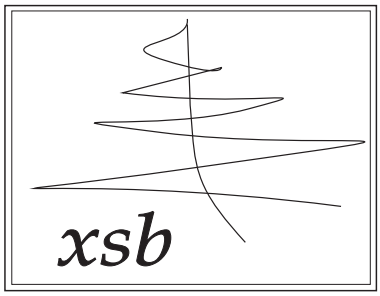
The XSB System
Version 2.2
Volume 1: Programmer's Manual



Konstantinos Sagonas
Terrance Swift
David S. Warren
Juliana Freire
Prasad Rao

with contributions from
Steve Dawson
Michael Kifer

April 4, 2000

Credits

Day-to-day care and feeding of XSB including bug fixes, ports, and configuration management has been done by Kostis Sagonas, David Warren, Terrance Swift, Prasad Rao, Steve Dawson, Juliana Freire, Baoqiu Cui and Michael Kifer.

In Version 2.2, the core engine development of the SLG-WAM has been mainly implemented by Terrance Swift, Kostis Sagonas, Prasad Rao and Juliana Freire. The breakdown, roughly, was that Terrance Swift wrote the initial tabling engine and builtins. Prasad Rao wrote the trie-based table access routines, and Kostis Sagonas implemented most of tabled negation. Juliana Freire revised the table scheduling mechanism starting from Version 1.5.0 to create a more efficient engine, and implemented the engine for local evaluation. Starting from XSB Version 2.0, XSB includes another tabling engine, CHAT, which was designed and developed by Kostis Sagonas and Bart Demoen. CHAT supports heap garbage collection (both based on a mark&slide and on a mark&copy algorithm) which was developed and implemented by Bart Demoen and Kostis Sagonas. Memory expansion code for WAM stacks was written by Ernie Johnson and Bart Demoen, while memory management code for CHAT areas was written by Bart Demoen and Kostis Sagonas. Rui Marques improved the trailing of the SLG-WAM and rewrote much of the engine to make it compliant with 64-bit architectures. Assert and retract code was based on code written by Jiyang Xu and significantly revised by David S. Warren and Rui Marques. Trie assert and retract code was written by Prasad Rao. The current version of `findall/3` was re-written from scratch by Bart Demoen.

In the XSB complier, Kostis Sagonas was responsible for HiLog compilation and associated builtins. Steve Dawson implemented Unification Factoring. The `auto_table` and `suppl_table` directives were written by Kostis Sagonas. The DCG expansion module was written by Kostis Sagonas. The handling of the `multifile` directive was written by Baoqiu Cui. C.R. Ramakrishnan wrote the mode analyzer for XSB. The safety check for tabling within the scope of cuts was written by Kate Dvortsova.

Michael Kifer rewrote parts of the XSB code to make XSB configurable with GNU's Autoconf. Harald Schroepfer helped the XSB group with the Solaris port, and Yiorgos Adamopoulos suggested the bits to use for the HP-700 series port. Steven Dawson, Larry B. Daniel and Franklin Chen were responsible for the MkLinux and Solaris x86 ports.

GPP, the source code preprocessor used by XSB, was written by Denis Auroux. He also wrote the GPP manual reproduced in Appendix A.

The starting point of XSB (in 1990) was PSB-Prolog 2.0 by Jiyang Xu. PSB-Prolog in its turn was based on SB-Prolog, primarily designed and written by Saumya Debray, David S. Warren, and Jiyang Xu. Thanks are also due to Weidong Chen for his work on Prolog clause indexing for SB-Prolog and to Richard O'Keefe, who contributed the Prolog code for the Prolog reader and the C code for the tokenizer.

Contents

1 Introduction **1**

2 Getting Started with XSB **5**

2.1 Installing XSB under UNIX 5

2.1.1 Possible Installation Problems 7

2.2 Installing XSB under Windows 8

2.2.1 Using Cygnus Software's CygWin32 8

2.2.2 Using Microsoft Visual C++ 8

2.3 Invoking XSB 9

2.4 Compiling XSB programs 10

2.5 Sample XSB Programs 10

2.6 Exiting XSB 11

3 System Description **12**

3.1 Entering and Exiting XSB 12

3.2 The System and its Directories 13

3.3 The Module system of XSB 13

3.4 The Dynamic Loader and its Search Path 16

3.4.1 Changing the Default Search Path and the Packaging System 16

3.4.2 Dynamically loading predicates in the interpreter 18

3.5 Command Line Arguments 18

3.6 Memory Management 22

3.7 Compiling and Consulting 22

3.8 The Compiler 24

3.8.1 Invoking the Compiler 24

3.8.2 Compiler Options . . . 25
3.8.3 Specialisation . . . 28
3.8.4 Compiler Directives . . . 30
3.8.5 Inline Predicates . . . 35

4 Syntax **37**
4.1 Terms . . . 37
4.1.1 Integers . . . 37
4.1.2 Floating-point Numbers . . . 38
4.1.3 Atoms . . . 38
4.1.4 Variables . . . 39
4.1.5 Compound Terms . . . 39
4.1.6 Lists . . . 40
4.2 From HiLog to Prolog . . . 42
4.3 Operators . . . 43

5 Using Tabling in XSB: A Tutorial Introduction **47**
5.1 XSB as a Prolog System . . . 47
5.2 Tabling in Definite Programs . . . 48
5.3 Stratified Normal Programs . . . 53
5.3.1 Non-stratified Programs . . . 55
5.3.2 On Beyond Zebra: Implementing Other Semantics for Non-stratified Programs 59
5.4 Tabled Aggregation . . . 61
5.4.1 Local Evaluation . . . 63

6 Standard Predicates **64**
6.1 Input and Output . . . 64
6.1.1 File Handling . . . 64
6.1.2 Character I/O . . . 66
6.1.3 Term I/O . . . 67
6.2 Special I/O . . . 69
6.3 Convenience . . . 69
6.4 Negation and Control . . . 70

6.5 Meta-Logical . . . 71
6.6 All Solutions and Aggregate Predicates . . . 85
6.6.1 Tabling Aggregate Predicates . . . 87
6.7 Comparison . . . 91
6.8 Meta-Predicates . . . 94
6.9 Information about the State of the Program . . . 94
6.10 Modification of the Database . . . 105
6.11 Execution State . . . 109
6.12 Tabling . . . 113

7 Debugging 120
7.1 High-Level Tracing . . . 120
7.2 Low-Level Tracing . . . 123

8 Definite Clause Grammars 125
8.1 General Description . . . 125
8.2 Translation of Definite Clause Grammar rules . . . 126
8.3 Definite Clause Grammar predicates . . . 128
8.4 Two differences with other Prologs . . . 129
8.5 Interaction of Definite Clause Grammars and Tabling . . . 130

9 Restrictions and Current Known Bugs 132
9.1 Current Restrictions . . . 132
9.2 Known Bugs . . . 133

A GPP - Generic Preprocessor 135
A.1 Description . . . 135
A.2 Syntax . . . 136
A.3 Options . . . 136
A.4 Syntax Specification . . . 139
A.5 Evaluation Rules . . . 142
A.6 Meta-macros . . . 143
A.7 Examples . . . 146
A.8 Advanced Examples . . . 151

A.9 Author 153

B Standard Predicates and Functions **154**

B.1 List of Standard Predicates 154

B.2 List of Standard Functions 158

B.3 List of Standard Operators 159

C List of Module names **160**

C.1 In syslib 160

C.2 In cmplib 160

Chapter 1

Introduction

XSB is a research-oriented Logic Programming system for Unix and Windows-based systems. In addition to providing all the functionality of Prolog, XSB contains several features not usually found in Logic Programming systems, including

- Evaluation according to the Well-Founded Semantics [44] through full SLG resolution;
- A compiled HiLog implementation;
- A variety of indexing techniques for asserted code, along with a novel transformation technique called *unification factoring* that can improve program speed and indexing for compiled code;
- A number of interfaces to other software systems, such a C, Java, Perl and Oracle.
- Extensive pattern matching libraries, which are especially useful for Web applications.
- Preprosessors and Interpreters so that XSB can be used to evaluate programs that are based on advanced formalisms, such as extended logic progams (according to the Well-Founded Semantics [1]); Generalized Annotated Programs [23]; and F-Logic [22].
- Source code availability for portability and extensibility.

Though XSB can be used as a Prolog system[1], we avoid referring 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 beyond those of a typical Prolog system. We feel that these capabilities justify viewing XSB as a new paradigm for Logic Programming.

To understand the implications of SLG resolution [8], recall that Prolog is based on a depth-first search through trees that are built using program clause resolution (SLD). As such, Prolog is susceptible to getting lost in an infinite branch of a search tree, where it may loop 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] Many of the Prolog components of XSB are based on PSB-Prolog [48], which itself is based on version 2.0 of SB-Prolog [13].

```
:- table ancestor/2.

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

will terminate in XSB, since `ancestor/2` is compiled as a tabled predicate; Prolog systems, however, 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, an `auto_table` compiler directive can be used to direct the system to invoke a simple static analysis to decide what predicates to table (see Section 3.8.4). This power to solve recursive queries has proven very useful in a number of areas, including deductive databases, language processing [24, 25], program analysis [12, 9, 5], model checking [32] and diagnosis [34]. 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 other tabling methods such as OLDT resolution [43] (see Chapter 5 for details).

Example 1.0.1 *The use of tabling also makes possible the evaluation of programs with non-stratified negation through its implementation of the* well-founded semantics *[44]. When logic programming rules have negation, paradoxes become possible. As an example consider one of Russell's paradoxes — the barber in a town shaves every person who does not shave himself — written as a logic program.*

```
:- table shaves/2.

shaves(barber,Person):- person(Person), tnot(shaves(Person,Person)).
person(barber).
person(mayor).
```

Logically speaking, the meaning of this progam should be that the barber shaves the mayor, but the case of the barber is trickier. If we conclude that the barber does not have himself our meaning does not reflect the first rule in the program. If we conclude that the barber does shave himself, we have reached that conclusion using information beyond what is provided in the progra. The well-founded semantics, does not treat`shaves(barber,barber)` *as either true or false, but as* undefined. *Prolog, of course, would enter an infinite loop. XSB's treatment of negation is discussed further in Chapter 5.*

The second important extension in XSB is support of HiLog programming [6, 39]. HiLog allows a 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(R)/2` is (syntactically) a second-order predicate which, given any relation `R`, returns its transitive closure relation `closure(R)`. With XSB, support is provided for reading and writing

HiLog terms, converting them to or from internal format as necessary (see Section 4.2). Special meta-logical standard predicates (see Section 6.5) are also provided for inspection and handling of HiLog terms. Unlike earlier versions of XSB (prior to version 1.3.1) the current version automatically provides *full compilation of HiLog predicates*. 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 [39].

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).
```

A further goal of XSB is to provide in implementation engine for both logic programming and for data-oriented applications such as in-memory deductive database queries and data mining [36]. One prerequisite for this functionality is the ability to load a large amount of data very quickly. We have taken care to code in C a compiler for asserted clauses. The result is that the speed of asserting and retracting code is faster in XSB than in any other Prolog system of which we are aware. At the same time, because asserted code is compiled into SLG-WAM code, the speed of executing asserted code in XSB is faster than that of many other Prologs as well. We note however, that XSB does not follow the semantics of assert specified in [27].

Data oriented applications may also require indices other than Prolog's first argument indexing. XSB offers a variety of indexing techniques for asserted code. Clauses can be indexed on a groups of arguments or on alternative arguments. For instance, the executable directive `index(p/4,[3,2+1])` specifies indexes on the (outer functor symbol of) the third argument *or* on a combination of (the outer function symbol of) the second and first arguments. If data is expected to be structured within function symbols and is in unit clauses, the directive `index(p/4,trie)` constructs an indexing trie of the `p/4` clauses using a left-to-right traversal through each clause. Representing data in this way allows discrimination of information nested arbitrarily deep within clauses. These modes of indexing can be combined: `index(p/4,[3,2+1],trie)` creates alternative trie indices beginning with the third argument and with the second and first argument. Using such indexing XSB can efficiently perform intensive analyses of in-memory knowledge bases with 1 million or so facts. Indexing techniques for asserted code are covered in Section 6.10.

For compiled code, XSB offers *unification factoring*, which extends clause indexing methods found in functional programming into the logic programming framework. Briefly, unification factoring can offer not only complete indexing through non-deterministic indexing automata, but can also *factor* elementary unification operations. The general technique is described in [11], and the XSB directives needed to use it are covered in Section 3.8.

A number of interfaces are available to link XSB to other systems. In UNIX systems XSB can be directly linked into C programs; in Windows-based system XSB can be linked into C programs through a DLL interface. On either class of operating system, C functions can be made callable from XSB either directly within a process, or using a socket library. XSB can access external data in a variety of ways: through an Oracle interface, through an ODBC interface, or through a variety of mechanisms to read data from flat files. These interfaces are all described in Volume 2 of this

manual.

Another feature of XSB is its support for extensions of normal logic programs through preprocessing libraries. Currently supported are Extended logic programs (under the well-founded semantics), F-Logic, and Annotated Logic Programs. These libraries are described in Volume 2 of this manual.

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

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). See Section 3.8.4 for more details. Also, the manual uses UNIX syntax for files and directories except when it specifically addresses other operating systems such as Windows.

Finally, we note that XSB is under continuous development, and this document —intended to be the user manual— reflects the current status (Version 2.2) of our system. While we have taken great effort to create a robust and efficient system, we would like to emphasize that XSB is also a research system and is to some degree experimental. When the research features of XSB — tabling, HiLog, and Indexing Techniques — are discussed in this manual, we also cite documents where they are fully explained. All of these documents can be found via the world-wide web or anonymous ftp from {`www/ftp`}`.cs.sunysb.edu`, the same host from which XSB can be obtained.

While some of Version 2.2 is subject to change in future releases, we will try to be as upward-compatible as possible. We would also like to hear from experienced users of our system about features they would like us to include. We do try to accomodate serious users of XSB whenever we can. Finally, we must mention that the use of undocumented features is not supported, and at the user's own risk.

Chapter 2

Getting Started with XSB

This section describes the steps needed to install XSB under UNIX and under Windows.

2.1 Installing XSB under UNIX

If you are installing on a UNIX platform, the version of XSB that you received may not include all the object code files so that an installation will be necessary. The easiest way to install XSB is to use the following procedure.

1. Decide in which directory in your file system you want to install XSB and copy or move XSB there.

2. Make sure that after you have obtained XSB by anonymous ftp (using the `bin`ary option) or from the web, you have uncompressed it by following the instructions found in the file `README`.

3. Note that after you uncompress and untar the XSB tar file, a subdirectory `XSB` will be tacked on to the current directory. All XSB files will be located in that subdirectory.

   In the rest of this manual, let us use `$XSB_DIR` to refer to this subdirectory. Note the original directory structure of XSB must be maintained, namely, the directory `$XSB_DIR` should contain all the subdirectories and files that came with the distribution. In particular, the following directories are required for XSB to work: `emu`, `syslib`, `cmplib`, `lib`, `packages`, `build`, and `etc`.

4. Change directory to `$XSB_DIR/build` and then run these commands:

   ```
   configure
   makexsb
   ```

   This is it!

   In addition, it is now possible to install XSB in a shared directory (*e.g.*, `/usr/local`) for everyone to use. In this situation, you should use the following sequence of commands:

```
configure --prefix=$SHARED_XSB
makexsb
makexsb install
```

where `$SHARED_XSB` denotes the shared directory where XSB is installed. In all cases, XSB can be run using the script

```
$XSB_DIR/bin/xsb
```

However, if XSB is installed in a central location, the script for general use is:

```
<central-installation-directory>/<xsb-version>/bin/xsb
```

Important: The XSB executable determines the location of the libraries it needs based on the full path name by which it was invoked. The "smart script" `bin/xsb` also uses its full path name to determine the location of the various scripts that it needs in order to figure out the configuration of your machine. Therefore, there are certain limitations on how XSB can be invoked.

Here are some legal ways to invoke XSB:

1. invoking the smart script `bin/xsb` or the XSB executable using their absolute or relative path name.
2. using an alias for `bin/xsb` or the executable.
3. creating a new shell script that invokes either `bin/xsb` or the XSB executable using their *full* path names.

Here are some ways that are guaranteed to not work in some or all cases:

1. creating a hard link to either `bin/xsb` or the executable and using *it* to invoke XSB. (Symbolic links should be ok.)
2. changing the relative position of either `bin/xsb` or the XSB executable with respect to the rest of the XSB directory tree.

Type of Machine. The configureation script automatically detects your machine and OS type, and builds XSB accordingly. Moreover, you can build XSB for different architectures while using the same tree and the same installation directory provided, of course, that these machines are sharing this directory, say using NFS or Samba. All you will have to do is to login to a different machine with a different architecture or OS type, and repeat the above sequence of comands.

The configuration files for different architectures reside in different directories, and there is no danger of an architecture conflict. Moreover, you can keep using the same `./bin/xsb` script regardless of the architecture. It will detect your configuration and will use the right files for the right architecture!

Choice of the C Compiler and Other options The `configure` script will attempt to use `gcc`, if it is available. Otherwise, it will revert to `cc` or `acc`. Some versions of `gcc` are broken, in which case you would have to give `configure` an additional directive `--with-cc`. If you must use some special comiler, use `--with-cc=your-own-compiler`. You can also `--disable-optimization` (to change the default), `--enable-debug`, and there are many other options. Type `configure --help` to see them all. Also see the file `$XSB_DIR/INSTALL` for more details.

Other options are of interest to advanced users who wish to experiment with XSB, or to use XSB for large-scale projects. In general, however users need not concern themselves with these options.

Type of Scheduling Strategy. The ordering of operations within a tabled evaluation can drastically affect its performance. XSB provides two scheduling strategies: Batched Evaluation and Local Evaluation. Batched Evaluation is the default scheduling strategy for XSB and evaluates queries to reduce the time to the first answer of a query. Local Evaluation can be chosen via the `--enable-local-scheduling` configure option. Detailed explanations can be found in [18].

Type of Memory Management. Routines for managing execution stacks for tabled evaluations can be quite complex, due to interdependencies of tabled subgoals. Indeed, memory management algorithms can be based on common elements are shared among computation states or are copied. The default configuration of XSB shares these elements while the option `--enable-chat` copies these elements. While sharing and copying have minor performance differences, the main reason to try the `--enable-chat` configuration is to use a heap garbage collector that has been written for it. See [35, 14, 15, 16] for in-depth discussion of the engine memory management.

2.1.1 Possible Installation Problems

Lack of Space for Optimized Compilation of C Code When making the optimized version of the emulator, the temporary space available to the C compiler for intermediate files is sometimes not 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 files `emuloop.c`, and `tries.c`, without changing the default `tmp` directory and increasing the swap space. Depending on your C compiler, the amount and nature of `/tmp` and swap space of your machine you may or may not encounter problems. 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 any 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 compiling. Usually changing the default directory can be done by declaring/modifying the `TMPDIR` environment variable as follows:

```
setenv TMPDIR dir
```

Missing XSB Object Files When an object (*.O) file is missing from the `lib` directories you can normally run the `make` command in that directory to restore it (instructions for doing so are given in Chapter 2). However, to restore an object file in the directories `syslib` and `cmplib`, one needs to have a separate Prolog compiler accessible (such as a separate copy of XSB), because the XSB compiler uses most of the files in these two directories and hence will not function when some of them are missing. For this reason, distributed versions normally include all the object files in `syslib` and `cmplib`.

2.2 Installing XSB under Windows

2.2.1 Using Cygnus Software's CygWin32

This is easy: just follow the Unix instructions. This is the preferred way to run XSB under Windows, because this ensures that all features of XSB are available.

2.2.2 Using Microsoft Visual C++

1. XSB will unpack into a subdirectory named `xsb`. Assuming that you have `XSB.ZIP` in the `$XSB_DIR` directory, you can issue the command

   ```
   unzip386 xsb.zip
   ```

   which will install XSB in the subdirectory `xsb`.

2. If you decide to move XSB to some other place, make sure that the entire directory tree is moved — XSB executable looks for the files it needs relatively to its current position in the file system.

You can compile XSB under Microsoft Visual C++ compiler to create a console-supported top loop or a DLL by following these steps:

1. cd build
2. Type:
   `makexsb_wind "CFG=option" ["DLL=yes"] ["ORACLE=yes"] ["SITE_LIBS=libraries"]`
   - The items in square brackets are optional.
   - The options for `CFG` are: *release* or *debug*. The latter is used when you want to compile XSB with debugging enabled.
   - The other parameters to `makexsb_wind` are optional. The `DLL` parameter tells Visual C++ to compile XSB as a DLL. The `ORACLE` parameter compiles XSB with support for Oracle DBMS. If `ORACLE` is specified, you **must** also specify the necessary Oracle libraries using the parameter `SITE_LIBS`.

3. The above command will compile XSB as requested and will put the XSB executable in:

   `$XSB_DIR\config\x86-pc-windows\bin\xsb.exe`

   If you requested to compile XSB as a DLL, then the DLL will be placed in

   `$XSB_DIR\config\x86-pc-windows\bin\xsb.dll`

Note: the XSB executable and the DLL can coexist in the same source tree structure. However, if you first compiled XSB as an executable and then want to compile it as a DLL (or vice versa), then you must run

```
makexsb_wind  clean
```

in between.

2.3 Invoking XSB

Under Unix, XSB can be invoked by the command:

```
$XSB_DIR/bin/xsb
```

if you have installed XSB in your private directory. If XSB is instaled in a shared directory (*e.g.*, $SHARED_XSB for the entire site (UNIX only), then you should use

```
$SHARED_XSB/bin/xsb
```

In both cases, you will find yourself in the top level interpreter. As mentioned above, this script automatically detects the system configuration you are running on and will use the right files and executables. (Of course, XSB should have been built for that architecture earlier.)

Under Windows, you should invoke XSB by typing:

`$XSB_DIR\config\x86-pc-windows\bin\xsb.exe`

You may want to make an alias such as `xsb` to the above commands, for convenience, or you might want to put the directory where the XSB command is found in the `$PATH` environment variable. However, you should **not** make hard links to this script or to the XSB executable. If you invoke XSB via such a hard link, XSB will likely be confused and will not find its libraries. That said, you **can** create other scripts and cal the above script from there.

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.

2.4 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 3.3.

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.

2.5 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).[1] 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> $XSB_DIR/bin/xsb
XSB Version 2.0 (Gouden Carolus) of June 27, 1999
[i586-pc-linux-gnu; mode: optimal; engine: slg-wam; scheduling: batched]
| ?- [queens].
[queens loaded]

yes
| ?- demo.

% ...... output from queens program .......

Time used: 0.4810 sec

yes
| ?- statistics.

memory (total)         1906488 bytes:       203452 in use,      1703036 free
  permanent space       202552 bytes
```

[1] This convention does not apply to the subdirectories of the examples directory, which illustrate advanced features of XSB.

```
  glob/loc space        786432 bytes:          432 in use,       786000 free
    global                                     240 bytes
    local                                      192 bytes
  trail/cp space        786432 bytes:          468 in use,       785964 free
    trail                                      132 bytes
    choice point                               336 bytes
  SLG subgoal space          0 bytes:            0 in use,            0 free
  SLG unific. space      65536 bytes:            0 in use,        65536 free
  SLG completion         65536 bytes:            0 in use,        65536 free
  SLG trie space             0 bytes:            0 in use,            0 free
   (call+ret. trie           0 bytes,     trie hash tables            0 bytes)

      0 subgoals currently in tables
      0 subgoal check/insert attempts inserted     0 subgoals in the tables
      0 answer  check/insert attempts inserted     0 answers  in the tables

       Time: 0.610 sec. cputime,  18.048 sec. elapsetime

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

End XSB (cputime 1.19 secs, elapsetime 270.25 secs)
my_favourite_prompt>
```

2.6 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

Throughout this chapter, we use `$XSB_DIR` to refer to the directory in which XSB was installed.

3.1 Entering and Exiting XSB

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

```
$XSB_DIR/bin/xsb
```

or, if, after being built, XSB is later installed at a central location, `$SHARED_XSB`.

```
$SHARED_XSB/bin/xsb
```

and indeed, using this command, invokes XSB's top level interpreter which is the usual way of using XSB

Version 2.2 of XSB can also directly execute object code files from the command line interface. Suppose you have a top-level routine `go` in a file `foo.P` that you would like to run from the UNIX or Windows command line. As long as `foo.P` contains a directive `:- go.`, and `foo` has been compiled to an object file (`foo.O`), then

```
$XSB_DIR/bin/xsb -B foo.O
```

will execute `go`, loading the appropriate files as needed. In fact the command `$XSB_DIR/bin/xsb` is equivalent to the command:

```
$XSB_DIR/bin/xsb -B $XSB_DIR/syslib/loader.O
```

There are several ways to exit XSB. A user may issue the command `halt.` or `end_of_file`, or simply type `CTRL-d` at the XSB prompt. To interrupt XSB while it is executing a query, strike `CTRL-c`.

3.2 The System and its Directories

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

1. `build`
2. `docs`
3. `emu`
4. `etc`
5. `examples`
6. `cmplib`
7. `lib`
8. `packages`
9. `syslib`

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 SLG-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 `packages` contains the various applications, written in XSB, which are not part of the system per se.

You must already be familiar with the `build` directory, which is what you must have used to build XSB. This directory contains XSB configuration scripts. The directory `etc` contains miscellaneous files used by XSB.

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

3.3 The Module system of XSB

XSB has been designed as a module-oriented Prolog system. Modules provide 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*. Briefly, 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 [1].

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 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,

[1] Operator symbols can be exported as any other symbols, but their precedence must be redeclared in the importing module.

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 a given module are not visible to 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) are contained in `usermod`.

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 predicate symbols that are defined in that module. As a consequence, a module *cannot* export predicate 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 abort the compilation.
- 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 that defines it, 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 into 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.

3.4 The Dynamic Loader and its Search Path

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

When a predicate imported from another module (see section 3.3) is called during execution, the dynamic loader is invoked automatically if the module is not yet loaded into the system, The default action of the dynamic loader is to search for the byte code file of the module first in the system library directories (in the order `lib, syslib`, and then `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 will be displayed on the current output stream reporting that the module was not found.

In fact, XSB loads the compiler and most system modules this way. Because of dynamic loading, the time it takes to compile a file is slightly longer than usual the first time the compiler is invoked in a session.

3.4.1 Changing the Default Search Path and the Packaging System

Users are allowed to supply their own library directories and also to override 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 `.xsb/xsbrc.P` in the user's home directory. The `.xsb/xsbrc.P` file, which is automatically consulted by the XSB interpreter, might look like the following:

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

After loading the module of the above example, the current working directory is searched first (as opposed to the default action of searching it last). Also, XSB's system library directories (`lib, syslib`, and `cmplib`), will now be searched *after* searching the user's, `my_friend`'s and the `"/usr/lib/sbprolog/"` directory.

In fact, XSB also uses `library_directory/1` for internal purposes. For instance, before the user's `.xsb/xsbrc.P` is consulted, XSB puts the `packages` directory and the directory

```
.xsb/config/$CONFIGURATION
```

on the library search path. The directory `.xsb/config/$CONFIGURATION` is used to store user libraries that are machine or OS dependent. (`$CONFIGURATION` for a machine is something that

looks like `sparc-sun-solaris2.6` or `pc-linux-gnu`, and is selected by XSB automatically at run time).

Note that the file `.xsb/xsbrc.P` is not limited to setting the library search path. In fact, arbitrary Prolog code can go there.

We emphasize that in the presense of a `.xsb/xsbrc.P` file *it is the user's responsibility to avoid module name clashes with modules in XSB's system library directories.* Such name clashes can cause the system to behave strangely since these modules will probably have different semantics from that expected by the XSB system code. The list of module names in XSB's system library directories can be found in appendix C.

Apart from the user libraries, XSB now has a simple packaging system. A *package* is an application consisting of one or more files that are organized in a subdirectory of one of the XSB system or user libraries. The system directory `$XSB_DIR/packages` has several examples of such packages. Packages are convenient as a means of organizing large XSB applications, and for simplifying user interaction with such applications. User-level packaging is implemented through the predicate

```
bootstrap_userpackage(+LibraryDir, +PackageDir, +PackageName).
```

which must be imported from the `packaging` module.

To illustrate, suppose you wanted to create a package, `foobar`, inside your own library, `my_lib`. Here is a sequence of steps you can follow:

1. Make sure that `my_lib` is on the library search path by putting an appropriate assert statement in your `xsbrc.P`.

2. Make subdirectory `~/my_lib/foobar` and organize all the package files there. Designate one file, say, `foo.P`, as the entry point, *i.e.*, the application file that must be loaded first.

3. Create the interface program `~/my_lib/foobar.P` with the following content:

   ```
   :- bootstrap_userpackage('~/my_lib', 'foobar', foobar), [foo].
   ```

   The interface program and the package directory do not need to have the same name, but it is convenient to follow the above naming schema.

4. Now, if you need to invoke the `foobar` application, you can simply type `[foobar].` at the XSB prompt. This is because both and `~/my_lib/foobar` have already been automatically added to the library search path.

5. If your application files export many predicates, you can simplify the use of your package by having `~/my_lib/foobar.P` import all these predicates, renaming them, and then exporting them. This provides a uniform interface to the `foobar` module, since all the package predicates are can now be imported from just one module, `foobar`.

In addition to adding the appropriate directory to the library search path, the predicate `bootstrap_userpackage/3` also adds information to the predicate `package_configuration/3`, so that other applications could query the information about loaded packages.

Packages can also be unloaded using the predicate `unload_package/1`. For instance,

```
:- unload_package(foobar).
```

removes the directory `~/my_lib/foobar` from the library search path and deletes the associated information from `package_configuration/3`.

3.4.2 Dynamically loading predicates in the interpreter

Modules are usually loaded into an environment when they are consulted (see section 3.7). Specific predicates from a module can also be imported into the run-time environment through the standard predicate `import PredList from Module`. Here, `PredList` can either be a Prolog list or a comma list. (The `import/1` can also be used as a directive in a source module (see section 3.3).

We provide a sample session for compiling, dynamically loading, and querying a user-defined module named `quick_sort`. For this example we assume that `quick_sort` is a file in the current working directory, and contains the definitions of the predicates `concat/3` and `qsort/2`, both of which are exported.

```
| ?- 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) the predicate is called.

3.5 Command Line Arguments

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

```
       xsb [flags] [-l] [-i]
xsb [flags] -n
xsb [flags] module
xsb [flags] -B boot_module [-D cmd_loop_driver] [-t] [-e goal]
xsb [flags] -B module_to_disassemble -d
xsb -[h | v]
xsb --help | --version | --nobanner | --quietload | --noprompt

memory management flags:
    -c tcpsize | -m glsize | -o complsize | -u pdlsize | -r | -g gc_type
miscellaneous flags:
    -s | -T

module:
    Module to execute after XSB starts up.
    Module should have no suffixes, no directory part, and
    the file module.O must be on the library search path.
       boot_module:
    This is a developer's option.
    The -B flags tells XSB which bootstraping module to use instead
    of the standard loader. The loader must be specified using its
    full pathname, and boot_module.O must exist.
     module_to_disassemble:
    This is a developer's option.
    The -d flag tells XSB to act as a disassembler.
    The -B flag specifies the module to disassemble.
cmd_loop_driver:
    The top-level command loop driver to be used instead of the
    standard one. Usually needed when XSB is run as a server.

           -i : bring up the XSB interpreter
       -e goal : evaluate goal when XSB starts up
           -l : the interpreter prints unbound variables using letters
           -n : used when calling XSB from C
    -B : specify the boot module to use in lieu of the standard loader
    -D : Sets top-level command loop driver to replace the default.
           -t : trace execution at the SLG-WAM instruction level
 (for this to work, build XSB with the --debug option)
    -d : disassemble the loader and exit
         -c N : allocate N KB for the trail/choice-point stack
         -m N : allocate N KB for the local/global stack
         -o N : allocate N KB for the SLG completion stackof
         -u N : allocate N KB for the SLG unification (table copy) stack
           -r : turn off automatic stack expansion
    -g gc_type : choose the garbage collection ("none", "sliding", or "copying")
```

```
          -s : maintains more detailed statistical information
          -T : print a trace of each called predicate
 -v, --version : print the version and configuration information about XSB.
    -h, --help : print this help message
    --nobanner : don't show the XSB banner on startup
   --quietload : don't show the 'module loaded' messages
    --noprompt : don't show prompt (for non-interactive use)
```

The order in which these options appear makes no difference.

`-i` Brings up the XSB interpreter. This is the normal use and because of this, use of this option is optional and is only kept for backwards compatibility.

`-l` Forces the interpreter to print unbound variables as letters, as opposed to the default setting which prints variables as memory locations prefixed with an underscore. For example, starting XSB's interpreter with this option will print the following:

```
| ?- Y = X, Z = 3, W = foo(X,Z).

Y = A
X = A
Z = 3
W = foo(A,3)
```

as opposed to something like the following:

```
| ?- Y = X, Z = 3, W = foo(X,Z).

Y = _10073976
X = _10073976
Z = 3
W = foo(_10073976,3);
```

`-n` used in conjunction with the `-i` option, to indicate that the usual read-eval-print top-loop is not to be entered, but instead will interface to a calling C program. See the chapter *Calling XSB from C* in Volume 2 for details.

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

`-c` ***size*** Allocates *initial size* Kbytes of space to the trail/choice-point stack area. The trail stack grows upward from the bottom of the region, and the choice point stack grows downward from the top of the region. Because this region is expanded automatically from Version 1.6.0 onward, this option should rarely need to be used. Default initial size: 768 Kbytes.

`-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 thecompletion stack area. Because this region is expanded automatically from Version 1.6.0 onward, this option should rarely need to be used. Default initial size 64 Kbytes.

`-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).

`-D` Tells XSB to use a top-level command loop driver specified here instead of the standard XSB interpreter. This is most useful when XSB is used as a server.

`-r` Turns off automatic stack expansion.

`-g gc_type` Chooses the garbage collection strategy that is employed; choice of the strategy is between `"none"` (meaning perform no garbage collection), or garbage collection based on `"sliding"` or on `"copying"`. Since garbage collection is only available when the emulator is based on a CHAT model (see also the installation options), this option only makes sense in this context; it is ineffective when the emulator is SLG-WAM based.

`-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.

`-T` Generates a trace at entry to each called predicate (both system and user-defined). This option is available mainly for people who want to modify and/or extend XSB, and it is *not* the normal way to trace XSB programs. For the latter, the builtin predicates `trace/0` or `debug/0` should be used (see Chapter 7).

Note: This option is not available when the system is being used at the non-tracing mode (see Section 7).

`-t` Traces through code at SLG-WAM 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 7).

`-e goal` Pass `goal` to XSB at starup. This goal is evaluated right before the first prompt is issued. For instance, `xsb -e "write(Hello!'), nl."'` will print a heart-warming message when XSB starts up.

`--nobanner` Start XSb without showing the startup banner. Useful in batch scripts and for interprocess communication (when XSB is launched as a subprocess).

`--quietload` Do not tell when a new module gets loaded. Again, is useful in non-interactive activities and for interprocess communication.

`--noprompt` Do not show te XSB prompt. This is useful only in batch mode and in interprocess communication when you do not want the prompt to clutter the picture.

As an example, a program which uses more heap and local stack than the default configuration of XSB might be run by invoking XSB with the command.

```
xsb -m 2000
```

3.6 Memory Management

All execution stacks are automatically expenaded in Version 2.2 including the local stack/heap region, the trail/choice point region and the completion stack region. Each of these regions begin with an initial value set by the user (or the default stated in Section 3.5), and double their size until it is not possible to do so with available system memory. At that point XSB tries to find the maximal amount of space that will still fit in system memory. Garbage collection is automatically performed for retracted clauses. In addition, heap garbage collection is automatically included when the `--enable-chat` configuration option is used. (It can be turned off; see the predicate `garbage_collection/1`.)

The program area (the area into which the code is loaded) is also dynamically expanded as needed, and the area occupied by dynamic code (created using `assert/1`, or the standard predicate `load_dyn/1`) is reclaimed when that code is retracted. Version 1.8 improves memory management for retracted dynamic code.

Version 2.2 provides memory management for table space as well. Space for tables is dynamically allocated as needed and reclaimed through use of the predicate `abolish_all_tables/0` (see Section 6.12).

3.7 Compiling and Consulting

In XSB, both compiled and interpreted code are transformed into SLG-WAM instructions. The main differences are that compiled code may be more optimized than interpreted code, and that compilation produces an object code file.

This section describes the actions of the standard predicate `consult/[1,2]` (and of `reconsult/[1,2]` which is defined to have the same actions as `consult/[1,2]`). `consult/[1,2]` is the most convenient method for entering rules into XSB's database. Though `consult` comes in many flavors, the most general form is:

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

At the time of the call both of its arguments should be instantiated (ground). `FileList` is a list of filenames or module names (see section 3.3) 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 3.8.

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:

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

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

Name Resolution : determine the module to be consulted.

Compilation : if necessary (and the source file is not too big), compile the module using predicate `compile/2` with the options specified.

Loading load the object code of the module into memory.

Importing import all the exported predicates of that module to the current working module (`usermod`).

Query Execution : execute any queries that the module may contain.

There are two steps to name resolution: determination of the proper directory prefix and determination of the proper extension. When `FileName` is absolute (i.e. in UNIX contains a slash `'/'`) determination of the proper directory prefix is straightforward. However, the user may also enter a name without any directory prefix. In this case, the directory prefix is a directory in the dynamic loader path (see section 3.4) where the source file exists. Once the directory prefix is determined, the file name is checked for an extension. If there is no extension the loader first checks for a file in the directory with the `.P` extension, (or `.c` for foreign modules) before searching for a file without the extension. Note that since directories in the dynamic loader path are searched in a predetermined order (see section 3.4), if the same file name appears in more than one of these directories, the compiler will consult the first one it encounters.

Compilation is performed if the update date of the the source file (`*.P`) is later than that of the the object file (`*.O`), *and* if the source file is not larger than the default compile size. This default compile is set to be 20,000 bytes (in `cmplib/config.P`), but can be reset by the user. If the source file is larger than the default compile size, the file will be loaded using `load_dyn/1`, and otherwise it will be compiled (`load_dyn/1` can also be called separately, see the section *Asserting Dynamic Code* for details. While `load_dyn` gives reasonibly good execution times, compilation can always be done by using `compile/[1,2]` explicitly. Currently (Version 2.2), 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.

Whether the file is compiled or dynamically loaded, the byte-code for the file is loaded into XSB's database. The default action upon loading is to delete any previous byte-code for predicates defined in the file. If this is not the desired behavior, the user may add to the file a declaration

```
:- multifile <Predicate_List> .
```

where `Predicate_List` is a list of predicates in *functor/arity* form. The effect of this declaration is to delete *only* those clauses of `predicate/arity` that were defined in the file itself.

After loading the module, all exported predicates of that module are imported into the current environment (the current working module `usermod`). For non-modules (see Section 3.3), all predicates are imported into the current working module.

Finally any queries — that is, any terms with principal functor `':-'/1` that are not directives like the ones described in Section 3.8 — are executed in the order that they are encountered.

3.8 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`.

Prior to compiling, XSB filters the programs through *GPP*, a preprocessor written by Denis Auroux (auroux@math.polytechnique.fr). This preprocessor maintains high degree of compatibility with the C preprocessor, but is more suitable for processing Prolog programs. The preprocessor is invoked with the compiler option `xpp_on` as described below. The various features of GPP are described in Appendix A.

XSB also allows the programmer to use preprocessors other than GPP. However, the modules that come with XSB distribution require GPP. This is explained below (see `xpp_on` compiler option).

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

3.8.1 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 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 compiler options specified in `OptionList` (but see Section 3.8.2 below for the precise details.)

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

is just a notational shorthand for the query:

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

The standard predicates `consult/[1,2]` call `compile/1` (if necessary). Argument `File` can be any syntactically valid UNIX or Windows 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:

$$\texttt{[}\ option_1\texttt{,}\ option_2\texttt{,}\ \ldots\texttt{,}\ option_n\ \texttt{]}\texttt{.}$$

where $option_i$ is one of the options described in Section 3.8.2.

The source file name corresponding to a given module is obtained by concatenating a directory prefix and the extension `.P` (or `.c`) to the module name. The directory prefix must be in the dynamic loader path (see Section 3.4). Note that these directories are searched in a predetermined order (see Section 3.4), so if a module with the same name appears in more than one of the directories searched, the compiler will compile the first one it encounters. In such a case, the user can override the search order by providing an absolute path name.

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

We recommend use of the extension `.P` for Prolog source file to avoid ambiguity. 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 of XSB the header files may be used to check interfaces across modules, hence it is a good programming practice to restrict header files to declarations alone.

The result of the compilation (an SLG-WAM object code file) is stored in a (⟨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 also 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` and a `.P` file must *not* exist. A header file (with extension `.H`) *must* be present for a foreign language module (see the chapter *Foreign Language Interface* in Volume 2.

3.8.2 Compiler Options

Compiler options can be set in three ways: from a global list of options (see `set_global_compiler_options/1`), from the compilation command (see `compile/2` and `consult/2`), and from a directive in the file to be compiled (see compiler directive `compiler_options/1`).

`set_global_compiler_options(+OptionsList)`

`OptionsList` is a list of compiler options (described below). Each can optionally be prefixed

by `+` or `-`, indicating that the option is to be turned on, or off, respectively. (No prefix turns the option on.) This evaluable predicate sets the global compiler options in the way indicated. These options will be used in any subsequent compilation, unless reset by another call to this predicate, or overridden by options provided in the compile invocation, or overridden by options in the file to be compiled.

The following options are currently recognized by the compiler:

`optimize` When specified, the compiler tries to optimize the object code. In Version 2.2, this option optimizes predicate calls, among other features, so execution may be considerably faster for recursive loops. 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 slightly longer. For these reasons, the use of the `optimize` option may not be suitable for the development phase, but is recommended once the code has been debugged.

`xpp_on` Filter the program through a preprocessor before sending it to the XSB compiler. By default (and for the XSB code itself), XSB uses GPP, a preprocessor developed by Denis Auroux (auroux@math.polytechnique.fr) that has high degree of compatibility with the C preprocessor, but is more suitable for Prolog syntax. In this case, the source code can include the usual C preprocessor directives, such as `#define`, `#ifdef`, and `#include`. This option can be specified both as a parameter to compile/2 and as part of the `compiler_options/1` directive inside the source file. See Appendix A for more details on GPP.

When an `#include "file"` statement is encountered, XSB directs GPP preprocessor to search for the files to include in the directories `$XSB_DIR/emu` and `$XSB_DIR/prolog_includes`. However, additional directories can be added to this search path by asserting into the predicate `xpp_include_dir/1`, which should be imported from module `parse`.

XSB predefines the constant `XSB_PROLOG`, which can be used for conditional compilation. For instance, you can write portable program to run under XSB and and other prologs that support C-style preprocessing and use conditional compilation to account for the differences:

```
#ifdef XSB_PROLOG
    XSB-specific stuff
#else
    other Prolog's stuff
#endif
    common stuff
```

However, as mentioned earlier, XSB lets the user filter programs (except the programs that belong to XSB distribution) through any preprocessor the user wants. To this end, one only needs to assert the appropriate command into the predicate `xpp_program`, which should be imported from module `parse`. The command should not include the file name—XSB appends the name of the file to be compiled to the command supplied by the user. For instance, executing

```
:- assert(xpp_program('/usr/bin/m4 -E -G')).
```

before calling the compiler will have the effect that the next XSB program passed to the compiler will be first preprocessed by the M4 macro package. Note that the XSB compiler automatically clears out the `xpp_program` predicate, so there is no need to tidy up each time. But this also means that if you need to compile several programs with a non-standard preprocessor then you must specify that non-standard preprocessor each time the program is compiled.

`auto_table` When specified as a compiler option, the effect is as described in Section 3.8.4. Briefly, a static analysis is made to determine which predicates may loop under Prolog's SLD evaluation. These predicates are compiled as tabled predicates, and SLG evaluation is used instead.

`suppl_table` The intention of this option is to direct the system to table for efficiency rather than termination. When specified, the compiler uses tabling to ensure that no predicate will depend on more than three tables or EDB facts (as specified by the declaration `edb` of Section 3.8.4). The action of `suppl_table` is independent of that of `auto_table`, in that a predicate tabled by one will not necessarily be tabled by the other. During compilation, `suppl_table` occurs after `auto_table`, and uses table declarations generated by it, if any.

`spec_repr` When specified, the compiler performs specialisation of partially instantiated calls by replacing their selected clauses with the representative of these clauses, i.e. it performs *folding* whenever possible. We note in general, 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. 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, singleton sets optimisations are not performed during specialisation. This option is necessary in Version 2.2 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 unification factoring to predicates defined in the `module` under compilation. The result is in Prolog source code form. See page 34 for more information on unification factoring.

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

`modeinfer` This option is used to trigger mode analysis. For each module compiled, the mode analyzer creates a *`module`*`.D` file that contains the mode information.

WARNING: Occasionally, the analysis itself may take a long time. As far as we have seen, the analysis times are longer than the rest of the compilation time only when the module contains recursive predicates of arity ≥ 10. If the analysis takes an unusually long time (say, more than 4 times as long as the rest of the compilation) you may want to abort and restart compilation without `modeinfer`.

`mi_warn` During mode analysis, the `.D` files corresponding to the imported modules are read in. The option `mi_warn` is used to generate warning messages if these `.D` files are outdated — *i.e.*, older than the last modification time of the source files.

`mi_foreign` This option is used *only* when mode analysis is performed on XSB system modules. This option is needed when analyzing `standard` and `machine` in `syslib`.

`sysmod` Mainly used by developers when compiling system modules. If specified, standard predicates (listed in Appendix B) are automatically available for use only if they are primitive predicates (see the file `syslib/machine.P` for a current listing of such predicates. When compiling in this mode, non primitive standard predicates must be explicitly imported from the appropriate system module.

`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.

`asm_dump, compile_off` Generates a textual representation of the SLG-WAM 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 might be cases where compiling a module with these options may cause generation of an incorrect `.A` and `.O` file. In such cases, the user can see the SLG-WAM 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 3.5).

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

3.8.3 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 3.8.2) 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 result of the specialisation transformation by using the `spec_dump` compiler option (see Section 3.8.2).

Finally, we have to mention that for technical reasons beyond the scope of this document, specialisation cannot be transparent to the user; predicates created by the transformation do appear during tracing.

3.8.4 Compiler Directives

The following compiler directives are recognized in Version 2.2 of XSB [2].

Mode Declarations

The XSB compiler accepts `mode` declarations of the form:

`:- mode` $ModeAnnot_1, \ldots, ModeAnnot_n$.

where each $ModeAnnot$ is a *mode annotation* (a *term indicator* whose arguments are elements of the set {`+`,`-`,`#`,`?`}). From Version 1.4.1 on, `mode` directives are used by the compiler for tabling directives, a use which differs from the standard use of modes in Prolog systems[3]. See Section 3.8.4 for detailed examples.

Mode annotations have the following meaning:

`+` This argument is an input to the predicate. In every invocation of the predicate, the argument position must contain a non-variable term. This term may not necessisarily be ground, but the predicate is guaranteed not to alter this argument).

```
:- mode see(+), assert(+).
```

`-` This argument is an output of the predicate. In every invocation of the predicate the argument position *will always be a variable* (as opposed to the `#` annotation below). This variable is unified with the value returned by the predicate. We note that Prolog does not enforce the requirement that output arguments should be variables; however, output unification is not very common in practice.

```
:- mode cputime(-).
```

[2] Any parallelisation directives (`parallel`) are simply ignored by the compiler, but do not result in syntax errors to enhance compatibility with various other earlier versions of PSB-Prolog.

[3] The most common uses of `mode` declarations in Prolog systems are to reduce the size of compiled code, or to speed up a predicate's execution.

`#` This argument is either:

- An output argument of the predicate for which a non-variable value may be supplied for this argument position. If such a value is supplied, the result in this position is unified with the supplied supplied value. The predicate fails if this unification fails. If a variable term is supplied, the predicate succeeds, and the output variable is unified with the return value.

```
:- mode '='(#,#).
```

- An input/output argument position of a predicate that has only side-effects (usually by further instantiating that argument). The `#` symbol is used to denote the $\pm$ symbol that cannot be entered from the keyboard.

`?` This argument does not fall into any of the above categories. Typical cases would be the following:

- An argument that can be used both as input and as output (but usually not with both uses at the same time).

```
:- mode functor(?,?,?).
```

- An input argument where the term supplied can be a variable (so that the argument cannot be annotated as `+`), or is instantiated to a term which itself contains uninstantiated variables, but the predicate is guaranteed *not* to bind any of these variables.

```
:- mode var(?), write(?).
```

We try to follow these mode annotation conventions throughout this manual.

Finally, we warn the user that `mode` declarations can be error-prone, and since errors in mode declarations do not show up while running the predicates interactively, unexpected behaviour may be witnessed in compiled code, optimised to take modes into account (currently not performed by XSB). However, despite this danger, `mode` annotations can be a good source of documentation, since they express the programmer's intention of data flow in the program.

Tabling Directives

Memoization is often necessary to ensure that programs terminate, and can be useful as an optimization strategy as well. 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

```
:- auto_table.
```

anywhere in the source file. `auto_table` declares predicates tabled so that the program will terminate.

To understand precisely how `auto_table` does this, it is necessary to mention a few properties of SLG. For 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. The `auto_table` compiler directive tables only those predicates of a module which appear to static analysis to contain an infinite loop, or which are called directly through `tnot/1`. It is perhaps more illuminating to demonstrate these conditions through an example rather than explaining them. For instance, in the program.

```
:- auto_table.

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

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

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

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

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

:- mode ap(+,+,-).
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 s/1 as a tabled predicate
% Compiling predicate r/1 as a tabled predicate
% Compiling predicate m/1 as a tabled predicate
% Compiling predicate mem/2 as a tabled predicate
```

Terminating conditions were detected for `ap1/3` and `ap/3`, but not for any of the other predicates.

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

Predicates can be explicitly declared as tabled as well, through the `table/1`. 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.

Another use of tabling is to filter out redundant solutions for efficiency rather than termination. In this case, suppose that the directive `edb/1` were used to indicate that certain predicates were likely to have a large number of clauses. Then the action of the declaration `:- suppl_table` in the program:

```
:- edb(r1/2).
:- edb(r2/2).
:- edb(r3/2).

:- suppl_table.

join(X,Z):- r1(X,X1),r2(X1,X2),r3(X2,Z).
```

would be to table `join/2`. The `suppl_table` directive is the XSB analogue to the deductive database optimization, *supplementary magic templates* [4]. `suppl_table/0` is shorthand for `suppl_table(2)` which tables all predicates containing clauses with two or more `edb` facts or tabled predicates. By specifying `suppl_table(3)` for instance, only predicates containing clauses with three or more `edb` facts or tabled predicates would be tabled. This flexibility can prove useful for certain data-intensive applications.

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`$^{\text{th}}$ 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 predicates that are dynamically loaded, this directive can be used to specify indexing on more than one argument, or indexing on a combination of arguments (see its description on page 107). 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.
```

Unification Factoring

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 unification factoring. Unification Factoring 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 $b/0$. Unification Factoring 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 unification factoring:[4].

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

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

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

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

By default, higher-order predicates (more precisely, predicates named *apply* with arity greater than 1) are compiled with unification factoring enabled. It can be disabled using the `ti_off` directive. For all other predicates, unification factoring 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 unification factoring may have no effect when a predicate is well indexed to begin with. For example, unification factoring has no effect on the following program:

[4]Unification factoring was once called transformational indexing, hence the abbreviation `ti` in the compiler directives

```
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 3.8.2).

Other Directives

XSB has other 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.

`:- ldoption(`*Options*`).`
: This directive is only recognized in the header file (`.H` file) of a foreign module. See the chapter *Foreign Language Interface* in Volume 2 for its explanation.

`:- compiler_options(`*OptionsList*`).`
: Indicates that the compiler options in the list *OptionsList* should be used to compile this file. This must appear at the beginning of the file. Thes options will override any others, including those given in the compilation command. The options may be optionally prefixed with `+` or `-` to indicate that they should be set on or off. (No prefix indicates the option should be set on.)

3.8.5 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 2.2. 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.

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
```

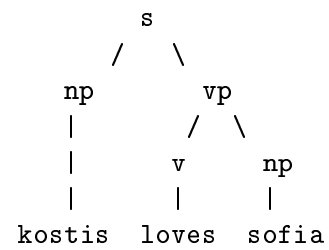
Note that a functor of arity 0 is represented as an atom.

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:



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 $t_1, t_2, \ldots, t_k$.

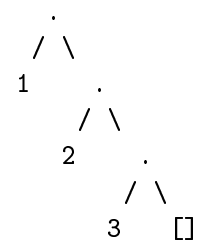
Sometimes it is convenient to write certain functors as *operators*. *Binary functors* (that is, functors that are applied to two arguments) may be declared as *infix operators*, and *unary functors* (that is, functors that are applied to one argument) may be declared as either *prefix or postfix operators*.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



which could be written using the standard syntax, as:

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

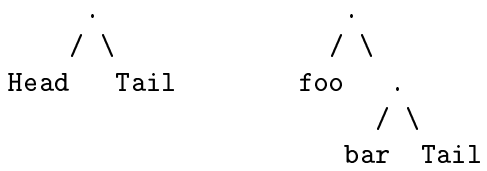
but which is normally written in a special list notation, as:

```
[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:



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 3.8.4) 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 3.8.4), 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 39 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`).

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 B.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 B.3. For example, the built-in operators `'+'` and `'-'` are as if they had been declared by the command:

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

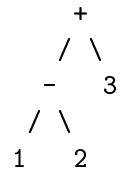
so that:

```
1-2+3
```

is valid syntax, and denotes the compound term:

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

or pictorially:



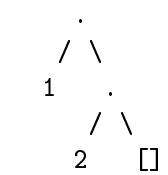
In XSB, the list functor `'.'/2` is one of the standard operators, that can be thought as declared by the command:

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

So, in XSB,

```
1.2.[]
```

represents the structure



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: A Tutorial Introduction

XSB has two ways of evaluating predicates. The default is to use Prolog-style evaluation, but by using various declarations a programmer can use also tabled resolution which allows for a different, more declarative programming style than Prolog. In this section we discuss the various aspects of tabling and how it is implemented in XSB. Our aim in this section is to provide a user with enough information to be able to program productively in XSB. It is best to read this tutorial with a copy of XSB handy, since much of the information is presented through a series of exercises.

For the theoretically inclined, XSB uses SLG resolution which can compute queries to non-floundering normal programs under the well-founded semantics [44], and is guaranteed to terminate when these programs have the *bounded term-depth property*. This tutorial covers only enough of the theory of tabling to explain how to program in XSB. For those interested, the web site contain papers covering in detail various aspects of tabling (often through the links for individuals involved in XSB). An overview of SLG resolution, and practical evaluation strategies for it are provided in [8, 41, 38, 19]. The engine of XSB, the SLG-WAM, is described in [35, 33, 18, 37, 7, 14] as it is implemented in Version 2.2 and its performance analyzed. Examples of large-scale applications that use tabling are overviewed in [42, 9, 12].

5.1 XSB as a Prolog System

Before describing how to program using tabling it is perhaps worthwhile reviewing some of the goals of XSB

1. To execute tabled predicates at the speed of compiled Prolog.
2. To ensure that the speed of compiled Prolog is not slowed significantly by adding the option of tabling.
3. To execute that the functionality of Prolog is not compromised by support for tabling.

4. To provide Prolog functionality in tabled predicates whenever it is semantically sensible to do so.

5. To provide standard predicates to manipulate tables taken as objects in themselves.

Goals 1 and 2 are addressed by XSBs engine, which in Version 2.2 is based on a memory-copying version of a virtual machine called the SLG-WAM. The overhead for SLD resolution using this machine is negligible. Thus when XSB is used simply as a Prolog system (i.e., no tabling is used), it is reasonably competitive with other Prolog implementations based on a WAM emulator written in C or assembly. For example, XSB Version 1.6 is about two to three times slower than Quintus 3.1.1 or emulated SICStus Prolog 3.1.

Goals 3, 4 and 5 have been nearly met, but there are a few instances in which interaction of tabling with a Prolog construct has been accomplished, or is perhaps impossibe. Accordingly we discuss these instances throughout this chapter. XSB is still under development however, so that future versions may support more transparent mixing of Prolog and tabled code (e.g. allowing tabled predicates in the scope of $'\backslash +' /1$) or adding Prolog functionality to tabled predicates (e.g. allowing non-ground negation in `tnot/1`).

5.2 Tabling in Definite Programs

Definite programs, also called Horn Clause Programs, are those programs without negation — In XSB, this means without the `\+/1`, `fail_if/1`, `not/1` or `tnot/1` operators. Consider the Prolog program

```
path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

together with the query `?- path(1,Y)`. This program has a simple, declarative meaning: there is a path from `X` to `Y` if there is a path from `X` to some node `Z` and there is a path from `Z` to `Y`, or if there is a direct path from `X` to `Y`. Prolog, however enters into an infinite loop when computing an answer to this query. The inability of Prolog to answer such queries, which arise frequently, comprises one of its major limitations as an implementation of logic.

A number of approaches have been developed to address this problem by reusing partial answers to the query `path(1,Y)` [17, 43, 3, 45, 46]. The ideas behind these algorithms can be described in the following manner. First, the implementation keeps track of all calls to tabled predicates, (or *tabled subgoals* such as `path(1,Y)` in the above example. Whenever a new tabled subgoal S is called, a check is first made to see whether S is in the table. If so, S is resolved against answers in the table; if not S is entered into the table and the subgoal is resolved against program clauses, as in Prolog. Answers are handled in the same way. When an answer to a tabled subgoal S is derived a check is made against the table for S to see if the answer is there. If the answer isn't in the table for S, the answer is added and scheduled to be returned to all instances where S has been called; if the answer is already in the table, the evaluation simply fails and backtracks to generate more answers.

Predicates can be declared tabled in a variety of ways. A common form is the compiler directive

$$\texttt{:-table } p_1/n_1, \ldots, p_k/n_k.$$

where p_i is a predicate symbol and n_i is an integer representing the arity of p_i. This directive can be added to the file containing the predicate to be tabled and then to compile the file.

Exercise 5.2.1 *Unless otherwise noted, the file table_examples.P in the directory* `$XSB_DIR/examples` *contains all code for the running examples in this section. Consult the file into XSB and type the query*

```
?- path(1,Y).
```

and continue hitting semi-colons until you have exhausted all answers. Type the query again. Can you guess why the order of answers is different? Now type

```
?- abolish_all_tables.
```

and retry the path query.

Exercise 5.2.2 *If you are curious, try rewriting the path query as it would be written in Prolog. Will it now terminate for the provided* `edge/2` *relation? (Remember, in XSB you can always hit* `<ctrl>-C` *if you go into an infinite loop).*

The return of answers in tabling aids in filtering out redundant computations – indeed it is this property which makes tabling terminate for many classes of programs. The *same generation* program furnishes a case of the usefulness of tabling for optimizing a Prolog program.

Exercise 5.2.3 *If you are* still *curious, load in the file* `cyl.P` *in the* `$XSB_DIR/examples` *directory using the command.*

```
?- load_dync(cyl.P).
```

and then type the query

```
?- same_generation(X,X),fail.
```

Now rewrite the `same_generation/2` *program so that it does not use tabling and retry the same query what happens? (Be patient — or use* `<ctrl>-C`*).*

The examples stress two differences between tabling and SLD resolution beyond termination properties. First, that each solution to tabled subgoal is returned only once — a property that is helpful not only for `path/2` but also for `same_generation/2` which terminates in Prolog. Second, because answers are sometimes obtained using program clauses and sometimes using answers, answers may be returned in an unaccustomed order.

In the language of tabling, the first instance of a tabled subgoal S is called a *generator* subgoal, and is expanded using program clauses as in SLD resolution (Prolog). Subsequent instances of S are referred to as *consuming* subgoals and are expanded using answers in the table for S instead of program clauses. Because *consuming* subgoals resolve against unique answers rather than repeatedly against program clauses, tabling will terminate whenever (1) a finite number of subgoals are encountered in query evaluation, (2) each of these subgoals have a finite number of answers. Indeed, it can be proven that for any program with the *bounded term depth property* (roughly, where all terms generated in a program have a maximum depth), SLG computation will terminate. These programs include the important class of *Datalog* programs.

Variant and Subsumptive Tabling The above description gives the general idea of how tabling affects definite programs but is imprecise on certain points. In XSB, a subgoal subgoals S_2 can use a table from S_1 if S_1 is a *variant* of S_2, that is, if S_1 and S_2 are the same up to variable renaming. Other tabling strategies may allow S_2 to use the table of S_1 if S_2 not more general than S_1, or S_2 is *subsumed by* S_1:

Example 5.2.1 *The terms* `p(f(Y),X,1)` *and* `p(f(Z),U,1)` *are variants, but* `p(f(Y),X,1)` *and* `p(f(Z),Z,1)` *are not. In fact, the former subsumes the latter.*

Just as a subsumption or variance relation can be used to decide when one subgoal can use the table of another, the two relations can be used to determine when an answer should be returned. In XSB's engine, a derived answer A will be considered new and returned to a subgoal S only if A is not a variant of some other previously derived answer for S. In Version 2.2 of XSB, subgoal subsumption is not supported: although work on an engine that includes subgoal subsumption is nearing completion. Answer subsumption, however, can be flexibly programmed as discussed in Section 5.4 [1].

Cuts and Tabling Tabling integrates well with most Prolog functionality, even for non-pure Prolog predicates. Meta-logical predicates like `var/1`, and 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.

The use of cuts with tabling is more problematic, as can be seen from the following exercise.

Exercise 5.2.4 *Consider the program*

```
:- table cut_p/1,cut_q/1,cut_r/0,cut_s/0.

cut_p(X):- cut_q(X),cut_r.
cut_r:- cut_s.
cut_s:- cut_q(_).
cut_q(1).       cut_q(2).

once(Term):- call(Term),!.
```

[1] We also note that the library subsumes contains routines for checking variance and subsumption.

What solutions are derived for the goal `?- cut_p(X)`*? Suppose that* `cut_p/1` *were rewritten as*

```
cut\_p1(X):- cut\_q1(X),once(cut\_r1).
```

How should this cut over a table affect the answers generated for `cut_p/1`*? What happens if you rewrite* `cut_p/1` *in this way and compile it in XSB?*

In Exercise 5.2.4, `cut_p(1)` and `cut_p(2)` should both be true. Thus, the cut in the `once(cut_r1)` literal in the revised may inadvertantly cut away solutions that are demanded by `cut_p/1`. Version 2.2 of XSB does not allow cuts over tabled predicates. XSB checks whether a tabled predicate statically lies in the scope of a cut at compile time. If so, the compilation is aborted[2]. At runtime, it also ensures that no incomplete tables are cut over whenever it executes a cut.

However, cuts are allowed *within* tabled predicates, subject (as always) to the restriction that the scope of a cut cannot include a call to a tabled predicate.

Example 5.2.2 *An example of using cuts in a tabled predicate is a tabled meta-interpreter.*

```
:- table demo/1.

demo(true).
demo((A,B)):-!,demo(A),demo(B).
demo(C):-call(C).
```

More elaborate tabled meta-interpreters can be extremely useful, for instance to implement various extensions of definite or normal programs.

In Version 2.2 of XSB a "cut" over tables occurs only 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 warning `"Removing incomplete tables..."` appear. Any complete tables will not be removed. They can be abolished by using one of XSB's predicates for abolishing tables.

Potential Pitfalls in Tabling 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, transforming a linear algorithm into a quadratic one.

Exercise 5.2.5 *If you need convincing that tabling can sometimes slow a query down, type the query:*

[2] A more sophisticated solution is proposed in [41].

```
?- genlist(1000,L),prolog_append(L,[a],Out).
```

and then type the query

```
?- genlist(1000,L),table_append(L,[a],Out).
```

`append/3` *is a particularly bad predicate to table. Type the query*

```
?- table_append(L,[a],Out).
```

(i.e. with no `genlist/2` *and backtrack through a few answers. Will* `table_append/3` *ever succeed for this predicate? Why not?*

Suppose DCG predicates (Section 8) are defined to be tabled. How is this similar to tabling append?

Another issue to be aware of when using tabling in XSB is tracing. XSB's tracer is a standard 4-port tracer, that interacts with the engine at each call, exit, redo, and failure of a predicate (see Chapter 7). When tabled predicates are traced, these events may occur in unexpected ways, as the following example shows.

Exercise 5.2.6 *Consider a tabled evaluation when the query* `?- a(0,X)` *is given to the following program*

```
:- table mut_ret_a/2, mut_ret_b/2.
mut_ret_a(X,Y):- mut_ret_d(X,Y).
mut_ret_a(X,Y):- mut_ret_b(X,Z),mut_ret_c(Z,Y).

mut_ret_b(X,Y):- mut_ret_c(X,Y).
mut_ret_b(X,Y):- mut_ret_a(X,Z),mut_ret_d(Z,Y).

mut_ret_c(2,2).      mut_ret_c(3,3).

mut_ret_d(0,1).      mut_ret_d(1,2).     mut_ret_d(2,3).
```

`mut_ret_a(0,1)` *can be derived immediately from the first clause of* `mut_ret_a/2`*. All other answers to the query depend on answers to the subgoal* `mut_ret_b(0,X)` *which arises in the evaluation of the second clause of* `mut_ret_a/2`*. Each answer to* `mut_ret_b(0,X)` *in turn depends on an answer to* `mut_ret_a(0,X)`*, so that the evaluation switches back and forth between deriving answers for* `mut_ret_a(0,X)` *and* `mut_ret_b(0,X)`*.*

Try tracing this evaluation, using creep and skip. Do you find the behavior intuitive or not?

Table Directives and Declarations Often it is tedious to decide which predicates must be tabled. To address this, XSB can automatically table predicates in files. The declaration `auto_table` chooses predicates to table to assist in termination, while `suppl_table` chooses predicates to table to optimize data-oriented queries. Both are explained in Section 3.8.4.

Exercise 5.2.7 *The reader may have noted that the command* `table` *was referred to as a directive, while* `auto_table` *and* `suppl_table` *were both referred to as declarations. The difference is that the user can execute a directive at the command line but not a compiler declaration. For instance, restart XSB and at the XSB prompt, type the directive*

```
?- table(dyn_path/2).
```

and

```
?- load_dyn(dyn_examples).
```

Try the queries to `path/2` *of the previous examples. Note that it is important to dynamically load* `dyn_examples.P` *— otherwise the code in the file will be compiled without knowledge of the tabling declaration.*

5.3 Stratified Normal Programs

Normal programs extend definite programs to include default negation, which posits a fact as false if all attempts to prove it fail. As shown in Example 1.0.1, which presented one of Russell's paradoxes as a logic program, the addition of default negation allows logic programs to express contradictions. As a result, some assertions, such as `shaves(barber,barber)` may be undefined, although other facts, such as `shaves(barber,mayor)` may be true. Formally, the meaning of normal programs may be given using the *well-founded semantics* and it is this semantics that XSB adopts for negation.

The Intuition behind Stratified Programs Before considering the full well-founded semantics, we discuss how XSB can be used to evaluate programs with *stratified negation*. Intuitively, a program uses stratified negation whenever there is no recursion through negation. Indeed, most programmers, most of the time, use stratified negation.

Exercise 5.3.1 *The program*

```
win(X):- move(X,Y),tnot(win(Y)).
```

is stratified when the `move/2` *relation is a binary tree. This can be seen by loading the file* `$XSB_DIR/examples/tree` *along with* `table_examples.P` *and typing the query*

```
?- win(1).
```

`win(1)` *calls* `win(2)` *through negation,* `win(2)` *calls* `win(4)` *through negation, and so on, but no subgoal ever calls itself recursively through negation.*

The previous example of `win/1` over a binary tree is a simple instance of a stratified program, but it does not even require tabling. A more complex example is presented below.

Exercise 5.3.2 *Consider the query* `?- lrd_s` *to the following program*

```
lrd_p:- lrd_q,tnot(lrd_r),tnot(lrd_s).
lrd_q:- lrd_r,tnot(lrd_p).
lrd_r:- lrd_p,tnot(lrd_q).
lrd_s:- tnot(lrd_p),tnot(lrd_q),tnot(lrd_r).
```

Should `lrd_s` *be true or false? Try it in XSB. Using the intuitive definition of "stratified" as not using recursion through negation, is this program stratified? Would the program still be stratified if the order of the literals in the body of clauses for* `lrd_p`, `lrd_q`, *or* `lrd_r` *were changed?*

The rules for `p`, `q` and `r` are involved in a positive loop, and no answers are ever produced. Each of these atoms can be failed, thereby proving `s`. Exercise 5.3.2 thus illustrates an instance of how tabling differs from Prolog in executing stratified programs since Prolog would not fail finitely for this program.

Completely Evaluated Subgoals Knowing when a subgoal is completely evaluated can be useful when programming with tabling. Simply put, a subgoal S is *completely evaluated* if an evaluation can produce no more answers for S. The computational strategy of XSB makes great use of complete evaluation so that understanding this concept and its implications can be of great help to a programmer.

Consider a simple approach to incorporating negation into tabling. Each time a negative goal is called, a separate table is opened for the negative call. This evaluation of the call is carried on to termination. If the evaluation terminates, its answers if any, are used to determine the success of failure of the calling goal. This general mechanism underlies early formulations for tabling stratified programs [21, 40]. Of course this method may not be efficient. Every time a new negative goal is called, a new table must be started, and run to termination. We would like to use information already derived from the computation to answer a new query, if at all possible — just as with definite programs.

XSB addresses this problem by keeping track of the *state* of each subgoal in the table. A call can have a state of *complete*, *incomplete* or *not_yet_called*. Calls that do have table entries may be either *complete* or *incomplete*. A subgoal in a table is marked *complete* only after it is determined to be completely evaluated; otherwise the subgoal is *incomplete*. If a tabled subgoal is not present in the table, it is termed *not_yet_called*. XSB contains predicates that allow a user to examine the state of a given table (Section 6.12).

Using these concepts, we can overview how tabled negation is evaluated for stratified programs. If a literal `tnot(S)` is called, where `S` is a tabled subgoal, the evaluation checks the state of `S`. If `S` is *complete* the engine simply determines whether the table contains an answer for `S`. Otherwise the engine *suspends* the computation path leading to `tnot(S)` until `S` is completed (and calls `S` if necessary). Whenever a suspended subgoal `tnot(S)` is completed with no answers, the engine resumes the evaluation at the point where it had been suspended. We note that because of this behavior, tracing programs that heavily use negation may produce behavior unexpected by the user.

`tnot/1` vs. `'\+'/1` Subject to some semantic restrictions, an XSB programmer can intermix the use of tabled negation (`tnot/1`) with Prolog's negation (`'\+'/1`, or equivalently `fail_if/1` or `not/1`). These restrictions are discussed in detail below — for now we focus on differences in behavior or these two predicates in stratified programs. Recall that `'\+'(S)` calls S and if S has a solution, Prolog , executes a cut over the subtree created by `'\+'(S)`, and fails. `tnot/1` on the other hand, does not execute a cut, so that all subgoals in the computation path begun by the negative call will be completely evaluated. The major reason for not executing the cut is to insure that XSB evaluates ground queries to Datalog programs with negation with polynomial data complexity. As seen in Section 5.2, this property cannot be preserved if negation "cuts" over tables.

There are other small differences between `tnot/1` and `'\+'/1`illustrated in the following exercise.

Exercise 5.3.3 *In general, making a call to non-ground negative subgoal in Prolog may be unsound (cf. [29]), but the following program illustrates a case in which non-ground negation is sound.*

```
ngr_p:- \+ ngr_p(_).
ngr_p(a).
```

Its tabled analog is

```
:- table ngr_tp/1.
ngr_tp:- tnot(ngr_tp(_)).
ngr_tp(a).
```

Version 2.2 of XSB will flounder on the call to `ngr_tp`*, but not on the call to* `ngr_p/0`*.*

The description of `tnot/1` *in Section 6.4 describes other small differences between* `'\+'/1` *and* `tnot/1` *as implemented in XSB.*

Before leaving the subject of stratification, we note that the concepts of stratification also underly XSB's evaluation of tabled findall: `tfindall/3`. Here, the idea is that a program is stratified if it contains no loop through tabled findall (See the description of predicate `tfindall/3` on page 86).

5.3.1 Non-stratified Programs

As discussed above, in stratified programs, facts are either true or false, while in non-stratified programs facts may also be undefined. XSB represents undefined facts as *conditional answers.*

Conditional Answers

Exercise 5.3.4 *Consider the behavior of the* `win/1` *predicate from Exercise 5.3.1.*

```
win(X):- move(X,Y),tnot(win(Y)).
```

when the when the `move/2` *relation is a cycle. Load the file* `$XSB_DIR/examplescycle1k.P` *into XSB and again type the query* `?- win(1)`. *Does the query succeed? Try* `tnot(win(1))`.

Now query the table with the standard XSB predicate `get_residual/2`, *e.g.* `?- get_residual(win(1),X)`. *Can you guess what is happening with this non-stratified program?*

The predicate `get_residual/2` (Section 6.12) unifies its first argument with a tabled subgoal and its second argument with the (possibly empty) delay list of that subgoal. The truth of the subgoal is taken to be conditional on the truth of the elements in the delay list. Thus `win(1)` is conditional on `tnot(win(2))`, `win(2)` in `tnot(win(3))` and so on until `win(1023)` which is conditional on `win(1)`.

From the perspective of the well-founded semantics, `win(1)` is undefined. Informally, true answers in the well-founded semantics are those that have a (tabled) derivation. False answers are those for which all possible derivations fail — either finitely as in Prolog or by failing positive loops. `win(1)` fits in neither of these cases – there is no proof of `win(1)`, yet it does not fail in the sense given above and is thus undefined.

However this explanation does not account for why undefined answers should be represented as conditional answers, or why a query with a conditional answer *and* its negation should both succeed. These features arise from the proof strategy of XSB, which we now examine in more detail.

Exercise 5.3.5 *Consider the program*

```
:- table simpl_p/1,simpl_r/0,simpl_s/0.
simpl_p(X):- tnot(simpl_s).

simpl_s:- tnot(simpl_r).
simpl_s:- simpl_p(X).

simpl_r:- tnot(simpl_s),simpl_r.
```

Is `simpl_p(X)` *true for any* `X`*? Try the query* `?- simpl_p(X)` *– be sure to backtrack through all possible answers. Now try the query again. What could possibly account for this behavior?*

At this point, it is worthwhile to examine closely the evaluation of the program in Exercise 5.3.5. The query `simpl_p(X)` calls `simpl_s` and `simpl_r` and executes the portion of the program shown below in bold:

simpl_p(X):- tnot(simpl_s).

simpl_s:- tnot(simpl_r).
simpl_s:- simpl_p(X).

simpl_r:- tnot(simpl_s),*simpl_r.*

Based on evaluating only the bold literals, the three atoms are all undefined since they are neither proved true, nor fail. However if the evaluation could only look at the literal in italics, *simpl_r*, it would discover that *simpl_r* is involved in a positive loop and, since there is only one clause for *simpl_r*, the evaluation could conclude that the atom was false. This is exactly what XSB does, `delays` the evaluation of `tnot(simpl_s)` in the clause for `simpl_r` and looks ahead to the next literal in the body of that clause. This action of looking ahead of a negative literal is called *delaying*. A delayed literal is moved into the *delay list* of a current path of computation. Whenever an answer is derived, the delay list of the current path of computation is copied into the table. If the delay list is empty, the answer is unconditional; otherwise it is conditional. Of course, for definite programs any answers will be unconditional — we therefore omited delay lists when discussing such programs.

In the above program, delaying occurs for the negative literals in clause for `simpl_p(X)`, `simpl_s`, and `simpl_r`. In the first two cases, conditional answers can be derived, while in the third, `simpl_r` will fail as mentioned above. Delayed literals eventually become evaluated through *simplification*. Consider an answer of the form

```
simpl_p(X):- tnot(simpl_s)|
```

where the `|` is used to represent the end of the delay list. If, after the answer is copied into the table, `simpl_s` turns out to be false, (after being initially delayed), the answer can become unconditional. If `simpl_s` turns out to be true, the answer should be removed, it is false.

In fact, it is this last case that occurs in Exercise 5.3.5. The answer

```
simpl_p(X):- tnot(simpl_s)|
```

is derived, and returned to the user (XSB does not currently print out the delay list). The answr is then removed through simplification so that when the query is re-executed, the answer does not appear.

We will examine in detail how to alter the XSB interface so that evaluation of the well-founded semantics need not be confusing. It is worthwhile to note that the behavior just described is uncommon.

Version 2.2 of XSB handles dynamically stratified programs through delaying negative literals when it becomes necessary to look to their right in a clause, and then simplifying away the delayed literals when and if their truth value becomes known. However, to ensure efficiency, literals are never delayed unless the engine determines them to not to be stratified under the LRD-stratified evaluation method.

When Conditional Answers are Needed A good Prolog programmer uses the order of literals in the body of a clause to make her program more efficient. However, as seen in the previous section, delaying can break the order that literals are evaluated within the body of a clause. It then becomes natural to ask if any guarantees can be made that XSB is not delaying literals unnecessarily.

Such a guarantee can in fact be made, using the concept of *dynamic stratification* [31]. Without going into the formalism of dynamic stratification, we note that a program is dynamically stratified

if and only if it has a two-valued model. It is also known that computation of queries to dynamically stratified programs is not possible under any fixed strategy for selecting literals within the body of a clause. In other words, some mechanism for breaking the fixed-order literal selection strategy must be used, such as delaying.

However, by redefining dynamic stratification to use an arbitrary fixed-order literal selection strategy (such as the left-to-right strategy of Prolog), a new kind of stratification is characterized, called *Left-to-Right Dynamic Stratification*, or *LRD-stratification*. LRD-stratified is not as powerful as dynamic stratification, but is more powerful than other fixed-order stratification methods, and it can be shown that for ground programs, XSB delays only when programs are not LRD-stratified. In the language of [38] XSB is *delay minimal*.

Programming in the Well-founded Semantics XSB delays literals for non-LRD-stratified programs and later simplifies them away. But how can the programmer determine when all simplification has been done? One method is to use local evaluation, discussed below in Section 5.4.1. A second method is to make a top-level call for a predicate, `p` as follows:

```
?- p,fail ; p.
```

when the second `p` in this query is called, all simplification on `p` will have been performed. However, this query will succeed if `p` is true *or* undefined.

Exercise 5.3.6 *Write a predicate* `wfs_call(+Tpred,?Val)` *such that if* `Tpred` *is a ground call to a tabled predicate,* `wfs_call(+Tpred,?Val)` *calls* `Tpred` *and unifies* `Val` *with the truth value of* `Tpred` *under the well-founded semantics.* Hint: use `get_residual/2`.

How would you modify `wfs_call(?Tpred,?Val)` *so that it properly handled cases in which* `Tpred` *is non-ground.*

Trouble in Paradise: Answer Completion The engine for XSB performs both Prolog style and answer resolution, along with delay and simplification. What it does not do is to perform an operation called *answer completion* which is needed in certain (pathological?) programs.

Exercise 5.3.7 *Consider the following program:*

```
:- table p/1,r/0,s/0.
ac_p(X):- ac_p(X).
ac_p(X):- tnot(ac_s).

ac_s:- tnot(ac_r).
ac_s:- ac_p(X).

ac_r:- tnot(ac_s),ac_r.
```

Using either the predicate from Exercise 5.3.6 or some other method, determine the truth value of `ac_p(X)`. *What should the value be? (hint: what is the value of* `ac_s/1`*?).*

For certain programs, XSB will delay a literal (such as `ac_p(X)` that it will not be able to later simplify away. In such a case, an operation, called *answer completion* is needed to remove the clause

```
p(X):- p(X)|
```

Without answer completion, XSB may consider some answers to be undefined rather than false. It is thus is sound, but not complete for terminating programs to the well-founded semantics. Answer completion is not available for Version 2.2 of XSB, as it is expensive and the need for answer completion arises rarely in practice. However answer completion will be included at some level in future versions of XSB.

5.3.2 On Beyond Zebra: Implementing Other Semantics for Non-stratified Programs

The Well-founded semantics is not the only semantics for non-stratified programs. XSB can be used to (help) implement other semantics that lie in one of two classes. 1) Semantics that extend the well-founded semantics to include new program constructs; or 2) semantics that contain the well-founded partial model as a submodel.

An example of a semantics of class 1) is (WFSX) [2], which adds explicit (or provable) negation to the default negation used by the Well-founded semantics. The addition of explicit negation in WFSX, can be useful for modeling problems in domains such as diagnosis and hierarchical reasoning, or domains that require updates [26], as logic programs. WFSX is embeddable into the well-founded semantics; and this embedding gives rise to an XSB meta-interpreter, or, more efficiently, to the preprocessor described in Section *Extended Logic Programs* in Volume 2. See [42] for an overview of the process of implementing extensions of the well-founded semantics.

An example of a semantics of class 2) is the stable model semantics. Every stable model of a program contains the well-founded partial model as a submodel. As a result, the XSB can be used to evaluate stable model semantics through the *residual program*, to which we now turn.

The Residual Program Given a program P and query Q, the residual program for Q and P consists of all (conditional and unconditional) answers created in the complete evaluation of Q.

Exercise 5.3.8 *Consider the following program.*

```
:- table ppgte_p/0,ppgte_q/0,ppgte_r/0,ppgte_s/0,
         ppgte_t/0,ppgte_u/0,ppgte_v/0.
ppgte_p:- ppgte_q.          ppgte_p:- ppgte_r.

ppgte_q:- ppgte_s.          ppgte_r:- ppgte_u.
ppgte_q:- ppgte_t.          ppgte_r:- ppgte_v.

ppgte_s:- ppgte_w.          ppgte_u:- undefined.
```

```
ppgte_t:- ppgte_x.          ppgte_v:- undefined.

ppgte_w:- ppgte(1).         ppgte_x:- ppgte(0).
ppgte_w:- undefined.        ppgte_x:- undefined.

ppgte(0).

:- table undefined/0.
undefined:- tnot(undefined).
```

Write a routine that uses `get_residual/2` *to print out the residual program for the query* `?- ppgte_p,fail`. *Try altering the tabling declarations, in particular by making* `ppgte_q/0`, `ppgte_r/0`, `ppgte_s/0` *and* `ppgte_t/0` *non-tabled. What effect does altering the tabling declarations have on the residual program?*

When XSB returns a conditional answer to a literal L, it does not propagate the delay list of the conditional answer, but rather delays L itself, even if L does not occur in a negative loop. This has the advantage of ensuring that delayed literals are not propagated exponentially through conditional answers.

Stable Models Stable models are one of the most popular semantics for non-stratified programs. The intuition behind the stable model semantics for a ground program P can be seen as follows. Each negative literal $notL$ in P is treated as a special kind of atom called an *assumption*. To compute the stable model, a guess is made about whether each assumption is true or false, creating an assumption set, A. Once an assumption set is given, negative literals do not need to be evaluated as in the well-founded semantics; rather an evaluation treats a negative literal as an atom that succeeds or fails depending on whether it is true or false in A.

Example 5.3.1 *Consider the simple, non-stratified program*

```
writes_manual(terry)-¬writes_manual(kostis),has_time(terry).
writes_manual(kostis)-¬writes_manual(terry),has_time(kostis).
has_time(terry).
has_time(kostis).
```

there are two stable models of this program: in one `writes_manual(terry)` *is true, and in another* `writes_manual(kostis)` *is true. In the Well-Founded model, neither of these literals is true. The residual program for the above program is*

```
writes_manual(terry)-¬writes_manual(kostis).
writes_manual(kostis)-¬writes_manual(terry).
has_time(terry).
has_time(kostis).
```

Computing stable models is an intractable problem, meaning that any algorithm to evaluate stable models may have to fall back on generating possible assumption sets, in pathological cases. For a ground program, if it is ensured that residual clauses are produced for *all* atoms, using the residual program may bring a performance gain since the search space of algorithms to compute stable models will be correspondingly reduced. In fact, by using XSB in conjunction with a Stable Model generator, Smodels [30], an efficient system has been devised for model checking of concurrent systems that is 10-20 times faster than competing systems [28].

5.4 Tabled Aggregation

The following shortest path predicate is a modification of the `path/2` predicate of Section 5.2:

```
:- table path/3.
path(X,Y,C) :- path(X,Z,C1), edge(Z,Y,C2), C is C1 + C2.
path(X,Y,C) :- edge(X,Y,C).
```

Exercise 5.4.1 `path/3` *has a simple declarative meaning: it computes the path between two vertices of a graph along with the cost of the path. Since* `path/3` *is tabled would you expect it to terminate? Try the query* `?- path(1,5,X)` *over the graph provided in the file* `table_examples.P`.

If we could use tabling to compute the path with least cost, or the shortest path, the program would not only omit extraneous information, but it would also terminate. Recall that for simple horn programs, variant-based tabling ensures termination by only returning a given answer A once, and failing on subsequent derivations of A. If this strategy could be extended so that the engine only returned a new answer if it was minimal, termination could be ensured. The XSB predicate, `filterReduce(?Pred,+Binary_operator,+Identity,Value)`, does just this.

Exercise 5.4.2 *The use of* `filterReduce/4` *can be seen most easily through an example such as the following, (which uses a closely related predicate* `filterReduce1/4`*).*

```
shorter_path(X,Y,C) :- filterReduce1(sp(X,Y),min,infinity,C).

sp(X,Y,C) :- shorter_path(X,Z,C1),
             edge(Z,Y,C2),C is C1 + C2.
sp(X,Y,C) :- edge(X,Y,C).

min(X,Y,Y):- \+ number(X),!.
min(X,Y,X):- \+ number(Y),!.
min(One,Two,Min):- One > Two -> Min = Two ; Min = One.
```

Note that the library predicate `filterReduce1/4` *is tabled, so that neither* `sp/3` *nor* `shorter_path/3` *need be tabled. Now try the query* `shorter_path(1,5,C)`.

`filterReduce1((?Pred,+Binary_operator,+Identity,Value)`, forms a new predicate out of `Pred` and `Value` to get a new predicate to call. `Binary_Operator` must define a binary function in

which the first two arguments determine the third. `Id` must be the identity of `Binary_operator`. `Value` becomes the result of applying `Op` to all the elements in the table that are variants of `Pred`. In our case, when a new answer `sp(X,Y,C)` is derived within `filterReduce1/4`, the later predicate returns only when `C` is a shorter path for `X` and `Y` than any so far derived.

While `shorter_path/4` terminates, it returns non-optimal solutions, and these solutions can in principle be costly — [18] cites a case in which the shorter path program, which should be less than cubic in the number of vertices in a graph, has exponential complexity because of the non-optimal solutions that are returned. Fortunately, this has an easy solution.

Exercise 5.4.3 *The actual* `shortest_path` *program has the following definition.*

```
filterReduce(Call,Op,Id,Res) :- filterReduce1(Call,Op,Id,Res), fail.
filterReduce(Call,Op,Id,Res) :- filterReduce1(Call,Op,Id,Res).

shortest_path(X,Y,C) :- filterReduce(sp(X,Y),min,infinity,C).

sp(X,Y,C) :- shortest_path(X,Z,C1),
             edge(Z,Y,C2),C is C1 + C2.
sp(X,Y,C) :- edge(X,Y,C).

min(X,Y,Y):- \+ number(X),!.
min(X,Y,X):- \+ number(Y),!.
min(One,Two,Min):- One > Two -> Min = Two ; Min = One.
```

Once again try the query `shortest_path(1,5,C)`.

By simply failing out of `filterReduce1/4` and then rereading the maximal value from the table, an efficient `shortest_path` algorithm is derived, whose complexity is roughly cubic in the number or vertices of the graph. This solution is not general for all predicates, but does work for deriving the shortest path. A more general solution is provided in Section 5.4.1.

`filterReduce/4` is an extremely useful predicate. It can write database aggregation functions, such as min, max, count, sum, and average. However, it can also be used to implement paraconsistent and quantitative reasoning through Generalized Annotated Programs [23], as detailed in the section on GAPs in Volume 2 of this manual.

Several predicates perform tabled aggregation besides `filterReduce/4`. One of these is the predicate `filterPO1(?Pred,?Preference_structure,+Partial_order)`. Analoguosly to `filterReduce1/4` if `Pred` is an n-ary predicate, `filterPO/4` forms a (n+1)-ary predicate `Pred1` whose last argument is `Preference_structure` and whose functor and all other arguments are determined by `Pred`. `filterPO(?Pred,?Preference_structure,+Partial_order)`, then calls `Pred1` and for each return of `Pred1` fails if there is some answer already in the table for `filterPO1/4` such that the first n arguments of `Pred` in the tabled answer unify with the first n arguments of `Pred` in the return and whose preference structure (last argument) is preferred to that of the return. A case study in the use of `filterPO/4` to construct preference logic grammars can be found in [10].

5.4.1 Local Evaluation

For the shortest path example, simply failing until a minimal answer was derived and then returning that solution was an effective technique for computing the shortest path. However, this approach will not always work. As we have seen in Exercise 5.2.6, programs can consist of sets of mutually recursive predicates and in principle these sets can be arbitrarily large. If these computations are to use tabled aggregation, the approach taken by `filterReduce/4` will not suffice. To see this, we make the notion of mutual recursion more precise. A tabled computation can be viewed as a directed graph, in which there is a link from one non-completed tabled predicate $P1$ to a non-completed tabled predicate $P2$ if $P2$ (or $tnot(P2)$) is called by $P1$. Of course, this graph constantly changes through an evaluation as resolution proceeds, subgoals are completed, and so on. Any directed graph can be uniquely partitioned into a set of maximal *strongly connected components* or SCCs, and these sets correspond to sets of mutually recursive predicates. The SCCs then, are reminiscent of the LRD-stratifiedstratification discussed in Section 5.3.1, except that both positive and negative links are counted as dependencies. From this view, to optimally compute tabled aggregation, non-optimal answers from a given subgoal S must be returned within the SCC of S, but not outside the SCC. This action is performed by *Local Scheduling*.

It is illustrative to compare local scheduling to *Batched Scheduling* the default scheduling of XSB. Batched scheduling returns answers as they are derived, and resembles Prolog's tuple at a time scheduling. Local scheduling was shown to be quite efficient in terms of time and space in [18], and is the fastest scheduling strategy that we know of for computing a sequence of answers. The same paper also introduced Local Scheduling, which computes all answers for each SCC and return only the best answer (or answers) out of the SCC, when the SCC is completely evaluated — exactly the thing for tabled aggregation.

XSB can be configured to use local scheduling via the configuration option `--enable-local-scheduling` and remaking XSB. This will not affect the default version of XSB, which will also remain available.

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).

`open(+File,+Mode,-Stream)`

`open/1` creates a stream for the file designated in *File*, and binds *Stream* to a structure representing that stream. *Mode* can be one of either `read` to create an input stream or `write` or `append` to create an output stream. If the mode is `write`, the contents of *File* are removed and *File* becomes a record of the output stream. If the mode is `append` the output stream is appended to the contents of *File*.

Exceptions (`read` mode)

`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.

Exceptions (`write` mode)

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

`instantiation_error` `F` is uninstantiated.

`close(+Stream)`
`close/1` closes the stream *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.

`nl(+Stream)`
A new line character is sent to the designated 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

Compatibility 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.

Compatibility 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 or designated 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.

`read(+Stream, ?Term)`

`read/2` has the same behavior as `read/1` but the input stream is explicitly designated using the first argument.

`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 explicitly 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 the section *Library Utilities* in Volume 2.

`write(+Stream, ?Term)`

`write/2` has the same behavior as `write/1` but the output stream is explicitly designated using the first argument.

`writeln(?Term)`

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

`writeln(+Stream,?Term)`

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

`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.

`write_prolog(+Stream,?Term)` HiLog

`write_prolog/2` has the same behavior as `write_prolog/1` but the output stream is explicitly designated using the first argument.

`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.

`writeq(+Stream, ?Term)`

`writeq/2` has the same behavior as `writeq/1` but the output stream is explicitly designated using the first argument.

`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,[]))`).

`read_canonical(-Term)` basics

Reads a term that is in canonical format from the current input stream and returns it in

`Term`. On end-of-file, it returns the atom `end_of_file`. If it encounters an error, it prints an error message on stderr and returns the atom `read_canonical_error`. This is significantly faster than `read/1`, but requires the input to be in canonical form.

`write_canonical(+Stream, ?Term)`
`write_canonical/2` has the same behavior as `write_canonical/1` but the output stream is explicitly designated using the first argument.

`print(?Term)`
This predicate is intended to provide a handle for user-defined pretty-printing. Currently it is defined as *write/1*.

6.2 Special I/O

`fmt_read(+Format, types(+T1,+T2,...), args(-A1,-A2,...), -RetCode)`
This predicate implements C-style formatted input. It reads the current input according to the `Format` string. `Format` has the same syntax as the input format in C. The term `types(...)` lists the types of the arguments; they must match the types specified in `Format`. Here, 1 means string, 2 means integer, and 3 means float. The term `args()` specifies the variables for the input. `RetCode` specifies the return code: 0 – ok; -1 – end of file.

`read_line(-Line, -Status)`
Reads the next line from the current input and puts it in `Line`. If the line is larger than the available buffer, then `Status` is 0. If the line was read in full, up to and including the newline character, then `Status` is 1.

`fmt_write(+Format, args(-A1,-A2,...))`
Similar to formatted write in C. The semantics of the arguments is the same as for `fmt_read/4`.

`fmt_write_string(-String, +Format, args(-A1,-A2,...))`
Like `fmt_write/2`, but the output string is placed in `String`.

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 5.1.

`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 according to the well-founded semantics. `P` must be a tabled predicate, For a detailed description of the actions of

tabled negation for in XSB Version 2.2 see [35, 37]. Chapter 5 contains further discussion of the functionality of `tnot/1`.

Exceptions:

`instantiation_error P` is not ground (floundering occurs).

`type_error P` is not a callable term.

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

`sk_not(+P)` Tabling

Same as `tnot/1` but does not check for floundering. Thus it permits variables in its subgoal argument and gives the semantics similar to Prolog's when there are variables. (This replaces the `'t not'/1` predicate of earlier XSB versions whose implementation and semantics were dubious.)

`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 compatibility 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
```

`is_charlist(+X)`
Succeeds if `X` is a Prolog string, *i.e.*, a list of characters. Examples:

```
| ?- is_charlist("abc").
yes
| ?- is_charlist(abc).
no
```

`is_charlist(+X,-Size)`
Like above, but also returns the length of that string in the second argument, which must be a variable.

`is_most_general_term(?X)`
Succeeds if `X` is compound term with all distinct variables as arguments, or if `X` is an atom. (It fails if X is a cons node.)

```
| ?- is_most_general_term(f(_,_,_,_)).
yes
| ?- is_most_general_term(abc).
yes
| ?- is_most_general_term(f(X,Y,Z,X)).
no
| ?- is_most_general_term(f(X,Y,Z,a)).
no
| ?- is_most_general_term([_|_]).
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 Prolog's `functor/3`, while the second is the extension of `functor/3` to handle HiLog terms. Like the Prolog's `functor/3` predicate, `hilog_functor/3` can be used in either of 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 $\texttt{Index}^{th}$ 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 conversion 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_codes/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. It is advisable that new programs use the predicates `atom_codes/2` and `number_codes/2` described below.

In Version 2.2 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_codes(?Atom, ?CharCodeList)`

The standard predicate `atom_codes/2` performs the conversion between an atom and its character list representation. If `Atom` 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_codes/2` are uninstantiated or `CharList` is not a proper list of ASCII characters, `atom_codes/2` aborts, and an error message will be sent to the standard error stream.

Examples:

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

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

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

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

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

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

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

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

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

| ?- atom_codes(X, [52,300,49]).
! Range error: in argument 2 of atom_codes/2
! ASCII code expected, but 300 found
! Aborting...
```

Exceptions:

`instantiation_error` Both arguments are uninstantiated, or argument 2 is not a proper list, or it contains a variable.

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

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

`atom_chars(?Number, ?CharAtomList)`

Like `atom_codes`, but the list returned (or input) is a list of characters *as atoms* rather than ASCII codes. For instance, `atom_chars(abc,X)` binds `X` to the list `[a,b,c]` instead of `[97,98,99]`.

`number_codes(?Number, ?CharCodeList)`

The standard predicate `number_codes/2` performs the conversion between a number and its character list representation. If `Number` is supplied (and is a number), `CharList` is unified with a list of ASCII codes comprising the printed representation of that `Number`. If on the other hand `Number` is a variable, then `CharList` must be a proper list of ASCII character codes that corresponds to the correct syntax of a number (either integer or float) In that case, `Number` is instantiated to that number, otherwise `number_codes/2` will simply fail.

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

Examples:

```
| ?- number_codes(123, L).
```

```
L = [49,50,51];

| ?- number_codes(N, [49,50,51]), integer(N).
N = 123

| ?- number_codes(31.4e+10, L).
L = [51,46,49,51,57,57,57,55,69,43,49,48]

| ?- number_codes(N, "314e+8").
N = 3.14e+10

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

Exceptions:

`instantiation_error` Both arguments are uninstantiated, or argument 2 is not a proper list, or it contains a variable.

`type_error Number` is not a variable or a number.

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

`number_chars(?Number, ?CharAtomList)`

Like `number_codes`, but the list returned (or input) is a list of characters *as atoms* rather than ASCII codes. For instance, `number_chars(123,X)` binds `X` to the list `['1','2','3']` instead of `[49,50,51]`.

`number_digits(?Number, ?DigitList)`

Like `number_chars`, but the list returned (or input) is a list of digits *as numbers* rather than ASCII codes (for floats, the atom '.', '+' or '-', and 'e' will also be present in the list). For instance, `number_digits(123,X)` binds `X` to the list `[1,2,3]` instead of `['1','2','3']`, and `number_digits(123.45,X)` binds `X` to `[1,.,2,3,4,5,0,0,e,+,0,2]`.

6.6 All Solutions and Aggregate Predicates

Often there are many solutions to a problem and it is necessary somehow to compare these solutions with one another. The most general way of doing this is to collect all the solutions into a list, which may then be processed in any way desired. So XSB provides several builtin all-solutions predicates which collect solutions into lists. Sometimes however, one wants simply to perform some aggregate operation over the set of solutions, for example to find the maximum or minimum of the set of solutions. XSB uses tabling and HiLog to provide a general and powerful aggregation facility through the use of two new builtins.

`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).

Exceptions: Same as predicate `call/1` (see Section 6.8).

`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.

Exceptions: Same as predicate `call/1` (see Section 6.8).

`findall(?X, +Goal, ?List)`

Similar to predicate `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. This makes `findall/3` deterministic (non-backtrackable). Unlike `setof/3` and `bagof/3`, if `Goal` is unsatisfiable, `findall/3` succeeds binding `List` to the empty list.

Exceptions: Same as predicate `call/1` (see Section 6.8).

`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/3`, the `Goal` must be a call to 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 Chapter 5 for further discussion of `tfindall/3`. Like `findall/3`, if `Goal` is unsatisfiable, `tfindall/3` succeeds binding `List` to the empty list.

Some of the differences between predicates `findall/3` and `tfindall/3` can be seen from the following example:

```
| ?- [user].
[Compiling user]
:- table p/1.
p(a).
p(b).
[user compiled, cpu time used: 0.639 seconds]
[user loaded]

yes
| ?- p(X), findall(Y, p(Y), L).

X = a
Y = _922928
```

```
L = [a];

X = b
Y = _922820
L = [a,b];

no
| ?- abolish_all_tables.

yes
| ?- p(X), tfindall(Y, p(Y), L).

X = b
Y = _922820
L = [b,a];

X = a
Y = _922820
L = [b,a];

no
```

Exceptions: Same as predicate `findall/3` (see above). Also:

`table_error` Upon execution `Goal` is not a subgoal of a tabled predicate.

`tbagof(?X, +Goal, ?List) / tsetof(?X, +Goal, ?List)` Tabling

The standard predicates `tbagof/3` and `tsetof/3` provide tabled versions of `bagof/3` and `setof/3` in a similar manner to the way in which `tfindall/3` provides a tabled version of `findall/3`.

`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.

6.6.1 Tabling Aggregate Predicates

HiLog provides an elegant way to introduce aggregate operations into XSB. HiLog allows a user to define named (and parameterized) sets (or bags). For example, say we have a simple database-like predicate, `employee(Name,Dept,Sal)`, which contains a tuple for each employee in our concern and contains the employee's name, department, and salary. From this predicate we can construct a set, or bag really, that contains all the salaries of employees in the relation:

```
:- hilog salaries.
salaries(Sal) :- employee(_Name,_Dept,Sal).
```

So `salaries` is the name of a unary predicate that is true of all salaries, or rather is the name of a *bag* of all salaries. It is a bag since it may contain the same salary multiple times. XSB provides a predicate `bagSum` which can be used to sum up the elements in a named bag. So given the definition of the HiLog predicate `salaries/1` above, we can get the sum of all the salaries with:

```
:- bagSum(salaries,TotalSals).
```

The first argument to `bagSum` is the name of a bag, and the second is bound to the sum of the elements in the bag.

We can also do a "group by" to get total salaries within departments as follows. We define a parameterized predicate, `sals(Dept)`, to be the bag of salaries of employees in department `Dept`, as follows:

```
sals(Dept)(Sal) :- employee(_Name,Dept,Sal).
```

This rule says that `Sal` is in the bag named `sals(Dept)` if there is an employee with some name who works in department `Dept` and has salary `Sal`.

Now with this definition, we can define a predicate, `deptPayroll/2`, that associates with each department the sum of all the salaries of employees in that department:

```
deptPayroll(Dept,Payroll) :- bagSum(sals(Dept),Payroll).
```

XSB provides analogous aggregate operators, described below, to compute the minimum, maximum, count, and average, of a bag, respectively. These predicates are all defined using a more basic predicate `bagReduce/4`.

`bagReduce(?SetPred,?Arg,+Op,+Id)` HiLog,Tabling

`filterReduce(?SetPred,?Arg,+Op,+Id)` Tabling

`SetPred` must be a HiLog set specification, i.e., a unary HiLog predicate. `Op` must be a Hilog operation, i.e., a 3-ary HiLog predicate that defines an associative operator. The predicate must define a binary function in which the first two arguments determine the third. `Id` must be the identity of the operator. `bagReduce` returns with `Arg` bound to the "reduce" of the elements of the bag determined by `SetPred` under the operation `Op`. I.e., `Arg` becomes the result of applying the operator to all the elements in the bag that unify with `SetPred`. See the `bagSum` operator below to see an example of `bagReduce`'s use.

`filterReduce/4` acts as `bagReduce/4` with two differences. First, it does not depend on HiLog, so that `filterReduce/4` will be more robust especially when XSB's module system is used. In addition, `filterReduce/4` aggregates solutions to `Pred` using a variance rather than unification. An example of the use of `filterReduce/4` is given in Chapter 5.

`bagPO(?SetPred,?Arg,+Order)` HiLog,Tabling

`filterPO(?SetPred,?Arg,+Order)` Tabling

`SetPred` must be a HiLog set specification, i.e., a unary HiLog predicate. `Order` must be a binary Hilog relation that defines a partial order. `bagPO` returns nondeterministically with `Arg` bound to the maximal elements, under `Order`, of the bag `SetPred`. `bagPO/3` can be used with `Order` being subsumption to reduce a set of answers and keep only the most general answers.

See the `bagMax` operator below to see an example of `bagPO`'s use.

`filterPO/3` acts as `bagPO/3` with the single difference that it does not depend on HiLog, so that `filterPO/3` will be more robust especially when XSB's module system is used.

`filterPO(#Pred,+Order)` Tabling

`filterPO(#Pred,+Order)` succeeds only for a solution $Pred\theta$ of `Pred` for which there is no solution $Pred\eta$ to `Pred` such that `Order(`$Pred\eta$`,`$Pred\theta$`)`.

Example:

For the following program

```
:- table p/2.
b(1,2).
p(1,3).
b(1,1).

prefer(b(X,X),b(X,Y)):- X ≅ Y.
```

the query

```
?- filterPO(b(X,Y)
```

will succeed only with the binding $X = 1, Y = 1$.

`bagMax(?SetPred,?Arg)` HiLog, Tabling

`SetPred` must be a HiLog set specification, i.e., a unary HiLog predicate. `bagMax` returns with `Arg` bound to the maximum element (under the Prolog term ordering) of the set `SetPred`. To use this predicate, it must be imported from aggregs, and you must give the following definitions in the main module `usermod`:

```
:- hilog maximum.
maximum(X,Y,Z) :- X @< Y -> Z=Y ; Z=X.
```

(These decarations are necessary because of a current limitation in how HiLog predicates can be used. This requirement will be lifted in a future release.) With this definition, `bagMax/2` can be (and is) defined as follows:

```
bagMax(Call,Var) :- bagReduce(Call,Var,maximum,_).
```

(Where variables are minimal in the term ordering.)

Another possible definition of `bagMax/2` would be:

```
:- hilog lt.
lt(X,Y) :- X @< Y.

bagMax(Call,Var) :- bagPO(Call,Var,lt).
```

This definition would work, but it is slightly less efficient than the previous definition since it is known that `bagMax` is deterministic.

`bagMin(?SetPred,?Arg)` HiLog,Tabling

`SetPred` must be a HiLog set specification, i.e., a unary HiLog predicate. `bagMin` returns with `Arg` bound to the minimum element (under the Prolog term ordering) of the set `SetPred`. To use this predicate, it must be imported from aggregs, and you must give the following definitions in the main module `usermod`:

```
:- hilog minimum.
minimum(X,Y,Z) :- X @< Y -> Z=X ; Z=Y.
```

(These decarations are necessary because of a current limitation in how HiLog predicates can be used. This requirement will be lifted in a future release.) With this definition, `bagMin/2` can be (and is) defined as:

```
bagMin(Call,Var) :- bagReduce(Call,Var,minimum,zz(zz)).
```

(where structures are the largest elements in the term ordering.)

`bagSum(?SetPred,?Arg)` HiLog,Tabling

`SetPred` must be a HiLog set specification, i.e., a unary HiLog predicate. `bagSum` returns with `Arg` bound to the sum of the elements of the set `SetPred`. To use this predicate, it must be imported from aggregs, and you must give the following definitions in the main module `usermod`:

```
:- hilog sum.
sum(X,Y,Z) :- Z is X+Y.
```

(These decarations are necessary because of a current limitation in how HiLog predicates can be used. This requirement will be lifted in a future release.) With this definition, `bagSum/2` can be (and is) defined as:

```
bagSum(Call,Var) :- bagReduce(Call,Var,sum,0).
```

`bagCount(?SetPred,?Arg)`

HiLog,Tabling `SetPred` must be a HiLog set specification, i.e., a unary HiLog predicate. `bagCount` returns with `Arg` bound to the count (i.e., number) of elements of the set `SetPred`. To use this predicate, it must be imported from aggregs, and you must give the following definitions in the main module `usermod`:

```
:- hilog successor.
successor(X,_Y,Z) :- Z is X+1.
```

(These decarations are necessary because of a current limitation in how HiLog predicates can be used. This requirement will be lifted in a future release.) With this definition, `bagCount/2` can be (and is) defined as:

```
bagCount(Call,Var) :- bagReduce(Call,Var,successor,0).
```

`bagAvg(?SetPred,?Arg)` HiLog,Tabling

`SetPred` must be a HiLog set specification, i.e., a unary HiLog predicate. `bagAvg` returns with `Arg` bound to the average (i.e., mean) of elements of the set `SetPred`. To use this predicate, it must be imported from aggregs, and you must give the following definitions in the main module `usermod`:

```
:- hilog sumcount.
sumcount([S|C],X,[S1|C1]) :- S1 is S+X, C1 is C+1.
```

(These decarations are necessary because of a current limitation in how HiLog predicates can be used. This requirement will be lifted in a future release.) With this definition, `bagAvg/2` can be (and is) defined as:

```
bagAvg(Call,Avg) :-
    bagReduce(Call,[Sum|Count],sumcount,[0|0]),
    Avg is Sum/Count.
```

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 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). Also, note that two anonymous variables are not identical terms. Unfortunately in the current implementation of our system (Version 2.2) 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, regardless of their numerical values.
- 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`.

Examples:

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

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

no
```

Exceptions:

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

`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`. The elements of list `L1` are scanned from left to right. Unlike `sort/2`, in `keysort/2` no merging of multiple occurring elements takes place. The time to perform the sorting is $O(nlogn)$ where n is the length of list `L1`. Note that the elements of `L1` are sorted only according to the value of `Key`, not according to the value of `Value`. The sorting of elements in `L1` is not guaranteed to be stable.

Examples:

```
| ?- keysort([3-a,1-b,2-c,1-a,3-a], L).

L = [1-b,1-a,2-c,3-a,3-a];

no
```

Exceptions:

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

`type_error` The elements of `L1` are not of the form `Key-Value`.

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)`

`once/1` is defined as `once(X):- call(X),!.` `once/1` should be used with care in tabled programs. The compiler can not determine whether a tabled predicate is called in the scope of `once/1`, and such a call may lead to runtime errors. If a tabled predicate may occur in the scope of `once/1`, use `table_once/1` instead.

Exceptions: The same as `call/1`.

`table_once(#X)`

`table_once/1` is a weaker form of `once/1`, suitable for situations in which a single solution is desired for a subcomputation that may involve a call to a tabled predicate. `table_once(?Pred)` succeeds only once even if there are many solutions to the subgoal `Pred`. However, it does not "cut over" the subcomputation started by the subgoal `Pred`, thereby ensuring the correct evaluation of tabled subgoals.

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 3.4). 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 flavours depending on the form of its argument (`Predicate_Indicator`):

1. If `Predicate_Indicator` is of the form `Module:Functor/Arity`, then the execution of `current_functor/1` will backtrack through all the current modules of the system (user defined, system defined and global modules).
2. If, however, `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 flavour 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 flavours 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 the Chapter *Foreign Language Interface* in Volume 2. 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 flavours depending on the form of its argument (`Predicate_Indicator`).

1. If `Predicate_Indicator` has the form `Module:Functor/Arity`, then the execution of `current_predicate/1` unifies the predicate indicator with predicates in all current modules (user defined, system defined and global modules).
2. If, however, `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"`). Since this is the common use of predicate `current_predicate/1`, this flavour is only for convenience. 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:

```
| ?- 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 flavours 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 a property in the following set:

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

{ loaded, dynamic, foreign }

(see predicate_property/2 below) 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_configuration(Feature_Name, ?Value)`

Succeeds iff the current value of the XSB feature `Feature_Name` is `Value`.

This predicate provides information on a wide variety of features related to how XSB was built, including the compiler used, the compiler and loader flags, the machine and OS on which XSB was built, the release number, the various directories that XSB uses to find its libraries, etc.

To find all features and their values, ask the following query:

```
| ?- xsb_configuration(FeatureName, Value), fail.
```

Here is how `xsb_configuration` might look like:

```
xsb_configuration(architecture, 'i686-pc-linux-gnu').
%% configuration is usualy the same as architecture, but it can also
%% contain special tags, {\it e.g.}, i686-pc-linux-gnu-dbg, for a verion
%% built with debugging enabled.
xsb_configuration(configuration, 'i686-pc-linux-gnu-dbg').
xsb_configuration(host_os, 'linux-gnu').
xsb_configuration(os_version, '2.34').
xsb_configuration(os_type, 'linux-gnu').
xsb_configuration(host_vendor, 'pc').
xsb_configuration(host_cpu,  'i686').
xsb_configuration(compiler, 'gcc').
xsb_configuration(compiler_flags, '  -ansi -pedantic -Wall -g').
xsb_configuration(loader_flags, ' -lm -ldl -Wl,-export-dynamic').
xsb_configuration(compile_mode, 'debug').
%% The following is XSB release information
xsb_configuration(major_version, '1').
xsb_configuration(minor_version, '9').
xsb_configuration(beta_version, '3').
xsb_configuration(version, '1.9-b3').
xsb_configuration(codename, 'Code Breaker').
xsb_configuration(release_date, date(1998, 10, 17)).
%% XSB query evaluation directive
xsb_configuration(scheduling_strategy, '(batched)').
%% Support for other languages
xsb_configuration(perl_support, 'yes').
xsb_configuration(perl_archlib, '/usr/lib/perl5/i386-linux/5.00404').
xsb_configuration(perl_cc_compiler, 'cc').
xsb_configuration(perl_ccflags, '-Dbool=char -DHAS_BOOL -I/usr/local/include').
xsb_configuration(perl_libs, '-lnsl -lndbm -lgdbm -ldb -ldl -lm -lc -lposix -lcrypt').
xsb_configuration(javac, '/usr/bin/javac').
/* Tells where XSB is currently residing; can be moved */
xsb_configuration(install_dir, InstallDir) :- ...
/* User home directory. Usually HOME. If that is null, then it would
   be the directory where XSB is currently residing.
   This is where we expect to find the .xsb directory */
xsb_configuration(user_home, Home) :- ...
/* Where XSB invocation script is residing */
xsb_configuration(scriptdir, ScriptDir) :- ...
/* where are cmplib, syslib, lib, packages, etc live */
xsb_configuration(cmplibdir, CmplibDir) :- ...
```

```
xsb_configuration(libdir, LibDir) :- ...
xsb_configuration(syslibdir, SyslibDir) :- ...
xsb_configuration(packagesdir, PackDir) :-  ...
xsb_configuration(etcdir, EtcDir) :- ...
/* architecture and configuration specific directories */
xsb_configuration(config_dir, ConfigDir) :- ...
xsb_configuration(config_libdir, ConfigLibdir) :- ...
/* site-specific directories */
xsb_configuration(site_dir, '/usr/local/XSB/site').
xsb_configuration(site_libdir, SiteLibdir) :- ...
/* site and configuration-specific directories */
xsb_configuration(site_config_dir, SiteConfigDir) :- ...
xsb_configuration(site_config_libdir, SiteConfigLibdir) :- ...
/* Where user's arch-specific libraries are found by default. */
xsb_configuration(user_config_libdir, UserConfigLibdir) :- ...
```

`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), fail.
```

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.
`goal`	the goal passed to XSB on command line with the '-e' switch; '`true.`' if nothing is passed.
`dcg_style`	the DCG style currently used; `xsb` or `standard` (standard is used in Quintus, SICSTUS, etc.). See Section 8.4 for more details.
`garbage_collection`	`"none"`, `"sliding"`, or `"copying"` depending on the garbage collection strategy that is currently being employed (see also Section 3.5).

Note that `xsb_flag` is used only for dynamic XSB settings, *i.e.*, settings that might change between sessions or within the same session. For static configuration information, the predicate `xsb_configuration/2` is used.

`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 exactly 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 an array of features for modifying the dynamic database. Using `assert/1`, clauses can be asserted using first-argument indexing in a manner that is now standard to Prolog implementations. While this is the default behavior for XSB, other behavior can be specified using the (executable) directives `index/3` and `index/2`. For instance, dynamic clauses can be declared to have multiple or joint indexes, while this indexing can be either hash-based as is typical in Prolog systems or based on *tries*. No matter what kind of indexing is used, space is dynamically allocated when a new clause is asserted and, unless specified otherwise, released when it is retracted. Furthermore, the size of any index table expands dynamically as clauses are asserted.

Consider first dynamic predicates that use traditional hash-based indexing. XSB asserts WAM code for such clauses, leading to execution times similar to compiled code for unit and binary clauses. Furthermore, tabling can be used with a dynamic predicate by explicitly declaring a predicate to be both dynamic and tabled. For clauses that are asserted as WAM code, the *"immediate semantics"* of dynamic predicates is used, not the so-called *"logical semantics"* of assert/retract [27]. 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.
```

Neither `retract/1` nor `retractall/1` support this behavior, due to their techniques for space reclamation. One alternative is to use `findall/3` to collect the intersection first, before retracting. Another is to use the predicates `retract_nr/1` and `reclaim_space/1`, described below.

Asserting clauses as WAM code might be considerably slow for some applications. To remedy this, XSB provides an alternative to `assert/1` which implements assert's functionality using the trie-based tabling data structures [33]. Though trie-based dynamic code can be created (and usually executed) significantly faster than using `assert/1`, users of the following predicates should be aware that trie-based assert can be used only for unit clauses where a relation is viewed as a set, and where the order of the facts is not important.

XSB does not at this time fully support dynamic predicates defined within compiled code. The only way to generate dynamic code is by explicitly asserting it, or by using the standard predicate `load_dyn/1` to read clauses from a file and assert them (see the section *Asserting Dynamic Code* in Volume 2). There is a `dynamic/1` predicate (see page 108) that declares a predicate within the system so that if the predicate is called when no clauses are presently defining it, the call 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))`. Default: first-argument indexing.

`asserta(+Clause)`
: If the index specification for the preicate is not `tries`, this predicate adds a dynamic clause, `Clause`, to the database *before* any other clauses for the same predicate currently in the database. If the index specification for the predicate is `trie`, the clause is asserted arbitrarily within the trie, and a warning message sent to `stderr`.

`assertz(+Clause)`
: If the index specification for the predicate is not `tries`, this predicate adds a dynamic clause, `Clause`, to the database *after* any other clauses for the same predicate currently in the database. If the index specification for the predicate is `trie`, the clause is asserted arbitrarily within the trie, and a warning message sent to `stderr`.

`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 behavior 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 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 order to reclaim space after using `retract_nr/1`, see `reclaim_space/1` below. Predicate `retract_nr/1` is not a standard predicate and must be imported from module `assert`. `retract_nr/1` is provided for (partial) compatibility with the `retract/1` predicate of SB-Prolog.

`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/1`. 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 must be made after execution has completely failed out of choice points for `q(X)` and `p(X)`. Predicate `reclaim_space/1` is not standard but must be imported from module `assert`. As with `retract_nr`, the use of this predicate is discouraged; it is provided for (partial) compatibility with SB-Prolog.

`index(+PredSpec, +IndexSpec)`

In general, XSB supports hash-based indexing on alternate arguments or combinations of arguments, along with trie-based indexing. The availability of various kinds of indexing depends on whether code is static (e.g. compiled) or dynamic (e.g. asserted or loaded with `load_dyn/1`). The executable directive `index/2` does *not* re-index an already existing predicate but takes effect only if the program store contains no clauses for `PredSpec`. Index directives can be given to the compiler as part of source code or executed during program execution (analogously to `op/3`).

- *Hash-based Indexing*
  - *Static Predicates* In this case `IndexSpec` must be a non-negative integer which indicates the argument on which an index is to be constructed. If `IndexSpec` is 0, then no index is kept (possibly an efficient strategy for predicates with only one or two clauses.)
  - *Dynamic Predicates* For a dynamic predicate, (to which no clauses have yet been asserted), `IndexSpec` is either an `IndexElt` or a list of `IndexElt`s. Each `IndexElt` specifies an argument or group of arguments on which to build an index. Syntactically, an `IndexElt`, in its turn is a non-negative integer or a sequence of up to three non-negative integers separated by `+`, e.g., `1+2+3`.
    For example, `index(p/3,[2,1])` indicates that clauses asserted to the predicate `p/3` should be indexed on both the second and the first argument. 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, it will check to see if the first argument is nonvariable and if so, use that index.
    As another example, one could specify: `index(p/5,[1+2,1,4])`. 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. As a last resort, 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 that appears in a joint specification later alone, as 1 in this example, but it never makes sense forcing the single argument to appear earlier. In that case the joint index would never be used.)
- *Trie-based Indexing* The executable declaration `index(Predspec,trie)` causes clauses for `Predspec` to be asserted using tries (see [33], which is available through the XSB web page). The name trie indexing is something of a misnomer since the trie itself both indexes the term and represents it. In XSB, the above trie index is formed using a left-to-right traversal of the unit clauses. These indexes can be very effective if discriminating information lies deep within a term, and if there is sharing of left-prefixes of a term, can reduce the space needed to represent terms. Furthermore, asserting a unit clause as a trie is much faster than asserting it using default WAM code.

  Despite these advantages, representing terms as tries leads to semantic differences from asserted code, of which the user should be aware. First, the order of clauses within a trie is arbitrary: using `asserta/1` or `assertz` for a predicate currently using trie indexing will give the same behavior as using `assert`. Also, the current version of XSB only allows trie indexing for unit clauses.

  Trie-based indexing is available only for dynamic predicates.

`dynamic(+PredSpec)`

is an executable predicate which converts a predicate specified as (Predicate/Arity) to a dynamic predicate. If `Predicate` 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 the

predicate is previously defined and dynamic, `dynamic/1` is a noop. If previously defined as compiled, `Predicate` will be converted to dynamic, which means that clauses can be added, although the compiled portion cannot be manipulated. Note that `dynamic/1` 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 [20] when the file contains clauses defining the so-specified predicate.

`table(+PredSpec)`
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 in order for the predicate to be tabled.

6.11 Execution State

`abort`
Abandons the current execution and returns to the top level. This predicate 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 behavior may occur after the program is aborted and restarted, unless the user manually closes the files.

Aborting closes all incomplete tables (those which may not have a complete set of answers). Closed tables 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` on UNIX systems). 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 incomplete tables (those which may not have a complete set of answers). Closed tables are unaffected, even if the tables were created during the computation for 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 > fail.

no
Yes master >
```

garbage_collection(+Option)

Sets the system so that subsequent heap garbage collecting will be done according to the specified Option. Option may be the atom none indicating that heap garbage collection is turned off; it may be the atom sliding indicating that sliding garbage collection will be done; or it may be the atom copying indicating that the copying garbage collector will be used.

cputime(-CPU_Time)

Returns the CPU_Time 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 3.5) containing the
  - global stack (heap) and local (environment) stack
  - trail and choice point stack
  - SLG subgoal space (tablestack)
  - SLG unification stack
  - SLG completion stack
  - the space occupied by subgoal and answer tables (in the form of tries).
- Current use of the above specified memory areas (allocated/in use/free).
- Information about process cpu and clock time.

Additionally, if the emulator is invoked with the `'-s'` option (see Section 3.5), information is printed out about

- Maximum use of the memory areas.

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

```
| ?- statistics.

memory (total)        1873737 bytes:     171317 in use,    1702420 free
   permanent space     169801 bytes
   glob/loc space      786432 bytes:       1080 in use,     785352 free
      global                                152 bytes
      local                                 928 bytes
   trail/cp space      786432 bytes:        436 in use,     785996 free
      trail                                 240 bytes
      choice point                          196 bytes
   SLG subgoal space        0 bytes:          0 in use,          0 free
   SLG unific. space    65536 bytes:          0 in use,      65536 free
   SLG completion       65536 bytes:          0 in use,      65536 free
   SLG trie space           0 bytes:          0 in use,          0 free
   (call+ret. trie          0 bytes,   trie hash tables          0 bytes)

   Maximum stack use: global 224, local 1384, trail 240, cp 492
   Maximum stack use: SLG completion 0.  Max level: 0

     0 Trail unwinds,       0 levels

 0.570 sec. cputime,  5.088 sec. elapsetime
```

shows how the emulator output looks if it is invoked with the `'-s'` option (without it the Maximum use line is not shown). Information about the allocation size is provided since the sizes can be changed through emulator options (see Section 3.5).

`shell(+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 `shell/2`).

For example, the call:

```
| ?- shell('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', shell(['cc -c ', File]).
```

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

Note that in UNIX systems, since `shell/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.

`shell(+SystemCall, -Result)`

Calls the operating system with the atom `SystemCall` as argument and returns the result of the call in `Result`. In comparison with `shell/1` this predicate always succeeds, even if the `SystemCall` cannot be successfully executed.

`ls`

Under UNIX, this command lists in the current output stream the files in the system's current directory if it can do so. If so, it succeeds. It is the same as `shell('ls -F', 0)`.

`cd(+Dir)`

Under UNIX and Windows, this predicate 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, and the system is executing under UNIX, 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, it is often useful to be able to explicitly inspect a table, or to alter its state. In the following predicates, which are provided for this purpose, `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` (cf. Section 3.3). 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(+Skeleton,-Subgoal_Structure_Pointer,-Return_Skeleton)` Tabling

Backtracks through the subgoal trie, unifying `Skeleton` with entries in the `call trie`. As it does so, `get_calls/3` binds `Subgoal_Structure_Pointer` with the pointer to the `subgoal structure` and binds `Return_Skeleton` to a term of the form ret/n where each of its arguments corresponds to a free variable in the call. Example:

For the following program and table

```
:- table p/2.
p(1,2).
p(1,3).
p(1,X).
p(2,3).
```

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/3` will act as follows

```
| ?- get_calls(p(X,Y),Cs,Ret).

X = _864816
Y = 3
Cs = 2927152
Ret = ret(_864816);

X = 1
Y = _864644
Cs = 2927104
Ret = ret(_864644);

no
| ?- get_calls(p(1,Y),Cs,Ret).

Y = 3
Cs = 2927152
Ret = ret(1);

Y = _864620
Cs = 2927104
Ret = ret(_864620);

no
| ?- get_calls(p(Y,3),Cs,Ret).

Y = _864792
Cs = 2927152
Ret = ret(_864792);

Y = 1
Cs = 2927104
Ret = ret(3);

no
| ?- get_calls(p(1,3),Cs,Ret).

Cs = 2927152
Ret = ret(1);

Cs = 2927104
Ret = ret(3);

no
```

`get_call(+Skeleton,-Subgoal_Structure_Pointer,-Return_Skeleton)` Tabling

This predicate binds `Subgoal_Structure_Pointer` and `Return_Skeleton` only if `Skeleton` is a variant (i.e., identical up to variable renaming) of some entry in the subgoal table. It may be compared to the previous predicate `get_call/3` which *unifies* `Skeleton` with entries in the subgoal trie. Repeating the example from `get_calls/3`, calls to `get_call/3` will act as follows

```
| ?- get_call(p(X,Y),Cs,Ret).

no
| ?- get_call(p(1,Y),Cs,Ret).

Y = _864620
Cs = 2927104
Ret = ret(_864620);

no
| ?- get_call(p(Y,3),Cs,Ret).

Y = _864792
Cs = 2927152
Ret = ret(_864792);

no
| ?- get_call(p(1,3),Cs,Ret).

no
```

`get_calls_for_table(+Skeleton,-Call)` Tabling

Succeeds whenever `Skeleton` unfies with an entry in the subgoal table. When this predicate suceeds, it returns a variant of the call entry with which `Skeleton` unifies.

This predicate does not provide any information about whether a table is complete. Use `table_state` to inquire about a table's state.

Calls to `get_calls_for_table/2` for the example in `get_calls/3` would act as follows

```
|?- get_calls_for_table(p(X,Y), Call).
X = _646608
Y = _646436
Call = p(1,_646724) ;

X = _646608
Y = _646436
Call = p(_646720,3) ;

no
| ?- get_calls_for_table(p(1,2), Call).
Call = p(1,_646676)

Call = p(_646672,3) ;

no
```

The second example backtracks through all entries in the table, since only skeletal information is used from the first argument.

Exception:

`instantiation_error` First argument is not instantiated.

`table_error` First argument is not a tabled predicate.

`get_returns(+Subgoal_Structure_Pointer,-Return_Skeleton)` Tabling

Backtracks through the answer trie for the subgoal whose `subgoal structure` is pointed to by `Subgoal_Structure_Pointer`, and instantiates the Return_Skeleton with the bindings corresponding to the return.

One way of accessing subgoals and answers of the example in in `get_calls/3` is as follows

```
| ?- get_calls(p(Y,3),Cs,Ret), get_returns(Cs,Ret).

Y = 2
Cs = 2561656
Ret = ret(2);

Y = 1
Cs = 2561656
Ret = ret(1);

Y = 1
Cs = 2559032
Ret = ret(3);

Y = 1
Cs = 2559032
Ret = ret(3);

no
```

`get_returns(+Subgoal_Structure_Pointer,-Return_Skeleton,-LeafNode)` Tabling

Same as above, except for the fact that the third argument is bound to the leaf node corresponding to the return in the return trie.

`get_returns_for_call(+Call,-Return)` Tabling

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 successive 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:

`instantiation_error` First argument is not instantiated.

`table_error` First argument is not a tabled predicate.

`get_residual(?Skeleton,?Delay_list)` Tabling

`get_residual/2` is used to backtrack through answers that unify with a skeleton. These answers may in fact come from any call that unifies with `Skeleton`. Since the `Delay_list` of an answer consists of those literals whose truth value is unknown in the well-founded model of the program (see Section 5) `get_residual/2` can be useful when extensions of the well-founded model are desired. Consider the program

For the following program and table

```
:- table p/2.
p(1,2).
p(1,3):- tnot(p(2,3)).
p(2,3):- tnot(p(1,3)).
```

Call	*Returns*
p(1,X)	p(1,2) p(1,3):- tnot(p(2,3)) p(2,3):- tnot(p(1,3))
p(1,3)	p(1,3):- tnot(p(2,3))
p(2,3)	p(2,3):- tnot(p(1,3))

calls to `get_residual/3` will act as follows

```
| ?- get_residual(p(X,Y),List).

X = 1
Y = 2;
Z = [];

X = 1
Y = 3;
Z = [tnot(p(2,3))];

X = 2
Y = 3;
Z = [tnot(p(1,3))];

X = 1
```

```
Y = 3;
Z = [tnot(p(2,3))];

X = 2
Y = 3;
Z = [tnot(p(1,3))];

no
```

`delete_return(+Subgoal_Structure_Pointer,+LeafNode)` Tabling

Deletes a return from the return trie. The LeafNode parameter should be obtained from a call to get_returns/3.

`abolish_all_tables` Tabling

`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 2.2, `abolish_all_tablesl/0` reclaims used space. The predicate always succeeds.

Note that incomplete 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(+PredSpecification` *or* `+Skeleton)` Tabling

Predicate `abolish_table_pred/1` abolishes tables for all calls to the predicate denoted by *PredSpecification* (as in `path/2`) or by *Skeleton* (as in `path(X,Y)`). The predicate remains tabled, but information about its calls and their returns is removed from the system. In Version 2.2, `abolish_table_pred/1` reclaims the space used by the calls to this predicate (and their answers). Note that incomplete 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:

`instantiation_error` Argument is a variable.

`type_error` Argument is not a predicate specification of a callable term.

`table_error` Argument is not a (specification of a) tabled predicate.

`abolish_table_call(?Call)` Tabling

Predicate `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 2.2, `abolish_table_call/1` does not reclaim used space. Note that incomplete 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.

`table_state(?Call,-State)` Tabling

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 meaning of these atoms is 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.

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 2.2), it does not work very well when debugging code involving tabled predicates. If one only creeps (see below), the tracing can provide some useful information. We do intend that future versions will have more complete 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.

The debugger also has profiling capabilities, which can measure the cpu time spent in each call. The cpu time is measured only down to 0.0001-th of a second.

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.

`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.

`q`: **Quasi-skip** This is like Skip except that it does not mask out spy points.

`S`: **Verbose skip** Similar to `Skip` mode, but trace continues to be printed. The user is prompted again when the current call terminates with success or failure. This can be used to obtain a full trace to the point where an error occurred or for code profiling. (See more about profiling below.)

`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. Optionally, only a predicate name can be provided, in which case it refers to all predicates of any arity currently defined in `usermod`. It may optionally may be prefixed by a module name, e.g. $ModName{:}P/N$. (Again, if the arity is omitted, the specification refers to all predicates of any arity with the given name currently defined in the given module.) 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.

`debug_ctl(option,value)`
: `debug_ctl/2` performs debugger control functions as described below. These commands can be entered before starting a trace or inside the trace. The latter can be done by responding with "`b`" at the prompt, which recursively invokes an XSB sub-session. At this point, you can enter the debugger control commands and type `end_of_file.` This returns XSB back to the debugger prompt, but with new settings.

1. `debug_ctl(prompt, off)` Set non-interactive mode globally. This means that trace will be printed from start to end, and the user will never be prompted during the trace.
2. `debug_ctl(prompt, on)` Make tracing/spying interactive.
3. `debug_ctl(profile, on)` Turns profiling on. This means that each time a call execution reaches the `Fail` or `Exit` port, CPU time spent in that call will be printed. The actual call can be identified by locating a `Call` prompt that has the same number as the "cpu time" message.
4. `debug_ctl(profile, off)` Turns profiling off.
5. `debug_ctl(redirect, +File)` Redirects debugging output to a file. This also includes program output, errors and warnings. Note that usually you cannot see the contents of `+File` until it is closed, *i.e.*, until another redirect operation is performed (usually `debug_ctl(redirect, tty)`, see next).
6. `debug_ctl(redirect, tty)` Attaches the previously redirected debugging, error, program output, and warning streams back to the user terminal.

7. `debug_ctl(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'].
8. `debug_ctl(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.
9. `debug_ctl(hide, +PredArityPairList)` The list must be of the form `[P1/A1, P2/A2, ...]`, *i.e.*, each either must specify a predicate-arity pair. Each predicate on the list will become non-traceable. That is, during the trace, each such predicate will be treated as an black-box procedure, and trace will not go into it.
10. `debug_ctl(unhide, ?PredArityPairList)` If the list is a predicate-arity list, every predicate on that list will become traceable again. Items in the list can contain variables. For instance, `debug_ctl(unhide, [_/2])` will make all 2-ary that were previously made untraceable traceable again. As a special case, if `PredArityPairList` is a variable, all predicates previously placed on the "untraceable"-list will be taken off.
11. `debug_ctl(hidden, -List)` This returns the list of predicates that the user said should not be traced.

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 3.5), 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.

XSB debugger also provides means for the low-level control of what must be traced. Normally, various low-level predicates are masked out from the trace, since these predicates do not make sense to the application programmer. However, if tracing below the application level is needed, you can retract some of the facts specified in the file `syslib/debugger_data.P` (and in some cases assert into them). All these predicates are documented in the header of that file. Here we only mention the four predicates that an XSB developer is more likely to need. To get more trace, you should retract from the first three predicates and assert into the last one.

- `hide_this_show(Pred,Arity)`: specifies calls (predicate name and arity) that the debugger should `not` show at the prompt. However, the evaluation of this hidden call `is` traced.
- `hide_this_hide(Pred,Arity)`: specifies calls to hide. Trace remains off while evaluating those predicates. Once trace is off, there is no way to resume it until the hidden predicate exits or fails.
- `show_this_hide(Pred,Arity)`: calls to show at the prompt. However, trace is switched off right after that.

- `trace_standard_predicate(Pred,Arity)`: Normally trace doesn't go inside standard predicates (*i.e.*, those specified in `syslib/std_xsb.P`. If you need to trace some of those, you must `assert` into this predicate.

In principle, by retracting all facts from the first three predicates and asserting enough facts into the last one, it is possible to achieve the behavior that approximates the `-T` option. However, unlike `-T`, debugging can be done interactively. This does not obviate `-T`, however. First, it is easier to use `-T` than to issue multiple asserts and retracts. Second, `-T` can be used when the error occurs early on, before the moment when XSB shows its first prompt.

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`, the user can specify any desired transformation to be done as clauses are read by the reader of XSB's parser. 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 appearance 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`, $'\backslash+'/1$ 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] 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 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

[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.

```
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:

`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 performed by predicate `expand_term/2` by defining their own compile-time transformations. This can be done by defining clauses for the predicate `term_expansion/2`. 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. Note that predicate `term_expansion/2` must be defined in the XSB's default read-in module (`usermod`) and should be loaded there before the compilation begins.

`'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.

`dcg(+DCG_Rule, ?Prolog_Clause)` `dcg`

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. `dcg/2` must be explicitly imported from the module `dcg`.

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* to 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 standard expansion provided by the other Prolog systems, the user can simply execute:

```
set_dcg_style(standard).
```

To switch back to the XSB-style DCG's, call

```
set_dcg_style(xsb).
```

This can be done anywhere in the program, or interactively. By default, XSB starts with the XSB-style DCG's. To change that, start XSB as follows:

```
xsb -e "set_dcg_style(standard)."
```

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 `'!'/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`, $'\backslash +' /1$, or `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* to 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 or doubly 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 of *chart parsing algorithms.*

In the simplest cases, one needs only to add the directive `:- auto_table` (see Section 3.8.4) 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 `:- auto_table` 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 expansion.

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. To avoid this problem, the input list can be transformed into a set of datalog atoms. Currently this must be done manually, as explained in [47], available in the tech reports directory.

Chapter 9

Restrictions and Current Known Bugs

In this chapter we indicate some features and bugs of XSB that may affect the users at some point in their interaction with the system.

If at some point in your interaction with the system you suspect that you have run across a bug not mentioned below, please report it to (`xsb-contact@cs.sunysb.edu`). 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. We will do our best to fix it or to assist you to bypass it.

9.1 Current Restrictions

- The maximum arity for predicate and function symbols is 255.
- The maximum length of atoms that can be stored in an XSB *object* code file is in principle $2^{32} - 1$, but in practice it is $2^{28} - 1$ (i.e., in 32-bit platforms it is bounded by the size of the maximum integer; see below).
- 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.
- XSB allows up to 1 Gigabyte of address space for 32-bit SUNs and 512 Megabytes of address space for other 32-bit platforms. For SUNs the address space for integers is -2^{28}—$(2^{28} - 1)$. For MIPS-based machines (e.g. Silicon Graphics machines), the address space for integers is -2^{26}—$(2^{26} - 1)$. For all other machines it is -2^{27}—$(2^{27} - 1)$. This restriction can cause unexpected results when numbers are computed. The amount of space allowed for floating point numbers is similar for each machine. For 64-bit platforms, addresses, integers, and floating point numbers are all stored in 60 bits. However, as the *object* code file format is the same as for the 32-bit versions, compiled constants are subject to 32-bit limitations.
- Indexing on floating-point numbers does not work, since, as implemented in XSB, the semantics of floating-point unification is murky in the best case. Therefore, it is advisable that if

you use floating point numbers in the first argument of a procedure, that you explicitly index the predicate in some other argument.

- The XSB compiler cannot distinguish the occurrences 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.

9.2 Known Bugs

- The current version of XSB does not fully support dynamic code. In fact the declartion `:- dynamic` essentially instructs XSB to fail on that code if it is undefined.

- Currently the C foreign language interface does not work when XSB is *also* compiled with the Oracle interface on Solaris.

- 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.

- The reader cannot read an infix operator immediately followed 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).
```

- The reader cannot properly read an operator defined as both a prefix and an infix operator. For instance the declaration

```
:- op(1200,xf,'<=').
:- op(1200,xfx,'<=').
```

will lead to a syntax error.

- 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.
- Currently, term comparisons (`==`,`@<=`,`@<`,`@>`, and `@>=`) do not work for terms that overflow the C-recursion stack (terms that contain more than 10,000 variables and/or function symbols).

Appendix A

GPP - Generic Preprocessor

Version 2.0 - (c) Denis Auroux 1996-99
`http://www.math.polytechnique.fr/cmat/auroux/prog/gpp.html`

As of version 2.1, XSB uses *gpp* as a source code preprocessor for Prolog programs. This helps maintain consistency between the C and the Prolog parts of XSB through the use of the same .h files. In addition, the use of macros improves the readability of many Prolog programs, especially those that deal with low-level aspects of XSB. Chapter 3.8 explains how *gpp* is invoked in XSB.

A.1 Description

gpp is a general-purpose preprocessor with customizable syntax, suitable for a wide range of preprocessing tasks. Its independence on any programming language makes it much more versatile than *cpp*, while its syntax is lighter and more flexible than that of *m4*.

gpp is targeted at all common preprocessing tasks where *cpp* is not suitable and where no very sophisticated features are needed. In order to be able to process equally efficiently text files or source code in a variety of languages, the syntax used by *gpp* is fully customizable. The handling of comments and strings is especially advanced.

Initially, gpp only understands a minimal set of built-in macros, called *meta-macros*. These meta-macros allow the definition of *user macros* as well as some basic operations forming the core of the preprocessing system, including conditional tests, arithmetic evaluation, and syntax specification. All user macro definitions are global, i.e. they remain valid until explicitly removed; meta-macros cannot be redefined. With each user macro definition gpp keeps track of the corresponding syntax specification so that a macro can be safely invoked regardless of any subsequent change in operating mode.

In addition to macros, gpp understands comments and strings, whose syntax and behavior can be widely customized to fit any particular purpose. Internally comments and strings are the same construction, so everything that applies to comments applies to strings as well.

A.2 Syntax

```
gpp [-o outfile] [-I/include/path] [-Dname=val ...]
    [-z|+z] [-x] [-m] [-n] [-C|-T|-H|-P|-U ... [-M ...]]
    [+c<n> str1 str2] [-c str1]
    [+s<n> str1 str2 c] [infile]
```

A.3 Options

gpp recognizes the following command-line switches and options:

- **-h**
  Print a short help message.
- **-o** outfile
  Specify a file to which all output should be sent (by default, everything is sent to standard output).
- **-I** /include/path
  Specify a path where the *#include* meta-macro will look for include files if they are not present in the current directory. The default is /usr/include if no -I option is specified. Multiple -I options may be specified to look in several directories.
- **-D** name=val
  Define the user macro *name* as equal to *val*. This is strictly equivalent to using the *#define* meta-macro, but makes it possible to define macros from the command-line. If *val* makes references to arguments or other macros, it should conform to the syntax of the mode specified on the command-line. Note that macro argument naming is not allowed on the command-line.
- **+z**
  Set text mode to Unix mode (LF terminator). Any CR character in the input is systematically discarded. This is the default under Unix systems.
- **-z**
  Set text mode to DOS mode (CR-LF terminator). In this mode all CR characters are removed from the input, and all output LF characters are converted to CR-LF. This is the default if gpp is compiled with the WIN_NT option.
- **-x**
  Enable the use of the *#exec* meta-macro. Since *#exec* includes the output of an arbitrary shell command line, it may cause a potential security threat, and is thus disabled unless this option is specified.
- **-m**
  Enable automatic mode switching to the cpp compatibility mode if the name of an included file ends in '.h' or '.c'. This makes it possible to include C header files with only minor modifications.

- **-n**
  Prevent newline or whitespace characters from being removed from the input when they occur as the end of a macro call or of a comment. By default, when a newline or whitespace character forms the end of a macro or a comment it is parsed as part of the macro call or comment and therefore removed from output. Use the -n option to keep the last character in the input stream if it was whitespace or a newline.

- **-U** arg1 ... arg9
  User-defined mode. The nine following command-line arguments are taken to be respectively the macro start sequence, the macro end sequence for a call without arguments, the argument start sequence, the argument separator, the argument end sequence, the list of characters to stack for argument balancing, the list of characters to unstack, the string to be used for referring to an argument by number, and finally the quote character (if there is none an empty string should be provided). These settings apply both to user macros and to meta-macros, unless the -M option is used to define other settings for meta-macros. See the section on syntax specification for more details.

- **-M** arg1 ... arg7
  User-defined mode specifications for meta-macros. This option can only be used together with -M. The seven following command-line arguments are taken to be respectively the macro start sequence, the macro end sequence for a call without arguments, the argument start sequence, the argument separator, the argument end sequence, the list of characters to stack for argument balancing, and the list of characters to unstack. See below for more details.

- **(default mode)**
  The default mode is a vaguely cpp-like mode, but it does not handle comments, and presents various incompatibilities with *cpp*. Typical meta-macros and user macros look like this:

  ```
  #define x y
  macro(arg,...)
  ```

  This mode is equivalent to

  ```
  -U "" "" "(" "," ")" "(" ")" "#" "\\"
  -M "#" "\n" " " " " " "\n" "(" ")"
  ```

- **-C**
  cpp compatibility mode. This is the mode where gpp's behavior is the closest to that of cpp. Unlike in the default mode, meta-macro expansion occurs only at the beginning of lines, and C comments and strings are understood. This mode is equivalent to

  ```
  -n -U "" "" "(" "," ")" "(" ")" "#" ""
  -M "\n#\w" "\n" " " " " " "\n" "" ""
  +c "/*" "*/" +c "//" "\n" +c "\\\n" ""
  +s "\"" "\"" "\\" +s "'" "'" "\\"
  ```

- **-T**
  TeX-like mode. In this mode, typical meta-macros and user macros look like this:

  ```
  \define{x}{y}
  \macro{arg}{...}
  ```

  No comments are understood. This mode is equivalent to

  ```
  -U "\\" "" "{" "}{" "}" "{" "}" "#" "@"
  ```

- **-H**
  HTML-like mode. In this mode, typical meta-macros and user macros look like this:

  ```
  <#define x|y>
  <#macro arg|...>
  ```

  No comments are understood. This mode is equivalent to

  ```
  -U "<#" ">" "\B" "|" ">" "<" ">" "#" "\\"
  ```

- **-P**
  Prolog-compatible cpp-like mode. This mode differs from the cpp compatibility mode by its handling of comments, and is equivalent to

  ```
  -n -U "" "" "(" "," ")" "(" ")" "#" ""
  -M "\n#\w" "\n" " " " " "\n" "" ""
  +ccss "\!o/*" "*/" +ccss "%" "\n" +ccii "\\\n" ""
  +s "\"" "\"" "" +s "\!#'" "'" ""
  ```

- **+c** <n> str1 str2
  Specify comments. Any unquoted occurrence of *str1* will be interpreted as the beginning of a comment. All input up to the first following occurrence of *str2* will be discarded. This option may be used multiple times to specify different types of comment delimiters. The optional parameter <*n*> can be specified to alter the behavior of the comment and e.g. turn it into a string or make it ignored under certain circumstances, see below.

- **-c** str1
  Un-specify comments or strings. The comment/string specification whose start sequence is *str1* is removed. This is useful to alter the built-in comment specifications of a standard mode, e.g. the cpp compatibility mode.

- **+s** <n> str1 str2 c
  Specify strings. Any unquoted occurrence of *str1* will be interpreted as the beginning of a string. All input up to the first following occurrence of *str2* will be output as is without any evaluation. The delimiters themselves are output. If *c* is non-empty, its first character is used as a *string-quote character*, i.e. a character whose presence immediately before an occurrence

of *str2* prevents it from terminating the string. The optional parameter <*n*> can be specified to alter the behavior of the string and e.g. turn it into a comment, enable macro evaluation inside the string, or make the string specification ignored under certain circumstances, see below.

- **-s** str1
  Un-specify comments or strings. Identical to -c.

- **infile**
  Specify an input file from which gpp reads its input. If no input file is specified, input is read from standard input.

A.4 Syntax Specification

The syntax of a macro call is the following : it must start with a sequence of characters matching the *macro start sequence* as specified in the current mode, followed immediately by the name of the macro, which must be a valid *identifier*, i.e. a sequence of letters, digits, or underscores ("_"). The macro name must be followed by a *short macro end sequence* if the macro has no arguments, or by a sequence of arguments initiated by an *argument start sequence*. The various arguments are then separated by an *argument separator*, and the macro ends with a *long macro end sequence*.

In all cases, the parameters of the current context, i.e. the arguments passed to the body being evaluated, can be referred to by using an *argument reference sequence* followed by a digit between 1 and 9. Macro parameters may alternately be named (see below). Furthermore, to avoid interference between the gpp syntax and the contents of the input file a *quote character* is provided. The quote character can be used to prevent the interpretation of a macro call, comment, or string as anything but plain text. The quote character "protects" the following character, and always gets removed during evaluation. Two consecutive quote characters evaluate as a single quote character.

Finally, to facilitate proper argument delimitation, certain characters can be "stacked" when they occur in a macro argument, so that the argument separator or macro end sequence are not parsed if the argument body is not balanced. This allows nesting macro calls without using quotes. If an improperly balanced argument is needed, quote characters should be added in front of some stacked characters to make it balanced.

The macro construction sequences described above can be different for meta-macros and for user macros: this is e.g. the case in cpp mode. Note that, since meta-macros can only have up to two arguments, the delimitation rules for the second argument are somewhat sloppier, and unquoted argument separator sequences are allowed in the second argument of a meta-macro.

Unless one of the standard operating modes is selected, the above syntax sequences can be specified either on the command-line, using the -M and -U options respectively for meta-macros and user macros, or inside an input file via the *#mode meta* and *#mode user* meta-macro calls. In both cases the mode description consists of 9 parameters for user macro specifications, namely the macro start sequence, the short macro end sequence, the argument start sequence, the argument separator, the long macro end sequence, the string listing characters to stack, the string listing characters to unstack, the argument reference sequence, and finally the quote character. As explained below

these sequences should be supplied using the syntax of C strings; they must start with a non-alphanumeric character, and in the first five strings special matching sequences can be used (see below). If the argument corresponding to the quote character is the empty string that functionality is disabled. For meta-macro specifications there are only 7 parameters, as the argument reference sequence and quote character are shared with the user macro syntax.

The structure of a comment/string is the following : it must start with a sequence of characters matching the given *comment/string start sequence*, and always ends at the first occurrence of the *comment/string end sequence*, unless it is preceded by an odd number of occurrences of the *string-quote character* (if such a character has been specified). In certain cases comment/strings can be specified to enable macro evaluation inside the comment/string: in that case, if a quote character has been defined for macros it can be used as well to prevent the comment/string from ending, with the difference that the macro quote character is always removed from output whereas the string-quote character is always output. Also note that under certain circumstances a comment/string specification can be *disabled*, in which case the comment/string start sequence is simply ignored. Finally, it is possible to specify a *string warning character* whose presence inside a comment/string will cause gpp to output a warning (this is useful e.g. to locate unterminated strings in cpp mode). Note that input files are not allowed to contain unterminated comments/strings.

A comment/string specification can be declared from within the input file using the *#mode comment* meta-macro call (or equivalently *#mode string*), in which case the number of C strings to be given as arguments to describe the comment/string can be anywhere between 2 and 4: the first two arguments (mandatory) are the start sequence and the end sequence, and can make use of the special matching sequences (see below). They may not start with alphanumeric characters. The first character of the third argument, if there is one, is used as string-quote character (use an empty string to disable the functionality), and the first character of the fourth argument, if there is one, is used as string-warning character. A specification may also be given from the command-line, in which case there must be two arguments if using the +c option and three if using the +s option.

The behavior of a comment/string is specified by a three-character modifier string, which may be passed as an optional argument either to the +c/+s command-line options or to the *#mode comment*/*#mode string* meta-macros. If no modifier string is specified, the default value is "ccc" for comments and "sss" for strings. The first character corresponds to the behavior inside meta-macro calls (including user-macro definitions since these come inside a *#define* meta-macro call), the second character corresponds to the behavior inside user-macro parameters, and the third character corresponds to the behavior outside of any macro call. Each of these characters can take the following values:

- **i**: disable the comment/string specification.
- **c**: comment (neither evaluated nor output).
- **s**: string (the string and its delimiter sequences are output as is).
- **q**: quoted string (the string is output as is, without the delimiter sequences).
- **C**: evaluated comment (macros are evaluated, but output is discarded).
- **S**: evaluated string (macros are evaluated, delimiters are output).

- **Q**: evaluated quoted string (macros are evaluated, delimiters are not output).

Important note: any occurrence of a comment/string start sequence inside another comment/string is always ignored, even if macro evaluation is enabled. In other words, comments/strings cannot be nested. In particular, the 'Q' modifier can be a convenient way of defining a syntax for temporarily disabling all comment and string specifications.

Syntax specification strings should always be provided as C strings, whether they are given as arguments to a *#mode* meta-macro call or on the command-line of a Unix shell. If command-line arguments are given via another method than a standard Unix shell, then the shell behavior must be emulated, i.e. the surrounding "" quotes should be removed, all occurrences of '\\' should be replaced by a single backslash, and similarly '\"' should be replaced by '"'. Sequences like '\n' are recognized by gpp and should be left as is.

Special sequences matching certain subsets of the character set can be used. They are of the form '\x', where x is one of:

- **b**: matches any sequence of one or more spaces or TAB characters ('\b' is identical to ' ').
- **w**: matches any sequence of zero or more spaces or TAB characters.
- **B**: matches any sequence of one or more spaces, tabs or newline characters.
- **W**: matches any sequence of zero or more spaces, tabs or newline characters.
- **a**: an alphabetic character ('a' to 'z' and 'A' to 'Z').
- **A**: an alphabetic character, or a space, tab or newline.
- **#**: a digit ('0' to '9').
- **i**: an identifier character. The set of matched characters is customizable using the *#mode charset id* command. The default setting matches alphanumeric characters and underscores ('a' to 'z', 'A' to 'Z', '0' to '9' and '_').
- **t**: a TAB character.
- **n**: a newline character.
- **o**: an operator character. The set of matched characters is customizable using the *#mode charset op* command. The default setting matches all characters in "+-*/\^<>=`~:.?@#&!%|", except in Prolog mode where '!', '%' and '|' are not matched.
- **O**: an operator character or a parenthesis character. The set of additional matched characters in comparison with '\o' is customizable using the *#mode charset par* command. The default setting is to have the characters in "()[]{}" as parentheses.

Moreover, all of these matching subsets except '\w' and '\W' can be negated by inserting a '!', i.e. by writing '\!x' instead of '\x'.

Note an important distinctive feature of *start sequences*: when the first character of a macro or comment/string start sequence is ' ' or one of the above special sequences, it is not taken to be part of the sequence itself but is used instead as a context check: for example a start sequence beginning with '\n' matches only at the beginning of a line, but the matching newline character is not taken to be part of the sequence. Similarly a start sequence beginning with ' ' matches only if some whitespace is present, but the matching whitespace is not considered to be part of the start sequence and is therefore sent to output. If a context check is performed at the very beginning of a file (or more generally of any body to be evaluated), the result is the same as matching with a newline character (this makes it possible for a cpp-mode file to start with a meta-macro call).

A.5 Evaluation Rules

Input is read sequentially and interpreted according to the rules of the current mode. All input text is first matched against the specified comment/string start sequences of the current mode (except those which are disabled by the 'i' modifier), unless the body being evaluated is the contents of a comment/string whose modifier enables macro evaluation. The most recently defined comment/string specifications are checked for first. Important note: comments may not appear between the name of a macro and its arguments (doing so results in undefined behavior).

Anything that is not a comment/string is then matched against a possible meta-macro call, and if that fails too, against a possible user-macro call. All remaining text undergoes substitution of argument reference sequences by the relevant argument text (empty unless the body being evaluated is the definition of a user macro) and removal of the quote character if there is one.

Note that meta-macro arguments are passed to the meta-macro prior to any evaluation (although the meta-macro may choose to evaluate them, see meta-macro descriptions below). In the case of the *#mode* meta-macro, gpp temporarily adds a comment/string specification to enable recognition of C strings ("...") and prevent any evaluation inside them, so no interference of the characters being put in the C string arguments to *#mode* with the current syntax is to be feared.

On the other hand, the arguments to a user macro are systematically evaluated, and then passed as context parameters to the macro definition body, which gets evaluated with that environment. The only exception is when the macro definition is empty, in which case its arguments are not evaluated. Note that gpp temporarily switches back to the mode in which the macro was defined in order to evaluate it: so it is perfectly safe to change the operating mode between the time when a macro is defined and the time when it is called. Conversely, if a user macro wishes to work with the current mode instead of the one that was used to define it it needs to start with a *#mode restore* call and end with a *#mode save* call.

A user macro may be defined with named arguments (see *#define* description below). In that case, when the macro definition is being evaluated, each named parameter causes a temporary virtual user-macro definition to be created; such a macro may only be called without arguments and simply returns the text of the corresponding argument.

Note that, since macros are evaluated when they are called rather than when they are defined, any attempt to call a recursive macro causes undefined behavior except in the very specific case when the macro uses *#undef* to erase itself after finitely many loop iterations.

Finally, a special case occurs when a user macro whose definition does not involve any arguments (neither named arguments nor the argument reference sequence) is called in a mode where the short user-macro end sequence is empty (e.g. cpp or TeX mode). In that case it is assumed to be an *alias macro*: its arguments are first evaluated in the current mode as usual, but instead of being passed to the macro definition as parameters (which would cause them to be discarded) they are actually appended to the macro definition, using the syntax rules of the mode in which the macro was defined, and the resulting text is evaluated again. It is therefore important to note that, in the case of a macro alias, the arguments actually get evaluated twice in two potentially different modes.

A.6 Meta-macros

These macros are always pre-defined. Their actual calling sequence depends on the current mode; here we use cpp-like notation.

- **#define** x y
  This defines the user macro *x* as *y*. *y* can be any valid gpp input, and may for example refer to other macros. *x* must be an identifier (i.e. a sequence of alphanumeric characters and '_'), unless named arguments are specified. If *x* is already defined, the previous definition is overwritten. If no second argument is given, *x* will be defined as a macro that outputs nothing. Neither *x* nor *y* are evaluated; the macro definition is only evaluated when it is called, not when it is declared.

  It is also possible to name the arguments in a macro definition: in that case, the argument *x* should be a user-macro call whose arguments are all identifiers. These identifiers become available as user-macros inside the macro definition; these virtual macros must be called without arguments, and evaluate to the corresponding macro parameter.

- **#defeval** x y
  This acts in a similar way to *#define*, but the second argument *y* is evaluated immediately. Since user macro definitions are also evaluated each time they are called, this means that the macro *y* will undergo *two* successive evaluations. The usefulness of *#defeval* is considerable, as it is the only way to evaluate something more than once, which can be needed e.g. to force evaluation of the arguments of a meta-macro that normally doesn't perform any evaluation. However since all argument references evaluated at define-time are understood as the arguments of the body in which the macro is being defined and not as the arguments of the macro itself, usually one has to use the quote character to prevent immediate evaluation of argument references.

- **#undef** x
  This removes any existing definition of the user macro *x*.

- **#ifdef** x
  This begins a conditional block. Everything that follows is evaluated only if the identifier *x* is defined, until either a *#else* or a *#endif* statement is reached. Note however that the commented text is still scanned thoroughly, so its syntax must be valid. It is in particular

legal to have the *#else* or *#endif* statement ending the conditional block appear as only the result of a user-macro expansion and not explicitly in the input.

- **#ifndef** x
  This begins a conditional block. Everything that follows is evaluated only if the identifier x is not defined.

- **#ifeq** x y
  This begins a conditional block. Everything that follows is evaluated only if the results of the evaluations of x and y are identical as character strings. Any leading or trailing whitespace is ignored for the comparison. Note that in cpp-mode any unquoted whitespace character is understood as the end of the first argument, so it is necessary to be careful.

- **#ifneq** x y
  This begins a conditional block. Everything that follows is evaluated only if the results of the evaluations of x and y are not identical (even up to leading or trailing whitespace).

- **#else**
  This toggles the logical value of the current conditional block. What follows is evaluated if and only if the preceding input was commented out.

- **#endif**
  This ends a conditional block started by a *#if...* meta-macro.

- **#include** file
  This causes gpp to open the specified file and evaluate its contents, inserting the resulting text in the current output. All defined user macros are still available in the included file, and reciprocally all macros defined in the included file will be available in everything that follows. The include file is looked for first in the current directory, and then, if not found, in one of the directories specified by the *-I* command-line option (or */usr/include* if no directory was specified). Note that, for compatibility reasons, it is possible to put the file name between "" or <>.

  Upon including a file, gpp immediately saves a copy of the current operating mode onto the mode stack, and restores the operating mode at the end of the included file. The included file may override this behavior by starting with a *#mode restore* call and ending with a *#mode push* call. Additionally, when the *-m* command line option is specified, gpp will automatically switch to the cpp compatibility mode upon including a file whose name ends with either '.c' or '.h'.

- **#exec** command
  This causes gpp to execute the specified command line and include its standard output in the current output. Note that this meta-macro is disabled unless the *-x* command line flag was specified, for security reasons. If use of *#exec* is not allowed, a warning message is printed and the output is left blank. Note that the specified command line is evaluated before being executed, thus allowing the use of macros in the command-line. However, the output of the command is included verbatim and not evaluated. If you need the output to be evaluated, you must use *#defeval* (see above) to cause a double evaluation.

- **#eval** expr
  The *#eval* meta-macro attempts to evaluate *expr* first by expanding macros (normal gpp evaluation) and then by performing arithmetic evaluation. The syntax and operator precedence for arithmetic expressions are the same as in C ; the only missing operators are <<, >>, ?: and assignment operators. If unable to assign a numerical value to the result, the returned text is simply the result of macro expansion without any arithmetic evaluation. The only exceptions to this rule are the == and != operators which, if one of the sides does not evaluate to a number, perform string comparison instead (ignoring trailing and leading spaces).

  Inside arithmetic expressions, the *defined(...)* special user macro is also available: it takes only one argument, which is not evaluated, and returns 1 if it is the name of a user macro and 0 otherwise.

- **#if** expr
  This meta-macro invokes the arithmetic evaluator in the same manner as *#eval*, and compares the result of evaluation with the string "0" in order to begin a conditional block. In particular note that the logical value of *expr* is always true when it cannot be evaluated to a number.

- **#mode** keyword ...
  This meta-macro controls gpp's operating mode. See below for a list of *#mode* commands.

The key to gpp's flexibility is the *#mode* meta-macro. Its first argument is always one of a list of available keywords (see below); its second argument is always a sequence of words separated by whitespace. Apart from possibly the first of them, each of these words is always a delimiter or syntax specifier, and should be provided as a C string delimited by double quotes (" "). The various special matching sequences listed in the section on syntax specification are available. Any *#mode* command is parsed in a mode where "..." is understood to be a C-style string, so it is safe to put any character inside these strings. Also note that the first argument of *#mode* (the keyword) is never evaluated, while the second argument is evaluated (except of course for the contents of C strings), so that the syntax specification may be obtained as the result of a macro evaluation.

The available *#mode* commands are:

- **#mode save / #mode push**
  Push the current mode specification onto the mode stack.

- **#mode restore / #mode pop**
  Pop mode specification from the mode stack.

- **#mode standard** name
  Select one of the standard modes. The only argument must be one of: default (default mode); cpp, C (cpp mode); tex, TeX (tex mode); html, HTML (html mode); prolog, Prolog (prolog mode). The mode name must be given directly, not as a C string.

- **#mode user** "s1" ... "s9"
  Specify user macro syntax. The 9 arguments, all of them C strings, are the mode specification for user macros (see the -U command-line option and the section on syntax specification). The meta-macro specification is not affected.

- **#mode meta** {user | "s1" ... "s7"}
  Specify meta-macro syntax. Either the only argument is *user* (not as a string), and the user-macro mode specifications are copied into the meta-macro mode specifications, or there must be 7 string arguments, whose significance is the same as for the -M command-line option (see section on syntax specification).

- **#mode quote** ["c"]
  With no argument or "" as argument, removes the quote character specification and disables the quoting functionality. With one string argument, the first character of the string is taken to be the new quote character. The quote character cannot be alphanumeric nor '_', and cannot be one of the special matching sequences either.

- **#mode comment** [xxx] "start" "end" ["c" ["c"]]
  Add a comment specification. Optionally a first argument consisting of three characters not enclosed in " " can be used to specify a comment/string modifier (see the section on syntax specification). The default modifier is *ccc*. The first two string arguments are used as comment start and end sequences respectively. The third string argument is optional and can be used to specify a string-quote character (if it is "" the functionality is disabled). The fourth string argument is optional and can be used to specify a string delimitation warning character (if it is "" the functionality is disabled).

- **#mode string** [xxx] "start" "end" ["c" ["c"]]
  Add a string specification. Identical to *#mode comment* except that the default modifier is *sss*.

- **#mode nocomment / #mode nostring** ["start"]
  With no argument, remove all comment/string specifications. With one string argument, delete the comment/string specification whose start sequence is the argument.

- **#mode preservelf** { on | off | 1 | 0 }
  Equivalent to the *-n* command-line switch. If the argument is *on* or *1*, any newline or whitespace character terminating a macro call or a comment/string is left in the input stream for further processing. If the argument is *off* or *0* this feature is disabled.

- **#mode charset** { id | op | par } "string"
  Specify the character sets to be used for matching the \o, \O and \i special sequences. The first argument must be one of *id* (the set matched by \i), *op* (the set matched by \o) or *par* (the set matched by \O in addition to the one matched by \o). *"string"* is a C string which lists all characters to put in the set. It may contain only the special matching sequences \a, \A, \b, \B, and \# (the other sequences and the negated sequences are not allowed). When a '-' is found inbetween two non-special characters this adds all characters inbetween (e.g. "A-Z" corresponds to all uppercase characters). To have '-' in the matched set, either put it in first or last position or place it next to a \x sequence.

A.7 Examples

Here is a basic self-explanatory example in standard or cpp mode:

```
#define FOO This is
#define BAR a message.
#define concat #1 #2
concat(FOO,BAR)
#ifeq (concat(foo,bar)) (foo bar)
This is output.
#else
This is not output.
#endif
```

Using argument naming, the *concat* macro could alternately be defined as

```
#define concat(x,y) x y
```

In TeX mode and using argument naming, the same example becomes:

```
\define{FOO}{This is}
\define{BAR}{a message.}
\define{\concat{x}{y}}{\x \y}
\concat{\FOO}{\BAR}
\ifeq{\concat{foo}{bar}}{foo bar}
This is output.
\else
This is not output.
\endif
```

In HTML mode and without argument naming, one gets similarly:

```
<#define FOO|This is>
<#define BAR|a message.>
<#define concat|#1 #2>
<#concat <#FOO>|<#BAR>>
<#ifeq <#concat foo|bar>|foo bar>
This is output.
<#else>
This is not output.
<#endif>
```

The following example (in standard mode) illustrates the use of the quote character:

```
#define FOO This is \
   a multiline definition.
#define BLAH(x) My argument is x
BLAH(urf)
\BLAH(urf)
```

Note that the multiline definition is also valid in cpp and Prolog modes despite the absence of quote character, because '\' followed by a newline is then interpreted as a comment and discarded.

In cpp mode, C strings and comments are understood as such, as illustrated by the following example:

```
#define BLAH foo
BLAH "BLAH" /* BLAH */
'It\'s a /*string*/ !'
```

The main difference between Prolog mode and cpp mode is the handling of strings and comments: in Prolog, a '...' string may not begin immediately after a digit, and a /*...*/ comment may not begin immediately after an operator character. Furthermore, comments are not removed from the output unless they occur in a #command.

The differences between cpp mode and default mode are deeper: in default mode #commands may start anywhere, while in cpp mode they must be at the beginning of a line; the default mode has no knowledge of comments and strings, but has a quote character ('\'), while cpp mode has extensive comment/string specifications but no quote character. Moreover, the arguments to meta-macros need to be correctly parenthesized in default mode, while no such checking is performed in cpp mode.

This makes it easier to nest meta-macro calls in default mode than in cpp mode. For example, consider the following HTML mode input, which tests for the availability of the *#exec* command:

```
<#ifeq <#exec echo blah>|blah
> #exec allowed <#else> #exec not allowed <#endif>
```

There is no cpp mode equivalent, while in default mode it can be easily translated as

```
#ifeq (#exec echo blah
) (blah
)
\#exec allowed
#else
\#exec not allowed
#endif
```

In order to nest meta-macro calls in cpp mode it is necessary to modify the mode description, either by changing the meta-macro call syntax, or more elegantly by defining a silent string and using the fact that the context at the beginning of an evaluated string is a newline character:

```
#mode string QQQ "$" "$"
#ifeq $#exec echo blah
$ $blah
$
\#exec allowed
```

```
#else
\#exec not allowed
#endif
```

Note however that comments/strings cannot be nested ("..." inside $...$ would go undetected), so one needs to be careful about what to include inside such a silent evaluated string.

Remember that macros without arguments are actually understood to be aliases when they are called with arguments, as illustrated by the following example (default or cpp mode):

```
#define DUP(x) x x
#define FOO and I said: DUP
FOO(blah)
```

The usefulness of the *#defeval* meta-macro is shown by the following example in HTML mode:

```
<#define APPLY|<#defeval TEMP|<\##1 \#1>><#TEMP #2>>
<#define <#foo x>|<#x> and <#x>>
<#APPLY foo|BLAH>
```

The reason why *#defeval* is needed is that, since everything is evaluated in a single pass, the input that will result in the desired macro call needs to be generated by a first evaluation of the arguments passed to APPLY before being evaluated a second time.

To translate this example in default mode, one needs to resort to parenthesizing in order to nest the #defeval call inside the definition of APPLY, but need to do so without outputting the parentheses. The easiest solution is

```
#define BALANCE(x) x
#define APPLY(f,v) BALANCE(#defeval TEMP f
TEMP(v))
#define foo(x) x and x
APPLY(\foo,BLAH)
```

As explained above the simplest version in cpp mode relies on defining a silent evaluated string to play the role of the BALANCE macro.

The following example (default or cpp mode) demonstrates arithmetic evaluation:

```
#define x 4
The answer is:
#eval x*x + 2*(16-x) + 1998%x

#if defined(x)&&!(3*x+5>17)
This should be output.
#endif
```

To finish, here are some examples involving mode switching. The following example is self-explanatory (starting in default mode):

```
#mode push
#define f(x) x x
#mode standard TeX
\f{blah}
\mode{string}{"$" "$"}
\mode{comment}{"/*" "*/"}
$\f{urf}$ /* blah */
\define{FOO}{bar/* and some more */}
\mode{pop}
f($FOO$)
```

A good example where a user-defined mode becomes useful is the gpp source of this document (available with gpp's source code distribution).

Another interesting application is selectively forcing evaluation of macros in C strings when in cpp mode. For example, consider the following input:

```
#define blah(x) "and he said: x"
blah(foo)
```

Obviously one would want the parameter x to be expanded inside the string. There are several ways around this problem:

```
#mode push
#mode nostring "\""
#define blah(x) "and he said: x"
#mode pop

#mode quote "‘"
#define blah(x) ‘"and he said: x‘"

#mode string QQQ "$$" "$$"
#define blah(x) $$"and he said: x"$$
```

The first method is very natural, but has the inconvenient of being lengthy and neutralizing string semantics, so that having an unevaluated instance of 'x' in the string, or an occurrence of '/*', would be impossible without resorting to further contorsions.

The second method is slightly more efficient, because the local presence of a quote character makes it easier to control what is evaluated and what isn't, but has the drawback that it is sometimes impossible to find a reasonable quote character without having to either significantly alter the source file or enclose it inside a *#mode push/pop* construct. For example any occurrence of '/*' in the string would have to be quoted.

The last method demonstrates the efficiency of evaluated strings in the context of selective evaluation: since comments/strings cannot be nested, any occurrence of '"' or '/*' inside the '$$' gets output as plain text, as expected inside a string, and only macro evaluation is enabled. Also note that there is much more freedom in the choice of a string delimiter than in the choice of a quote character.

A.8 Advanced Examples

Here are some examples of advanced constructions using gpp. They tend to be pretty awkward and should be considered as evidence of gpp's limitations.

The first example is a recursive macro. The main problem is that, since gpp evaluates everything, a recursive macro must be very careful about the way in which recursion is terminated, in order to avoid undefined behavior (most of the time gpp will simply crash). In particular, relying on a *#if/#else/#endif* construct to end recursion is not possible and results in an infinite loop, because gpp scans user macro calls even in the unevaluated branch of the conditional block. A safe way to proceed is for example as follows (we give the example in TeX mode):

```
\define{countdown}{
  \if{#1}
  #1...
  \define{loop}{\countdown}
  \else
  Done.
  \define{loop}{}
  \endif
  \loop{\eval{#1-1}}
}
\countdown{10}
```

The following is an (unfortunately very weak) attempt at implementing functional abstraction in gpp (in standard mode). Understanding this example and why it can't be made much simpler is an exercise left to the curious reader.

```
#mode string "`" "`" "\\"
#define ASIS(x) x
#define SILENT(x) ASIS()
#define EVAL(x,f,v) SILENT(
  #mode string QQQ "`" "`" "\\"
  #defeval TEMP0 x
  #defeval TEMP1 (
    \#define \TEMP2(TEMP0) f
  )
  TEMP1
  )TEMP2(v)
```

```
#define LAMBDA(x,f,v) SILENT(
  #ifneq (v) ()
  #define TEMP3(a,b,c) EVAL(a,b,c)
  #else
  #define TEMP3(a,b,c) \LAMBDA(a,b)
  #endif
  )TEMP3(x,f,v)
#define EVALAMBDA(x,y) SILENT(
  #defeval TEMP4 x
  #defeval TEMP5 y
  )
#define APPLY(f,v) SILENT(
  #defeval TEMP6 ASIS(\EVA)f
  TEMP6
  )EVAL(TEMP4,TEMP5,v)
```

This yields the following results:

```
LAMBDA(z,z+z)
  => LAMBDA(z,z+z)

LAMBDA(z,z+z,2)
  => 2+2

#define f LAMBDA(y,y*y)
f
  => LAMBDA(y,y*y)

APPLY(f,blah)
  => blah*blah

APPLY(LAMBDA(t,t t),(t t))
  => (t t) (t t)

LAMBDA(x,APPLY(f,(x+x)),urf)
  => (urf+urf)*(urf+urf)

APPLY(APPLY(LAMBDA(x,LAMBDA(y,x*y)),foo),bar)
  => foo*bar

#define test LAMBDA(y,‘#ifeq y urf
y is urf#else
y is not urf#endif
‘)
APPLY(test,urf)
```

```
    => urf is urf

  APPLY(test,foo)
    => foo is not urf
```

A.9 Author

Denis Auroux, e-mail: auroux@math.polytechnique.fr.

Please send me e-mail for any comments, questions or suggestions.

Many thanks to Michael Kifer for valuable feedback and for prompting me to go beyond version 1.0.

Appendix B

Standard Predicates and Functions

B.1 List of Standard Predicates

```
abolish(Name/Arity)
abolish(Name, Arity)
abolish_all_tables
abolish_table_call(Term)
abolish_table_pred(Pred)
abort
abort(Message)
analyze_table(Pred)
arg(Index, Term, Arg)
arg0(Index, Term, Arg)
assert(Clause)
asserta(Clause)
assertz(Clause)
atom(Term)
atomic(Term)
atom_codes(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)
compile(Module)
compile(Module, Options)
compound(Term)
```

```
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)
garbage_collection(Option)
get(Char)
get0(Char)
get_call(Skeleton,Subgoal_Structure_Pointer,Return_Skeleton)
get_calls(Skeleton,Subgoal_Structure_Pointer,Return_Skeleton)
get_calls_for_table(Skeleton,Call,Empty)
get_resudual(Call, Return)
get_returns(Skeleton,Delay_list)
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_absolute_filename(Path)
is_list(Term)
is_charlist(Term)
is_charlist(Term,Size)
keysort(Input, Output)
listing
listing(Predicate)
load_dyn(Module)
load_dync(Module)
ls
module_property(Module, Property)
name(Term, CharList)
nl
nl(Stream)
nodebug
nonvar(Term)
nospy(Predicate_List)
not(Goal)
notrace
number(Term)
number_codes(Number,Character_list)
once(Goal)
op(Precedence, Format, Operator)
open(Stream)
otherwise
phrase(Phrase, List)
phrase(Phrase, List, Remains)
predicate_property(Predicate, Property)
print(Term)                    Currently the same as write/1.
proper_hilog(Term)
put(Char)
read(Term)
read(Stream,Term)
read_canonical(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`	
`set_global_compiler_options(OptionsList)`	
`setof(Elem, Goal, Set)`	
`skip(Char)`	
`sort(Input, Output)`	
`spy(Predicate_List)`	
`statistics`	
`statistics(Number)`	
`structure(Term)`	
`tab(Count)`	
`table_prop(Pred, Type, Value)`	
`table_prop(Pred, Type, OldValue, NewValue)`	
`table_state(Call, State)`	
`tbagof(Elem, Goal, List)`	
`tell(File)`	
`telling(File)`	
`tfindall(Elem, Goal, List)`	
`tilde_expand_filename(File_name,Expanded_Filename)`	
`tnot(Goal)`	
`told`	
`trace`	
`true`	
`tsetof(Elem, Goal, List)`	
`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.
`shell(Command)`	
`shell(Command, Result)`	
`var(Term)`	
`write(Term)`	
`write(Stream,Term)`	
`writeln(Term)`	
`write_prolog(Term)`	
`write_prolog(Stream,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
```

B.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)
```

B.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, [ hilog, dynamic, multifile ])
op(1100,  fy, [ index, ti, ti_off ])
op(1100,  fx, [ ;, table, edb, 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, [ ^ ])
```

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 C

List of Module names

C.1 In syslib

assert	basics	consult	curr_sym	dcg
db	dbclause	dbcmpl	dcg_qsc	debugger
domain	eval	file_op	hilogsym	loader
machine	num_vars	standard	std_sbp	string
tables	term_exp	x_interp	xsb_read	xsb_tok
xsb_writ				

C.2 In cmplib

asm	asm_inst	asm_opt	asmpass2	auxlry
builtin	compile	config	cp_opt	cutcheck
flatten	inprog	listutil	parse	peephole
pre_cond	preproc	printres	gensym	sanity
singlton	spec	suppltab	symtab	tabdef
ti_mod	tp_comp	tp_cond	tp_eval	tp_goal
tp_index	tp_var	tpinline	tprog	useinfer
varproc	writeasm			

Bibliography

[1] J. Alferes, C. Damasio, and L. Pereira. SLX: a top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *International Logic Programming Symp*, pages 424–439, 1994.

[2] J. Alferes, C. Damasio, and L. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 1995.

[3] F. Banchilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*. ACM, 1986.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Logic Programming*, 10(3):255–299, 1991.

[5] D. Boulanger. Fine-grained goal-directed declarative analysis of logic programs. *Proceedings of the International Workshop on Verification, Model Checking and Abstract Interpretation*, 1997. Available through http://www.dsi.unive.it/ bossi/VMCAI.html.

[6] W. Chen, M. Kifer, and D. Warren. HiLog: A foundation for higher-order logic programming. *J. Logic Programming*, 15(3):187–230, 1993.

[7] W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. *J. Logic Programming*, 24(3):161–199, September 1995.

[8] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[9] M. Codish, B. Demoen, and K. Sagonas. Semantics-based program analysis for logic-based languages using XSB. *Springer International Journal of Software Tools for Technology Transfer*, 2(1):29–45, Nov. 1998.

[10] B. Cui, T. Swift, and D. S. Warren. A case study in using preference logic grammars for knowledge representation. Available at http://www.cs.sunysb.edu/~tswift, 1998.

[11] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *Proc. of the 22nd Symposium on Principles of Programming Languages*, pages 247–258. ACM, 1995.

[12] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *ACM PLDI*, pages 117–126, May 1996.

[13] S. Debray. *SB-Prolog System, Version 3.0, A User Manual*, 1988.

[14] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLILP'98, Held Jointly with the 6th International Conference, ALP'98*, number 1490 in LNCS, pages 21–35, Pisa, Italy, Sept. 1998. Springer.

[15] B. Demoen and K. Sagonas. Memory Management for Prolog with Tabling. In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pages 97–106, Vancouver, B.C., Canada, Oct. 1998. ACM Press.

[16] B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. In G. Gupta, editor, *Practical Aspects of Declarative Languages: First International Workshop*, number 1551 in LNCS, pages 106–121, San Antonio, Texas, Jan. 1999. Springer.

[17] S. Dietrich. *Extension Tables for Recursive Query Evaluation.* PhD thesis, SUNY at Stony Brook, 1987.

[18] J. Freire, T. Swift, and D. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*, 1998.

[19] J. Freire, T. Swift, and D. Warren. A formal framework for scheduling in SLG. In *International Workshop on Tabling in Parsing and Deduction*, 1998.

[20] ISO working group JTC1/SC22. Prolog international standard. Technical report, International Standards Organization, 1995.

[21] D. Kemp and R. Topor. Completeness of a top-down query evaluation procedure for stratified databases. In *Logic Programming: Proc. of the Fifth International Conference and Symposium*, pages 178–194, 1988.

[22] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, July 1995.

[23] M. Kifer and V. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. Logic Programming*, 12(4):335–368, 1992.

[24] R. Larson, D. S. Warren, J. Freire, and K. Sagonas. *Syntactica.* MIT Press, 1995.

[25] R. Larson, D. S. Warren, J. Freire, K. Sagonas, and P. Gomez. *Semantica.* MIT Press, 1996.

[26] J. Leite. Logic program updates. Technical report, Universidade Nova de Lisboa, 1997.

[27] T. Lindholm and R. O'Keefe. Efficient implementation of a defensible semantics for dynamic PROLOG code. In *Proceedings of the International Conference on Logic Programming*, pages 21–39, 1987.

[28] X. Liu, C. R. Ramakrishnan, and S. Smolka. Fully local and efficient evaluation of alternating fixed points. In *TACAS 98: Tools and Algorithms for Construction and Analysis of Systems*, pages 5–19. Springer-Verlag, 1998.

[29] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[30] I. Niemela and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.

[31] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS*, pages 11–21, 1989.

[32] Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Proceedings of CAV 97*, 1997.

[33] P. Rao, I. Ramakrishnan, K. Sagonas, T. Swift, and D. Warren. Efficient table access mechanisms for logic programs. *Journal of Logic Programming*, 38(1):31–54, Jan. 1999.

[34] M. Rules. Diagnostica. Technical report, Medicine Rules, 1999. http://medicinerules.com.

[35] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM TOPLAS*, 20(3):586 – 635, May 1998.

[36] K. Sagonas, T. Swift, and D. Warren. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conference*. ACM, 1994.

[37] K. Sagonas, T. Swift, and D. Warren. An abstract machine for computing the well-founded semantics. In *Joint International Conference and Symposium on Logic Programming.*, pages 274–289, 1996.

[38] K. Sagonas, T. Swift, and D. S. Warren. The limits of fixed-order computation. *Theoretical Computer Science*, 2000. To apperar. Preliminary version apeared in International Workshop on Logic and Databases, LNCS.

[39] K. Sagonas and D. Warren. Efficient execution of HiLog in WAM-based Prolog implementations. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 349–363. MIT Press, June 1995.

[40] H. Seki. On the power of Alexandrer templates. In *Proc. of 8th PODS*, pages 150–159. ACM, 1989.

[41] T. Swift. A new formulation of tabled resolution with delay. In *Recent Advances in Artifiial Intelligence*, 1999. Available at http://www.cs.sunysb.edu/~tswift.

[42] T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 1999. To Appear. Available at http://cs.sunysb.edu/ tswift.

[43] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, 1986.

[44] A. van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

[45] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

[46] A. Walker. Backchain iteration: Towards a practical inference method that is simple enough to be proved terminating, sound, and complete. *J. Automated Reasoning*, 11(1):1–23, 1993. Originally formulated in New York University TR 34, 1981.

[47] D. S. Warren. Programming the PTQ grammar in XSB. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 217–234. Kluwer Academic Publishers, 1995.

[48] J. Xu. *The PSB-Prolog User Manual*, 1990.