode_File}).$$

where *ByteCode_File* is a Prolog atom (see Section 3.1) that is the name of a byte code file.

The *load* predicate invokes the dynamic loader, which carries out a search according to the sequence specified by the environment variable `SIMPATH` (see Section 2.1). It is therefore not necessary to always specify the full path name to the file to be loaded.

Byte code files may be concatenated together to produce other byte code files. Thus, for example, if *foo1* and *foo2* are byte code files resulting from the compilation of two Prolog source programs, then the file *foo*, obtained by executing the shell command

$$\text{cat foo1 foo2 > foo}$$

is a byte code file as well, and may be loaded and executed. In this case, loading and executing the file *foo* would give the same result as loading *foo1* and *foo2* separately, which in turn would be the same as concatenating the original source files and compiling this larger file. This makes it easier to compile large programs: one need only break them into smaller pieces, compile the individual pieces, and concatenate the resulting byte code files together.

2.4.3 Consulting Programs

Instead of compiling a file to generate a byte code file which then has to be loaded, a program may be executed interpretively by “consulting” the corresponding source file:

```
| ?- consult(SourceFile [, OptionList ] ).
```

or

```
| ?- consult(SourceFile, OptionList, PredList).
```

where *SourceFile* is a Prolog atom which is the name of a file containing a Prolog source program; *OptionList* is a list of options to consult; and *PredList* is a list of terms *P/N*, where *P* is a predicate name and *N* its arity, specifying which predicates have been consulted from *SourceFile*; its principal use is for setting trace points on the predicates in the file (see Section 6). Notice that *PredList* can only appear in *consult/3*.

At this point, the options recognized for *consult* are the following:

- t** “trace”. Causes a trace point to be set on any predicate in the current file that does not already have a trace point set.
- v** “verbose”. Causes information regarding which predicates have been consulted to be printed out. Default: off.

In addition to the above, options for the macro expander are also recognized (see Section 10)).

consult will create an index on the principal functor of the first argument of the predicates being consulted, unless this is changed using the *index/3* directive. In particular, note that if no index is desired on a predicate *foo/n*, then the directive

```
:- index(foo, n, 0).
```

should be given.

It is important to note that SB-Prolog’s *consult* predicate is similar to that of Quintus Prolog, and behaves like C-Prolog’s *reconsult*. This means that if a predicate is defined across two or more files, consulting them will result in only the clauses in the file consulted last being used.

2.5 Execution Directives

Execution directives may be specified to *compile* and *consult* through `:-/1`. If, in the read phase of *compile* or *consult*, a term with principal functor `:-/1` is read in, this term is executed directly via *call/1*. This enables the user to dynamically modify the environment, e.g. via *op* declarations (see Section 3.2), *asserts* etc.

A point to note is that if the environment is modified as a result of an execution directive, the modifications are visible only in that environment. This means that consulted code, which runs in the environment in which the source program is read (and which is modified by such execution directives) feel the effects of such execution directives. However, byte code resulting from compilation, which, in general, executes in an environment different from that in which the source was compiled, does not inherit the effects of such directives. Thus, an *op* declaration can be used in a source file to change the syntax and allow the remainder of the program to be parsed according to the modified syntax; however, these modifications will not, in general, manifest themselves if the byte code is executed in another environment. Of course, if the byte code is loaded into the same environment as that in which the source program was compiled, e.g. through

```
| ?- compile(foo, bar), load(bar).
```

the effects of execution directives will continue to be felt.

3 Syntax

3.1 Terms

The syntax of SB-Prolog is by and large compatible with that of C-Prolog. The data objects of the language are called *terms*. A term is either a *constant*, a *variable* or a *compound term*. Constants can be *integers* or *atoms*. The symbol for an atom must begin with a lower case letter or the dollar sign \$, and consist of any number of letters, digits, underscores and dollar signs; if it contains any character other than these, it must be enclosed within single quotes.³ As in other programming languages, constants are definite elementary objects.

³Users are advised against using symbols beginning with '\$' or '_\$', however, in order to minimize the possibility of conflicts with symbols internal to the system.

Variables are distinguished by an initial capital letter or by the initial character “_” for example

X Value A A1 _3 _RESULT _result

If a variable is only referred to once, it does not need to be named and may be written as an *anonymous* variable, indicated by the underline character “_”.

A variable should be thought of as standing for some definite but unidentified object. A variable is not simply a writable storage location as in most programming languages; rather it is a local name for some data object, cf. the variable of pure LISP and constant declarations in Pascal.

The structured data objects of the language are the compound terms. A compound term comprises a *functor* (called the *principal* functor of the term) and a sequence of one or more terms called *arguments*. A functor is characterized by its *name*, which is an atom, and its *arity* or number of arguments. For example the compound term whose functor is named ‘point’ of arity 3, with arguments X, Y and Z, is written

point(X,Y,Z)

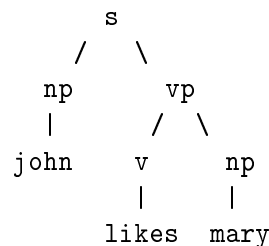
An atom is considered to be a functor of arity 0.

A functor or predicate symbol is uniquely identified by its name and arity (in other words, it is possible for different symbols having different arities to share the same name). A functor or predicate symbol *p* with arity *n* is usually written *p/n*.

One may think of a functor as a record type and the arguments of a compound term as the fields of a record. Compound terms are usefully pictured as trees. For example, the term

s(np(john),vp(v(likes),np(mary)))

would be pictured as the structure



Sometimes it is convenient to write certain functors as *operators* — 2-ary functors may be declared as *infix* operators and 1-ary functors as *prefix* or *postfix* operators. Thus it is possible to write

$$X+Y \ (P;Q) \ X<Y \ +X \ P;$$

as optional alternatives to

$$+(X,Y) \ ;(P,Q) \ <(X,Y) \ +(X) \ ;(P)$$

Operators are described fully in the next section.

Lists form an important class of data structures in Prolog. They are essentially the same as the lists of LISP: a list either is the atom `[]`, representing the empty list, or is a compound term with functor `'./2` and two arguments which are respectively the head and tail of the list. Thus a list of the first three natural numbers is the structure (shown in Figure 1) which could be written, using the standard syntax, as `.(1,.(2,.(3,[])))`, but

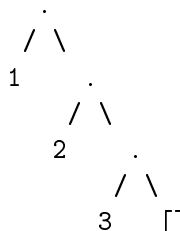


Figure 1: Structure for the Function `.(1,.(2,.(3,[])))`

which is normally written, in a special list notation, as `[1,2,3]`. The special list notation in the case when the tail of a list is a variable is exemplified by

$$[X|L] \qquad [a,b|L]$$

representing the structures shown in Figure 2 respectively.

Note that this list syntax is only syntactic sugar for terms of the form `'._(, _)` and does not provide any new facilities that were not available otherwise.

For convenience, a further notational variant is allowed for lists of integers which correspond to ASCII character codes. Lists written in this notation are called *strings*. For example, `"Prolog"` represents exactly the same list as `[80,114,111,108,111,103]`.

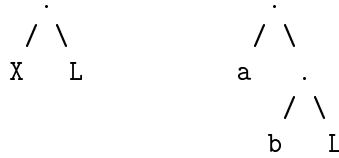


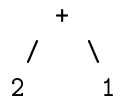
Figure 2: Structures for the Functions $[X|L]$ and $[a,b|L]$

3.2 Operators

Operators in Prolog are simply a notational convenience. For example, the expression

$$2 + 1$$

could also be written $+(2,1)$. It should be noticed that this expression represents the structure



and not the number 3. The addition would only be performed if the structure was passed as an argument to an appropriate procedure (such as **eval**/2 — see Section 5.2).

The Prolog syntax caters for operators of three main kinds — *infix*, *prefix* and *postfix*. An infix operator appears between its two arguments, while a prefix operator precedes its single argument and a postfix operator is written after its single argument.

Each operator has a *precedence*, which is a number from 1 to 1200. The precedence is used to disambiguate expressions where the structure of the term denoted is not made explicit through parenthesization. The general rule is that the operator with the *highest* precedence is the principal functor. Thus if ‘+’ has a higher precedence than ‘/’, then $a+b/c$ and $a+(b/c)$ are equivalent and denote the term $+(a,/(b,c))$. Note that the infix form of the term $/(+(a,b),c)$ must be written with explicit parentheses, $(a+b)/c$.

If there are two operators in the subexpression having the same highest precedence, the ambiguity must be resolved from the *types* of the operators. The possible types for an infix operator are

xfx xfy yfx

With an operator of type ‘*xfx*’, it is a requirement that both of the two subexpressions which are the arguments of the operator must be of *lower* precedence than the operator itself, i.e. their principal functors must be of lower precedence, unless the subexpression is explicitly bracketed (which gives it zero precedence). With an operator of type ‘*xfy*’, only the first or left-hand subexpression must be of lower precedence; the second can be of the *same* precedence as the main operator; and vice versa for an operator of type ‘*yfx*’.

For example, if the operators ‘+’ and ‘−’ both have type ‘*yfx*’ and are of the same precedence, then the expression “a−b+c” is valid, and means “(a−b)+c”, i.e. “+(-(a,b),c)”. Note that the expression would be invalid if the operators had type ‘*xfx*’, and would mean “a−(b+c)”, i.e. “-(a,+(b,c))”, if the types were both ‘*xfy*’.

The possible types for a prefix operator are

fx fy

and for a postfix operator they are

xf yf

The meaning of the types should be clear by analogy with those for infix operators. As an example, if ‘not’ were declared as a prefix operator of type ‘*fy*’, then

not not P

would be a permissible way to write not(not(P)). If the type were ‘*fx*’, the preceding expression would not be legal, although

not P

would still be a permissible form for not(P).

In SB-Prolog, a functor named *name* is declared as an operator of type *type* and precedence *precedence* by calling the evaluable predicate **op**:

| ?- op(*precedence*, *type*, *name*).

The argument *name* can also be a list of names of operators of the same type and precedence.

It is possible to have more than one operator of the same name, so long as they are of different kinds, i.e. infix, prefix or postfix. An operator of any kind may be redefined by a new declaration of the same kind. This applies

```

:- op( 1200, xfx, [ :-, -> ]).
:- op( 1200, fx, [ :- ]).
:- op( 1198, xfx, [ ::- ]).
:- op( 1150, fy, [ mode, public, dynamic ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ', ' ]). /* See note below */
:- op( 900, fy, [ not, \ +, spy, nospy ]).
:- op( 700, xfx, [ =, is, =.., ==, \ ==, @<, @>, @=<, @>=,
                 =:=, = \ =, <, >, =<, >=, ?=, \ = ]).

:- op( 661, xfy, [ '.' ]).
:- op( 500, yfx, [ +, -, /\, \/ ]).
:- op( 500, fx, [ +, - ]).
:- op( 400, yfx, [ *, /, //, <<, >> ]).
:- op( 300, xfx, [ mod ]).
:- op( 200, xfy, [ ^ ]).

```

Table 1: Operator Priorities

equally to operators which are provided as *standard* in SB-Prolog, namely the ones shown in Table 1.

Operator declarations are most usefully placed in directives at the top of your program files. In this case the directive should be a command as shown above. Another common method of organization is to have one file just containing commands to declare all the necessary operators. This file is then always consulted first.

Note that a comma written literally as a punctuation character can be used as though it were an infix operator of precedence 1000 and type ‘xfy’:

$$X, Y \text{ ', ' } (X, Y)$$

represent the same compound term. But note that a comma written as a quoted atom is *not* a standard operator.

Note also that the arguments of a compound term written in standard syntax must be expressions of precedence *below* 1000. Thus it is necessary to parenthesize the expression $P \text{ :- } Q$ in

$$\text{assert}((P \text{ :- } Q))$$

The following syntax restrictions serve to remove potential ambiguity associated with prefix operators.

- In a term written in standard syntax, the principal functor and its following (must *not* be separated by any whitespace. Thus

`point (X,Y,Z)`

is invalid syntax (unless `point` were declared as a prefix operator).

- If the argument of a prefix operator starts with a (, this (must be separated from the operator by at least one space or other non-printable character. Thus

`:- (p;q),r.`

(where `--` is the prefix operator) is invalid syntax, and must be written as

`:- (p;q),r.`

- If a prefix operator is written without an argument, as an ordinary atom, the atom is treated as an expression of the same precedence as the prefix operator, and must therefore be bracketed where necessary. Thus the brackets are necessary in

`X = (?-)`

3.3 Clause?

The syntax of a clause is as follows. *What the hell IS the syntax for a clause?*

3.4 Rule?

The syntax of a rule is as follows. *What the hell IS the syntax for a rule?*

3.5 Query?

The syntax of a query is as follows. *What the hell IS the syntax for a query?*

4 SB-Prolog: Operational Semantics

4.1 Standard Execution Behaviour

The normal execution behaviour of SB-Prolog follows the usual left to right order of literals within a clause, and the textual top to bottom order of clauses for a predicate. This corresponds to a depth first search of the leftmost SLD-tree for the program and the given query. Unification without occurs check is used, and execution backtracks to the most recent choice point when unification fails.

4.2 Cuts and If-Then-Else

This standard execution behaviour of SB-Prolog can be changed using constructs like *cut* (!) and *if-then-else* (->). In SB-Prolog, cuts are usually treated as *hard*, i.e. discard choice points of all the literals to the left of the cut in the clause containing the cut being executed, and also the choice point for the parent predicate, i.e. any remaining clauses for the predicate containing the cut being executed. There are some situations, however, where the scope of a cut is restricted to be smaller than this. Restrictions apply under the following conditions:

1. The cut occurs in a term which has been constructed at runtime and called through *call/1*, e.g. in

$$\dots, X = (p(Y), \text{!}, q(Y)), \dots, \text{call}(X), \dots$$

In this case, the scope of the cut is restricted to be within the *call*, unless one of the following cases also apply and serve to restrict its scope further.

2. The cut occurs in a negated goal, or within the scope of the test of an if-then-else (in an if-then-else of the form *Test -> TruePart; FalsePart*, the test is the goal *Test*). In these cases, the scope of the cut is restricted to be within the negation or the test of the if-then-else, respectively.

In cases involving nested occurrences of these situations, the scope of the cut is restricted to that for the deepest such nested construct, i.e. most restricted. For example, in the construct

..., not((p(X) -> not((q(X), (r(X) -> s(X) ; (t(X), !,
u(X))))))), ...

the scope of the cut is restricted to the inner negation, and does not affect any choice point that may have been set up for p(X).

4.3 Unification of Floating Point Numbers

As far as unification is concerned, no type distinction is made between integers and floating point numbers, and no explicit type conversion is necessary when unifying an integer with a float. However, due to the finite precision representation of floating point numbers and cumulative round-off errors in floating point arithmetic, comparisons involving floating point numbers may not always give the expected results. An effort is made to minimize surprises by considering two numbers x and y (at least one of which is a float) to be unifiable if $(\|x\| - \|y\|) / \min(\|x\|, \|y\|)$ to be less than 10^{-5} . However, this does not guarantee immunity against round-off errors. For the same reason, users are warned that indexing on predicate arguments (see Section 8.4.2) may not give the expected results if floating point numbers are involved.

5 Evaluable Predicates

This section describes (most of) the evaluable predicates provided by SB-Prolog. These can be divided into three classes: *inline* predicates, *builtin* predicates and *library* predicates.

Inline predicates represent “primitive” operations in the WAM. Calls to inline predicates are compiled into a sequence of WAM instructions in-line, i.e. without actually making a call to the predicate. Thus, for example, relational predicates ($>/2$, $\geq/2$, etc.) compile to, essentially, a subtraction and a conditional branch. Inline predicates cannot be redefined by the user. Table 2 lists the SB-Prolog inline predicates.

Unlike inline predicates, builtin predicates are implemented by C functions in the simulator, and accessed via the inline predicate ‘*_\$builtin*’/1. Thus, if a builtin predicate *foo*/3 was defined as builtin number 38, there would be a definition in the system of the form

`foo(X,Y,Z) :- '$_builtin'(38).`

In effect, a builtin is simply a segment of code in a large case (i.e. *switch*) statement. Each builtin is identified internally by an integer, referred to as

<code>arg/3</code>	<code>=/2</code>	<code>< /2</code>	<code>=< /2</code>
<code>>= /2</code>	<code>> /2</code>	<code>/\ /2</code>	<code>'\ '/2</code>
<code><</2</code>	<code>>>/2</code>	<code>:=/2</code>	<code>= \ = /2</code>
<code>is/2</code>	<code>?=/2</code>	<code>\ =</code>	<code>\ /1</code>
<code>'_\$builtin'/1</code>	<code>'_\$call'/1</code>	<code>nonvar/1</code>	<code>var/1</code>
<code>integer/1</code>	<code>real/1</code>	<code>halt/0</code>	<code>true/0</code>
<code>fail/0</code>			

Table 2: Inline Predicates of SB-Prolog

its “builtin number”, associated with it. The code for a builtin with builtin number k corresponds to the k^{th} case in the switch statement. SB-Prolog limits the total number of builtins to 256.

Builtins, unlike inline predicates, can be redefined by the user. For example, the predicate `foo/3` above can be redefined simply by compiling the new definition into a directory such that during dynamic loading, the new definition would be encountered first and loaded.

A list of the builtins currently provided is listed in Appendix A. Appendix C describes the procedure to be followed in order to define new builtin predicates.

Like builtin predicates, library predicates may also be redefined by the user. The essential difference between builtin and library predicates is that whereas the former are coded into the simulator in C, the latter are written in Prolog.

5.1 Input and Output

Input and output are done with respect to the current input and output streams. These can be set, reset or checked using the file handling predicates described below. The default input and output streams are denoted by **user**, and refer to the user’s terminal.

5.1.1 File Handling

see(F) F becomes the current input stream. F must be instantiated to an atom at the time of the call.

seeing(F) F is unified with the name of the current input file.

seen Closes the current input stream.

tell(*F*) *F* becomes the current output stream. *F* must be instantiated to an atom at the time of the call.

telling(*F*) *F* is unified with the name of the current output file.

told Closes the current output stream.

\$exists(*F*) Succeeds if file *F* exists.

5.1.2 Term I/O

read(*X*) The next term, delimited by a full stop (i.e. a . followed by a carriage-return or a space), is read from the current input stream and unified with *X*. The syntax of the term must accord with current operator declarations. If a call **read(*X*)** causes the end of the current input stream to be reached, *X* is unified with the atom 'end_of_file'. Further calls to **read** for the same stream will then cause an error failure.

write(*X*) The term *X* is written to the current output stream according to operator declarations in force.

display(*X*) The term *X* is displayed on the terminal.

writeq(*Term*) Similar to **write(*Term*)**, but the names of atoms and functors are quoted where necessary to make the result acceptable as input to **read**.

print(*Term*) Prints out the term *Term* onto the current output stream. This predicate provides a handle for user-defined pretty-printing. If *Term* is a variable then it is written using **write/1**; otherwise, if a user-defined predicate **portray/1** is defined, then a call is made to **portray(*Term*)**; otherwise, **print/1** is equivalent to **write/1**.

writename(*Term*) If *Term* is an uninstantiated variable, its name, which looks a lot like an address in memory, is written out; otherwise, the principal functor of *Term* is written out.

writeqname(*Term*) As for **writename**, but the names are quoted where necessary.

print_al(N , A) Prints A (which must be an atom or a number) left-aligned in a field of width N , with blanks padded to the right. If A 's print name is longer than the field width N , then A is printed but with no right padding.

print_ar(N , A) Prints A (which must be an atom or a number) right-aligned in a field of width N , with blanks padded to the left. If A 's print name is longer than the field width N , then A is printed but with no left padding.

portray_term($Term$) Writes out the term $Term$ on the current output stream. Variables are treated specially: an uninstantiated variable is printed out as Vn , where n is a number.

portray_clause($Term$) Writes out the term $Term$, interpreted as a clause, on the current output stream. Variables are treated as in *portray_term*/1.

5.1.3 Character I/O

nl A new line is started on the current output stream.

get0(N) N is the ASCII code of the next character from the current input stream. If the current input stream reaches its end of file, a -1 is returned (however, unlike in C-Prolog, the input stream is not closed on encountering end-of-file).

get(N) N is the ASCII code of the next non-blank printable character from the current input stream. It has the same behaviour as **get0** on end of file.

put(N) ASCII character code N is output to the current output stream. N must be an integer.

tab(N) N spaces are output to the current output stream. N must be an integer.

5.2 Arithmetic

Arithmetic is performed by evaluable predicates which take as arguments *arithmetic expressions* and *evaluate* them. An arithmetic expression is a term built from *evaluable functors*, numbers and variables. At the time of evaluation, each variable in an arithmetic expression must be bound to a

number or to an arithmetic expression. Each evaluable functor stands for an arithmetic operation.

The evaluable functors are as follows, where X and Y are arithmetic expressions.

$X + Y$ addition.

$X - Y$ subtraction.

$X * Y$ multiplication.

X / Y division.

$X // Y$ integer division.

$X \text{ (mod } Y)$ X (integer) modulo Y .

$-X$ unary minus.

$X \& Y$ integer bitwise conjunction.

$X \mid Y$ integer bitwise disjunction.

$X \ll Y$ integer bitwise left shift of X by Y places.

$X \gg Y$ integer bitwise right shift of X by Y places.

$\sim X$ integer bitwise negation.

As far as unification is concerned, no type distinction is made between integers and floating point numbers, and no explicit type conversion is necessary when unifying an integer with a float. However, due to the finite precision representation of floating point numbers and cumulative round-off errors in floating point arithmetic, comparisons involving floating point numbers may not always give the expected results. An effort is made to minimize surprises by considering two numbers x and y (at least one of which is a float) to be unifiable if $(\|x\| - \|y\|) / \min(\|x\|, \|y\|)$ to be less than 10^{-5} . The user should note, however, that this does not guarantee immunity against round-off errors.

The arithmetic evaluable predicates are as follows, where X and Y stand for arithmetic expressions, and Z for some term. Note that this means that **is** only evaluates one of its arguments as an arithmetic expression (the right-hand side one), whereas all the comparison predicates evaluate both their arguments.

Z is X Arithmetic expression *X* is evaluated and the result, is unified with *Z*. Fails if *X* is not an arithmetic expression. Unlike many other Prolog systems, variables in the expression *X* may be bound to other arithmetic expressions as well as to numbers.

eval(*E*, *X*) Evaluates the arithmetic expression *E* and unifies the result with the term *X*. Fails if *E* is not an arithmetic expression. (Thus, **eval**/2 is, except for the switched argument order, the same as **is**/2. It's around mainly for historical reasons.)

X==*Y* The values of *X* and *Y* are equal. If either *X* or *Y* involve compound subexpressions that are created at runtime, they should first be evaluated using **eval**/2.

X \ = *Y* The values of *X* and *Y* are not equal. If either *X* or *Y* involve compound subexpressions that are created at runtime, they should first be evaluated using **eval**/2.

X<*Y* The value of *X* is less than the value of *Y*. If either *X* or *Y* involve compound subexpressions that are created at runtime, they should first be evaluated using **eval**/2.

X>*Y* The value of *X* is greater than the value of *Y*. If either *X* or *Y* involve compound subexpressions that are created at runtime, they should first be evaluated using **eval**/2.

X =< *Y* The value of *X* is less than or equal to the value of *Y*. If either *X* or *Y* involve compound subexpressions that are created at runtime, they should first be evaluated using **eval**/2.

X >= *Y* The value of *X* is greater than or equal to the value of *Y*. If either *X* or *Y* involve compound subexpressions that are created at runtime, they should first be evaluated using **eval**/2.

floor(*X*, *Y*) If *X* is a floating point number in the call and *Y* is free, then *Y* is instantiated to the largest integer whose absolute value is not greater than the absolute value of *X*; if *X* is uninstantiated in the call and *Y* is an integer, then *X* is instantiated to the smallest float not less than *Y*.

floatc(*F*, *M*, *E*) If *F* is a number while *M* and *E* are uninstantiated in the call, then *M* is instantiated to a float *m* (of magnitude less than 1),

and E to an integer n , such that

$$F = m \times 2^n$$

If F is uninstantiated in the call while M is a float and E an integer, then F becomes instantiated to $M \times 2^E$.

exp(X, Y) If X is instantiated to a number and Y is uninstantiated in the call, then Y is instantiated to e^X (where $e = 2.71828\dots$); if X is uninstantiated in the call while Y is instantiated to a positive number, then X is instantiated to $\log_e(Y)$.

square(X, Y) If X is instantiated to a number while Y is uninstantiated in the call, then Y becomes instantiated to X^2 ; if X is uninstantiated in the call while Y is instantiated to a positive number, then X becomes instantiated to the positive square root of Y (if Y is negative in the call, X becomes instantiated to 0.0).

sin(X, Y) If X is instantiated to a number (representing an angle in radians) and Y is uninstantiated in the call, then Y becomes instantiated to $\sin(X)$ (the user should check the magnitude of X to make sure that the result is meaningful). If Y is instantiated to a number between $-\pi/2$ and $\pi/2$ and X is uninstantiated in the call, then X becomes instantiated to $\sin^{-1}(Y)$.

5.3 Convenience

P, Q P and then Q .

$P; Q$ P or Q .

true Always succeeds.

$X=Y$ Defined as if by the clause “ $Z=Z$ ”, i.e. X and Y are unified.

$X \setminus = Y$ Succeeds if X and Y are not unifiable, fails if X and Y are unifiable. It is thus equivalent to $\text{not}(X = Y)$, but is significantly more efficient.

$X? = Y$ Succeeds if X and Y are unifiable and fails if they are not, but does not instantiate any variables. Thus, it tests whether X and Y are unifiable. Equivalent to $\text{not}(\text{not}(X = Y))$, but is significantly more efficient.

5.4 Extra Control

! Cut (discard) all choice points made since the parent goal started execution. (The scope of cut in different contexts is discussed in Section 4.2).

not P If the goal P has a solution, fail, otherwise succeed. It is defined as if by

```
not(P) :- P, !, fail.  
not(_).
```

$P \rightarrow Q$; Analogous to if P then Q else R , i.e. defined as if by

```
P -> Q ; R :- P, !, Q.  
P -> Q ; R :- R.
```

$P \rightarrow Q$ When occurring other than as one of the alternatives of a disjunction, is equivalent to

```
P -> Q ; fail.
```

repeat Generates an infinite sequence of backtracking choices. It is defined by the clauses:

```
repeat.  
repeat :- repeat.
```

fail Always fails.

5.5 Meta-Logical

var(X) Tests whether X is currently instantiated to a variable.

nonvar(X) Tests whether X is currently instantiated to a non-variable term.

atom(X) Checks that X is currently instantiated to an atom (i.e. a non-variable term of arity 0, other than a number).

integer(X) Checks that X is currently instantiated to an integer.

real(X) Checks that X is currently instantiated to a floating point number.

float(X) Same as **real**/1, checks that X is currently instantiated to a floating point number.

number(X) Checks that X is currently instantiated to a number, i.e. that it is either an integer or a real.

atomic(X) Checks that X is currently instantiated to an atom or number.

structure(X) Checks that X is currently instantiated to a compound term, i.e. to a nonvariable term that is not atomic.

is_buffer(X) Succeeds if X is instantiated to a buffer.

functor(T, F, N) The principal functor of term T has name F and arity N , where F is either an atom or, provided N is 0, a number. Initially, either T must be instantiated to a non-variable, or F and N must be instantiated to, respectively, either an atom and a non-negative integer or an integer and 0. If these conditions are not satisfied, an error message is given. In the case where T is initially instantiated to a variable, the result of the call is to instantiate T to the most general term having the principal functor indicated.

arg(I, T, X) Initially, I must be instantiated to a positive integer and T to a compound term. The result of the call is to unify X with the I th argument of term T . The arguments are numbered from 1 upwards.) If the initial conditions are not satisfied or I is out of range, the call merely fails.

$X = ..Y$ Y is a list whose head is the atom corresponding to the principal functor of X and whose tail is the argument list of that functor in X . E.g.

```
product(0,N,N-1) =.. [product,0,N,N-1]
```

```
N-1 =.. [-,N,1]
```

```
product =.. [product]
```

If X is instantiated to a variable, then Y must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is a number.

name(X, L) If X is an atom or a number then L is a list of the ASCII codes of the characters comprising the name of X . E.g.

```
name(product, [112, 114, 111, 100, 117, 99, 116])
```

i.e. `name(product, "product")`.

If X is instantiated to a variable, L must be instantiated to a list of ASCII character codes. E.g.

```
| ?- name(X, [104, 101, 108, 108, 111]).
X = hello
```

```
| ?- name(X, "hello").
X = hello
```

call(X) If X is a nonvariable term in the program text, then it is executed exactly as if X appeared in the program text instead of `call(X)`, e.g.

```
..., p(a), call( (q(X), r(Y)) ), s(X), ...
```

is equivalent to

```
..., p(a), q(X), r(Y), s(X), ...
```

However, if X is a variable in the program text, then if at runtime X is instantiated to a term which would be acceptable as the body of a clause, the goal **call**(X) is executed as if that term appeared textually in place of the **call**(X), *except that* any cut (!) occurring in X will remove only those choice points in X . If X is not instantiated as described above, an error message is printed and **call** fails.

X (where X is a variable) Exactly the same as **call**(X). However, we prefer the explicit usage of `call/1` as good programming practice, and the use of a top level variable subgoal elicits a warning from the compiler.

conlength(C, L) Succeeds if the length of the print name of the constant C (which can be an atom, buffer or integer), in bytes, is L . If C is a buffer (see Section 5.8), it is the length of the buffer; if C is an integer, it is the length of the decimal representation of that integer, i.e., the number of bytes that a *\$writename* will use.

5.6 Sets

When there are many solutions to a problem, and when all those solutions are required to be collected together, this can be achieved by repeatedly backtracking and gradually building up a list of the solutions. The following evaluable predicates are provided to automate this process.

setof(X, P, S) Read this as S is the set of all instances of X such that P is provable". If P is not provable, **setof**(X, P, S) succeeds with S instantiated to the empty list $[]$. The term P specifies a goal or goals as in **call**(P). S is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 5.7). If there are uninstantiated variables in P which do not also appear in X , then a call to this evaluable predicate may backtrack, generating alternative values for S corresponding to different instantiations of the free variables of P . Variables occurring in P will not be treated as free if they are explicitly bound within P by an existential quantifier. An existential quantification is written:

$$Y \wedge Q$$

meaning there exists a Y such that Q is true, where Y is some Prolog term (usually, a variable, or tuple or list of variables).

bagof(X, P, \textit{Bag}) This is the same as **setof** except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. If P is unsatisfiable, *bagof* succeeds binding *Bag* to the empty list. The effect of this relaxation is to save considerable time and space in execution.

findall(X, P, L) Similar to *bagof*/3, except that variables in P that do not occur in X are treated as local, and alternative lists are not returned for different bindings of such variables. The list L is, in general, unordered, and may contain duplicates. If P is unsatisfiable, *findall* succeeds binding S to the empty list.

$X \wedge P$ The system recognises this as meaning there exists an X such that P is true, and treats it as equivalent to **call**(P). The use of this explicit existential quantifier outside the **setof** and **bagof** constructs is superfluous.

5.7 Comparison of Terms

These evaluable predicates are meta-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should *not* be used when what you really want is arithmetic comparison (Section 5.2) or unification. The predicates make reference to a standard total ordering of terms, which is as follows:

- variables, in a standard order (roughly, oldest first — the order is *not* related to the names of variables);
- numbers, from $-\infty$ to $+\infty$;
- atoms, in alphabetical (i.e. ASCII) order;
- complex terms, ordered first by arity, then by the name of principal functor, then by the arguments (in left-to-right order).

For example, here is a list of terms in the standard order:

```
[ X, -9, 1, fie, foe, fum, X = Y, fie(0,2), fie(1,1) ]
```

The basic predicates for comparison of arbitrary terms are:

$X == Y$ Tests if the terms currently instantiating X and Y are literally identical (in particular, variables in equivalent positions in the two terms must be identical). For example, the question

```
| ?- X == Y.
```

fails (answers no) because X and Y are distinct uninstantiated variables. However, the question

```
| ?- X = Y, X == Y.
```

succeeds because the first goal unifies the two variables (see page ?).

$X \setminus == Y$ Tests if the terms currently instantiating X and Y are not literally identical.

$T1 @< T2$ Term $T1$ is before term $T2$ in the standard order.

$T1 @> T2$ Term $T1$ is after term $T2$ in the standard order.

$T1 @=< T2$ Term $T1$ is not after term $T2$ in the standard order.

$T1 @>= T2$ Term $T1$ is not before term $T2$ in the standard order.

Some further predicates involving comparison of terms are:

compare($Op, T1, T2$) The result of comparing terms $T1$ and $T2$ is Op , where the possible values for Op are:

‘=’ if $T1$ is identical to $T2$,

‘<’ if $T1$ is before $T2$ in the standard order,

‘>’ if $T1$ is after $T2$ in the standard order.

Thus **compare(=, $T1, T2$)** is equivalent to $T1 == T2$.

sort($L1, L2$) The elements of the list $L1$ are sorted into the standard order, and any identical (i.e. ‘=’) elements are merged, yielding the list $L2$.

keysort($L1, L2$) The list $L1$ must consist of items of the form *Key-Value*. These items are sorted into order according to the value of *Key*, yielding the list $L2$. No merging takes place.

5.8 Buffers

SB-Prolog supports the concept of buffers. A buffer is actually a constant and the characters that make up the buffer is the name of the constant. However, the symbol table entry for a buffer is not hashed and thus is not added to the obj-list, so two different buffers will never unify. Buffers can be allocated either in permanent space or on the heap. Buffers in permanent space stay there forever; buffers on the heap are deallocated when the “allocate buffer” goal is backtracked over.

A buffer allocated on the heap can either be a simple buffer, or it can be allocated as a subbuffer of another buffer already on the heap. A subbuffer will be deallocated when its superbuffer is deallocated.

There are occasions when it is not known, in advance, exactly how much space will be required and so how big a buffer should be allocated. Sometimes this problem can be overcome by allocating a large buffer and then, after using as much as is needed, returning the rest of the buffer to the system. This can be done, but only under *very* limited circumstances: a buffer is allocated from the end of the permanent space, the top of the heap, or from the next available space in the superbuffer; if no more space has been used

beyond the end of the buffer, a tail portion of the buffer can be returned to the system. This operation is called “trimming” the buffer.

The following is a list of library predicates for buffer management:

alloc_perm(*Size*, *Buff*) Allocates a buffer with a length *Size* in the permanent (i.e. program) area. *Size* must be bound to a number. On successful return, *Buff* will be bound to the allocated buffer. The buffer, being in the permanent area, is never de-allocated.

alloc_heap(*Size*, *Buff*) Allocates a buffer of size *Size* on the heap and binds *Buff* to it. Since it is on the heap, it will be deallocated on backtracking.

trimbuff(*Type*, *Buff*, *Newlen*) This allows (in some very restricted circumstances) the changing of the size of a buffer. *Type* is 0 if the buffer is permanent, 1 if the buffer is on the heap. *Buff* is the buffer. *Newlen* is an integer: the size (which should be smaller than the original length of the buffer) to make the buffer. If the buffer is at the top of the heap (if heap buffer) or the end of the program area (if permanent) then the heap-top (or program area top) will be readjusted down. The length of the buffer will be modified to *Newlen*. This is (obviously) a very low-level primitive and is for hackers only to implement grungy stuff.

conlength(*Constant*, *Length*) Succeeds if the length of the print name of the constant *Constant* (which can be an atom, buffer or integer), in bytes, is *Length*. If *Constant* is a buffer, it is the length of the buffer; if *Constant* is an integer, it is the length of the decimal representation of that integer, i.e., the number of bytes that a *\$writename* will use.

5.9 Modification of the Program

The predicates defined in this section allow modification of the program as it is actually running. Clauses can be added to the program (*asserted*) or removed from the program (*retracted*). At the lowest level, the system supports the asserting of clauses with upto one literal in the body. It does this by allocating a buffer and compiling code for the clause into that buffer. Such a buffer is called a “clause reference” (*clref*). The clref is then added to a chain of clrefs. The chain of clrefs has a header, which is a small buffer called a “predicate reference” (*prref*), which contains pointers to the beginning and end of its chain of clrefs. Clause references are quite similar to “database references” of C-Prolog, and can be called.

When clauses are added to the program through *assert*, an index is normally created on the principal functor of the first argument in the head of the clause. The argument on which the index is being created may be changed via the *index/3* directive. In particular, if no index is desired on a predicate, this should be specified using the *index/3* directive with the argument number set to zero, e.g. if no index is desired on a predicate *foo/3*, then the directive

```
:- index(foo, 3, 0).
```

should be specified.

The predicates that can be used to modify the program are the following:

assert(*C*) The current instance of *C* is interpreted as a clause and is added to the program (with new private variables replacing any uninstantiated variables), at the end of the list of clauses for that predicate. *C* must be instantiated to a non-variable.

assert(*C*, *Ref*) As for *assert/1*, but also unifies *Ref* with the clause reference of the clause asserted.

asserti(*C*, *N*) The current instance of *C*, interpreted as a clause, is asserted to the program with an index on its *Nth* argument. If *N* is zero, no index is created.

asserta(*C*) Similar to **assert(*C*)**, except that the new clause becomes the *first* clause of the procedure concerned.

asserta(*C*, *Ref*) Similar to **asserta(*C*)**, but also unifies *Ref* with the clause reference of the clause asserted.

assertz(*C*) Similar to **assert(*C*)**, except that the new clause becomes the *last* clause of the procedure concerned.

assertz(*C*, *Ref*) Similar to **assertz(*C*)**, but also unifies *Ref* with the clause reference of the clause asserted.

assert_union(*P*, *Q*) The clauses for *Q* are added to the clauses for *P*. For example, the call

```
| ?- assert_union(p(X,Y),q(X,Y)).
```

has the effect of adding the rule

$p(X,Y) \text{ :- } q(X,Y).$

as the last rule defining $p/2$. If P is not defined, it results in the call to Q being the only clause for P .

The variables in the arguments to *assert_union/2* are not significant, e.g. the above would have been equivalent to

$| \text{ ?- } \text{assert_union}(p(Y,X),q(X,Y)).$

or

$| \text{ ?- } \text{assert_union}(p(-,-),q(-,-)).$

However, the arities of the two predicates involved must match, e.g. even though the goal

$| \text{ ?- } \text{assert_union}(p(X,Y), r(X,Y,Z)).$

will succeed, the predicate $p/2$ will not in any way depend on the clauses for $r/3$.

assert(*Clause*, *AZ*, *Index*, *Clref*) Asserts a clause to a predicate. *Clause* is the clause to assert. *AZ* is 0 for insertion as the first clause, 1 for insertion as the last clause. *Index* is the number of the argument on which to index (0 for no indexing). *Clref* is returned as the clause reference of the fact newly asserted. If the main functor symbol of *Clause* has been declared (by *\$assertf_alloc_t/2*, see below) to have its clauses on the heap, the clref will be allocated there. If the predicate symbol of *Clause* is undefined, it will be initialized and *Clause* added. If the predicate symbol has compiled clauses, it is first converted to be dynamic (see *symtype/2*, Section 5.10) by adding a special clref that calls the compiled clauses. *Fact*, *AZ* and *Index* are input arguments, and should be instantiated at the time of call; *Clref* is an output argument, and should be uninstantiated at the time of call.

clause(*P*,*Q*) *P* must be bound to a non-variable term, and the program is searched for a clause *Cl* whose head matches *P*. The head and body of the clause *Cl* is unified with *P* and *Q*, respectively. If *Cl* is a unit clause, *Q* will be unified with ‘true’. Only interpreted clauses, i.e. those created through *assert*, can be accessed via *clause/2*.

clause(*Head, Body, Ref*) Similar to **clause(*Head,Body*)** but also unifies *Ref* with the database reference of the clause concerned. *clause/3* can be executed in one of two modes: either *Head* must be instantiated to a non-variable term at the time of the call, or *Ref* must be instantiated to a database reference. As in the case of *clause/2*, only interpreted clauses, i.e. those created through *assert*, can be accessed via *clause/3*.

retract(*Clause*) The first clause in the program that unifies with *Clause* is deleted from the program. This predicate may be used in a non-deterministic fashion, i.e. it will successively backtrack to retract clauses whose heads match *Head*. *Head* must be initially instantiated to a non-variable. In the current implementation, *retract* works only for asserted (e.g. consulted) clauses.

abolish(*P*) Completely remove all clauses for the procedure with head *P* (which should be a term). For example, the goal

```
| ?- abolish( p(., ., .) ).
```

removes all clauses for the predicate *p/3*.

abolish(*P, N*) Completely remove all clauses for the predicate *P* (which should be an atom) with arity *N* (which should be an integer).

5.10 Internal Database

recorded(*Key, Term, Ref*) The internal database is searched for terms recorded under the key *Key*. These terms are successively unified with *Term* in the order they occur in the database; at the same time, *Ref* is unified with the database reference of the recorded item. The key must be given, and may be an atom or complex term. If it is a complex term, only the principal functor is significant.

recorda(*Key, Term, Ref*) The term *Term* is recorded in the internal database as the first item for the key *Key*, where *Ref* is its database reference. The key must be given, and only its principal functor is significant.

recordz(*Key, ITerm, Ref*) The term *Term* is recorded in the internal database as the last item for the key *Key*, where *Ref* is its database reference. The key must be given, and only its principal functor is significant.

erase(*Clref*) The recorded item or clause whose database reference is *Clref* is deleted from the internal database or program. *Clref* should be instantiated at the time of call.

instance(*Ref*, *Term*) A (most general) instance of the recorded term whose database reference is *Ref* is unified with *Term*. *Ref* must be instantiated to a database reference. Note that *instance/2* will not be able to access terms that have been erased.

5.11 Information about the State of the Program

listing Lists in the current output stream the clauses for all the interpreted predicates in the program, except predicates that are “internal”, i.e. whose names begin with ‘\$’ or ‘_\$', or which are provided as predefined (builtin or library) predicates. A bug in the current system is that even though the user is allowed to redefine such predicates, *listing/0* does not know about such redefinitions, and will not list such predicates (they may, however, be accessed through *listing/1* if they are interpreted).

listing(*A*) The argument *A* may be a predicate specification of the form *Name/Arity* in which case only the clauses for the specified predicate are listed. Alternatively, it is possible for *A* to be a list of predicate specifications, e.g.

```
| ?- listing([concatenate/3, reverse/2, go/0]).
```

Only *interpreted* clauses, i.e. clauses created via *assert*, can be accessed through *listing/1*.

current_atom(*Atom*) Generates (through backtracking) all currently known atoms, and unifies each in turn with *Atom*. However, atoms considered “internal” symbols, i.e. those whose names begin with \$ or _\$ are not returned. The intrepid user who wishes to access such internal atoms as well can use the goal

```
?- $current_atom(Atom, 1).
```

current_functor(*Name*, *Term*) Generates (through backtracking) all currently known functors (which includes function and predicate symbols), and for each one returns its name and most general term as *Name*

and *Term* respectively. However, functors considered “internal” symbols, i.e. those whose names begin with **\$** or **_\$**, or which are provided as predefined predicates, are not returned if both arguments to *current_functor/2* are variables. Internal symbols (of which there are a *great* many) as well as external ones may be accessed via

```
?- $current_functor(Name, Term, 1).
```

A bug in the current implementation is that even though the user is allowed to redefine “internal” (builtin or library) predicates, *current_functor/2* does not know whether they have been redefined, and hence will not return such predicates if both arguments to *current_functor/2* are variables.

current_predicate(*Name*, *Term*) Generates (through backtracking) all currently known predicates, and for each one returns its name and most general term as *Name* and *Term* respectively. However, predicates considered “internal”, i.e. those whose names begin with **\$** or **_\$**, or which are provided as predefined predicates, are not returned if both arguments to *current_predicate/2* are variables. Internal symbols (of which there are a *great* many) as well as external ones may be accessed via

```
?- $current_predicate(Name, Term, 1).
```

A bug in the current implementation is that even though the user is allowed to redefine “internal” (builtin or library) predicates, *current_predicate/2* does not know whether they have been redefined, and hence will not return such predicates if both arguments to *current_predicate/2* are variables.

predicate_property(*Term*, *Property*) If *Term* is a term whose principal functor is a predicate, *Property* is unified with the currently known properties of the corresponding predicate. If *Term* is a variable, then it is unified (successively, through backtracking) with the most general term for a predicate whose known properties are unified with *Property*. For example, all the interpreted predicates in the program may be enumerated using

```
?- predicate_property(X, interpreted).
```

If the first argument to *predicate_property/2* is uninstantiated at the time of the call, “internal” predicates will not be returned. A bug in the current implementation is that even though the user is allowed to redefine such “internal” predicates, *predicate_property/2* does not know about such redefinitions, and will not return such predicates if its first argument is uninstantiated. Currently, the only properties that are considered are **interpreted** and **compiled**.

5.12 Environmental

op(priority, type, name) Treat *name* as an operator of the stated *type* and *priority* (see Section 3.2). *name* may also be a list of names, in which all are to be treated as operators of the stated *type* and *priority*.

break Causes the current execution to be suspended at the next procedure call. Then the message [**Break** (level 1)] is displayed. The interpreter is then ready to accept input as though it was at the top level (except that at break level $n > 0$, the prompt is $n: ?-$). If another call of **break** is encountered, it moves up to level 2, and so on. To close the break and resume the execution which was suspended, type the END-OF-INPUT character. Execution will be resumed at the procedure call where it had been suspended. Alternatively, the suspended execution can be aborted by calling the evaluable predicate **abort**, which causes a return to the top level.

abort Aborts the current execution, taking you back to top level.

save(F) The system saves the current state of the system into file *F*.

restore(F) The system restores the saved state in file *F* to be the current state. One restriction imposed by the current system is that various system parameters (e.g. stack sizes, permanent space, heap space, etc.) of the saved state have to be the same as that of the current invocation. Thus, it is not possible to save a state from an invocation where 50000 words of permanent space had been allocated, and then restore the same state in an invocation with 100000 words of permanent space.

cputime(X) Unifies *X* with the time elapsed, in milliseconds, since the system was started up.

\$getenv(Var, Val) *Val* is unified with the value of the Unix environment variable *Var*. Fails if *Var* is undefined.

statistics Prints out the current allocations and amounts of space used for each of the four main areas: the permanent area, the local stack, the global stack and the trail stack. Does not work well unless the simulator has been called with the `-s` option (see Section 7.2).

statistics(*Keyword*, *List*) Usually used with *Keyword* instantiated to a keyword, e.g. ‘runtime’, and *List* unbound. It unifies *List* with a list of statistics determined by *Keyword*. The keys and values are summarized in Table 5.12. Times are given in milliseconds and sizes are given in bytes.

<i>Keyword</i>	List
runtime	[cpu time used by Prolog, cpu time since last call to statistics/2]
memory	[total virtual memory, 0]
core	(<i>same as for the keyword</i> memory)
program	[program space in use, program space free]
heap	(<i>same as for the keyword</i> program)
global_stack	[global stack in use, global stack free]
local_stack	[local stack in use, local stack free]
trail	[trail stack in use, trail stack free]
garbage_collection	[0, 0]
stack_shifts	[0, 0]

Table 3: Run Time Statistics Predicates

Note:

1. For the keyword ‘memory’ the second element of the returned list is always 0.
2. For the keyword ‘trail’, the second element of the returned list is the amount of trail stack free. This is similar to Sicstus Prolog (version 0.5), but different from Quintus Prolog (version 1.6).
3. Currently, SB-Prolog does not have garbage collection or stack shifting, hence the list values returned for these are [0, 0].

nodynload(*P*, *N*) Flags the predicate *P* with arity *N* as one that should not be attempted to be dynamically loaded if it is undefined. If a predicate so flagged is undefined when a call to it is encountered, the call fails quietly without trying to invoke the dynamic loader or giving

an error message. P and N should be instantiated to an atom and an integer, respectively, at the time of call to *nodynload/2*.

symtype(T, N) Unifies N with the “internal type” of the principal functor of the term T , which must be instantiated at the time of the call. N is bound to 0 if T does not have an entry point defined (i.e. cannot be executed); to 1 if the principal functor of T is “dynamic”, i.e. has asserted code; to 2 if the principal functor for T is a compiled predicate; and 3 if T denotes a buffer. Thus, for example, if the predicate $p/2$ is a compiled predicate which has been loaded into the system, the goal

```
| ?- symtype(p(_,_), X).
```

will succeed binding X to 2; on the other hand, the goal

```
| ?- assert(q(a,b,c)), symtype(q(_,_,_), X).
```

will succeed binding X to 1.

system($Call$) Calls the operating system with the atom $Call$ as argument. For example, the call

```
| ?- system('ls').
```

will produce a directory listing. Since *system/1* is executed by forking off a shell process, it cannot be used, for example, to change the working directory of the simulator.

syscall($N, Args, Res$) Executes the Unix system call number N with arguments $Args$, and returns the result in Res . N is an integer, and $Args$ a Prolog list of the arguments to the system call. For example, to execute the system call **creat**($File, Mode$), knowing that the syscall number for the Unix command *creat*(2) is 8, we execute the goal

```
| ?- syscall(8, [File, Mode], Des).
```

where Des is the file descriptor returned by *creat*. The syscall numbers for some Unix system calls are given in Table 4.

exit	1	fork	2
read	3	write	4
open	5	close	6
creat	8	link	9
unlink	10	chdir	12
chmod	15	lseek	19
access	33	kill	37
wait	84	socket	97
connect	98	accept	99
send	101	recv	102
bind	104	setsockopt	105
listen	106	recvmsg	113
sendmsg	114	getsockopt	118
recvfrom	125	sendto	133
socketpair	135	mkdir	136
rmdir	137	getsockname	150

Table 4: Syscall Numbers for Some Unix Systems Calls

5.13 Global Values

SB-Prolog has some primitives that permit the programmer to manipulate global values. These are provided primarily as an efficiency hack, and needless to say, should be used with a great deal of care.

globalset(*Term*) Allows the user to save a global value. *Term* must be bound to a compound term, say $p(V)$. V must be a number or a constant or a variable. If V is a number or a constant, the effect of *globalset*($p(V)$) can be described as:

`retract(p(_)), assert(p(V)).`

I.e., p is a predicate that when called will, from now on (until some other change by *globalset*/1), deterministically return V . If V is a variable, the effect is to make V a global variable whose value is accessible by calling p . For example, executing *globalset*($p(X)$) makes X a global variable. X can be set by unification with some other term. On backtracking, X will be restored to its earlier value.

gennum(*Newnum*) *gennum*/1 sets its argument to a new integer every time it is invoked.

gensym(*C*, *Newsym*) gensym/2 sets its second argument to an atom whose name is made by concatenating the name of the atom *C* to the current gennum number. This new constant is bound to *Newsym*. For example, if the current *gennum* number is 37, then the call

```
| ?- gensym(aaa,X)
```

will succeed binding *X* to the atom ‘aaa37’.

5.14 Exotica

This section describes some low-level routines that are sometimes useful in mucking around with buffers. These are for serious hackers only.

\$alloc_buff(*Size*,*Buff*,*Type*,*Supbuff*,*Retcode*) Allocates a buffer. *Size* is the length (in bytes) of the buffer to allocate; *Buff* is the buffer allocated, and should be unbound at the time of the call; *Type* indicates where to allocate the buffer: a value of 0 indicates that the buffer is to be allocated in permanent space, 1 that it should be on the heap, and 2 indicates that it should be allocated from a larger heap buffer; *Supbuff* is the larger buffer to allocate a subbuffer out of, and is only looked at if the value of *Type* is 2; *Retcode* is the return code: a value of 0 indicates that the buffer has been allocated, while a value of 1 indicates that the buffer could not be allocated due to lack of space. The arguments *Size*, *Type*, and *Supbuff* (if *Type* = 2) are input arguments, and should be bound at the time of the call; *Buff* and *Retcode* are output arguments, and should be unbound at the time of the call.

call_ref(*Call*, *Ref*) Calls the predicate whose database reference (prref) is *Ref*, using the literal *Call* as the call. This is similar to **call_ref(*Call*, *Ref*, 0)**.

call_ref(*Call*, *Ref*, *Tr*) Calls the predicate whose database reference (prref) is *Ref*, using the literal *Call* as the call. *Tr* must be either 0 or 1: if *Tr* is 0 then the call *Call* is made assuming the “trust” optimization will be made; if *Tr* is 1 then the “trust” optimization is not used, so that any new fact added before final failure will be seen by *Call*. (Also, this currently does not take advantage of any indexing that might have been constructed.) *Call*, *Ref* and *Tr* are all input arguments, and should be instantiated at the time of call.

\$assertf_alloc_t(*Palist*, *Size*) Declares that each predicate in the list *Palist* of predicate/arity pairs (terms of the form $'/(P,N)$ where P is a predicate symbol and N the arity of P) is to have any facts asserted to them stored in a buffer on the heap, to be allocated here. This allocates a superbuffers of size *Size* on the heap. Future assertions to these predicates will have their clauses put in this buffer. When this call is backtracked over, any clauses asserted to these predicates are deallocated, and a subsequent call to any of those predicates will cause the simulator to report an error and fail. Both *Palist* and *Size* are input arguments, and should be instantiated at the time of call.

\$db_new_prref(*Prref*, *Where*, *Supbuff*) Creates an empty *Prref*, i.e. one with no facts in it. If called, it will simply fail. *Where* indicates where the *prref* should be allocated: a value of 0 indicates the permanent area, while a value of 2 indicates that it is to be allocated as a subbuffer. *Supbuff* is the superbuffers from which to allocate *Prref* if *Where* is 2. *Where* should be instantiated at the time of call, while *Prref* should be uninstantiated; in addition, if *Where* is 2, *Supbuff* should be instantiated at the time of call.

\$db_assert_fact(*Fact*, *Prref*, *AZ*, *Index*, *Clref*, *Where*, *Supbuff*) *Fact* is a fact to be asserted; *Prref* is a predicate reference to which to add the asserted fact; *AZ* is either 0, indicating the fact should be inserted as the first clause in *Prref*, or 1, indicating it should be inserted as the last; *Index* is 0 if no index is to be built, or n if an index on the n^{th} argument of the fact is to be used. (Asserting at the beginning of the chain with indexing is not yet supported.) *Where* indicates where the *clref* is to be allocated: a value of 0 indicates that it should be in the permanent area, while a value of 2 indicates that it should be allocated as a subbuffer of *Supbuff*. *Clref* is returned and it is the clause reference of the asserted fact. *Fact*, *Prref*, *AZ*, *Index*, and *Where* are input arguments, and should be instantiated at the time of call; in addition, if *Where* is 2, then *Supbuff* should also be instantiated. *Clref* is an output argument, and should be uninstantiated at the time of call.

\$db_add_clref(*Fact*, *Prref*, *AZ*, *Index*, *Clref*, *Where*, *Supbuff*) Adds the *clref* *Clref* to the *prref* *Prref*. *Fact* is the fact that has been compiled into *Clref* (used only to get the arity and for indexing). The other parameters are as for *\$db_assert_fact/7*.

\$db_call_prref(*Call*,*Prref*) Calls the prref *Prref* using the literal *Call* as the call. The call is done by simply branching to the first clause. New facts added to *Prref* after the last fact has been retrieved by *Call*, but before *Call* is failed through, will *not* be used. Both *Call* and *Prref* are input arguments, and should be instantiated at the time of call.

\$db_call_prref_s(*Call*,*Prref*) This also calls the prref *Prref* using *Call* as the call. The difference from *\$db_call_prref* is that this does not use the “trust” optimization, so that any new fact added before final failure will be seen by *Call*. (Also, this currently does not take advantage of any indexing that might have been constructed, while *\$db_call_prref* does.) Both *Call* and *Prref* are input arguments, and should be instantiated at the time of call.

\$db_get_clauses(*Prref*,*Clref*,*Dir*) This returns, nondeterministically, all the clause references *Clref* for clauses asserted to prref *Prref*. If *Dir* is 0, then the first clref on the list is returned first; if *Dir* is 1, then they are returned in reverse order. *Prref* and *Dir* are input arguments, and should be instantiated at the time of call; *Clref* is an output argument, and should be uninstantiated at the time of call.

6 Debugging

6.1 High-Level Tracing

The preferred method of tracing execution is through the predicate *trace/1*. This predicate takes as argument a term *P/N*, where *P* is a predicate name and *N* its arity, and sets a “trace point” on the corresponding predicate; it can also be given a list of such terms, in which case a trace point is set on each member of the list. For example, executing

```
| ?- trace(pred1/2), trace([pred2/3, pred3/2]).
```

sets trace points on predicates *pred1/2*, *pred2/3* and *pred3/2*. Only those predicates are traced that have trace points set on them.

If all the predicates in a file are to be traced, it is usually convenient to use the *PredList* parameter of *compile/4* or *consult/3*, e.g.:

```
| ?- compile(foo, 'foo.out', [t,v], Preds), load('foo.out'),
      trace(Preds).
```

or

```
| ?- consult(foo, [v], Preds), trace(Preds).
```

Notice that in the first case, the `t` compiler option (see Section 8.2) should be specified in order to turn off certain assembler optimizations and facilitate tracing. In the second case, the same effect may be achieved by specifying the `t` option to *consult*.

The trace points set on predicates may be overwritten by loading byte code files via *load/1*, and in this case it may be necessary to explicitly set trace points again on the loaded predicates. This does not happen with *consult*: predicates that were being traced continue to have trace points set after consulting.

The tracing facilities of SB-Prolog are in many ways very similar to those of C-Prolog. However, leashing is not supported, and only those predicates can be traced which have had trace points set on them through *trace/1*. This makes *trace/1* and *spy/1* very similar: essentially, trace amounts to two levels of spy points. In SB-Prolog, tracing occurs at *Call* (i.e. entry to a predicate), successful *Exit* from a clause, and *Failure* of the entire call. The tracing options available during debugging are the following:

- c, newline: Creep** Causes the system to single-step to the next port (i.e. either the entry to a traced predicate called by the executed clause, or the success or failure exit from that clause).
- a: Abort** Causes execution to abort and control to return to the top level interpreter.
- b: Break** Calls the evaluable predicate *break*, thus invoking recursively a new incarnation of the system interpreter. The command prompt at break level *n* is

n: ?-

The user may return to the previous break level by entering the system end-of-file character (e.g. ctrl-D), or typing in the atom *end_of_file*; or to the top level interpreter by typing in *abort*.

- f: Fail** Causes execution to fail, thus transferring control to the Fail port of the current execution.
- h: Help** Displays the table of debugging options.

- l: Leap** Causes the system to resume running the program, only stopping when a spy-point is reached or the program terminates. This allows the user to follow the execution at a higher level than exhaustive tracing.
- n: Nodebug** Turns off debug mode.
- q: Quasi-skip** This is like Skip except that it does not mask out spy points.
- r: Retry (fail)** Transfers to the Call port of the current goal. Note, however, that side effects, such as database modifications etc., are not undone.
- s: Skip** Causes tracing to be turned off for the entire execution of the procedure. Thus, nothing is seen until control comes back to that procedure, either at the Success or the Failure port.

Other predicates that are useful in debugging are:

- untrace(*Preds*)** where *Preds* is a term P/N , where P is a predicate name and N its arity, or a list of such terms. Turns off tracing on the specified predicates. *Preds* must be instantiated at the time of the call.
- spy(*Preds*)** where *Preds* is a term P/N , where P is a predicate name and N its arity, or a list of such terms. Sets spy points on the specified predicates. *Preds* must be instantiated at the time of the call.
- nospy(*Preds*)** where *Preds* is a term P/N , where P is a predicate name and N its arity, or a list of such terms. Removes spy points on the specified predicates. *Preds* must be instantiated at the time of the call.
- debug** Turns on debugging mode. This causes subsequent execution of predicates with trace or spy points to be traced, and is a no-op if there are no such predicates. The predicates *trace*/1 and *spy*/1 cause debugging mode to be turned on automatically.
- nodebug** Turns off debugging mode. This causes trace and spy points to be ignored.
- debugging** Displays information about whether debug mode is on or not, and lists predicates that have trace points or spy points set on them.
- tracepreds(*L*)** Binds L to a list of terms P/N where the predicate P of arity N has a trace point set on it.

spypreds(*L*) Binds *L* to a list of terms *P/N* where the predicate *P* of arity *N* has a spy point set on it.

There is one known bug in the package: attempts to set trace points, via *trace/1*, on system and library predicates that are used by the trace package can cause bizarre behaviour.

6.2 Low-Level Tracing

SB-Prolog also provides a facility for low-level tracing of execution. This can be activated by invoking the simulator with the `-T` option, or through the predicate *\$trace/0*. It causes trace information to be printed out at every call (including those to system trap handlers). The volume of such trace information can very become large very quickly, so this method of tracing is not recommended in general.

Low-level tracing may be turned off using the predicate *untrace/0*.

7 The Simulator

The simulator resides in the SB-Prolog system directory `sim`. The following sections describe various aspects of the simulator.

7.1 Invoking the Simulator

The simulator is invoked by the command

```
sbprolog bc_file
```

where *bc_file* is a byte code file resulting from the compilation of a Prolog program. In almost all cases, the user will wish to interact with the SB-Prolog *query evaluator*, in which case *bc_file* will be *\$readloop*, and the command will be

```
sbprolog Path/$readloop
```

where *Path* is the path to the directory containing the command interpreter *\$readloop*. This directory, typically, is the system directory `modlib`.

The command interpreter reads in a query typed in by the user, evaluates it and prints the answer(s), repeating this until it encounters an end-of-file (the standard end-of-file character on the system, e.g. ctrl-D), or the user types *end_of_file* or *halt*.

The user should ensure that the directory containing the executable file `sim` (typically, the system directory `sim`) is included in the shell variable `path`; if not, the full path to the simulator will have to be specified.

In general, the simulator may be invoked with a variety of options, as follows:

```
sbprolog -options bc_file
```

or

```
sbprolog -option1 -option2 ... -optionn bc_file
```

The options recognized by the simulator are described below.

When called with a byte code file `bc_file`, the simulator begins execution with the first clause in that file. The first clause in such a file, therefore, should be a clause without any arguments in the head (otherwise, the simulator will attempt to dereference argument pointers in the head that are really pointing into deep space, and usually come to a sad end). If the user is executing a file in this manner rather than using the command interpreter, he should also be careful to include the undefined predicate handler ‘`_$undefined_pred`’/1, which is normally defined in the file `modlib/$init_sys.P`.

7.2 Simulator Options

The following is a list of options recognized by the simulator.

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

As an example, the command

```
sbprolog -s -p 60000 -m 150000 \ $readloop
```

starts the simulator executing the command interpreter with 60000 bytes of program space, 150000 bytes of local and global stack space and (by default) 30000 bytes of trail stack space; the *s* option also results in statistics information being maintained.

7.3 Interrupts

SB-Prolog provides a facility for exception handling using user-definable interrupt handlers. This can be used both for external interrupts, e.g. those generated from the keyboard by the user or from signals other processes; or internal traps, e.g. those caused by stack overflows, encountering undefined predicates, etc. For example, the “undefined predicate” interrupt is handled, by default, by the predicate ‘*_\$undefined_pred*’/1, which is defined in the files `modlib/src/$init_sys.P` and `modlib/src/$readloop.P`. The default action on encountering an undefined predicate is to attempt to dynamically load a file whose name matches that of the undefined predicate. However, the user may easily alter this behaviour by redefining the undefined predicate handler.

In general, interrupts are handled by the predicate ‘*_\$interrupt*’/2: a call to this predicate is of the form ‘*_\$interrupt*’(*Call*, *Code*), where *Call* is the call that generated the interrupt, and *Code* is an integer indicating the nature of the interrupt. For each interrupt code, the interrupt handler then calls a handler that is designed to handle that particular kind of interrupt. At this point, the following interrupt codes have predefined meanings:

- 0 undefined predicate;
- 1 keyboard interrupt (\textasciitilde C);
- 2 stack overflow.

Other interrupt codes may be incorporated by modifying the definition of the predicate ‘*_\$ interrupt*’/2 in the file `modlib/src/$readloop.P`.

Interrupts during execution are signalled from within the WAM simulator. The general method for raising an interrupt is using the function *set_intercode* in the file `sim/sub_inst.c`: to raise an interrupt whose code is *n*, the line

```
lpcreg = set_intercode(n);
```

is added to the appropriate place in the main loop of the interpreter, defined in `sim/main.c`.

8 The Compiler

The compiler translates Prolog source files into byte-code object files. It is written entirely in Prolog. The byte code for the compiler can be found in the SB-Prolog system directory `cmplib`, with the source code resident in `cmplib/src`.

Byte code files may be concatenated together to produce other byte code files. Thus, for example, if *foo1* and *foo2* are byte code files resulting from the compilation of two Prolog source programs, then the file *foo*, obtained by executing the shell command

```
cat foo1 foo2 > foo
```

is a byte code file as well, and may be loaded and executed. In this case, loading and executing the file `foo` would give the same result as loading *foo1* and *foo2* separately, which in turn would be the same as concatenating the original source files and compiling this larger file. This makes it easier to compile large programs: one need only break them into smaller pieces, compile the individual pieces, and concatenate the byte files together.

The following sections describe the various aspects of the compiler in more detail.

8.1 Invoking the Compiler

The compiler is invoked through the Prolog predicate *compile*:

```
| ?- compile(InFile [, OutFile ] [, OptionsList ]).
```

where optional parameters are enclosed in brackets. *InFile* is the name of the input (i.e. source) file; *OutFile* is the name of the output file (i.e. byte code) file; *OptionsList* is a list of compiler options (see below).

The input and output file names must be Prolog atoms, i.e. either begin with a lower case letter or dollar sign '\$', and consist only of letters, digits, and underscores; or, be enclosed within single quotes. If the output file name is not specified, it defaults to *InFile.out*. The list of options, if specified, is a Prolog list, i.e. a term of the form

`[option1, option2, ..., optionn].`

If left unspecified, it defaults to the empty list `[]`.

In fact, the output file name and the options list may be specified in any order. Thus, for example, the queries

`| ?- compile('/usr/debray/foo', foo_out, [v]).`

and

`| ?- compile('/usr/debray/foo', [v], foo_out).`

are equivalent, and specify that the Prolog source file `'/usr/debray/foo'` is to be compiled in verbose mode (see “Compiler Options” below), and that the byte code is to be generated into the file `foo_out`.

The *compile* predicate may also be called with a fourth parameter:

`| ?- compile(InFile, OutFile, OptionsList, PredList).`

where *InFile*, *OutFile* and *OptionsList* are as before; *compile*/4 unifies *PredList* with a list of terms *P/N* denoting the predicates defined in *InFile*, where *P* is a predicate name and *N* its arity. *PredList*, if specified, is usually given as an uninstantiated variable; its principal use is for setting trace points on the predicates in the file (see Section 6), e.g. by executing

`| ?- compile('/usr/debray/foo', foo_out, [v], L),
load(foo_out), trace(L).`

Notice that *PredList* can only appear in *compile*/4.

8.2 Compiler Options

The following options are currently recognized by the compiler:

- a** Specifies that an “assembler” file is to be created. The name of the assembler file is obtained by appending `.asl` to the source file name. While the writing out of assembly code slows down the compilation process to some extent, it allows the assembler to do a better job of optimizing away indirect subroutine linkages (since in this case the assembler has assembly code for the entire program to work with at once, not just a single predicate). This results in code that is faster and more compact.
- d** Dumps expanded macros to the user (see Section 10).

- e Expand macros (see Section 10).
- t If specified, turns off assembler optimizations that eliminate indirect branches through the symbol table in favour of direct branches. This is useful in debugging compiled code. It is *necessary* if the extension table feature is to be used.
- v If specified, compiles in “verbose” mode, which causes messages regarding progress of compilation to be printed out.

8.3 Assembly

The SB-Prolog assembler can be invoked by loading the compiler and using the predicate `$asm/3`:

```
| ?- $asm(InFile, OutFile, OptionsList).
```

where *InFile* is a Prolog atom which is the name of a WAM assembly source file (e.g. the “.asl” file generated when a Prolog program is compiled with the “a” option), *OutFile* is an atom which is the name of the intended byte code file, and *OptionsList* is a list of options. The options recognized by the assembler are:

- v “Verbose” mode. Prints out information regarding progress of assembly.
- t “Trace”. If specified, the assembler generates code to force procedure calls to branch indirectly via the symbol table, instead of using a direct branch. This is useful for tracing compiled code. It is *necessary* if the extension table feature is to be used.

The assembler is intended primarily to support the compiler, so the assembly language syntax is quirky in places. The interested reader is advised to look at the assembly files resulting from compilation with the “a” option for more on SB-Prolog assembler syntax.

8.4 Compiler Directives

8.4.1 Mode Declarations

The user may declare input and output arguments of predicates using mode declarations. These declarations, for an *n*-ary predicate *p*, are of the form

```
:- mode p( Mode ).
```

where *Mode* consists of *n* mode values; or

```
:- mode(p, n, ModeList)
```

where *ModeList* is a list of mode values of length *n*. Mode values may be the following:

- c, ++** Indicates that the corresponding argument position is always a ground term in any call to the predicate. The argument is therefore an input argument.
- nv, +** Indicates that the corresponding argument position is always a non-variable term (i.e. is instantiated) in any call in any call to the predicate. The argument is therefore an input argument.
- f, -** Indicates that the corresponding argument position is always an uninstantiated variable in any call to the predicate. The argument is therefore an output argument.
- d, ?** Indicates that the corresponding argument may be any term in calls to the predicate.

For example, a 3-ary predicate *p* whose first argument is always a ground term in a call, whose second argument is always uninstantiated, and whose third argument can be any term, may have its mode declared as

```
:- mode p(++ , -- , d)
```

or as

```
:- mode(p, 3, [c, f, d]).
```

Currently, mode information is used by the compiler in two ways. First, it often allows more compact code to be generated. The second use is in guiding program transformations that allow faster code to be generated. For example, the predicate

```
part([], _, [], []).
part([E|L], M, [E|U1], U2) :- E =< M, part(L, M, U1, U2).
part([E|L], M, U1, [E|U2]) :- E > M, part(L, M, U1, U2).
```

executes about 30% faster with the mode declaration

```
:- mode part(++ , ++ , - , -).
```

than without.

8.4.2 Indexing Directives

The compiler usually generates an index on the principal functor of the first argument of a predicate. The user may direct the compiler to generate an index on any other argument by means of an indexing directive. This is of the form

```
:- index(Pred, Arity, IndexArg)
```

indicating that an index should be created on the $IndexArg^{th}$ argument of the predicate $Pred/Arity$. All of the values $Pred$, $Arity$ and $IndexArg$ should be bound in the directive: $Pred$ should be an atom, $Arity$ a nonnegative integer, and $IndexArg$ an integer between 0 and $Arity$. If $IndexArg$ is 0, then no index is created for that predicate. As an example, if we wished to create an index on the third argument of a 5-ary predicate *foo*, the compiler directive would be

```
:- index(foo, 5, 3).
```

An index directive may be placed anywhere in the file containing the predicate it refers to.

9 Libraries

To describe how libraries are currently supported in our system, we must describe the interrupt handler `_$undefined_pred/1`. The system keeps a table of libraries and routines that are needed from each. When a predicate is found to be undefined, the table is searched to see if it is defined by some library file. If so, that file is loaded (if it hasn't been previously loaded) and the association is made between the routine name as defined in the library file, and the routine name as used by the invoker.

The table of libraries and needed routines is:

```
defined_mods(Modname, [pred1/arity1, ..., predn/arityn]).
```

where $Modname$ is the name of the library. It exports n predicate definitions. The first exported pred is of arity $arity_1$, and needs to be invoked by the name of $pred_1$.

The table of libraries that have already been loaded is given by

```
loaded_mods(Modname).
```

A library file is a file of predicate definitions, together with a fact defining a list of predicates exported by it; and a set of facts, each of which specifies, for some other library file, the predicates imported from that library file. For example, consider a library name ‘*p*’. It contains a single fact, named *p_export*, that is true of the list of predicate/arities that are exported. E.g.

`p_export([p1/2, p2/4])`

indicates that the module *p* exports the predicates *p1/2* and *p2/4*. For each library *m* which contains predicates needed by the library *p*, there is a fact for *p_use*, describing what library is needed and the names of the predicates defined there that are needed. For example, if library *p* needs to import predicates *ip1/2* and *ip2/3* from library *q*, there would be a fact

`p_use(q, [ip1/2, ip2/3])`

where *q* is a module that exports two predicates: one 2-ary and one 3-ary. This list corresponds to the export list of library *q*.

The correspondence between the predicates in the export list of an exporting library, and those in the import or *use* list of a library which imports one or more of them, is by position, i.e. the predicate names at the exporting and importing names may be different, and the association between names in the two lists is by the position in the list. If the importing library does not wish to import one or more of the predicates exported by the exporting module, it may put an anonymous variable in the corresponding position in its *use* list. Thus, for example, if library *p* above had wished to import only the predicate *ip2/3* from library *q*, the corresponding use fact would be

`p_use(q, [_, ip2/3]).`

The initial set of predicates and the libraries from which they are to be loaded is set up by an initial call to *\$prorc/0* (see the SB-Prolog system file `modlib/src/$prorc.P`). This predicate makes initial calls to the predicate `$define_mod` which set up the tables described above so that the use of standard predicates will cause the correct libraries to be loaded in the *\$_undefined_pred* routine, and the correct names to be used.

10 Macros

SB-Prolog features a facility for the definition and expansion of macros that is fully compatible with the runtime system. Its basic mechanism is a simple

partial evaluator. It is called by both *consult* and *compile*, so that macro expansion occurs independently of whether the code is interpreted or compiled (but not when asserted). Moreover, the macro definitions are retained as clauses at runtime, so that invocation of macros via *call/1* at runtime (or from asserted clauses) does not pose a problem. This means, however, that if the same macro is used in many different files, it will be loaded more than once, thus leading to wasted space. This ought to be thought about and fixed.

The source for the macro expander is in the SB-Prolog system file `modlib/src/$mac.P`.

10.1 Defining Macros

‘Macros’, or predicates to be evaluated at compile-time, are defined by clauses of the form

`Head :- Body`

where facts have ‘true’ as their body. The partial evaluator will expand any call to a predicate defined by `:-/2` that unifies with the head of only one clause in `:-/2`. If a call unifies with the head of more than one clause in `:-/2`, it will not be expanded. Notice that this is not a fundamental restriction, since ‘;’ is permitted in the body of a clause. The partial evaluator also converts each definition of the form

`Head :- Body.`

to a clause of the form

`Head :- Body.`

and adds this second clause to the other “normal” clauses that were read from the file. This ensures that calls to the macro at runtime, e.g. through *call/1* or from unexpanded calls in the program do not cause any problems.

The partial evaluator is actually a Prolog interpreter written ‘purely’ in Prolog, i.e., variable assignments are explicitly handled. This is necessary to be able to handle impure constructs such as `var(X)`, `X=a`. As a result this is a *very slow* Prolog evaluator.

Since naïve partial evaluation can go into an infinite loop, SB-Prolog’s partial evaluator maintains a depth-bound and will not expand recursive calls deeper than that. The depth is determined by the `globalset` predicate `$mac_depth`. The default value for `$mac_depth` is 50. This can be changed to some other value *n* by executing

`| ?- globalset($mac_depth(n)).`

10.2 Macro Expander Options

The following options are recognized by the macro expander:

- d** Dumps all clauses to the user after expansion. Useful for debugging.
- e** Expand macros. If omitted, the expander simply converts each `:-/2` clause to a normal `:-/2` clause.
- v** “Verbose” mode. Prints macros that are/are not being expanded.

11 Extension Tables: Memo Relations

Extension tables store the calls and answers for a predicate. If a call has been made before, answers are retrieved from the extension table instead of being recomputed. Extension tables provide a caching mechanism for Prolog. In addition, extension tables affect the termination characteristics of recursive programs. Some Prolog programs, which are logically correct, enter an infinite loop due to recursive predicates. An extension table saved on recursive predicates can find all answers (provided the set of such answers is finite) and terminate for some logic programs for which Prolog’s evaluation strategy enters an infinite loop. Iterations over the extension table execution strategy provides complete evaluation of queries over function-free Horn clause programs.

To be able to use the simple extension table evaluation on a set of predicates, the source file should either be consulted, or compiled with the `t` option (the `t` option keeps the assembler from optimizing subroutine linkage and allows the extension table facility to intercept calls to predicates).

To use extension table execution, all predicates that are to be saved in the extension table must be passed to `et/1`. For example,

```
| ?- et([pred1/1, pred2/2]), et(pred3/2)
```

will set up “ET-points” for the predicates `pred1/1`, `pred2/2` and `pred3/2`, which will cause extension tables for these predicates to be maintained during execution. At the time of the call to `et/1`, these predicates must be defined, either by having been loaded, or through *consult*.

The predicate `noet/1` takes a list of predicate/arity pairs for which ET-points should be deleted. Notice that once an ET-point has been set up for a predicate, it will be maintained unless explicitly deleted via `noet/1`. If the definition of a predicate which has an ET-point defined is to be updated,

the ET-point must first be deleted via *noet/1*. The predicate can then be reloaded and a new ET-point established. This is enforced by the failure of the goal “et(P/N)” if an ET-point already exists for the argument predicate. In this case, the following error message will be displayed:

```
*et* already defined for: P/N
```

There are, in fact, two extension table algorithms: a simple one, which simply caches calls to predicates which have ET-points defined; and a complete ET algorithm, which iterates the simple extension table algorithm until no more answers can be found. The simple algorithm is more efficient than the complete one; however, the simple algorithm is not complete for certain especially nasty forms of mutual recursion, while the complete algorithm is. To use the simple extension table algorithm, predicates can simply be called as usual. The complete extension table algorithm may be used via the query

```
| ?- et_star(Query).
```

The extension table algorithm is intended for predicates that are “essentially pure”, and results are not guaranteed for code using impure code. The extension table algorithm saves only those answers which are not instances of what is already in the table, and uses these answers if the current call is an instance of a call already made. For example, if a call $p(X, Y)$, with X and Y uninstantiated, is encountered and inserted into the extension table, then a subsequent call $p(X, b)$ will be computed using the answers for $p(X, Y)$ already in the extension table. Notice that this might not work if var/nonvar tests are used on the second argument in the evaluation of p .

Another problem with using impure code is that if an ET predicate is cut over, then the saved call implies that all answers for that predicate were computed, but there are only partial results in the ET because of the cut. So on a subsequent call the incomplete extension table answers are used when all answers are expected. An example is shown in Figure 11

```
r(X,Y) :- p(X,Y),q(Y,Z),!,fail.
| ?- r(X,Y) ; p(X,Y).
```

Figure 3: Extension Table Example

Let p be an ET predicate whose evaluation yields many tuples. In the evaluation of the query, $r(X,Y)$ makes a call to $p(X,Y)$. Assuming that there

is a tuple such that $q(Y,Z)$ succeeds with the first p tuple then the evaluation of p is cut over. The call to $p(X,Y)$ in the query uses the extension table because of the previous call in the evaluation of $r(X,Y)$. Only one answer is found, whereas the relation p contains many tuples, so the computation is not complete. Note that “cuts” used within the evaluation of an ET predicate are ok, as long as they don’t cut over the evaluation of another ET predicate. The evaluation of the predicate that uses cuts does not cut over any ET processing (such as storing or retrieving answers) so that the tuples that are computed are saved. In the following example, the ET is used to generate prime numbers where an ET point is put on `prime/1`. Example:

```
prime(I) :- globalset(globalgenint(2)),fail.  /* Generating Primes */
prime(I) :- genint(I), not(div(I)).
div(I) :- prime(X), 0 is I mod X.

genint(N) :-
repeat,
globalgenint(N),
N1 is N+1,
globalset(globalgenint(N1)).
```

The following summarizes the library predicates supporting the extension table facility:

et(*L*) Sets up an ET-point on the predicates *L*, which causes calls and answers to these predicates to be saved in an “extension table”. *L* is either a term *Pred/Arity*, where *Pred* is a predicate symbol and *Arity* its arity, or a set of such terms represented as a list. *L* must be instantiated, and the predicates specified in it defined, at the time of the call to *et/1*. Gives error messages and fails if any of the predicates in *L* is undefined, or if an ET-point already exists on any of them; in this case, no ET-point is set up on any of the predicates in *L*.

et_star(*Goal*) Invokes the complete extension table algorithm on the goal *Goal*.

et_points(*L*) Unifies *L* with a list of predicates for which an ET-point is defined. *L* is the empty list [] if there are no ET-points defined.

noet(*L*) Deletes ET-points on the predicates specified in *L*. *L* is either a term *P/N*, where *P* is the name of a predicate and *N* its arity, or a set of such terms represented as a list. Gives error messages and fails if there is no ET-point on any of the predicates specified in *L*. Deleting an ET-point for a predicate also removes the calls and answers stored in the extension table for that predicate. The extension tables for all predicates for which ET-points are defined may be deleted using *et_points/1* in conjunction with *noet/1*.

L must be instantiated at the time of the call to *noet/1*.

et_remove(*L*) Removes both calls and answers for the predicates specified in *L*. In effect, this results in the extension table for these predicates to be set to empty. *L* must be instantiated at the time of the call to either a term *P/N*, where *P* is a predicate with arity *N*, or a list of such terms. An error occurs if any of the predicates in *L* does not have an ET-point set.

All extension tables can be emptied by using *et_points/1* in conjunction with *et_remove/1*.

et_answers(*P/N*, *Term*) Retrieves the answers stored in the extension table for the predicate *P/N* in *Term* one at a time. *Term* is of the form *P*(*t*₁, ..., *t*_{*N*}). An error results and *et_answers/2* fails if *P/N* is not fully specified (ground), or if *P/N* does not have an ET-point set.

et_calls(*P/N*, *Term*) Retrieves the calls stored in the extension table for the predicate *P/N* in *Term* one at a time. *Term* is of the form *P*(*t*₁, ..., *t*_{*N*}). An error results and *et_calls/2* fails if *P/N* is not fully specified (ground), or if *P/N* does not have an ET-point set.

12 Definite Clause Grammars

Definite clause grammars are an extension of context free grammars, and may be conveniently expressed in Prolog. A grammar rule in Prolog has the form

$$Head \text{ --> } Body.$$

with the interpretation “a possible form for *Head* is *Body*”. Extra conditions, in the form of explicit Prolog literals or control constructs such as *if-then-else* (*->*) or *cut* (!), may be included in *Body*.

The syntax of DCGs supported by SB-Prolog is as follows:

1. A non-terminal symbol may be any Prolog term other than a variable.
2. A terminal symbol may be any Prolog term. To distinguish terminals from nonterminals, a sequence of terminal symbols

$$a, b, c, d, \dots$$

is written as a Prolog list $[a, b, c, d, \dots]$, with the empty sequence written as the empty list $[]$. If the terminal symbols are ASCII character codes, they can be written (as elsewhere) as strings.

3. Extra conditions, in the form of Prolog literals, can be included in the right-hand side of a rule by enclosing such conditions in curly braces, $\{$ and $\}$. E.g., one can write

$$\text{natnum}(X) \text{ --> } \{ \text{integer}(X), X \geq 0 \}.$$

4. The left hand side of a rule consists of a single nonterminal. Notice that “push-back lists” are thus not supported.
5. The right hand side of a rule may contain alternatives (written using the disjunction operator ‘;’ or $|$), and control primitives such as if-then-else (-->), *not/1* and *cut* (!). The use of *not/1* on the right hand side of grammar rules is not recommended, however, because their semantics in this context is murky at best. All other control primitives, e.g. *repeat/0*, must explicitly be enclosed within curly braces if they are not to be interpreted as nonterminals.

Except for the restriction of lists of terminals in the left hand sides of rules, the translation of DCGs in SB-Prolog is very similar to that in Quintus Prolog.

Library predicates supporting DCGs are the following:

dcg(*Rule*, *Clause*) Succeeds if the DCG rule *Rule* corresponds to the Prolog clause *Clause*. At the time of call, *Rule* must be bound to a term whose principal functor is $\text{-->}/2$.

phrase(*Phrase*, *List*) The usual way to commence execution of grammar rules. The list *List* is a phrase (i.e., sequence of terminals) generated

by *Phrase* according to the current grammar rules. *Phrase* is a non-terminal (in general, the right hand side of a grammar rule), and must be instantiated to a nonvariable term in the call. If *List* is bound to a list of terminals in the call, then the goal corresponds to parsing *List*; if *List* is unbound in the call, then the grammar is being used for generation.

expand_term(*T1*, *T2*) This predicate is used to transform terms that are read in, when a file is consulted or compiled. The usual use is to transform grammar rules into Prolog clauses: if *T1* is a grammar rule, then *T2* is the corresponding Prolog clause. Users may define their own transformations by defining the predicate *term_expansion/2*. When a term *T1* is read in when a file is being compiled or consulted, *expand_term/2* first calls *term_expansion/2*: if the expansion succeeds, the transformed term so obtained is used; otherwise, if *T1* is a grammar rule, then it is expanded using *dcg/2*; otherwise, *T1* is used as is.

‘C’(*S1*, *Terminal*, *S2*) Used to handle terminal symbols in the expansion of grammar rules. Not usually of direct use to the user. This is defined as

‘C’([X|S], X, S).

13 Profiling Programs

There is an experimental utility for profiling programs interactively. Two kinds of profiling are supported: one may count the number of calls to a predicate, or compute the time spent in a predicate. It is important that the predicates being profiled are either consulted, or compiled with the *t* option, so that calls to the relevant predicates can be intercepted by the profiler.

To use the profiler, predicates whose calls are to be counted must be passed to *count/1*, e.g.

```
| ?-- count([p/1, q/2]), count(r/3).
```

will set up “count-points” on the predicates *p/1*, *q/2* and *r/3*. Predicates whose calls are to be timed have to be passed to *time/1*, e.g.

```
| ?-- time([s/1, t/2]), time(u/3).
```

will set up “time-points” on the predicates $s/1$, $t/2$ and $u/3$. It is possible to set both count-points and time-points on the same predicate. After count-points and time-points have been set, the program may be executed as many times as desired: the profiling system will accumulate call counts and execution times for the appropriate predicates. Execution profiles may be obtained using the predicates *prof_stats/0* or *prof_stats/1*. Using *prof_stats/0* to display the execution profile will cause the call counts and execution times of predicates being profiled to be reset to 0 (this may be avoided by using *prof_stats/1*).

It should be noted that in this context, the “execution time” for a predicate is an estimate of the total time spent in the subtrees below calls to that predicate (including failed subtrees): thus, the execution time figures may be dilated slightly if the subtree below a timed predicate contains predicates that are being profiled, because of the time taken for updating the call counts and execution times. For each predicate, the execution time is displayed as the fraction of time spent, in computation in subtrees under calls to that predicate, relative to the time elapsed from the last time profiling was timed on or the last time profiling statistics were taken, whichever was more recent.

Bugs: May behave bizarrely if a predicate being profiled contains cuts.

The following summarizes the library predicates supporting profiling:

count(*L*) Sets up a count-point on the predicates *L*, which causes calls to these predicates to be counted, and turns profiling on. *L* is either a term *Pred/Arity*, where *Pred* is a predicate symbol and *Arity* its arity, or a set of such terms represented as a list. *L* must be instantiated, and the predicates specified in it defined, at the time of the call to *count/1*.

time(*L*) Sets up a time-point on the predicates *L*, which causes execution times for calls to these predicates to be accumulated, and turns profiling on. *L* is either a term *Pred/Arity*, where *Pred* is a predicate symbol and *Arity* its arity, or a set of such terms represented as a list. *L* must be instantiated, and the predicates specified in it defined, at the time of the call to *time/1*.

nocount(*L*) Deletes the count-point on the predicates *L*. *L* is either a term *Pred/Arity*, where *Pred* is a predicate symbol and *Arity* its arity, or a set of such terms represented as a list. *L* must be instantiated, and the predicates specified in it defined, at the time of the call to *nocount/1*.

notime(*L*) Deletes the time-point on the predicates *L*. *L* is either a term *Pred/Arity*, where *Pred* is a predicate symbol and *Arity* its arity, or a

set of such terms represented as a list. *L* must be instantiated, and the predicates specified in it defined, at the time of the call to *time/1*.

profiling Displays information about whether profile mode is on or not, and lists predicates that have count- and time-points set on them.

prof_reset(*L*) Resets call counts and/or execution times for the predicates *L*. *L* is either a term *Pred/Arity*, where *Pred* is a predicate symbol and *Arity* its arity, or a set of such terms represented as a list. *L* must be instantiated, and the predicates specified in it defined, at the time of the call to *prof_reset/1*.

resetcount(*L*) Resets call counts for the predicates *L*. *L* is either a term *Pred/Arity*, where *Pred* is a predicate symbol and *Arity* its arity, or a set of such terms represented as a list. *L* must be instantiated, and the predicates specified in it defined, at the time of the call to *resetcount/1*.

resetttime(*L*) Resets execution times for the predicates *L*. *L* is either a term *Pred/Arity*, where *Pred* is a predicate symbol and *Arity* its arity, or a set of such terms represented as a list. *L* must be instantiated, and the predicates specified in it defined, at the time of the call to *resetttime/1*.

profile Turns profiling on. This causes subsequent execution of predicates with count- or time-points to be profiled, and is a no-op if there are no such predicates. The predicates *count/1* and *time/1* cause profiling to be turned on automatically.

noprofile Turns profiling off. This causes count- and time-points to be ignored.

timepreds(*L*) Unifies *L* to a list of terms *P/N* where the predicate *P* of arity *N* has a time point set on it.

countpreds(*L*) Unifies *L* to a list of terms *P/N* where the predicate *P* of arity *N* has a count point set on it.

prof_stats Causes the call counts and/or execution times accumulated since the last call to *prof_stats/0* to be printed out for predicates that are being profiled. The execution times are given as fractions of the total time elapsed since the last time profiling was turned on, or the last time *prof_stats* was called, whichever was most recent. This also results in

the call counts and relative execution times of these predicates being reset to 0. Equivalent to *prof_stats*(1).

prof_stats(*N*) Causes the call counts and/or execution times accumulated since the last call to *prof_stats*/0 to be printed out for predicates that are being profiled. The execution times are given as fractions of the total time elapsed since the last time profiling was turned on, or the last time *prof_stats* was called, whichever was most recent. If *N* is 1, then this also results in the call counts and execution times of these predicates being reset to 0; otherwise, the call counts and execution times are not reset.

14 Other Library Utilities

The SB-Prolog library contains various other utilities, some of which are listed below.

\$append(*X*, *Y*, *Z*) Succeeds if list *Z* is the concatenation of lists *X* and *Y*.

\$member(*X*, *L*) Checks whether *X* unifies with any element of list *L*, succeeding more than once if there are multiple such elements.

\$memberchk(*X*, *L*) Similar to *\$member*/2, except that *\$memberchk*/2 is deterministic, i.e. does not succeed more than once for any call.

\$reverse(*L*, *R*) Succeeds if *R* is the reverse of list *L*. If *L* is not a fully determined list, i.e. if the tail of *L* is a variable, this predicate can succeed arbitrarily many times.

\$merge(*X*, *Y*, *Z*) Succeeds if *Z* is the list resulting from “merging” lists *X* and *Y*, i.e. the elements of *X* together with any element of *Y* not occurring in *X*. If *X* or *Y* contain duplicates, *Z* may also contain duplicates.

\$absmember(*X*, *L*) Similar to *\$member*/2, except that it checks for identity (through *==*/2) rather than unifiability (through *=*/2) of *X* with elements of *L*.

\$nthmember(*X*, *L*, *N*) Succeeds if the *N*th element of the list *L* unifies with *X*. Fails if *N* is greater than the length of *L*. Either *X* and *L*, or *L* and *N*, should be instantiated at the time of the call.

\$member2(X, L) Checks whether X unifies with any of the actual elements of L . The only difference between this and *\$member/2* is on lists with a variable tail, e.g. `[a, b, c | _]`: while *\$member/2* would insert X at the end of such a list if it did not find it, *\$member2/2* only checks for membership but does not insert it into the list if it is not there.

length(L, N) Succeeds if the length of the list L is N . This predicate is deterministic if L is instantiated to a list of definite length, but is nondeterministic if L is a variable or has a variable tail.

subsumes(X, Y) Succeeds if the term X subsumes the term Y (i.e. if Y is an instance of X).

15 CREDITS

The initial development of SB-Prolog, from 1984 to August 1986, was at SUNY at Stony Brook, where Versions 1.0 and 2.0 were developed. Since August 1986, its development has continued at the University of Arizona, Tucson.

A large number of people were involved, at some time or another, with the Logic Programming group at SUNY, Stony Brook, and deserve credit for helping to bring SB-Prolog to its present form. David Scott Warren led the project at Stony Brook. Most of the simulator and builtins were written by Jiyang Xu and David S. Warren (I added the later stuff, Versions 2.1 onwards). Much of the library was also by David, with some contributions from me. Weidong Chen did the work on clause indexing. Suzanne Dietrich wrote the Extension Table package. I wrote most of the compiler.

Several people helped debug previous versions, including Leslie Rohde; Bob Beck of Sequent Computers; and Mark Gooley of the University of Illinois at Urbana-Champaign.

Special thanks are due to Richard O’Keefe, who contributed the Prolog code for the parser (in the form of the predicates **read/1** and **read/2**), the C code for the tokenizer, and the code for **setof/3** and **bagof/3**.

I am grateful to Fernando Pereira for permission to use material from the C-Prolog manual for the descriptions of Prolog syntax and many of the builtins in this User Manual. Steve Kelem produced the LaTeX version of this manual from an earlier troff version.

— S.K.D.

Index

- !/0, 18, **26**, 28, 58, 60, 61, 76
- < /2, **24**
- =< /2, **24**
- = \ = /2, 24
- > /2, **24**
- >=/2, **24**
- \=/2, **25**
- \ == /2, **30**
- \wedge , 29
- \wedge /2, **29**
- ,/2, **25**
- > /2, **26**
- :-/1, 11, 16, 57
- ::-/2, **56**
- ;/2, **25**
- =../2, **27**
- =/2, **25**
- :=/2, **24**
- ==/2, **30**
- ?=/2, **25**
- < /2, **30**
- =< /2, **31**
- > /2, **30**
- >= /2, **31**
- \$absmember/2, **65**
- \$alloc_buff/5, **42**
- \$append/3, **65**
- \$asm/3, **52**
- \$assertf_alloc_t, **43**
- \$current_atom/2, **36**
- \$current_functor/3, **36**
- \$current_predicate/3, **37**
- \$db_add_clref/7, **43**
- \$db_assert_fact/5, **43**
- \$db_call_prref/2, **44**
- \$db_call_prrefs/2, **44**
- \$db_get_clauses/3, **44**
- \$db_new_prref/3, **43**
- \$exists/1, **21**
- \$getenv/2, **38**
- \$member/2, **65**
- \$member2/2, **66**
- \$memberchk/2, **65**
- \$merge/3, **65**
- \$nthmember/3, **65**
- \$reverse/2, **65**
- \$trace/0, 47
- \$untrace/0, 47
- _\$interrupt/2, 49
- ‘C’/3, **62**
- abolish
 - /1, **35**
 - /2, **35**
- abort
 - trace facility, 45
- abort/0, **38**, 38
- alloc_heap/2, **32**
- alloc_perm/2, **32**
- arg/3, **27**, 77
- arguments
 - processing all from a term, 77
- arithmetic, 22
- assembler
 - options, 52
- assembly, 52
- assert, 33
 - /1, **33**
 - /2, **33**
 - /4, **34**
- assert_union/2, **33**
- asserta

- /1, **33**
 - /2, **33**
- asserti/2, **33**
- assertz
 - /1, **33**
 - /2, **33**
- atom/1, **26**
- atomic/1, **27**
- atoms, 11
- backtrack points, 75
- bagof/3, **29**
- behaviour, standard execution, 18
- break/0, **38**
- buffers, 31
- builtins, adding, 79
- byte code
 - files, 5–8, 11, 47, 52
 - compiler, 50
 - concatenating, 9, 50
 - loading, 9
 - overwriting trace points, 45
 - translator, 7
- call/1, **28**
- call_ref
 - /2, **42**
 - /3, **42**
- character I/O, 22
- clause, 17
 - /2, **34**
 - /3, **35**
- cmplib, 7, 50
- compare/3, **31**
- comparison of terms, 30
- compile
 - /1, **8**
 - /2, **8**
 - /3, **8**
- /4, **8**
- Compiler, 50
 - directives, 52
 - invoking, 50
 - options, 51
- compiling programs, 8
- conlength/2, **28, 32**
- constants, 11
- consult, 8, 10
 - /1, **10**
 - /2, **10**
 - options, 10
- consulting programs, 10
- control, extra, 26
- count/1, **63**
- countpreds/1, **64**
- cputime/1, **38**
- Credits, 66
- current_atom/1, **36**
- current_functor/2, **36**
- current_predicate/2, **37**
- cut, 18, **26**, 28, 58, 60–61, 76
- cuts and If-Then-Else, 18
- database, internal, 35
- dcg/2, **61**
- debug/0, **46**
- debugging, 44
 - /0, **46**
- declarations
 - mode, 52
- definite clause grammars, 60
- definitions
 - macros, 56
- directives
 - Compiler, 52
 - indexing, 54
- directories, system, 7
- display/1, **21**

- dynamic loader search path, 6
- efficiency, coding for, 75
- environmental predicates, 38
- erase/1, **36**
- et/1, **59**
- et_answers/2, **60**
- et_calls/2, **60**
- et_points/1, **59**
- et_remove/1, **60**
- et_star/1, **59**
- eval/2, **24**
- evaluable predicates, 19, 72
- executing programs, 8
- execution behaviour, standard, 18
- execution directives, 11
- exotica, 42
- exp/2, **25**
- expand_term/2, **62**
- extension tables
 - memo relations, 57
- fail/0, **26**
- file handling, 20
- findall/3, **29**
- float/1, **27**
- floatc/3, **24**
- floating point numbers, unification
 - of, 19
- floor/2, **24**
- functor/3, **27**
- gennum/1, 41
- gensym/2, **42**
- get/1, **22**
- get0/1, **22**
- getting started, 6
- global values, 41
- globalset/1, **41**
- grammars
 - definite clause, 60
- high-level tracing, 44
- I/O
 - term, 21
- If-Then-Else and cuts, 18
- index/3, 33, **54**
- indexing, 34
 - directives, 54
 - on floating point, 19
- input, 20
- instance/2, **36**
- integer/1, **26**
- integers, 11
- internal database, 35
- interrupts, 49
- invoking the Compiler, 50
- invoking the simulator, 7, 47
- is/2, **24**
- is_buffer/1, **27**
- keysort/2, **31**
- length/2, **66**
- libraries, 54
- linking, dynamic search path, 6
- listing
 - /0, **36**
 - /1, **36**
- load/1, **9**
- loader, dynamic search path, 6
- loading byte code files, 9
- low-level predicates, 42
- low-level tracing, 47
- Macro Expander options, 57
- macros, 55
 - definition of, 56
- memo relations

- extension tables, 57
- meta-logical predicates, 26
- mode
 - declarations, 52
 - values, 53
- mode/3, **53**
- modification of the program, 32
- name/2, **28**
- nl/0, **22**
- nocount/1, **63**
- nodebug/0, **46**
- nodynload/2, **39**
- noet/1, **60**
- nonvar/1, **26**
- noprofile/0, **64**
- nospy/1, **46**
- not unifiable, *see* \=/2
- not/1, **26**
- notime/1, **63**
- number/1, **27**
- occurs check
 - unification without, 18
- op/3, **15, 38**
- operational semantics, 18
- operators, 14
- options
 - Compiler, 51
 - Macro Expander, 57
 - Simulator, 48
- output, 20
- path, search, 6
- phrase/2, **61**
- portray_clause/2, **22**
- portray_term/2, **22**
- predicate_property/2, **37**
- predicates
 - evaluable, 72
 - environmental, 38
 - evaluable, 19
 - low-level, 42
 - meta-logical, 26
- print/1, **21**
- print_al/2, **22**
- print_ar/2, **22**
- prof_reset/1, **64**
- prof_stats
 - /0, **64**
 - /1, **65**
- profile/0, **64**
- profiling programs, 62
- profiling/0, **64**
- program, state of, 36
- put/1, **22**
- query, 17
- query evaluator, 7, 47
- read/1, **21**
- real/1, **26**
- reconsult, 10
- recorda/3, **35**
- recorded/3, **35**
- recordz/3, **35**
- registers
 - minimizing data movement between, 77
- repeat/0, **26**
- resetcount/1, **64**
- resettime/1, **64**
- restore/1, **38**
- retract/1, **35**
- rounding, 23
- rule, 17
- save/1, **38**
- search path, 6
- see/1, **20**

- seeing/1, **20**
- seen/0, **20**
- semantics, operational, 18
- setof/3, **29**
- sets, 29
- SIMPATH, 6, 9
- Simulator, 47
 - options, 48
- simulator, invoking, 7, 47
- sin/2, **25**
- sort/2, **31**
- spy/1, **46**
- sypreds/1, **47**
- square/2, **25**
- standard execution behaviour, 18
- starting, 6
- state of the program, 36
- statistics
 - /0, **39**
 - /2, **39**
- strings, 13
- structure/1, **27**
- subsumes/2, **66**
- syntype/2, **40**
- syntax, 11
- syscall/3, **40**
- system directories, 7
- system/1, **40**
- tab/1, **22**
- tell/1, **21**
- telling/1, **21**
- term
 - processing all arguments of, 77
- term I/O, 21
- term_expansion/2, 62
- terms, 11
 - comparison of, 30
- testing unifiability, 78
- time/1, **63**
- timepreds/1, **64**
- told/0, **21**
- trace
 - options, 45
- trace/1, **44**
- tracepreds/1, **46**
- tracing
 - high-level, 44
 - low-level, 47
- trimbuff/3, **32**
- true/0, **25**
- undefined_pred/1, **8**
- unifiability
 - testing, 78
- unification
 - floating point numbers, 19
 - without occurs check, 18
- Unix
 - system calls, **40**
- untrace/1, **46**
- var/1, **26**
- WAM, 5, 19, 49, 52
- write/1, **21**
- writename/1, **21**
- writeln/1, **21**
- writelnname/1, **21**

A Evaluable Predicates of SB-Prolog

An entry of “B” indicates a builtin predicate, “I” an inline predicate, and “L” a library predicate. A “P” indicates that the predicate is handled by the pre-processor during compilation and/or consulting. A “D” denotes a compiler directive.

!/0 (P), 26	\$db_call_prref/2 (L), 44
< /2 (I), 24	\$db_call_prref_s/2 (L), 44
=< /2 (I), 24	\$db_get_clauses/3 (L), 44
= \ = /2 (I), 24	\$db_new_prref/3 (L), 43
> /2 (I), 24	\$exists/1 (B), 21
>=/2 (I), 24	\$getenv/2 (L), 38
\ =/2 (I), 25	\$member/2 (L), 65
\ == /2 (B), 30	\$member2/2 (L), 66
^/2 (L), 29	\$memberchk/2 (L), 65
,/2 (I), 25	\$merge/3 (L), 65
-> /2 (P), 26	\$nthmember/3 (L), 65
:-/1 (P), 11	\$reverse/2 (L), 65
::-/2 (P), 56	\$trace/0 (L), 47
;/2 (I), 25	\$untrace/0 (L), 47
=../2 (L), 27	\$_interrupt/2 (L), 49
=/2 (I), 25	‘C’/3 (L), 62
:=/2 (I), 24	abolish/1 (L), 35
==/2 (B), 30	abolish/2 (L), 35
?=/2 (I), 25	abort/0 (B), 38
< /2 (B), 30	alloc_heap/2 (L), 32
=< /2 (B), 31	alloc_perm/2 (L), 32
> /2 (B), 30	arg/3 (I), 27
>= /2 (B), 31	assert/1 (L), 33
\$absmember/2 (L), 65	assert/2 (L), 33
\$alloc_buff/5 (L), 42	assert/4 (L), 34
\$append/3 (L), 65	assert_union/2 (L), 33
\$asm/3, 52	asserta/1 (L), 33
\$assertf_alloc_t (L), 43	asserta/2 (L), 33
\$current_atom/2 (L), 36	asserti/2 (L), 33
\$current_functor/3 (L), 36	assertz/1 (L), 33
\$current_predicate/3 (L), 37	assertz/2 (L), 33
\$db_add_clref/7 (L), 43	atom/1 (B), 26
\$db_assert_fact/5 (L), 43	

atomic/1 (B), 27
 bagof/3 (L), 29
 break/0 (L), 38
 call/1 (P), 28
 call_ref/2 (L), 42
 call_ref/3 (L), 42
 clause/2 (L), 34
 clause/3 (L), 35
 compare/3 (B), 31
 compile/1 (L), 8
 compile/2 (L), 8
 compile/3 (L), 8
 compile/4 (L), 8
 conlength/2 (B), 28
 conlength/2 (L), 32
 consult/1 (L), 10
 consult/2 (L), 10
 count/1 (L), 63
 countpreds/1 (L), 64
 cputime/1 (B), 38
 current_atom/1 (L), 36
 current_functor/2 (L), 36
 current_predicate/2 (L), 37
 dcg/2 (L), 61
 debug/0 (L), 46
 debugging/0 (L), 46
 display/1 (L), 21
 erase/1 (L), 36
 et/1 (L), 59
 et_answers/2 (L), 60
 et_calls/2 (L), 60
 et_points/1 (L), 59
 et_remove/1 (L), 60
 et_star/1 (L), 59
 eval/2 (L), 24
 exp/2 (B), 25
 expand_term/2 (L), 62
 fail/0 (I), 26
 findall/3 (L), 29
 float/1 (I), 27
 floatc/3 (B), 24
 floor/2 (B), 24
 functor/3 (L), 27
 gennum/1 (L), 41
 gensym/2 (L), 42
 get/1 (B), 22
 get0/1 (B), 22
 globalset/1 (L), 41
 index/3 (D), 54
 instance/2 (L), 36
 integer/1 (I), 26
 is/2 (L), 24
 is_buffer/1 (B), 27
 keysort/2 (L), 31
 length/2 (L), 66
 listing/0 (L), 36
 listing/1 (L), 36
 load/1 (B), 9
 mode/3 (D), 53
 name/2 (B), 28
 nl/0 (B), 22
 nocount/1 (L), 63
 nodebug/0 (L), 46
 nodynload/2 (L), 39
 noet/1 (L), 60
 nonvar/1 (I), 26
 noprofile/0 (L), 64
 nospy/1 (L), 46
 not/1 (P), 26
 notime/1 (L), 63

number/1 (B), 27
 op/3 (L), 15, 38
 phrase/2 (L), 61
 portray_clause/2 (L), 22
 portray_term/2 (L), 22
 predicate_property/2 (L), 37
 print/1 (L), 21
 print_al/2 (L), 22
 print_ar/2 (L), 22
 prof_reset/1 (L), 64
 prof_stats/0 (L), 64
 prof_stats/1 (L), 65
 profile/0 (L), 64
 profiling/0 (L), 64
 put/1 (B), 22
 read/1 (B), 21
 real/1 (I), 26
 recorda/3 (L), 35
 recorded/3 (L), 35
 recordz/3 (L), 35
 repeat/0 (L), 26
 resetcount/1 (L), 64
 resettime/1 (L), 64
 restore/1 (B), 38
 retract/1 (L), 35
 save/1 (B), 38
 see/1 (B), 20
 seeing/1 (B), 20
 seen/0 (B), 20
 setof/3 (L), 29
 sin/2 (B), 25
 sort/2 (L), 31
 spy/1 (L), 46
 spy_preds/1 (L), 47
 square/2 (B), 25
 statistics/0 (B), 39
 statistics/2 (L), 39
 structure/1 (B), 27
 subsumes/2 (L), 66
 symtype/2 (B), 40
 syscall/3 (B), 40
 system/1 (B), 40
 tab/1 (B), 22
 tell/1 (B), 21
 telling/1 (B), 21
 term_expansion/2 (U), 62
 time/1 (L), 63
 time_preds/1 (L), 64
 told/0 (B), 21
 trace/1 (L), 44
 trace_preds/1 (L), 46
 trimbuff/3 (L), 32
 true/0 (I), 25
 undefined_pred/1 (L), 8
 untrace/1 (L), 46
 var/1 (I), 26
 write/1 (L), 21
 writename/1 (B), 21
 writeq/1 (L), 21
 writeqname/1 (B), 21

B A Note on Coding for Efficiency

The SB-Prolog system tends to favour programs that are relatively pure. Thus, for example, *asserts* tend to be quite expensive, encouraging the user to avoid them if possible. This section points out some syntactic constructs that lead to the generation of efficient code. These involve (i) avoiding the creation of backtrack points; and (ii) minimizing data movement between registers. Optimization of logic programs is an area of ongoing research, and we expect to enhance the capabilities of the system further in future versions.

B.1 Avoiding Creation of Backtrack Points

Since the creation of backtrack points is relatively expensive, program efficiency may be improved substantially by using constructs that avoid the creation of backtrack points where possible. The SB-Prolog compiler recognizes conditionals involving certain complementary inline tests, and generates code that does not create choice points for such cases. Two inline tests $p(t_1, \dots, t_n)$ and $q(t_1, \dots, t_n)$ are *complementary* if and only if $p(t_1, \dots, t_n) \equiv \text{not}(q(t_1, \dots, t_n))$. For example, the literals ' $X > Y$ ' and ' $X \leq Y$ ' are complementary. At this point, complementary tests are recognized as such only if their argument tuples are identical. The inline predicates that are treated in this manner, with their corresponding complementary literals, are shown in Table B.1. The syntactic constructs recognized

<i>Inline Test</i>	<i>Complementary Test</i>
$> / 2$	$\leq / 2$
$\leq / 2$	$> / 2$
$\geq / 2$	$< / 2$
$< / 2$	$\geq / 2$
$=: = / 2$	$= \setminus = / 2$
$= \setminus = / 2$	$=: = / 2$
$? = / 2$	$\setminus = / 2$
$\setminus = / 2$	$? = / 2$
$\text{var} / 1$	$\text{nonvar} / 1$
$\text{nonvar} / 1$	$\text{var} / 1$

Table 5: Complementary Tests Recognized by the Compiler

are:

(i) Disjuncts of the form

$$head(\dots): -(test(t_1, \dots, t_n), \dots); (not(test(t_1, \dots, t_n), \dots)).$$

or

$$head(\dots): -(test(t_1, \dots, t_n), \dots); ((comp_test(t_1, \dots, t_n), \dots)).$$

where *test* is one of the inline tests in the table above, and *comp_test* the corresponding complementary test (note that the arguments to *test* and *comp_test* have to be identical).

(ii) Conditionals of the form

$$head: -(test_1, \dots, test_n) - > True_Case; False_Case.$$

or

$$head: -(test_1; \dots; test_n) - > True_Case; False_Case.$$

where each *test_i* is an inline test, as mentioned in the table above.

The code generated for these cases involves a test and conditional branch, and no choice point is created. We expect future versions of the translator to recognize a wider class of complementary tests.

Notice that this discourages the use of explicit cuts. For example, whereas a choice point will be created for

```
part(M, [E|L], U1, U2) :-
  ((E =< M, !, U1 = [E|U1a], U2 = U2a) ;
  (U1 = U1a, U2 = [E|U2a])),
  part(M, L, U1a, U2a).
```

no choice point will be created for either

```
part(M, [E|L], U1, U2) :-
  (E =< M -->
  (U1 = [E|U1a], U2 = U2a) ;
  (U1 = U1a, U2 = [E|U2a])),
  part(M, L, U1a, U2a).
```

or

```
part(M, [E|L], U1, U2) :-  
  ((E =< M, U1 = [E|U1a], U2 = U2a) ;  
   (E > M, U1 = U1a, U2 = [E|U2a])),  
  part(M, L, U1a, U2a).
```

Thus, either of the two later versions will be more efficient than the version with the explicit cut (this is a design decision we have consciously made, in the hope of discouraging blatantly non-declarative code where efficient declarative code can be written).

B.2 Minimizing Data Movement Between Registers

Data movement between registers for parameter passing may be minimized by leaving variables in the same argument position wherever possible. Thus, the clause

$$p(X, Y) \text{ :- } p1(X, Y, 0).$$

is preferable to

$$p(X, Y) \text{ :- } p1(0, X, Y).$$

because the first definition leaves the variables X and Y in the same argument positions (first and second, respectively), while the second definition does not.

B.3 Processing All Arguments of a Term

It is often the case that we wish to process each of the arguments of a term in turn. For example, to decide whether a compound term is ground, we have to check that each of its arguments is ground. One possibility is to create a list of those arguments, and traverse the list processing each element. Using this approach, a predicate to check for groundness would be

```
ground(T) :- atomic(T).  
ground(T) :- structure(T), T =.. [_ | Args], groundargs(Args).  
groundargs([]).  
groundargs([A | ARest]) :- ground(A), groundargs(ARest).
```

This is not the most efficient way to process all the arguments of a term, because it involves the creation of intermediate lists, which is expensive both in space and time. A much better alternative is to use *arg/3* to index into the

term and retrieve arguments. Using this approach, the *ground/1* predicate above would be written as

```

ground(T) :- atomic(T).
ground(T) :- structure(T), functor(T, P, N), groundargs(1, N, T).
groundargs(M, N, T) :-
M =< N ->
(arg(M, T, A), ground(A), M1 is M + 1, groundargs(M1, N, T)) ;
true.

```

The second approach is likely to be more efficient than the first in SB-Prolog.

If the arguments of the term do not need to be processed in ascending order, then it is more efficient to process them in descending order using *arg/3* to access them. For example, the predicate for groundness checking could be written as

```

ground(T) :- atomic(T).
ground(T) :- structure(T), functor(T, P, N), groundargs(N, T).
groundargs(M, T) :-
M == 0 ->
true ;
(arg(M, T, A), ground(A), M1 is M - 1, groundargs(M1, T)).

```

This is even more efficient than the earlier version, because (i) *groundargs* needs to have one fewer parameter to be passed to it at each iteration; and (ii) testing “*M == 0*” is simpler and more efficient than checking “*M =< N*”, and takes fewer machine instructions.

B.4 Testing Unifiability

Often, it is necessary to check whether or not a term has a particular value. If we know that the term will be bound to a number, we can use the evaluable predicates *==/2* or *= \ =/2*, as explained earlier. For other values, it may often be cheaper, in the appropriate circumstances, to use the predicates *?=/2* or *\ =/2*. For example, consider a predicate *p/2* that calls *q/1* with its second argument if its first argument unifies with *a*, and *r/1* otherwise.

A naïve definition might be

```

p(a, X) :- !, q(X).
p(Y, X) :- r(X).

```

However, the call to *p/2* results in the (temporary) creation of a backtrack point. A solution that avoids this backtrack point creation is

```

p(Y, X) :- Y ?= a -> q(X) ; r(X).

```

Of course, if the argument order in $p/2$ could be reversed in this case, then data movement would be reduced even further (see above), and the code would be even more efficient:

```
p(X, Y) :- Y ?= a -> q(X) ; r(X).
```

C Adding Builtins to SB-Prolog

Adding a builtin involves writing the C code for the desired case and installing it into the simulator. The files in the directory `sim/builtin` contain the C code for the builtin predicates supported by the system. The following procedure is to be followed when adding a builtin to the system:

1. *Installing C Code:*

- (a) Go to the directory `sim/builtin`.
- (b) Look at the `#defines` in the file `builtin.h`, and choose a number *N1* (between 0 and 255) which is not in use to be the builtin number for the new builtin.
- (c) Add to the file `builtin.h` the line
- (d) The convention is that the code for builtin will be in a parameter-less procedure named `b_NEWBUILTIN`. Modify the file `init_branch.c` in the directory `sim/builtin` by adding these lines:

```
#define NEWBUILTIN N1
```

```
extern int b_NEWBUILTIN();
```

and

```
set_b_inst ( NEWBUILTIN, b_NEWBUILTIN );
```

in the appropriate places.

- (e) The builtins are compiled together into one object file, `builtin`. Update the file `Makefile` by appending the name of your object code file at the end of the line “`OBJS = ...`” and insert the appropriate commands to compile your C source file, e.g.:
- ```
OBJS = [... other file names ...] newbuiltin.o
:
newbuiltin.o: $(HS)
cc $(CFLAGS) newbuiltin.c
```

- (f) Execute the updated make file to create an updated object file *builtin*.
- (g) Go to the directory **sim** and execute *make* to install the new file **builtin**.

## 2. Installing Prolog Code:

Assume that the builtin predicate to be added is *newbuiltin/4*. The procedure for installing the Prolog code for this is as follows:

- (a) Go to the SB-Prolog system directory **lib/src**, where the Prolog source for the library routines is kept.
- (b) Each builtin definition is of the form

$$\text{pred}(\dots) \text{ :- } \text{'\_builtin'}(N).$$

where  $N$  is an integer, the builtin number of *pred*.

- (c) Create a Prolog source file *newbuiltin.P* (notice correspondence with the name of the predicate being defined) containing the definition

$$\text{newbuiltin}(A,B,C,D) \text{ :- } \text{'\_builtin'}(N1).$$

where  $N1$  is the builtin number of the predicate *newbuiltin*, obtained when installing the C code for the builtin (see above).

- (d) Compile this Prolog predicate, using the simulator and the *compile* predicate, into a file *newbuiltin* (notice correspondence with the name of the predicate being defined) in the SB-Prolog directory **lib**.