The XSB Programmer's Manual
Version 1.4.0

Konstantinos F. Sagonas
Terrance Swift
David S. Warren

Department of Computer Science
SUNY @ Stony Brook
Stony Brook, NY 11794-4400, U.S.A.

with contributions from
Steve Dawson
Prasad Rao

May 1994

Credits and Acknowledgements

The primary designers and developers of XSB were David S. Warren, Terrance Swift and Kostis Sagonas. [1] XSB has benefited greatly from the involvement of Steve Dawson who implemented *transformational indexing*, and of Prasad Rao who implemented *trie-based tabling* and the corresponding table access routines. We would also like to thank Kate Dvortsova for her help in several features of the system. Amanda Perez designed the logo.

The starting point of XSB was PSB-Prolog 2.0 by Jiyang Xu. PSB-Prolog in its turn was based on SB-Prolog, designed and written primarily by Saumya Debray, David S. Warren, and Jiyang Xu. Without all of their well-written code XSB would not have been possible. We have also borrowed material from the PSB-Prolog User's Manual [Xu 91] and from the SB-Prolog Version 3.0 User's Manual [Debr 88].

Credits are also due to Weidong Chen for his early work on Prolog clause indexing and Richard O'Keefe, who contributed the Prolog code for the Prolog reader and the C code for the tokenizer. Harald Schroepfer helped us with the Solaris port.

Finally, we would like to thank our significant others, Paula Pelletier, Amanda Perez, and Sofia Gourna for putting up with our alternating fatigue and enthusiasm, and our absences.

[1] The order of names here and anywhere else in the system or manual is arbitrary.

Contents

Chapter 1

Introduction

XSB is a research-oriented Logic Programming system for Unix[1]-based systems. Many of its components are based on PSB-Prolog [Xu 91] which itself is based on version 2.0 of SB-Prolog [Debr 88].

In addition to providing all the functionality of full Prolog, XSB contains several features not usually found in Logic Programming systems, including

- SLG resolution.
- A full HiLog implementation.
- Source Code Availability for portability and extensibility.

Though XSB can be used as a Prolog system, we avoid refering to XSB as such, because of the availability of SLG resolution and the handling of HiLog terms. These facilities, while seemingly simple, significantly extend its capabilities far beyond those of a Prolog system. We feel these capabilities justify viewing XSB as a new system for Logic Programming.

To begin to see some of the implications of SLG resolution, recall that Prolog is based on a depth-first backtracking search through the tree of SLD derivations. As such Prolog is susceptible to getting lost in the search of an unproductive portion of the search tree, and thus looping infinitely. SLG evaluation, available in XSB, can correctly evaluate many such logic programs. To take the simplest of examples, any query to the program:

[1]Unix is a trademark of AT&T.

```
:- table ancestor/2.

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
```

will terminate in XSB, since predicate `ancestor/2` is compiled as a tabled predicate; Prolog systems would go into an infinite loop. The user can declare that SLG resolution is to be used for a predicate by using `table` declarations, as here. Alternately, the `table_all` declaration can be used to direct the system to invoke a simple static analysis to decide what predicates to table. XSB extends SLG resolution to support evaluation of programs with stratified negation. This power to solve recursive queries has proven very useful in a number of areas, including deductive databases, language processing, meta-interpreters, and program analysis. For efficiency, we have implemented SLG at the abstract machine level so that tabled predicates will be executed with the speed of compiled Prolog. We finally note that for definite programs SLG resolution is similar to OLDT resolution [TS 86].

The second important extension in XSB is support of HiLog programming. HiLog allows a certain form of higher-order programming, in which predicate "symbols" can be variable or structured. For example, definition and execution of *generic predicates* like this generic transitive closure relation are allowed:

```
closure(R)(X,Y) :- R(X,Y).
closure(R)(X,Y) :- R(X,Z), closure(R)(Z,Y).
```

where `closure` is (syntactically) a second-order function which, given any relation `R`, returns its transitive closure `closure(R)`. For a detailed description of HiLog see [CKW 93, SW 94]. Support is provided for reading and writing HiLog terms, converting them to/from internal format as necessary (see section 4.2). Special meta-logical standard predicates (see section 6.5) are provided for inspection and handling of HiLog terms. Unlike versions of XSB prior to version 1.3.1 that did not provide any special compilation of HiLog predicates, the current version automatically provides *full compilation of HiLog*. As a result, most uses of HiLog execute at essentially the speed of compiled Prolog. For more information about the compilation scheme for HiLog employed in XSB see [SW 94].

HiLog can also be used with tabling, so that the program above can also be written as:

```
:- table closure(_)(_,_).

closure(R)(X,Y) :- R(X,Y).
closure(R)(X,Y) :- closure(R)(X,Z), R(Z,Y).
```

Source code is provided for the whole of XSB, including the engine and its supporting functions written in C, and the compiler, top-level interpreter and libraries written in Prolog.

One reason for making everything available is to allow users to port XSB to other UNIX platforms, as was done with its ancestor SB-Prolog. The system can be compiled using the GNU C compiler, and currently we have tested XSB on SUN-3 workstations, SUN-4/SPARCstations (running SunOS 4.x.x or Solaris 2.x), NeXT machines (running Mach), 80386/80486/80586 based machines running Linux, 386BSD, or NeXTSTEP, Silicon Graphics machines running IRIX, Hewlett-Packard 300/400 series machines running HP-UX, DECstations running Ultrix, Amiga machines running AmigaDOS, and IBM RS/6000 machines running AIX. Other ports are planned, particularly to Sequent Symmetry. Parties interested in doing other ports should contact us at the address in Chapter 13.

The other advantage of source code availability is that it allows users to extend the system themselves. The user can experiment with new builtins, implemenent alternate clause indexing, or even add new emulator instructions at a fraction of the effort involved in creating a system from scratch. Of course, while we encourage experimentation, users must support their own code in such cases.

XSB can also be used as a Prolog system, without SLG or HiLog. Indeed, while support of SB-Prolog at SUNY Stony Brook and at University of Arizona has ended, we intend to support XSB (to the extent we can) in the future. We hope that former users of SB-Prolog will switch to XSB and notice a more robust system with faster compilation.

As a Prolog system, there are several features of XSB which may be of interest to users.

- Object file compilation.
- Module system.
- Dynamic loading of Prolog and C code.
- Inference of symbol usage at compilation.

XSB, as its ancestor SB-Prolog, does not compile programs directly into memory but rather generates XWAM object code (called *byte code*) into files. The object files can be executed directly by the emulator or, as is more usual, loaded by the interpreter (which itself is a byte code file executed by the emulator). Indeed, the compiler and libraries of XSB system are all stored in byte code files, though source code is also included in the standard system distribution.

The module system of XSB is *file based* (one module per file), *atom based* (but all non predicate symbols are normally assumed to be global to all modules), and *flat* (modules cannot be nested). Though the user is not obliged to use the module system, the dynamic loader, the use inferencer, and the foreign language interface are based upon it.

The dynamic loader lazily loads files containing predicates when they are first called, which can be useful for reducing program space and start-up time. (Files can also be loaded explicitly by the use of standard predicates `consult/[1,2]`). The use inferencer indicates at compile-time whether symbols in a module are unused (not called by an exported predicate) or undefined.

While these features are explained in detail below, this document does not intend to be a tutorial for using the Prolog language; rather it focuses on the extensions of XSB to Prolog systems. For users new to Prolog, a textbook such as [Brat 86, CM 84, MW 88, SS 86] should be used along with this manual.

It should be mentioned that we adopt some standard notational conventions, such as the name/arity convention for describing predicates and functors, `+` to denote input arguments, `-` to denote output arguments, `?` for arguments that may be either input or output and `#` for arguments that are both input and output (can be changed by the procedure).

Finally, we note that XSB is under continous development, and this document —intended to be the user manual— reflects the current status (Version 1.4.0) of our system. In future versions, we intend to provide a *bulk-load* facility to load large data files into tables, and some further tabling optimizations, including better space reclamation, and a more fully optimized `table_copy/2` predicate. While we have taken a great deal of effort to create a robust and efficient system, we would like to emphasize that the system is also experimental. Hence, some of what is described is subject to change in future releases, though we will try to be as upward-compatible as possible. We also would like to warn the users that the use of undocumented features is not supported, and at the users' own risk.

Chapter 2

Getting Started with XSB

This section is written for those who are eager to start using XSB. Most of the material in this chapter will be covered more fully in later sections, along with many options that are ignored for now.

2.1 Installing XSB

XSB can be installed by the following procedure.

1. Make sure that after you have obtained XSB by anonymous ftp (using the `binary` option) you have uncompressed it by following the instructions found in the file `README`.
2. Decide in which directory in your file system you want to install XSB and copy or move XSB there. Note that this directory *cannot* be moved (or renamed) after the installation, or a new installation is needed.

   In the rest of this chapter, let us use `$XSB_DIR` to refer to this directory.

   Note the original subdirectory structure must be maintained, namely, the directory `$XSB_DIR` should at least contain the following four subdirectories: `emu`, `syslib`, `cmplib`, and `lib`.
3. Run the C-shell script `makeall` in the directory `$XSB_DIR`. [1] It will prompt for a machine type. Type in `sun` if you are using a SUN system (either SUN3 or SPARC) running SunOS 4.x.x, `solaris` if you are using a SPARCstation running Solaris 2.x,

[1] This script cannot be run for Amiga machines running AmigaDOS and the whole process has to be done manually according to the instructions found in the `README` file.

`NeXT` if you are using a NeXT machine or a 80x86 machine running NeXTSTEP, `linux` if you want to install XSB on a 80386/80486/80586 based machine running Linux or some version of 386BSD, `SGI` for a Silicon Graphics machine running IRIX, `HP300` for a Hewlett-Packard 300/400 series machine running HP-UX, `DECstation` for a DEC station running Ultrix, or `IBM` for an IBM RS/6000 machine running AIX. It will also prompt for the *name* of the C compiler to be used. Type `cc` if the name of the C compiler to be used is cc, or type `gcc` if the name of the C compiler to be used is gcc. If for some reason something goes wrong while installing the system consult chapter 3, where you can find a more detailed description of the installation options.

2.2 Invoking XSB

XSB can be invoked by the command:

```
$XSB_DIR/emu/xsb -i
```

and you will find yourself in the top level interpreter.

You may want to make an alias such as `xsb` to the command for convenience.

Most of the "standard" Prolog predicates are supported by XSB, so those of you who consider yourselves champion entomologists, can try to test them for bugs now. Details are in Chapter 6.

2.3 Compiling XSB programs

All source programs should be in files whose names have the suffix `.P`. One of the ways to compile a program from a file in the current directory and load it into memory, is to type the query:

```
[my_file].
```

where `my_file` is the name of the file, or preferably, the name of the module (obtained from the file name by deleting the suffix `.P`). To find more about the module system of XSB see Section 3.4.

If you are eccentric (or you don't know how to use an editor) you can also compile and load predicates input directly from the terminal by using the command:

```
[user].
```

A `CTRL-d` or the atom `end_of_file` followed by a period terminates the input stream.

2.4 Sample XSB Programs

If for some reason you don't feel like writing your own XSB programs, there are several sample XSB programs in the directory: `$XSB_DIR/examples`. All contain source code.

The entry predicates of all the programs in that directory are given the names `demo/0` (which prints out results) and `go/0` (which does not print results). Hence, a sample session might look like (the actual times shown below may vary and some extra information is given using comments after the % character):

```
my_favourite_prompt> cd $XSB_DIR/examples
my_favourite_prompt> ../emu/xsb -i -s
XSB Version 1.3.0 (93/9/13)
[sequential, single word, optimal mode]
| ?- [prof_lib].      % loads into the system predicate measure/1
[prof_lib loaded]

yes
| ?- [queens].
[queens loaded]

yes.
| ?- demo.
..........           % output of 8-queens problem here

yes.
| ?- measure(demo).
..........           % same junk printed here
Time used: 2.409 sec
== Statistics ===
Permanent space: 131752 in use, 384K allocated.
Stack space allocated: 64K+64K+768K+768K+8K
Stacks in use : global 164, local 88, trail 48, cp 220
SLG Stacks in use : table 0, tab_heap 0 opentables 0
num unwinds: 731
Max stack used: global 66152, local 500, trail 420, cp 2780
TIME: cputime: 2.41*1 sec, elapsetime 8.02 sec

yes
| ?- [map].          % This will generate some warnings but don't worry
++Warning: demo/0 (type 2) was defined in another module!
```

```
++Warning: go/0 (type 2) was defined in another module!
[map loaded]

yes.
| ?- demo.
..........          % output of map coloring problem here

yes.
| ?- measure(go).   % Don't give me all this junk again!
Statistics is reset.
Time used: 0.629  sec
== Statistics ===
Permanent space: 132328 in use, 384K allocated.
Stack space allocated: 64K+64K+768K+768K+8K
Stacks in use : global 164, local 88, trail 48, cp 220
SLG Stacks in use : table 0, tab_heap 0 opentables 0
num unwinds: 708
Max stack used: global 66432, local 576, trail 612, cp 2216
TIME: cputime: 0.63*1 sec, elapsetime 0.68 sec

yes
| ?- halt.          % I had enough !!!

End XSB (cputime 7.52s, elapsetime 125.69s)
my_favourite_prompt>
```

2.5 Exiting XSB

If you want to exit XSB, issue the command `halt.` or simply type `CTRL-d` at the XSB prompt. To exit XSB while it is executing queries, strike `CTRL-c` a number of times.

Chapter 3

System Description

3.1 Options for Installing XSB

The distributed version of XSB that you received may not include all the object code files, in which case they must be created from the source files. An installation procedure is needed even when all the object code files are present but the entire system has been moved to a different place in your file system (which is certainly the case the first time you install the system). This is due to the fact that the emulator makes use of the information of the installed directory `$XSB_DIR` in order to find its libraries.

If all the object files in the `emu` directory are present, all you need to do is to edit the make file `$XSB_DIR/emu/Makefile` in order to modify the macro `DIR` so that it contains the absolute pathname of the directory (`$XSB_DIR`) where XSB is to be installed, and then run this makefile.

When an object file is missing from any of the directories, you can normally run the makefile in that directory to restore it. However, to restore an object file in the directories `syslib` and `cmplib`, one needs to have another Prolog compiler accessible (or even XSB), simply because the XSB compiler uses most of the files in these two directories and hence cannot be used when some of them are missing. For this reason, distributed versions normally include all the object files in `syslib` and `cmplib`.

The simplest way to install XSB is to run the C-shell script `makeall` in the directory where the system resides. [1] It will prompt for a machine type. Type in `sun` if you are using a SUN3/SUN4/SPARC system running SunOS 4.x.x, `solaris` if you are using a SPARCsta-

[1]This script cannot be run for Amiga machines running AmigaDOS and the whole process has to be done manually according to the instructions found in the `README` file.

tion running Solaris 2.x, `NeXT` if you are using a NeXT machine (running Mach) or a 80x86 machine running NeXTSTEP, `linux` if you want to install XSB on a 80386/80486/80586 based machine running Linux or some version of 386BSD, `SGI` for a Silicon Graphics machine running IRIX, `HP300` for a Hewlett-Packard 300/400 series machine running HP-UX, `DECstation` for a DEC station running Ultrix, or `IBM` for an IBM RS/6000 machine running AIX. It will also prompt for the *name* of the C compiler to be used. Type `cc` if the name of the C compiler to be used is cc, or type `gcc` if the name of the C compiler to be used is gcc. Note that it does not ask for the type of the C compiler but only for its name. `makeall` checks for a particular object file in each of the three Prolog source/object directories and issues `make` commands when that file is not present. If for some reason that file is present, `makeall` leaves that directory as it is.

Besides the directory where XSB resides, the installation options currently supported for XSB are

Type of Machine. The type of the machine that is used should be known to the system at installation time, so that it may be compiled in accordance with the idiosyncracies of the host machine. Furthermore, the type of the machine may limit the available features of XSB. For example, currently we provide a foreign language interface for SUN machines running SunOS 4.x.x or Solaris 2.x, and 80386/80486 based machines running Linux but not for NeXT, Silicon Graphics, or HP-300 machines.

The type of the machine that is used is either set manually by modifying the file `emu/Makefile` or is set automatically by the system if the system is installed by running the `makeall` script. In either case one of the following definitions must appear in the makefile:

```
MACHINE = SUN
MACHINE = SOLARIS
MACHINE = NeXT
MACHINE = linux
MACHINE = SGI
MACHINE = HP300
MACHINE = DECstation
MACHINE = AMIGA
MACHINE = IBM
```

Optimal vs. Debugging mode. There is another option that you can set by one of the following macro definitions in the makefile:

```
DEBUGOPTION = -O[4|2] -DTRACING
DEBUGOPTION = -g -DDEBUG -DTRACING
```

As a matter of fact, XSB can be compiled with some subset of the above definitions, but we suggest those above. The first one is recommended for normal XSB users, while the use of the second one is recommended for XSB developers, or in cases when for some reason the C compiler cannot optimize the files. [2]

Choice of the C Compiler. The developers of XSB have tried to write most of the emulator in fairly standard (but not always ANSI) C, so users should not have to change any source code. To make the emulator directory one needs a C compiler available. We recommend the use of the Gnu C Compiler. [3] In any case, according to the *naming conventions* for C compilers in your system, one line like the following ones should appear in the `Makefile`:

```
CC = cc
CC = gcc
```

Choice of the Tabling Data Structures. From version 1.4.0 on, users can choose between alternative tabling data structures, namely hash-based and trie-based tabling. We suggest the use of the trie-based tabling, that, among other things, eliminates the dependence of tabling performance on the call and return hash sizes, and allocates space for tables dynamically. However, users can still experiment with hash-based tabling. So, one of the following two lines should appear in the `Makefile`:

```
TABLE_OPTION = -DTRIE_BASED_CODE
TABLE_OPTION =
```

We warn the user that when making the emulator with the `"-O*"` options the temporary space available to the C compiler for intermediate files may not be sufficient. For example on one of our SPARCstations that had very little `/tmp` space the `"-O4"` option could not be used for the compilation of file `main.c`, without changing the default `tmp` directory and increasing the swap space. In your system depending on the amount of `/tmp` space of your machine there may be other files or you may not have any problems at all. If you are using the SUN C compiler, and have disk space in one of your directories, say `dir`, add the following option to the entries of the files that cannot be compiled:

```
-temp=dir
```

If you are using the Gnu C compiler, consult its manual pages to find out how you can change the default `tmp` directory or how you can use pipes to avoid the use of temporary space during

[2] It has been reported to us that some of the DECstation's and HP's C compilers cannot compile XSB with the `-c -O*` combination of compiler options.

[3] As a matter of fact, in some of the above mentioned machines XSB cannot be compiled without the Gnu C compiler.

compiling. Usually changing the default directory can be done by declaring/modifying the `TMPDIR` environment variable as follows:

```
setenv TMPDIR dir
```

3.2 Entering and Exiting XSB

After the system has been installed, the emulator's executable code appears in the file:

```
$XSB_DIR/emu/$EXECUTABLE
```

where `EXECUTABLE` contains the name of the emulator's executable file. Unless the user has modified the value of this macro in the `Makefile` this file name is `xsb` so for convenience we will use this name for the rest of this section.

So, to invoke the XSB's top level interpreter (the usual way of using XSB), simply issue the command:

```
$XSB_DIR/emu/xsb -i
```

In general, the emulator expects the name of a file containing byte code to execute. The XSB's top level interpreter resides in the file `$XSB_DIR/syslib/loader.O`. (This file contains only the start-up code, mainly the dynamic loader for the interpreter; the other parts of the interpreter are in several different files and will be loaded automatically when and if they are needed.) The command `$XSB_DIR/emu/xsb -i` is equivalent to the command:

```
$XSB_DIR/emu/xsb $XSB_DIR/syslib/loader.O
```

To exit XSB, issue the command `halt.` or simply type `CTRL-d` at the XSB prompt. Another way to accomplish the same result is to issue the prolog command `end_of_file`. To quit XSB while it is executing some queries, strike `CTRL-c` a number of times.

You can also run the raw emulator directly with a byte code file. The byte code file should be generated from a source program that contains at least one command (a clause started by "`:-`" which is not a directive). Otherwise the emulator does null operation, terminating at once.

Since the emulator itself does not contain a dynamic loader, a program directly executed by the emulator must be *self-contained*, that is, all the predicates must be defined in the program. In particular, because most common standard predicates are implemented at the Prolog level, they cannot be used in the program unless their definitions are copied in. You must therefore use the primitive predicates that the emulator can interpret directly instead. All such primitive predicates are listed in Appendix B.

3.3 The System and its Directories

The XSB system, when installed, resides in a single directory that contains the following six subdirectories:

1. `emu`
2. `syslib`
3. `cmplib`
4. `lib`
5. `examples`
6. `manual`

For convenience, we shall use the shorthand `$XSB_DIR` to refer to the directory where XSB is installed.

The directory `emu` contains the source and object code for the XSB emulator, which is written in C.

The directories `syslib`, `cmplib` and `lib` contain source and object code for the basic Prolog libraries, the compiler, and the extended Prolog libraries, respectively. All the source programs are written in XSB, and all object (byte code) files contain WAM instructions that can be executed by the emulator. These byte-coded instructions are machine-independent, so usually no installation procedure is needed for the byte code files.

The directory `manual` contains this manual in LaTeX, dvi and Postscipt format, and the directory `examples` contains some sample programs to demonstrate various features of XSB.

3.4 The Module system of XSB

XSB has been designed with the intention to be used as a module oriented Prolog system. This is a small step towards *logic programming "in the large"* that facilitates large programs or projects to be put together from components which can be developed, compiled and tested separately. Also module systems enforce the *principle of information hiding* and can provide a basis for *data abstraction*.

The module system of XSB, unlike the module systems of most other Prolog systems is *atom-based*. Without getting into too many technical details, the main difference between atom-based module systems and predicate-based ones is that in an atom-based module system *any symbol in a module can be imported, exported or be a local symbol* as opposed to the predicate-based ones where this can be done only for predicate symbols.

Usually the following three files are associated with a particular module:

- A single *source* file, whose name is the module name plus the suffix "`.P`".
- An optional *header* file, whose name is the module name plus the suffix "`.H`".
- An *object (byte-code)* file, whose name consists of the module name plus the suffix "`.O`".

The header file is normally used to contain declarations and directives while the source file usually contains the actual definitions of the predicates defined in that module. The module hierarchy of XSB is therefore *flat* — nested modules are not possible.

In order for a file to be a module, it should contain one or more *export declarations*, which specify that a set of symbols appearing in that module is visible and therefore can be used by any other module. A module can also contain *local declarations*, which specify that a set of symbols are *visible by this module only*, and therefore cannot be accessed by any other module. Any file (either module or not) may also contain *import declarations*, which allow symbols defined in and exported by other modules to be used in the current module. We note that only exported symbols can be imported; for example importing a local symbol will cause an **`environment conflict error`**.

Export, local, and import declarations can appear anywhere in the source (or preferably in the header) files and have the following forms:

`:- export` sym_1, ..., sym_l.

`:- local` sym_1, ..., sym_m.

`:- import` sym_1, ..., sym_n `from` *module*.

where sym_i has the form $functor/arity$.

If the user does not want to use modules, he can simply bypass the module system by not supplying any export declarations. Such exportless files (non-modules) will be loaded into the module `usermod`, which is the working module of the XSB interpreter.

Currently the module name is stored in its byte code file, which means that if the byte code file is renamed, the module name is not altered, and hence may cause confusion to the user and/or the system. So, it is advisable that the user not rename byte code files generated for modules by the XSB compiler. However, byte code files generated for non-modules can be safely renamed. We will try to fix the problem described above in future releases.

In order to understand the semantics of modules, the user should keep in mind that in a module oriented system, the name of each symbol is identified as if it were prefixed with its module name, hence two symbols of the same *functor/arity* but different module prefixes are distinct symbols. Currently the following set of rules is used to determine the module prefix of a symbol:

- Every predicate symbol appearing in a module (i.e. that appears as the head of some clause) is assumed to be *local* to that module unless it is declared otherwise (via an export or import declaration). Symbols that are local to the module that defines them *cannot* be used in other modules.
- Every other symbol (essentially function symbols) in a module is assumed to be *global* (its module prefix is `usermod`) unless declared otherwise.
- If a symbol is imported from another module (via an explicit import declaration), the module prefix of the symbol is the module it is imported from; any other symbol takes the module where the symbol occurs as its module prefix.
- The XSB interpreter is entered with `usermod` as its working module.
- Symbols that are either defined in non-modules loaded into the system or that are dynamically created (by the use of standard predicates such as `read/1, functor/3, '=..'/2`, etc) have the same module as the current working module (which is always `usermod` in the current implementation, because we do not allow the user to change the working module). This feature should not be a major problem for the average user.

The following facts about the module system of XSB may not be immediately obvious:

- If users want to use a symbol from another module, they must explicitly import it otherwise the two symbols are different even if they are of the same *functor/arity* form.
- A module can only export symbols that are defined in that module. As a consequence, a module *cannot* export symbols that are imported from other modules. This happens

because an `import` declaration is just a request for permission to use a symbol from a module where its definition and an `export` declaration appear.

- The implicit module for a particular symbol appearing in a module must be uniquely determined. As a consequence, a symbol of a specific *functor/arity cannot* be declared as both exported and local, or both exported and imported from another module, or declared to be imported from more than one module, etc. These types of environment conflicts are detected at compile-time and result in aborting the compilation process.

- It is an error to import a symbol from a module that does not export it. This error is *not* detected at compile-time but at run-time when a call to that symbol is made. If the symbol is defined in, but not exported from the module, an environment conflict error will take place. If the symbol is not defined in that module an undefined predicate/function error will be be reported to the user.

- In the current implementation, at any time only one symbol of a specific *functor/arity* form can appear in a module. As an immediate consequence of this fact, only one *functor/arity* symbol can be loaded in the current working module (`usermod`). An attempt to load a module that redefines that symbol results in a warning to the user and the newly loaded symbol *overrides* the definition of the previously loaded one.

3.5 The Dynamic Loader and its Search Path

One of XSB's differences with other Prolog systems, is its dynamic (or automatic) loader. In XSB, the loading of Prolog libraries (including the XSB compiler itself) and user modules is delayed until they are actually needed, saving program space for large Prolog applications. The delay in the loading is done automatically, unlike systems, where it must be explicitly specified for non system libraries.

When a predicate which is imported from another module (see section 3.4) is called during execution, and the module is not yet loaded into the system, the dynamic loader is invoked automatically. The default action of the dynamic loader is to search for the byte code file of the module (see section 3.4) in the system library directories first (in the order `lib, syslib`, and `cmplib`), and finally in the current working directory. If the module is found in one of these directories, then it will be loaded (*on a first-found basis*). Otherwise, an error message reporting that the module was not found will be displayed on the current output stream.

Most of the system modules and the compiler are loaded this way. Because of dynamic loading, the time it takes to compile a file is longer than usual when the compiler is invoked for the first time.

3.5.1 Changing the default Search Path

Users are allowed to supply their own library directories as well as overriding the default search path of the dynamic loader. User-supplied library directories are searched by the dynamic loader *before* searching the default library directories.

The default search path of the dynamic loader can easily be changed by having a file named `.xsbrc.P` in the user's home directory. The `.xsbrc.P` file, which is automatically consulted by the XSB interpreter, must be a module (see section 3.4), and it may look like the following:

```
:- export library_directory/1.

library_directory('./').
library_directory('~/').
library_directory('~my_friend/').
library_directory('/usr/lib/sbprolog/').
```

Note that all directories specified in predicate `library_directory/1` must end with a `'/'`. After loading the module of the above example, the current working directory is searched first (while the default action was to search it last). Also, XSB's system library directories (`lib, syslib`, and `cmplib`), will now be searched *after* searching the user's and `my_friend`'s home directories and the `"/usr/lib/sbprolog/"` directory.

Though a `.xsbrc.P` module may in future releases be used for other purposes as well as changing the default search path of the dynamic loader, in the current version it *must* define and export the predicate `library_directory/1`.

We should emphasize that in the presense of a `.xsbrc.P` file *it is the user's responsibility to ensure that module name clashes with XSB's modules in system library directories are avoided.* Such name clashes can cause the system to behave very strangely (different modules to be loaded into the run time environment instead of the intended ones). The list of module names in XSB's system library directories can be found in appendix C.

3.5.2 Dynamically loading predicates in the interpreter

The usual way of loading a module into the run time environment is by consulting it (see section 3.8). However, the standard predicate `import PredList from Module` (where `PredList` can either be a Prolog list or a comma list), not only can be used as a directive in a source module (see section 3.4), but it can also be entered as a query to import predicates into the run time environment of the interpreter.

A sample session for compiling, dynamically loading and querying a user defined module named **quick_sort** (for this example we assume that it is found in the current working directory), containing the definitions of a **concat/3** and a quick sort predicate **qsort/2** (which must be exported) may look like:

```
| ?- compile(quick_sort).
[Compiling ./quick_sort]
[quick_sort compiled, cpu time used: 1.439 seconds]

yes
| ?- import concat/3, qsort/2 from quick_sort.

yes
| ?- concat([1,3], [2], L), qsort(L, S).

L = [1,3,2]
S = [1,2,3]

yes.
```

The standard predicate **import/1** does not load the module containing the imported predicates, but simply informs the system where it can find the definition of the predicate when (and if) a call to this predicate is encountered.

3.6 Command Line Arguments

There are several command line options for the emulator. The general synopsis is:

```
xsb [-c size] [-m size] [-o size] [-u size] [-x size]
    [-D path] [-s] [-d] [-t] [-T]  -i | byte_code_file
```

The order in which these options appear makes no difference.

-i Brings up the XSB interpreter. This is the normal use.

-c *size* Allocates *size* Kbytes of space to the trail/choice-point stack area. Default: 768 Kbytes. The trail stack grows upward from the bottom of the region, and the choice point stack grows downward from the top of the region.

-m *size* Allocates *size* Kbytes of space to the local/global stack area. The global stack grows upward from the bottom of the region, and the local stack grows downward from the top of the region. Default: 768 Kbytes.

`-o` *size* Allocates *size* Kbytes of space to the opentable stack area. Default 64 Kbytes. (This option should rarely need to be used).

`-u` *size* Allocates *size* Kbytes of space to the unification (and table copy) stack. Default 64 Kbytes. (This option should rarely need to be used).

`-x` *size* Allocates *size* Kbytes of space to the table area. Default: 768 Kbytes.

`-D` *path* changes the definition of `XSB_DIR` (which is set at installation time) to *path*. This option can be used when the system has been moved to a different place in your file system but not re-installed. It is also useful when you have developed your own libraries and want to use them as the default system libraries.

`-s` Maintains information on the size of program stacks for the predicate `statistics/0`. This option may be expected to slow execution by around 10%. Default: off.

`-d` Produces a disassembled dump of `byte_code_file` to `stdout` and exits.

`-t` Traces through code at XWAM instruction level. This option is for internal debugging and is not fully supported. It is also not available when the system is being used at the non-debug mode (see Section 3.1).

`-T` Generates a trace at entry to each called routine. This option is not available when the system is being used at the non-tracing mode (see Section 3.1).

As an example, a program which used more heap and table space than common might be run with the command.

```
xsb -m 2000 -x 2000 -i
```

3.7 Memory Management

The developers of XSB and of SB-Prolog before it have gone to great effort to provide a useful, robust programming system. However, we have not been able to include many features that would be extremely useful to the system. Foremost among these is better memory management for the program stacks and table area.

As can be seen from the last section, the user may sometimes be required to guess the maximum amount of space needed by various areas of the system. This is because the system presently has no means of automatically expanding program stacks. We hope to fix this deficiency in the future. In the meantime, the `'-s'` option is provided for viewing the maximum amount needed for each area, and the overflow interrupts have been tuned to indicate which area has overflowed.

Unlike the stacks, the program area (the area into which the code is loaded) is dynamically expanded as needed, and the area occupied by dynamic code (created using `assert/1`) is reclaimed when that code is retracted.

3.8 Compiling and Consulting

In XSB there is no distinction between compiled and interpreted code. Though XSB compiles a file when consulting it, this section mainly describes the actions of standard predicates `consult/[1,2]` and `reconsult/[1,2]`. For a detailed description of the XSB compiler and predicates `compile/[1,2]` see section 3.9.

Predicates `consult/[1,2]` are simply aliases for standard predicates `reconsult/[1,2]`. So, we restrict our attention to describing predicates `consult/[1,2]`; the same things hold for `reconsult/[1,2]`. Though `consult` comes in many flavors, the most general form of `consult/2` is:

```
consult(+FileList, +CompilerOptionList)
```

and at the time of the call both of its arguments should be sufficiently instantiated (ground). `FileList` is a list of filenames or module names (see section 3.4) and `CompilerOptionList` is a list of options that are to be passed to the compiler when (and if) it should be invoked. For a detailed description of the format and the options that can appear in this list see section 3.9.

If the user wants to consult one module (file) only, she can provide an atom instead of a list for the first argument of `consult/2`. Furthermore, if there isn't any need for special compilation options the following two forms:

```
[FileName].
consult(FileName).
```

are just notational shorthands for the form:

```
consult(FileName, []).
```

Consulting a module (file) generally consists of the following five steps which are analyzed in detail in the next paragraphs.

1. determine the module to be consulted.
2. if necessary, compile the module using predicate `compile/2` with the options specified.
3. load the object code of the module into memory.
4. import all the exported predicates of that module to the current working module (`usermod`).
5. execute any queries that the module may contain.

The first step is straightforward when `FileName` contains a slash (`'/'`) (i.e. is an absolute file name), since the corresponding file is uniquely determined. However, we allow the user to supply a module name (file name without any slash). In such a case, the source file name corresponding to that module is obtained from the given module name by concatenating the extension `.P` (or `.c` for foreign modules) and a directory prefix. The directory prefix is a directory in the dynamic loader path (see section 3.5) where the source file exists. Note that since these directories are searched in a predetermined order (see section 3.5) if a module with the same name appears in more than one of these directories, the one consulted will always be the first encountered.

The second step will be preformed depending on the creation time of the source file (`*.P`) and the object file (`*.O`). When the source file is newer than the object file, `consult/[1,2]` recompiles the module. Clearly, if no object file exists, the file will be compiled. A minor feature of the current version is that it does not check the creation time of the module's header file (`*.H`), so the module will not be recompiled if there is a change in its header file only. We intend to take care of this feature in the next release of our system. Currently (Version 1.4.0), a foreign language module is compiled when at least one of files `*.c` or `*.H` has been changed from the time the corresponding object files have been created.

The third step is pretty self-explanatory; the byte-code file that corresponds to this module (file) is loaded into XSB's database, after first deleting any any previous versions of the predicates that are defined in that module. In Version 1.4.0, we provide a `multifile/1` declaration for dynamically loaded clauses only (see the library predicate `load_dyn/1` in section 10.2). The program space occupied by the previous versions of predicates in the database is reclaimed.

After loading the module all exported predicates of that module are imported into the current environment (the current current working module `usermod`). For non-modules (see section 3.4), all predicates are imported into the current current working module.

Finally if there are any queries, that is any terms with principal functor `':-'/1` that are not directives like the ones described in section 3.10, then these are executed in the order that they are encountered.

3.9 The Compiler

The XSB compiler translates XSB source files into byte-code object files. It is written entirely in Prolog. Both the sources and the byte code for the compiler can be found in the XSB system directory `cmplib`.

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

3.9.1 Invoking the Compiler

The compiler is invoked directly at the interpreter level (or in a program) through the Prolog predicates `compile/[1,2]`.

The general forms of predicate `compile/2` are:

```
compile(+File, +OptionList)
compile(+FileList, +OptionList)
```

and at the time of the call both of its arguments should be sufficiently instantiated (ground).

The second form allows the user to supply a proper list of file names as the parameter for `compile/[1,2]`. In this case the compiler will compile all the files in `FileList` with the same set of compiler options as the ones specified in the `OptionList`.

```
| ?- compile(Files).
```

is just a notational shorthand for the query:

```
| ?- compile(Files, []).
```

The standard predicate `consult/[1,2]` calls `compile/1` (if necessary). Argument `File` can be any syntactically valid Unix file name (in the form of a Prolog atom), but the user can also supply a module name.

The list of compiler options `OptionList`, if specified, should be a proper Prolog list, i.e. a term of the form:

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

where $option_i$ is one of the options described in section 3.9.2.

The source file name corresponding to that module is obtained from the given module name by concatenating the extension `.P` (or `.c`) and a directory prefix. The directory prefix is a directory in the dynamic loader path (see section 3.5) where the source file exists. Note that these directories are searched in a predetermined order (see section 3.5), so if a module with the same name appears in more than one of these directories, the compiler will always compile the first one it encounters. In such a case, the user should provide an absolute path name to compile the intended module.

If `File` contains no extension, the extension `.P` (or `.c`) is concatenated and an attempt to compile the file `File.P` (or `File.c`) is made before trying compiling the file with name `File`.

To avoid ambiguity, use of the extension `.P` for Prolog source file names is recommended. Optionally, users can also provide a header file for a module (denoted by the module name suffixed by `.H`). In such a case, the XSB compiler will first read the header file if it exists and then the source file. Currently the compiler makes no special treatment of header files. They are simply included in the beginning of the corresponding source files, and code can, in principle, be placed in either. In future versions the header files may be used to check interfaces across modules, hence it is a good programming practice to put only declarations into the header files.

The result of the compilation (XWAM instructions encoded as bytes) is stored in an object file (⟨filename⟩.O) but `compile/[1,2]` does *not* load the object file it creates. (The standard predicates `consult/[1,2]` and `reconsult/[1,2]` both recompile the source file, if needed, and load the object file into the system.) The object file created is always written into the directory where the source file resides (the user should therefore have write permission in that directory).

If desired, when compiling a module (file), clauses and directives can be transformed as they are read. This is indeed the case for definite clause grammar rules (see Chapter 8), but it can be done for clauses of any form by providing a definition for predicate `term_expansion/2` (see section 8.3).

Predicates `compile/[1,2]` can also be used to compile foreign language modules. In this case, the names of the source files should have the extension `.c`. A header file (with extension `.H`) *must* be present for a foreign language module (see Chapter 11).

3.9.2 Compiler Options

The following options are currently recognized by the compiler:

`optimize` When specified, the compiler tries to optimize the object code. In general, the byte-code generated will be considerably faster than the one generated in the absence of this option. However, due to the nature of the optimizations, the user may not be able to trace all calls to predicates in the program. Also the Prolog code should be *static*. In other words, the user is *not* allowed to alter the entry point of these compiled predicates by asserting new clauses. As expected, the compilation phase will also be longer. For all these reasons, the use of the `optimize` option may not be suitable for the development phase, but it is certainly recommended once the code has been debugged.

`sysmod` Mainly used when compiling system modules. If specified, all the standard predicates listed in Appendix A are not automatically available for use. Instead they must be explicitly imported from the appropriate system module. On the other hand, when compiling with this option all XSB primitive predicates (see Appendix B) are automatically available for use.

`table_all` When specified, it has the same effect as the synonymous directive described in section 3.10. That is, some predicates whose complete answer set may be unobtainable under Prolog's evaluation (SLD) are compiled as tabled (SLG evaluation is used).

`spec_repr` When specified, the compiler performs specialisation of the partially instantiated calls by replacing their selected clauses with the representative of these clauses, i.e. it performs *folding* whenever possible. We note the code replacement operation is not always sound, i.e. there are cases when the original and the residual program are not computationally equivalent. The compiler checks for sufficient (but not necessary) conditions that guarantee computational equivalence, and if these conditions are not met, specialisation is not performed for the violating calls.

`spec_off` When specified, the compiler does not perform specialisation of partially instantiated calls.

`unfold_off` When specified, the singleton sets optimisations are not performed during specialisation. This option is necessary for the specialisation of `table` declarations that select only a single chain rule of the predicate.

`spec_dump` Generates a `module.spec` file, containing the result of specialising partially instantiated calls to predicates defined in the `module` under compilation. The result is in Prolog source code form.

`ti_dump` Generates a `module.ti` file containing the result of applying transformational indexing to predicates defined in the `module` under compilation. The result is in Prolog source code form. See page 30 for more information on transformational indexing.

`ti_long_names` Used in conjunction with `ti_dump`, generates names for predicates created by transformational indexing that reflect the clause head factoring done by the transformation.

asm_dump Generates a textual representation of XWAM assembly code and writes it into the file `module.A` where `module` is the name of the module (file) being compiled.

WARNING: This option was created for compiler debugging and is not intended for general use. There are cases where compiling a module with the **asm_dump** option on may cause the compiler to generate an incorrect `.A` and `.O` file. In such cases, the user can see the XWAM instructions that are generated for a module by compiling the module as usual and then using the `-d module.O` command-line option of the XSB emulator (see Section 3.6).

`init_var_off` When specified, the compiler will only give a warning (instead of an error) upon finding that a potentially uninitialized variable is being used. *Potentially uninitialized variables* are variables that appear in only one branch of an **or** or an **if-then-else** goal in the body, and, furthermore, are used after that goal. ... There are cases where the variable is always initialized after the **or** or the **if-then-else** goal, because the execution cannot continue through the path of the branch that does not initialize the variable. In these cases, the `init_var_off` option can be useful, though the user is cautioned against using this option.

WARNING: The object-file generated by the compiler when this option is used may not execute correctly (or even cause XSB to core dump!) if the variable is indeed uninitialized when used.

`index_off` When specified, the compiler does not generate indices for the predicates compiled.

`compile_off` When specified, a byte code object-file is not generated. Rather a textual assembly file (`.A`) is generated instead. This option was created for compiler debugging.

`verbo` Compiles the files (modules) specified in "verbose" mode, printing out information about the progress of the compilation of each predicate.

`profile` This option is usually used when modifying the XSB compiler. When specified, the compiler prints out information about the time spent in each phase of the compilation process.

3.9.3 Specialisation

From version 1.4.0 on, the XSB compiler automatically performs specialisation of partially instantiated calls. Specialisation can be thought as a source-level program transformation of a program to a residual program in which partially instantiated calls to predicates in the original program are replaced with calls to specialised versions of these predicates. The expectation from this process is that the calls in the residual program can be executed more efficiently that their non-specialised counterparts. This expectation is justified mainly because of the following two basic properties of the specialisation algorithm:

Compile-time Clause Selection The specialised calls of the residual program directly select (at compile time) a subset containing only the clauses that the corresponding calls of the original program would otherwise have to examine during their execution (at run time). By doing so, laying down unnecessary choice points is at least partly avoided, and so is the need to select clauses through some sort of indexing.

Factoring of Common Subterms Non-variable subterms of partially instantiated calls that are common with subterms in the heads of the selected clauses are factored out from these terms during the specialisation process. As a result, some head unification (`get_*` or `unify_*`) and some argument register (`put_*`) WAM instructions of the original program become unnecessary. These instructions are eliminated from both the specialised calls as well as from the specialised versions of the predicates.

Though these properties are sufficient to get the idea behind specialisation, the actual specialisation performed by the XSB compiler can be better understood by the following example. The example shows the specialisation of a predicate that checks if a list of HiLog terms is ordered:

```
ordered([]).
ordered([X]).
ordered([X,Y|Z]) :-
    X @=< Y, ordered([Y|Z]).
```

$\longrightarrow$

```
ordered([]).
ordered([X]).
ordered([X,Y|Z]) :-
    X @=< Y, _$ordered(Y, Z).

:- index _$ordered/2-2.
_$ordered(X, []).
_$ordered(X, [Y|Z]) :-
    X @=< Y, _$ordered(Y, Z).
```

The transformation (driven by the partially instantiated call `ordered([Y|Z])`) effectively allows predicate `ordered/2` to be completely deterministic (when used with a proper list as its argument), and to not use any unnecessary heap-space for its execution. We note that appropriate `:- index` directives are automatically generated by the XSB compiler for all specialised versions of predicates.

The default specialisation of partially instantiated calls is without any folding of the clauses that the calls select. Using the `spec_repr` compiler option (see Section 3.9.2) specialisation with replacement of the selected clauses with the representative of these clauses is performed. Using this compiler option, predicate `ordered/2` above would be specialised as follows:

```
ordered([]).
ordered([X|Y]) :- _$ordered(X, Y).

:- index _$ordered/2-2.
_$ordered(X, []).
_$ordered(X, [Y|Z]) :- X @=< Y, _$ordered(Y, Z).
```

We note that in the presense of cuts or side-effects, the code replacement operation is not always sound, i.e. there are cases when the original and the residual program are not computationally equivalent (with respect to the answer substitution semantics). The compiler checks for sufficient (but not necessary) conditions that guarantee computational equivalence, and if these conditions are not met, specialisation is not performed for the violating calls.

The XSB compiler prints out messages whenever it specialises calls to some predicate. For example, while compiling a file containing predicate **ordered/1** above, the compiler would print out the following message:

```
% Specialising partially instantiated calls to ordered/1
```

The user may examine the results of the specialisation transformation by using the **spec_dump** compiler option (see section 3.9.2).

Finally, we have to mention that specialisation cannot be transparent to the user; predicates created by the transformation do appear during tracing.

3.9.4 Compiler Directives

In XSB Version 1.4.0 the following compiler directives are recognized. Directives like **mode** and **parallel** are simply ignored by the compiler, but do not result in syntax errors to enhance compatibility with various other earlier versions of SB-Prolog. In future versions of XSB the above directives may not be simply ignored.

Tabling Directives

Memoization is often necessary to ensure that programs terminate, and can be useful as an optimization strategy as well. (See section 9.2). The underlying engine of XSB is based on SLG, a memoization strategy, which, in our version, maintains a table of calls and their answers for each predicate declared as *tabled*. Predicates that are not declared as tabled execute as in Prolog, eliminating the expense of tabling when it is unnecessary.

The simplest way to use tabling is to include the directive

```
:- table_all.
```

in the source file. To understand precisely what **table_all** does, it is necessary to mention a few properties of SLG on *stratified* programs. (Stratified programs can generally be understood as programs which contain no recursion through negation). For stratified programs which have no function symbols, or where function symbols always have a limited depth, SLG resolution ensures that any query will terminate after it has found all correct answers. In the rest of this section, we restrict consideration to such programs.

Obviously, not all predicates will need to be tabled for a program to terminate. **table_all** tables only those predicates which appear to its static analysis to contain an infinite loop. It is perhaps more illuminating to demonstrate these conditions than to explain them. For instance, in the program.

```
:- table_all.

p(a) :- p(f(a)).

s(X) :- s(f(a)).

r(X) :- q(X,W),r(Y).

m(X) :- m(f(X)).

ap1([H|T],L,[H|L1]) :- ap1(T,L,L1).

ap([],F,F).
ap([H|T],L,[H|L1]) :- ap(T,L,L1).

mem(H,[H|T]).
mem(H,[_|T]) :- mem(H,T).
```

The compiler prints out the messages

```
% Compiling predicate p/1 as a tabled predicate
% Compiling predicate s/1 as a tabled predicate
% Compiling predicate r/1 as a tabled predicate
% Compiling predicate m/1 as a tabled predicate
% Compiling predicate ap1/3 as a tabled predicate
```

Loop-free conditions were detected for **ap/3** and **mem/3**, but not for any of the other predicates.

table_all gives an approximation of tabled programs which we hope will be useful for most programs. The minimal set of tabled predicates needed to insure termination for a given program is uncomputable. Practically, refining the set of tabled predicates deduced by **table_all** is still an open research problem.

It should be noted that the presence of meta-predicates such as **call/1** makes any static analysis useless, so that the **table_all** directive should not be used in such cases.

Predicates can be explicitly declared as tabled as well, through the **table/[1,2,3]**. When **table/1** is used, the directive takes the form

```
:- table(F/A).
```

where **F** is the functor of the predicate to be tabled, and **A** its arity. Variant forms of the directive are useful for program optimization: with them, the user may alter the size of the *call hash table* or of the *return hash table*, both of which can crucially affect the speed of a query.

The call hash table is used whenever a call is made to a tabled predicate. If the system contains a table for a call which is a variant to the one being made, the returns from the table, if any, will be used rather than program clause resolution as answers to the call. In the current version of XSB, call indexing is done on the first argument, with depth 1. (That is, on the principle functor of the first argument as with program clause indexing). Future versions of XSB are intended to allow more configurable indexing.

Each non-variant call of each tabled predicate contains a return hash table which is used to check for variants (duplicates) when adding returns. In the current version, hashing in the return hash table is done on the sum of the arguments, each at depth 1. It is important to note that the increasing the size of the return hash table may speed up queries which construct a table in the process of deriving answers. It does not improve the speed of queries which use answers from a completed table. In fact, since in Version 1.4.0 the empty hash buckets must be searched in looking for solutions in a completed table, large increases in return hash table size can potentially *increase* the time for queries to a completed table.

The standard predicate **analyze_table/1** prints out the total number of return and call hash collisions. (See section 6.11). Most computations tend to be more sensitive to the size of the return hash table than to the call hash table. The directive

```
:- table(F/A, RetSize).
```

sets the size of the return hash table to be **RetSize**, while the directive

```
:- table(F/A, RetSize, CallSize).
```

sets the size of the return hash table to be `RetSize`, and the size of the call hash table to be `CallSize`.

Indexing Directives

The XSB compiler usually generates an index on the principal functor of the first argument of a predicate. Indexing on the appropriate argument of a predicate may significantly speed up its execution time. In many cases the first argument of a predicate may not be the most appropriate argument for indexing and changing the order of arguments may seem unnatural. In these cases, the user may generate an index on any other argument by means of an indexing directive. This is a directive of the form:

```
:- index Functor/Arity-IndexArg.
```

indicating that an index should be created for predicate `Functor`/`Arity` on its `IndexArg`$^{\mathrm{th}}$ argument. One may also use the form:

```
:- index(Functor/Arity, IndexArg, HashTableSize).
```

which allows further specification of the size of the hash table to use for indexing this predicate if it is a *dynamic* (i.e., asserted) predicate. For a compiled predicate the size of the hash table is computed automatically, so `HashTableSize` is ignored.

All of the values `Functor`, `Arity`, `IndexArg` (and possibly `HashTableSize`) should be ground in the directive. More specifically, `Functor` should be an atom, `Arity` an integer in the range 0..255, and `IndexArg` an integer between 0 and `Arity`. If `IndexArg` is equal to 0, then no index is created for that predicate. An `index` directive may be placed anywhere in the file containing the predicate it refers to.

As an example, if we wished to create an index on the third argument of predicate `foo/5`, the compiler directive would be:

```
:- index foo/5-3.
```

Transformational Indexing

When the clause heads of a predicate have portions of arguments common to several clauses, indexing on the principal functor of one argument may not be sufficient. Indexing may be improved in such cases by the use of transformational indexing. Transformational indexing is a program transformation that "factors out" common parts of clause heads, allowing differing parts to be used for indexing, as illustrated by the following example:

```
p(f(a),X) :- q(X).          p(f(X),Y) :- _$p(X,Y).
p(f(b),X) :- r(X).    ⟶     _$p(a,X) :- q(X).
                            _$p(b,X) :- r(X).
```

The transformation thus effectively allows $p/2$ to be indexed on atoms $a/0$ and $b/0$. Transformational indexing is transparent to the user; predicates created by the transformation are internal to the system and do not appear during tracing.

The following compiler directives control the use of transformational indexing:

`:- ti(F/A).` Specifies that predicate F/A should be compiled with transformational indexing enabled.

`:- ti_off(F/A).` Specifies that predicate F/A should be compiled with transformational indexing disabled.

`:- ti_all.` Specifies that all predicates defined in the file should be compiled with transformational indexing enabled.

`:- ti_off_all.` Specifies that all predicates defined in the file should be compiled with transformational indexing disabled.

By default, higher-order predicates (more precisely, predicates named *apply* with arity greater than 1) are compiled with transformational indexing enabled. It can be disabled using the `ti_off` directive. For all other predicates, transformational indexing must be enabled explicitly via the `ti` or `ti_all` directive. If both `:- ti(F/A).` (`:- ti_all.`) and `:- ti_off(F/A).` (`:- ti_off_all.`) are specified, `:- ti_off(F/A).` (`:- ti_off_all.`) takes precedence. Note that transformational indexing may have no effect when a predicate is well indexed to begin with. For example, transformational indexing has no effect on the following program:

```
p(a,c,X) :- q(X).
p(b,c,X) :- r(X).
```

even though the two clauses have $c/0$ in common. The user may examine the results of the transformation by using the `ti_dump` compiler option (see section 3.9.2).

3.9.5 Inline 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 (like `>/2`, `>=/2`, etc.) compile to, essentially, a subtraction followed by a conditional branch. Inline predicates are expanded specially by the compiler and thus *cannot be redefined by the user without changing the compiler*. The user does not need to import these predicates from anywhere. There are available no matter what options are specified during compiling.

Table 3.1 lists the inline predicates of XSB Version 1.4.0. Those predicates that start with `_$` are internal predicates that are also expanded in-line during compilation.

`'='/2`	`'<'/2`	`'=<'/2`	`'>='/2`	`'>'/2`
`'=:='/2`	`'=\='/2`	`is/2`	`'@<'/2`	`'@=<'/2`
`'@>'/2`	`'@>='/2`	`'=='/2`	`'\=='/2`	`fail/0`
`true/0`	`var/1`	`nonvar/1`	`halt/0`	`'!'/0`
`'_$cutto'/1`	`'_$savecp'/1`	`'_$builtin'/1`		

Table 3.1: The Inline Predicates of XSB

We warn the user to be very cautious when defining predicates whose functor starts with `_$` since the names of these predicates may interfere with some of XSB's internal predicates. The situation may be particularly severe for predicates like `'_$builtin'/1` that are treated specially by the XSB compiler.

3.10 Other declarations

Besides the declarations for the module system introduced in section 3.4, XSB has several other new directives not found in other Prolog systems.

`:- hilog` $atom_1, \ldots, atom_n$.
: Declares symbols $atom_1$ through $atom_n$ as HiLog symbols. The `hilog` declaration should appear *before* any use of the symbols. See Chapter 4 for a purpose of this declaration.

`:- table_all.`
: The `table_all` directive may be placed anywhere in the file.

`:- tabled` $pred_1, \ldots, pred_n$.
: Notifies the compiler that predicates $pred_1$ through $pred_n$ are to be compiled as tabled predicates. More about this directive and its variants can be found in section 3.9.4.

`:- index` *Name*/*arity-argno*.

Notifies the compiler to index the predicate *name/arity* on its *argno*-th argument. The default action of the compiler is to generate an index on the principal functor of the first argument of the predicate. Indexing on the appropriate argument of a predicate may significantly speed up the execution time for accessing the rules of that predicate. Refer to Section 3.9.4, for a more detailed description of this declaration.

`:- ldoption(`*Options*`).`

This directive is only recognized in the header file (`.H` file) of a foreign module. See Chapter 11 for its explanation.

Chapter 4

Syntax

The syntax of XSB is taken from C-Prolog with extensions. This chapter mainly introduces the extensions. The syntax of XSB is that of the HiLog language. The syntax of HiLog is a proper superset of the Prolog syntax.

4.1 Terms

The data objects of the HiLog language are called *terms*. A *HiLog term* can be constructed from any logical symbol or a term followed by any finite number of arguments. In any case, a *term* is either a *constant*, a *variable*, or a *compound term*.

A *constant* is either a *number* (integer or floating-point) or an *atom*. Constants are definite elementary objects, and correspond to proper nouns in natural language.

4.1.1 Integers

The printed form of an integer in HiLog consists of a sequence of digits optionally preceded by a minus sign (`'-'`). These are normally interpreted as base 10 integers. It is also possible to enter integers in other bases (2 through 36); this can be done by preceding the digit string by the base (in decimal) followed by an apostrophe (`'`). If a base greater than 10 is used, the characters `A-Z` or `a-z` are used to stand for digits greater than 9.

Using these rules, examples of valid integer representations in XSB are:

```
1    -3456    95359    9'888    16'1FA4    -12'A0    20'
```

representing respectively the following integers in decimal base:

```
1    -3456    95359    728    8100    -120    0
```

Note that the following:

```
+525    12'2CF4    37'12    20'-23
```

are not valid integers of XSB.

A base of 0 (zero) will return the ASCII code of the (single) character after the apostrophe; for example,

```
0'A = 65
```

4.1.2 Floating-point Numbers

A HiLog floating-point number consists of a sequence of digits with an embedded decimal point, optionally preceded by a minus sign (`'-'`), and optionally followed by an exponent consisting of uppercase or lowercase `'E'` and a signed base 10 integer.

Using these rules, examples of HiLog floating point numbers are:

```
1.0    -34.56    817.3E12    -0.0314e26    2.0E-1
```

Note that in any case there must be at least one digit before, and one digit after, the decimal point.

4.1.3 Atoms

A HiLog atom is identified by its name, which is a sequence of up to 1000 characters (other than the null character). Just like a Prolog atom, a HiLog atom can be written in any of the following forms:

- Any sequence of alphanumeric characters (including `'_'`), starting with a lowercase letter.

- Any sequence from the following set of characters (except of the sequence '/*', which begins a comment):

```
+ - * / \ ^ < > = ` ~ : . ? @ # &
```

- Any sequence of characters delimited by single quotes, such as:

```
'sofaki'    '%'    '_$op'
```

If the single quote character is to be included in the sequence it must be written twice. For example:

```
'don''t'    ''''
```

- Any of the following:

```
!  ;  []  {}
```

Note that the bracket pairs are special. While '[]' and '{}' are atoms, '[', ']', '{', and '}' are not. Like Prolog, the form [X] is a special notation for lists (see section 4.1.6), while the form {X} is just "syntactic sugar" for the term '{}'(X).

Examples of HiLog atoms are:

```
h   foo   ^=..   ::=   'I am also a HiLog atom'   []
```

4.1.4 Variables

Variables may be written as any sequence of alphanumeric characters (including '_') beginning with either a capital letter or '_'. For example:

```
X   HiLog   Var1   _3   _List
```

If a variable is referred to only once in a clause, it does not need to be named and may be written as an *anonymous variable*, represented by a single underscore character '_'. Any number of anonymous variables may appear in a HiLog clause; all of these variables are read as distinct variables. Anonymous variables are not special at runtime.

4.1.5 Compound Terms

Like in Prolog, the structured data objects of HiLog are *compound terms* (or *structures*). The external representation of a HiLog compound term comprises a *functor* (called the *principal functor* or the *name* of the compound term) and a sequence of one or more terms called *arguments*. Unlike Prolog where the functor of a term must be an atom, in HiLog the functor of a compound term *can be any valid HiLog term*. This includes numbers, atoms, variables or even compound terms. Thus, since in HiLog a compound term is just a term followed by any finite number of arguments, all the following are valid external representations of HiLog compound terms:

```
foo(bar)               prolog(a, X)            hilog(X)
123(john, 500)         X(kostis, sofia)        X(Y, Z, Y(W))
f(a, (b(c))(d))        map(double)([], [])     h(map(P)(A, B))(C)
```

Like a functor in Prolog, a functor in HiLog can be characterized by its *name* and its *arity* which is the number of arguments this functor is applied to. For example, the compound term whose principal functor is `'map(P)'` of arity 2, and which has arguments `L1`, and `L2`, is written as:

```
map(P)(L1, L2)
```

As in Prolog, when we need to refer explicitly to a functor we will normally denote it by the form $Name/Arity$. Thus, in the previous example, the functor `'map(P)'` of arity 2 is denoted by:

```
map(P)/2
```

Note that a functor of arity 0 is represented as an atom.

In Prolog, a compound term of the form $p(t_1, t_2, \ldots, t_k)$ is usually pictured as a tree in which every node contains the name p of the functor of the term and has exactly k children each one of which is the root of the tree of terms $t_1, t_2, \ldots, t_k$.

For example, the compound term

```
s(np(kostis), vp(v(loves), np(sofia)))
```

would be pictured as the following tree:

```
              s
            /   \
         np       vp
         |       /  \
         |      v     np
         |      |     |
      kostis  loves  sofia
```

The principal functor of this term is `s/2`. Its two arguments are also compound terms. In illustration, the principal functor of the second argument is `vp/2`.

Likewise, any external representation of a HiLog compound term $t(t_1, t_2, \ldots, t_k)$ can be pictured as a tree in which every node contains the tree representation of the name t of the functor of the term and has exactly k children each one of which is the root of the tree of terms $t_1, t_2, \ldots, t_k$.

Sometimes it is convenient to write certain functors as *operators*. *Binary functors* (that is, functors that are applied to two arguments) may be declared as *infix operators*, and *unary functors* (that is, functors that are applied to one argument) may be declared as either *prefix or postfix operators*. Thus, it is possible to write the following:

```
X+Y     (P;Q)     X<Y      +X     P;
```

More about operators in HiLog can be found in section 4.3.

4.1.6 Lists

As in Prolog, lists form an important class of data structures in HiLog. They are essentially the same as the lists of Lisp: a list is either the atom `'[]'`, representing the empty list, or else a compound term with functor `'.'` and two arguments which are the head and tail of the list respectively, where the tail of a list is also a list. Thus a list of the first three natural numbers is the structure:

```
     .
    / \
   1   .
      / \
     2   .
        / \
       3   []
```

which could be written using the standard syntax, as:

```
.(1,.(2,.(3,[])))
```

but which is normally written in a special list notation, as:

```
[1,2,3]
```

Two examples of this list notation, as used when the tail of a list is a variable, are:

```
[Head|Tail]      [foo,bar|Tail]
```

which represent the structures:

```
     .               .
    / \             / \
Head   Tail      foo   .
                      / \
                   bar   Tail
```

respectively.

Note that the usual list notation `[H|T]` does not add any new power to the language; it is simply a notational convenience and improves readability. The above examples could have been written equally well as:

```
.(Head,Tail)      .(foo,.(bar,Tail))
```

For convenience, a further notational variant is allowed for lists of integers that correspond to ASCII character codes. Lists written in this notation are called *strings*. For example,

```
"I am a HiLog string"
```

represents exactly the same list as:

```
[73,32,97,109,32,97,32,72,105,76,111,103,32,115,116,114,105,110,103]
```

4.2 From HiLog to Prolog

From the discussion about the syntax of HiLog terms, it is clear that the HiLog syntax allows the incorporation of some higher-order constructs in a declarative way within logic programs. As we will show in this section, HiLog does so while retaining a clean first-order declarative semantics. The semantics of HiLog is first-order, because every HiLog term (and formula) is automatically *encoded (converted)* in predicate calculus in the way explained below.

Before we briefly explain the encoding of HiLog terms, let us note that the HiLog syntax is a simple (but notationally very convenient) encoding for Prolog terms, of some special form. In the same way that in Prolog:

```
1 + 2
```

is just an (external) shorthand for the term:

```
+(1, 2)
```

in the presence of an infix operator declaration for `+` (see section 4.3), so:

```
X(a, b)
```

is just an (external) shorthand for the Prolog compound term:

```
apply(X, a, b)
```

Also, in the presence of a **`hilog`** declaration (see section 3.10) for **`h`**, the HiLog term whose external representation is:

```
h(a, h, b)
```

is a notational shorthand for the term:

```
apply(h, a, h, b)
```

Notice that even though the two occurrences of **`h`** refer to the same symbol, only the one where **`h`** appears in a functor position is encoded with the special functor **`apply`**$/n, n \geq 1$.

The encoding of HiLog terms is performed based upon the existing declarations of *hilog symbols*. These declarations (see section 3.10), determine whether an atom that appears in a functor position of an external representation of a HiLog term, denotes a functor or the first argument of a set of special functors `apply`. The actual encoding is as follows:

- The encoding of any variable or parameter symbol (atom or number) that does not appear in a functor position is the variable or the symbol itself.
- The encoding of any compound term `t` where the functor *f* is an atom that is not one of the hilog symbols (as a result of a previous `hilog` declaration), is the compound term that has *f* as functor and has as arguments the encoding of the arguments of term *t*. Note that the arity of the compound term that results from the encoding of *t* is the same as that of *t*.
- The encoding of any compound term `t` where the functor *f* is either not an atom, or is an atom that is a hilog symbol, is a compound term that has `apply` as functor, has first argument the encoding of *f* and the rest of its arguments are obtained by encoding of the arguments of term*t*. Note that in this case the arity of the compound term that results from the encoding of *t* is one more than the arity of *t*.

Note that the encoding of HiLog terms described above, implies that even though the HiLog terms:

```
p(a, b)
h(a, b)
```

externally appear to have the same form, in the presence of a `hilog` declaration for `h` but not for `p`, they are completely different. This is because these terms are shorthands for the terms whose internal representation is:

```
p(a, b)
apply(h, a, b)
```

respectively. Furthermore, only `h(a,b)` is unifiable with the HiLog term whose external representation is `X(a, b)`.

We end this short discussion on the encoding of HiLog terms with a small example that illustrates the way the encoding described above is being done. Assuming that the following declarations of parameter symbols have taken place,

```
:- hilog h.
:- hilog (hilog).
```

before the compound terms of page 37 were read by XSB, the encoding of these terms in predicate calculus using the described transformation is as follows:

```
        foo(bar)                         prolog(a,X)
     apply(hilog,X)                 apply(123,john,500)
 apply(X,kostis,sofia)         apply(X,Y,Z,apply(Y,W))
   f(a,apply(b(c),d))         apply(map(double),[],[])
           apply(apply(h,apply(map(P),A,B)),C)
```

4.3 Operators

From a theoretical point of view, operators in Prolog are simply a notational convenience and add absolutely nothing to the power of the language. For example, in most Prologs '+' is an infix operator, so

```
2 + 1
```

is an alternative way of writing the term `+(2, 1)`. That is, `2 + 1` represents the data structure:

```
  +
 / \
2   1
```

and not the number 3. (The addition would only be performed if the structure were passed as an argument to an appropriate procedure, such as `is/2` described in section 6.2).

However, from a practical or a programmer's point of view, the existence of operators is highly desirable, and clearly handy.

Prolog syntax allows operators of three 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 follows its single argument.

Each operator has a precedence, which is an integer from 1 to 1200. The precedence is used to disambiguate expressions in which the structure of the term denoted is not made explicit through the use of parentheses. The general rule is that the operator with the highest precedence is the principal functor. Thus if '+' has a higher precedence than '/', then the following

```
a+b/c     a+(b/c)
```

are equivalent, and both denote the same term `+(a,/(b,c))`. Note that in this case, the infix form of the term `/(+(a,b),c)` must be written with explicit use of parentheses, as in:

```
(a+b)/c
```

If there are two operators in the expression having the same highest precedence, the ambiguity must be resolved from the *types* (and the implied *associativity*) of the operators. The possible types for an infix operator are

```
yfx     xfx     xfy
```

Operators of type `'xfx'` are not associative. Thus, it is required that both of the arguments of the operator must be subexpressions of lower precedence than the operator itself; that is, the principal functor of each subexpression must be of lower precedence, unless the subexpression is written in parentheses (which automatically gives it zero precedence).

Operators of type `'xfy'` are *right-associative*: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the same precedence as the main operator. *Left-associative* operators (type `'yfx'`) are the other way around.

An atom named `Name` can be declared as an operator of type `Type` and precedence `Precedence` by the command;

```
:- op(Precedence, Type, Name).
```

The same command can be used to redefine one of the predefined XSB operators (see appendix A.3). However, it is not allowed to alter the definition of the comma (`','`) operator. An operator declaration can be cancelled by redeclaring the `Name` with the same `Type`, but `Precedence` 0.

As a notational convenience, 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: infix, prefix, or postfix. An operator of any kind may be redefined by a new declaration of the same kind. Declarations for all these built-in operators can be found in appendix A.3. For example, the built-in operators `'+'` and `'-'` are as if they had been declared by the command:

```
:- op(500, yfx, [+,-]).
```

so that:

```
1-2+3
```

is valid syntax, and denotes the compound term:

```
(1-2)+3
```

or pictorially:

```
      +
     / \
    -   3
   / \
  1   2
```

In XSB, the list functor `'.'/2` is one of the standard operators, that can be thought as declared by the command:

```
:- op(661, xfy, .).
```

So, in XSB,

```
1.2.[]
```

represents the structure

```
    .
   / \
  1   .
     / \
    2   []
```

Contrasting this picture with the picture above for `1-2+3` shows the difference between `'yfx'` operators where the tree grows to the left, and `'xfy'` operators where it grows to the right.

The tree cannot grow at all for `'xfx'` type operators. It is simply illegal to combine `'xfx'` operators having equal precedences in this way.

If these precedence and associativity rules seem rather complex, remember that you can always use parentheses when in any doubt.

In XSB, at the time when this is written, the possible types for prefix operators are:

`fx` `fy` `hx` `hy`

and the possible types for postfix operators are:

`xf` `yf`

We end our discussion about operators by just mentioning that prefix operators of type `hx` and `hy` are *proper HiLog operators*. The discussion of proper HiLog operators and their properties is deferred for the manual of a future version.

Chapter 5

Using Tabling in XSB

XSB differs from Prolog systems in that its evaluation strategy is based on SLG resolution, which uses a type of *memoization* rather than on SLD resolution, which does not. The difference in evaluation strategy means that, with appropriate declarations, XSB is complete for left-to-right modularly stratified programs with the bounded term-depth property, as will be explained below. Furthermore, XSB allows a different, more declarative programming style than Prolog for a number of problems. The following sections provide a brief introduction to SLG resolution, or *tabling* as we will informally call it. They also summarize points the programmer should know about XSB's implementation of tabling, and present a brief overview of applications of tabling in logic programming. Many of the topics will be covered in fuller detail in other sections of this manual.

5.1 SLG Evaluation

Consider the Prolog program

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
```

together with the query `?-ancestor(1,Y)`. This program has a simple, declarative meaning: `Y` is an ancestor of `X` if `Y` is a parent of `X` or if `Y` is a parent of an ancestor of `X`. Prolog, however cannot compute the answer to this query. It enters an infinite loop. The inability of Prolog to answer such queries, which arise frequently, comprises one of its major limitations.

A number of approaches have been developed which solve programs like ancestor by reusing partial answers to the query `ancestor(1,Y)` [DW 86, TS 86, BMSU 86, Viel 87, Walk ??]. Briefly, the ideas behind these algorithms can be described in the following manner. First, the program keeps track of nodes where calls to `ancestor(1,Y)` have been made. By maintaining this information, the repeated call to `ancestor(1,Y)` in the body of the second clause can be detected. The next idea is to store partial solutions to the call in a *table*, which consists of calls associated with their answers. These answers are substituted into the call to `ancestor(1,Y)` in the body of the second clause. The substitutions, after they are joined to the second literal of the second clause will in general produce new solutions to the query, and these new solutions in their turn are fed into the body of the second clause until a fixpoint (if any) is reached. More formally, and in the language of [TS 86], the first call of a (leftmost) goal is called a solution node, and is expanded using program clauses as in SLD resolution (Prolog). Correct answer substitutions to the solution node are interned in the *table entry* (call plus returns) for a call. Subsequent calls (that is, subsequent variant or subsumed calls) are referred to as lookup nodes, and are expanded using answers derived for that goal instead of program clauses. The computation ends when all possible program clauses have been used to expand all solution nodes, and when all answers have been returned to all lookup nodes.

Because lookup nodes are expanded using answers rather than program clauses, SLG resolution is complete and will terminate whenever the set of answers to all queries in a derivation is finite – a situation which occurs when the program has a finite model, for instance. Indeed, it can be proven that for any program with the *bounded term depth property* where all terms have a maximum depth, SLG computation is complete and will terminate. These programs include the important class of *Datalog* programs.

It is important to notice two differences between SLG resolution and SLD resolution beyond its termination properties. First, termination requires that each solution to a call be returned only once. Second, because of the way that solutions are stored and returned to lookup nodes in XSB, answers may be returned to the user in an unaccustomed order.

Most of the original definitions for tabling algorithms considered only definite programs, though [KT 88, SI 88] extended of SLG to programs with stratified negation. (A program uses stratified negation whenever there is no recursion through negation). It is not hard to see, however, how this extension can be made. Briefly, each time a negative goal is called, a separate table is opened for the negative call. This evaluation is carried on to termination. If it terminates, as it will for the classes of programs in the last paragraph, its answers if any, are used to determine the success of failure of the calling goal.

As an example, using a method like the one above, any query to a program like

```
m :- c,a.
c :- b
c.
b :- c,d.
a :- ~b.
```

can be solved.

The method just sketched is correct, but is not terribly efficient. Every time a new negative goal is called, a new table must be started, and run to completion. We would like to use information already derived from the computation to answer a new query, if at all possible.

XSB addresses this problem by keeping track of the *state* of each call in the program table. A call can have a state of *not_yet_called*, *incomplete*, or *complete*. The value not_yet_called means that there is in fact no table entry. Calls that do have table entries may be either complete or incomplete. A call is complete only when its table entry contains all possible solutions that are implied by the program, otherwise the call is incomplete. The scheduling strategy of program clause expansions and answer returns in XSB allows for efficient detection of when a call may be marked complete.

Returning to the example of a negative goal, the (ground) negative goal is suspended until either the answer is returned or until the table is marked as complete. XSB evaluates all programs that are *left-to-right modularly stratified* [Ross 91]. [1] The use of completion also forms the basis of evaluation of programs with stratified findall. (Using **tfindall/3**).

Tabling with completion can also be used to optimize programs that contain neither negation nor aggregates, and provides a declarative method of dynamic programming. In many programs it is necessary to solve the same subgoal many times. Tabling can optimize a program by reusing solutions. In the well-studied Fibonacci program

```
fib(0,1).
fib(1,1).
fib(M,N) :-
      M>1, M1 is N - 1, M2 is M - 2,
      fib(M1,N1), fib(M2,N2),
      N is N1 + N2.
```

the use of tabling can reduce the order of complexity of the query from $O(2^N)$ to $O(N)$. The Fibonacci program is functional.

While it is easy to see the advantages of tabling in the Fibonacci example, it is perhaps atypical in two respects. First, tabling will not generally produce such a drastic gain in well-written programs as it does in Fibonacci. Second, Fibonacci is a functional program, with each goal having a single answer, and does not express the power of XSB's tabling, which memos *alternate* solutions to a goal. (See Section 9.2 for further examples).

[1] Research is currently being done on how to extend the engine of XSB so that it computes the well-founded semantics.

While the judicious use of tabling can make some programs faster, its indiscriminate use can make other programs slower. Naively tabling `append/3` is one case

```
append([],L,L).
append([H|T],L,[H|T1]) :-
      append(T,L,T1).
```

can, in the worst case, copy N sublists of the first and third arguments into the table for no particular advantage. (XSB allows a measure of control over copying into a table, see Section 6.12.

In XSB, predicates are executed Prolog style by default, and execute by SLG resolution only if they are declared as tabled at compile time using declarations like

```
:- table(<Predicate>/<Arity>).
```

The user can either make these declarations explicitly, or can get the system to make them using the `table_all` directive.

5.2 Tabling in a Prolog Environment

Four of the goals of XSB are

- To execute tabled predicates at the speed of compiled Prolog.
- To ensure that the speed of compiled Prolog is not slowed significantly by adding the option of tabling.
- To provide Prolog functionality in tabled predicates whenever possible.
- To provide standard predicates to manipulate tables taken as objects in themselves.

In order to accomplish the first two of these goals, the *Warren Abstract Machine (WAM)* which serves as the engine of Prolog was redesigned to allow tabling. (The new engine is called the XWAM). While most WAM instructions were affected by the redesign, XWAM execution is only 5-15% slower than a WAM execution on a similar engine. Beyond the comparison with Prolog, our implementation strategy has several implications which may affect the user.

First, XSB's evaluation is top-down, and solutions are derived a tuple at a time. This characterization allows it to be contrasted with, say, magic sets [BMSU 86] which evaluate queries bottom-up and derive solutions a set at a time. There are advantages and disadvantages to XSB's evaluation method. One advantage is that the top down derivation maintains a representation of the state of a search, and because of this, answers are never returned to improper goals, then filtered out, as can happen in a bottom-up evaluation method. Another is that, since tabling is done at the XWAM level, the engine is very efficient, and unnecessary copies and unifications are minimized. There are associated disadvantages though. Since XSB's evaluation maintains a representation of the state of a search, it may requires more space than other methods. Also, since its evaluation is tuple-at-a-time, it is not particularly efficient for queries to large deductive databases whose relations must be kept on disk.

The speed of tabling is critically dependent on the size of the hash tables that it uses to index calls and returns. As mentioned above, each call to a tabled predicate must be checked to see if it should be resolved against program clauses or against answers from the table. A *call hash table* is used for this purpose, and the call hash table size can be configured as in Section 3.9.4. Of even greater importance to many problems is the size of the *return hash table* which is used to check for duplicate solutions, and directions for its use are given in the same section.

Tabling integrates well with most Prolog functionality. The use of negation and aggregates in tabled predicates was discussed in the precious section. Other Prolog predicates, such as meta-logical predicates like `var/1`, predicates with side-effects like `read/1` and `write/1` can be used freely in tabled predicates as long as it is remembered that only the first call to a goal will execute program clauses: the rest will look up answers from a table. If the user wants to reexecute the program clauses for any reason, the Prolog predicates `abolish_all_tables/0, abolish_table_pred/1` or `abolish_table_call/1` can be used, which will cause new calls to a goal to execute program clauses, in addition to removing the information in tables. Standard tabling predicates are described in Section 6.12

Cuts are allowed in tabled predicates, subject to the restriction that the scope of a cut cannot include a call to a tabled predicate. The way this description works can be seen most easily from an example. Suppose t/2 is a tabled predicate. Then the definition

```
t(A,B) :-
      p(A,B), ! ...
t(A,B) :-
      r(A,B), ...
```

is allowed, but a body with

```
:
t(A,B),
!,
:
```

is not allowed because the table corresponding to the call of `t(A,B)` will never be completed. Furthermore, there can also be cuts in the dependency graphs of `p/2` and `r/2`, subject to the scope restriction. The XSB compiler checks for violation of the scope restriction.

A similar situation occurs when the user makes a call to a tabled predicate from the interpreter level, but does not generate all solutions. In such a case, the user will see the message **`Removing open tables.`** appear. Any complete tables will not be removed. They can be abolished using one of the predicates mentioned above.

As a final point, it is often useful to manipulate tables in a non-logical way, for instance to create meta-interpreters. To do so we provide a number of standard predicates. The user is provided with the means to backtrack through all calls in a table, to backtrack through all returns in a call, to find out the state of a given table, to reset properties of given tables or to abolish them.

5.3 Applications of Tabling

β **Release Note:** This section remains to be written.

Chapter 6

Standard Predicates

Standard predicates are always available to the Prolog interpreter, and do not need to be imported or loaded explicitly as do other Prolog predicates. Our standard predicates are listed below. Standard predicates whose semantics depend on HiLog terms or on SLG evaluation are marked as HiLog or Tabling.

It is possible for the user to add standard predicates not provided in the standard release. See the section on Customizing XSB.

6.1 Input and Output

Presently, input and output can only be 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 internally denoted by userin and userout (the user accesses them both via the name `''user''`, and they refer to the user's terminal).

6.1.1 File Handling

`see(+F)`

Makes file `F` the current input stream.

- If there is an open input stream associated with the file that has `F` as its file name, and that stream was opened previously by `see/1`, then it is made the current input stream.
- Otherwise, the specified file is opened for input and made the current input stream. If the file does not exist, `see/1` fails.

Also note that different file names (that is, names which do not unify) represent different input streams (even if these different file names correspond to the same file).

Exceptions:

permission_error File **F** is directory or file is not readable.

instantiation_error **F** is not instantiated at the time of call.

existence_error File **F** does not exist.

seeing(?F)

F is unified with the name of the current input stream. This is exactly the same with predicate **current_input/1** described in section 6.9, and it is only provided for upwards compatibility reasons.

seen

Closes the current input stream. Current input reverts to **''userin''** (the standard input stream).

tell(+F)

Makes file **F** the current output stream.

- If there is an open output stream associated with F and that was opened previously by **tell/1**, then that stream is made the current output stream.
- Otherwise, the specified file is opened for output and made the current output stream. If the file does not exist, it is created.

Also note that different file names (that is, names which do not unify) represent different output streams (even if these different file names correspond to the same file).

Exceptions:

permission_error File **F** does not have write permission, or is a directory.

instantiation_error **F** is uninstantiated.

telling(?F)

F is unified with the name of the current output stream. This predicate is exactly the same with predicate **current_output/1** described in section 6.9, and it is only provided for upwards compatibility reasons.

told

Closes the current output stream. Current output stream reverts to "userout" (the standard output stream).

file_exists(+F)

Succeeds if file **F** exists. **F** must be instantiated to an atom at the time of the call, or an error message is displayed on the standard error stream and the predicate aborts.

Exceptions:

`instantiation_error` `F` is uninstantiated.

6.1.2 Character I/O

`nl`

A new line character is sent to the current output stream.

`get0(-N)`

`N` is the ASCII code of the next character read from the current input stream. If the current input stream reaches its end of file, a `-1` is returned

Compatability Note: Unlike other Prologs, such as C-Prolog, the input stream is not closed on encountering the end-of-file character.

`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 **`get0/1`** when an end of file character is encountered.

Compatability Note: Unlike other Prologs, such as C-Prolog, the input stream is not closed on encountering the end-of-file character.

`put(+N)`

Puts the ASCII character code `N` to the current output stream.

Exceptions:

`instantiation_error` `N` is not instantiated at the time of the call.

`type_error` `N` is not an integer at the time of the call.

`tab(+N)`

Puts `N` spaces to the current output stream.

Exceptions:

`instantiation_error` `N` is not instantiated at the time of the call.

`type_error` `N` is not an integer at the time of the call.

6.1.3 Term I/O

`read(-Term)`

A HiLog term is read from the current input stream, and unified with `Term` according to the operator declarations in force. (See Section 4.1 for the definition and syntax of HiLog terms). The term must be delimited by a full stop (i.e. a "." followed by a carriage-return, space or tab). Predicate `read/1` does not return until a valid HiLog term is successfully read; that is, in the presense of syntax errors `read/1` does not fail but continues reading terms until a term with no syntax errors is encountered. If a call to `read(Term)` causes the end of the current input stream to be reached, variable `Term` is unified with the term `end_of_file`. In that case, further calls to `read/1` for the same input stream will cause an error failure.

Exceptions:

`existence_error end_of_file` is reached before the current term is read.

`write(?Term)`

The HiLog term `Term` is written to the current output stream, according to the operator declarations in force. Any uninstantiated subterm of term `Term` is written as an anonymous variable (an underscore followed by a non-negative integer).

All *proper HiLog terms* (HiLog terms which are not also Prolog terms) are not written in their internal Prolog representation. Predicate `write/1` always succeeds without producing an error.

The HiLog terms that are output by `write/1` cannot in general be read back using `read/1`. This happens for two reasons:

- The atoms appearing in term `Term` are not quoted. In that case the user must use `writeq/1` or `write_canonical/1` described below, which quote around atoms whenever necessary.
- The output of `write/1` is not terminated by a full-stop; therefore, if the user wants the term to be accepted as input to `read/1`, the terminating full-stop must be explicitely sent to the current output stream.

Predicate `write/1` treats terms of the form `'$VAR'(N)` specially: it writes `'A'` if `N`=0, `'B'` if `N`=1, ..., `'Z'` if `N`=25, `'A1'` if `N`=26, etc. Terms of this form are generated by `numbervars/[1,3]` described in Section 10.3.

`writeln(?Term)`

`writeln(Term)` can be defined as `write(Term), nl`.

`display(?Term)`

The HiLog term `Term` is displayed on the terminal (standard output stream), according to the operator declarations in force. In other words, `display/1` is similar to `write/1` but the result is always written on `''userout''`. Like `write/1`, `display/1` always succeeds without producing an error. After returning from a call to this predicate, the current output stream remains unchanged.

`write_prolog(?Term)` HiLog

This predicate acts as does `write/1` except that any HiLog term `Term` is written as a

Prolog term. `write_prolog/1` outputs `Term` according to the operator declarations in force. Because of this, it differs from `write_canonical/1` described below, despite the fact that both predicates write HiLog terms as Prolog terms.

`writeq(?Term)`

Acts as `write(Term)`, but atoms and functors are quoted whenever necessary to make the result acceptable as input to `read/1`. `writeq/1` treats treats terms of the form `'$VAR'(N)` the same way as `write/1`, writing `A` if `N`$=0$, etc.

`writeq/1` always succeeds without producing an error.

`write_canonical(?Term)`

This predicate is provided so that the HiLog term `Term`, if written to a file, can be read back using `read/1` regardless of special characters appearing in `Term` or prevailing operator declarations. Like `write_prolog/1`, `write_canonical/1` writes all proper HiLog terms to the current output stream using the standard Prolog syntax (see Section 4.1 on the standard syntax of HiLog terms). `write_canonical/1` also quotes atoms and functors as `writeq/1` does, to make them acceptable as input of `read/1`. Operator declarations are not taken into consideration, and compound terms are therefore always written in the form:

$$\langle predicate\ name\rangle(\langle arg_1\rangle, \ldots, \langle arg_n\rangle)$$

Unlike `writeq/1`, `write_canonical/1` does not treat terms of the form `'$VAR'(N)` specially. It writes square bracket lists using `'.'/2` and `[]` (that is, `[foo, bar]` is written as `'.'(foo,'.'(bar,[]))`).

`print(?Term)`

This predicate is intended to provide a handle for user-defined pretty-printing. Currently it is defined as *write/1*.

6.2 Arithmetic

6.3 Convenience

These predicates are standard and often self-explanatory, so they are described only briefly.

`true`

Always succeeds.

`otherwise`

Same as `true/0`.

`fail`

Always fails.

`X = Y`

Defined as if by the clause "Z=Z", i.e. `X` and `Y` are unified.

`X \= Y`

Succeeds if `X` and `Y` are not unifiable, fails if `X` and `Y` are unifiable. It is thus equivalent to `\+(X = Y)`.

6.4 Negation and Control

`'!'/0`

Cut (discard) all choice points made since the parent goal started execution. Cuts across tabled predicates are not valid. The compiler checks for such cuts, although whether the scope of a cut includes a tabled predicate is undecidable in the presence of meta-predicates like `call/1`. Further discussion of conditions allowing cuts and of their actions can be found in Section 5.2.

`fail_if(+P)`

If the goal `P` has a solution, fails, otherwise it succeeds. Equivalently, it is true iff `call(P)` (see section 6.8) is false. Argument `P` must be ground for sound negation as failure, although no runtime checks are made by the system.

The standard predicate `fail_if/1` is compiled by the XSB compiler.

Exceptions:

`instantiation_error` `P` is not instantiated.

`type_error` `P` is not a callable term.

`\+ +P`

Exactly the same as `fail_if/1`. Its existence is only for compatibility with other Prolog systems.

`not +P`

If the goal `P` has a solution, fails, otherwise it succeeds. It is defined by:

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

Argument `P` must be ground for sound negation, although no runtime checks are made by the system.

Note that in contrast to the other two kinds of negation as failure ('\+'/1 and `fail_if/1`), predicate `not/1` is not compiled by the compiler but the above definition is used.

Exceptions: The same as `call/1` (see section 6.8).

`tnot(+P)` **Tabling**

The semantics of `tnot/1` allows for correct execution of programs with stratified negation. `P` must be a tabled predicate, and the actions of `tnot/1` are

```
If P is not completed, suspend until P is complete.
When P is complete,
        if there exists an answer for P, fail.
        otherwise succeed.
```

Chapter 5 contains further discussion of `tnot/1`.

Exceptions:

`table_error` P is not a call to a tabled predicate.

`P -> Q ; R`

Analogous to if `P` then `Q` else `R`, i.e. defined as if by

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

`P -> 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 choice points (in other words it provides a very convenient way of executing a loop). It is defined by the clauses:

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

6.5 Meta-Logical

To facilitate manipulation of terms as objects in themselves, XSB provides a number meta-logical predicates. These predicates include the standard meta-logical predicates of Prolog, along with their usual semantics. In addition are provided predicates which provide special operations on HiLog terms. For a full discussion of Prolog and HiLog terms see Section 4.1.

`var(?X)`

Succeeds if `X` is currently uninstantiated (i.e. is still a variable); otherwise it fails.

Term `X` is uninstantiated if it has not been bound to anything, except possibly another uninstantiated variable. Note in particular, that the HiLog term X(Y,Z) is considered to be instantiated. There is no distinction between a Prolog and a HiLog variable.

Examples:

```
| ?- var(X).
yes
| ?- var([X]).
no
| ?- var(X(Y,Z)).
no
| ?- var((X)).
yes
| ?- var((X)(Y)).
no
```

`nonvar(?X)`

Succeeds if `X` is currently instantiated to a non-variable term; otherwise it fails. This has exactly the opposite behaviour of `var/1`.

`atom(?X)`

Succeeds only if the `X` is currently instantiated to an atom, that is to a Prolog or HiLog non-numeric constant.

Examples:

```
| ?- atom(HiLog).
no
| ?- atom(10).
no
| ?- atom('HiLog').
yes
| ?- atom(X(a,b)).
no
| ?- atom(h).
```

```
yes
| ?- atom(+).
yes
| ?- atom([]).
yes
```

`integer(?X)`
Succeeds if `X` is currently instantiated to an integer; otherwise it fails.

`real(?X)`
Succeeds if `X` is currently instantiated to a floating point number; otherwise it fails.

`float(?X)`
Same as **`real/1`**. Succeeds if `X` is currently instantiated to a floating point number; otherwise it fails. This predicate is included for compatability with earlier versions of SBProlog.

`number(?X)`
Succeeds if `X` is currently instantiated to either an integer or a floating point number (real); otherwise it fails.

`atomic(?X)`
Succeeds if `X` is currently instantiated to an atom or a number; otherwise it fails.

Examples:

```
| ?- atomic(10).
yes
| ?- atomic(p).
yes
| ?- atomic(h).
yes
| ?- atomic(h(X)).
no
| ?- atomic("foo").
no
| ?- atomic('foo').
yes
| ?- atomic(X).
no
| ?- atomic(X((Y))).
no
```

`compound(?X)`
Succeeds if `X` is currently instantiated to a compound term (with arity greater that zero), i.e. to a nonvariable term that is not atomic; otherwise it fails.

Examples:

```
| ?- compound(1).
no
| ?- compound(foo(1,2,3)).
yes
| ?- compound([foo, bar]).
yes
| ?- compound("foo").
yes
| ?- compound('foo').
no
| ?- compound(X(a,b)).
yes
| ?- compound((a,b)).
yes
```

structure(?X)

Same as **compound/1**. Its existence is only for compatibility with SB-Prolog version 3.1.

is_list(+X)

Succeeds if `X` is a *proper list*. In other words if it is either the atom `[]` or `[H|T]` where H is any Prolog or HiLog term and T is a proper list; otherwise it fails.

Examples:

```
| ?- is_list([p(a,b,c), h(a,b)]).
yes
| ?- is_list([_,_]).
yes
| ?- is_list([a,b|X]).
no
| ?- is_list([a|b]).
no
```

callable(?X)

Succeeds if `X` is currently instantiated to a term that standard predicate **call/1** could take as an argument and not give an instantiation or type error. Note that it only checks for errors of predicate **call/1**. In other words it succeeds if `X` is an atom or a compound term; otherwise it fails. Predicate **callable/1** has no associated error conditions.

Examples:

```
| ?- callable(p).
yes
| ?- callable(p(1,2,3)).
yes
```

```
| ?- callable([_,_]).
yes
| ?- callable(_(a)).
yes
| ?- callable(3.14).
no
```

`proper_hilog(+X)` HiLog

Succeeds if `X` is a proper HiLog term; otherwise it fails.

Examples: (In this example and the rest of the examples of this section we assume that `h` is the only parameter symbol that has been declared a HiLog symbol).

```
| ?- proper_hilog(X).
no
| ?- proper_hilog(foo(a,f(b),[A])).
no
| ?- proper_hilog(X(a,b,c)).
yes
| ?- proper_hilog(3.6(2,4)).
yes
| ?- proper_hilog(h).
no
| ?- proper_hilog([a, [d, e, X(a)], c]).
yes
| ?- proper_hilog(a(a(X(a)))).
yes
```

`functor(?Term, ?Functor, ?Arity)`

Succeeds if the *functor* of the Prolog term `Term` is `Functor` and the *arity* (number of arguments) of `Term` is `Arity`. `Functor` can be used in either the following two ways:

1. If `Term` is initially instantiated, then
   - If `Term` is a compound term, `Functor` and `Arity` are unified with the name and arity of its principal functor, respectively.
   - If `Term` is an atom or a number, `Functor` is unified with `Term`, and `Arity` is unified with 0.
2. If `Term` is initially uninstantiated, then either both `Functor` and `Arity` must be instantiated, or `Functor` is instantiated to a number, and
   - If `Arity` is an integer in the range 1..255, then `Term` becomes instantiated to *the most general Prolog term* having the specified `Functor` and `Arity` as principal functor and number of arguments, respectively. The variables appearing as arguments of `Term` are all distinct.

- If `Arity` is 0, then `Functor` must be either an atom or a number and it is unified with `Term`.
- If `Arity` is anything else, then `functor/3` aborts.

Exceptions:

`domain_error` `Functor` is instantiated to a compound term.

`instantiation_error` Both `Term`, and either `Functor`, or `Arity` are uninstantiated.

Examples:

```
| ?- functor(p(f(a),b,t), F, A).
F = p
A = 3

| ?- functor(T, foo, 3).
T = foo(_595708,_595712,_595716)

| ?- functor(T, 1.3, A).
T = 1.3
A = 0

| ?- functor(foo, F, 0).
F = foo

| ?- functor("foo", F, A).
F = .
A = 2

| ?- functor([], [], A).
A = 0

| ?- functor([2,3,4], F, A).
F = .
A = 2

| ?- functor(a+b, F, A).
F = +
A = 2

| ?- functor(f(a,b,c), F, A).
F = f
A = 3

| ?- functor(X(a,b,c), F, A).
F = apply
A = 4

| ?- functor(map(P)(a,b), F, A).
F = apply
```

```
A = 3

| ?- functor(T, foo(a), 1).
++Error: Wrong type in argument 2 of functor/3
Aborting...

| ?- functor(T, F, 3).
++Error: Uninstantiated argument 2 of functor/3
Aborting...

| ?- functor(T, foo, A).
++Error: Uninstantiated argument 3 of functor/3
Aborting...
```

hilog_functor(?Term, ?F, ?Arity) HiLog

The XSB standard predicate **hilog_functor/3** succeeds

- when **Term** is a Prolog term and the principal function symbol (*functor*) of **Term** is **F** and the *arity* (number of arguments) of **Term** is **Arity**, or
- when **Term** is a HiLog term, having *name* **F** and the number of arguments **F** is applied to, in the HiLog term, is **Arity**.

The first of these cases corresponds to the "usual" behaviour of the Prolog predicate **functor/3**, while the second is the extension of **functor/3** to handle HiLog terms. Like the Prolog **functor/3** predicate, **hilog_functor/3** can be used in either the following two ways:

1. If **Term** is initially instantiated, then
   - If **Term** is a Prolog compound term, **F** and **Arity** are unified with the name and arity of its principal functor, respectively.
   - If **Term** is an atom or a number, **F** is unified with **Term**, and **Arity** is unified with 0.
   - If **Term** is any other HiLog term, **F** and **Arity** are unified with the name and the number of arguments that **F** is applied to. Note that in this case **F** may still be uninstantiated.
2. If **Term** is initially uninstantiated, then at least **Arity** must be instantiated, and
   - If **Arity** is an integer in the range 1..255, then **Term** becomes instantiated to *the most general Prolog or HiLog term* having the specified **F** and **Arity** as name and number of arguments **F** is applied to, respectively. The variables appearing as arguments are all unique.
   - If **Arity** is 0, then **F** must be a Prolog or HiLog constant, and it is unified with **Term**. Note that in this case **F** cannot be a compound term.
   - If **Arity** is anything else, then **hilog_functor/3** aborts.

In other words, the standard predicate **hilog_functor/3** either decomposes a given HiLog term into its *name* and *arity*, or given an arity —and possibly a name— constructs the corresponding HiLog term creating new uninstantiated variables for its arguments. As happens with **functor/3** all constants can be their own principal function symbols.

Examples:

```
| ?- hilog_functor(f(a,b,c), F, A).
F = f
A = 3

| ?- hilog_functor(X(a,b,c), F, A).
X = _595836
F = _595836
A = 3

| ?- hilog_functor(map(P)(a,b), F, A).
P = _595828
F = map(_595828)
A = 2

| ?- hilog_functor(T, p, 2).
T = p(_595708,_595712)

| ?- hilog_functor(T, h, 2).
T = apply(h,_595712,_595716)

| ?- hilog_functor(T, X, 3).
T = apply(_595592,_595736,_595740,_595744)
X = _595592

| ?- hilog_functor(T, p(f(a)), 2).
T = apply(p(f(a)),_595792,_595796)

| ?- hilog_functor(T, h(p(a))(L1,L2), 1).
T = apply(apply(apply(h,p(a)),_595984,_595776),_596128)
L1 = _595984
L2 = _595776

| ?- hilog_functor(T, a+b, 3).
T = apply(a+b,_595820,_595824,_595828)
```

arg(+Index, +Term, ?Argument)

Unifies **Argument** with the **Index**th argument of **Term**, where the index is taken to start at 1. Initially, **Index** must be instantiated to any integer and **Term** to any non-variable Prolog or HiLog term. The arguments of the **Term** are numbered from 1 upwards. An atomic term has 0 arguments. If the initial conditions are not satisfied or I is out of range, the call quietly fails.

Examples:

```
| ?- arg(2, p(a,b), A).
A = b

| ?- arg(2, h(a,b), A).
A = a

| ?- arg(0, foo, A).
no

| ?- arg(2, [a,b,c], A).
A = [b,c]

| ?- arg(2, "HiLog", A).
A = [105,108,111,103]

| ?- arg(2, a+b+c, A).
A = c

| ?- arg(3, X(a,b,c), A).
X = _595820
A = b

| ?- arg(2, map(f)(a,b), A).
A = a

| ?- arg(1, map(f)(a,b), A).
A = map(f)

| ?- arg(1, (a+b)(foo,bar), A).
A = a+b
```

arg0(+Index, +Term, ?Argument)

Unifies **Argument** with the **Index**th argument of **Term** if **Index** > 0, or with the functor of **Term** if **Index** $= 0$.

hilog_arg(+Index, +Term, ?Argument) HiLog

If **Term** is a Prolog term, it has the same behaviour as **arg/3**, but if **Term** is a proper HiLog term, **hilog_arg/3** unifies **Argument** with the (**Index** $+ 1)^{th}$ argument of the Prolog representation of **Term**. Semantically, **Argument** is the **Index**th argument to which the *HiLog functor* of **Term** is applied. The arguments of the **Term** are numbered from 1 upwards. An atomic term is taken to have 0 arguments.

Initially, **Index** must be instantiated to a positive integer and **Term** to any non-variable Prolog or HiLog term. If the initial conditions are not satisfied or I is out of range, the call quietly fails. Note that like **arg/3** this predicate does not succeed for **Index**=0.

Examples:

```
| ?- hilog_arg(2, p(a,b), A).
A = b

| ?- hilog_arg(2, h(a,b), A).
A = b

| ?- hilog_arg(3, X(a,b,c), A).
X = _595820
A = c

| ?- hilog_arg(1, map(f)(a,b), A).
A = a

| ?- hilog_arg(2, map(f)(a,b), A).
A = b

| ?- hilog_arg(1, (a+b)(foo,bar), A).
A = foo

| ?- hilog_arg(1, apply(foo), A).
A = foo

| ?- hilog_arg(1, apply(foo,bar), A).
A = bar
```

Note the difference between the last two examples. The difference is due to the fact that `apply/1` is a Prolog term, while `apply/2` is a proper HiLog term.

`?Term =.. [?Functor |?ArgList]`
Succeeds when `Term` is any (Prolog or) HiLog term, `Functor` is its Prolog functor and `ArgList` is the list of its arguments. The use of `=../2` (pronounced *univ*) although convenient, can nearly always be avoided. Whenever efficiency is critical, it is advisable to use the predicates `functor/3` and `arg/3`, since `=../2` is implemented by calls to these predicates. The behaviour of `=../2` is as follows:

- If initially `Term` is uninstantiated, then the list in the second argument of `=../2` must be instantiated either to a *proper list* (list of determinate length) whose head is an atom, or to a list of length 1 whose head is a number.
- If the arguments of `=../2` are both uninstantiated, or if either of them is not what is expected, `=../2` aborts, producing an appropriate error message.

Examples:

```
| ?- X - 1 =.. L.
X = _595692
L = [-,_595692,1]
```

```
| ?- p(a,b,c) =.. L.
L = [p,a,b,c]

| ?- h(a,b,c) =.. L.
L = [apply,h,a,b,c]

| ?- map(p)(a,b) =.. L.
L = [apply,map(p),a,b]

| ?- T =.. [foo].
T = foo

| ?- T =.. [3|X].
T = 3
X = []

| ?- T =.. [apply,X,a,b].
T = apply(X,a,b)

| ?- T =.. [1,2].
++Error: Wrong type(s) in argument 2 of =../2
Aborting...

| ?- T =.. [a+b,2].
++Error: Wrong type(s) in argument 2 of =../2
Aborting..

| ?- X =.. [foo|Y].
++Error: Argument 2 of =../2 is not a proper list
Aborting...
```

Exceptions:

instantiation_error Argument 2 of `=../2` is not a proper list.

type_error Head of argument 2 of `=../2` is not an atom or number.

`?Term ^=.. [?F |?ArgList]` HiLog

When `Term` is a Prolog term, this predicate behaves exactly like the Prolog `=../2`. However when `Term` is a proper HiLog term, `^=../2` succeeds unifying `F` to its HiLog functor and `ArgList` to the list of the arguments to which this HiLog functor is applied. Like `=../2`, the use of `^=../2` can nearly always be avoided by using the more efficient predicates **hilog_functor/3** and **hilog_arg/3**. The behaviour of `^=../2`, on HiLog terms is as follows:

- If initially `Term` is uninstantiated, then the list in the second argument of `^=../2` must be instantiated to a *proper list* (list of determinate length) whose head can be any Prolog or HiLog term.

- If the arguments of `^=../2` are both uninstantiated, or if the second of them is not what is expected, `^=../2` aborts, producing an appropriate error message.

Examples:

```
| ?- p(a,b,c) ^=.. L.
L = [p,a,b,c]

| ?- h(a,b,c) ^=.. L.
L = [h,a,b,c]

| ?- map(p)(a,b) ^=.. L.
L = [map(p),a,b]

| ?- T ^=.. [X,a,b].
T = apply(X,a,b)

| ?- T ^=.. [2,2].
T = apply(2,2)

| ?- T ^=.. [a+b,2].
T = apply(a+b,2)

| ?- T ^=.. [3|X].
++Error: Argument 2 of ^=../2 is not a proper list
Aborting...
```

Exceptions:

instantiation_error Argument 2 of `^=../2` is not a proper list.

copy_term(+Term, -Copy)

Makes a `Copy` of `Term` in which all variables have been replaced by brand new variables which occur nowhere else. It can be very handy when writing (meta-)interpreters for logic-based languages. The version of **copy_term/2** provided is *space efficient* in the sense that it never copies ground terms. Predicate **copy_term/2** has no associated errors or exceptions.

Examples:

```
| ?- copy_term(X, Y).

X = _598948
Y = _598904

| ?- copy_term(f(a,X), Y).

X = _598892
Y = f(a,_599112)
```

name(?Constant, ?CharList)

The standard predicate **name/2** performs the convertion between a constant and its character list representation. If **Constant** is supplied (and is any atom or number), **CharList** is unified with a list of ASCII codes representing the *"name"* of the constant. In that case, **CharList** is exactly the list of ASCII character codes that appear in the printed representation of **Constant**. If on the other hand **Constant** is a variable, then **CharList** must be a proper list of ASCII character codes. In that case, **name/2** will convert a list of ASCII characters that can represent a number to a number rather than to a character string. As a consequence of this, there are some atoms (for example **'18'**) which cannot be constructed by using **name/2**. If conversion to an atom is preferred in these cases, the standard predicate **atom_chars/2** should be used instead. The syntax for numbers that is accepted by **name/2** is exactly the one which **read/1** accepts. Predicate **name/2** is provided for backwards compatibility with older Prologs, and to be used in the place of predicate **number_chars/2** that is currently not provided. We intend to supply this predicate in a future release.

It is advisable that new programs use the predicate **atom_chars/2** described below.

In Version 1.4.0 predicate **name/2** is not yet implemented for converting from a real number to its character list representation, and if the representation of a real is provided as **CharList**, it will be converted to an atom.

If both of the arguments of **name/2** are uninstantiated or **CharList** is not a proper list of ASCII characters, **name/2** will abort and an error message will be sent to the standard error stream.

Examples:

```
| ?- name('Foo', L).
L = [70,111,111]

| ?- name([], L).
L = [91,93]

| ?- name(431, L).
L = [52,51,49]

| ?- name(X, [102,111,111]).
X = foo

| ?- name(X, []).
X = ''

| ?- name(X, "Foo").
X = 'Foo'

| ?- name(X, [52,51,49]).
X = 431
```

```
| ?- name(X, [45,48,50,49,51]), integer(X).
X = -213

| ?- name(3.14, L).
++Error: Predicate name/2 for reals is not implemented yet
Aborting...
```

Exceptions:

instantiation_error Both arguments are uninstantiated, or argument 2 of **name/2** contains a variable or is not a proper list.

type_error `Constant` is not a variable, an atom or a number.

range_error `CharList` is not a list of ASCII characters.

implementation_error `Constant` is a real number (conversion from a real to its character list representation is not implemented yet).

atom_chars(?Atom, ?CharList)

The standard predicate **atom_chars/2** performs the convertion between an atom and its character list representation. If `Constant` is supplied (and is an atom), `CharList` is unified with a list of ASCII codes representing the *"name"* of that atom. In that case, `CharList` is exactly the list of ASCII character codes that appear in the printed representation of `Atom`. If on the other hand `Atom` is a variable, then `CharList` must be a proper list of ASCII character codes. In that case, `Atom` is instantiated to an atom containing exactly those characters, even if the characters look like the printed representation of a number.

If both of the arguments of **atom_chars/2** are uninstantiated or `CharList` is not a proper list of ASCII characters, **atom_chars/2** aborts, and an error message will be sent to the standard error stream.

Examples:

```
| ?- atom_chars('Foo', L).
L = [70,111,111]

| ?- atom_chars([], L).
L = [91,93]

| ?- atom_chars(X, [102,111,111]).
X = foo

| ?- atom_chars(X, []).
X = ''

| ?- atom_chars(X, "Foo").
X = 'Foo'
```

```
| ?- atom_chars(X, [52,51,49]).
X = '431'

| ?- atom_chars(X, [52,51,49]), integer(X).
no

| ?- atom_chars(X, [52,Y,49]).
! Instantiation error in argument 2 of atom_chars/2
! Aborting...

| ?- atom_chars(431, L).
! Type error: in argument 1 of atom_chars/2
! atom expected, but something else found
! Aborting...
```

Exceptions:

`instantiation_error` Both arguments are uninstantiated, or argument 2 of predicate `atom_chars/2` contains a variable or is not a proper list.

`type_error` `Atom` is not a variable or an atom.

`range_error` `CharList` is not a list of ASCII characters.

6.6 All Solutions Predicates

Often there are many solutions to a problem and it is necessary or convenient to gather those solutions together explicitly. This can be achieved by repeatedly backtracking and building up a list of the solutions. The following predicates are provided to automate this process.

`setof(?X, +Goal, ?Set)`

This predicate may be read as "`Set` is the set of all instances of `X` such that `Goal` is provable". If `Goal` is not provable, `setof/3` fails. The term `Goal` specifies a goal or goals as in `call(Goal)`. `Set` is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 6.7). If there are uninstantiated variables in `Goal` which do not also appear in `X`, then a call to this evaluable predicate may backtrack, generating alternative values for `Set` corresponding to different instantiations of the free variables of `Goal`. Variables occurring in `Goal` will not be treated as free if they are explicitly bound within `Goal` by an existential quantifier. An existential quantification can be specified as:

```
Y ^ G
```

meaning there exists a `Y` such that `G` is true, where `Y` is some Prolog term (usually, a variable).

`bagof(X, Goal, Bag)`
This predicate has the same semantics as `setof/3` except that the third argument returns an unordered list that may contain duplicates.

`findall(X, Goal, List)`
Similar to `bagof/3`, except that variables in `Goal` that do not occur in `X` are treated as existential, and alternative lists are not returned for different bindings of such variables. If `Goal` is unsatisfiable, `findall/3` succeeds binding `List` to the empty list.

`tfindall(X, Goal, List)` Tabling
Like `findall/3`, `tfindall/3` treats all variables in `Goal` that do not occur in `X` as existential. However in `tfindall`, `Goal` must be a single tabled predicate.

`tfindall/3` allows the user to build programs that use stratified aggregation. If the table to `Goal` is incomplete, `tfindall/3` suspends until the table has been completed, and only then computes `List`. See Section 5 for further discussion of `tfindall/3`. If `Goal` is unsatisfiable, `tfindall/3` succeeds binding `List` to the empty list.

Exceptions:

`table_error` `Goal` is not a call to a tabled predicate.

`X ^ Goal`
The system recognises this as meaning there exists an `X` such that `Goal` is true, and treats it as equivalent to `call(Goal)`. The use of this explicit existential quantifier outside predicates `setof/3` and `bagof/3` constructs is superfluous.

6.7 Comparison

The evaluable predicates described in this section are meta-logical. They are used to compare and order terms, rather than to evaluate or process them. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. Each of these predicates simply succeeds or fails; there is no side-effect, substitution or error condition associated with them. The predicates described in this section should *not* be used when what the user really wants is arithmetic comparison predicates (see section 6.2) or unification predicates (see section 6.3).

The predicates described take into account a standard total ordering of terms, which has as follows:

$$variables \ @< \ floating\ point\ numbers \ @< \ integers \ @< \ atoms \ @< \ compound\ terms$$

Within each one of the categories, the ordering is as follows:

- variables are put in a standard order (roughly, the oldest first — the order is *not* related to the names of variables). Note that two anonymous variables are not identical terms. Unfortunately in the current implementation of our system (Version 1.4.0) variables "tend to move" rather quickly as a result of unification, and thus the ordering may not continue to hold if the variables get unified to some other variables. We intend to ameliorate this bug in future releases.
- floating point numbers and integers are put in numeric order, from $-\infty$ to $+\infty$. Note that a floating point number is always less than an integer.
- atoms are put in alphabetical (i.e. ASCII) order;
- compound terms are ordered first by arity, then by the name of their principal functor and then by their arguments (in a left-to-right order).
- lists are compared as ordinary compound terms having arity 2 and functor `'.'`.

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

```
[ X, 3.14, -9, fie, foe, fum(X), [X], X = Y, fie(0,2), fie(1,1) ]
```

The basic predicates for comparison of arbitrary terms are:

`T1 == T2`
: Tests if the terms currently instantiating `T1` and `T2` 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 variables. However, the question

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

succeeds because the first goal unifies the two variables (see section 6.3).

`T1 \== T2`
: Tests if the terms currently instantiating `T1` and `T2` are not literally identical.

`T1 @< T2`
: Succeeds if term `T1` is before term `T2` in the standard order.

`T1 @> T2`
: Succeeds if term `T1` is after term `T2` in the standard order.

`T1 @=< T2`
Succeeds if term `T1` is not after term `T2` in the standard order.

`T1 @>= T2`
Succeeds if term `T1` is not before term `T2` in the standard order.

Some further predicates involving comparison of terms are:

`compare(?Op, +T1, +T2)`
Succeeds if 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`. Predicate **`compare/3`** has no associated error conditions.

`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`. The time to perform the sorting is $O(nlogn)$ where n is the length of list `L1`.

Exceptions:

`instantiation_error` Argument 1 of **`sort/2`** is a variable or is not a proper list.

Examples:

```
| ?- sort([3.14,X,a(X),a,2,a,X,a], L).

L = [X,3.14,2,a,a(X)];

no
```

`keysort(+L1, ?L2)`
The list `L1` must consist of elements of the form **`Key-Value`**. These elements are sorted into order according to the value of **`Key`**, yielding the list `L2`. Unlike **`sort/2`**, in **`keysort/2`** no merging of elements takes place.

6.8 Meta-Predicates

`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/1` fails.

Exceptions:

`instantiation_error` Argument 1 of `call/1` is not instantiated.

`type_error` Argument 1 of `call/1` is not a callable term.

`+X`

(where `X` is a variable) executes exactly the same as `call(X)`. However, the explicit use of `call/1` is considered better programming practice. The use of a top level variable subgoal elicits a warning from the compiler.

`once(+X)`

The same as `call(X), !`.

Exceptions: The same as `call/1`.

6.9 Information about the State of the Program

In XSB various aspects of the program state — information about predicates, modules, clauses, and their object files can all be inspected in ways similar to many Prolog systems. However, because the atom-based module system of XSB may associate structures with particular modules, predicates are provided to inspect these elements as well. The following descriptions of *state* predicates use the terms *predicate indicator*, *term indicator* and *current module* to mean the following:

- By *predicate indicator* we mean a *compound term* of the form `M:F/A` or simply `F/A`. When the predicate indicator is fully instantiated, `M` and `F` are atoms representing the *module name* and the *functor* of the predicate respectively and `A` is a non negative integer representing its *arity*.

  Example: `usermod:append/3`

- By *term indicator* we mean a predicate or function symbol of arity N followed by a sequence of N variables (enclosed in parentheses if N is greater than zero). A term indicator may optionally be prefixed by the module name, thus it can be of the form `M:Term`.

  Example: `usermod:append(_,_,_)`

- A module `M` becomes a *current (i.e. "known") module* as soon as it is loaded in the system or when another module that is loaded in the system imports some predicates from module `M`.

  Note that due to the dynamic loading of XSB, a module can be current even if it has not been loaded, and that some predicates of that module may not be defined. In fact, a module can be current even if it does not exist. This situation occurs when a predicate is improperly imported from a non-existent module. Despite this, a module can never lose the property of being *current*.

`current_input(?Stream)`
: Succeeds iff stream `Stream` is the current input stream, or procedurally unifies `Stream` with the current input stream. There are no error conditions for this predicate.

`current_output(?Stream)`
: Succeeds iff stream `Stream` is the current output stream, or procedurally unifies `Stream` with the current output stream. There are no error conditions for this predicate.

`current_module(?Module)`
: The standard predicate `current_module/1` allows the user to check whether a given module is *current* or to generate (through backtracking) all currently known modules. Succeeds iff `Module` is one of the modules in the database. This includes both user modules and system modules.

  Note that predicate `current_module/1` succeeds for a given module even if that module is not a real module (in the sense taht it does not export any predicates). There are no error conditions associated with this predicate; if its argument does not unify with one of the current modules, `current_module/1` simply fails.

`current_module(?Module, ?ObjectFile)`
: Predicate `current_module/2` gives the relationship between the modules and their

associated object file names. The file name `ObjectFile` must be absolute and always ends with `'.O'`.

It is possible for a current module to have no associated file name (as is the case for modules like `"usermod"` and `"global"`), or for the system to be unable to determine the file name of a current module. In both cases, predicate `current_module/1` will succeed for this module, while `current_module/2` will fail. The system is unable to determine the file name of a given module if that module is not in one of the directories of the search path (see Section 3.5). Once again, there are no error conditions associated with this predicate; if the arguments of `current_module/2` are not correct, or `Module` has no associated `File`, the predicate will simply fail.

`current_atom(?Atom_Indicator)`

Generates (through backtracking) all currently known atoms, and unifies each in turn with `Atom_Indicator`.

`current_functor(?Predicate_Indicator)`

The standard predicate `current_functor/1` can be used to find all the currently known terms appearing in a particular current module. It succeeds iff `Predicate_Indicator` is a predicate indicator for any term that appears in the database. Note that this includes terms both in system and in user defined modules, even terms that may be not yet loaded in the system. The behaviour of `current_functor/1` may be contrasted with that of `current_predicate/1`, which reports only those predicates which have been loaded in the system (both Prolog and foreign predicates) or are dynamic predicates. There are no error conditions associated with this predicate; if its argument is not a predicate indicator the predicate simply fails.

Predicate `current_functor/1` comes in two flavors depending on the form of its argument (`Predicate_Indicator`):

1. If `Predicate_Indicator` has the form `Module:Functor/Arity`, then predicate `current_functor/1` backtracks through all the current modules of the system (user defined, system defined and global modules).
2. If on the other hand, `Predicate_Indicator` is uninstantiated or has the form `Functor/Arity`, then predicate `current_functor/1` backtracks only through the terms appearing in the global modules of the system (in other words searches only modules `"usermod"` and `"global"`). This flavor is only for convenience, since this is the common use of predicate `current_functor/1`. Note that all the following are equivalent:

```
| ?- current_functor(Functor/Arity).
| ?- current_functor(Predicate).
| ?- current_functor(usermod:Predicate).
| ?- current_functor(global:Predicate).
```

So, to backtrack through all of the functors of positive arity (function and predicate symbols) that appear in the global modules of the system regardless of whether they are system or a user defined, use:

```
| ?- current_functor(Functor/Arity), Arity > 0.
```

`current_functor(?Name, ?Term_Indicator)`

Succeeds iff `Term_Indicator` is the most general term corresponding to one of the currently known terms having `Name` as their functor appearing in a current module. (Both system and user defined modules are checked). Or procedurally, `current_functor/2` unifies `Name` with the name of a functor known to the database, and `Term_Indicator` with the most general term corresponding to that functor. The flavors of this predicate are analogous to the ones of `current_functor/1` according to whether `Term_Indicator` has one of the following two forms:

1. `Module:Term`.
2. `Term` (for global modules).

If `Term_Indicator` is uninstantiated, then this predicate succeeds only for global modules. As in `current_functor/1` even unloaded predicates are reported (if they have been imported and are are known to the database).

For example, if a predicate `foo/2` and and a function symbol `foo/1` are defined into module `blah`, the following query will return:

```
| ?- current_functor(foo, blah:Term).

Term = foo(_638788,_638792);

Term = foo(_638788);

no
```

If a module is specified, `current_functor/2` succeeds only for those functors (function and predicate symbols) which are defined in that module. Unless the module is one of the global modules, `current_functor/2` fails for the predicates which are imported into that module.

On the other hand, the goal:

```
| ?- current_functor(Name, Term).
```

can be used to backtrack through every known term `Term` in the global modules of XSB's database that has `Name` as its functor.

Note that the order of term generation is undetermined. Once again, there are no error conditions associated with this predicate; if its arguments are inappropriate, the predicate simply fails.

`current_predicate(?Predicate_Indicator)`

The predicate `current_predicate/1` can be used to find all the predicates that are defined and loaded in a particular current module. The module can be either a Prolog or a foreign module (see Chapter 11). This predicate succeeds iff `Predicate_Indicator` is a predicate indicator for one of the procedures (both Prolog and foreign language ones) that are loaded in the database or that are dynamic. Note that this includes procedures both in system and in user defined modules. Unlike `current_functor/1` which reports all predicates that are somehow known to the database, `current_predicate/1` reports only those predicates that are either created dynamically (for example using `assert/1`) or loaded in the system. (I.e. It excludes those predicates which have been imported, but not loaded). There are no error conditions associated with this predicate; if its argument is not what it should be, the predicate simply fails.

Like `current_functor/1`, predicate `current_predicate/1` comes in two flavors depending on the form of its argument (`Predicate_Indicator`).

1. If `Predicate_Indicator` has the form `Module:Functor/Arity`, then predicate `current_predicate/1` unifies the predicate indicator with predicates in all current modules. (User defined, system defined and global modules).
2. If on the other hand, `Predicate_Indicator` is uninstantiated or has the form `Functor/Arity`, then `current_predicate/1` backtracks only through the predicates loaded in the global modules of the system (in other words searches only modules `"usermod"` and `"global"`). This flavor is only for convenience, since this is the common use of predicate `current_predicate/1`. Note that all the following are equivalent:

```
| ?- current_predicate(Functor/Arity).
| ?- current_predicate(Predicate).
| ?- current_predicate(usermod:Predicate).
| ?- current_predicate(global:Predicate).
```

So, to backtrack through all of the predicates defined and loaded in module `blah`, regardless of whether `blah` is a system or a user defined module [1] , use:

```
| ?- current_predicate(blah:Predicate).
```

In this case `Predicate` will have the form: `Functor/Arity`.

To backtrack through all predicates defined and loaded in any current module, use:

```
| ?- current_predicate(Module:Functor/Arity).
```

This succeeds once for every predicate that is loaded in XSB's database.

To find the predicates having arity 3 that are loaded in the global modules of the system, use:

[1]The only limitation is that `blah` must indeed be a module in the sense that it exports at least one symbol.

```
| ?- current_predicate(Functor/3).
```

while to find all predicates loaded in the global modules of the system regardless of their arity, use:

```
| ?- current_predicate(Predicate).
```

current_predicate(?Name, ?Term_Indicator)

Succeeds iff `Term_Indicator` is the most general term corresponding to one of the predicates having functor `Name` that are defined and loaded in a particular module in the database. (The module can be either system or user defined). Or procedurally, `current_predicate/2` unifies `Name` with the name of a loaded predicate, and `Term_Indicator` with the most general term corresponding to that predicate. The flavors of this predicate are analogous to those of `current_predicate/1` and behave according to whether `Term_Indicator` has one of the following two forms:

1. `Module:Term`.
2. `Term` (module is assumed to be global or usermod).

If `Term_Indicator` is uninstantiated, then this predicate succeeds only for global modules. Like `current_predicate/1` only predicates that have the a property (see `predicate_property/2`) in the following set:

{ `loaded, dynamic, foreign` }

are reported.

For example, if predicates `foo/1` and `foo/3` are defined and loaded into module `blah`, the following query will return:

```
| ?- current_predicate(foo, blah:Term).

Term = foo(_638788,_638792,_638796);

Term = foo(_638788);

no
```

If a module is specified, `current_predicate/2` succeeds only for those predicates which are defined and loaded in that module. Unless the module is one of the global modules, `current_predicate/2` fails for those predicates which are imported into that module.

On the other hand, the goal:

```
| ?- current_predicate(Name, Term).
```

can be used to backtrack through every predicate that is loaded in the global modules of XSB's database.

Note that the order of term generation is undetermined. Once again, there are no error conditions associated with this predicate; if its argument is not what it should be, the predicate simply fails.

`predicate_property(?Term_Indicator, ?Property)`

The standard predicate `predicate_property/2` can be used to find the properties of any predicate which is visible to a particular module. Succeeds iff `Term_Indicator` is a term indicator for a current predicate whose principal functor is a predicate having `Property` as one of its properties. Or procedurally, `Property` is unified with the currently known properties of the predicate having `Term_Indicator` as its skeletal specification.

A brief description of `predicate_property/2` is as follows:

- If `Term_Indicator` is instantiated, then `Property` is successively unified with the various properties associated with `Term_Indicator`.
- If `Property` is bound to a valid predicate property, then `predicate_property/2` successively unifies `Term_Indicator` with the skeletal specifications of all known to the system predicates having the specified `Property`.
- If `Term_Indicator` 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`.
- If `Term_Indicator` is a skeletal specification not a known to the system, or `Property` is not a valid predicate property, the call simply fails.

For example, all the loaded predicate skeletal specifications in module `"usermod"` may be enumerated using:

```
| ?- predicate_property(Pred, loaded).
```

Also the following query finds all predicate skeletal specifications that are exported by module `blah`:

```
| ?- predicate_property(blah:Pred, exported).
```

Currently, the following properties are associated with predicates either implicitly or by declaration (where double lines show property categories, and a predicate can have at most one property of each category).

Property	*Explanation*
`unclassified`	The predicate symbol is not yet classified according to this category. This property has various meanings. Usually for exported predicate symbols in system or user defined modules it means that the predicate is yet unloaded (because it has not been used). In global modules it usually means that the predicate is either a function symbol, or an unloaded predicate symbol (including constants).
`dynamic`	The predicate is dynamic.
`loaded`	The predicate (including internal predicates) is a Prolog predicate loaded into the module in question; this is always the case for predicates in global modules.
`unloaded`	The predicate is yet unloaded into the module in question.
`foreign`	The predicate is a foreign predicate. This implies that the predicate is already loaded in the system, because currently there is no way for XSB to know that a predicate is a foreign predicate until it is loaded in the system.
`exported`	The predicate symbol is exported by the module in question; in other words the predicate symbol is visible to any other module in the system.
`local`	The predicate symbol is local to the module in question.
`imported_from(Mod)`	The predicate symbol is imported into the module in question from module `Mod`.
`spied`	The predicate symbol has been declared spied (either conditionally or unconditionally).
`tabled`	The predicate has been declared tabled.
`built_in`	The predicate symbol has the same Functor and Arity as one of XSB's builtin (standard) predicates.

Finally, since `dynamic` is usually declared as an operator with precedence greater than 999, writing the following:

```
| ?- predicate_property(X, dynamic).
```

will cause a syntax error. The way to achieve the desired result is to parenthesize the operator like in:

```
| ?- predicate_property(X, (dynamic)).
```

`module_property(?Module, ?Property)`

The standard predicate `module_property/2` can be used to find the properties of any current module. Succeeds iff `Module` is the name of a current module having `Property` as one of its properties. Or procedurally, `Property` is unified with the currently known properties of the module having `Module` as its name.

Currently, the following properties are associated with modules implicitly

Property	*Explanation*
`unloaded`	The module (including system modules) though it is current, is yet unloaded in the system.
`loaded`	The module (including system modules) is loaded in the system; this is always the case for global modules.

`listing`

Lists in the current output stream the clauses for all dynamic predicates found in module `usermod`. Note that `listing/0` does not list any compiled predicates unless they have the `dynamic` property (see `predicate_property/2`). A predicate gets the `dynamic` property when it is explicitly declared as `dynamic`, or automatically acquires it when some clauses for that predicate are asserted in the database. In cases where a predicate was compiled but converted to `dynamic` by asserting additional clauses for that predicate, `listing/0` will just display an indication that there exist compiled clauses for that predicate and only the dynamically created clauses of the predicate will be listed. For example:

```
| ?- [user].
[Compiling user]
a(X) :- b(X).
a(1).
[user compiled, cpu time used:  0.3 seconds]
[user loaded]

yes
| ?- assert(a(3)).

yes
| ?- listing.

a(A) :-
      $compiled.
a(3).

yes
```

Predicate `listing/0` always succeeds. The query:

```
| ?- listing.
```

is just a notational shorthand for the query:

```
| ?- listing(X).
```

`listing(+Predicate_Indicator)`

If `Predicate_Indicator` is a variable then `listing/1` is equivalent to `listing/0`. If `Predicate_Indicator` is an atom, then `listing/1` lists the dynamic clauses for all predicates of that name found in module `usermod` of the database. The argument `Predicate_Indicator` can also be a predicate indicator of the form `Name/Arity` in which case only the clauses for the specified predicate are listed. Finally, it is possible for `Predicate_Indicator` to be a list of predicate indicators and/or atoms; e.g.

```
| ?- listing([foo/2, bar, blah/4]).
```

If `Predicate_Indicator` is not a variable, an atom or a predicate indicator (or list of predicate indicators) of the form `Name/Arity`, predicate `listing/1` will simply fail.

In future releases of XSB, we intend to allow the user to specify a predicate indicator of the form `Module:Name/Arity` as argument of `listing/1`.

`xsb_flag(?Flag_Name, ?Value)`

Succeeds iff the current value of the XSB flag `Flag_Name` is `Value`. So, one can enumerate all the flag names which the system currently understands, together with their current values by using the following query:

```
| ?- xsb_flag(FlagName, Value).
```

The flag names currently supported are:

Flag Name	*Purpose*
`debugging`	`"on"` iff debug mode is on; `"off"` otherwise.
`tracing`	`"on"` iff trace mode is on; `"off"` otherwise.
`host_type`	The type of machine for which XSB is installed.
`xsb_directory`	The absolute path name of the directory where XSB is installed.
`version`	An atom indicating what version of XSB is being run.

`hilog_symbol(?Symbol)`

Succeeds iff `Symbol` has been declared as a HiLog symbol, or procedurally unifies `Symbol` with one of the currently known (because of a prior declaration) HiLog symbols. The HiLog symbols are always atoms, but if the argument of `hilog_symbol`, though instantiated, is not an atom the predicate simply fails. So, one can enumerate all the HiLog symbols by using the following query:

```
| ?- hilog_symbol(X).
```

current_op(?Precedence, ?Type, ?Name)
This predicate is used to examine the set of operators currently in force. It succeeds when the atom `Name` is currently an operator of type `Type` and precedence `Precedence`. None of the arguments of `current_op/3` need to be instantiated at the time of the call, but if they are, they must be of the following types:

Precedence it must be an integer in the range from 1 to 1200.

Type it must be one of the atoms:

`xfx xfy yfx fx fy hx hy xf yf`

Name it must be either an atom or a list of atoms.

Exceptions (not yet implemented):

domain_error `Precedence` is not between 1–1200, or `Type` is not one of the listed atoms.

type_error `Name` is not an atom.

hilog_op(?Precedence, ?Type, ?Name)
This predicate has exaclty the same behaviour as `current_op/3` with the only difference that `Type` can only have the values `hx` and `hy`.

6.10 Modification of the Database

XSB provides the usual simple predicates for modifying the dynamic database. By default, asserted clauses are indexed on the first argument, using a fixed-size hash table, but this can be changed using the directive (and builtin) `index/3`. With this builtin, the user can specify multiple indexes and joint indexes in a variety of combinations. Space is dynamically allocated when a new clause is asserted and released when it is retracted.

Note that the *"immediate semantics"* of dynamic predicates is used, not the so-called *"logical semantics"* of assert/retract. This means that significant care must be taken when modifying the definition of a predicate which is currently being executed. Notice that this makes some operations difficult. For example, one might try to retract from dynamically asserted predicates, `p/1` and `q/1`, exactly their intersection, by issuing the following query:

```
:- p(X), q(X), retract(p(X)), retract(q(X)), fail.
```

This code (even with `retract/1` replaced by `retractall/1`) does NOT work, since for nondeterministic `p/1`, the system will have to backtrack into `p/1` after it has retracted from it. One solution is to use `findall/3` to collect the intersection first, before deleting. Another is to use the predicates `retract_nr/1` and `reclaim_space/1`, described below.

XSB does not at this time support dynamic predicates in compiled code. The only way to generate dynamic code is by asserting it, or by using `load_dyn/1` to read it from a file and assert it. There is a `dynamic/1` predicate that initializes a predicate to be empty, so that when called before any asserts, it will quietly fail instead of issuing an "**Undefined predicate**" error message.

`assert(+Clause)`
: adds a dynamic clause, `Clause`, to the database. `Clause` must be of one of the forms: `Head` or `Head :- Body`. Note, that because of the precedence of `:-/2`, using the second form requires an extra set of parentheses: `assert((Head :- Body))`. First-argument indexing is done.

`asserta(+Clause)`
: adds a dynamic clause, `Clause`, to the database *before* any other clauses for the same predicate currently in the database. Otherwise it is the same as `assert/1`.

`assertz(+Clause)`
: adds a dynamic clause, `Clause`, to the database *after* any other clauses for the same predicate currently in database. Otherwise it is the same as `assert/1`.

`retract(+Clause)`
: removes through backtracking all clauses in the database that match with `Clause`. `Clause` must be of one of the forms: `Head` or `Head :- Body`. Note, that because of the precedence of `:-/2`, using the second form requires an extra set of parentheses: `retract((Head :- Body))`. Space is reclaimed when a clause is retracted.

`retractall(+Head)`
: removes every clause in the database whose head matches with `Head`. The predicate whose clauses have been retracted retains the `dynamic` property (contrast this behaviour with that of predicates `abolish/[1,2]` below). Predicate `retractall/1` is determinate and always succeeds. The term `Head` is not further instantiated by this call.

`abolish(+PredSpec)`
: removes the definition of the specified predicate. `PredSpec` is of the form `Pred/Arity`. Everything about the abolished predicate is completely forgotten by the system (including the `dynamic` property). There is also an `abolish/2` which takes `Pred` and `Arity` as its two arguments.

`clause(+Head,?Body)`
: returns through backtracking all dynamic clauses in the database whose head matches `Head` and Body matches `Body`. For facts the `Body` is `true`.

retract_nr(+Clause)

performs just as **retract/1** does, except that it does not reclaim the space used by the retracted clause. This is provided to allow programmers to modify dynamic clauses while executing them (a practice that is strongly discouraged.) For example, to retract an intersection, as described above, one could do:

```
:- p(X), q(X), retract_nr(p(X)), retract_nr(q(X)), fail.
```

In this case, the **retract_nr** deletes the clauses, but will not drastically modify the clause data structure, and this code will execute. Of course, space is not reclaimed for further use (but see **reclaim_space/1** below). Predicate **retract_nr/1** is not a standard predicate but must be imported from module **assert**. The use of this predicate is discouraged; it is provided for (partial) compatibility with the **retract/1** predicate provided by SB-Prolog that did not reclaim space.

reclaim_space(+Head)

runs through the dynamic code for the predicate indicated by **Head**, and reclaims space for any clauses that have been deleted from that predicate by **retract_nr**. This cannot safely be used when execution is still within some invocation of the specified predicate, or will backtrack into such a scope. To complete our example of retracting the intersection of dynamic predicates:

```
:- p(X), q(X), retract_nr(p(X)), retract_nr(q(X)), fail ;
        reclaim_space(p(_)), reclaim_space(q(_)).
```

would do the trick. Notice that the **reclaim_space** calls cannot appear before the fail, but must be executed after exectuion has completely failed out of `q(X)` and `p(X)`. Predicate **reclaim_space/1** is not a standard predicate but must be imported from module **assert**. The use of this predicate is discouraged; it is provided for (partial) compatibility with SB-Prolog.

index(+PredSpec, +IndexArgSpec, +HashTableSize)

is an executable predicate that declares the predicate indicated by **PredSpec** to be indexed according to **IndexArgSpec** using a hash table of size **HashTableSize**. After this directive is given, all clauses asserted to **PredSpec** will be so indexed, and whenever a hash table needs to be constructed for the index, it will be created at a size indicated by **HashTableSize**. This directive can either be given as a compiler directive or executed during program execution (analogously to **op/3**.) It is normally given before any clauses are asserted to the predicate. If **IndexArgSpec** is 0 (or []), then no index is kept (which is more efficient for predicates with only one or two clauses.) If **IndexArgSpec** is a negative integer, then indexing information regarding the indicated predicate is deleted, and its handling reverts to the default.

Note that the index directive is a declaration; i.e., it does *not* re-index an already existing predicate but applies only to future asserts.

The indexing that is supported is different for compiled and asserted code. For compiled code `IndexArgSpec` must be an integer which indicates the argument on which an index is to be constructed. For a predicate to which clauses are asserted, `IndexArgSpec` may be a list of `IndexArgs`. An `IndexArg` is an integer that indicates the argument on which to index. So one may specify, for example, `index(p/3,[2,1],150)` to indicate that clauses asserted to the predicate `p/3` should be indexed on both the second and the first argument. So subsequent calls to `p/3` will first check to see if the second argument is nonvariable, and if so use that index. If the second argument is variable, then it will check to see if the first argument is nonvariable and if so, use that index. With this declaration, both hash tables will be of size 150. One may have any number of distinct indexes. For asserted clauses an `IndexArg` may also be a sequence of up to three integers sparated by `+`, e.g., `1+2+3`. This indicates that an index is to be built using those specified field values together. To use such an index in a call, all the indicated fields must be nonvariable.

As an example, one could specify: `index(p/5,[1+2,1,4],300)`. After clauses are asserted to it, a call to `p/5` would first check to see if both the first and second arguments are nonvariable and if so, use an index based on both those values. Otherwise, it would see if the second argument is nonvariable and if so, use an index based on it. Otherwise, it would see if the fourth argument is nonvariable and if so use an index based on it. Otherwise, it would use no index but backtrack through all the clauses in the predicate. (Notice that it may well make sense to include an argument later alone that appears in a joint specification earlier, as 1 in this example, but it never makes sense foring the single argument to appear earlier. In that case the joint index would never be used.)

`dynamic(+PredSpec)`

is an executable predicate. Given a predicate specification (Predicate/Arity), it converts it to a dynamic predicate. If it is not previously defined, it will be initialized to empty (so that calls to it quietly fail, instead of issuing "**Undefined predicate**" error messages.) If it is previously defined and dynamic, it is a noop. If previously defined as compiled, it will be converted to dynamic, which means that clauses can be added, but the compiled portion cannot be manipulated. Note that this can be used like a compiler directive, since it will be passed through to be executed when the module is loaded. Note, however, that the semantics is different from that of the standard when the file contains clauses defining the so-specified predicate.

`table(+PredSpec,+RetSize,+CallSize)`

is an executable predicate, where PredSpec is a predicate specification for a dynamic predicate. (This is also a compiler directive when `PredSpec` specifies a compiled predicate. See the section of this manual on compiler directives.) This predicate declares a dynamic predicate to be tabled. It simply saves information to be used at the time of assert and so it must be called before any clauses are asserted into the specified predicate. `RetSize` must be an integer that specifies the size of the hash table that will be allocated to handle returns; `CallSize` must be an integer and it is used as the size when allocating the hash table for the memoed calls.

6.11 Execution State

`abort`

Abandons the current execution and returns to the top level. This is a fairly drastic predicate and is normally used in one of the following two cases:

- when some error condition or exception has occurred and carrying on the computation is of no further use.
- when using the debugger (see Chapter 7).

Currently, all exception handling routines terminate with a call to predicate `abort/0`, so an exception encountered at some break level other than the top level will return the interpreter at the top level.

The user should be aware of the fact that `abort/0` does not close any files which may have been opened. If the program under execution is doing file manipulation using `see/1` and `tell/1`, then strange behaviour may occur after the program is aborted and restarted, unless the user remembers to close the files himself.

Aborting closes all open tables (those for which it is not known that the answer set is complete). Closed tables, for which the answer set is complete, are unaffected, even if the tables were created during the aborted computation.

`break`

Causes the current execution to be suspended at the beginning of the next call. The interpreter then enters break level 1 and is ready to accept input as if it were at top level. If another call to `break/0` is encountered, it moves up to break level 2, and so on. While execution is done at break level $n > 0$ the prompt changes to n: `?-`.

To close a break level and resume the suspended execution, the user can type the the atom `end_of_file` or the end-of-file character applicable on the system (usually `CTRL-d`). Predicate `break/0` then succeeds (note in the following example that the calls to `break/0` do not succeed), and the execution of the interrupted program is resumed. Alternatively, the suspended execution can be abandoned by calling the standard predicate `abort/0`, which causes a return to the top level.

An example of `break/0` 's use is the following:

```
| ?- break.
[ Break (level 1) ]
1:  ?- break.
[ Break (level 2) ]
2:  ?- end_of_file.
[ End break (level 2) ]

yes
1:  ?-
```

Entering a break closes all open tables (those for which it is not known that the answer set is complete). Closed tables, for which the answer set is complete, are unaffected, even if the tables were created during the computation from which the break was entered.

`halt`
: Exits the XSB session regardless of the break level. On exiting the system cpu and elapsed time information is displayed.

`prompt(+NewPrompt, ?OldPrompt)`
: Sets the prompt of the top level interpreter to `NewPrompt` and returns the old prompt in `OldPrompt`.

An example of `prompt/2` 's use is the following:

```
| ?- prompt('Yes master > ', P).

P = | ?- ;

no
Yes master > true.

yes
Yes master >
```

`cputime(-CPUTime)`
: Returns the `CPUTime` at the time of the call in seconds. The difference between results of successive calls to this predicate can measure the time spent in specific predicates.

`statistics`
: Prints on the current output stream:

- Information about allocation of memory (see Section 3.6) containing the
  - unification stack (pdlstack)
  - tablestack and tableheap

  - global stack (heap) and local (environment) stack
  - trail and choice point stack
  - opentable stack.
- Current use of the memory areas.
- Information about process cpu and clock time.

Additionally, if the emulator is invoked with the `'-s'` option (see Section 3.6), information is printed out about

- Maximum use of the memory areas.

The `'-s'` option slows down the emulator by about 10%.

Example:

```
| ?- statistics.
== Statistics ===
Permanent space: 138705 allocated.
Stack space allocated: 64K+64K+768K+768K+64K
Stacks in use : global 92, local 88, trail 48, cp 196
SLG Stacks in use : table 19304, tab_heap 0, opentables 2
level num: 0
TIME: cputime: 1.80*1 sec, elapsetime 162.87 sec

yes
```

shows how the emulator output looks if it is invoked without the `'-s'` option. Information about the allocation size is provided since the sizes can be changed through emulator options. (See Section 3.6).

Under the '-s' option, the statistics output might look like the following:

Example:

```
== Statistics ===
Permanent space: 130921 allocated.
Stack space allocated: 64K+64K+768K+768K+64K
Stacks in use : global 100, local 88, trail 48, cp 196
SLG Stacks in use : table 19304, tab_heap 0, opentables 2
level num: 0
Max stack used: global 65832, local 460, trail 40464, cp 488
Max stack used: opentable 8, level 1
TIME: cputime: 0.68*1 sec, elapsetime 16.56 sec
```

`analyze_table(+Pred_indicator)`

This predicate provides additional means for the user to profile execution of tabled predicates. Given a predicate indicator for a tabled predicate, **`analyze_table/1`** prints the following information.

- "⟨Num⟩ call hash buckets used out of ⟨Total⟩". The number of call hash buckets containing at least one element.
- "avg call bucket length (of used) ⟨Avg⟩". The average length of call hash buckets containing at least one element.
- "max call bucket length ⟨Length⟩".
- "⟨Num⟩ ret hash buckets used out of ⟨Total⟩". The number of ret hash buckets containing at least one element.
- "avg ret bucket length (of used) ⟨Avg⟩". The average length of ret hash buckets containing at least one element.
- "max ret bucket length ⟨Length⟩".

In Version 1.4.0, **analyze_table/1** is most readily used to determine proper call and return hash sizes. The hash table sizes can be reset using **set_table_prop/3** from the interpreter level (see Section 6.12) or by recompiling the source code with new declarations (see Section 3.9.4).

Example:

```
| ?- analyze_table(ancestor/2).
            1 hash buckets full out of 97
            avg call bucket length (of filled) 1.00
            max call bucket length 1
            ret buckets used 97 out of 97
            avg return bucket length 11.92
            max ret bucket length 17

yes
```

The output of **analyze_table** indicates that the return hash size for ancestor is too small for the query. Changing the return hash size will speed up similar queries which add answers to tables, but the size of the return hash table has no effect on the speed of queries which use completed tables.

unix(+SystemCall)

Calls the operating system with the atom **SystemCall** as argument. It succeeds if **SystemCall** is executed successfully, otherwise it fails. As a notational convenience, the user can supply **SystemCall** in the form of a list (something currently not possible for **unix/2**).

For example, the call:

```
| ?- unix('echo $HOME').
```

will output in the current output stream of XSB the name of the user's home directory; while the call:

```
| ?- File = 'test.c', unix(['cc -c ', File]).
```

will call the C compiler to compile the file `test.c`.

Note that since `unix/1` is executed by forking off a shell process, it cannot be used, for example, to change the working directory of the interpreter. For that reason the standard predicate `cd/1` described below should be used.

`unix(+SystemCall, -Result)`
Calls the operating system with the atom `SystemCall` as argument and returns the result of the call in `Result`. In comparison with `unix/1` this predicate always succeeds, even if the `SystemCall` cannot be successfully executed.

`ls`
Lists in the current output stream the files in the system's current directory and succeeds if it can do so. It is the same as `unix('ls -F', 0)`.

`cd(+Dir)`
Changes the interpreter's working directory to `Dir`. If the directory specified does not exist or is not a directory, or the user does not have execute permission for that directory, predicate `cd/1` simply fails raising no permission error.

Exceptions:

`instantiation_error` `Dir` is not instantiated at the time of call.

`type_error` `Dir` is not an atom.

`edit(+Module)`
Provided that the environment variable `EDITOR` has been set, a call to `edit(foo)` will call the default editor on the file named `foo.P`. The argument to `edit/1`, should be instantiated, it can be an absolute or a relative file name, but it should *not* contain the suffix `.P`. Users can also set their preferred options of calling the default editor by setting an environment variable named `EDITOR_OPTIONS`.

Examples of possible uses of predicate `edit/1` are:

1. If the environment variables have been set as follows:

   ```
   setenv EDITOR /usr/ucb/vi
   setenv EDITOR_OPTIONS -l
   ```

   a call like:

   ```
   | ?- edit(foo).
   ```

   will call the vi editor in the mode where left and right parentheses and curly brackets are checked for balance for the file `foo.P` in the current working directory.

2. If, on the other hand, they have been set as follows:

   ```
   setenv EDITOR /usr/local/bin/emacs
   setenv EDITOR_OPTIONS -r
   ```

   a call like:

   ```
   | ?- edit('~/foo').
   ```

   will call the emacs editor in reverse video for the file `foo.P` in user's home directory.

6.12 Tabling

In XSB, tables are designed so that they can be used transparently by computations. However, during program development, it is often useful to be able to explicitly inspect a table, or alter its state. The following predicates are supplied for these purposes. Information for optimizing tables can be found in Section 9.2.

In the following predicates, `Skeleton` refers to information about the function and arity of a predicate. If p/2 is a predicate, its skeleton can be represented as $p(\langle arg1\rangle, \langle arg2\rangle)$ where *arg1* and *arg2* can be any instantiation pattern. Thus the information derived from the skeletons *p(1,2)* and *p(A,B)* would be the same.

The user should be aware that skeletons which are dynamically created (e.g. by functor/3) are located in usermod. In such a case, the tabling predicates below may not behave in the desired manner if the tabled predicates themselves have not been imported into usermod.

get_calls_for_table(+Skeleton,-Call,-Empty)

Succeeds whenever `Skeleton` corresponds to a predicate that has been declared tabled. For these predicates `get_calls_for_table/3` upon backtracking returns the calls of all current table entries. Also returned with each call is an atom which is a member of the set

{ `empty, notempty` }.

indicating whether the table is empty or non-empty.

This predicate does not provide any information about whether a table is complete. Use `table_state` to inquire about a table's state.

Example:

For the program

```
:- table(p/2).
p(1,2).
p(1,3).
p(1,X).
p(2,3).
```

and table

Call	*Returns*
p(1,X)	p(1,2) p(1,3) p(1,X)
p(X,3)	p(1,3) p(2,3)

Calls to **get_calls_for_table/3** would act as follows

```
| ?- get_calls_for_table(p(X,Y), Call, Empty).
X = _646608
Y = _646436
Call = p(1,_646724)
Empty = notempty;

X = _646608
Y = _646436
Call = p(_646720,3)
Empty = notempty

| ?- get_calls_for_table(p(1,2), Call, Empty).
Call = p(1,_646676)
Empty = notempty

Call = p(_646672,3)
Empty = notempty
```

The second example backtracks through all entries in the table, since only skeletal information is used from the first argument.

Exception:

table_error First argument is not callable.

β **Release Note:** In Version 1.0, this predicate may be changed to allow partial instantiation of the second argument.

get_returns_for_call(+Call,-Return)

Succeeds whenever there is a table entry for a variant of `Call`. If this is the case, Return is instantiated with an entry for the table corresponding to call, and succesive entries are returned by backtracking. If there are no answers in the table, or no table entry for `Call`, the goal fails.

This predicate creates fresh variables for the return, rather than unifying them with variables in the first argument. An explicit unification of a call with its return can be done if so desired.

Example: Let us continue from the example in **get_calls_for_table/3**.

```
| ?- get_returns_for_call(p(1,X), Return).

X = _646412
Return = p(1,_646628);

X = _646412
Return = p(1,2);

X = _646412
Return = p(1,3);

no
| ?- get_returns_for_call(p(X,Y), Return).
no

| ?- get_returns_for_call(p(1,2), Return).
no
```

Exception:

table_error Argument 1 is not a callable predicate.

β **Release Note:** In version 1.0, this predicate may be changed to allow partial instantiation of the second argument.

abolish_all_tables

abolish_all_tables/0 abolishes all tables presently in the system. Predicates which have been declared tabled remain tabled, but information about calls and returns is deleted. In Version 1.4.0, **abolish_all_tablesl/0** reclaims used space. The predicate always succeeds.

Note that open tables are abolished automatically by the system on exceptions and when the interpreter level is resumed. In these cases, the user does not need to abolish tables to maintain correctness.

abolish_table_pred(?Skeleton)

`abolish_table_pred/1` abolishes tables for all calls to the predicate denoted by *Skeleton*. The predicate remains tabled, but information about its calls and their returns is removed from the system. In Version 1.4.0, `abolish_table_pred/1` does not reclaim used space. Note that open tables are abolished automatically by the system on exceptions and when the interpreter level is resumed. In these cases, the user does not need to abolish tables to maintain correctness.

Exception:

`table_error` Not a tabled predicate.

abolish_table_call(?Call)

`abolish_table_call/1` abolishes tables for a particular call to a predicate denoted by *Call*. The predicate remains tabled, and information about all other tables remains intact. In Version 1.4.0, `abolish_table_call/1` does not reclaim used space. Note that open tables are abolished automatically by the system on exceptions and when the interpreter level is resumed. In these cases, the user does not need to abolish tables to maintain correctness.

Example:

Continuing the example started in the description of predicate `get_calls_for_table/3` the call `abolish_table_call(p(1,X))` would produce the table

Call	*Returns*
p(X,3)	p(1,3)
	p(2,3)

Exception:

`table_error` Argument 1 is not a callable predicate.

β **Release Note:** This predicate will not be ready until Version 1.0.

table_state(?Call,-State)

If the first argument is a valid Term indicator, `table_state/1` unifies state with one of the set {`not_yet_called, complete, incomplete, undef`}. The meanings of these atoms are defined as

- `not_yet_called` iff the predicate corresponding to Call has been declared tabled, but there is no table entry for call.
- `complete` iff the table entry for `Call` contains all solutions.
- `incomplete` iff the table entry for `Call` may not contain all solutions.
- `undef` iff the predicate corresponding to `Call` has not been declared tabled.

Exceptions:

`type_error` Argument 1 is not a callable predicate.

`table_prop(?Pred_indicator,+Type,?Old_val,+New_val)`
This predicate is described together with the following one.

`table_prop(?Pred_indicator,+Type,-Value)`
This predicate is used either to provide information about a table property, or to reset one of properties to a particular value. These values can also be set by the table declarations in the source code. `table_prop` provides a method for doing so without having to recompile code.

In Version 1.4.0 the properties visible to the user are the call and return hash sizes. Each tabled predicate contains two types of hash tables. A *call hash table* is used to find whether the variant of a call to a tabled predicate has occurred before in the course of a computation. Each tabled call has a *return hash table* which is used during a computation to determine whether a given return needs to be added to the table. Appropriate indexing for call and return lookup, including proper hash table sizes, is crucial to good performance.

Hash table sizes are non-negative integers denoting the number of buckets in the hash table. When `Pred_indicator` denotes a tabled predicate, the predicate can take one of two modes depending on its arity. In `table_prop/4`, the `New_value` is provided by the user and `Old_value` is unified with the precious value of the table property. In `table_prop/3` the present value of the property is returned.

Presently, the hash table sizes can only be reset when there are no tabled calls to `pred_indicator`. This can be ensured by a call to `abolish_table_pred/1`

Future versions of XSB may include additional indexing strategies as configurable table properties.

Exceptions:

`table_error` Attempt to reset hash table size for predicate with tabled calls.

`table_error` Argument 1 is not a tabled predicate.

`table_copy(+From,-To)`
Table copy provides an explicit mechanism for the user to reduce the amount of copying into and out of a table. `table_copy/2` succeeds when `From` is ground, and returns a copy of that structure (or list) in the table space. No copying of that structure will be required when calls or returns containing it are tabled, and no copying will be necessary when returns using it are resolved with goals.

For an example of the use of table copy, see Section 8.5

Chapter 7

Debugging

7.1 High-Level Tracing

XSB supports a version of the Byrd four-port debugger for debugging Prolog code. In this release (Version 1.4.0), it does not help when debugging code involving tabled predicates. Hopefully, future versions will have debugging help for tabled evaluation.

To turn on tracing, use `trace/0`. To turn tracing off, use `notrace/0`. When tracing is on, the system will print a message each time a predicate is:

1. initially entered (Call),
2. successfully returned from (Exit),
3. failed back into (Redo), and
4. completely failed out of (Fail).

At each port, a message is printed and the tracer stops and prompts for input. (See the predicates `show/1` and `leash/1` described below to modify what is traced and when the user is prompted.)

In addition to single-step tracing, the user can set spy points to influence how the tracing/debugging works. A spy point is set using `spy/1`. Spy points can be used to cause the system to enter the tracer when a particular predicate is entered. Also the tracer allows "leaping" from spy point to spy point during the debugging process.

When the tracer prompts for input, the user may enter a return, or a single character followed by a return, with the following meanings:

`c`, `<CR>`: **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.

`e`: **Exit** Causes immediate exit from XSB back to the operating system.

Other standard predicates that are useful in debugging are:

`spy(Preds)`

where `Preds` is a spy specification or a list of such specifications, and must be instantiated. This predicate sets spy points (conditional or unconditional) on predicates. A spy specification can be of several forms. Most simply, it is a term of the form P/N, where P is a predicate name and N its arity. It may optionally may be prefixed by a module name, e.g. $ModName$:P/N. A spy specification may also indicate a conditional spy point. A conditional spy specification is a Prolog rule, the head indicating the predicate to spy, and the body indicating conditions under which to spy. For example, to spy the predicate p/2 when the first argument is not a variable, one would write: $spy(p(X,_{)} : -nonvar(X))$. (Notice that the parentheses around the rule are necessary). The body may be empty, i.e., the rule may just be a fact. The head of a rule may also be prefixed (using :) with a module name. One should not put both conditional and unconditional spy points on the same predicate.

`nospy(Preds)`
: where **`Preds`** is a spy specification, or a list of such specifications, and must be instantiated at the time of call. What constitutes a spy specification is described above under **`spy`**. **`nospy`** removes spy points on the specified predicates. If a specification is given in the form of a fact, all conditional spy points whose heads match that fact are removed.

`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.

`show(PortList)`
: Allows the user to specify at which ports should trace messages be printed. **`PortList`** must be a list of port names, i.e., a sublist of ['Call', 'Exit', 'Redo', 'Fail']. **`show/1`** is not a standard predicate but must be imported from the module **`debugger`** before it can be called.

`leash(PortList)`
: Allows the user to specify at which ports the tracer should stop and prompt the user for direction. **`PortList`** must be a list of port names, i.e., a sublist of ['Call', 'Exit', 'Redo', 'Fail']. Only ports that are **`show`**-n can be **`leash`**-ed. **`leash/1`** is not a standard predicate but must be imported from the module **`debugger`** before it can be called.

7.2 Low-Level Tracing

XSB also provides a facility for low-level tracing of execution. This can be activated by invoking the emulator with the `-T` option (see section 3.6), 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.

Chapter 8

Definite Clause Grammars

8.1 General Description

Definite clause grammars (DCGs) are an extension of context free grammars that have proven useful for describing natural and formal languages, and that may be conveniently expressed and executed in Prolog. A Definite Clause Grammar rule in Prolog is executable because it is just a notational variant of a Prolog term that has the following general form:

Head `-->` *Body.*

with the declarative interpretation that "a possible form for *Head* is *Body*". The procedural interpretation of a grammar rule is that it takes an input list of symbols or character codes, analyses some initial portion of that list, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output lists are not written explicitly in the DCG rule, but are added when the rule is translated (expanded) into an ordinary Prolog clause during parsing. By defining the hook predicate **term_expansion/2** (in module term_expansion), the user can specify any desired transformation to be done as clauses are read by the reader. Extra conditions, in the form of explicit Prolog literals or control constructs such as *if-then-elses* (`'->'/2`) or *cuts* (`'!'/0`), may be included in the *Body* of the DCG rule and they work exactly as one would expect.

An overview of the syntax of DCGs supported by XSB is as follows:

1. A non-terminal symbol may be any HiLog term other than a variable or a number. A variable which appears in the body of a rule is equivalent to the appearence of a call to the built-in predicate **phrase/3** as it is described below.

2. A terminal symbol may be any HiLog term. In order to distinguish terminals from nonterminals, a sequence of one or more terminal symbols $\alpha, \beta, \gamma, \delta, \ldots$ is written within a grammar rule as a Prolog list `[` $\alpha, \beta, \gamma, \delta, \ldots$ `]`, with the empty sequence written as the empty list `[]`. The list of terminals may contain variables but it has to be a proper list, or else an error message is sent to the standard error stream and the expansion of the grammar rule that contains this list will fail. If the terminal symbols are ASCII character codes, they can be written (as elsewhere) as strings.

3. Extra conditions, expressed in the form of Prolog predicate calls, can be included in the body (right-hand side) of a grammar rule by enclosing such conditions in curly brackets, `'{'` and `'}'`. For example, one can write:

   `positive_integer(N) --> [N], {integer(N), N > 0}.` [1]

4. The left hand side of a DCG rule must consist of a single non-terminal, possibly followed by a sequence of terminals (which must be written as a *unique* Prolog list). Thus in XSB, unlike SB-Prolog version 3.1, "push-back lists" are supported.

5. The right hand side of a DCG rule may contain alternatives (written using the usual Prolog's disjunction operator `';'` or using the usual BNF disjunction operator `'|'`.

6. The Prolog control primitives *if-then-else* (`'->'/2`), *nots* (`not/1`, `fail_if/1`, `'\+'/1` or `tnot/1`) and *cut* (`'!'/0`) may also be included in the right hand side of a DCG rule. These symbols need not be enclosed in curly brackets. [2] However, the use of *nots* on the right hand side of grammar rules is not generally recommended. (Negation of literal in a grammar rule is unsound, for instance, whenever the output list of that literal is unbound). All other Prolog's control primitives, such as `repeat/0`, must be enclosed explicitly within curly brackets if they are not meant to be interpreted as non-terminal grammar symbols.

8.2 Translation of Definite Clause Grammar rules

The procedural interpretation of a DCG rule is that it takes an input list of symbols or character codes, analyzes some initial portion of that list, and produces the remaining portion (possibly enlarged, if pushback lists are used) as output for further analysis. As an abbreviation, the arguments required for the input and output lists are not written explicitly in a grammar rule, but are added when the rule is translated into an ordinary Prolog clause. In

[1] A term like `{foo}` is just a syntactic-sugar for the term `'{}'(foo)`.

[2] Readers familiar with Quintus Prolog may notice the difference in the treatment of the various kinds of not. For example, in Quintus Prolog a `not/1` that is not enclosed within curly brackets is interpreted as a non-terminal grammar symbol.

this section we informally describe this translation, which resembles the DCG rules of other Prologs in most particulars.

Each grammar rule is translated into a Prolog clause as it is consulted or compiled. This DCG term expansion is as follows:

A DCG rule such as:

```
p(X) --> q(X).
```

will be translated (expanded) into the Prolog rule:

```
p(X, Li, Lo) :-
      q(X, Li, Lo).
```

If there is more than one non-terminal on the right-hand side, as in

```
p(X, Y) --> q(X), r(X, Y), s(Y).
```

the corresponding input and output arguments are identified, translating into:

```
p(X, Y, Li, Lo) :-
      q(X, Li, L1),
      r(X, Y, L1, L2),
      s(Y, L2, Lo).
```

Terminals are translated using the built-in predicate `'C'/3` (See section 8.3 for its description). For instance:

```
p(X) --> [go, to], q(X), [stop].
```

is translated into:

```
p(X, S0, S) :-
      'C'(S0, go, S1),
      'C'(S1, to, S2),
      q(X, S2, S3),
      'C'(S3, stop, S).
```

Extra conditions expressed as explicit procedure calls naturally translate into themselves. For example,

```
positive_number(X) -->
      [N], {integer(N), N > 0},
      fraction(F), {form_number(N, F, X)}.
```

translates to:

```
positive_number(X, Li, Lo) :-
      'C'(Li, N, L1),
      integer(N),
      N > 0,
      L1 = L2,
      fraction(F, L2, L3),
      form_number(N, F, N),
      L3 = Lo.
```

Similarly, a cut is translated literally.

Push-back lists (a proper list of terminals on the left-hand side of a DCG rule) translate into a sequence of `'C'/3` goals with the first and third arguments reversed. For example,

```
it_is(X), [is, not] --> [aint].
```

becomes

```
it_is(X, Li, Lo) :-
      'C'(Li, aint, L1),
      'C'(Lo, is, L2),
      'C'(L2, not, L1).
```

Disjunction has a fairly obvious translation. For example, the DCG clause:

```
expr(E) -->
        expr(X), "+", term(Y), {E is X+Y}
      | term(E).
```

translates to the Prolog rule:

```
expr(E, Li, Lo) :-
      ( expr(X, Li, L1),
        'C'(L1, 43, L2),          % 0'+ = 43
        term(Y, L2, L3)
        E is X+Y,
        L3 = Lo
      ; term(E, Li, Lo)
      ).
```

8.3 Definite Clause Grammar predicates

The library predicates of XSB that support DCGs are the following:

`dcg(+DCG_Rule, ?Prolog_Clause)`
: Succeeds iff the DCG rule `DCG_Rule` translates to the Prolog clause `Prolog_Clause`. At the time of call, `DCG_Rule` must be bound to a term whose principal functor is `'-->'/2` or else the predicate fails.

`phrase(+Phrase, ?List)`
: This predicate is true iff the list `List` can be parsed as a phrase (i.e. sequence of terminals) of type `Phrase`. `Phrase` can be any term which would be accepted as a nonterminal of the grammar (or in general, it can be any grammar rule body), and must be instantiated to a nonvariable term at the time of the call; otherwise an error message is sent to the standard error stream and the predicate fails. This predicate is the usual way to commence execution of grammar rules.

  If `List` is bound to a list of terminals by the time of the call, then the goal corresponds to parsing `List` as a phrase of type `Phrase`; otherwise if `List` is unbound, then the grammar is being used for generation.

`phrase(+Phrase, ?List, ?Rest)`
: This predicate is true iff the segment between the start of list `List` and the start of list `Rest` can be parsed as a phrase (i.e. sequence of terminals) of type `Phrase` . In other words, if the search for phrase `Phrase` is started at the beginning of list `List`, then `Rest` is what remains unparsed after `Phrase` has been found. Again, `Phrase` can be any term which would be accepted as a nonterminal of the grammar (or in general, any grammar rule body), and must be instantiated to a nonvariable term at the time of the call; otherwise an error message is sent to the standard error stream and the predicate fails.

  Predicate `phrase/3` is the analogue of `call/1` for grammar rule bodies, and provides a semantics for variables in the bodies of grammar rules. A variable `X` in a grammar rule body is treated as though `phrase(X)` appeared instead, `X` would expand into a call to `phrase(X, L, R)` for some lists `L` and `R`.

`expand_term(+Term1, ?Term2)`
: This predicate is used to transform terms that appear in a Prolog program before the program is compiled or consulted. The standard transformation performed by `expand_term/2` is that when `Term1` is a grammar rule, then `Term2` is the corresponding Prolog clause; otherwise `Term2` is simply `Term1` unchanged. If `Term1` is not of the proper form, or `Term2` does not unify with its clausal form, predicate `expand_term/2` simply fails.

Users may override the standard transformations of **expand_term/2** by defining their own compile-time transformations. This can be done by defining clauses for the predicate **term_expansion/2** but this definition must be done in a module named **term_expansion**. When a term **Term1** is read in when a file is being compiled or consulted, **expand_term/2** calls **term_expansion/2** first; if the expansion succeeds, the transformed term so obtained is used and the standard grammar rule expansion is not tried; otherwise, if **Term1** is a grammar rule, then it is expanded using **dcg/2**; otherwise, **Term1** is used as is.

When the user defines predicate **term_expansion/2** he must override the default action of **term_expansion/2** provided by XSB. For that reason, the user's **term_expansion** module must be in a directory searched by the dynamic loader before the system's **syslib** directory (see Section 3.5 for a description of how the dynamic loader works and how to change the way the loader searches for modules).

'C'(?L1, ?Terminal, ?L2)

This predicate generally is of no concern to the user. Rather it is used in the transformation of terminal symbols in grammar rules and expresses the fact that **L1** is connected to **L2** by the terminal **Terminal**. This predicate is needed to avoid problems due to source-level transformations in the presence of control primitives such as *cuts* (`'!'/0`), or *if-then-elses* (`'->'/2`) and is defined by the single clause:

```
'C'([Token|Tokens], Token, Tokens).
```

The name 'C' was chosen for this predicate so that another useful name might not be pre-empted.

8.4 Two differences with other Prologs

The DCG expansion provided by XSB is in certain cases different from the ones provided by some other Prolog systems (e.g. Quintus Prolog, SICStus Prolog and C-Prolog). The most important of these differences are:

1. XSB expands a DCG clause in such a way that when a `'!'/0` is the last goal of the DCG clause, the expanded DCG clause is always *steadfast*.

   That is, the DCG clause:

   ```
   a --> b, !  ; c.
   ```

   gets expanded to the clause:

   ```
   a(A, B) :- b(A, C), !, C = B ; c(A, B).
   ```

and *not* the clause:

```
a(A, B) :- b(A, B), !  ; c(A, B).
```

as in Quintus, SICStus and C Prolog.

The latter expansion is not just optimized, but it can have a *different (unintended) meaning* if `a/2` is called with its second argument bound.

However, to obtain the alternate expansion, the user she may simply interchange file `dcg.P` with file `dcg_qsc.P` found in the `syslib` system directory.

Problems of DCG expansion in the presence of *cuts* have been known for a long time and almost all Prolog implementations expand a DCG clause with a single `'!'/0` in its body in such a way that its expansion is steadfast and has the intended meaning when called with its second argument bound. For that reason almost all Prologs translate the DCG clause:

```
a --> !  ; c.
```

to the clause:

```
a(A, B) :- !, B = A ; c(A, B).
```

But in our opinion this is just a special case of a `'!'/0` being the last goal in the body of a DCG clause.

2. Most of the control predicates of XSB need not be enclosed in curly brackets. A difference with say Quintus is that predicates `not/1` and `fail_if/1` do not get expanded when encountered in a DCG clause.

   That is, the DCG clause:

   ```
   a --> (true -> X = f(a) ; not(p)).
   ```

   gets expanded to the clause:

   ```
   a(A,B) :- (true(A,C) -> =(X,f(a),C,B) ; not p(A,B))
   ```

   and *not* the clause:

   ```
   a(A,B) :- (true(A,C) -> =(X,f(a),C,B) ; not(p,A,B))
   ```

   that Quintus Prolog expands to.

   However, note that all non-control but standard predicates (for example `true/0` and `'='/2`) get expanded if they are not enclosed in curly brackets.

8.5 Interaction of Definite Clause Grammars and Tabling

Tabling can be used in conjunction with Definite Clause Grammars to get the effect of a more complete parsing strategy. When Prolog is used to evaluate DCG's, the resulting parsing algorithm is "recursive descent". Recursive descent parsing, while efficiently implementable, is known to suffer from several deficiencies: 1) its time can be exponential in the size of the input, and 2) it may not terminate for certain context-free grammars (in particular, those that are left-recursive.) By appropriate use of tabling, both of these limitations can be overcome. With appropriate tabling, the resulting parsing algorithm is a variant of Earley's algorithm and the chart parsing algorithms.

In the simplest cases, one needs only to add the declaration *:- table_all* to the source file containing a DCG specification. This should generate any necessary table declarations so that infinite loops are avoided (for context-free grammars.) That is, with a *:- table_all* declaration, left-recursive grammars can be correctly processed. (Of course, individual *table* directives may also be used, but note that the arity must be specified as two more than that shown in the DCG source, to account for the extra arguments added by the preprocessor.)

However, due to our current implementation of structures in tabling, there are new inefficiencies that can arise. In particular, when using the standard list representation of the input string in a DCG, there may be a large amount of copying and a great deal of space used. What happens is that the input string (i.e. list) may be copied into and out of the table many times. This can be avoided by using the builtin predicate `table_copy/2`. By using `table_copy/2` to copy the initial ground input list into the table, the user can avoid this overhead.

For example, to read an input string (as a list) from the user and then to see if it parses to a *sentence*, using a DCG compiled with tabling, the following query could be used:

```
:- read(Str), table_copy(Str,TabStr), phrase(sentence(Parse),TabStr).
```

See the examples in the directory distributed with XSB to see how the facility can be used.

Chapter 9

Writing Efficient XSB Programs

Logic programming systems at their best, allow a user to create an executable specification of a problem. These specifications often run surprisingly well, but it is still important to understand the basics of writing efficient programs. Prolog, after all, is the language that introduced permutation sorts to the world. When tabling is not used, techniques for writing efficient Prolog programs work for XSB programs as well. There are many excellent books which teach efficient Prolog programming, among them [Brat 86, CM 84, MW 88, SS 86]. This chapter reiterates some points of efficient Prolog programming that are of particular use for XSB and then presents a few basic points of writing efficient tabled programs.

9.1 Writing Efficient Non-Tabled Predicates

When predicates aren't tabled, they can be optimized in the same way as Prolog predicates. This section presents a few techniques for writing efficient programs.

Use Proper Indexing

One of most important ways to optimize Prolog programs is to properly index clause selection. This technique sounds basic, and it is, but it is surprising how many programs can be speeded up by indexing.

Prolog clauses are indexed on the functor of their first argument by default, using a hash-based lookup. Thus, a clause with head `p(f(1),G)` would be indexed on `f`. (Of course, any lookup for the predicate `p/2` would be either direct or indexed). For compiled code, indexing not only controls clause selection, but may prevent laying down of a choice point, one of the more expensive operations in Prolog. For instance in `append/3`

```
append([],L,L).
append([H|T],L,[H|T1]):-
    append(T,L,T1).
```

indexing prevents a choice point from ever being constructed, since no bound value of the first argument can unify with more than one clause.

Indexing in Prolog (and XSB) is crude, but often effective. There are two major cases where it is not. The first occurs when calling patterns are such that the predicate is called with two different instantiations. This occurs, for instance, in the same generation example.

```
same_generation(X,X).
same_generation(X,Y):-
        parent(X,Z),
        same_generation(Z,Z1),
        parent(Y,Z1).
```

Here **parent/2** is called with patterns *bound,free* and *free,bound*. In Version 1.4.0 there is no way of providing efficient indexing for this problem, other than by making a separate copy of `parent/2`. It is possible that future versions of the compiler may allow better index creation for problems like these.

Another case where first argument indexing is ineffective occurs when the first argument alone does not provide sufficient selectivity. This case arises sometimes in deductive database applications where a call may be made to a relation of the form

```
part(Part,Supplier,Location,Cost).
```

Finding the cost of particular part, supplied by a particular supplier at a particular location may involve a certain amount of backtracking.

Version 1.4.0 of XSB does allow one small exception to first argument indexing. If indexing is desired on, say, the third argument of `foo/5` rather than on the first the declaration

```
:-index foo/5-3.
```

can be put into the source file. See Section 3.9.4 for details.

Make programs deterministic whenever possible, as early as possible.

One of the advantages of logic programming is that it computes relations in an efficient, elegant manner. Still, it is often the case that given an input pattern, a predicate may be *functional* – it may return a single solution. In a case like this, the overhead necessary for computing relations – choice points and trailing – may be avoided.

Certainly the best way to avoid the overhead of computing relations is to use indexing whenever possible. When the first argument of **append/3** is bound, for example, is is computed as if it were a function. As mentioned in the last section, though, **append/3** has the property that no instantiated value of the first argument can unify with more than one clause. This is not always the case, particularly when so-called "defaulty reasoning" is used. The most familiar example of defaulty reasoning may occur in meta-interpreters.

```
solve(true).
solve(',')(A,B)):-
    solve(A),solve(B).
solve(A):-
    clause(A,B),solve(B).
```

The third clause of `solve/1` is the default case: whenever either of the first two clauses are chosen, they will create a choice point. In a case like this, it is sometimes necessary to use a non-logical construct like a cut or a conditional to destroy the choice point. Even so, the time to create the choice point can never be regained, and the cut itself takes a couple of XWAM instructions.

When cuts need to be used, they should be made as early as possible in a program. The reasons for this are technical. Prolog provides for backtracking by keeping a stack called a *trail*. When a variable was created before the last choice point and is bound, its address is trailed so that its previous value can be restored upon backtracking. Clearly the more choice points created, the more trailing, and as a corollary, the sooner a choice point is destroyed, the less trailing occurs. As another technical comment, due to the requirements of tabling, trailing in XSB is slightly slower than in Prolog, and takes up more space. Cutting early will improve performance of XSB programs even more than Prolog programs.

9.2 Writing Efficient Tabled Predicates

β **Release Note:** This section remains to be written.

Chapter 10

Library Utilities

In this chapter we introduce some useful predicates that are supplied with the system. These predicates are available only when imported them from (or explicitly consult) the corresponding modules.

10.1 List Processing

The XSB library contains various list utilities, some of which are listed below. These predicates should be explicitly imported from the module specified after the skeletal specification of each predicate. There are a lot more useful list processing predicates in various modules of the XSB system, and the interested user can find them by looking at the sources.

`append(?List1, ?List2, ?List3)` basics
: Succeeds if list `List3` is the concatenation of lists `List1` and `List2`.

`member(?Element, ?List)` basics
: Checks whether `Element` unifies with any element of list `List`, succeeding more than once if there are multiple such elements.

`memberchk(?Element, ?List)` basics
: Similar to `member`/2, except that `memberchk`/2 is deterministic, i.e. does not succeed more than once for any call.

`ith(?Index, ?List, ?Element)` basics
: Succeeds if the `Index`th element of the list `List` unifies with `Element`. Fails if `Index` is not a positive integer or greater than the length of `List`. Either `Index` and `List`, or

`List` and `Element`, should be instantiated (but not necessarily ground) at the time of the call.

`length(?List, ?Length)` **basics**

Succeeds if the length of the list `List` is `Length`. This predicate is deterministic if `List` is instantiated to a list of definite length, but is nondeterministic if `List` is a variable or has a variable tail. If `List` is uninstantiated, it is unified with a list of length `Length` that contains variables.

`select(?Element, ?L1, ?L2)` **basics**

`List2` derives from `List1` by selecting (removing) an `Element` non-deterministically.

`reverse(+List, ?ReversedList)` **basics**

Succeeds if `ReversedList` is the reverse of list `List`. If `List` is not a proper list, `reverse/2` can succeed arbitrarily many times.

`merge(+List1, +List2, ?List3)` **listutil**

Succeeds if `List3` is the list resulting from "merging" lists `List1` and `List2`, i.e. the elements of `List1` together with any element of `List2` not occurring in `List1`. If `List1` or `List2` contain duplicates, `List3` may also contain duplicates.

`absmerge(+List1, +List2, ?List3)` **listutil**

Predicate `absmerge/3` is similar to `merge/3`, except that it uses predicate `absmember/2` described below rather than `member/2`.

`absmember(+Element, +List)` **listutil**

Similar to `member`/2, except that it checks for identity (through the use of predicate `'=='/2`) rather than unifiability (through `'='/2`) of `Element` with elements of `List`.

`member2(?Element, ?List)` **listutil**

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

`delete_ith(+Index, +List, ?Element, ?RestList)` **listutil**

Succeeds if the `Index`$^{\mathtt{th}}$ element of the list `List` unifies with `Element`, and `RestList` is `List` with `Element` removed. Fails if `Index` is not a positive integer or greater than the length of `List`.

`closetail(?List)` **listutil**

Predicate `closetail/1` closes the tail of an open-ended list.

`remove_dupls(+List1, ?List2)`

There is no need to provide such a predicate. If you need one use the `sort/2` standard predicate (described in section 6.7).

10.2 Asserting Dynamic Code

The module `basics` in directory `lib` provides a very handy library predicate that can assert the contents of a file into XSB's database. The use of this predicate may be necessary when the code needs to be `dynamic` (so that it is retractable), or when it contains atoms whose length is more than 255 that cannot be handled by the XSB compiler.

The format of this predicate is as follows:

`load_dyn(+FileName)` basics

Asserts the contents of file `FileName` into the database. All existing clauses of the predicates in the file that already appear in the database, are retracted unless there is a `multifile/1` directive for them.

As usual, clauses of predicats are not retracted if they are compiled instead of dynamically asserted.

10.3 Ground, Numbervars, Subsumption, Variant

`ground(+X)` basics

Succeeds if `X` is currently instantiated to a term that is completely bound (has no uninstantiated variables in it); otherwise it fails. Predicate `ground/1` has no associated error conditions.

`numbervars(+Term, +FirstN, ?LastN)` numbervars

This predicate provides a mechanism for grounding a (HiLog) term so that it may be analyzed. Each variable in the (HiLog) term `Term` is instantiated to a term of the form `'$VAR'(N)`, where `N` is an integer starting from `FirstN`. `FirstN` is used as the value of `N` for the first variable in `Term` (starting from the left). The second distinct variable in `Term` is given a value of `N` satisfying `"N is FirstN + 1"` and so on. The last variable in `Term` has the value `LastN-1`.

`numbervars(+Term)` numbervars

This predicate is defined as:

```
numbervars(Term, 0, _).
```

It is included solely for convenience.

`subsumes(?Term1, +Term2)` **subsumes**

Term subsumption is a sort of one-way unification. Term `Term1` and `Term2` unify if they have a common instance, and unification in Prolog instantiates both terms to that (most general) common instance. `Term1` subsumes `Term2` if `Term2` is already an instance of `Term1`. For our purposes, `Term2` is an instance of `Term1` if there is a substitution that leaves `Term2` unchanged and makes `Term1` identical to `Term2`. Predicate `subsumes/2` does not work as described if `Term1` and `Term2` share common variables.

`subsumes_chk(+Term1, +Term2)` **subsumes**

The `subsumes_chk/2` predicate is true when `Term1` subsumes `Term2`; that is, when `Term2` is already an instance of `Term1`. This predicate simply checks for subsumption and does not bind any variables either in `Term1` or in `Term2`.

Examples:

```
| ?- subsumes_chk(a(X,f,Y,X),a(U,X,b,S)).

no
| ?- subsumes_chk(a(X,Y,X),a(b,b,b)).

X = _595884
Y = _595624
```

`variant(?Term1, ?Term2)` **subsumes**

This predicate is true when `Term1` and `Term2` are alphabetic variants. That is, you could imagine that `variant/2` as being defined like:

```
variant(Term1, Term2) :-
     subsumes_chk(Term1, Term2),
     subsumes_chk(Term2, Term1).
```

but the actual implementation of `variant/2` is considerably more efficient.

10.4 Formatted I/O

`fmt_read(+Fmt,+Types,-Term,-Ret)` **machine**

This predicate provides a low-level routine for reading data from the current input file (must have been already opened by using `see/1`) according to a C format, as used in the C function `scanf`. To use it, it must be imported from the module `machine`. `Fmt` must be an atom representing the format that will be passed to the C call to `scanf`.

See the C documentation for `scanf` for the meaning of this string. Note that this argument is an atom, not a string, and the C escape characters are NOT recognized. So, for example, to include a newline, one must put a newline directly in the atom, and not use `\n`. `Types` is a term whose subfields describe the types of the fields to be read. For example, if three fields are to be read, first an integer, then a string, and then a floating point number, the term `type(2,1,3)` could be used. The functor symbol of `Type` is ignored. The arguments indicate the types of the subfields: 1 indicates a string value; 2 indicates an integer value; and 3 indicates a floating point value. `Term` is a term whose arguments will be unified with the field values read in. (Again, the functor symbol of `Term` is ignored.) It is an error (and in some cases the results are unpredictable) if the number of arguments and their types are not consistent among `Fmt`, `Types`, and `Term`. `Ret` must be a variable and it will be assigned a return value by the predicate: a negative integer if end-of-file is encountered; otherwise the number of fields read (as returned by `scanf`.)

`fmt_write(+Fmt,+Term)` machine

This predicate provides a low-level routine for writing data to the current output file (must have been already opened by using `tell/1` according to a C format, as used in the C function `printf`. To use it, it must be imported from the module `machine`. `Fmt` must be an atom representing the format that will be passed to the C call to `printf`. See the C documentation for `printf` for the meaning of this string. Note that this is an atom, not a string, and the C escape characters are NOT recognized. So, for example, to include a newline, one must put a newline directly in the atom, and not use `\n`. `Term` is a term whose arguments will be output. They must agree in number and in type with the field indicators in the format `Fmt`. (The functor symbol of `Term` is ignored.)

10.5 Arrays

The module `array1` in directory `lib` provides a very simple backtrackable array implementation. The predicates through which the array objects are manipulated are:

`array_new(-Array, +Size)` array1

Creates a one dimensional empty array of size `Size`. All the elements of this array are variables.

`array_elt(+Array, +Index, ?Element)` array1

True iff `Element` is the `Index`-th element of array `Array`.

`array_update(+Array, +Index, +Elem, -NewArray)` array1

Updates the array `Array` such that the `Index`-th element of the new array is `Elem` and returns the new array in `NewArray`. The implementation is quite efficient in that it avoids the copying of the entire array.

A small example that shows the use of these predicates is the following:

```
| ?- import [array_new/2, array_elt/3, array_update/4] from array1.

yes
| ?- array_new(A, 4), array_update(A,1,1,B), array_update(B,2,2,C),
     ( array_update(C,3,3,D), array_elt(D,3,E)
     ; array_update(C,3,6,D), array_elt(D,3,E)
     ; array_update(C,3,7,D), array_elt(D,3,E)
     ).

A = array(1,2,3,_874600)
B = array(1,2,3,_874600)
C = array(1,2,3,_874600)
D = array(1,2,3,_874600)
E = 3;

A = array(1,2,6,_874600)
B = array(1,2,6,_874600)
C = array(1,2,6,_874600)
D = array(1,2,6,_874600)
E = 6;

A = array(1,2,7,_874600)
B = array(1,2,7,_874600)
C = array(1,2,7,_874600)
D = array(1,2,7,_874600)
E = 7;

no
```

10.6 The Profiling Library

Here are the predicates in the profiling library `prof_lib`. These predicates are available by loading the library through `[prof_lib]`.

`measure(+Call)` prof_lib

The same as `once/1` (see section 6.8), but it also prints out statistical information about the call, that looks like the information printed out by predicate `statistics/0` (see section 6.11).

Chapter 11

Foreign Language Interface

When XSB is used to build real-world systems, a foreign-language interface may be necessary to

- combine XSB with existing programs and libraries, thereby forming composite systems;
- interface XSB with the operating system, graphical user interfaces or other system level programs;
- speed up certain critical operations.

XSB provides a simple yet powerful foreign language interface. We have tried to keep the interface part of a foreign predicate similar to that of a normal Prolog predicate. Currently we only provide a C interface for SUN3/SPARC machines (running SunOS 4.x.x or Solaris 2.x), and 80386/80486-based machines running Linux, to which the discussion below is restricted. In the future releases we intend to provide the same foreign language interface for the other platforms where XSB can be used.

Foreign predicates must always appear in modules, which can contain only foreign predicates. The main difference between a normal module and a foreign module is the very natural one: the source file of the module implementation, which is in C, must appear in a `*.c` file rather than a `*.P` file. This `*.c` file cannot contain a `main()` function. Furthermore, a `*.P` file with the same name *must not* be present or else the `*.c` file is ignored and the module is compiled as a regular Prolog module. The interface part of a foreign module, which has the same syntax as that of a normal module, is written in Prolog and hence must appear in a `*.H` file. This `*.H` file contains `export` declarations for each and every one of the foreign predicates that are to be used by other modules.

Directives such as **index**, **hilog**, **table**, **table_all** or even **import** make no sense in the case of a foreign module and thus are ignored by the compiler. However, another directive, namely **ldoption**, is recognized in a foreign module and is used to instruct the dynamic loading and linking of the module. The syntax of the `ldoption` directive is simply:

```
:- ldoption(Option).
```

where `Option` should be an atom (currently of length less or equal to 255 characters).

ld combines object programs to create an executable file or another object program suitable for further ld processing

A foreign predicate assumes that its arguments are always of the correct types and modes. It is the caller's responsibility (in the Prolog level) to guarantee this.

At the C level, the procedure that implements the foreign predicate must have the same name as the predicate (that is declared in the `*.H` file), and it must be *parameterless*. The Prolog level arguments are converted to C data structures through several predefined functions rather than through direct parameter passing.

Our foreign language interface currently supports exchange of Prolog's atomic data types (atoms, integers, and floating-point numbers). The data has to be converted between Prolog's internal representation and the internal representation of the foreign language. For that reason, the following C functions are used to convert Prolog and C datatypes to each other.

`ptoc_int(N)`
: Argument `N` is assumed to hold a Prolog integer, and this function returns its integer value in C format.

`ptoc_float(N)`
: Argument `N` is assumed to hold a Prolog floating point number, and this function returns its floating point value in C format. (Precision is less than single word floating point).

`ptoc_string(N)`
: Argument `N` is assumed to hold a Prolog atom, and this function returns the C string (of type **char ***) that corresponds to this Prolog atom.

`ctop_int(N, V)`
: Argument `N` is assumed to hold a Prolog free variable, and this function binds that variable to an integer of value `V`.

`ctop_float(N, V)`
: Argument `N` is assumed to hold a Prolog free variable, and this function binds that variable to a floating point number of value `V`.

`ctop_string(N, V)`
Argument `N` is assumed to hold a Prolog free variable, and this function binds that variable to a Prolog atom of value `V`.

In the current implementation, the Prolog procedures that are attached to foreign predicates are deterministic, in the sense that they succeed at most once for a given call and are not re-entered on backtracking. Note that this requirement imposes no serious limitation, since it is always possible to divide a foreign predicate into the part to be done on the first call and the part to be redone on backtracking. Backtracking can then take place at the Prolog level where it is more naturally expressed.

A foreign module can be `compile`d or `consult`ed just like a normal Prolog module. Currently, predicates `consult/[1,2]` recompile both the `*.c` and the `*.H` files of a foreign module when at least one of them has been changed from the time the corresponding object files have been created (see section 3.8).

There are some sample programs in the `examples`, directory that exhibit most of the features of the foreign language interface, and will probably make clear many questions that users may have. We promise a more detailed description of the foreign language interface, in some future version.

Chapter 12

Restrictions and Bugs

In this chapter we indicate some key features (sic) of XSB that may affect the users at some point in their interaction with the system. Most of the current restrictions and known bugs are relatively rare and probably will not interfere with most of the users' uses of the system.

If at some point in your interaction with the system you suspect that you have run across a bug, please report it in the address found in chapter 13, using the suggested format. We will do our best to fix it or to assist you to bypass it.

12.1 Current Restrictions

- The maximum arity for predicate and function symbols is 255.
- Though the maximum length of the atoms that can be read by the reader is currently 1000, when the atoms are compiled, their maximum length that can be stored in the byte code file is by definition 255. So, users should avoid compiling files containing atoms whose length is more than 255. Instead, the files containing these atoms could be loaded into the system using the library predicate `load_dyn/1` (it can be imported from module `basics`) that asserts the contents of a file in XSB's database (see section 10.2). To increase this limit beyond 1000 requires a redefinition of MaxStrLen in emu/token.c, and recompilation.
- In the current version, you should never try to rename a byte code file generated for a module, though you can move it around in your file system. Since the module name is stored in the file, renaming it causes the system to load it into wrong places. However, byte code files for non-modules can be renamed at will.
- In the single word representation of cells (which is currently the only representation

supported) integers and floating-point numbers can be up to 28 bits long for non MIPS-based machines, and up to 27 bits long for MIPS-based machines (for example Silicon Graphics machines). This restriction can cause unexpected results when numbers are computed. The problem is especially severe for floating-point numbers.

- Indexing on floating point numbers does not work. Therefore, it is advisable that you avoid using floating point numbers in the first argument of a procedure, since the compiler generates index information for the first argument by default, or that you explicitly index the predicate in some other argument.

- The user is not allowed to define clauses for the empty list (`[]`). This is because in XSB the empty list has a special internal representation. As a result, calls to the empty list will result in a segmentation fault, and therefore are not a good idea.

- The XSB compiler cannot distinguish the occurences of a 0-ary predicate and a name of a module (of an import declaration) as two different entities. For that reason it fails to characterise the same symbol table entry as both a predicate and a module at the same time. As a result of this fact, a compiler error is issued and the file is not compiled. For that reason we suggest the use of mutually exclusive names for modules and 0-ary predicates, though we will try to amend this restriction in future versions of XSB.

12.2 Known Bugs

- As mentioned in sectiom 3.7 the stacks (except the program space) are not dynamically expanded. When stack overflows occur, you should try to enlarge the appropriate stack space through the command line option (see section 3.6).

- Compilation of *very* large files *may* cause a core dump due to unfinished memory management.

- Variables that appear in *compiled* arithmetic comparison predicates should only be bound to numbers and not evaluable arithmetic expressions. That is, the variables are not evaluated to obtain an arithmetic value, but the XSB compiler assumes that they are evaluated. For example, executing compiled code for the following program will cause an "`Arithmetic exception`" error:

```
p(X) :- X =:= 1.

?- p(cos(0)).
```

This behaviour is only exhibited in *compiled* code. For variables that is not known whether they will always be bound to numbers, it is advisable that these variables are evaluated by using `is/2`. In our example, predicate `p/1` could be written as:

```
p(X) :- X1 is X, X1 =:= 1.
```

- The reader refuses to read an infix operator immediately followed (without any blank) by a left parenthesis. In such a case you get a syntax error. To avoid the syntax error just leave a blank between the infix operator and the left parenthesis. For example, instead of writing:

  ```
  | ?- X=(a,b).
  ```

  write:

  ```
  | ?- X= (a,b).
  ```

- When the code of a predicate is reloaded many times, if the old code is still in use at the time of loading, unexpected errors may occur, due to the fact that the space of the old code is reclaimed and may be used for other purposes.

Chapter 13

Bug Reports and Other Comments

Please address all correspondence to:

XSB
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
U.S.A.
(`xsb-contact@cs.sunysb.edu`)

We strongly encourage you to register as an XSB user in the address given above. By doing so, you will be informed about future releases or major bug-fixes.

When you *strongly suspect* a bug (i.e., after reading this manual, and carefully examine the symptoms), please try to find the *smallest* program that illustrates the bug and mail it to this address together with a script that shows the problem.

Bibliography

[BMSU 86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–15, March 1986.

[Brat 86] I. Bratko Prolog Programming for Artificial Intelligence. Addison-Wesley. 1986.

[CKW 93] W. Chen, M. Kifer, and D.S. Warren. HiLog: A Foundation for Higher-Order Logic Programming. In the *Journal of Logic Programming*, 15:3, pp. 187–230, February 1993

[CW 93] W. Chen and D.S. Warren. Efficient ... (submitted for publication)

[CSW 93] W. Chen, T. Swift and D.S. Warren. Implementing ... (submitted for publication)

[CM 84] W. Clocksin C. Mellish, Programming in Prolog. Springer-Verlag, 1984.

[Debr 88] Saumya Debray, SB-Prolog System, Version 3.0, A User Manual, *Department of Computer Science, University of Arizona*, September 1988.

[DW 86] S.W. Dietrich and D.S. Warren, Extension tables: Memo relations in logic programming. *Technical Report 86/18, Department of Computer Science, SUNY at Stony Brook*, 1986.

[KT 88] D.B. Kemp and R.W. Topor, Completeness of a top-down query evaluation procedure for stratified databases. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 178–194, 1988.

[MW 88] D. Maier and D.S. Warren, Computing with Logic: Logic Programming with Prolog Benjamin Cummings Publishing Co. 1988.

[Ross 91] K.A. Ross. The Semantics of Deductive Databases. *PhD thesis, Department of Computer Science, Stanford University*, August 1991.

[SI 88] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended cwa. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 195–211, 1988.

[SS 86] L. Sterling and E. Shapiro, The Art of Prolog. MIT Press. 1986.

[STW 94] K. Sagonas, T. Swift and D.S. Warren XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Minneapolis, Minnesota, May 1994.

[SW 94] K. Sagonas and D.S. Warren A Compilation Scheme for HiLog. Submitted for publication. Also available by anonymous ftp from `cs.sunysb.edu` in the directory `pub/TechReports/warren`.

[TS 86] H. Tamaki and T. Sato. Old resolution with tabulation. In *Intl. Conference on Logic Programming*, pages 84–98, 1986.

[Ullm 89] J. Ullman. Principles of Database and Knowledge-base Systems. Computer Science Press, 1987.

[Viel 87] L. Vieille. A database-complete proof procedure based upon sld-resolution. In *Intl. Conference on Logic Programming*, 1987.

[Walk ??] A. Walker. Backchain iteration: towards a practical inference method that is simple enough to be proved terminating, sound and complete. *Journal of Automated Reasoning.*

[Warr 92] D.S. Warren. Memoing for logic programs with application to abstract interpretation and partial deduction. *CACM*, 1992.

[Xu 91] Jiyang Xu, PSB-Prolog System, Version 1.40, User Manual, *ECRC*, 1991

[Xu 89] Jiyang Xu, Metalogical Built-in Predicates: Semantics, Implementation and Applications, draft, 1989.

Appendix A

Standard Predicates and Functions

A.1 List of Standard Predicates

```
abolish(Name/Arity)
abolish(Name, Arity)
abolish_all_tables
abolish_table_call(Term)
abolish_table_pred(Pred)
abort
analyze_table(Pred)
arg(Index, Term, Arg)
arg0(Index, Term, Arg)
assert(Clause)
asserta(Clause)
assertz(Clause)
atom(Term)
atomic(Term)
atom_chars(Atom, CharList)
bagof(Elem, Goal, Bag)
break
'C'(List1, Token, List2)
call(Term)
callable(Term)
cd(Dir)
clause(Head, Body)
close(FileName)
compare(Res, Term1, Term2)
```

```
compound(Term)
compile(Module)
compile(Module, Options)
consult(Module)
consult(Module, Options)
copy_term(Term, Copy)
cputime(Time)
current_atom(Atom)
current_functor(Functor)
current_functor(Functor, Term)
current_input(File)
current_module(Module)
current_module(Module, File)
current_op(Precedence, Type, Name)
current_output(File)
current_predicate(Predicate)
current_predicate(Predicate, Term)
debug
debugging
edit(File)
erase(Reference)
expand_term(Term, Expanded_Term)
fail
fail_if(Goal)
file_exists(File)
findall(Elem, Goal, List)
float(Term)
functor(Term, Functor, Arity)
get(Char)
get0(Char)
get_calls_for_table(Pred, Call, Empty)
get_returns_for_call(Call, Return)
halt
hilog(Symbol)
hilog_arg(Index, Term, Arg)
hilog_functor(Term, Functor, Arity)
hilog_op(Precedence, Type, Name)
hilog_symbol(Symbol)
import PredList from Module
index(Predicate, ArgNo, HashSize)
instance(Ref, Instance)
integer(Term)
```

`is(Result, Expression)`	
`is_list(Term)`	
`keysort(Input, Output)`	
`listing`	
`listing(Predicate)`	
`ls`	
`mc(Module, Options)`	This predicate is used in the `Makefiles` when self-compiling XSB.
`module_property(Module, Property)`	
`name(Term, CharList)`	
`nl`	
`nodebug`	
`nonvar(Term)`	
`nospy(Predicate_List)`	
`not(Goal)`	
`notrace`	
`number(Term)`	
`once(Goal)`	
`op(Precedence, Format, Operator)`	
`otherwise`	
`phrase(Phrase, List)`	
`phrase(Phrase, List, Remains)`	
`predicate_property(Predicate, Property)`	
`print(Term)`	Currently the same as `write/1`.
`prompt(New, Old)`	
`proper_hilog(Term)`	
`put(Char)`	
`read(Term)`	
`real(Term)`	
`reconsult(Module)`	
`reconsult(Module, Options)`	
`record(Key, Term, Ref)`	
`recorda(Key, Term, Ref)`	
`recorded(Key, Term, Ref)`	
`recordz(Key, Term, Ref)`	
`rename(OldName, NewName)`	
`repeat`	
`retract(Term)`	
`retractall(Predicate)`	
`see(File)`	
`seeing(File)`	
`seen`	

`setof(Elem, Goal, Set)`	
`skip(Char)`	
`sort(Input, Output)`	
`spy(Predicate_List)`	
`statistics`	
`statistics(Number)`	
`structure(Term)`	
`tab(Count)`	
`table_copy(Old, New)`	
`table_prop(Pred, Type, Value)`	
`table_prop(Pred, Type, OldValue, NewValue)`	
`table_state(Call, State)`	
`tell(File)`	
`telling(File)`	
`tfindall(Elem, Goal, List)`	
`tnot(Goal)`	
`told`	
`true`	
`trace`	
`ttywrite(Term)`	Same as `write` but always writes to the standard output.
`ttywritenl(Terms)`	`Terms` can be a comma list of terms. The predicate calls `ttywrite/1` to print all the terms in the list and then prints a new line symbol at the end.
`unix(Command)`	
`unix(Command, Result)`	
`var(Term)`	
`write(Term)`	
`writeln(Term)`	
`write_prolog(Term)`	
`writeq(Term)`	
`write_canonical(Term)`	
`xsb_flag(Flag, Value)`	
`\+ Query`	
`!`	
`X =:= Y`	
`X =\= Y`	
`X < Y`	
`X > Y`	
`X =< Y`	
`X >= Y`	
`X = Y`	

```
Term =.. List
Term ^=.. List
X == Y
X \== Y
X @< Y
X @> Y
X @=< Y
X @>= Y
[X|Y]
X ; Y
X , Y
X -> Y
X ^ Goal
```

A.2 List of Standard Functions

`X + Y`	
`X - Y`	
`X * Y`	
`X / Y`	
`X // Y`	integer division
`X mod Y`	
`-X`	
`X \/ Y`	bitwise OR
`X /\ Y`	bitwise AND
`\ X`	bitwise negate
`X >> Y`	logical shift right
`X << Y`	logical shift left
`sin(X)`	
`cos(X)`	
`tan(X)`	
`float(X)`	
`floor(X)`	
`exp(X)`	
`log(X)`	logarithm with base e
`log10(X)`	logarithm with base 10
`sqrt(X)`	
`asin(X)`	
`acos(X)`	

```
atan(X)
```

A.3 List of Standard Operators

The following operators are provided with the XSB system:

```
op(1200, xfx, [ :-, --> ])
op(1200,  fx, [ :-, ?- ])
op(1198, xfx, [ ::- ])
op(1150, xfx, [ dynamic, hilog, multifile ])
op(1100,  fy, [ index ])
op(1100,  fx, [ ;, table, mode, export, local, parallel ])
op(1100, xfx, [ using ])
op(1050,  fy, [ import ])
op(1050, xfx, [ from ])
op(1050, xfy, [ -> ])
op(1000, xfy, [ ',' ])
op( 900,  fy, [ not, \+, spy, nospy ])
op( 700, xfx, [ =, \=, ==, \==, @<, @=<, @>, @=>,
                =.., ^=.., is, =:=, =\=, <, =<, >, >= ])
op( 661, xfy, [ '.'  ])
op( 600, xfy, [ : ])
op( 500, yfx, [ +, -, /\, \/ ])
op( 500,  fx, [ +, - ])
op( 400, yfx, [ *, /, //, mod, <<, >>, \ ])
op( 200, xfy, [ ^ ])
op( 100,  hy, [ ~, ^~ ])          HiLog operators.
```

while the following is the list of operators in the Prolog draft standard that are *not* provided:

```
op(1150,  fx, [ discontiguous ])
op( 400, yfx, [ rem ])
op( 200, xfx, [ ** ])
```

Appendix B

List of Primitive Predicates

```
psc_name(PSC, String)
psc_arity(PSC, Arity)
psc_type(PSC, Type)
psc_prop(PSC, Term)
psc_set_type(PSC, Type, Perm)
psc_set_prop(PSC, Term, Perm)
file_open(FileName, RWAMode, File)
file_close(File)
file_get(File, Char)
file_put(File, Char)
term_psc(Term, PSC)
term_type(Term, Type)
term_compare(Term1, Term2, Res)
term_new(PSC, Term)
term_arg(Term, Index, Arg)
term_set_arg(Term, Index, Arg, Perm)
stat_flag(Flag, Value)
stat_set_flag(Flag, Value, Perm)
buff_alloc(Size, Buffer, Perm)
buff_dealloc(Buffer, OldSize, NewSize, Perm)
buff_word(Buffer, Displacement, Value)
buff_set_word(Buffer, Displacement, Value)
buff_byte(Buffer, Displacement, Value)
buff_set_byte(Buffer, Displacement, Value)
buff_cell(Buffer, Displacement, Term)
buff_set_cell(Buffer, Displacement, Type, Term)
buff_set_var(Buffer, Displacement, BufferSize, Var)
```

```
code_call(CodeAddr, Term, CodeType)
str_len(String, Length)
str_cpy(String, Buffer)
str_cat(String1, String2, String3)
str_cmp(String1, String2, Res)
str_hsh(String, Arity, HashSize, HashVal)
str_insert(InString, OutString)
stat_sta(Amount)
stat_cputime(CPUTime)
code_load(ByteCodeFileName, InitAddr, ExportCode)
file_getword(File, Word)
file_putword(File, Word)
psc_insert(Name, Arity, PSC, ModuleName)
psc_import(Pname, Arity, ModuleName)
file_getbuf(File, Bytes, Buffer, Offset)
file_putbuf(File, Bytes, Buffer, Offset)
file_gettoken(File, Char, Type, Value, NextChar)
file_puttoken(File, Type, Value)
psc_insertmod(ModName, Def, PSC)
load_seg(SegNo, TextBytes, IndexBytes, File, InitAddr)
term_hash(Term, HashSize, HashVal)
unload_seg(CodeBuffer)
load_obj(FileName, ModulePSC, LD_Option, InitAddr)
expand_file_name(FileName, ExpandedFileName)
getenv(EnvVariable, EnvValue)
sys_syscall(CallNo, Result)
sys_system(Command, Result)
sys_gethost(HostName, Result)
sys_errno(ErrorNumber)
file_time(File, Time)
buff_assign_word(Buf, Displacement, Value)
psc_env(PSC, Environment)
psc_spy(PSC, Spied)
psc_tabled(PSC, Tabled)
tip_prop(Tip, Field, Get/Set, Value)
dereference_it_damn_it(Pair_Pointer, Pair)
pair_psc(PSC_Pair, PSC)
pair_next(PSC_Pair, Next_PSC_Pair)
next_bucket(HashTablePair, NextHashTablePair)
heap_copyarg(From, To)
table_intern(Pointer, Internal_Pointer)
is_xwammode(Flag)
```

```
close_open_tables
call_hash(CallPtr, HashCallPtr, Arity, HashSize, TablePtr)
print_predicate_table(PSC)
analyze_predicate_table(PSC)
comp_code_call(CodeAddr, Term)
abolish_table_info
assert_code_to_buff(Clause, Size)
assert_buff_to_clref(Head, Arity, Prref, AZ, RInd, HashTabSize, Clref)
fmt_write(FileDescr, Format, ValueList)
fmt_read(FileDescr, Format, TypeList, ValueList, Return)
```

Appendix C

List of Module names

C.1 In syslib

current_symbols	dcg	dcg_qsc	debugger
file_op	file_read	file_readtoken	file_write
hilog_symbol	interpreter	loader	machine
standard	std_sbp	string	term_expansion

C.2 In cmplib

asm	asm_inst	asm_opt	asm_pass2
aux	build_graph	builtin	compile
config	cp_opt	flatten	inprog
listutil	parse	peephole	pre_cond
preprocess	printresult	prof_sbp	sanity
singleton	spec	symtab	tabdef
ti_mod	tp_comp	tp_cond	tp_eval
tp_geninline	tp_goal	tp_index	tp_var
tprog	useinfer	varproc	writeasm

C.3 In lib

array	array1	assert	basics

consult	copyterm	db	dbclause
dbcmpl	delaylib	domain	eval
foreign	numbervars	prof_lib	record
setof	subsumes	tables	unix

Appendix D

Non Supported Features of SB-Prolog version 3.1

In this appendix we indicate some key features (sic) of XSB that may affect the users at some point in their interaction with the system.

- In XSB there is no distinction between compiled and interpreted code. Predicates `consult/[1,2]`, `reconsult/[1,2]` always compile the files that are given as their arguments instead of interpreting them. Interpreted code is only generated when using predicates `assert/1`, `asserta/1` and `assertz/1`, or the library predicate `load_dyn/1`.
- The user cannot specify the name of the byte code file during compilation. The name of the byte code file is obtained in the way explained in Section 3.9.1.
- The byte code files produced by the compiler cannot be concatenated together to produce other byte code files. This happens because of the module system. The byte code generated by the compiler, even for non-modules, contains information that indicates the fact that it is (it is not) a module and the name of the module.
- There is no assembler in the sense of the SB-Prolog assembler described in Section 8.3 of the SB-Prolog manual [Debr 88]. However, assembly files are either generated by the compiler (see Section 3.9.2), or can be produced by disassembling object code files using the `-d` command line option of the emulator (see Section 3.6).
- Mode information (i.e. `mode` compiler directives) is currently ignored by the XSB compiler, but otherwise they cause no problem in parsing the file.