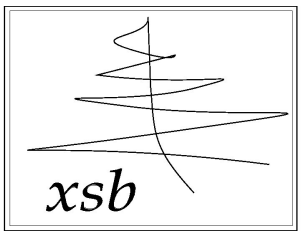
The XSB System
Version 3.8.x
Volume 1: Programmer's Manual



Theresa Swift *David S. Warren*

Konstantinos Sagonas
Juliana Freire
Prasad Rao
Baoqiu Cui
Ernie Johnson
Luis de Castro
Rui F. Marques
Diptikalyan Saha
Steve Dawson
Michael Kifer

October 16, 2017

Credits

Day-to-day care and feeding of XSB including bug fixes, ports, and configuration management is currently done by David Warren and Theresa Swift with the help of Michael Kifer and others. In the past Kostis Sagonas, Prasad Rao, Steve Dawson, Juliana Freire, Ernie Johnson, Baoqiu Cui, Bart Demoen and Luis F. Castro have provided tremendous help.

In Version 3.8, the core engine development of the SLG-WAM has been mainly implemented by Theresa Swift, David Warren, Kostis Sagonas, Prasad Rao, Juliana Freire, Ernie Johnson, Luis Castro and Rui Marques. The breakdown, very roughly, was that Theresa Swift wrote the initial tabling engine, the SLG-WAM, and its built-ins; and leads the current development of the tabling subsystem. Prasad Rao reimplemented the engine's tabling subsystem to use tries for variant-based table access and Ernie Johnson extended and refactored these routines in a number of ways, including adding call subsumption. Kostis Sagonas implemented most of tabled negation. Juliana Freire revised the table scheduling mechanism starting from Version 1.5.0 to create the batched and local scheduling that is currently used. Baoqiu Cui revised the data structures used to maintain delay lists, and added attributed variables to the engine. Luis Castro rewrote the emulator to use jump tables and wrote a heap-garbage collector for the SLG-WAM. Rui Marques was responsible for the concurrency control algorithms used for shared tables, and mainly responsible for making the XSB engine multi-threaded. The incremental table maintenance subsystem was designed and first implemented by Diptikalyan Saha, and its design and development has been continued by Theresa Swift. Answer subsumption was written by David Warren and Theresa Swift. David Warren implemented hash-consed, or "interned" tables. Call abstraction and answer abstraction (restraint) were written by Theresa Swift.

Other engine work includes the following. Memory expansion code for WAM stacks was written by Ernie Johnson, Bart Demoen and David S. Warren. Heap garbage collection was written by Luis de Castro, Kostis Sagonas and Bart Demoen. Atom space garbage collection was written by David Warren; table garbage collection was written by Theresa Swift based in part on space reclamation code written by Prasad Rao. Rui Marques 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; it significantly revised by David S. Warren, who added alternative, multiple, and star indexing and by Theresa Swift who implemented dynamic clause

garbage collection. Trie assert/retract code, and trie interning code was written by Prasad Rao. Neng-fa Zhou, Theresa Swift and David Warren upgraded XSB from ASCII to the character sets UTF-8, C1253, and LATIN-1. The current version of `findall/3` was re-written from scratch by Bart Demoen, as was XSB's original throw and catch mechanism. 64-bit floats were added by Charles Rojo. The interface from C to Prolog and DLL interface were implemented by David Warren and extended to multi-threading by Theresa Swift; the interface from Prolog to C (foreign language interface) was developed by Jiyang Xu, Kostis Sagonas, Steve Dawson and David Warren.

In terms of core system Prolog code, Kostis Sagonas was responsible for HiLog compilation and associated built-ins as well as coding or revising many standard predicates. Steve Dawson implemented Unification Factoring. The revision of XSB's I/O into ISO-compatible streams was done by Michael Kifer and Theresa Swift. The `auto_table` and `suppl_table` directives were written by Kostis Sagonas. The DCG expansion module was written by Kostis Sagonas for non-tabled code and by Baoqiu Cui, David Warren and Theresa Swift for tabled code. The handling of the `multifile` directive was written by Baoqiu Cui and David Warren. C.R. Ramakrishnan wrote the mode analyzer for XSB. Michael Kifer implemented the `storage` module. The multi-threaded API was written by Theresa Swift and Rui Marques. Walter Wilson has written several of XSB's library predicates for tabling. Paulo Moura has added several predicates to make XSB more consistent with other Prologs.

Michael Kifer has been in charge of XSB's installation procedures, rewriting parts of the XSB code to make XSB configurable with GNU's Autoconf, implementing XSB's package system, and integrated GPP with XSB's compiler. GPP, the source code preprocessor used by XSB, was written by Denis Auroux, who also wrote the GPP manual reproduced in Appendix A.

The starting point of XSB (in 1990) was PSB-Prolog 2.0 by Jiyang Xu and David Warren. 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, to Richard O'Keefe, who contributed the Prolog code for the Prolog reader and the C code for the tokenizer, to Ciao Prolog whose `write_term/[2,3]` we use, and to SWI Prolog for their CLP(R) package.

... Now what did I forget this time ?

Contents

1 Introduction **1**

1.1 Using This Manual 7

2 Getting Started with XSB **8**

2.1 Installing XSB under UNIX 8

2.1.1 Possible Installation Problems 12

2.2 Installing XSB under Windows 13

2.2.1 Using Cygwin32 and Cygwin64 13

2.2.2 Using Microsoft Visual C++ 14

2.3 Invoking XSB 16

2.4 Compiling XSB programs 17

2.5 Sample XSB Programs 17

2.6 Exiting XSB 19

3 System Description **20**

3.1 Entering and Exiting XSB from the Command Line 20

3.2 The System and its Directories 21

3.3 How XSB Finds Files: Source File Designators 22

3.4 The Module System of XSB 24

3.4.1 How the Compiler Determines the Module of a Term 28

3.4.2 Atoms and 0-Ary Structure Symbols 29

3.4.3 Dynamic Loading and How XSB Finds Code Files 30

3.4.4 Consulting a Module . . . 30
3.4.5 Usage Inference and the Module System . . . 31
3.4.6 Using Import to Load Usermod Source Files . . . 31
3.4.7 Parameterized Modules in XSB . . . 33
3.5 Standard Predicates in XSB . . . 35
3.6 The Dynamic Loader and its Search Path . . . 36
3.6.1 Changing the Default Search Path and the Packaging System 36
3.6.2 Dynamically loading predicates in the interpreter . . . 39
3.7 Command Line Arguments . . . 40
3.7.1 Command-line Options . . . 42
3.7.2 General Flags . . . 42
3.7.3 Memory Management Flags . . . 45
3.8 Memory Management . . . 45
3.9 Compiling, Consulting, and Loading . . . 47
3.9.1 Static Code . . . 47
3.9.2 Dynamic Code . . . 49
3.9.3 The multifile directive . . . 50
3.10 The Compiler . . . 50
3.10.1 Invoking the Compiler . . . 50
3.10.2 Compiler Options . . . 52
3.10.3 Specialization . . . 60
3.10.4 Compiler Directives . . . 62
3.10.5 Conditional Compilation . . . 64
3.10.6 Inline Predicates . . . 71
3.11 A Note on ISO Compatibility . . . 71

4 Syntax **73**
4.1 Terms . . . 73
4.1.1 Integers . . . 73

4.1.2 Floating-point Numbers . . . 75
4.1.3 Atoms . . . 75
4.1.4 Variables . . . 77
4.1.5 Compound Terms . . . 77
4.1.6 Lists . . . 79
4.2 From HiLog to Prolog . . . 80
4.3 Operators . . . 82

5 Using Tabling in XSB: A Tutorial Introduction 86
5.1 Tabling in the Context of a Prolog System . . . 87
5.2 Definite Programs . . . 87
5.2.1 Call Variance vs. Call Subsumption . . . 91
5.2.2 Tabling with Interned Terms . . . 94
5.2.3 Table Scheduling Strategies . . . 95
5.2.4 Interaction Between Prolog Constructs and Tabling . . . 97
5.2.5 Potential Pitfalls in Tabling . . . 100
5.3 Normal Programs . . . 102
5.3.1 Stratified Normal Programs . . . 102
5.3.2 Non-stratified Programs . . . 106
5.3.3 On Beyond Zebra: Implementing Other Semantics for Non-stratified Programs . . . 110
5.4 Answer Subsumption . . . 113
5.4.1 Types of Answer Subsumption . . . 113
5.4.2 Examples of Answer Subsumption . . . 115
5.4.3 Term-Sets . . . 118
5.5 Tabling for Termination . . . 122
5.5.1 Term Size Abstraction in XSB . . . 124
5.5.2 Subgoal Abstraction . . . 125
5.5.3 XSB's Approach to Bounded Rationality . . . 126

5.6 Incremental Table Maintenance . . . 130
5.6.1 Transparent Incremental Tabling . . . 131
5.6.2 Updating in a Three-Valued Logic . . . 133
5.6.3 Incremental Tabling using Interned Tries . . . 135
5.6.4 Abstracting the IDG for Better Performance . . . 136
5.6.5 Summary and Implementation Status . . . 138
5.6.6 Predicates for Incremental Table Maintenance . . . 138
5.7 Compatibility of Tabling Modes and Predicate Attributes . . . 144
5.8 A Weaker Semantics for Tabling . . . 144

6 Standard and General Predicates 148
6.1 Input and Output . . . 148
6.1.1 I/O Streams in XSB . . . 148
6.1.2 Character Sets in XSB . . . 151
6.1.3 Predicates for ISO Streams . . . 151
6.1.4 DEC-IO Style File Handling . . . 158
6.1.5 Character I/O . . . 161
6.1.6 Term I/O . . . 167
6.1.7 Special I/O . . . 176
6.2 Interactions with the Operating System . . . 182
6.2.1 The `path_sysop/2` interface . . . 186
6.3 Evaluating Arithmetic Expressions through `is/2` . . . 188
6.3.1 Evaluable Functors for Arithmetic Expressions . . . 189
6.4 Convenience . . . 193
6.5 Negation and Control . . . 193
6.6 Unification and Comparison of Terms . . . 198
6.6.1 Sorting of Terms . . . 203
6.7 Meta-Logical . . . 205
6.8 Cyclic Terms . . . 220

6.8.1 Unification with and without Occurs Check 220
6.8.2 Cyclic Terms 221
6.9 Manipulation of Atomic Terms 223
6.10 All Solutions and Aggregate Predicates 236
6.11 Meta-Predicates 241
6.11.1 Timed Calls and Co-routining 246
6.12 Information about the System State 252
6.13 Execution State 271
6.14 Asserting, Retracting, and Other Database Modifications 281
6.14.1 Reading Dynamic Code from Files 292
6.14.2 The `storage` Module: Associative Arrays and Backtrackable Updates 296
6.15 Tabling Declarations and Builtins 299
6.15.1 Declaring and Modifying Tabled Predicates 300
6.15.2 Predicates for Table Inspection 302
6.15.3 Predicates for Table Inspection: Lower-level 309
6.15.4 Abolishing Tables and Table Components 314
6.15.5 Indexing using Tables 324

7 Multi-Threaded Programming in XSB 326
7.1 Getting Started with Multi-Threading 326
7.2 Communication among Threads 328
7.3 Thread Statuses: Joinable and Detached Threads 331
7.4 Prolog Message Queues 333
7.5 Thread Cancellation and Signalling 335
7.6 Performance and other Considerations 337
7.7 Examples of Multi-Threaded Programs in XSB 337
7.8 Configuring the Multi-threaded Engine under Windows 338
7.9 Predicates for Multi-Threading 341

7.9.1 Predicates for Thread Synchronization and Communication . . 348

8 Storing Facts in Tries 356

8.1 Examples of Using Tries 358
8.2 Space Management for Tries 359
8.3 Predicates for Tries 360
8.4 Low-level Trie Manipulation Utilities 367
8.4.1 A Low-Level API for Interned Tries 367

9 Hooks 371

9.1 Adding and Removing Hooks 371
9.2 Hooks Supported by XSB 372

10 Debugging and Profiling 374

10.1 Prolog-style Tracing and Debugging 374
10.1.1 Control of Prolog-Style Tracing and Debugging 379
10.2 Trace-based Execution Analysis through Forest Logging 380
10.2.1 Tracing a tabled evaluation through forest logging 381
10.2.2 Analyzing the log; seeing the forest through the trees 387
10.2.3 Discussion 394
10.2.4 Predicates for Forest Logging 394
10.3 Inspecting a Tabled Derivation 395
10.3.1 Inspecting Tables with `table_dump` 396
10.3.2 Inspection Predicates for Dependency Graphs 399
10.3.3 Summary: Inspection Predicates 412
10.3.4 Setting Tripwires on Tabled Derivations 413

11 Definite Clause Grammars 423

11.1 General Description 423
11.2 Translation of Definite Clause Grammar rules 425

11.2.1 Definite Clause Grammars and Tabling 427
11.3 Definite Clause Grammar predicates 428
11.4 Two differences with other Prologs 432

12 Exception Handling **434**
12.1 The Mechanics of Exception Handling 434
12.1.1 Exception Handling in Non-Tabled Evaluations 434
12.1.2 Exception Handling in Tabled Evaluation 439
12.2 XSB's Standard Format for Errors 441
12.2.1 Error Tags 442
12.2.2 XSB-Specific Information in Error Terms 443
12.3 Predicates to Throw and Handle Errors 444
12.3.1 Predicates to Throw Errors 444
12.3.2 Predicates used in Handling Errors 446
12.4 Convenience Predicates 447
12.5 Backtraces 449

13 Foreign Language Interface **451**
13.1 Foreign Language Modules 452
13.2 Lower-Level Foreign Language Interface 453
13.2.1 Context Parameters 457
13.2.2 Exchanging Basic Data Types 458
13.2.3 Exchanging Complex Data Types 459
13.3 Foreign Modules That Call XSB Predicates 470
13.4 Foreign Modules That Link Dynamically with Other Libraries 471
13.5 Higher-Level Foreign Language Interface 473
13.5.1 Declaration of high level foreign predicates 473
13.6 Compiling Foreign Modules on Windows and under Cygwin 476
13.7 Functions for Use in Foreign Code 477

14 Embedding XSB in a Process **480**
14.1 Calling XSB from C . . . 481
14.2 Examples of Calling XSB . . . 482
14.2.1 The XSB API for the Sequential Engine Only . . . 482
14.2.2 The General XSB API . . . 486
14.2.3 Managing Multiple XSB Threads through the API . . . 490
14.2.4 Calling Multiple XSB Threads using Multiple C Threads . . . 492
14.3 A C API for XSB . . . 494
14.3.1 Initializing and Closing XSB . . . 494
14.3.2 Passing Commands to XSB . . . 496
14.3.3 Querying XSB . . . 497
14.3.4 Obtaining Information about Errors . . . 502
14.3.5 Thread Management from Calling Programs . . . 503
14.4 The Variable-length String Data Type . . . 504
14.5 Passing Data into an XSB Module . . . 506
14.6 Creating an XSB Module that Can be Called from C . . . 507

15 Library Utilities **509**
15.1 List Processing . . . 509
15.1.1 Processing Comma Lists . . . 512
15.2 Attributed Variables . . . 512
15.2.1 Low-level Interface . . . 513
15.3 `constraintLib`: a library for CLP . . . 517
15.4 Formatted Output . . . 519
15.5 Low-level Atom Manipulation Predicates . . . 522
15.6 Script Writing Utilities . . . 524
15.6.1 Communication with Subprocesses . . . 525
15.7 Socket I/O . . . 533
15.8 Arrays . . . 540

15.9 The Profiling Library 541
15.10Gensym 544
15.11Random Number Generator 544
15.12Loading Delimiter-Separated Files 546
15.13Scanning in Prolog 547
15.14XSB Lint 548
15.15"Pure" Meta-programming in XSB with prolog_db.P 551
15.16Range Trees 552
15.17Miscellaneous Predicates 556

A GPP - Generic Preprocessor 558

A.1 Description 558
A.2 Syntax 559
A.3 Options 559
A.4 Syntax Specification 563
A.5 Evaluation Rules 567
A.6 Meta-macros 568
A.7 Examples 573
A.8 Advanced Examples 578
A.9 Author 580

Chapter 1

Introduction

XSB is a research-oriented, commercial-grade Logic Programming system for Unix and Windows-based platforms. In addition to providing nearly all functionality of ISO-Prolog, XSB includes the following features:

- Evaluation of queries according to the Well-Founded Semantics [88] through full SLG resolution (tabling with negation). XSB's tabling implementation supports incremental tabling, as well as call and answer subsumption.
- A fully multi-threaded engine with thread-shared static code, and that allows dynamic code and tables to be thread-shared or thread-private. This engine fully supports the draft ISO standard for multi-threading [39].
- Constraint handling for tabled programs based on an engine-level implementation of annotated variables and various costraint packages, including `clpqr` for handling real constraints, and `bounds` a simple finite domain constraint library.
- A package for Constraint Handling Rules [32] which can be used to implement user-written constraint libraries.
- A variety of indexing techniques for asserted code including variable-depth indexing on several alternate arguments, fixed-depth indexing on combined arguments, trie-indexing.
- A set of mature packages, to extend XSB to evaluate F-logic [43] through the *FLORA-2* package (distributed separately from XSB), to model check concurrent systems through the *XMC* system, to manage ontologies through the *Cold*

Dead Fish package, to support literate programming through the `xsbdoc` package, and to support answer set programming through the `XASP` package among other features.

- A number of interfaces to other software systems, such a C, Java, Perl, ODBC, SModels [60], and Oracle.
- Fast loading of large files by the `load_dync` predicate, and by other means.
- A compiled HiLog implementation;
- Backtrackable updates through XSB's `storage` module that support the semantics of transaction logic [6].
- Extensive pattern matching packages, and interfaces to `libwww` routines, all of which are especially useful for Web applications.
- A novel transformation technique called *unification factoring* that can improve program speed and indexing for compiled code;
- Macro substitution for Prolog files via the `xpp` preprocessor (included with the XSB distribution).
- Preprocessors and Interpreters so that XSB can be used to evaluate programs that are based on advanced formalisms, such as extended logic programs (according to the Well-Founded Semantics [2]); Generalized Annotated Programs [44].
- Source code availability for portability and extensibility under the GNU General Public Library License.

Though XSB can be used as a Prolog system, 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. We briefly discuss some of these features; others are discussed in Volumes 1 and 2 of the XSB manual, as well as the manuals for various XSB packages such as FLORA, XMC, Cold Dead Fish, xsbdoc, and XASP.

Well-Founded Semantics To understand the implications of SLG resolution [16], 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:

```
:- 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 3.10.5). This power to solve recursive queries has proven very useful in a number of areas, including deductive databases, language processing [45, 46], program analysis [22, 17, 7], model checking [63] and diagnosis [33]. 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 [86] (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 *[88]. 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 program should be that the barber shaves the mayor, but the case of the barber is trickier. If we conclude that the barber does not shave 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 program. 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.*

Multi-threading From Version 3.0 onward, XSB has been thoroughly revised to support multi-threading using POSIX or Windows threads. Detached XSB threads can be created to execute specific tasks, and these threads will exit when the query succeeds (or fails, or throws an exception) and all thread memory reclaimed. While a thread's execution state is, of course, private, it shares many resources with other threads, such as static code and I/O streams. Dynamic code and tables can be either thread-shared or thread-private by default or by explicit declaration.

Constraint Support XSB supports logic-based constraint handling at a low level through attributed variables and associated packages (e.g. `setarg/3`). In addition, constraints may be handled through Constraint Handling Rules. Constraint logic programs that use attributed variables may be tabled; those that use Constraint Handling Rules may be efficiently tabled if the `CHRd` package is used. Constraint programming in XSB is mainly covered in Volume 2.

Indexing Methods Data oriented applications may 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 group 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 depth-first, left-to-right traversal through each clause. Representing data in this way allows discrimination of information nested arbitrarily deep within clauses. Advantages of both kinds of indexing can be combined via *star-indexing*. Star-indexing indicates that up to the first 5 fields in an argument will be used for indexing (the ordering of the fields is via a depth-first traversal). For instance, `index(p/4,[*(4),3,2+1])` acts as above, but looks within 4th argument of `p/4` before examining the outer functor of argument 3 (and finally examining the outer functors of arguments 2 and 1 together. Using such indexing, XSB routinely performs efficiently intensive analyses of in-memory knowledge bases with millions of highly structured facts. Indexing techniques for asserted code are covered in Section 6.14.

Interfaces 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 also inter-communicate with Java through the InterProlog interface [1] or using YJXSB. Within InterProlog, XSB and Java can be linked either through Java's JNI interface, or through sockets. XSB can access external data in a variety of ways: through an ODBC interface, through an Oracle interface, or through a variety of mechanisms to read data from flat files. These interfaces are all described in Volume 2 of this manual.

Fast Loading of Code 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 [69]. 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, even when some of the sophisticated indexing mechanisms described above are employed. 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 ISO-semantics of assert [51].

HiLog XSB also supports HiLog programming [14, 72]. 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)`. XSB supports reading and writing of HiLog terms, converting them to or from internal format as necessary (see Section 4.2). Special meta-logical standard predicates (see Section 6.7) 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

[1] InterProlog is available at www.declarativa.com/InterProlog/default.htm.

of compiled Prolog. For more information about the compilation scheme for HiLog employed in XSB see [72].

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

```
:- hilog closure.
:- table apply/3.

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

as long as the underlying relations (the predicate symbols to which R will be unified) are also declared as Hilog. For example, if `a/2` were a binary relation to which the `closure` predicate would be applied, then the declaration `:- hilog a.` would also need to be included.

Unification Factoring 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 [21], and the XSB directives needed to use it are covered in Section 3.10.

XSB Packages Based on these features, a number of sophisticated packages have been implemented using XSB. For instance, XSB supports a sophisticated object-oriented interface called *Flora*. *Flora* (`http://flora.sourceforge.net`) is available as an XSB package and is described in its own manual, available from the same site from which XSB was downloaded. Another package, XMC `http://www.cs.sunnysb.edu/~lmc` depends on XSB to perform sophisticated model-checking of concurrent systems. Within the XSB project, the Cold Dead Fish package supports maintenance of, and reasoning over ontologies; xsbdoc supports literate programming in XSB, and XASP provides an interface to Smodels to support Answer Set programming. XSB packages also support Perl-style pattern matching and POSIX-style pattern matching. In addition, experimental preprocessing libraries currently supported are Extended logic programs (under the well-founded semantics), and Annotated Logic Programs. These latter libraries are described in Volume 2 of this manual.

1.1 Using This Manual

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 3.10.5 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 3.8) 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 without difficulty on the web.

While some of Version 3.8 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 accommodate 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, 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 created in the current directory. All XSB files will be located in that subdirectory. In the rest of this manual, we 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.

The configuration script allows many different options to be specified. A full listing can be obtained by typing `$XSB_DIR/build/configure -help`.

Type of Machine. The configuration script automatically detects your machine and OS type, and builds XSB accordingly. On 64-bit platforms, the default compilation of XSB will reflect the default for the C compiler (e.g. gcc) on that platform. 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 commands – or configure with different parameters.

The configuration files for different architectures reside in different directories, and there is no danger of an architecture conflict. In fact, 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!

If XSB is being built on a Windows machine in which Cygwin is installed, Cygwin and Windows are treated as separate operating systems, as their APIs are completely different. On such a machine, XSB can be built either for Cygwin or Windows.

Choice of the C Compiler and compiler-related options On Unix systems, XSB is developed and tested mainly using gcc. Accordingly, 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 for particular platforms or `gcc` may not have been installed; in which case you would have to give `configure` an additional directive `-with-cc` (or `-with-acc`). If you must use some special compiler, use `-with-cc=your-own-compiler`. You can also use the `-with-optimization` option to change the default C compiler optimization level. (or `-disable-optimization` to disable all compiler optimizations). `-enable-debug` is mainly a devlopment option that allows XSB to be debugged using gdb – there are many other compiler-based options options. Type `configure -help` to see them all. Also see the file `$XSB_DIR/INSTALL` for more details.

Word Size XSB's configuration script checks whether the default compilation mode of a platform is 32- or 64-bits, and will build a version of XSB accoringly. Some platforms, however, support both 32-bit and 64-bit compilation. On such a

platform, a user can explicitly specify the type of compilation using the options `with-bits32` and `with-bits64`.

XSB and Site-specific Information Using the option `-prefix=PREFIX` installs architecture-independent files in the directory `PREFIX`, e.g. `/usr/local`, which can be useful if XSB is to be shared at a site. Using the option `-site-prefix=DIR` installs site-specific libraries in `DIR/site`. Other options indicate directories in which to search for site-specific static and dynamic libraries, and for include files.

Multi-threading Version 3.0 of XSB was the first version that supports multi-threading. On some platforms, the multi-threaded engine is slightly slower than the single-threaded engine, mostly due to its need for concurrency control. To obtain the benefits of multiple threads on a platform that supports either POSIX or Windows threads (i.e. nearly all platforms) users must configure XSB with the directive `enable-mt` (see Section 7.8 for instructions specific to Windows. The multi-threaded engine works with other configuration options, multi-threading can be compiled with batched or local scheduling, with the ODBC or InterProlog interfaces, and so on.

Interfaces Certain interfaces must be designated at configuration time, including those to Oracle, ODBC, Smodels, Tck/Tk, and Libwww. However, the XSB-calling-C interface interface does not need to be specified at configuration time. If you wish to use the InterProlog Java interface that is based on JNI, you must specify this at configuration time; otherwise if you wish to use the sockets-based InterProlog interface, it does not need to be specified at configuration time. See Volume 2 and the InterProlog site `www.declarativa.com` for details of specific interfaces

While the XSB configuration mechanism can detect most include and library paths, use of certain interfaces may require information about particular directories. In particular the `-with-static-libraries` option might be needed if compiling with support for statically linked packages (such as Oracle) or if your standard C libraries are in odd places. Alternately, dynamic libraries on odd places may need to be specified at configuration time using the `-with-dynamic-libraries` option. and finally, the `-with-includes` option might be needed if your standard header files (or your `jni.h` file) are in odd places, or if XSB is compiled with ODBC support. Type `configure -help` for more details.

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. Local Evaluation ensures

that, whenever possible, subgoals are fully evaluated before there answers are returned, and provides superior behavior for programs in which tabled negation is used. Batched Evaluation evaluates queries to reduce the time to the first answer of a query. Both evaluation methods can be useful for different programs. Since Version 2.4, Local Evaluation has been the default evaluation method for XSB. Batched Evaluation can be chosen via the `-enable-batched-scheduling` configure option. Detailed explanations of the scheduling strategies can be found in [30], and further experimentation in [12].

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.

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 (*.xwam) 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`.

XSB on 64-bit platforms XSB has been fully tested on 64-bit Debian Linux, 64-bit and Mac OS X. However, the sockets library may have problems in Version 3.8. If this limitation prove a problem, please contact `xsb-development@lists.sourceforge.net` [1].

Typically, if the 64-bit system generates 32-bit code by default, XSB will run just as in 32-bit mode (including 64-bit floats). 64-bit compilation can be forced for XSB by configuring with the option `-with-bits64`, and in a similar manner 32-bit compilation can be forced with the option `-with-bits32`. Users who employ either option should be aware of issues that may arise when linking XSB to external C code.

- When XSB calls C code the C file must have been compiled with the same memory option as XSB. This is done automatically if the C file is compiled via a call from XSB's compiler, but must be handled by the user otherwise. For instance, if XSB were configured `-with-bits32` on a 64-bit machine defaulting to 64-bits, then C files called by XSB require the `-m32` option in `gcc` (if not compiled by XSB).
- The appropriate memory option must be used when embedding XSB into a C or Java process. For instance, if a XSB is to be linked into a 32-bit application on a 64-bit platform defaulting to 64-bits, XSB must be configured `-with-bits32`, and the linking of `xsb.o/so` to the calling program must specify `-m32`.

2.2 Installing XSB under Windows

2.2.1 Using Cygwin32 and Cygwin64

This is easy: just follow the Unix instructions. XSB can be built under CygWin64 or CygWin32, but in the latter case CygWin32 must be installed on a 32 bit version of Windows. XSB *cannot* be built under CygWin32 if the latter is installed on a 64 bit Windows.

Note: XSB is not fully functional under Cygwin—external C modules cannot be linked and so several packages will not work.

[1] 64-bit XSB was broken in a recent releases prior to Version 3.1 because for a time the developers did not have access to a 64-bit machine.

2.2.2 Using Microsoft Visual C++

1. Check XSB out from SVN:
   `svn checkout svn://svn.code.sf.net/p/xsb/src/trunk xsb-src`
2. Compile XSB as described below. This requires that Microsoft Visual Studio is installed.
3. After compiling XSB, it is OK to move it to some other place, if needed. However, make sure that the entire directory tree is moved — the XSB executable looks for the files it needs relatively to its current position in the file system.

The first thing is to ensure that Microsoft Visual Studio that includes a C++ compiler, so *download the free of charge Microsoft Visual Studio, Community Edition* from

```
https://www.visualstudio.com/vs/community/
```

Make sure you *select the C++ compiler* as one of the additional components to include (e.g., choose "Desktop development with C++"). The installer will place the studio in `C:\Program Files\Microsoft Visual Studio\`.

One way to compile XSB under Windows is to use the automatic installer:

```
cd $XSB_DIR
java -jar InstallXSB.jar
```

where `$XSB_DIR` is the XSB's instalation directory, and follow the prompts. The trickiest of these prompts will ask you to provide the full file name of the studio's settings batch file. For Visual Studio 2017, Community Edition, that file is `vcvarsx86_amd64.bat` (for 64 bit apps) or `vcvars32.bat` (for 32 bit apps), located in the directory

`C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build`

In other versions of the studio, that file is elsewhere and can be found using the Windows search widget. For instance, in the 2015 version of the studio, that directory is `C:\Program Files (x86)\Microsoft Visual Studio 2015\VC\bin`

If the automatic method just described does not work or if one needs customized installation, one has to compile XSB the "hard way," as described below.

1. *Find the settings file, which you need to execute in a command window in order to set the compilation environment*, as described above.

2. Open a Windows command prompt window and drag the appropriate batch file (`vcvarsx86_amd64.bat` or `vcvars32.bat`) into it. Type `<Enter>` to execute that batch file.

3. `cd $XSB_DIR\build`

4. To compile XSB as a 64 bit application, use the following command, where the items in square brackets are optional and usually can be dropped:
   `makexsb64 ["CFG=opt"] ["ORACLE=yes"] ["MY_LIBRARY_DIRS=libs"] ["MY_INCLUDE_DIRS=opts"]`

   - The options for `CFG` are: *release* (default) or *debug*. The latter is used when you want to compile XSB with debugging enabled.
   - The `ORACLE` parameter (default is "no") compiles XSB with native support for Oracle DBMS. If `ORACLE` is specified, you **must** also specify the necessary Oracle libraries using the parameter `SITE_LIBS`. Native Oracle support is rarely used and ODBC is the recommended way to connect to databases.
   - `MY_LIBRARY_DIRS` is used to specify the external libraries and `libs` there has the form `/LIBPATH:"libdir1" /LIBPATH:"libdir2" ....`
   - `MY_INCLUDE_DIRS` is used to specify additional directories for included files. Here `opts` has the form `/I"incdir1" /I"incdir2" ....`

   Instead of specifying the options on command line, it might be more convenient (and more general) to create the file

   ```
   XSB\build\windows64\custom_settings.mak
   ```

   and put the required options there. For instance,

   ```
   XSB_INTERPROLOG=yes
   MY_INCLUDE_DIRS=/I"C:\Program Files\Java\jdk1.8.0_131\include" \
         /I"C:\Program Files\Java\jdk1.8.0_131\include\win32"
   MY_LIBRARY_DIRS=/LIBPATH:"C:\pthreads\pthreadVC1.lib" /libpath:"C:\oracle"
   ORACLE=yes
   ```

   Make sure you do not misspell the name of that file or else none of the specified options will take effect!

5. The above command will compile XSB as requested and will put the XSB executable and its DLL in:

```
$XSB_DIR\config\x64-pc-windows\bin\xsb.exe
$XSB_DIR\config\x64-pc-windows\bin\xsb.dll
```

6. To compile XSB as a 32 bit application (**not** recommended), use `makexsb` instead of `makexsb64`. The compiled code will be installed in

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

The `custom_settings.mak` file must then be in

```
XSB\build\windows\custom_settings.mak
```

Note: if you compiled XSB with one set of parameters and then want to recompile with a different set, it is recommended that you run

```
makexsb64  clean
```

in between the compilations (or `makexsb clean` in the 32-bit case). This also applies to recompilations for 64/32 bits.

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 installed 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\bin\xsb
```

This script tries to find the XSB executable and invoke it. If, for some reason, it fails to do so, the user should call the executable directly.

```
$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 call the above script from there.

ISO"standard" Prolog predicates are supported by XSB, in addition to many other predicates: 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

One way to compile a program from a file, such as `myfile.P` 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. Chapter 3 contains a full discussion of the compiling and consulting.

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

There are several sample XSB source programs in the directory: `$XSB_DIR/examples` illustrating a number of standard features, as well as a number of non-standardized

or XSB-specific features including plain tabling, incremental tabling, tabling with negation, attributed variables, annotated programs, constraint handling rules, XSB embedded in a C program, XSB calling C functions, sockets, and various semantic web appliation

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 3.1 (Incognito) of August 10, 2007
[i386-apple-darwin8.9.1; mode: optimal; engine: slg-wam; scheduling: local; word size: 32]
| ?- [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
  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 from the Command Line

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

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

If, after being built, XSB is later installed at a central location, `$SHARED_XSB`, the emulators executable code appears in

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

Either of these commands invokes XSB's top-level interpreter, which is the most common way of using XSB.

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, e.g. `:- go.`, and `foo.P` has been compiled to an object file (`foo.xwam`), then

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

will execute `go` (and any other directives), loading the appropriate files as needed [1]. In fact the command `$XSB_DIR/bin/xsb` is equivalent to the command:

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

There is one other way to execute XSB from a command line. Using the `-e` command-line option any goal can be can be executed, up to 1024 characters. For instance

```
$XSB_DIR/bin/xsb -e "writeln('hello world'),halt."
```

writes "hello world" and exits XSB. Within the 1024 character limit, any query or command can be executed, including consulting files, so this method is actually quite general footnoteVarious options can suppress XSB's startup and end messages, as discussed below..

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

When installed, the XSB system resides in a single directory that contains several subdirectories. For completeness, we review the information in all subdirectories. Normally, only the documentation and files in the Prolog subdirectories, particularly `examples`, `lib`, and `packages` will be of interest to users.

1. `bin` contains scripts that call XSB executables for various configurations.
2. `build` contains XSB configuration scripts. You may already be familiar with the `build` directory, which is used to build XSB.
3. `config` contains executables and other files specific to particular configurations.
4. `docs` contains the user manuals and other documentation, including the technical documentation manual for developers.

[1] In XSB, all extensions except '.pl' and '.prolog' — (default '.P', '.H', '.xwam', '.D' (output by mode inferencing), and '.A' (assembly dump) — are defined in C and Prolog code using macros in `$XSB_DIR/emu/extensions_xsb.h` and can be changed by a user if desired. Of course, such a step should not be taken lightly, as it can cause severe compatibility problems.

5. `emu` contains the C source code for the XSB emulator, for I/O and for various interfaces.
6. `etc` contains miscellaneous files used by XSB.
7. `examples` contains some examples for Prolog, tabling, HiLog and various interfaces.
8. `cmplib` contains Prolog source and object code for the compiler.
9. `gpp` contains a copy of the Gnu pre-processor used to preprocess Prolog files.
10. `lib` contains Prolog source and object code for extended libraries.
11. `packages` The directory `packages` contains the various applications, such as FLORA, the XMC model checker and many others. These applications are written in XSB and can be quite useful, but are not part of the XSB system per se.
12. `Prolog_includes` contains include files for the Prolog libraries, which are preprocessed using GPP.
13. `syslib` contains Prolog source and object code for core XSB libraries.

All Prolog 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.

If you are distributing an application based on XSB and need to cut down space, the `packages`, `examples` and `docs` directories are not usually needed (unless of course you are using one of the packages in your application). `lib` may not be needed, (most core system files are in syslib) nor are Prolog source files necessary. Unless your application needs to rebuild XSB, the `emu` and `build` directories do not need to be distributed.

3.3 How XSB Finds Files: Source File Designators

Three files are associated with Prolog source code in XSB [2].

[2]Other types of files may be associated with foreign code — see Volume 2.

- A single *source* file, whose name is the *base file name* plus an optional extension suffix `.P`, `.pl`, or `.prolog`.
- An *object (byte-code)* file, whose name consists of the base file name plus the suffix `.xwam`.
- An optional *header* file, whose name is the base file name plus the suffix "`.H`". When used, the header file normally contains file-level declarations and directives while the source file usually contains the actual definitions of the predicates defined in that module. However, such information can be equivalently put into the `.P` (`.pl`, or `.prolog`) file.

Most of the XSB system predicates for compiling, consulting, and loading code, such as `consult/[1,2]`, `compile/[1,2]`, `load_dyn/1` and others are somewhat flexible in how they designate the file of interest. Each of these predicates take as input a *source file designator* which can be a base file name, a source file name; or the relative or absolute paths to a base or source file name. Unfortunately, the exact semantics of a file designator differs among system predicates in Version 3.8, as well as among platforms.

In general, however, when given a source file designator, system predicates perform *name resolution*. There are two steps to name resolution: determining the proper directory prefix and determining the proper file extension. When `FileName` is absolute (i.e. it contains a path from the file to the root of the file system) determining the proper directory prefix is straightforward. If `FileName` is relative, i.e. it contains a `'/'` in Unix or `'/'` in Windows, `FileName` is expanded to a directory prefix in an OS-dependent way, resolving symbols like `'.'`, `'..'` and `'~'` when applicable. However, the user may also enter a name without any directory prefix. In this case, XSB tries to determine the directory prefix using a set of directories it knows about: those directories in the dynamic loader path (see Section 3.6). As it searches through directory prefixes, different forms of the file name may be checked. If the source file designator has 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, and finally for a file with a `.pl` or `.prolog` extension. Note that since directories in the dynamic loader path are searched in a predetermined order (see Section 3.6), if the same file name appears in more than one of these directories, the first one encountered will be used.

3.4 The Module System of XSB

XSB has been designed as a module-oriented Prolog system. Modules provide a step towards *logic programming "in the large"* that facilitates the construction of large programs or projects from components that are developed, compiled and tested separately. Also, module systems support the principle of information hiding and can provide a basis for data abstraction. And modules form the basis for XSB's implementation of its standard predicates.

The module system of XSB is *file based* – one module per file – and *flat* – modules cannot be nested. In addition, XSB's module system is essentially *atom-based* (or *structure-symbol-based*), where any symbol in a module can be imported, exported or be a local symbol, as opposed to a *predicate-based* one where this can be done only for predicate symbols [3]. Every structure symbol (and thus structured term) is associated with a module, and structure symbols with the same name but in different modules are different symbols and thus do not unify. As we will discuss, this leads to certain differences of XSB's module system from those of some other Prologs, and to certain incompatibilities with the (proposed) ISO standard for modules (which is not supported by most Prologs). At the same time, XSB's module system has enough commonalities with those of other Prologs to be able to support the Prolog Commons libraries.

In XSB (as in all Prolog systems) predicate definitions (aka Clauses) are associated with structure symbols. A predicate is a structure symbol with an associated definition. Predicates are either static or dynamic. Static predicates get their definitions from source code files that are compiled and loaded into memory. Dynamic predicates get their definitions from the execution of the builtin meta-predicate `assert/1` (and friends).

In XSB every structure symbol is associated with a module. A term is said to be in the module of its main structure symbol. Terms in different modules are different terms and do **not** unify. So two terms whose main structure symbols (or any structure symbols) have the same name but different modules, are different terms and do not unify. So, for example, terms printed as `p(a,b)` and `p(a,b)` would not unify if the first structure symbol named `p/2` is in a different module from the second structure symbol named `p/2`.

The "default" module is named `usermod`. Whenever a term is constructed, and a module is not explicitly provided, `usermod` is the module used. For example, when `functor/3` (or `univ/2`) is used to construct a term, that term is put in `usermod`.

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

Any term that is read from a file (or at the top-level prompt) is put in `usermod`. All (usual) XSB source files, when compiled and loaded, define predicates in `usermod`.

So how are terms that are not in usermod constructed? The most important use of modules by far is to organize predicates (and thus their definitions.) So a module is associated with a set of predicate definitions, which in XSB is a Prolog source file, a file with `.P` extension. In XSB, a source file is compiled (to a `.xwam` file) and then loaded into memory to provide definitions for the predicates with clauses in that file. For "usual" XSB source files, all the defined predicates are in `usermod`. However when a source file includes an export directive, such as:

`:- export` *Pred/Arity*`.`

the definitions in that source file will be interpreted as defining predicates in a module. The name of the module is the name of the XSB source file (without the extension `.P`). Predicates that have definitions in such a file will all be put in the module of that name. A predicate that is exported must be defined in the source file and will be made available for use in other source files, when imported. An import directive, such as:

`:- import` *Pred/Arity* `from` *Module*`.`

in another source file allows its definitions to use that exported predicate. For example, the file:

```
%% file: mod1.P
:- export p/2.

p(a,b).
p(X,Y) :- q(X,Y).

q(b,c).
```

when compiled and loaded, defines a predicate `p/2` in `mod1` (i.e., `p/2` terms that define the facts have their main structure symbols put in module `mod1`, and the code implementing those clauses are associated with that structure symbol in that module.) It also defines a predicate `q/2` in the same module. (And, of course, the call to `q(X,Y)` in the body of the rule for `p/2` is also put in that same module.) The predicate `p/2` is exported and is thus available for use by other code.

For example, we can create another file (not a module in this case), which uses the definition of `p/2` above:

```
%% file: my_code.P
```

```
:- import p/2 from mod1.

q(X,Y) :- p(X,Y).
```

Here there is no `export` directive, so all definitions in this file will go into module `usermod`. The clause here defines `q/2` in `usermod`, which is a different predicate from the `q/2` defined above in the module `mod1`. The `import` of `p/2` in this file causes the `p(X,Y)` term in the body of the rule for `q/2` to be interpreted as being in module `mod1`. Thus, when this file is compiled and loaded, `q/2` is defined in `usermod` and its code calls `p/2` in module `mod1`.

A module source file may want to access a predicate defined in `usermod`, which can be done by explicitly importing the predicate from `usermod`.

There are situations in which a programmer wants to explicitly provide a module name to "override" the module associated with a term. For example, one might want to call the goal `p(X,Y)` to invoke the code associated with `p/2` in module `mod1` at the top level, regardless of what module the `p/2` structure symol is associated with. In this case, one can write:

```
| ?- call(mod1:p(X,Y)).
```

Here `call` will construct the term `p(X,Y)` with structure `p/2` in module `mod1` (ignoring the module associated with the `p/2` structure symbol) and then call that term, which accesses the code of `p/2` in module `mod1`. In this particular case the original term `mod1:p(X,Y)` had the `p/2` structure in `usermod`, since that's where the top-level read puts it. But `call/1` interprets this term (with main structure symbol `:/2`) as a coercion of the term `p(X,Y)` into the module `mod1`. In XSB, in most contexts in which a term is interpreted as a goal, the syntax of `Mod:Goal` is interpreted as a coercion of term `Goal` into the module `Mod`. And in fact, the top-level goal:

```
| ?- mod1:p(X,Y).
```

is equivalent to the goal above.

And instead of:

```
:- import pr/2 from mod3.
...
q(X,Y) :- ... pr(X,Z), ....
```

one can directly write:

```
q(X,Y) :- ... mod3:pr(X,Z), ....
```

In general, the use of `import` is recommended, even though it may sometimes be more verbose. The use of `imort` allows for better visibility and easier analysis of module dependencies.

In XSB, the declaration:

`:- module(`$filename$`,[`sym_1`, ..., `sym_l`.]).`

is syntactic sugar for:

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

as long as the $filename$ is the same as the name of the file in which it was contained. Similarly,

`:- use_module(`$module$`,[`sym_1`, ..., `sym_l`.]).`

is treated as semantically equivalent to

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

Accordingly, `use_module/2` and `module/1` can be used interchangibly with `import/2` and `export/1`. However the declaration

`:- use_module(`$module$`).`

which is often used in other Prolog systems, is *not* equivalent to an XSB import statement, as each XSB import statement must explicitly declare a list of predicates that are used from each module. Such a declaration will raise a compilation error.

The declaration

`:- import `sym` from `$module$` as `sym'`.`

allows a predicate to be imported from a module, but renamed as sym' within the importing module. In this case the structure symbol sym' is placed in the current module and its code pointer is identified with that of the structure symbol sym in module $module$. Such a feature is useful when porting a library written for another Prolog (e.g. a constraint library) to XSB. It is also useful when one wants to import two predicates with the same name from different modules. In that case (at least) one of the names needs to be changed on import.

For modules, the base file name is stored in its byte code (`.swam` file, so that renaming a byte-code file for a mule may cause problems, as the renaming will not affect the information within the byte-code file. However, byte code files generated for non-modules can be safely renamed.

3.4.1 How the Compiler Determines the Module of a Term

When XSB compiles a source code file, it must determine the module for every term it encounters. For non-module source files (i.e., those with an `export` directive), all terms are associated with `usermod` except for those whose structure symbols are imported. Any occurrence of an imported structure symbol is associated with the module from which it is imported.

For module source code files, i.e., those containing at least one `export` directive, the process of determining the module of a structure symbol is more complicated. The idea is that all terms in the source file that refer to predicates are placed in the module of the file, and all terms that are record structures are by default placed in `usermod`. All occurrences of the same structure symbol in a file are normally associated with the same module[4] So if a structure symbol appears both as a predicate symbol (e.g., as a subgoal in the body of a rule) and as a record structure (perhaps to be passed to some other predicate to later be called), both occurrences will be associated with the current module. Of course, imported structure symbols are associated with the module from which they are imported.

The compiler recognizes as predicate symbols any symbol that:

1. appears as the main structure symbol in the head of a rule,
2. appears as a subgoal in the body of a rule,
3. appears as the main structure symbol of terms passed to known meta-predicates, such as `assert/1` and `retract`,
4. is declared as `dynamic`.

Otherwise a structure symbol is associated with `usermod`.

Note that these rules imply that all structure symbols used just for record structures are placed by default in `usermod`. This is usually what a user wants. But there are times a user might want a record structure to be associated with the current module. This can be used to provide a measure of information hiding: Since no other module (or usermod) will construct a term associated with this module, another module can't use unification to look at the subfields inside such a term. So in this way, a module can return to a caller a complex term, and the caller can pass it around and back to the module in a later call, and only the module code can manipulate that

[4] but see `import .. as ..` for an exception.

data structure.[5] The programmer can tell the compiler to place a particular structure symbol in the current module by using the `local` directive:

`:- local` *Sym/Arity*`.`

which will force all uses of the indicated structure symbol to be associated with the current module.

An XSB programmer can also export a structure symbol (that is not used as a predicate), and others can import and use it as a structure symbol.

Standard predicates (those defined in the XSB environment) are actually defined in system modules and the compiler implicitly provides the necessary imports to allow the programmer to use them. Standard predicates are described in Section 3.5.

For clarity, we state a few consequences of these rules.

- The 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.
- In Version 3.8, a module *cannot* export a predicate symbol that is directly imported from another module, since this would require that symbol to be in two modules. But one can import $symbol_1$ from a module `as` $symbol_2$ and then export $symbol_2$. (And $symbol_1$ and $symbol_2$ are allowed to be the same symbol.)
- If a module `m1` imports a predicate `p/n` from a module `m2`, but `m2` does not export `p/n`, nothing is detected at the time of compilation. If `p/n` is defined in `m2`, a runtime warning about an environment conflict will be issued. However, if `p/n` is not defined in `m2`, a runtime existence error will be thrown [6].

3.4.2 Atoms and 0-Ary Structure Symbols

XSB uses different internal representations for **atoms** and for **0-ary structure symbols**. Atoms cannot have definitions associated with them (i.e., cannot be predicates)

[5] The hiding is only partial, since other code can use `functor/3` or `univ/2` to look inside such terms. Also the very low-level builtin `term_new_mod/3` can be used to explicitly coerce a term into an arbitrary module.

[6] This behavior can be altered through the Prolog flag `unknown`.

and are not associated with modules. But 0-ary predicates can and are. The system automatically coerces atoms to 0-ary structure symbols and vice versa as necessary. But when coercing an atom to a 0-ary structure symbol, it **always** associates the generated structure symbol with `usermod`. This can sometimes lead to unexpected results. As long as the programmer explicitly exports and imports atoms (or 0-ary predicate symbols), all works as expected. But passing an atom as an argument, and then calling it will always call it in `usermod`.

3.4.3 Dynamic Loading and How XSB Finds Code Files

When `export` and `import` directives are used, XSB dynamically (compiles if necessary and) loads the code on demand. When an imported predicate is called, if the code of the module has not been loaded into memory, the system finds the code file, compiles it if necessary, and loads the `.xwam` file into memory. Then it invokes the imported predicate. See Section 3.6 for the details of how the system finds and processes the appropriate XSB source files.

3.4.4 Consulting a Module

Normally all access to predicates defined in a module is by means of import declarations. However, to debug a module it is often convenient just to consult it at the top-level and then call the exported predicates with test parameters, which is how non-modules are handled. However, note that the predicate to be called after a module is loaded is in that loaded module, and **not** in `usermod`. To allow the programmer to call a predicate exported from the consulted module without having to explicitly provide the module name, when a module is consulted, all exported predicates are also defined in `usermod` with their same definitions. (In effect, for exported `p/2`, XSB implements `:- import p/2 from` *module* `as p/2.` in `usermod` to provide direct access to `p/2`'s code in *module* from the `p/2` predicate in `usermod`.)

It is bad form to use this property and consult a module in an executing program to get access to its exported predicates through `usermod`. One should always explicitly import the predicates one wants to use and let the dynamic loader automatically load the module code on demand.

3.4.5 Usage Inference and the Module System

The import and export statements of a module M are used by the compiler for inferring usage of predicates. At compilation time, if a predicate P/N occurs as callable in the body of a clause defined in M, but P is neither defined in M nor imported into M from some other module, a warning is issued that P/N is undefined. Here "occurs as callable" means that P/N is found as a literal in the body of a clause, or within a system meta-predicate, such as `assert/1`, `findall/3`, etc. Currently, occurrences of a term inside user-defined meta-predicates are not considered as callable by XSB's usage inference algorithm. Alternatively, if P/N is defined in M, it is *used* if P/N is exported by M, or if P/N occurs as callable in a clause for a predicate that is used in M. The compiler issues warnings about all unused predicates in a module. On the other hand, since all modules are compiled separately, the usage inference algorithm has no way of checking whether a predicate imported from a given module is actually exported by that module.

Usage inference can be highly useful during code development for ensuring that all predicates are defined within a set of files, for eliminating dead code, etc. In addition, import and export declarations are used by the `xsbdoc` documentation system to generate manuals for code.[7] For these reasons, it is sometimes the case that usage inference is desired even in situations where a given file is not ready to be made into a module, or it is not appropriate for the file to be a module for some other reason. In such a case the directives `document_export/1` and `document_import/1` can be used, and have the same syntax as `export/1` and `import/1`, respectively. These directives affect only usage inference and `xsbdoc`. A file is treated as a module if and only if it includes an `export/1` statement, and only `import/1` statements affect dynamic loading and name resolution for predicates.

3.4.6 Using Import to Load Usermod Source Files

When the module system is used to import predicates, code files for modules are automatically found and dynamically (compiled and) loaded on first access. But normally non-module source files must be explicitly consulted or ensure_load-ed by some executing program. To provide the convenience (and declarativity) of dynamic loading to usermod source files, XSB supports a directive of the form:

`:- import` *Pred*`/`*Arity* `from usermod(`*File*`).`

Here *File* must be the name of a file that contains XSB source code and is **not**

[7] Further information on `xsbdoc` can be found in `$XSB_DIR/packages/xsbdoc`.

a module, i.e., it defines its predicates in `usermod`. It must define the predicate *Pred/Arity*. In this case, when a goal to *Pred/Arity* is called and does not yet have a definition, the file *File* is (compiled and) loaded, and the goal is called. If *File* is a base filename (without a slash), then the `library_directory/1` paths are used to find the correct file (as for normal modules.) If the predicate already has a definition, that one is used.[8]

So this facility allows code in non-module files to be treated somewhat like module files. But, as usual, it is the user's responsibility to ensure that different imported non-module files do not define the same predicate. This facility, when carefully used, can eliminate the need for runtime `consult/1` and `ensure_loaded/1` commands. The form `usermod(`*File*`)` is called a pseudo-module reference, and can be used in place of module references in import statements.

Note that:

`:- document_import` *Pred/Arity* `from` *File*`.`

can be replaced with

`:- import` *Pred/Arity* `from usermod(`*File*`).`

XSB does not automatically treat the former as the latter, for backwards compatibility. They can have differing effects if the given file does not define the given predicate.

XSB also supports a similar import directive form, exemplified by the following:

`:- import` *Pred/Arity* `from usermod(load_dyn(`*File*`)).`

This will cause the file *File* to be loaded dynamically on first use. It must, of course, define *Pred/Arity*. The `load_dyn` in this example may be replaced by any file-loading predicate whose first argument is the name of the file to load. For example, one might also use:

`:- import` *Pred/Arity* `from usermod(consult:load_dync(`*File*`,0)).`

to dynamically load a file whose contents are canonical terms to be asserted in reverse order. In fact, one may use:

`:- import` *Pred/Arity* `from usermod(proc_files:load_dsv(`*File*`,`*Pred/Arity*`,[])).`

to load a comma-separated file with each line containing two fields to define the predicate *Pred/Arity*. (See 15.12 for details.)

[8] If the existing definition can be determined to have come from a different file, a warning is generated.

3.4.7 Parameterized Modules in XSB

The XSB module system now supports parameterized modules: A module can be parameterized by other modules. A parameterized module is declared by including a directive of the form:

`:- module_parameters(`$atom_1$`, ..., `$atom_n$`).`

in the module code file. The atoms, $atom_1, \ldots, atom_n$, are formal module parameters; when the module is loaded, those module names will be replaced by actual module names passed to the load operation. Therefore, module names are now specified by ground terms: the main structure symbol specifies the base name of the file containing the module code (as before); the (optional) arguments of the module term indicate the names of (the other modules that are) the actual parameters to the (parameterized) module defined in this file. Note that the parameters to modules *must* be other modules, and cannot be constants or any XSB term. Parameterized modules are a conservative extension of the former unparameterized module system.

Parameterized modules support a form of higher-order programming which makes it possible to program some tasks more declaratively. As a simple example, consider a module that takes a graph, an initial node in the graph, and a set of final nodes in the graph, and returns all final nodes reachable through the graph from the initial node. A parameterized module for this task, named `search`, is:

```
%% file: search.P

:- module_parameters(example_mod).
:- export reachable_final/1.
:- import initial/1, move/2, final/1 from example_mod.

reachable_final(F) :- reachable(F), final(F).

:- table reachable/1.
reachable(N) :- initial(N).
reachable(N) :- reachable(P), move(P,N).
```

This module, `search`, is parameterized by another module that defines and exports (at least) 3 predicates: `initial/1`, `move/2`, and `final/1`. When this module is loaded, a particular such module, exporting those predicates, must be provided to the loader, and the formal parameter `example_mod` will be replaced by that module and the predicates imported from that module will be used here in the definitions of `reachable_final/1` and `reachable/1`. So assuming a (non-parameterized) module

named `simple_ex` exports those 3 predicates, both:

```
| ?- import reachable_final/1 from search(simple_ex).
| ?- reachable_final(ReachableFinalState).
```

and:

```
| ?- search(simple_ex):reachable_final(ReachableFinalState).
```

will return the reachable final states for the problem defined by `simple_ex`.

This is second-order in the sense that a module parameter represents a set of predicates. Note that this example is (in some sense) fully declarative, in that there is no explicit procedural code necessary to load the code for a particular example. All loading is handled by XSB's existing dynamic loader. And this same search module can be run with many different examples.

Parameterized modules are implemented in XSB as follows. When a parameterized module is to be loaded into memory, the formal parameters are replaced by the actual parameters and that code is loaded. (This is actually done by renaming symbols as they are loaded, so there is minimal effect on loading time.) This implementation has two consequences: 1) the performance of code in parameterized modules is *exactly* the same as if the user had explicitly written the module with the actual parameter modules, and 2) every instance of a parameterized module has its own version of the module code. So loading a thousand different instances of a parameterized module will take a thousand times the space of a single instance. In most uses this is not a significant problem, but it should be kept in mind.

One could load another instance of the above module to test the search algorithm with a different example by:

```
| ?- search(hard_ex):reachable_final(ReachableFinalState).
```

This would load another, different, instance of the search module, named `search(hard_ex)`. Both would be in memory and usable by the user and by other programs and modules.

So modules in XSB's runtime system can now identified by module names that are terms, not simply constants. Accordingly, anywhere a module name is required, a parameterized module name, i.e., a module term, can be used. The module name must be ground at the time it is required for use in order to load specific code; and all structure symbols and atoms in the structured module identifier must identify actual files that contain the appropriate module's code; and finally those files must be able to be found by the XSB loader.

To write well-structured and maintainable code, it is strongly recommended that all uses of parameterized modules be done through `use_module/2` or `import` directives explicitly appearing in XSB code. The explicit form of using the ':' operator to give a module name at runtime should be avoided. (The sole exception is when the user types in such a goal on the top-level command line.) Using only explicit import directives allows compile-time analysis of module dependencies which can be critical in maintaining large XSB code bases. This also requires that the extension of the `library_directory/1` predicate can be known at compile time, which implies programmer discipline in changing that predicate as well.[9].

While parameterized modules can be used in many ways, one of the most important is in the construction of so-called "view systems." A view (in the traditional relational database sense) is a relational operator that takes a set of input relations and views, and produces an output relation. By composing views one can build large and complex systems of data transformations in a completely declarative way. With such systems, one often receives base (i.e., input) data from a source, and then wants to apply a view system to that data, generating the derived views for use in other applications. One can do this declaratively by using parameterized modules. Each module is a view definition, exporting the view it defines, and importing the base and view relations it depends on. These input relations can be defined in base modules, and a view module is parameterized by the base modules it depends on. Then the same view module can be applied to the particular input tables obtained from a particular source.

3.5 Standard Predicates in XSB

Whenever XSB is invoked, a large set of *standard* predicates are defined and can be called from the interpreter or other interface [10]. These predicates include the various ISO predicates [37], along with predicates for tabling, I/O, for interaction with the operating system, for HiLog, and for much other functionality. Standard predicates are listed in this manual under the index heading *Standard predicates* and at an implementation level are declared in the file `$XSB_DIR/syslib/std_xsb.P`. If a user wishes to redefine a standard predicate, she has several choices. First, the appropriate fact in `$XSB_DIR/syslib/std_xsb.P` should be commented out. Once this is done, a user may define the predicate as any other user predicate. Alternately, the compiler option `allow_redefinition` can be used to allow the compiler to redefine a standard predicate (Section 3.10.2). If a user wants to make a new definition or new predicate

[9] As may be obvious, this has been learned through much painful experience. -dsw

[10] Such predicates are sometimes called "built-ins" in other Prologs.

standard, the safest course is to put the predicate into a module in the `lib` directory, and add or modify an associated fact in `$XSB_DIR/syslib/std_xsb.P`.

3.6 The Dynamic Loader and its Search Path

XSB differs from some other Prolog systems in its ability to *dynamically* load modules. In XSB, the loading of user modules and Prolog libraries (such as the XSB compiler) is delayed until predicates in them are actually needed, saving program space for large Prolog applications. Dynamic loading is done by default, unlike other systems where it is not the default for non-system libraries.

When a predicate imported from another module (see Section 3.4.7) 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 error stream reporting that the module was not found. Because system modules are dynamically loaded, the time it takes to compile a file is slightly longer the first time the compiler is invoked in a session than for subsequent compilations.

3.6.1 Changing the Default Search Path and the Packaging System

The default search path of the dynamic loader is based on the dynamic predicate `library_directory/1` so it can easily be changed. For instance, to make sure a user's home directory is loaded, the goal `add_lib_dir(('~/'))` needs to be executed from the command line or from within a program (assuming this is not the current working directory). If you always want XSB to search particular directories, the easiest way is to have a file named `.xsb/xsbrc.P` in your home directory. User-supplied library directories are searched by the dynamic loader *before* searching the default library directories. The `.xsb/xsbrc.P` file, which is automatically consulted by the XSB interpreter, might look like the following:

```
:- add_lib_dir(('~/')).
:- add_lib_dir(('/usr/lib/xsbprolog')).
```

`add_lib_dir(+Directories)`

`add_lib_dir(+Root,+Directories)`

The standard predicate `add_lib_dir(Directories)` adds the directories of `Directories` to the system predicate `library_directory/1`. `Directories` is either a single directory name or a comma-list of directory names. A directory name may be an atom or a simple structure of the form `a(DirName)` which indicates that the directory `DirName` should be added as the first directory in the `library_directory/1` facts; otherwise it will be added as the last directory.

In the example above `add_lib_dir(('~/'))`, note that the "extra parentheses" are needed since `add_lib_dir/1` takes a single argument, here a comma-pair. Also the trailing slash in a directory name is optional.

The standard predicate `add_lib_dir(+Root,+RelativeDirectories)` concatenates the directory indicated by `Root` to each of the relative directory names in (the comma-list) `RelativeDirectories` and adds them all to `library_directory/1`.

For example, to add two XSB library directories from a set of libraries stored under a particular directory containing all XSB libraries, one might do:

```
:- add_lib_dir('/usr/lib/xsb_libs', (string_lib,table_lib)).
```

(Note that the necessary slash-separators are automatically added if necessary.)

If `Root` is a term of the form `ancestordir(DirFileName)` where `DirFileName` is an atom, the system will search up from the current directory to find a containing directory named `DirFileName`, and the full pathname of that directory will be considered as the `Root` directory. This can be used to help in making XSB code less dependent at compile-time on the exact full filename of XSB code files, and allowing directories of libraries to be moved.

A user's configuration file: `xsbrc.P`

Returning to the previous example, executing the two directives causes the user's home directory to be searched first, then `"/usr/lib/xsbprolog/"`, and finally XSB's system library directories (`lib, syslib`, `cmplib`), and finally the current working directory. The directives themselves can be executed by expicitly loading an XSB file, by executing the directives at the command line, or automatically using an `xsbrc.P` file.

This file works as follows. Before the user's `.xsb/xsbrc.P` is consulted, XSB puts both 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). If a user wished, say, to search the current working directory *before* her home directory, she could simply add

```
:- asserta(library_directory('./')).
```

or better

```
:- add_lib_dir(a('./')).
```

to her `.xsb/xsbrc.P` file (or anywhere else). The file `.xsb/xsbrc.P` is not limited to setting the library search path. In fact, arbitrary Prolog code can go there so that XSB can be initialized in any manner desired.

We emphasize that in the presence 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 unexpected behavior as system code may try to load a user's predicates. The list of module names in XSB's system library directories can be found by looking through the directories `$XSB_DIR/{syslib,cmplib,lib}`.

Packages 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 a number examples of such packages, many of which are documented in Volume 2 of this manual, or contain their own manuals. 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 a 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`.

Finally, if you have developed and tested a package that you think is generally useful and you would like to distribute it with XSB, please contact `xsb-development@sourceforge.net`.

3.6.2 Dynamically loading predicates in the interpreter

Modules are usually loaded into an environment when they are consulted (see Section 3.9). 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 3.4.7).

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.P` 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.

3.7 Command Line Arguments

There are several command line options for the emulator. The general synopsis obtained by the command `$XSB_DIR/bin/xsb -help` is:

```
xsb [flags] [-l]
xsb [flags] module
xsb [flags] -B boot_module [-D cmd_loop_driver] [-t]
xsb [flags] -B module.suffix -d
xsb [-h | -v | --help | --version]

module:
    Module to execute after XSB starts up.
    Module should have no suffixes, and either be an absolute pathname
    the file module.xwam must be on the library search path.
boot_module:
    This is a developer's option.
```

```
    The -B flags tells XSB which bootstrapping module to use instead
    of the standard loader.  The loader must be specified using its
    full pathname, and boot_module.xwam 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.

                -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
      -v, --version : print the version and configuration information about XSB
         -h, --help : print this help message

Flags:
              -e goal : evaluate goal when XSB starts up
                   -p : enable Prolog profiling through use of profile_call/1
                   -l : the interpreter prints unbound variables using letters
           --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)
                   -S : set default tabling method to call-subsumption
 --max_subgoal_size  N : set maximum tabled subgoal size to N (default is maximum integer)
--max_subgoal_action A : set action on maximum subgoal depth: e(rror)/a(bstract)/w(arn)
        --max_tries N : allow up to N tries for interning terms
      --max_threads N : maintain information for up to N threads (MT engine only)
      --max_mqueues N : allow up to N message queues (MT engine only)
   --shared_predicates : make predicates thread-shared by default
            -g gc_type : choose heap garbage collection ("indirection","none" or "copying")
          -c N [unit] : initially allocate N units (default KB) for the trail/choice-point sta
          -m N [unit] : initially allocate N units (default KB) for the local/global stack
          -o N [unit] : initially allocate N units (default KB) for the SLG completion stack
                   -r : turn off automatic stack expansion
                   -T : print a trace of each called predicate

         unit: k/K memory in kilobytes; m/M in megabytes; g/G in gigabytes
```

3.7.1 Command-line Options

These options tend to be most useful for developers.

`-t` Traces through code at SLG-WAM instruction level. This option is intended for developers and is not fully supported. It is also not available when the system is being used at the non-debug mode (see Section 10).

`-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.

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

3.7.2 General Flags

The order in which flags appear makes no difference.

`-e goal` Pass `goal` to XSB at startup. 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.

`-p` Enables the engine to collect information that can be used for profiling. See Volume 2 of this manual for details.

`-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 = _h118
X = _h118
Z = 3
W = foo(_h118,3);
```

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

```
 xsb -e "writeln('hello world'),halt."
[xsb_configuration loaded]
[sysinitrc loaded]

XSB Version 3.1 (Incognito) of August 10, 2007
[i386-apple-darwin8.9.1; mode: optimal; engine: slg-wam; scheduling: local; word si


Evaluating command line goal:
| ?-  writeln('hello world'),halt.

| ?- hello world

End XSB (cputime 0.02 secs, elapsetime 0.02 secs)
```

Prints out quite a bit of verbiage. Using the `-nobanner` option reduces this verbiage somewhat.

```
xsb --nobanner -e "writeln('hello world'),halt."
[xsb_configuration loaded]
[sysinitrc loaded]

Evaluating command line goal:
| ?-  writeln('hello world'),halt.

| ?- hello world
```

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

```
xsb --quietload --nobanner -e "writeln('hello world'),halt."
| ?-
| ?- hello world
```

`-noprompt` Do not show the XSB prompt.

`-nofeedback` Do not print the feedback messages such as "yes" and "no" after queries. This and the `-noprompt` options are useful only in batch mode and in interprocess communication when you do not want the prompt to clutter the picture. Putting all this together, we finally get:

```
xsb --noprompt --quietload --nobanner --nofeedback -e "writeln(hello),halt."

hello world
```

So that XSB can be used to write reasonable scripts.

`-max_threads N` Allows XSB to maintain information for up to `N` threads. This means that XSB can currently run `N` threads that are active, or that are inactive, non-detached, and not yet joined. Has no effect if the engine has been configured without multi-threading.

`-S` Indicates that tabled predicates are to be evaluated using subsumption-based tabling as a default for tabled predicates whose tabling method is not specified by using `table Predspec as subsumptive` or `table Predspec as variant`(see Section 6.15.1). If this option is not specified, variant-based tabling will be used as the default tabling method by XSB.

`-shared_predicates` In the multi-threaded engine, makes all predicates thread-shared by default; has no effect in the single-threaded engine.

`-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 standard predicates `trace/0` or `debug/0` should be used (see Chapter 10). Note: This option is not available when the system is being used at the non-tracing mode (see Section 10).

`-max_subgoal_size` ***N*** : set maximum tabled subgoal size to N (default is maximum integer). This flag sets the size of a tabled subgoal upon which an action may be taken (such as throwing an error, abstracting, or issuing a warning.

`-max_subgoal_action` ***A*** : set action on maximum subgoal depth: e(rror)/a(bstract)/w(arn)

3.7.3 Memory Management Flags

`-g gc_type` Chooses the heap garbage collection strategy that is employed; choice of the strategy is between the default `indirection` or `none`. See [11] for a description of the indirection garbage collector.

`-c` ***size [units]*** Allocates *initial size* units 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. If units is not provided or is `k` or `K`, the size is allocated in kilobytes; if `m` or `M` in megabytes; and if `g` or `G` in gigabytes. Because this region is expanded automatically, this option is rarely needed. If this option is not specified a default initial size is used; this size may differ for the single-threaded and multi-threaded engine.

`-m` ***size [units]*** Allocates *initial size* units 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. If units is not provided or is `k` or `K`, the size is allocated in kilobytes; if `m` or `M` in megabytes; and if `g` or `G` in gigabytes. Because this region is expanded automatically, this option is rarely needed. If this option is not specified a default initial size is used; this size may differ for the single-threaded and multi-threaded engine.

`-o` ***size [units]*** Allocates *initial size* units of space to the completion stack area. If units is not provided or is `k` or `K`, the size is allocated in kilobytes; if `m` or `M` in megabytes; and if `g` or `G` in gigabytes. Because this region is expanded automatically, this option is rarely needed. If this option is not specified a default initial size is used; this size may differ for the single-threaded and multi-threaded engine.

`-r` Turns off automatic stack expansion. This can occasionally be useful for isolating memory management problems. (Usually when working with XSB developers.)

3.8 Memory Management

All execution stacks are automatically expanded in Version 3.8, including the local stack/heap region, the trail/choice point region, and the completion stack region. Execution stacks increase their size (usually by doubling) 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. For the main thread, each of these regions begin with an initial value set by the user at the command-line or with

a default value (see Section 3.7). When a thread is created within an XSB process, the size of the thread's execution stacks may be set by `thread_create/3`, otherwise the default values indicated in Section 3.7 are used. Once XSB is running, these default values may be modified using the appropriate Prolog flags (see Section 6.12). In addition, whenever a thread exits, memory specific to that thread is reclaimed.

Heap garbage collection is automatically included in XSB [11, 25]. (To change the algorithm used for heap garbage collection or to turn it off altogether, see the predicate `garbage_collection/1` or Section 3.7 for command-line options). In Version 3.8 the default behavior is indirect garbage collection. Starting with Version 3.0, heap garbage collection may automatically invokes garbage collection of XSB's "string" table, which stores Prolog's atomic constants. Expansion and garbage collection of execution stacks can occur when multiple threads are active; however atom garbage collection will not be invoked if there is more than one active XSB thread.

The program area (the area into which XSB byte-code is loaded) is also dynamically expanded as needed. For dynamic code (created using `assert/1`, or standard predicates such as `load_dyn/1` and `load_dync/1`) index size is also automatically reconfigured. Space reclaimed for dynamic code depends on several factors. If there is only one active thread, space is reclaimed for retracted clauses and abolished predicates as long as (1) there are no choice points that may backtrack into the retracted or abolished code, and (2) if the dynamic predicate is tabled, all of its tables are completed. Otherwise, the code is marked for later garbage collection. If more than one thread is active, private predicates behave as just described, however space reclamation for shared predicates will be delayed until there is a single active thread. See Section 6.14 for details.

Space for tables is dynamically allocated as needed and reclaimed through use of `abolish_all_tables/0`, `abolish_table_pred/1`, `abolish_table_call/1` and other predicates. As with dynamic code, space for tables may be reclaimed immediately or marked for later garbage collection depending on whether choice points may backtrack into the abolished tables, on the number of active threads, etc. Tabling also includes various stacks used to copy information into or out of tables, most of which are dynamically allocated and expanded. These stacks may be thread-private or shared among threads: space for thread-private stacks is reclaimed when a thread exits. See Section 6.15.4 for details.

Perhaps more than a standard Prolog system, XSB is used to evaluate queries in knowledge representation languages that have a higher level of declarativity than Prolog and as a result may consume a great deal of space. If XSB needs memory that is unobtainable from the operating sytsem, it will usually abort with a resource error, and become ready for a new query from its command line or API. In such

a case, a user or program can use `statistics/[0,1,2]` to investigate whether and how XSB is consuming memory. Other options to bounding memory include the use of `bounded_call/4` or the use of the `max_memory` flag. Use of the `max_memory` flag is recommended in cases where XSB is embedded in a C program through the C/XSB interface, or is embedded in or communicating with a java program through InterProlog. In such a case, XSB will abort with a resource error whenever a memory allocation would exceed the user-defined threshold [11].

3.9 Compiling, Consulting, and Loading

Like other Prologs, XSB provides for both statically compiled code and dynamically asserted code. Static compiled code may be more optimized than asserted code, particularly for clauses that have large bodies, but certain types of indexing, such as trie and star indexing are (currently) available only for dynamically asserted predicates (see `index/2`).

3.9.1 Static Code

In XSB, there is no difference between compiled and consulted static code: "compiling" in XSB means creation of a file containing SLG-WAM byte-code; "consulting" means loading such a byte-code file, after compiling it (if the source file was altered later than the object file).

`consult(+Files,+OptionList)`

`consult(+Files)`

`[+Files]`

The standard predicate `consult/[1,2]` is the most convenient method for entering static source code rules into XSB's database [12]. Files is either s source file designator (see Section 3.3) or a list of source file designators, and `Options` is a list of options to be passed to XSB's compiler if the file needs to be compiled (see Section 3.10). `consult(Files)` is defined as `consult(Files,[])`, as is `[Files]`.

Consulting a file `File` (module) conceptually consists of the following five steps which are described in detail in the following paragraphs.

[11] In rare cases, XSB will exit if the inability to allocate more memory will leave it in an inconsistent state (e.g. if XSB cannot allocate needed memory during heap garbage collection).

[12] In XSB, `reconsult/[1,2]` is defined to have the same actions as `consult/[1,2]`.

Name Resolution: determine the file that `File` designates, including directory and drive location and extension, as discussed in Section 3.3.

Compilation: if the source file or header has changed later than the object file (or if there is no byte-code file) compile the file using `compile/2` with the options specified, creating a byte-code file. This strategy is used whether the source file is Prolog, C, or C++.

Loading: load the byte-code file into memory.

Importing: if the file is a module, import any exported predicates of that module to `usermod`.

Query Execution: execute any queries that the file may contain, i.e. any terms with principal functor `'?-'/1`, or with the principal functor `':-'/1` and that are not directives like the ones described in Section 3.10. The queries are executed in the order in which they appear in the source file.

Error conditions for `consult(+File,+Options)` are as follows:

- `File` is not instantiated
  - `instantiation_error`
- `File` is not an atom
  - `type_error(atom,File)`
- `File` does not exist in the current set of library directories
  - `existence_error(file,File)`
- `File` has an object code extension (e.g. `.xwam`)
  - `permission_error(compile,file,File)`
- `File` has been loaded previously in the session *and* there is more than one active thread.
  - `misc_error`

Error conditions of compiler options are determined by `compile/2` which `consult/[1,2]` calls.

In addition, `ensure_loaded/[1,2]` acts much like `consult/[1,2]`

`ensure_loaded(+FileName)` ISO

This predicate checks to see whether the object file for `FileName` is newer than the source code and header files for `FileName`, and compiles `FileName` if not. If

`FileName` is loaded into memory, `ensure_loaded/1` does not reload it, unlike `consult/1` which will always reload. In addition, `ensure_loaded/2` can be used to load a file with dynamic code. It is fully documented in Section 6.14.1.

3.9.2 Dynamic Code

In XSB, most source code file can also be "consulted" dynamically via the predicates `load_dyn/[1,2]`, `load_dync/[1,2]` and `ensure_loaded/2`. These predicates act as `consult/2` in that if a given file `File` has already been dynamically loaded, old versions of predicates defined in `File` will be retracted and their new definitions made to correspond to those in `File` (except for predicates in which a `multifile/1` declaration is present in `File`). Dynamic loading can be performed using XSB's reader of canonical terms (which does not include operators, but does allow list and comma-list notation) via `load_dync/2`; dynamic loading using XSB's general reader for Hilog terms is performed via `load_dyn/2`.

The predicates mentioned above are described more fully in Chapter 6. Here, we simply compare the tradeoffs of static and dynamic loading.

- Advantages for Dynamic Loading
  - For large files, containing $10^4 - 10^7$ clauses, dynamic loading is much faster than XSB's compiler, especially when the canonical reader is used.
  - Dynamically loaded files have advantages of dynamic code including star-, trie, compound, and alternate indexes, as well as being modifiable via assert and retract.
- Advantages for Static Compilation
  - Although dynamically loaded predicates are compiled into SLG-WAM code, compiled static clauses are more optimized than dynamically predicates, particularly when the clauses have large bodies or when arithmetic is used. For facts and pure binary predicates (those containing a single literal in their body) however, static and dynamic byte code is essentially the same.
  - Dynamic loading does not allow module/export declarations, mode declarations, or unification factoring. It does however, allow files to import predicates, allows tabling and dynamic declarations (except for `auto_table` and `suppl_table`, and operator declarations (when a canonical read is not used).

3.9.3 The multifile directive

The default action upon loading a file or module is to delete all 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. *If a predicate P is to be treated as multifile, the* `multifile/1` *directive for P must appear in all files that contain clause definitions for P.* If P is dynamic, this means that the multifile declaration for P must appear in files defining P whether they are compiled and consulted, or dynamically loaded via `load_dyn/[1,2]` or `load_dync/[1,2]`.

3.10 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/1` compiler option).

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

3.10.1 Invoking the Compiler

In addition to invoking the compiler through `consult/[1,2]`, the compiler can be invoked directly at the interpreter level (or in a program) through the Prolog predicates `compile/[1,2]`.

```
compile(+Files,+OptionList)
```

`compile(+Files)`

`compile/2` compiles all files specified, using the compiler options specified in `OptionList` (see Section 3.10.2 below for the precise details.) `Files` is either an absolute or relative filename, or a ground list of absolute or relative file names; and `OptionList` is a ground list of compiler options. Since options can be set globally via the predicate `set_global_compiler_options/1`, each option in `OptionsList` can optionally be prefixed by `+` or `-`, indicating that the option is to be turned on, or off, respectively. (No prefix turns the option on.)

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

is just a notational shorthand for the query:

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

For a given, `File` to be compiled, the source file name corresponding to `File` is obtained by concatenating a directory prefix and the extension `.P`, `.pl`, `.prolog`, or other filenames as discussed in Section 3.3. The directory prefix must be in the dynamic loader path (see Section 3.6). Note that these directories are searched in a predetermined order (see Section 3.6), 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`, `File.pl`, `File.prolog`, or other extensions 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.

The result of the compilation (an SLG-WAM object code file) is stored in ($\langle filename \rangle$.xwam), but `compile/[1,2]` does *not* load the object file it creates. (The standard predicate `consult/[1,2]` loads the object file into the system, after recompiling the source file if needed.) The object file created is always written into the directory where the source file resides: the user must therefore have write permission in that directory to avoid an error.

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 11), but it can also be done for clauses of any form by providing a definition for predicate `term_expansion/2` (see Section 11.3).

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).

Error Cases In the cases below, `File` refers to an element of `Files` if `Files` is a list and otherwise refers to `Files` itself.

- `Files` is a variable, or a list containing a variable element.
  - `instantiation_error`.
- `File` is a neither an atom nor a list of atoms.
  - `type_error(atom_or_list_of_atoms,File)`
- `File` does not exist in the current set of library directories
  - `existence_error(file,File)`
- `File` has an object code extension (e.g. `.xwam`)
  - `permission_error(compile,file,File)`
- `File` has been loaded previously in the session *and* there is more than one active thread.
  - `misc_error`
- `OptionList` is a partial list or contains an option that is a variable
  - `instantiation_error`
- `OptionList` is neither a list nor a partial list
  - `type_error(list,OptionsList)`
- `OptionList` contains an option, `Option` not described in Section 3.10.2
  - `domain_error(xsb_compiler_option,Option)`

3.10.2 Compiler Options

Compiler options can be set in three ways: from a global list of options (`set_global_compiler_options/1`) from the compilation command (`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 they are reset by another call to this predicate, 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:

`singleton_warnings_off` Does not print out any warnings for singleton variables during compilation. This option can be useful for compiling XSB programs that have been generated by some other program.

`optimize` When specified, the compiler tries to optimize the object code. In Version 3.8, 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. 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.

`allow_redefinition` By default the compiler refuses to compile a file that contains clauses that would redefine a standard predicate (unless the `sysmod` option is in effect.) By specifying this option, the user can direct the compiler to quietly allow redefinition of standard predicates.

`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 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 `gpp_include_dir/1`, **which must be imported from**

module `parse` [13]. For example if you want additional directories to be searched, then the following statements must be executed:

```
:- import gpp_include_dir/1 from parse.
:- assert(gpp_include_dir('some-other-dir')).
```

Note that when compiling XSB programs, GPP searches the current directory and the directory of the parent file that contains the include-directive *last*. If you want Gpp to search directories in a different order, `gpp_options/1` can be used (see below).

Note: if you assert something into `gpp_include_dir/1` then you must also execute `retractall(gpp_include_dir(_))` later on or else subsequent Prolog compilations might not work correctly.

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
```

`gpp_options` This dynamic predicate must be imported from module `parse`. If some atom is asserted into `gpp_options` then this atom is assumed to be the list of command line options to be used by the preprocessor (only the first asserted atom is ever considered). If this predicate is empty, then the default list of options is used (which is `'-P -m -nostdinc -nocurinc'`, meaning: use Prolog mode and do not search the standard C directories and the directory of the parent file that contains the include-instruction).

As mentioned earlier, when XSB invokes Gpp, it uses the option `-nocurinc` so that Gpp will not search the directory of the parent file. If a particular application requires that the parent file directory must be searched, then this can be accomplished by executing `assert(gpp_options('-P -m -nostdinc'))`.

[13] For compatibility, XSB also supports the ISO predicate `include/1` which also allows extra files to be included during compilation.

Note: if you assert options into `gpp_options/1` then do not forget to also execute `retractall(gpp_options(_))` after that or else subsequent Prolog compilations might not work correctly.

`xpp_dump` This causes XSB to dump the output from the GPP preprocessor into a file. If the file being compiled is named `file.P` then the dump file is named `file.P_gpp`. This option can be included in the list of options in the `compiler_options/1` directive, but usually it is used for debugging, as part of the `compile/2` predicate. If `xpp_dump` is specified directly in the file using `compiler_options/1` directive, then it should *not* follow the `gpp_on` option in the list (or else it will be ignored).

Note: multiple occurrences of `xpp_on` and `xpp_dump` options are allowed, but only the *first one* takes effect—all the rest are ignored!

`xpp_on/N` **and** `xpp_dump/N`
XSB also allows one to filter program files through a pipeline of external preprocessors in addition to or instead of GPP. This can be specified with the N-ary versions of `xpp_on` and `xpp_dump`:

```
xpp_on(spec1,...,specN)
xpp_dump(spec1,...,specN)
```

Each `spec1`, ..., `specN` is a preprocessor specification of the form `preprocessor_name` or `preprocessor_name(options)`. Each preprocessor is applied in a pipeline passing its output to the next preprocessor. The first preprocessor is applied to the file being compiled. The preprocessor name is an atom or a function symbol and `options` must be an atom. If `preprocessor_name` is `gpp`, then the GPP preprocessor will be invoked. Note that `gpp` can appear anywhere in the aforesaid sequence of specs (or not appear at all), so it is possible to preprocess XSB files before and/or after (or instead of) GPP. Note that `xpp_on(gpp)` and `xpp_dump(gpp)` are equivalent to the earlier 0-ary compiler options `xpp_on` and `xpp_dump`, respectively.

To use a preprocessor other than GPP two things must be done:

- A 4-ary Prolog predicate must be provided, which takes three input arguments and produces in its 4th argument a syntactically correct shell (Unix or Windows) command for invoking the preprocessor. The first preprocessor in the pipeline must be taking its input from a file, but the subsequent preprocessors must expect their input from the standard input. All preprocessors must send their results to the standard output. The arguments to the 4-ary predicate in question are:

- File: this is the XSB input file to be processed. Usually this argument is left unused (unbound), but might be useful for producing error messages or debugging.
- Preprocessor name: this is the name under which the preprocessor is *registered* (see below). It is the same as `processor_name` referred to above. This name is up to the programmer; it is to be used to refer to the preprocessor (it does not need to be related in any way to the shell-command-producing predicate or to the OS's pathname for the preprocessor).
- Options: these are the command-line options that the preprocessor might need. If the preprocessor spec mentioned above is `foo(bar)` then the preprocessor name (argument 2) would be bound to `foo` and options (argument 3) to `bar`.
- Shell command: this is the only output argument. It is supposed to be the shell command to be used to invoke the preprocessor. The shell command must *not* include the file name to be processed—that name is added automatically as the last option to the shell command.

  Special considerations for using XSB as a preprocessor. XSB can be used as a preprocessor for XSB programs by constructing a shell command that invokes XSB. However, several conventions need to be observed. First, the file to be preprocessed is automatically attached as the last argument of the aforesaid shell command, but XSB does not accept file names in that place as a command-line option (except with special flags used by XSB developers only and for other purposes). Therefore, the file name to be read and preprocessed by XSB must be passed to XSB by some other means (e.g., using the `-e "`*`command`*`"` option). In addition, the last command line option for that XSB-based command must be `-ignore`, which will cause XSB to ignore the remaining options, including the aforesaid file name.
  Also, if a preprocessor appears in the pipeline as the second preprocessor or later (i.e., after the first argument in `xpp_dump`), that preprocessor's shell command line must expect to receive the output of the preceding preprocessor *on the standard input*. Therefore, in order to serve as the second or later preprocessor in the pipeline, XSB must be invoked with the `-e "see(userin)."` option followed by a call to the predicate that would actually do the preprocessing.
  Here are a few examples. To invoke XSB as the first preprocessor in the pipeline, one could construct the following shell command (shown

below as an atom of the kind that one needs to construct in the "Shell command" argument being discussed):

`'.../xsb` *options* `-e "preprocessPred(''`*MyFile*`''),halt." --ignore'`

Note that here the file to be preprocessed, *MyFile*, needs to be passed to the preprocessing predicate as an argument. To use XSB as the second and later preprocessor in the pipeline, the appropriate command could be

`'.../xsb` *options* `-e "see(userin),preprocessPred,halt."'`

Here the file to be preprocessed will come on the standard input of XSB. No need for the `-ignore` option here because no file names would be attached at the end of this command (since the file is piped through the standard input).

In both cases, the file passed to `preprocessPred/1` or `preprocessPred/0` could be processed using `read/1` and `write_canonical/1`. The typical options that one would want to pass in both cases (to replace *options*) are

```
--noprompt --quietload --nobanner --nofeedback
```

Note that other commands might need to be executed under the `-e` option in order to bootstrap the preprocessor (e.g., additional XSB files might need to be loaded).

- The preprocessor must be *registered* using the following query:

```
:- import register_xsb_preprocessor/2 from parse.
?- register_xsb_preprocessor(preproc_name,preproc_predicate(_,_,_,_)).
```

Here the argument `preproc_name` is the user-given name for the preprocessor, while `preproc_predicate` is the 4-ary shell-command-producing predicate described earlier.

The registration query must be executed *before* the start of the preprocessing of the input XSB file. Clearly, this implies that the shell-command-producing predicate must be in a different file than the one being preprocessed.

Note: one *cannot* register the same preprocessor twice. The second time the same name is used, it is ignored. However, it *is* possible to register the same shell-command-producing predicate twice, if the user registers the these shell-command-producing predicates under different preprocessor names.

The difference between `xpp_on/N` and `xpp_dump/N` is that the latter also saves the output of each preprocessing stage in a separate file. For instance, if

the XSB file to be preprocessed is `abc.P` and the `xpp_dump/N` option has the form `xpp_dump(foo,gpp,bar)` then three files will be produced: `abc.P_foo`, `abc.P_gpp`, `abc.P_bar`, each containing the result of the respective stage in preprocessing.

Here is an example. Suppose that `foobar.P` includes the definition of the following predicate

```
make_append_cmd(_File,_Name,Options,ResultingCmd) :-
       fmt_write_string(ResultingCmd, '"/bin/cat" "%s"', arg(Options)).
```

and also has the following registration query:

```
?- parse:register_xsb_preprocessor(appendfile,make_append_cmd(_,_,_,_)).
```

Suppose that the file `abc.P` includes the following compiler directive:

```
:- compiler_options([xpp_on(appendfile('data.P'),gpp)]).
```

If the file `foobar.P` is loaded before compiling `abc.P` then the file `data.P` will be first appended to `abc.P` and then the result will be processed by GPP. The final result will be parsed and compiled by XSB.

Note that although the parameters `_File` and `_Name` are not used by `make_append_cmd/4` in our example, when this predicate is called they will be bound to `foobar.P` and `appendfile`, respectively, and could be used by the shell-command-producing predicates for various purposes.

`quit_on_error` This causes XSB to exit if compilation of a program end with an error. This option is useful when running XSB from a makefile, when it is necessary to stop the build process after an error has been detected. For instance, XSB uses this option during its own build process.

`auto_table` When specified as a compiler option, the effect is as described in Section 3.10.5. 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 3.10.5). 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 specialization of partially instantiated calls by replacing their selected clauses with the representative of these clauses, i.e. it performs *folding* whenever possible. In general specialization with replacement is correct only under certain conditions. XSB's compiler checks for sufficient conditions that guarantee correctness, and if these conditions are not met, specialization with replacement is not performed for the violating calls.

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

`unfold_off` When specified, singleton sets optimizations are not performed during specialization. This option is necessary in Version 3.8 for the specialization of `table` declarations that select only a single chain rule of the predicate.

`spec_dump` Generates a `module.spec` file, containing the result of specializing 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 69 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 and used for bootstrapping. If specified, standard predicates (see `/$XSB_DIR/syslib/std_xsb.P`) are automatically available for use only if they are primitive predicates (see the file `$XSB_DIR/syslib/machine.P` for a current listing of primitive predicates.) When compiling in this mode, non-primitive standard predicates must be explicitly imported from the appropriate system module. Also standard predicates are permitted to be defined.

`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 certain phases 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 `.xwam` 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.xwam` command-line option of the XSB emulator (see Section 3.7).

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

3.10.3 Specialization

From Version 1.4.0 on, the XSB compiler automatically performs specialization of partially instantiated calls. Specialization 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 specialized 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-specialized counterparts. This expectation is justified mainly because of the following two basic properties of the specialization algorithm:

Compile-time Clause Selection The specialized 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 specialization 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 specialized calls as well as from the specialized versions of the predicates.

Though these properties are sufficient to get the idea behind specialization, the actual specialization performed by the XSB compiler can be better understood by the following example. The example shows the specialization 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 specialized versions of predicates.

The default specialization of partially instantiated calls is without any folding of the clauses that the calls select. Using the `spec_repr` compiler option (see Section 3.10.2) specialization with replacement of the selected clauses with the representative of these clauses is performed. Using this compiler option, predicate `ordered/2` above would be specialized 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 presence 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, specialization 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 specialization transformation by using the `spec_dump` compiler option (see Section 3.10.2).

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

3.10.4 Compiler Directives

Consider a directive

```
:- foo(a).
```

That occurs in a file that is to be compiled. There are two logical interpretations of such a directive.

1. `foo(a)` is to be executed upon loading the file; or
2. `foo(a)` provides information used by the compiler in compiling the file.

By default, the interpretation of a directive is as in case (1) *except* in the case of the compiler directives listed in this section, which as their name implies, are taken to provide information to the compiler. Some of the directives, such as the `mode/1` directive, have no meaning as an executable directive, while others, such as `import/2` do. In fact as an executable directive `import/2` imports predicates into `usermod`. For such a directive, a statement beginning with `?-`, such as

```
?- import foo/1 from myfile.
```

indicates that the directive should be executed upon loading the file, and should have no meaning to the compiler. On the other hand, the statement

```
:- import foo/1 from myfile.
```

Indicates that `foo/1` terms in the file to be compiled are to be understood as `myfile:foo/1`. In other words, the statement is used by the compiler and will not be executed upon loading. For non-compiler directives the use of `?-` and `:-` has no effect — in both cases the directive is executed upon loading the file.

The following compiler directives are recognized in Version 3.8 of XSB

Including Files in a Compilation

`include(+FileName)` ISO

The ISO directive

```
:- include(FileName)
```

Causes the compiler to act as if the code from `FileName` were contained at the position where the directive was encountered. XSB's preprocessor can perform the same function via the command `#include FileName` and can support more sophisticated substitutions, but `include/1` should be used if code portability is desired.

3.10.5 Conditional Compilation

Section 3.10.2 described a way of performing conditional compilation using XSB's interation with GPP. Conditional compilation can also be done through XSB's compiler, using the directives `:- if(+Condition)`, `:- elseif(+Condition)`, `:- else`, and `:- endif`. For instance the fragment

```
:- if(current_prolog_flag(dialect,xsb)).
:- include('file2.P').
:- elseif(current_prolog_flag(dialect,swi)).
:- include('file3.P').
:- endif.
```

allows different Prolog code to be included for XSB and for another Prolog. This framework is very general: for instance, as long as `if...elseif...endif` blocks are not nested, any Prolog code can be used in the consequents of the (else)if. The condition of `if/1` or `elseif/1` can be any Prolog goal, although care should be used in selecting `Condition`. For instance, the goal

```
:- if(file_exists('file1.P')).
```

might be true during compilation, but if the object file produced by the compilation is moved, the condition might no longer be true.

`if(?Condition)`

`elseif(?Condition)`

`else`

`endif`

Directives to invoke conditional compilation as described above. If `Condition` is a "changeable" goal such as `file_exists/1`, a warning will be issued but no error will be raised.

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[14]. See Section 3.10.5 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 necessarily 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(-).
```

`#` 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(?,?,?).
  ```

[14] 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.

- 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 behavior may be witnessed in compiled code, optimized 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 ensure termination for a given program is undecidable. 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* [5]. `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 by default 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 286). 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 [15]:

`:- 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.

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

`:- 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:

```
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 3.10.2).

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. These 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.)

3.10.6 Inline Predicates

Inline predicates represent "primitive" operations in the (extended) 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. As a result, calls to inline predicates will not be trapped by the debugger, and their evaluation will not be visible during a trace of program execution. 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 3.8. 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`
`min/2`	`max/2`	`'><'/2`	`**/2`	`sign/1`
`'_$cutto'/1`	`'_$savecp'/1`	`'_$builtin'/1`		

Table 3.1: The Inline Predicates of XSB

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

3.11 A Note on ISO Compatibility

In Version 3.8, an effort has been made to ensure compatibility with the core Prolog ISO standard [37]. In this section, we summarize the differences with the ISO standard. XSB implements almost all ISO built-ins and evaluable functions, although there are semantic differences between XSB's implementation and that of the ISO standard in certain cases.

The main difference of XSB from the ISO semantics is that XSB does not support the logical update semantics for assert and retract, but instead supports an immediate semantics. XSB does, however support an ISO-like semantics for incremental tables.

Version 3.8 of XSB mostly supports the full ISO syntax for Prolog, and its I/O system is based on UTF-8 encoding, which includes ASCII as a subset of its characters. Beyond XSB's support for Hilog, most differences between ISO syntax and XSB syntax are fairly minor. However, as XSB supports only UTF-8, ISO predicates relating to different character sets, such as `char_conversion/2`, `current_char_conversion/2` and others are not supported.

A somewhat more minor difference involves XSB's implementation of ISO streams. XSB can create streams from several First class objects, including pipes, atoms, and consoles in addition to files. However by default, XSB opens streams in binary mode, rather than text mode in opposition to the ISO standard, which opens streams in text mode. This makes no difference in UNIX or LINUX, for which text and binary streams are identical, but does make a difference in Windows, where text files are processed more than binary files.

As a final point, XSB currently throws an *error/3* term in its error ball, rather than an *error/2* term.

Most other differences with the core standard are mentioned under portability notes for the various predicates.

XSB supports most new features mentioned in the revisions to the core standard [38], including `call_cleanup/2` and various library predicates such as `subsumes/2`, `numbervars/3` and so on. XSB also has strong support for the working multi-threading Prolog standard [39], and XSB has been one of the first Prologs to support this standard. However, because XSB has an atom-based module system it does *not* support the ISO standard for Prolog modules.

Chapter 4

Syntax

The syntax of XSB is based on ISO Prolog [37], although it lacks a few of the ISO standard's somewhat arcane features. Beginning with Version 3.8, XSB supports Unicode through UTF-8 atoms as described in Section 4.1.3. XSB's reader also contains extensions to support HiLog [14], which adds certain features of second-order syntax to Prolog.

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* [1] Constants are definite elementary objects, and correspond to proper nouns in natural language.

4.1.1 Integers

ISO Integers

The printed form of an integer normally consists of a sequence of digits optionally preceded by a minus sign (`'-'`), interpreted, of course, as base 10 integers. It is also possible to enter integers in other bases:

[1] This Prolog usage contradicts the usage of the word "atom" in logic as short for "atomic formula".

- `0b`*nnn* represents an integer in base 2, e.g.,

```
| ?- X = 0b110.

X = 6
```

- `0o`*nnn* represents an integer in base 8, e.g.,

```
| ?- X = 0o110.

X = 72
```

- `0x`*nnn* represents an integer in base 16, e.g.,

```
| ?- X = 0x110.

X = 272
```

Character code constants are integers of the form `0'`*nnn*, where nnn is the decimal form of any UTF-8 codepoint. E.g.,

```
| ?- 0'A = X

X = 65
```

Escape characters (cf. Section 4.1.3) can be written similarly (if this is ever needed):

```
| ?- 0'\n = X

X = 10
```

Other Integer Representations

It is also possible to enter integers in bases 2 through 36; this can be done by preceding the digit sequence 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.

Character code constants, mentioned above, can be seen as integers in "base zero".

4.1.2 Floating-point Numbers

XSB supports ISO floating-point numbers, which consist 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 an optionally signed base 10 integer.

Using these rules, examples of 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

An atom consists of a sequence of characters that follow the following rules.

- *Non-guoted Atoms* begin with the ASCII character `a-z` and are followed by a sequence of *ISO alphanumeric characters*: `a-z`, `A-Z`, `0-9`, and underscore `_`.
- *Quoted Atoms* begin and end with the ASCII character `'` and may contain any sequence of
  - Printable UTF-8 characters

- Meta-escaped quotes. E.g.,

```
| ?- X = 'a''b'.

X = a'b
```

  (Unfortunately, the current version of XSB does not support escaped quotes (\').)

- ISO escape characters and sequences
  - \b the newline character (ASCII 7).
  - \b the newline character (ASCII 8).
  - \f the form feed character (ASCII 12).
  - \n the newline character (ASCII 10).
  - \r the carriage return character (ASCII 13).
  - \t a tab character (ASCII 9).
  - \v a vertical tab character (ASCII 11).
  - Octal escapes of the form \nnn\, where nnn is the octal number corresponding to an ASCII code. E.g.,

```
| ?- write('\60\').
0
```

  - Hexidecimal escapes of the form \xnn, where nn is the hexidecimal number corresponding to an ASCII code [2]. . E.g.,

```
| ?- write('\30\').
0
```

- UTF-8 escape sequences have the form \unnnn where *nnnn* is the hexidecimal number corresponding to a UTF-8 codepoint.

- *Operator-based Atoms* are defined as any sequence from the following set of characters (except of the sequence '/*', which begins a comment):

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

  Examples of such atoms are:

```
^=..    ::=    ===
```

[2]The current version of XSB differs from the ISO specification in that hexidecimal escapes do not have a trailing slash.

- *Special Atoms* are

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

  Note that the bracket pairs are special. While `'[]'` and `'{}'` are atoms, `'['`, `']'`, `'{'`, and `'}'` are not [3].

4.1.4 Variables

Variables may be written as any sequence of (ASCII) ISO alphanumeric characters 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 clause; all of these variables are read as distinct variables.

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)
```

[3] 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)`.

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
```

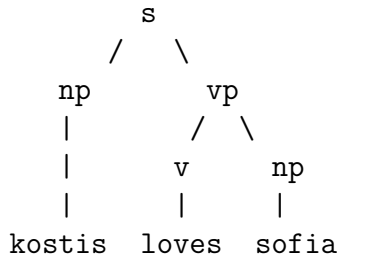
Note that a functor of arity 0 is represented as an atom.

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)

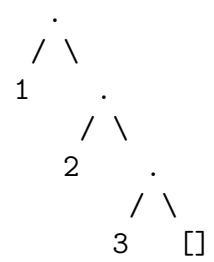
may be declared as either *prefix or postfix operators*. Thus, it is possible to write the following:

```
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,[])))
```

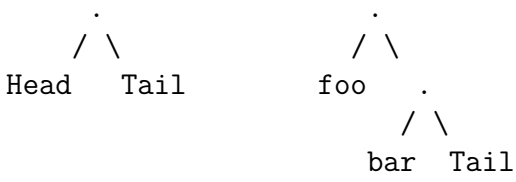
but which is normally written in a special list notation, as:

```
[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 UTF-8< character codes. Lists written in this notation are called *strings*. For example,

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

represents exactly the same list as:

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

4.2 From HiLog to Prolog

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

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

```
1 + 2
```

is just an (external) shorthand for the term:

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

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

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

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

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

Also, in the presence of a `hilog` declaration (see section 3.10.5) for `h`, the HiLog term whose external representation is:

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

is a notational shorthand for the term:

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

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

The encoding of HiLog terms is performed based upon the existing declarations of *hilog symbols*. These declarations (see section 3.10.5), 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 77 were read by XSB, the encoding of these terms in predicate calculus using the described transformation is as follows:

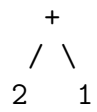
```
     foo(bar)                    prolog(a,X)
  apply(hilog,X)              apply(123,john,500)
apply(X,kostis,sofia)       apply(X,Y,Z,apply(Y,W))
  f(a,apply(b(c),d))       apply(map(double),[],[])
      apply(apply(h,apply(map(P),A,B)),C)
```

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:



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)` ISO

The same command can be used to redefine one of the predefined XSB operators (obtainable via `current_op/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. For example, the built-in operators `'+'` and `'-'` are as if they had been declared by the command:

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

so that:

```
1-2+3
```

is valid syntax, and denotes the compound term:

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

or pictorially:

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

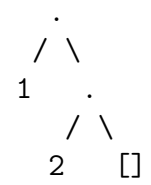
In XSB, the list functor `'.'/2` is one of the standard operators, that can be thought as declared by the command:

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

So, in XSB,

```
1.2.[]
```

represents the structure



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 Version 3.8 of XSB 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 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. [4]

[4] As a known bug, XSB's reader cannot properly read an operator defined as both a prefix and an infix operator. For instance the declaration of both `:- op(1200,xf,'<=').` and `:- op(1200,xfx,'<=').` will lead to a syntax error.

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 also use tabled resolution which can provide a different, more declarative programming style than Prolog. In this section we discuss various aspects of tabling and their implementation in XSB. Our aim in this section is to provide a user with enough information to be able to program productively with tables 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 [88], 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 contains 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 [16, 78, 71, 31]. The engine of XSB, the SLG-WAM, is an extension of the WAM [92, 1], and is described in [68, 64, 30, 70, 15, 24, 40, 19, 20, 12, 56, 81, 57, 84] as it is implemented in Version 3.8 and its performance analyzed. Examples of large-scale applications that use tabling are overviewed in [45, 46, 17, 22, 63, 7, 18, 33, 85].

5.1 Tabling in the Context of a Prolog System

Before describing how to program using tabling it is perhaps worthwhile to review some of the goals of XSB's implementation of tabling. Among them are:

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 ensure that the functionality of Prolog is not compromised by support for tabling.
4. To provide Prolog functionality in tabled predicates and operators whenever it is semantically sensible to do so.
5. To provide standard predicates to manipulate tables taken as data structures in themselves.

Goals 1 and 2 are addressed by XSB's engine, which in Version 3.8 is based on a virtual machine called the SLG-WAM. The overhead for SLD resolution using this machine is small, and usually less than 5%. 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, when compiled as a threaded interpreter (see Chapter 3) XSB Version 3.8 is about two 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 not been accomplished, or is perhaps impossible. 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.

5.2 Definite Programs

Definite programs, also called *Horn Clause Programs*, are Prolog programs without negation or aggregation. In XSB, this means without the `\+/1`, `fail_if/1`, `not/1`, `tnot/1`, `setof/3`, `bagof/3`, `tt findall/3` or other aggregation 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 an edge from `Z` to `Y`, or if there is an edge 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)` [27, 86, 4, 89, 90]. The ideas behind these algorithms can be described in the following manner. Calls to tabled predicates, such as `path(1,Y)` in the above example, are stored in a searchable structure together with their proven instances. This collection of *tabled subgoals* paired with their *answers*, generally referred to as a *table*, is consulted whenever a new call, C, to a tabled predicate is issued. If C is sufficiently similar to a tabled subgoal S, then the set of answers, $\mathcal{A}$, associated with S may be used to satisfy C. In such instances, C is resolved against the answers in $\mathcal{A}$, and hence we refer to the call C as a *consumer* of $\mathcal{A}$ (or S). If there is no such S, then C is entered into the table and is resolved against program clauses as in Prolog — i.e., using SLD resolution. As each answer is derived during this process, it is inserted into the table entry associated with C if it contains information not already in $\mathcal{A}$. In this second case, we refer to C as a *generator*, or *producer*, as resolution of C in this manner produces the answers stored in its table entry. If the answer is in fact added to this set, then it is additionally scheduled to be returned to all consumers of C. If instead it is rejected as redundant, then the evaluation simply fails and backtracks to generate more answers.

Notice that since consuming subgoals resolve against unique answers rather than repeatedly against program clauses, tabling will terminate whenever

1. a finite number of subgoals are encountered during query evaluation, and
2. each of these subgoals has 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.

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

`:- table` $P_1, \ldots, P_n$.

where each P_i is a predicate indicator or callable term. More generally

`:- table` $P_1, \ldots, P_n$ `as` *Options*.

allows a user to specify different types of tabling through `Options` along with other properties of the designated predicates For static predicates, these directives must be added to the file containing the clauses of the predicate(s) to be tabled, and the directives cause the predicates to be compiled with tabling [1]. For dynamic predicates, the executable directives

`?- table` $P_1, \ldots P_n$.

and

`?- table` $P_1, \ldots, P_n$ `as` *Options*.

cause a P_i to be tabled (with the appropriate options) if no clauses have been asserted for P_i.

Exercises Unless otherwise noted, the file `$XSB_DIR/examples/table_examples.P` contains all the code for the running examples in this section. Invoke XSB with its default settings (i.e., don't supply additional options) when working through the following exercises.

Exercise 5.2.1 *Consult* `$XSB_DIR/examples/table_examples.P` *into XSB and and try the goal*

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

and continue typing `;<RETURN>` *until you have exhausted all answers. Now, try rewriting the* `path/2` *predicate as it would be written in Prolog — and without a tabling declaration. 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 5.2.2 *If you are* still *curious, load in the file* `cyl.P` *in the* `$XSB_DIR/examples` *directory using the command.*

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

[1] In Version 3.8, tabling does not work together with multi-file predicates.

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`*).* □

Exercise 5.2.3 *The file* `table_examples.P` *contains a set of facts*

```
ordered_goal(one).
ordered_goal(two).
ordered_goal(three).
ordered_goal(four).
```

Clearly, the query `?- ordered_goal(X)` *will return the answers in the expected order.* `table_examples.P` *also contains a predicate*

```
:- table table_ordered_goal/1.
table_ordered_goal(X):- ordered_goal(X).
```

which simply calls `ordered_goal/1` *and tables its answers (tabling is unnecessary in this case, and is only used for illustration). Call the query* `?- table_ordered_goal(X)` *and backtrack through the answers. In what order are the answers returned?*

The examples stress two differences between tabling and SLD resolution beyond termination properties. First, that each solution to a 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 the table, answers may be returned in an unaccustomed order.

Tabling Dynamic Predicates Dynamic predicates may be tabled just as static predicates, as the following exercise shows.

Exercise 5.2.4 *For instance, restart XSB and at the 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.* □

In general, as long as the directive `table/1` is executed before asserting (or dynamically loading) the predicates referred to in the directive, any dynamic predicate can be tabled.

Letting XSB Decide What to Table Other tabling declarations are also provided. 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 3.10.2. [2].

5.2.1 Call Variance vs. Call Subsumption

The above description gives a general characterization of tabled evaluation for definite programs but glosses over certain details. In particular, we have not specified the criteria for

- *Call Similarity* – whereby a newly issued subgoal S is determined to be "sufficiently similar" to a tabled subgoal S_{tab} so that S can use the answers from the table of S_{tab} rather than re-deriving its own answers. In the first case where S uses answers of a tabled subgoal it is termed a consumer; in the second case when S produces its own answers it is called a generator or producer.
- *Answer Similarity* – whereby a derived answer to a tabled subgoal is determined to contain information similar to that already in the set of answers for that subgoal.

Different measures of similarity are possible. XSB's engine supports two measures for call similarity: variance and subsumption. XSB's engine supports a variance-based measure for answer similarity, but allows users to program other measures in certain cases. We discuss call similarity here, but defer the discussion of answer similarity until Section 5.4.

[2]The reader may have noted that `table/1`, is referred to as a *directive*, while `auto_table/0` and `suppl_table/0` were referred to as *declarations*. The difference is that at the command line, user can execute a directive but not a compiler declaration.

Determining Call Similarity via Variance By default, XSB determines that a subgoal S is similar to a tabled subgoal S_{tab} if S is a *variant* of S_{tab}, that is if S and S_{tab} are identical up to variable renaming [3]. As an example `p(X,Y,X)` is a variant of `p(A,B,A)`, but not of `p(X,Y,Y)`, or `p(X,Y,Z)`. Under variance-based call similarity, or *call variance*, when a tabled subgoal S is encountered, a search for a table entry containing a variant subgoal S_{tab} is performed. Notice that if S_{tab} exists, then *all* of its answers are also answers to S, and therefore will be resolved against it. Call variance was used in the original formulation of SLG resolution [16] for the evaluation of normal logic programs according to the well-founded semantics and interacts well with many of Prolog's extra-logical constructs.

Determining Call Similarity via Subsumption Call similarity can also be based on *call subsumption*. A term T_1 *subsumes* a term T_2 if T_2 is more specific than T_1 [4]. Furthermore, we say that T_1 *properly subsumes* T_2 if T_2 subsumes T_1, but is not a variant of T_1. Under call subsumption, when a tabled subgoal S is encountered, a search is performed for a table entry containing a *subsuming* subgoal S_{tab}. Notice that, if such an entry exists, then its answer set $\mathcal{A}$ logically contains all the solutions to satisfy C. The subset of answers $\mathcal{A}' \subseteq \mathcal{A}$ which unify with C are said to be *relevant to* C.

Notice that call subsumption permits greater reuse of computed results, thus avoiding even more program resolution, and thereby can lead to time and space performances superior to call variance. In addition, beginning with Version 3.2, call-subsumption based tabling fully supports well-founded negation under the default local scheduling strategy. However, there are downsides to this paradigm. First of all, subsumptively tabled predicates do not interact well with certain Prolog constructs with which variant-tabled predicates can (see Example 5.2.4 below). Second, call subsumption does not yet support calls with tabled attributed variables or answer subsumption [5].

Example 5.2.1 The terms T_1: `p(f(Y),X,1)` and T_2: `p(f(Z),U,1)` are *variants* as one can be made to look like the other by a renaming of the variables. Therefore, each *subsumes* the other.

[3] Formally, S and S_{tab} are variants if they have an mgu θ such that the domain and range of $\theta 1$ consists only of variables.

[4] Formally, T_1 subsumes T_2 if there is a substitution θ whose domain consists only of variables from T_1 such that $T_1\theta = T_2$.

[5] Beginning with Version 3.2, XSB supports attributed variables in answers under call subsumption, although not in calls.

The term t_3: `p(f(Y),X,1)` *subsumes* the term t_4: `p(f(Z),Z,1)`. However, they are *not variants*. Hence t_3 *properly subsumes* t_4. □

The above examples show how a variant-based tabled evaluation can reduce certain redundant subcomputations over SLD. However, even more redundancy can be eliminated, as the following example shows.

Exercise 5.2.5 Begin by abolishing all tables in XSB, and then type the following query

```
?- abolish_all_tables.
?- path(X,Y), fail.
```

Notice that only a single table entry is created during the evaluation of this query. You can check that this is the case by invoking the following query

```
?- get_calls_for_table(path/2,Call).
```

Now evaluate the query

```
?- path(1,5), fail.
```

and again check the subgoals in the table. Notice that two more have been added. Further notice that these new subgoals are *subsumed* by that of the original entry. Correspondingly, the answers derived for these newer subgoals are already present in the original entry. You can check the answers contained in a table entry by invoking `get_returns_for_call/2` on a tabled subgoal. For example:

```
?- get_returns_for_call(p(1,_),Answer).
```

Compare these answers to those of `p(X,Y)` and `p(1,5)`. Notice that the same answer can, and in this case does, appear in multiple table entries.

Now, let's again abolish all the tables and change the evaluation strategy of `path/2` to use subsumption.

```
?- abolish_all_tables.
?- table path/2 as subsumptive.
```

And re-perform the first few queries:

```
        ?- path(X,Y),fail.
?- get_calls_for_table(path/2,Call).
?- path(1,5).
?- get_calls_for_table(path/2,Call).
```

Notice that this time the table has not changed! Only a single entry is present, that for the original query `p(X,Y)`.

When using call subsumption, XSB is able to recognize a greater range of "redundant" queries and thereby make greater use of previously computed answers. The result is that less program resolution is performed and less redundancy is present in the table. However, subsumption is not a panacea. The elimination of redundant answers depends upon the presence of a subsuming subgoal in the table when the call to `p(1,5)` is made. If the order of these queries were reversed, one would find that the same entries would be present in this table as the one constructed under variant-based evaluation.

Declarations for Call Variance and Call Subsumption By default tabled predicate use call variance. However, call subsumption can be made the default by giving XSB the `-S` option at invocation (refer to Section 3.7). More versatile constructs are provided by XSB so that the tabling method can be selected on a *per predicate* basis. Use of the directive

```
table p/n as subsumptive
```

or

```
table p/n as variant
```

described in Section 6.15.1, ensures that a tabled predicate is evaluated using the desired strategy regardless of the default tabling strategy.

5.2.2 Tabling with Interned Terms

XSB supports, on request, a special representation of *ground* terms, known as interned terms (see `intern_term/2`.) This representation is also sometimes known as a "hash-consing" representation. All interned terms are stored in a global area and each such term is stored only once, with all instances of a given interned (sub-)term pointing to that one stored representation. This can allow for a much more succinct representation of sets of ground terms that share subterms. Importantly interned ground terms, in principle, do not need to be copied into and out of tables.

To take advantage of this possibility, a table must be declared as `intern`. As an example of a possible use of this mechanism, consider a simple DCG that recognizes all strings of a's starting with a single b:

```
:- table bas/2 as intern.

bas --> [b].
bas --> bas, [a].
```

This predicate must be tabled in order to terminate, since the grammar is left-recursive. If we use the usual list representation of an input string and use variant tabling, every call to `bas/2` and every return will copy the remaining list into the table, and recognition will be quadratic. (For example on my laptop, recognizing a list of one b followed by 10,000 a's takes about 1.84 seconds, and 20,000 a's about 7.285 seconds.) If we table `bas/2 as intern`, the initial ground input list will be interned (copied to intern space) on the first call, and after that every subsequent call of `bas/2` will be given an interned term, which need not be copied into (or out of) the table. In this case the complexity will be linear. (For example on my laptop, recognizing a list of one b and 1,000,000 a's takes less than a second.)

When a table is declared `as intern`, at the time of a call, all arguments are automatically interned (with `intern_term/2`) before the call is looked up in the table, and on return, every answer is interned before being added to the table. Copying an interned subterm into or out of a table requires just a pointer copy, which takes, of course, constant time.

Because an interned term is treated just like a atom (with no indexing done on its structure), tabling as intern always uses variant tabling, and thus cannot be combined with subsumptive tabling. Also it cannot be combined with answer subsumption tabling.

For more information on tabling as intern, see [93].

5.2.3 Table Scheduling Strategies

Recall that SLD resolution works by selecting a goal from a list of goals to be proved, and selecting a program clause C to resolve against that goal. During resolution of a top level goal G, if the list of unresolved goals becomes empty, G succeeds, while if there is no program clause to resolve against the selected goal from the list resolution against G fails. In Prolog clauses are selected in the order they are asserted, while literals are selected in a left-to-right selection strategy. Other strategies are possible

for SLD, and in fact completeness of SLD for definite programs depends on a non-fixed literal selection strategy. This is why Prolog, which has a fixed literal selection strategy is not complete for definite programs, even when they have bounded term-depth.

Because tabling uses program clause resolution, the two parameters of clause selection and literal selection also apply to tabling. Tabling makes use of a dynamic literal selection strategy for certain non-stratified programs (via the delaying mechanism described in Section 5.3.2), but uses the same left-to-right literal selection strategy as Prolog for definite programs. However, in tabling there is also a choice of when to return derived answers to subgoals that consume these answers. While full discussion of scheduling strategies for tabling is not covered here (see [30]) we discuss two scheduling strategies that have been implemented for XSB Version 3.8 [6].

- *Local Scheduling* Local Scheduling depends on the notion of a *subgoal dependency graph*. For the state of a tabled evaluation, a non-completed tabled subgoal S_1 directly depends on a non-completed subgoal S_2 when S_2 is in the SLG tree for S_1 – that is when S_2 is called by S_1 without any intervening tabled predicate. The edges of the subgoal dependency graph are then these direct dependency relations, so that the subgoal dependency graph is directed. As mentioned, the subgoal dependency graph reflects a given state of a tabled evaluation and so may changed as the evaluation proceeds, as new tabled subgoals are encountered, or encountered in different contexts, as tables complete, and so on. As with any directed graph, the subgoal dependency graph can be divided up into strongly connected components, consisting of tabled subgoals that depend on one another. Local scheduling then fully evaluates each maximal SCC (a SCC that does not depend on another SCC) before returning answers to any subgoal outside of the SCC [7].

- *Batched Scheduling* Unlike Local Scheduling, Batched Scheduling allows answers to be returned outside of a maximal SCC as they are derived, and thus resembles Prolog's tuple at a time scheduling.

Both Local and Batched Scheduling have their advantages, and we list points of comparison.

[6] Many other scheduling strategies are possible. For instance, [29] describes a tabling strategy implemented for the SLG-WAM that emulates magic sets under semi-naive evaluation. This scheduling strategy, however, is not available in Version 3.8 of XSB.

[7] XSB's implementation maintains a slight over-approximation of SCCs – see [30].

- *Time for left recursion* Batched Scheduling is somewhat faster than Local Scheduling for left recursion as Local Scheduling imposes overhead to prevent answers from being returned outside of a maximal SCC.
- *Time to first answer* Because Batched Scheduling returns answers out of an SCC eagerly, it is faster to derive the first answer to a tabled predicate.
- *Stack space* Local evaluation generally requires less space than batched evaluation as it fully explores a maximal SCC, completes the SCC's subgoals, reclaims space, and then moves on to a new SCC.
- *Integration with cuts* As discussed in Exercise 5.2.6 and throughout Section 5.2.4, Local Scheduling integrates better with cuts, although this is partly because tabled subgoals may be fully evaluated before the cut takes effect.
- *Efficiency for call subsumption* Because Local Evaluation completes tables earlier than Batched Evaluation it may be faster for some uses of call subsumption, as subsumed calls can make use of completed subsuming tables.
- *Negation and tabled aggregation* As will be shown below, Local Scheduling is superior for tabled aggregation as only optimal answers are returned out of a maximal SCC. Local Scheduling also can be more efficient for non-stratified negation as it may allow delayed answers that are later simplified away to avoid being propagated.

On the whole, advantages of Local Scheduling outweigh the advantages of Batched Scheduling, and for this reason Local Scheduling is the default scheduling strategy for Version 3.8 of XSB. XSB can be configured to use batched scheduling via the configuration option `-enable-batched-scheduling` and remaking XSB. This will not affect the default version of XSB, which will also remain available.

5.2.4 Interaction Between Prolog Constructs and Tabling

Tabling integrates well with most non-pure aspects of Prolog. 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 while the rest will look up answers from a table. However, other extra-logical constructs like the cut (`!`) pose greater difficulties. Tabling with call subsumption is also theoretically precluded from correct interaction with certain meta-logical predicates.

Cuts and Tabling The semantics for cuts in Prolog is largely operational, and is usually defined based on an ordered traversal of an SLD search tree. Tabling, of course, has a different operational semantics than Prolog – it uses SLG trees rather than SLD trees, for instance – so it is not surprising that the interaction of tabling with cuts is operational. In Prolog, the semantics for a cut can be expressed in the following manner: a cut executed in the body of a predicate P frames from the top (youngest end) of the choice point stack down to and including the call for P. In XSB *a cut is allowed to succeed as long as it does not cut over a choice point for a non-completed tabled subgoal, otherwise, the computation aborts.* This means, among other matters, that the validity of a cut depends on the *scheduling strategy* used for tabling, that is on the strategy used to determine when an answer is to be returned to a consuming subgoal. Scheduling strategy was discussed Section 5.2.3: for now, we assume that XSB's default local scheduling is used in the examples for cuts.

Exercise 5.2.6 *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).
```

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

```
cut_p(X) :- cut_q(X), once(cut_r).
```

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 5.2.6, `cut_p(1)` and `cut_p(2)` should both be true. Thus, the cut in the literal `once(cut_r)` must not inadvertently cut away solutions that are demanded by `cut_p/1`. In the default local scheduling of XSB Version 3.8 tabled subgoals are fully evaluated whenever possible before returning any of their answers. Thus the first call `cut_q(X)` in the body of the clause for `cut_p/1` is fully evaluated before proceeding to the goal `once(cut_r)`. Because of this any choice points for `cut_q(X)` are to a completed table. For other scheduling strategies, such as batched scheduling, non-completed choice points for `cut_p/1` may be present on the choice point stack so that the cut would be disallowed. In addition, it is also possible to construct examples where a cut is allowed if call variance is used, but not if call subsumption is used.

Example 5.2.2 *A further 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 XSB's compilation, the cut above is compiled so that it is valid to use with either local or batched (a non-default) evaluation. An example of a cut that is valid neither in batched nor in local evaluation is as follows.

Example 5.2.3 *Consider the program*

```
:- table cut_a/1, cut_b/1.

cut_a(X):- cut_b(X).
cut_a(a1).

cut_b(X):- cut_a(X).
cut_b(b1).
```

For this program the goal `?- cut_a(X)` *produces two answers, as expected:* `a1` *and* `b1`*. However, replacing the first class of the above program with*

```
cut_a(X):- once(cut_b(X)).
```

will abort both in batched or in local evaluation. □

To summarize, the behavior of cuts with tables depends on dynamic operational properties, and we have seen examples of programs in which a cut is valid in both local and batched scheduling, in local but not batched scheduling, and in neither batched nor local scheduling. In general, any program and goal that allows cuts in batched scheduling will allow them in local scheduling as well, and there are programs for which cuts are allowed in local scheduling but not in batched.

Finally, we note that in Version 3.8 of XSB a "cut" over tables implicitly occurs when the user makes a call to a tabled predicate from the interpreter level, but does not generate all solutions. This commonly occurs in batched scheduling, but can also occur in local scheduling if an exception occurs. 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.

Call Subumption and Meta-Logical Predicates Meta-logical predicates like `var/1` can be used to filter the choices made during an evaluation. However, this is dangerous when used in conjunction with call subsumption, since call subsumption assumes that if a specific relation holds — e.g., `p(a)` — then a more general query — e.g., `p(X)` — will also hold.

Example 5.2.4 *Consider the following simple program*

```
p(X) :- var(X), X = a.
```

to which the queries

```
?- p(X).
?- p(a).
```

are posed. Let us compare the outcome of these queries when `p/1` *is (1) a Prolog predicate, (2) a variant-tabled predicate, and (3) a subsumptive-tabled predicate.*

Both Prolog and variant-based tabling yield the same solutions: `X = a` *and* `no`*, respectively. Under call subsumption, the query* `?- p(X).` *likewise results in the solution* `X = a`*. However, the query* `?- p(a).` *is subsumed by the tabled subgoal* `p(X)` *— which was entered into the table when that query was issued — resulting in the incorrect answer* `yes`*.* □

As this example shows, *incorrect answers* can result from using meta-logical with subsumptive predicates in this way.

5.2.5 Potential Pitfalls in Tabling

Over-Tabling While the judicious use of tabling can make some programs faster, its indiscriminate use can make other programs slower. Naively tabling `append/3`

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

is one such example. Doing so 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 5.2.7 *If you need convincing that tabling can sometimes slow a query down, type the query:*

```
?- 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).
```

leaving off the call to `genlist/2`*, and backtrack through a few answers. Will* `table_append/3` *ever succeed for this predicate? Why not?*

Suppose DCG predicates (Section 11) are defined to be tabled. How is this similar to tabling append? □

We note that XSB has special mechanisms for handling tabled DCGs. See Section 11 for details.

Tabled Predicates and Tracing 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 10). When tabled predicates are traced, these events may occur in unexpected ways, as the following example shows.

Exercise 5.2.8 *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? □

5.3 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 (we note that in Version 3.8 the well-founded semantics is implemented only for variant-based tabling).

5.3.1 Stratified Normal 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. To see this, load the files* `tree1k.P` *and* `table_examples.P` *from the directory* `$XSB_DIR/examples` *and type 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 [8].

Completely Evaluated Subgoals Knowing when a subgoal is completely evaluated can be useful when programming with tabling. Simply put, a subgoal S is

[8] LRD-stratifiedstratification may be reminiscent of the Subgoal Dependency Graphs of Section 5.2.3 but differ in several respects, most notably in that stratification considers only cycles through negative dependencies.

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 [42, 75]. 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 6.15).

There are in fact two ways that a tabled subgoal S can be determined to be completely evaluated. If S is part of an SCC $\mathcal{S}$, (a mutually recorsive component), then S can be completed once it is ensure that all resolution steps have been done to all subgoals in $\mathcal{S}$. Otherwise, if there is a derivation of an answer that is identical to S, S can be completed before the rest of the subgoals in $\mathcal{S}$ since further evaluation of S itself will not produce useful information. In this case, we sometimes say that S is *early completed*.

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 ensure that XSB evaluates ground queries to Datalog programs with negation with polynomial data complexity. As seen [16], 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. [53]), but the following program illustrates a case in which non-ground negation is sound.*

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

One tabled analog is

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

ngr_tp:- tnot(ngr_tp(_)).
```

Version 3.8 of XSB will flounder on the call to `ngr_tp`*, but not on the call to* `ngr_p/0`*. On the other hand if* `not_exists/1` *is used*

```
ngr_skp:- not_exists(ngr_tp(_)).
```

the non-ground semantics is allowed.

`not_exists/1` works by asserting a new tabled subgoal, abstractly

```
:- table '_$ngr_tp'
'_$skolem_ngr_tp' :- ngr_tp(_).
```

to avoid the problem with variables. In addition, since `not_exists/1` creates a new tabled predicate, it can be used to call non-tabled predicates as well, ensuring tabling.

The description of `tnot/1` in Section 6.5 describes other small differences between $'\backslash +'/1$ 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 238).

5.3.2 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 6.15) 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.
```

Try the query `?- simpl_p(X)`. *If you have a copy of XSB defined using Batched Scheduling load the examples program and query* `?- simpl_p(X)` *– be sure to backtrack through all possible answers. Now try the query again. What could possibly account for the difference in behavior between Local and Batched Scheduling?*

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, it *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 omitted delay lists when discussing such programs.

In the above program, delaying occurs for the negative literals in clauses 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 answer 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 3.8 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* [62]. 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 [71] XSB is *delay minimal*.

Programming in the Well-founded Semantics XSB delays literals for non-LRD-stratified programs and later simplifies them away. In Local Scheduling, all simplification will be done before the first answer is returned to the user. In Batched Scheduling it is usually better 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 program clause 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 ac_p/1,ac_r/0,ac_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

```
ac_p(X):- ac_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 3.8 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.

5.3.3 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) [3], 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 [47], 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 [79] 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 [60], an efficient system has been devised for model checking of concurrent systems that is 10-20 times faster than competing systems [52]. In addition, using the XASP package (see the separate manual, [13] in XSB's packages directory) a consistency checker for description logics has also been created [80].

5.4 Answer Subsumption

By default XSB adds an answer A to a table T only if A is not a variant of some other answer already in T, a technique termed *answer variance*. While answer variance is sufficient to allow tabling to compute the well-founded semantics and to terminate for programs with bounded term-depth, other choices of when and how to add an answer can be made. Using *partial order answer subsumption*, A would be added to T only if A is maximal with respect to other answers in T according to a given partial order $>_O$. Furthermore if A is added, any answers in T that A subsumes (i.e., is greater than in $>_O$) are deleted. When using *lattice answer subsumption*, A itself may not be added to T, rather the join is taken of A and another answer A' in T, with A' being deleted. Despite its conceptual simplicity, answer subsumption can be a powerful tool. Partial order answer subsumption allows a table to retain only answers that are maximal according to a metric or to a preference relation; lattice answer subsumption can form the basis of multi-valued logics, quantitative logics, and of abstract interpretations for programs and process logics.

5.4.1 Types of Answer Subsumption

Partial Order Answer Subsumption.

We illustrate the use of partial order answer subsumption through a shortest-path predicate (Figure 5.1) that counts the number of edges between two vertices.

```
sp(X,Y,1):- edge(X,Y).
sp(X,Z,N):- sp(X,Y,N1),edge(Y,Z),N is N1 + 1.
```

Figure 5.1: A Shortest Path Predicate

As mentioned above, partial-order answer subsumption retains in a table T only those answers that are maximal according to a given partial order $>_O$. In the case of the shortest-path predicate of Figure 5.1, $sp(A_1, A_2, A_3) >_O sp(B_1, B_2, B_3)$ if, $A_1 = B_1$, $A_2 = B_2$, and $A_3 < B_3$. Note that that minimal distances are maximal in $<_O$, and that $<_O$ is undefined if A_3 or B_3 is non-numeric. In XSB, partial order answer subsumption is specified for `sp/3` using the declaration

```
:- table sp(_,_,po((<)/2)).
```

In a given state of computation, only those answers that are maximal according to $>_O$ are available for resolution. Thus, for a finite graph with cycles, `sp/3` will terminate

using answer subsumption, but not with answer variance. Other partial orders beyond distance metrics may be useful. For instance, $>_O$ may specify a preference ordering between derived atoms so that answer subsumption provides an alternative to default-based methods for computing preferences.

The treatment of variables in calls to partial order answer subsumptive tabled predicates deserves mention. Variables in arguments not in the subsumption position are treated as "group-by" variables: i.e., for each value such a variable can take, a different aggregate is computed. So for example a call to `sp(a,X,SD)` will succeed for each node reachable from `a`, binding `X` to that node and `SD` to the shortest distance from `a` to that node. One can place a `^` in a non-subsumption position of table declaration, e.g.,

```
:- table sp(_,^,po((<)/2)).
```

to indicate that values of that position should be aggegated over. For example, with this table declaration, the call `sp(a,X,SD)` will find the distance to the closest node reachable from `a`, (which, if `a` has any successors, will be 1, since a successor to `a` will be a nearest reachable successor at distance 1 from `a`.)

Non-variables in the subsumption position in a call will be treated as selecting what answers are included in the aggregation.

Lattice Answer Subsumption.

An upper semi-lattice is a partial order for which any two elements have a unique least upper bound. Because the ordering for the third argument of `sp/3` is total, it also forms an upper semi-lattice, and so can be computed using lattice answer subsumption. [9]. In XSB lattice answer subsumption for `sp/3` is declared as

```
:- table sp(_,_,lattice(min/3)).
```

with `min/3` defined as `min(X,Y,Z):- Z is min(X,Y).` Operationally, this means that whenever an answer $sp(A_1, A_2, A_3)$ is derived, if there is another answer $sp(B_1, B_2, B_3)$ where $A_1 = B_1$ and $A_2 = B_2$ the join J_3 of A_3 and B_3 is taken, and only $sp(A_1, A_2, J_3)$ is available for resolution. As with a partial order, the join operation ensures termination for shortest path over a finite graph with cycles.

As the following proposition shows, lattice answer subsumption can be modeled either starting with a lattice, or starting with a function with appropriate properties.

[9] The terminology lattice answer subsumption is employed even though only the join of the lattice is used.

Proposition 5.4.1 *Let* op *be an associative, commutative, and idempotent binary function. Then there is a partial order P, such that P is an upper semi-lattice with join* op.

Conversely, if a function does not have the above properties, it is not suitable for lattice answer subsumption. Accordingly the aggregate functions count and sum cannot be computed using lattice answer subsumption [10]. Lattice answer subsumption has a variety of applications. [84] shows how it is used for social-network analysis and Section 5.4.2 shows its use for an application of multi-valued logics, [79] describes how a similar formalism can implement a quantitative logic, and [65, 66] describes how XSB's `PITA` package is based on answer subsumption (see Volume 2 of this manual).

Partial Order Answer Subsumption with Abstraction.

Computation over an abstract domain may require certain maximal answers to be abstracted. In many cases, abstraction can be modeled by a join operation, but in others the abstraction represents an implicit induction step in the following sense. Given a set $\mathcal{A}$ of answers, it may be detected that the program computed does not have a finite model. An abstraction operation then is applied so that $\mathcal{A}$ and its extensions can be symbolically represented by a single answer A. Using answer subsumption, this abstraction can be taken only if needed during program execution. Abstractly, partial order answer subsumption with abstraction uses the declaration

```
:- table p(_,_,po(rel/2,abs/3)).
```

where `rel/2` is a partial order, and `abs/3` is the abstraction operation. Section 5.4.2 provides a detailed example of how such an approach is used to analyze a process logic.

5.4.2 Examples of Answer Subsumption

Answer Subsumption and Abstract Interpretation

Net-style formalisms, such as Petri Nets, Workflow Nets, etc. have been used extensively for process modeling. Reachability is a central problem in analyzing properties

[10] Since count and sum are not idempotent their semantics is based on multi-sets, rather than sets. Incorporating these as tabling features requires modifying their semantics to be set-based, in a manner similar to aggregation ASP systems.

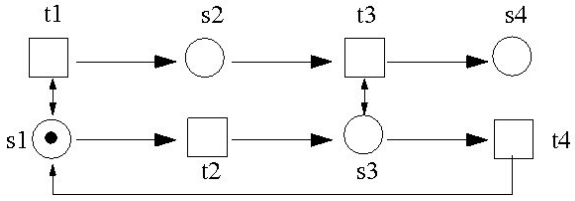


Figure 5.2: A PT-net and configuration with an infinite number of reachable configurations

of such nets, to which properties such as liveness, deadlock-freedom, and the existence of home states can be reduced. However, many interesting net formalisms cannot guarantee a finite number of configurations in a given net, so abstraction methods must be applied for their analysis.

For instance, the lack of finiteness is a problem in analyzing Place/Transition (PT) Nets. PT nets have no guard conditions or after-effects, and do not distinguish between token types. However, PT nets do allow a place to hold more than one token, leading to a potentially infinite number of configurations. This can be seen in the simple network of Figure 5.2 (from [26]) in which transitions are denoted by squares and places by circles. Each transition removes one token from the places that are the sources of its input edges and adds one token to each place at the target of each of its output edges. Starting from the configuration in Figure 5.2, repeated application of transition `t1` leads to place `s2` containing an unbounded number of tokens; repeated application of the sequence `t1,t2,t3,t4` leads to place `s4` containing an unbounded number of tokens.

Despite such examples, reachability in PT nets is decidable and can be determined using an abstraction method called ω-sequences, (see e.g. [26]). The main idea in determining ω sequences is to define a partial order $\geq_\omega$ on configurations as follows. If configurations C_1 and C_2 are both reachable, C_1 and C_2 have tokens in the same set PL of places, C_1 has at least as many tokens in each place as C_2, and there exists a non-empty $PL_{sub} \subseteq PL$, such that for each $pl \in Pl_{sub}$ C_1 has strictly more tokens than C_2, then $C_1 >_\omega C_2$. When evaluating reachability, if C_2 is reached first, and then C_1 was subsequently reached, C_1 is abstracted by marking each place in PL_{sub} with the special token ω which is taken to be greater than any integer. If C_1 was reached first and then C_2, C_2 is treated as having already been seen.

Tabling combined with partial order answer subsumption requires slightly over 100 lines of code to model reachability in PT nets using ω-sequences. Due to space restrictions, the program cannot be fully described here, but the top-level reachability predicate is shown in Figure 5.3. Despite its succinctness, it can evaluate reachability

```
:- table reachable(_,po(omega_gte/2,omega_abs/3)).
reachable(InConf,NewConf):-
        reachable(InConf,NewConf),
        hasTransition(Conf,NewConf).
reachable(InConf,NewConf):- hasTransition(InConf,NewConf).
```

Figure 5.3: Top-level predicate for PT net reachability

in networks with millions of states in a few minutes. This use of tabling to determine reachability in PT nets can be seen as a special case of tabling for abstract interpretation (cf. [41] and other works). However the framework for answer subsumption described here allows tabling to be used to efficiently perform abstract interpretation within a general Prolog system

Scalability for multi-valued and quantitative logics

The technique of program justification (cf. e.g. [61]) has been used for debugging tabled programs that cannot be debugged by traditional means. Here, we consider justification in the context of the Silk system, currently under development at Vulcan, Inc. Silk is a commercial knowledge representation and rule system built on top of Flora-2, which is implemented using XSB. One of the salient features of Silk is its default reasoning, which is based on a parameterized argumentation theory evaluated under the well-founded semantics [91]. One issue in using Silk is that knowledge engineers must have a way of understanding the reasoning of the system, a task complicated by the use of the well-founded semantics and the intricacies of the argumentation theory. We describe an experimental approach to justification of Silk-style argumentation theories using multi-valued logics.

As noted in [91], argumentation theories in Silk are usually extensions of the default theories of Courteous Logic Programs (CLP) and are based on two user-defined predicates: `opposes/2` and `overrides/2`. Two atoms *oppose* each other if no model of a program can contain both atoms: an atom and its explicit negation oppose each other, but opposition can capture many other types of contradictions. Given two opposing atoms, one atom may *override* the other, and so be given preference. For atoms A_1 and A_2, if A_1 and A_2 are both derivable and oppose each other but neither overrides the other, A_1 and A_2 mutually *rebut* each other. If in addition A_1, say, overrides A_2, A_1 *refutes* A_2 [11]. Within Silk and Flora-2, the compilation of an argumentation theory ensures that rebutted atoms have an undefined truth value, as

[11] In [91] argumentation theories are built on named rules, here we base them on derived atoms.

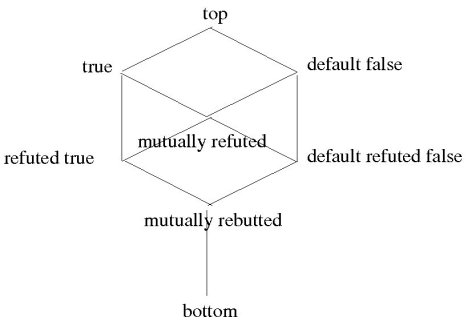


Figure 5.4: A Truth Lattice for a Simplified Version of Courteous Argumentation Theory

do atoms that refute themselves (i.e. if the `overrides/2` predicate is cyclic). However, for justification, it is meaningful to distinguish those facts that are undefined due to a negative loop in the argumentation theory from those that are undefined due to a negative loop in the program itself. In addition, it is meaningful to distinguish an atom that is true because it overrides some other atom, from an atom whose derivation does not depend on the argumentation theory. Similar distinctions can be made for default false literals leading to the truth lattice shown in Figure 5.4.

5.4.3 Term-Sets

XSB provides support for a programming technique for representing sets of terms, called term-sets. (While it is not closely related to answer subsumption, it is partially implemented through tabling and a table declaration, and so this facility is documented here.)

We begin in an example. We can represent a set of Prolog terms by using a particular term of the form `{Var:Goal}` where Goal has (only) `Var` free in it. Then we will use this *set-term* to represent the set of terms obtained by evaluating `Goal` and taking the values of `Var` that are obtained. I.e., they would be the terms in the list `L` returned by the Prolog call to `setof(Var,Goal,L)`. For example, the set-term:

```
{X : member(X,[a,b,c])}
```

represents the set of terms `{a,b,c}`.

Now a *term-set* is a Prolog term that may contain set-terms as subterms. For example,

```
m({X:member(X,[a,b,c])},g(d,{Y:member(Y,[e,f,g])}),h)
```

is a term-set, and it represents the set of terms obtained from it by replacing (recursively) any embedded set-term by a term in that set-term. So the above term-set represents the 9 terms:

```
m(a,g(d,e),h)        m(a,g(d,f),h)        m(a,g(d,g),h)
m(b,g(d,e),h)        m(b,g(d,f),h)        m(b,g(d,g),h)
m(c,g(d,e),h)        m(c,g(d,f),h)        m(c,g(d,g),h)
```

This example shows an advantage of this representation. Say a term-set has k subset-terms each of which is of the member form in this example where each member has a list of atoms of length n. To represent this set of terms explicitly takes $O(n^k)$ space, whereas to represent them with the term-set takes only $O(n \times k)$ space. So a term-set representation can take exponentially less space than an explicit representation.

It is relatively easy to write a predicate, `member_termset/2`, which takes a variable and a term-set and nondeterministically generates all concrete terms represented by the term-set, called *extensionalizing* the term-set. Some care must be taken since a call to goal to extensionalize a set-term may itself return a term-set. Also term-sets can be self-recursive and thus represent infinitely many Prolog terms. For example, consider the term-set:

```
{X : p(X)} where
  p(a).
  p(f({X:p(X)})).
```

This term-set represents the terms for which `p/1` is true. Now `p(a)` is true, so `a` is in the term-set. Since `a` is in `{X:p(X)}`, then `p(f(a))` is true because of the second fact for `p/1`, and so `f(a)` is in the term-set. And so on. So this term-set contains the infinitely many terms:

```
a, f(a), f(f(a)), f(f(f(a))), ...
```

A particularly interesting use of term-sets is in conjunction with tabling. Consider the term-set `{X:p(1,2,X)}` where `p/3` is tabled. If `p(1,2,_}` has been called and so its table is filled, then extensionalizing this term-set requires just a table lookup; in some sense we can think of such a term-set as standing for a pointer into a table to a set of terms. This can be elegantly used to solve an important problem in handling parse trees in context-free parsing.

Consider the following DCG for the language `a`*:

```
:- table a/3.
a(a(P1,P2)) --> a(P1),a(P2).
a(a) --> [a].
```

which recognizes a string of `a`'s and constructs its parse trees.

To generate all answers, this DCG will take time exponential in the length of the input string; not surprising since there are exponentially many parses. But say we give it an input string of n `a`'s followed by one `b`. In this case it will take exponential time to fail, since it will construct all the exponentially many partial parse trees for the initial k `a`'s. We would like the parser in this case to fail in polynomial time. We can do this by representing the parse trees as a term-set during the recognition of the string. Then after the string is recognized, we extensionalize the set-term that represents the parse trees. In this way we can get the behavior we want. The set-term that represents the parse trees for any grammar will be constructed in polynomial time; the extensionalization of that term-set will take exponential time only if there are exponentially many parses.

We can cause XSB to automatically use the term-set representation for the grammar by adding to the above program the declaration:

```
:- table a(termset,_,_).
```

which tells XSB to use the term-set representation of the first argument of nonterminal `a/3`.

With this declaration, the XSB compiler transforms the above program into the following:

```
:- table a/3.

a(a(P1,P2),S0,S) :- '_$a'(P1,S0,S1),'_$a'(P2,S1,S).
a(a,S0,S1) --> 'C'(S0,a,S1).

:- table '_$a'/3 as subsumptive.
'_$a'({X:'_$a'(X,S0,S)},S0,S) :- a(_,S0,S).
```

A new predicate `'_$a'/3` has been introduced, and all calls to the original predicate `a/3` are replaced by calls to the new one. It is defined to call the original `a/3` but to return the term-set instead of the concrete parse tree in the argument declared to be a term-set.

We can see that a call to `a/3` in this new program will have exactly as many answers as the corresponding call to `a/2` in the original recognizing DCG, since given values for `S0` and `S`, a call to `'_$a'/2` returns only one value in its first argument. So a call to `a/3` with have the polynomial complexity of the recognizer. So now when this representation is used, one gets the concrete parse tree for a string by writing, for example:

```
| ?- a(Pts,[a,a,a,a,a,a,a],[]), member_termset(Parse,Pts).
```

Here the term-set representing the parses for the sequence of `a`'s will be returned in the variable `Pts`, and then `member_termset` is used to extensionalize it to the produce the actual explicit parse tree. With this way of handling parse trees in arbitrary context-free grammars, the complexity of parsing to create the term-set is always polynomial, and then extensionalizing the term-set may be exponential if all parses are desired and there are exponentially many of them. (In fact, if the grammar contains a rule such as `A --> A`, there may be infinitely many parses.) Of course, if the parsing call to `a/3` fails, then there is no extensionalization to do, and the process is polynomial.

Note that the transformation uses subsumptive tabling for the newly introduced auxiliary predicate. This is important for this example, since the parsing calls to `'_$a'/3` will normally have `S0` bound and `S` free, yet when extensionalizing the constructed term-set to obtain the parse trees, the calls will have both `S0` and `S` bound. We do not want to recompute the parse during extensionalizion, which would happen were we to use variant tabling, and so we use subsumptive tabling.

Problems in graph traversal provide another example of the effective use of term-sets. For graph reachability, we have the very familiar:

```
:- table reach/2.
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

which is linear in the number of edges in the graph. But say that we now want to construct the path from X to Y when Y *is* reachable from X. One simple way to do it (collecting the intermediate nodes in the path in reverse order) is:

```
:- table path/3.
path(X,Y,[]) :- edge(X,Y).
path(X,Y,[Z|Path]) :- path(X,Z,Path), edge(Z,Y).
```

For an acyclic edge graph, this works fine, but for a graph with cycles, this will go into an infinite loop. Indeed, it must, since in a cyclic graph there *are* infinitely many

different paths between some nodes. However, we can use term-set to handle this situation more flexibly. We modify the above program by adding:

```
:- table path(_,_,termset).
```

With this declaration, every call to `path/3` (for a finite edge graph) will terminate in time linear in the number of edges. And all the paths will be presented in the term-set returned in the third argument. Here we have an advantage similar to the one we had in the grammar example above: if there is no path from our source to our target node, we will find that out in linear time. Without the term-set declaration, this might take exponential time, while the program builds all the paths to all the nodes that *are* reachable from our source node. Also, if we want only *one* possible path from our source to our target, we can easily retrieve only one member of the term-set during extensionalization, and the whole process is still linear.

Now consider what happens with when the graph has cycles. In this case, the term-set may be recursive and represent the infinitely many paths between nodes. For example, the term-set representing all paths from `a` to `a` in the graph with a single edge from `a` to `a` will have the same structure as the example of an infinite term-set given at the beginning of this subsection. Once the path term-set is constructed (in time linear in the number of edges for a single source), producing paths reduces to processing the term-set structure. For example to generate all paths between nodes which do not contain repeated intermediate nodes, one could write an extensionalization predicate that passes a list of term-sets in the process of being expanded, and refuse to re-expand one currently being expanded. This is the technique often used in Prolog without tabling to compute reachability in cyclic graphs.

All of these examples can be seen as special cases of constructing proof trees or justifications of goals. Indeed, term-sets could be effectively used in the construction of a justification or explanation system.

5.5 Tabling for Termination

As noted throughout this manual, tabling adds important termination properties to programs and queries. In this section we state more precisely what these termination properties are, and how the properties can be strengthened through declarations and settings for *subgoal abstraction* and for sound bounded rationality through a type of answer abstraction called *radial restraint* as well as by limiting the number of answers to a subgoal through *answer count restraint*.

Before proceeding, it is important to set the context for where issues of termination may arise. Consider first a pure normal program in which every predicate is tabled. This means a program where rules may only call other rules, possibly through negation (`tnot/1`, `not_exists/1` or `u_not/1` in XSB); but where there are no calls to built-in all-solutions predicates, or other built-ins. If such a fully-tabled pure normal program does *not* have function symbols, XSB will always terminate for any query. For instance, XSB will terminate for fully tabled pure datalog programs – even if the head of a rule is "unsafe" in that it contains variables that do not occur in the body of that rule [12]. Such programs are sometimes called datalog programs.

While datalog programs are useful for certain kinds of knowledge representation, they are not powerful enough for general programming as they do not allow recursive structures such as lists. Thus, for the rest of this section we consider pure programs that may contain function symbols. Consider a pure definite program in which every predicate is tabled. Such a program would create a table for each tabled subgoal (up to variance) exactly once if call variance were used, and at most once if call subsumption were used. In addition, tabling guarantees that each answer will be returned to each call to a tabled subgoal at most once. This means that there are two sources of non-termination. Either there can be an infinite number of subgoals, or there can be an infinite number of answers.[13]

An Infinite Number of Subgoals If a definite program produces an infinite number of subgoals *but* has a finite number of answers, the program can be made to terminate by abstracting the subgoal. For instance, consider the program fragment:

```
:- table p/1.
p(X) :- p(f(X)).
```

The goal `?- p(1)` can create an infinite number of tabled subgoals: `p(f(1))`, `p(f(f(1)))`, `p(f(f(f(1))))` and so on. Note that since all of the subgoals are ground, none subsume one another, so that call subsumption will not help here. (Although call subsumption is extremely useful in other circumstances, and would help if the goal were `?- p(X)`).

Infinite Answers Of course, subgoal abstraction can't handle cases where there are an infinite number of answers, as in the program fragment:

[12] Evaluations that call non-ground negative literals will terminate through floundering, although this can be avoided in most cases by using `not_exists/1`.

[13] Here, forest of trees model of tabling (cf. Section 10.2) is being implicitly used.

```
p(f(X)) :- p(X).
```

when given the query `?- p(X)`.

We consider each case in turn.

5.5.1 Term Size Abstraction in XSB

Both subgoal and answer abstraction in XSB are based on limiting the size of any argument of a term T that forms a subgoal or answer. The specific definition of size used is slightly complicated, but offeres advantages discussed below. Each argument T_a of T is traversed as follows. The size of T_i is initialized to 0, then T_a is traversed from left to right. Each time a non-constant functor or list symbol is encountered, the size of T_i is incremented by 1 – regardless of the type of functor symbol that is encountered. If the size of T_a exceeds the associates size limit for T (as declared in the next section), all further non-constant functor symbols encountered in T_i will be abstracted (rewritten as free variables). Once T_a has been fully traversed, further arguments of T will be traversed in the exact same manner.

Example 5.5.1 *Applying the above definition of size abstraction with limit 2 to the term*

`p(d(e(1),a,f(`c_1`)),b,g(`c_2`),[`c_3`,[`c_4`,`c_5`]))`

produces the term

`p(d(e(`X_1`),a,`X_2`),b,g(`c_2`),[`c_4`|`X_3`]).`

In the traversal, the size limit is reached once the `e/1` *functor is encountered. To the right of* `e/1`*, all non-constant functor symbols are abstracted when they occur at depth greater than 0. This causes* `f/1` *to be abstracted, as it occurs at depth 1; however* `g/1` *in the third occurs at depth 0, and so is retained. Similarly in the fourth argument, the outer list symbol and head is preserved, while the tail of the list is abstracted.*

Example 5.5.1 indicates that the size abstraction used in XSB excludes symbols of depth 0, and so is something of a hybrid approach, although we continue to call it size abstraction.

Other metrics could be used, such as term depth, which would offer conceptual clarity. However size-based abstraction allows finer-grained optimization than depth-based abstraction and offers the following general advantages.

- From the point of view of implementation, the abstraction can be perfomed with manner that has minimal if any impact on the speed of XSB's tabling engine.
- By not abstracting functor symbols at depth 0 and by abstracting each argument individually, both multi-argument indexing and star indexing of subgoals will be often be preserved.

5.5.2 Subgoal Abstraction

In a nutshell, subgoal abstraction allows a goal like `p(f(f(f(1))))` to be rewritten as

```
p(f(f(X))),X = f(1).
```

If all subgoals that have a term size – or term depth – over a given finite threshold are abstracted, any query can produce only a finite number of subgoals (since there are a finite number of predicate, function and constant symbols in any program). If a program has a finite well-founded model, it can be shown that any query to a program will terminate if that program uses subgoal abstraction [67]. For normal programs, the situation is not much different at a conceptual level. A goal such as `tnot(p(f(f(f(1)))))` would execute as `p(f(f(X)))` and then ensure that none of the answers to this goal have a binding for `X` that allows it to unify with `f(1)`. Using this intuition, it can be shown that if a program has a well-founded model with a finite number of true or undefined answers it will terminate using tabling with subgoal abstraction [66, 67].

Despite its theoretical power, subgoal abstraction can also cause problems if used indiscriminately. For instance, if the second argument of the subgoal

```
?- member(e,[a,b,c,d,e])
```

is abstracted forming the goal

```
?- member(e,[a,b,c|X])
```

leading to an infinite number of answers. a goal that terminates without abstraction will not terminate after abstraction. Note that any program containing `member/2` and at least one constant does not have a finite model (although any given ground query will have a finite number of answers). While an experienced programmer would not usually table `member/2`, he well may want to table a grammar or other program that performs recursion through a finite structure.

Declaring Subgoal Abstraction

XSB can perform subgoal abstraction based on the size limit described above. It will do so for goals called positively, but not for goals called negatively as this would give rise to unsound negation. Thus a goal G inside a construct such as `tnot/1` or `not_exists/1` will throw an exception (or suspend into break mode) if it surpasses the specified term size. In addition, subgoal abstraction is only implemented for call variance, *and applies equally to all functors, whether they are lists or non-lists.* Despite these restrictions, a tabled evaluation can be still guaranteed to terminate for queries to safe programs (cf. [66]).

Subgoal abstraction can be declared by setting a value for the maximum size of a subgoal and for the action to take when a subgoal is encountered that reaches that size.

- **size** The maximum size can be set to n for a set of predicates $\langle PredSpec \rangle$ by including the specifier `subgoal_abstract(n)` as part of the tabling declaration

  `:- table` $\langle PredSpec \rangle$ `as ...,subgoal_abstract(n),...`

  Specifying `subgoal_abstract(0)` turns abstraction off for predicates in $\langle PredSpec \rangle$. The size can also be set globally by seting the flag `max_table_subgoal_size` to the desired maximal size. If the subgoal size has been set of a given predicate via a tabling declaration the declared size will override the global size.

- **action** When a subgoal is encountered of maximum size, abstraction is enabled if the Prolog flag `max_table_subgoal_action` to `abstract`. Other possible values for the action are `error` and `suspend` (cf. pg. 253 ff.).

Unless otherwise specified, XSB starts up with `max_table_subgoal_action` set to `error` and `max_table_subgoal_size` set to 0, indicating it is turned off. Under this default behavior, XSB will throw an error if a subgoal has size greater than `max_table_subgoal_size`. As an alternative to setting flags, subgoal abstraction can be set by calling XSB with the command-line arguments `-max_subgoal_action a` and `-max_subgoal_size n` with `a` the desired action and `n` the desired size limit.

5.5.3 XSB's Approach to Bounded Rationality

Bounded rationality is a subfield of Artificial Intelligence that studies how the reasoning performed by a computation can be automatically bounded so that an agent

or other program can be guaranteed to arrive at a decision "quickly". By bounding reasoning, an agent may be used in a setting that requires reactivity or where a simulation of human reasoning is needed.

Thus, the approximation that XSB computes is *informationally sound* in the sense that no incorrect answer will be derived, although the truth value of some atoms won't be known that might have been if the size bound had been set higher.

XSB's approach to bounded rationality computes a finite approximation to the well-founded model that is *informationally sound* in the sense that no incorrect answer will be derived, although the truth value of some atoms won't be known. In other words, if bounded rationality is employed, it can be guaranteed that only a finite number of answers will be derived [34]. Furthermore, any true atom that XSB derives is true in the well founded model of a program; and any goal that fails is false in the well-founded model. However, by bounding rationality XSB's search is restrained so that it will not fully explore certain subderivations and so may consider as undefined some atoms that are true or false in the well-founded model. We sometimes call this approach to bounded rationality *restraint*. Currently XSB supports both *radial restraint* and *answer count restraint*

Radial Restraint Through Answer Abstraction

Radial restraint resembles subgoal abstraction (Section 5.5.2) in certain ways, as can be seen in the following example. If the query `p(X)` to the program

```
p(f(X)) :- p(X).
p(0).
```

were evaluated using radial restraint with a size limit of 3, the answers, `p(0)`, `p(f(0))`, `p(f(f(0)))` and `p(f(f(f(X))))` would be generated; ***however***, `p(f(f(f(X))))` would have the truth value of *undefined*. Note that by abstracting in this way, both of the goals `p(f(f(f(0))))`, and `p(f(f(f(1))))` will unify with `p(f(f(f(X))))` and so will succeed with a truth value of *undefined*. Similarly `tnot(p(f(f(f(0)))))`, and `tnot(p(f(f(f(1)))))` will both succeed with a value of *undefined* (perhaps better called *unknown* in this context). It can be seen that since all predicates and function symbols have a maximum arity (256 in XSB) bounding the size of an answer ensures that only a finite number of answers are returned [14].

[14] If a program has a infinite number of true answers and a finite number of false answers, one possible approach might be to "dualize" the program so that only false answers are computed. Note that since most programs with function symbols have an infinite number of both true and false answers, this approach won't work in general.

Semantically when radial restraint is used, XSB computes an approximation to the three-valued well-founded model of a program, called a *restrained model*. To see this, suppose the proof of a query Q does not depend on negation. If Q has a derivation that does not require any answers whose size is greater than n, it is proven as usual. Similarly, if Q is false in the well-founded model of a program, and none of the subgoals explored in the derivation of Q derive answers whose size is greater than n, XSB will derive that Q is false. The higher the size bound that is set, the better the approximation. Due to undecidability, there is no way to know in general what size to set for answer abstraction, or whether any bound needs to be set at all.

If a restrained model is derived, answers that are directly undefined through radial restraint can be desinguished from answers that are undefined in the well-founded model of a program, or for other reasons such as unsafe negation. If an answer A was abstracted due to a size check, the query `get_residual(A,Delay)` would bind `Delay` to a list containing the atom `radial_restraint`, where `radial_restraint/0` is simply a predicate defined as

```
radial_restraint:- tnot(radial_restraint)
```

Using Radial Restraint Radial restraint is currently implemented only for tabling with call variance. However it works with most other tabling features, such as call abstraction, and incremental tabling. Similarly to the use of subgal abstraction, answer abstraction is the implementational basis of radial restraint. *It is important to note that the size limit applies to the answer substitution, not to the of the answer itself.*

Example 5.5.2 *Suppose an answer size limit is set to 1, and consider the goal* `p(X)`. *The answer* `p(s(s(0)))` *has size 2 and so would be abstracted to* `p(s(`X_1`))` *as expected, as the corresponding abswer substitution is* $X = s(s(0))$. *However for the goal* `p(s(X))` *the answer substitution for the answer* `p(s(s(0)))` *is* $X = s(0)$ *which has a size of only 1 and so this answer would not be abstracted in the context of this subgoal. Despite this difference in how the size metric is computed, the termination and approximation properties of radial restraint still hold.*

Radial restraint can be declared by setting a value for the maximum size of an answer and for the action to take when an answer is encountered that reaches that size.

- **size** The maximum size can be set to `n` for a set of predicates via including the specifier `answer_abstract(n)` as part of their tabling declaration

`:- table` $<$ *PredSpec* $>$ `as ...,answer_abstract(n),...`

Specifying `answer_abstract(0)` turns answer abstraction off for predicates in $\langle PredSpec \rangle$. The size can also be set globally by seting the flag `max_table_answer_size` to the desired maximal size. If the answer size of a given predicate has been set via a tabling declaration, the predicate-specific declared size will override the global size.

- **action** When an answer is encountered of maximum size, abstraction is enabled if the Prolog flag `max_table_answer_action` to `bounded_rationality`. Other possible values for the action are `error`, `suspend` and `fail` (cf. Section 10.3.4 for further information).

Unless otherwise specified, XSB starts up with `max_table_answer_size_action` set to `error` and `max_table_answer_size` set to 0.

Answer CountRestraint

As discussed above, finite termination can always be ensured through a mixture of subgoal abstraction and radial restraint. Alternately, it can also be ensured through subgoal abstraction and answer count() restraint.

Example 5.5.3 *Consider the program*

```
:- table p/4.
p(M,N,X,Y):- between(1,M,X),between(1,N,Y).
```

and query `p(3,3,Y,Z)`*: it is easy to see that 9 answers will be produced. However, if answer count restraint is used to restrict the maximal number of answers to each subgoal to 5, the first 5 answers computed above will be returned, along with a new answer:*

```
p(3,3,Y,Z)
```

whose truth value is undefined, with the atom `answer_count_restraint` *in its delay list.*

Using the arguments from the previous section, it is easy to see that answer count restraint ensures sound finite termination when used with subgoal abstraction. However Example 5.5.3 also illustrates on a small scale how answer count restraint can be used to soundly complete a subgoal S once a minimal number of answers have been derived, even if S has a large, but finite number of answers.

Using Answer Count Restraint Answer count restraint is currently implemented only for tabling with call variance. However it works with most other tabling features, such as call abstraction, and incremental tabling.

Currently, answer count restraint can only be set by global flags as follows.

- **size** The size can be set globally via the flag `max_table_answer_size` to the desired maximal size. Setting the flag to 0 turns off answer count restraint.
- **action** When an answer is encountered of maximum size, abstraction is enabled if the Prolog flag `max_table_answer_action` to `bounded_rationality`. Other possible values for the action are `error` and `suspend` (cf. Section 10.3.4 for further information).

Justifying or Explaining Restraint

An atom affected directly by radial or answer count restraint has in its delay list either the atom `radial_restraint` or `answer_count_restraint`. The indirect dependency of an atom on a form of restraint can be obtained either thorugh the predicate `explain_u_val/3`, or `get_residual_sccs/[3,5]`. Both of these predicates traverse the residual dependency graph to provide information about why a literal is undefined.

5.6 Incremental Table Maintenance

XSB allows the user to declare that the system should maintain the correctness of a given table with respect to dynamically changing facts and rules through so-called *incremental tables* [74, 73, 82]. After a database update or series of updates Δ, an incremental table T that depends on Δ is by default updated transparently: that is T and all tables upon which T depends are automatically updated (if needed) whenever a future subgoal calls T. In either case, incremental tabling brings XSB closer to the functionality of deductive databases. If tables are thought of as materialized database views (or snapshots), then the incremental table maintenance subsystem enables incremental view maintenance; also as discussed below, if choice points are thought of as database cursors then incremental tabling also provides view consistency [15].

[15] In the current version of XSB, there are certain restrictions on how incremental tabling can be used: cf. Section 5.7.

5.6.1 Transparent Incremental Tabling

To demonstrate incremental table maintenance (informally called *incremental tabling*), consider first the following simple program that does not use incremental tabling:

```
:- table p/2.
p(X,Y) :- q(X,Y),Y =< 5.

:- dynamic q/2.
q(a,1).    q(b,3).   q(c,5).    q(d,7).
```

and the following queries and results:

```
| ?- p(X,Y),writeln([X,Y]),fail.
[c,5]
[b,3]
[a,1]

no
| ?- assert(q(d,4)).

yes
| ?- p(X,Y),writeln([X,Y]),fail.
[c,5]
[b,3]
[a,1]

no
```

In this program, the table for `p/2` depends on the contents of the dynamic predicate `q/2`. We first evaluate a query, `p(X,Y)`, which creates a table. Then we use `assert/1` to add a fact to the `q/2` predicate and re-evaluate the query. We see that the answers haven't changed, because the table is already created and the second query just retrieves answers directly from that existing table. However the answers are inconsistent with the model of `p/2` after the assert. I.e., if the table didn't exist (e.g. if `p/2` weren't tabled), the answer `[d,4]` would also be derived. Without incremental table maintenance, the only solution to this problem is for the XSB programmer to explicitly abolish a table whenever changing (with assert or retract) a predicate on which the table depends. By declaring that the tables for `p/2` should be incrementally maintained, XSB automatically keeps the tables for `p/2` correct.

Consider a slight rewrite of the above program:

```
:- table p/2 as incremental.
p(X,Y) :- q(X,Y),Y =< 5.

:- dynamic q/2 as incremental.
q(a,1).    q(b,3).    q(c,5).     q(d,7).
```

in which `p/2` is declared to be incrementally tabled and `q/2` is declared to be both dynamic and incremental, meaning that an incremental table depends on it. Consider the following goals and execution:

```
| ?- import incr_assert/1 from increval.
yes
| ?- p(X,Y),writeln([X,Y]),fail.
[c,5]
[b,3]
[a,1]

no
| ?- incr_assert(q(d,4)).

yes
| ?- p(X,Y),writeln([X,Y]),fail.
[d,4]
[c,5]
[b,3]
[a,1]

no
```

The transparent approach to incremental updating works as follows. When `incr_assert/1` is called, it sparks an invalidation phase in which tables that depend on `q(d,4)` are marked as *invalid* (i.e., possibly inconsistent with respect to underlying dynamic code). An *Incremental Dependency Graph (IDG)* is used to obtain the right tables to invalidate. However, if the invalidation phase finds an affected table that is incomplete, a permission error is thrown, since it is unclear whether sensible semantics can be given to updating a subgoal that is incomplete. After the invalidation phase is completed, when/if a subgoal calls an invalid table T the engine interrupts itself to recompute T and any tables upon which T depends. On the other hand, if no calls

are ever made to an invalid incremental table T', T' will never incur the cost of an update.

View Consistency

As described above, transparent incremental tablings's use of lazy updating ensures that a new query Q will always be consistent with the state of the dynamic code at the time Q is called. However, transparent incremental tabling enforces a stronger property of view consistency similar to those of database systems: that answers to a query Q should be those derivable at the time Q was called, *and should not be affected by any updates*. Because XSB's incremental tabling does not allow updates that affect tables that are still being computed, supporting view consistency effectively means ensuring consistency for choice points into completed incremental tables. As such choice points correspond to database cursors, we term them *Open Cursor Choice Points, (OCCPs)*.

XSB's support for view consistency is designed so that no perceptable overhead in incurred if there are no OCCPs whose view needs to be maintained. Not surprisingly, numerous long-lived OCCPs whose views need to be maintained across updates causes an overhead for the engine, a situation that is in some sense similar to the cost of maintaining views for cursors in database system.

5.6.2 Updating in a Three-Valued Logic

As discussed earlier in this chapter, answers that are undefined in the well-founded semantics are represented as conditional answers. Beginning with version 3.3.7, incremental updates work correctly with conditional answers [16]. Nno special care needs to be taken for updating in the well-founded semantics as the following example illustrates.

```
:- dynamic data/1 as incremental.

:- table opaque_undef/0 as opaque.
opaque_undef:- tnot(opaque_undef).

:- table p/1 as incremental.
p(_X):- opaque_undef.
```

[16] Before Version 3.3.7, incremental updates only worked correctly on stratified tables: those with only unconditional answers.

```
p(X):- data(X).
```

Note that `opaque_undef/1` upon which `p/1` depends is explicitly declared as opaque [17]. When the above program is loaded, XSB will behave as follows.

```
| ?- p1(1).

undefined
| ?- incr_assert(data(1)).

yes
| ?- p1(1).

yes
| ?- incr_retract(data(1)).

yes
| ?- p1(1).

undefined
| ?- get_residual(p1(1),C).

C = [opaque_undef]
```

Declaring Predicates to be Incremental

In XSB, tables can have numerous properties: such as *subsumptive, variant, incremental, opaque, dynamic, private*, and *shared*, and can use answer subsumption, answer abstraction or call abstraction. XSB also has variations in forms of dynamic predicates: *tabled, incremental, private*, and *shared*. XSB extends the `table` and `dynamic` compiler and executable directives with modifiers that allow users to indicate the kind of tabled or dynamic predicate they want. For example,

```
:- table p/3,s/1 as subsumptive,private.

:- table q/3 as incremental,variant.
```

[17] An *opaque* predicate P is tabled and is used in the definition of some incrementally tabled predicate but should not be maintained incrementally. In this case the system assumes that the programmer will abolish tables for P in such a way so that re-calling it will always give semantically correct answers.

```
:- dynamic r/2,t/1 as incremental.
```

In the current version of XSB, incremental tabling works with subgoal abstraction, answer abstraction, and well-founded negation. However several combinations involving incremental tabling are not supported and will throw an error (cf. page 300 and page 289, respectively). Incremental tabling has not yet been ported to the multi-threaded engine and it currently does *not* works for predicates that use call subsumption or answer subsumption.

5.6.3 Incremental Tabling using Interned Tries

Sometimes it is more convenient or efficient to maintain facts in interned tries rather than as dynamically asserted facts (cf. Chapter 8). Tables based on interned tries can be automatically updated when terms are interned or uninterned just as they can be automatically updated when a fact is asserted or retracted. Consider the example from Section 5.6.1 rewritten to use interned tries. As usual, an incrementally updated table is declared as such:

```
:- table p/2 as incremental.
```

However, the declaration for dynamic data changes: rather than using the declaration

```
:- dynamic q/2 as incremental
```

a trie is specified as incremental in its creation.

```
trie_create(Trie_handle,[incremental,alias(inctrie)])
```

As described in Chapter 8, the trie handle returned is an integer, but can be aliased just as with any other trie. The trie may then be initially loaded:

```
trie_intern(q(a,1),inctrie),trie_intern(q(b,3),inctrie),
trie_intern(q(c,5),inctrie),trie_intern(q(d,7),inctrie).
```

At this stage a query to `p/2` acts as before:

```
p(X,Y) :- trie_interned(q(X,Y),inctrie),Y =< 5.
```

```
| ?- p(X,Y),writeln([X,Y]),fail.
[c,5]
[b,3]
[a,1]
```

The following sequence ensures that `p/2` is incrementally updated as `inctrie` changes:

```
| ?- import incr_trie_intern/2.

yes
| ?- incr_trie_intern(inctrie,q(d,4)).

yes
| ?- p(X,Y),writeln([X,Y]),fail.
[d,4]
[c,5]
[b,3]
[a,1]

no
```

Given the proper directives to make a trie incremental, transparent incremental tabling works for changes made to interned tries just as it does for regular dynamic code and for trie-indexed dynamic code.

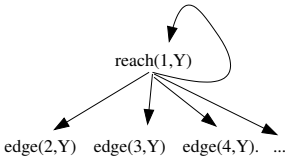
5.6.4 Abstracting the IDG for Better Performance

As mentioned above, incremental table mantenance makes use of an IDG. Specifically, the nodes of the IDG are the incrementally tabled subgoals; and each such table contains information about its incident edges: those subgoals upon which a node directly depends or directly affects. While the IDG is a critical data structure to efficiently update incremental tables, in certain situations constructing the IDG can cause non-trivial overheads in query time and table space. These overheads can be addressed in many cases by *abstracting* the IDG. When a tabled subgoal S is called, rather than creating an edge between S and its nearest tabled ancestor S' (if any), one could abstract S, S' or both, potentially collapsing a large number of nodes and edges of the IDG. If S is an incremental table, then performing subgoal abstraction on S as introduced in Section 5.5, will abstract the IDG – rather than having n nodes $S_1, \ldots, S_n$ and their associated links, the IDG will contain a single node $abstract(S)$.

However, subgoal abstraction will not work to abstract the leaf nodes of the IDG, which are subgoals to non-tabled dynamic incremental predicates.

In Version 3.8 of XSB, IDG nodes for dynamic incremental predicates may undergo depth abstraction: given a subgoal S and integer k, subterms of S with depth $k+1$ are replaced by unique new variables. For instance, abstracting *q(f(1))* at level 1 gives *q(f(X_1))*; abstracting at level 0 gives *q(X_1)*. Figure 5.5 illustrates an important case where abstracting dynamic incremental predicates can be critical to good performance for incremental tabling. In the case of left-linear recursion, if no abstraction is used a new node will be created for each call to `edge/2` as shown on the left side of this figure. If a large number of data elements are in fact reachable, the size of the IDG can be very large. If calls to the `edge/2` predicate make use of depth-0 abstraction, the graph may be much smaller as seen on the right side of Fig. 5.5. Whether abstracting a IDG in this manner is useful or not is application dependent; however, performance results indicate that for left-linear recursion, abstraction greatly reduces both query time and space.

```
:- table reach/2 as incremental.
:- dynamic edge/2 as incremental.
reach(X,Y):- edge(X,Y).
reach(X,Y):- reach(X,Z),edge(Z,Y).
```



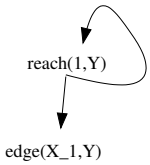


Figure 5.5: A left-linear program and schematic IDGs: Left without IDG abstraction; Right: with IDG abstraction

Abstracting the `edge/2` predicate has subtle differences from abstracting tabled subgoals. As mentioned, the `edge/2` predicate of Fig. 5.5 is not tabled. Furthermore, the actual `edge/2` subgoal itself should not be abstracted to depth 0 since losing the first argument instantiation would prevent the use of indexing. Rather, only the IDG's representation of the subgoal should be abstracted. Abstraction of dynamic code for the IDG can be specified via the declaration:

```
:-dynamic edge/2 as incremental, abstract(0).
```

In Version 3.8 dynamic incremental code can be abstracted, but incremental interned tries (Section 5.6.3) cannot be. Also, currently only depth 0 abstraction is

supported.

5.6.5 Summary and Implementation Status

The main design choices of incremental tabling are as usual what to table, and also what dynamic predicates or tries should be made incremental. In addition, performance optimizations may be made through a mixture of subgoal abstraction and dynamic predicate abstraction. This optimization can be informed by use of `statistics/0` which includes summary information about the IDG, or using the IDG inspection predicates of Section 5.6.6 if more details are needed.

In the current version of XSB, incremental tabling has not yet been ported to the multi-threaded engine. In addition, incremental tabling only works for predicates that use both call and answer variance. However, incremental tabling does work with for the full well-founded semantics, for trie indexed dynamic code (in addition to regular dynamic code) and with interned tries as described in Section 5.6.3. The space reclamation predicates `abolish_all_tables/0`, `abolish_table_call/[1,2]` and `abolish_table_pred/[1,2]` can be safely used with incremental tables.

5.6.6 Predicates for Incremental Table Maintenance

A Note on Terminology Suppose `p/1` and `q/1` are incrementally tabled, and that there is a clause

```
p(X):- q(X).
```

In this case we say that `p(X)` *depends_on* `q(X)` and that `q(X)` *affects* `p(X)`. A recursive predicate both depends on and affects itself.

Declarations The following directives support incremental tabling based on changes in dynamic code:

`table +PredSpecs as incremental` Tabling
: is a executable predicate that indicates that each tabled predicate specified in `PredSpec` is to have its tables maintained incrementally. `PredSpec` is a list of skeletons, i.e. open terms, or `Pred/Arity` specifications [18]. The tables must use call variance and answer variance and must be compiled and loaded into

[18] No explicit module references are allowed.

the single-threaded engine. If a predicate is declared with tabling attributes that are not supported with incremental tabling a permission error is thrown. This predicate implies that its arguments are tabled predicates. See page 300 for further discussion of tabling options.

We also note that any tabled predicate that is called by a predicate tabled as incremental must also be tabled as incremental or as opaque. On the other hand, a dynamic predicate `d/n` that is called by a predicate tabled as incremental may or may not need to be declared as incremental. However if `d/n` is not declared incremental, then changes to it will not be propagated to incrementally maintained tables.

`dynamic +PredSpecs as incremental` Tabling

is an executable predicate that indicates that each predicate in `PredSpecs` is dynamic and used to define an incrementally tabled predicate and will be updated using `incr_assert/1` and/or `incr_retractall/1` (or relatives.) Note that dynamic incremental predicates cannot themselves be tabled. This predicate implies that its arguments are dynamic predicates. See page 289 for further discussion of dynamic options.

`table +PredSpecs as opaque` Tabling

is an executable predicate that indicates that each predicate P in `PredSpecs` is tabled and is used in the definition of some incrementally tabled predicate but should not be maintained incrementally. In this case the system assumes that the programmer will abolish tables for P in such a way so that re-calling it will always give semantically correct answers. In other words, instead of maintaining information to support incremental table maintenance, the system re-calls the opaque predicate whenever its results are required to recompute an answer. One example of an appropriate use of opaque is for tabled predicates in a DCG used to parse some string. Rather than incrementally maintain all dependencies on all input strings, the user can declare these intermediate tables as opaque and abolish them before any call to the DCG. This predicate implies that its arguments are tabled predicates.

Basic Incremental Maintenance Predicates The following predicates are used to manipulate incrementally maintained tables:

`incr_assert(+Clause)` module: `increval`
`incr_assertz(+Clause)` module: `increval`
`incr_asserta(+Clause)` module: `increval`
`incr_retract(+Clause)` module: `increval`

`incr_retractall(+Term)` module: `increval`

are versions of `assert/1` and other standard Prolog predicates. They modify dymamic code just as their Prolog counterparts, but they first invalidate all incrementally maintained tables that depend on `Clause`.

Error Cases are the same as `assert<a/z>/1`, `retract/1` and `retractall/1` with the additional error conditions that relate to the semantics of incremental tabling. Note that if these error conditions arise, the update will `not` occur.

- The head of the clause `Clause` or the `Term` refers to a predicate that is not incremental and dynamic.
  - `type error(dynamic_incremental, Term)`
- `Clause` affects an incremental table that is incomplete (and so is in the course of being computed).
  - `permission_error`

`incr_invalidate_calls(+Goal)` module: `increval`

Let $\mathcal{T}$ be the least set of all incrementally maintained tables whose goals that unify with `Goal`, or whose tables are (transitively) affected by a goal in $\mathcal{T}$. This predicate invalidats all tables in $\mathcal{T}$. Any subsequent call to a goal G associated with $\mathcal{T}$ will be automatically be incrementally updated if necessary. (As will any goals that G depends on that are in need of updating.) In a similar manner, an invocation of `incr_table_update/[0,1,2]` will cause tables in $\mathcal{T}$ to be updated.

Note that this predicate is needed for exceptional cases only. Calls to `incr_assert/1` and similar predicates mentioned above perform invalidation automatically, as does `abolish_table_call/[1,2]`. However, `incr_invaldate_calls/1` is useful if a tabled predicate depends on some external data and not (only) on dynamic incremental predicates. For example, such a predicate might depend on a relation stored in an external relational database (perhaps accessed through the ODBC interface). Of course, in such a case, the application programmer must know when the external relation changes and invoke `incr_invaludate_calls/1` as necessary.

Error Cases

- `Goal` is tabled, but not incrementally tabled
  - `permission_error(invalidate,non-incremental predicate,Goal)`

Incremental Maintenance using Interned Tries The following predicates are used to modify incremental tries, and can be freely intermixed with predicates for modifying incremental dynamic code, as well as with predicates for invalidating or updating tables (Section 5.6.6).

`incr_trie_intern(+TrieIdOrAlias,+Term)` module: `intern`
: is a version of `trie_intern/2` for tries declared as incremental. A call to this predicate interns `Term` in `TrieIdOrAlias` and then invalidates all incrementally maintained tables that depend on this trie.

`incr_trie_uninternall(+TrieIdOrAlias,+Term)` module: `intern`
: is a version of `trie_unintern/2` for tries declared as incremental. A call to this predicate removes all terms unifying with `Term` in `TrieIdOrAlias` and then invalidates all incrementally maintained tables that depend on this trie.

Inspecting the State of the Incremental Dependency Graph The predicates in this section allow a user to inspect properties of IDG that can be useful in debugging, profiling or optimizing a computation [19]. In addition they provide information about which subgoals in the IDG are invalid – i.e., which subgoals depend on a dynamic code that has changed, but have not been updated.

As explained below, IDG nodes can be accessed via the predicate `is_incremental_subgoal/1`, while IDG edges can be accessed via `incr_directly_depends/2`. The predicates `get_incr_scc/[1,2]` and `get_incr_scc_with_deps/[3,4]` can be used to efficiently materialize the dependency graph in Prolog, including SCC information. Similarly, the predicates `incr_invalid_subgoals/1` and `incr_is_invalid/1` can be used to determine which subgoals are invalid.

`is_incremental_subgoal(?Subgoal)` module: `increval`
: This predicate non-deterministically unifies `Subgoal` with incrementally tabled subgoals that are currently table entries.

`incr_directly_depends(?Goal`$_1$`,?Goal`$_2$`)` module: `increval`
: accesses the edges of the IDG: the incremental goals (Tables) that directly depend on or directly affect one another. At least one of `Goal`$_1$ or `Goal`$_2$ must be bound.

[19] The predicates for traversing the incremental dependency graph are somewhat analogous to those for traversing the residual dependency graph (Section 6.15.2).

- If $\texttt{Goal}_1$ is bound, then this predicate will return in $\texttt{Goal}_2$ through backtracking the goals for all incrementally maintained tables on which $\texttt{Goal}_1$ directly depends.
- If $\texttt{Goal}_2$ is bound, then it returns in $\texttt{Goal}_1$ through backtracking the goals for all incrementally maintained tables that $\texttt{Goal}_2$ directly affects – in other words all goals that directly depend on $\texttt{Goal}_2$.

Error Cases

- Neither $\texttt{Goal}_1$ nor $\texttt{Goal}_2$ is bound
  - `instatiation_error`
- $\texttt{Goal}_1$ and/or $\texttt{Goal}_2$ is bound, but is not incrementally tabled
  - `table_error`

`incr_trans_depends(?Goal`$_1$`,?Goal`$_2$`)` module: `increval`

is similar to `incr_directly_depends/2` except that it returns goals according to the transitive closure of the "directly depends" relation. Error conditions are the same as `incr_directly_depends/2`.

`get_incr_sccs(?SCCList)` module: `increval`
`get_incr_sccs_with_deps(?SCCList,?DepList)` module: `increval`
`get_incr_sccs(+SubgoalList,?SCCList)` module: `increval`
`get_incr_sccs_with_deps(+SubgoalList,?SCCList,?DepList)` module: `increval`

Warning: these predicates may be obsolescent, cf. Section 10.3.2 for newer predicates that are more powerful.

Most linear algorithms for SCC detection over a graph use destructive assignment on a stack to maintain information about the connecteness of a component; as a result such algorithms are difficult to write efficiently in Prolog.

`get_incr_sccs/1` unifies `SCCList` with SCC information for the incremental dependency graph that is represented as a list whose elements are of the form

```
ret(Subgoal,SCC).
```

`SCC` is a numerical index for the SCCs of Subgoal. Two subgoals are in the same SCC iff they have the same index, however no other dependency information can be otherwise directly inferred from the index [20].

[20] The actual number for each SCC index depends on how the incremental dependency graph happens to be traversed; as a result it is best to rely on the index only as a "generated" name for each SCC.

If dependency information is also desired, `get_incr_scc_with_dependencies/2` should be called. In addition to the SCC information as above, `DepList` is unified with a list of dependency terms of the form

```
depends(SCC1,SCC2)
```

for each pair `SCC1` and `SCC1` such that some subgoal with index `SCC1` directly depends on some subgoal with index `SCC1`. If it is necessary to know which subgoal(s) in `SCC1` directly depends on which subgoal(s) in `SCC2`, the information can be easily reconstructed using `incr_directly_depends/2` above. Similarly, `incr_directly_depends/2` can be used to determine the actual edges within a given SCC.

Ordinarily a user will want to see the entire dependency graph and in such a case the predicates described above should be used. However, note that if the dependency graph is the result of several indepdendent queries it may not be connected. `get_incr_scc/2` takes as input a list of incremental subgoals, `SubgoalList`. For each `Subgoal` in `SubgoalList`, this predicate finds the set of subgoals connected to `Subgoal` by any mixture of depends and affects relations, unions these sets together, and finds the SCCs of all subgoals in the unioned set.

SCC detection is implemented using Tarjan's algorithm [87] in C working directly on XSB's data structures. The algorithm is $\mathcal{O}(|V|+|E|)$ where $|V|$ is the number of vertices and $|E|$ the number of edges in the dependency graph. As a result, `get_incr_sccs/[1,2]` provides an efficient means to materialize the high-level topography of the dependency graph [21].

Error Cases

- `SubgoalList` is a variable
  - `instantiation_error`
- `SubgoalList` is not a list
  - `type_error`
- `SubgoalList` contains a predicate that is not tabled
  - `permission_error`

`incr_invalid_subgoals(-List)` module: `increval`

This predicate unifies `List` with a sorted list of the incremental subgoals that are currently invalid.

[21] Currently, the materialization of dependency information between SCCs is implemented in a naive manner, so that `get_incr_sccs_with_deps/[2,3]` is $\mathcal{O}(|V|^2)$.

`incr_is_invalid(+Subgoal)` module: `increval`

Succeeds if `Subgoal` is an incrementally tabled subgoal that is invalid, and fails otherwise.

5.7 Compatibility of Tabling Modes and Predicate Attributes

As discussed in this chapter, there are several choices for how to table a predicate. Either call subsumption or call variance may be used, incremental tabling might or might not be used, and answer subsumption might or might not be used. Furthermore, a tabled predicate, like any other predicate, may be static or dynamic and thread shared or thread private. Together, there are 48 different combinations, not all of which are supported in Version 3.8 of XSB. To analyze further, all combinations are supported for call-variance and for thread private predicates. However, call subsumption has not been fully integrated with dynamic code or thread shared predicates, and cannot currently be combined with incremental tabling or with answer subumption. Similarly incremental tabling is not yet supported in the multi-threaded engine (it is supported for "thread private" computations only in the sequential engine). The compatibilities are listed in Table 5.1. Further combinations will be supported in future versions of XSB as resources allow.

The combinations in Table 5.1 allow full well-founded computation, constrained variables in calls and answers (including the residual program), and safe space reclamation, with the following exceptions. Answer subsumption does support non lrd-stratified programs; and call subsumption does not yet support attributed variables in calls.

5.8 A Weaker Semantics for Tabling

Recall that the well-founded semantics (WFS) is weaker than, say, stable model semantics. For instance a program like

```
p:- not q.          q:- not p.
```

has two stable models: $\{p\}$ and $\{q\}$. On the other hand, WFS has a single model, where both p and q are undefined. This, of course, is characteristic of the way WFS treats atoms whose only non-failed derivations are based on a "negative loop".

variant	static	private	nonincremental	no answer subsumption	yes
variant	static	private	nonincremental	answer subsumption	yes
variant	static	private	opaque	no answer subsumption	yes
variant	static	private	opaque	answer subsumption	no
variant	static	private	incremental	no answer subsumption	yes
variant	static	private	incremental	answer subsumption	no
variant	static	shared	nonincremental	no answer subsumption	yes
variant	static	shared	nonincremental	answer subsumption	yes
variant	static	shared	opaque	no answer subsumption	no
variant	static	shared	opaque	answer subsumption	no
variant	static	shared	incremental	no answer subsumption	no
variant	static	shared	incremental	answer subsumption	no
variant	dynamic	private	nonincremental	no answer subsumption	yes
variant	dynamic	private	nonincremental	answer subsumption	yes
variant	dynamic	private	opaque	no answer subsumption	no
variant	dynamic	private	opaque	answer subsumption	no
variant	dynamic	private	incremental	no answer subsumption	no
variant	dynamic	private	incremental	answer subsumption	no
variant	dynamic	shared	nonincremental	no answer subsumption	yes
variant	dynamic	shared	nonincremental	answer subsumption	yes
variant	dynamic	shared	opaque	no answer subsumption	no
variant	dynamic	shared	opaque	answer subsumption	no
variant	dynamic	shared	incremental	no answer subsumption	no
variant	dynamic	shared	incremental	answer subsumption	no
subsumptive	static	private	nonincremental	no answer subsumption	yes
subsumptive	static	private	nonincremental	answer subsumption	yes
subsumptive	static	private	opaque	no answer subsumption	no
subsumptive	static	private	opaque	answer subsumption	no
subsumptive	static	private	incremental	no answer subsumption	no
subsumptive	static	private	incremental	answer subsumption	no
subsumptive	static	shared	nonincremental	no answer subsumption	no
subsumptive	static	shared	nonincremental	answer subsumption	no
subsumptive	static	shared	opaque	no answer subsumption	no
subsumptive	static	shared	opaque	answer subsumption	no
subsumptive	static	shared	incremental	no answer subsumption	no
subsumptive	static	shared	incremental	answer subsumption	no
subsumptive	dynamic	private	nonincremental	no answer subsumption	yes
subsumptive	dynamic	private	nonincremental	answer subsumption	yes
subsumptive	dynamic	private	opaque	no answer subsumption	no
subsumptive	dynamic	private	opaque	answer subsumption	no
subsumptive	dynamic	private	incremental	no answer subsumption	no
subsumptive	dynamic	private	incremental	answer subsumption	no
subsumptive	dynamic	shared	nonincremental	no answer subsumption	no
subsumptive	dynamic	shared	nonincremental	answer subsumption	no
subsumptive	dynamic	shared	opaque	no answer subsumption	no
subsumptive	dynamic	shared	opaque	answer subsumption	no
subsumptive	dynamic	shared	incremental	no answer subsumption	no
subsumptive	dynamic	shared	incremental	answer subsumption	no

Table 5.1: Support for different tabling modes in XSB Version 3.8

However, an even weaker logic is possible where derivations based on positive loops are also considered undefined. In other words the program

```
r:- r.                    r:- false.
```

would assign the truth value *undefined* to r, although both WFS and stable models would assign r as *false*. But why use such a weak logic?

Consider a woman who asks her husband when he'll clean the garage, and the husband says:

I'll get around to it when I get around to it.

The wife would probably consider it ambiguous not only when her husband might clean the garage, but whether he would do so at all. The wife's reasoning (slightly simplified) could be rendered in logic as:

```
clean_the_garage:- clean the garage.
```

and we'd like to assign *undefined* or *unknown* to `clean_the_garage`.[22]

Although this example is somewhat fanciful, it turns out that this interpretation accords with the results of cognitive science experiments about human reasoning [76], and is known in the logic programming community as the "completion semantics" (CS). CS differs from WFS only in assigning the truth value *undefined* to derivations that depend on a positive loop, and that are otherwise not satisfiable (cf. [53]).

So as useful as WFS and stable models are for programming, they don't reflect how human beings have been shown to reason in daily life. However, there is another difference between WFS and the sort of common-sense reasoning that humans perform. WFS has a strong *closed-world* assumption. Suppose a query `?- s` were made to a program where `s` were not defined. WFS would assign the value *false* to `s`, but this is not always what humans do: rather humans would treat the unknown predicate `s` as in fact *unknown* or *undefined*. More generally, if (sub-)goal G refers to an undefined predcate,the *weak* completion semantics (WCS) also assigns *undefined* to G, rather than *false* as WFS does, or throwing an error (as XSB also does by default)z.

These features can be set globally either separately or together:

[22] Actually, many wives would go ahead and assign *false* to this statement, but we are modeling an optimistic wife.

- Setting the ISO Prolog flag `unknown` to `undefined` makes calls to unknown predicates return the truth value *undefined*. This flag can also be set to the standard ISO values, `fail`. `warning` or `error` (the last of which is the default).
- Setting the Prolog flag `alt_semantics` to `cs` causes XSB to globally evaluate the completion semantics.
- Setting the Prolog flag `alt_semantics` to `wcs` causes XSB to globally evaluate the weak completion semantics, and is equivalent to setting the `alt_semantics` flag to `cs` and the ISO flag `unknown` to `undefined`.
- Setting the Prolog flag `alt_semantics` to `wfs` turns causes XSB to behave in its default mode. I.e., to globally evaluate queries according to the well-founded semantics, and to throw an error when encountering an unknown predicate.

Examples As a simple example, consider the program:

```
simple_loop(X):- simple_loop(X).      simple_loop(X):- p(X).
p(a).
```

The query `?- simple_loop(X)` returns two answers: `X = a` as *true*, and `X` unbound as *undefined*.

For a more complex example, consider the program:

```
:- table m_1_1/1,m_1_2/1,m_1_3/1,m_1_4/1.
m_1_1(X):- m_1_2(X).          m_1_1(a):- m_1_2(a).
m_1_2(X):- m_1_3(X).          m_1_2(a):- m_1_3(a).
m_1_3(X):- m_1_4(X),fail.     m_1_3(a):- m_1_4(a).
m_1_4(X):- m_1_1(X).          m_1_4(a):- m_1_1(a).
```

The derivation of the query `?- m_1_1(X)` creates a positive SCC with with numerous interrelated positive cycles, but these cycles can be broken down into two groups. The first group includes a dependency edge from `m_1_3(X)` to `m_1_4(X)`, while the other set does not include this edge. Due to the first clause of `m_1_3(X)`, all derivations in the first group fails, although derivations that do not include this edge succeed. Thus the only answer to `m_1_1(X)` has `X = a` with truth value *undefined* (and a single delay list of `[m_1_2(a)]`.

Chapter 6

Standard Predicates and Predicates of General Use

This chapter mainly describes *standard* predicates, which are always available to the Prolog interpreter, and do not need to be imported or loaded explicitly as do other Prolog predicates. By default, it is a compiler error to redefine standard predicates.

In the description below, certain standard predicates depend on HiLog semantics; the description of such predicates have the token HiLog at the right of the page. Similarly predicates that depend on SLG evaluation are marked as Tabling, and predicates whose semantics is defined by the ISO standard (or whose implementation is reasonably close to that definition) are marked as `ISO`. Occasionally, however, we include in this section predicates that are not standard. In such cases we denote their module in `text` font towards the middle of the page.

6.1 Input and Output

6.1.1 I/O Streams in XSB

XSB's I/O is based on ISO-style streams, although it also supports older DEC-10 style file handling. The use of streams provides a unified interface to a number of different classes of sources and sinks. Currently these classes include textual and binary files, console input and output, pipes, and atoms; in the future sockets and urls may be handled under the stream interface. When streams are opened, certain actions may occur depending on the class of the source or sink and on the wishes of the user. For instance when a file `F` is opened for output mode, an existing file

`F` may be truncated (in write mode) or not (in append mode). In addition, various operations may or may not be valid depending on the class of stream. For instance, repositioning is valid for an atom or file but not a pipe or console.

XSB provides several default I/O streams, which make it easier for a user to embed XSB in other applications. These streams include the default input and output streams. They also include the standard error stream, to which XSB writes all error messages. By default the standard error stream is the same as the standard output stream, but it can be redirected either by UNIX shell-style I/O redirection or by the predicates `file_reopen/4` and `file_clone/3`. Similarly there is the standard warning stream (to which all system warnings are written), the standard message stream, the standard debugging stream (to which debugging information is written), and the standard feedback stream (for interpreter prompts, yes/no answers, etc). All of these streams are aliased by default to standard output, and can be redirected by the predicates `file_reopen/4` and `file_clone/3`. Such redirection can be useful for logging, or other purposes.

Streams may also be aliased: the default input and output streams are denoted by `user_input` and `user_output` and they refer to the standard input and standard output streams of the process [1]. Similarly, XSB's error, warning and message streams uses the aliases `user_error`, `user_warning` and `user_message` respectively.

Streams are distinguished by their `class` – whether they are file or atom, etc.; as well as by various properties. These properties include whether a stream is positionable or not and whether a (file) stream is textual or binary.

- `Console`: The default streams mentioned above are console streams, which are textual and not repositionable.
- `File`: A file stream corresponds to an operating system file and is repositionable. On Windows, binary files and textual files differ, while on UNIX they are the same.
- `Atom`: XSB can read from an atom, just as it can from a file. Atoms are considered to be textual and repositionable. Writing to atoms via streams is not currently available in XSB, although the predicate `term_to_atom/[2,3]` contains much of the functionality that such streams would provide.
- `Pipe`: XSB can also open pipes either directly, or as part of its ability to spawn processes. When made into streams, pipes are textual and not repositionable.

[1] For backwards compatibility, the default input stream can also be aliased by `user` or `userin`, and the default output stream by `user` or `userout`.

I/O Stream Implementation

A user may notice that XSB's I/O streams are small integers, but they should not be confused with the file descriptors used by the OS. The OS file descriptors are objects returned by the C `open` function; XSB I/O streams indices into the internal XSB table of open files and associated information. The OS does not know about XSB I/O streams, while XSB (obviously) does know about the OS file descriptors. An OS file descriptor may be returned by certain predicates (e.g. `pipe_open/2` or user-defined I/O). In the former case, a file descriptor can be promoted to XSB stream by `open/{3,4}` and in the latter by using the predicate `fd2iostream/2`.

When it starts, XSB opens a number of standard I/O streams that it uses to print results, errors, debugging info, etc. The descriptors are described in the file `prolog_includes/standard.h`. This file provides the following symbolic definitions:

```
#define STDIN           0
#define STDOUT          1
#define STDERR          2
#define STDWARN         3    /* output stream for xsb warnings  */
#define STDMSG          4    /* output for regular xsb messages */
#define STDDBG          5    /* output for debugging info       */
#define STDFDBK         6    /* output for XSB feedback
                                (prompt/yes/no/Aborting/answers) */

#define AF_INET     0     /* XSB-side socket request for Internet domain */
#define AF_UNIX     1     /* XSB-side socket request for UNIX domain */
```

These definitions can be used in user programs, if the following is provided at the top of the source file:

```
compiler_options([xpp_on]).
#include "standard.h"
```

If this header is used, the various streams can be used as any other output stream – e.g. `?- write(STDWARN,'watch it!')`. (Note: the XSB preprocessor is not invoked on clauses typed into an interactive XSB session, so the above applies only to programs loaded from a file using `consult` and such.)

6.1.2 Character Sets in XSB

Beginning in Version 3.5 of XSB, alternate character sets are supported.

- *UTF-8* which on input atomatically interprets the sequence of bytes as UTF-8 byte sequences and decodes them to obtain the unicode code points; and on output converts from the unicode code points to UTF-8 byte sequences.
- *LATIN-1* which performs no transformation on byte sequences (i.e. treats each byte directly as a unicode code point.)
- *CP1252* which implements Windows code page 1252 encoding, the default for most Windows systems.

Other character sets, in particular, UTF-16, may be supported in the future.

In the current version of XSB, UTF-8 is the default character set when XSB is configured on UNIX-style systems such as Linux and Mac OSX. CP1252 is the the default character set on Windows-style systems. The character set may be changed at any time via the Prolog flag `character_set`, whose value must be one of `utf_8`, `cp1252`, or `latin_1`. The character set in effect at the time of opening a stream is the character set that will be used to read (or write) the stream.

6.1.3 Predicates for ISO Streams

`open(+SourceSink,+Mode,-Stream)` ISO

`open/1` creates a stream for the source or sink designated in `SourceSink`, and binds `Stream` to a structure representing that stream.

- If `SourceSink` is an atom, or the term `file(File)` where `File` is an atom, the stream is a file stream. In this case `Mode` can be
  - `read` to create an input stream. In Windows, whether the file is textual or binary is determined by the file's properties.
  - `write` to create an output stream. Any previous file with a similar path is removed and a (textual) file is created which becomes a record of the output stream.
  - `write_binary` to create an output stream. Any previous file with a similar path is removed and a file is created which becomes a record of the output stream. The file created is binary in Windows, while in UNIX `write_binary` has the same effect as `write`.

  - `append` to create an output stream. In this case the output stream is appended to the contents of the file, if it exists, and otherwise a new file is created for (textual) output
  - `append_binary` to create an output stream. In this case the output stream is appended to the contents of the file, if it exists, and otherwise a new file is created for (binary) output

- If `SourceSink` is the term `atom(Atom)` where `Atom` is an atom, the stream is an atom stream. In this case `Mode` currently can only be `read`. This stream class, which reads from interned atoms, is analogous to C's `sscanf()` function.
- If `SourceSink` is the term `pipe(FIleDescriptor)` where `FileDescriptor` is an integer, then a pipe stream is opened in the mode for `FileDescriptor`.

ISO Compatibility Note: This predicate extends the ISO definition of `open/3` to include strings and pipes as well as the file modes `write_binary` and `append_binary`.

Error Cases

- `SourceSink` or `Mode` is not instantiated
  - `instantiation_error`
- `Mode` is not a valid I/O mode
  - `domain_error(io_mode,Mode)`
- `SourceSink` is a file and cannot be opened, or opened in the desired mode
  - `permission_error(open,file,SourceSink)`

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

`open/4` behaves as does `open/3`, but allows a list of options to be given. The current options are a subset of ISO options and are:

- `alias(A)` allows the stream to be aliased to an atom `A`.
- `type(T)` has no effect on file streams in UNIX, which are always textual, but in Windows if `T` is `binary` a binary file is opened.

Error Cases Error cases are the same as `open/3` but with the addition:

- `Option_list` contains an option `O` that is not a (currently implemented) stream option.
  - `domain_error(stream_option,O)`

- An element of `OptionsList` is alias(A) and A is already associated with an existing thread, queue, mutex or stream
  - `permission_error(create,alias, A)`
- An element of `OptionsList` is alias(A) and A is not an atom
  - `type_error(atom,A)`

ISO Compatibility Note: The ISO option `reposition(Boolean)` currently has no effect on streams, because whether or not the stream is repositionable or not depends on the stream class. The ISO option `eof_action(Action)` currently has no effect on file streams. If these options are encountered in `Options`, a warning is issued to `STDWARN`.

`close(+Stream_or_alias,+OptionsList)` ISO

`close/2` closes the stream or alias `Stream_or_alias`. `OptionsList` allows the user to declare whether a permission error will be raised in XSB upon a resource or system error from the closing function (e.g. `fclose()` or other system function). If `OptionsList` is non-empty and contains only terms unifying with `force(true)` then such an error will be ignored (possibly leading to unacknowledged loss of data). Otherwise, a permission error is thrown if `fclose()` or other system function returns an error condition. If the stream class of `Stream_or_alias` is an atom, then the only action taken is to close the stream itself – the interned atom itself is not affected.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable, nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open stream
  - `existence_error(stream,Stream_or_alias)`
- `OptionList` contains an option `O` that is not a closing option.
  - `domain_error(close_option,O)`
- `OptionList` contains conflicting options
  - `domain_error(close_option,OptionList)`
- Closing the stream produces an error (and `OptionsList` is a non-empty list containing terms of the form `force(true)`).

- `permission_error(close,file,Stream_or_alias)`

`close(+Stream_or_alias)` ISO

`close/1` closes the stream or alias `Stream_or_alias`.
Behaves as `close(Stream_or_alias,[force(false)])`.

`set_input(+Stream_or_alias)` ISO

Makes file `Stream_or_alias` the current input stream.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable, nor a a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not an open input stream
  - `existence_error(stream,Stream_or_alias)`

`set_output(+Stream_or_alias)` ISO

Makes file `Stream_or_alias` the current output stream.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable, nor a a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open output stream
  - `existence_error(stream,Stream_or_alias)`

`stream_property(?Stream,?Property)` ISO

This predicate backtracks through the various stream properties that unify with `Property` for the stream `Stream`. Currently, the following properties are defined.

- `stream_class(C)` gives the stream class for a file: i.e. `file`, `atom`, `console` or `pipe`.
- `file_name(F)` is a property of `Stream`, if `Stream` is a file stream and `F` is the file name associate with `Stream`. The full operating system path is used.

- `type(T)` is a property of `Stream`, if `Stream` is a file stream and `T` is the file type of `Stream`: `text` or `binary`.
- `mode(M)` is a property of `Stream`, if `M` represents the I/O mode with which `Stream` was opened: i.e. `read`, `write`, `append`, `write_binary`, etc., as appropriate for the class of `Stream`.
- `alias(A)` is a property of `Stream`, if `Stream` was opened with alias `A`.
- `input` is a property of `Stream`, if `Stream` was opened in the I/O mode: `read`.
- `output` is a property of `Stream`, if `Stream` was opened in the I/O mode: `write`, `append`, `write_binary`, or `append_binary`.
- `reposition(Bool)` is true, if `Stream` is repositionable, and false otherwise.
- `end_of_stream(E)` returns `at` if the end of stream condition for `Stream` is true, and `not` otherwise.
- `position(Pos)` returns the current position of the stream as determined by `fseek` or the byte-offset of the current stream within an atom. In either case, if an end-of-stream condition occurs, the token `end_of_file` is returned.
- `eof_action(Action)` is `reposition` if the stream class is `console`, `eof_code` if the stream class is `file`, and `error` is the stream class is `pipe` or `atom`.

`flush_output(+Stream_or_alias)` ISO

Any buffered data in `Stream_or_alias` gets flushed. If `Stream` is not buffered (i.e. if it is of class `atom`), no action is taken.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable, nor a a stream term nor an alias.
  - `domain_error(Stream_or_alias,Stream)`
- `Stream` is not associated with an open output stream
  - `existence_error(Stream_or_alias,Stream)`
- Flushing (i.e. `fflush()`) returns an error.
  - `permission_error(flush,stream,Stream)`

`flush_output` ISO

Any buffered data in the current output stream gets flushed.

`set_stream_position(+Stream_or_alias,+Position)` ISO

If the stream associated with `Stream_or_alias` is repositionable (i.e. is a file or atom), sets the stream position indicator for the next input or output operation. Position is a positive integer, taken to be the number of bytes the stream is to be placed from the origin.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable, nor a a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Position` is not instantiated to a positive integer.
  - `domain_error(stream_position,Position)`
- `Stream_or_alias` is not associated with an open stream
  - `existence_error(stream,Stream_or_alias)`
- `Stream_or_alias` is not repositionable, or repositioning returns an error.
  - `permission_error(resposition,stream,Stream_or_alias)`

`at_end_of_stream(+Stream_or_alias)` ISO

Succeeds if `Stream_or_alias` has position at or past the end of stream.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable, nor a a stream term nor an alias.
  - `domain_error(stream,Stream_or_alias)`
- `Stream_or_alias` is not an open stream
  - `existence_error(stream,Stream_or_alias)`

`at_end_of_stream` ISO

Acts as `at_end_of_stream/1` but using the current input stream.

Other Predicates using ISO Streams

`file_reopen(+FileName,+Mode,+Stream,-RetCode)`
Takes an existing I/O stream, closes it, then opens it and attaches it to a file. This can be used to redirect I/O from any of the standard streams to a file. For instance,

```
| ?- file_reopen('/dev/null', w, 3, Error).
```

redirects all warnings to the Unix black hole.

On success, `RetCode` is 0; on error, the return code is negative.

`file_clone(+SrcStream,?DestStream,-RetCode)`
This is yet another way to redirect I/O. It is a Prolog interface to the C `dup` and `dup2` system calls. If `DestStream` is a variable, then this call creates a new XSB I/O stream that is a clone of `SrcStream`. This means that I/O sent to either stream goes to the same place. If `DestStream` is not a variable, then it must be a number corresponding to a valid I/O stream. In this case, XSB closes `DestStream` and makes it into a clone of `SrcStream`.

For instance, suppose that 10 is a I/O Stream that is currently open for writing to file `foo.bar`. Then

```
| ?- file_clone(10,3,_).
```

causes all messages sent to XSB standard warnings stream to go to file `foo.bar`. While this could be also done with `file_reopen`, there are things that only `file_clone` can do:

```
| ?- file_clone(1,10,_).
```

This means that I/O stream 10 now becomes clone of standard output. So, all subsequent I/O will now go to standard output instead of `foo.bar`.

On success, `RetCode` is 0; on error, the return code is negative.

`file_truncate(+Stream, +Length, -Return)` module: `file_io`
The regular file referenced by the Stream`Stream` is chopped to have the size of `Length` bytes. Upon successful completion `Return` is set to zero.

Portability Note: Under Windows (including Cygwin) `file_truncate/2` is implemented using `_chsize()`, while on Unix `ftruncate()` is used. There are

minor semantic differences between these two system calls, which are reflected by the behavior of `file_truncate/2` on different platforms.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable, nor a a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open stream
  - `existence_error(stream,Stream_or_alias)`
- `Length` is a variable
  - `instantiation_error`
- `Length` is neither a variable nor an integer
  - `type_error(integer,Length)`

`tmpfile_open(-Stream)`

Opens a temporary file with a unique filename. The file is deleted when it is closed or when the program terminates.

`flush_all_output_streams` module: `error_handler`

Flushes output streams, both user and system `STDOUT`, `STDERR`, etc. This convenience predicate is written as

```
flush_all_open_streams:-
        stream_property(S,mode(X)),(X = append ; X = write),flush_output(S),fail.
flush_all_open_streams.
```

6.1.4 DEC-IO Style File Handling

`see(+File_or_stream)`

Makes `File_or_stream` the current input stream.

- If there is an open input stream associated with the file that has `File_or_stream` as its file name, and that stream was opened previously, 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` throws a permission error.

Note that `see/1` is incompatible with ISO aliases – calling `see(Alias)` with an ISO alias will try to open a file named `Alias` rather than using the alias. 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).

Error Cases

- `File_or_stream` is a variable
  - `instantiation_error`
- `File_or_stream` is neither a variable nor an atomic file identifier nor a stream identifier.
  - `domain_error(stream_or_path,F)`
- File `File_or_stream` is directory or file is not readable.
  - `permission_error(open,file,F)`
- File `File_or_stream` does not exist.
  - `existence_error(stream_or_path,F)`

`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 6.12, 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).

The implementation of the ISO predicate `set_output/1`, is essentially that of `tell/1`.

Error Cases

- `File_or_stream` is a variable
  - `instantiation_error`
- `File_or_stream` is neither a variable nor an atomic file identifier nor a stream identifier.
  - `domain_error(stream_or_path,F)`
- File `File_or_stream` is directory or file is not readable.
  - `permission_error(open,file,F)`
- File `File_or_stream` does not exist.
  - `existence_error(stream_or_path,F)`

`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 6.12, and it is only provided for upwards compatibility reasons.

`told`

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

`file_exists(+F)`

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

Error Cases

`instantiation_error` `F` is uninstantiated.

`url_encode(+Filename,-EncodedFilename)`

This predicate is useful when one needs to create a file whose name contains forbidden characters, such as `>`, `<`, and the like. It takes a string and encodes any forbidden character using an appropriate %-sequence of characters that is acceptable as a file name in any OS: Unix, Windows, or Mac. For instance,

```
| ?- url_encode('http://foo''>$',X).

X = http%3a%2f%2ffoo%27%3e%24
```

`url_decode(+Filename,-EncodedFilename)`

This predicate performs the inverse operation with respect to `url_encode/2`. For instance,

```
| ?- url_decode('http%3a%2f%2ffoo%27%3e%24',X).

X = http://foo'>$
```

6.1.5 Character I/O

Beginning with Version 3.8, XSB supports Unicode in the form of UTF-8 characters. Due to this change, we recommend using ISO-compliant character I/O predicates, rather than older predicates such as `get/1`, `get0/1`, `put/1` and so on. As the use of these older predicates may sometines give unexpected answers when used with non-ASCII characters, they are deprecated, athough they are still available for backward compatibility.

`get_char(+Stream_or_alias,?Char)` ISO

Unifies `Char` with the next UTF-8 character from `Stream_or_alias`, advancing the position of the stream. `Char` is unified with the atom `end_of_file` if an end of file condition is detected.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`
- `Char` is not a variable or character.
  - `domain_error(character_or_variable,Char)`

`get_char(?Char)` ISO

Behaves as `get_char/2`, but reads from the current input stream.

Error Cases

- `Char` is not a variable or character.
  - `domain_error(character_or_variable,Char)`

`get_code(+Stream_or_alias,?Code)` ISO

`Code` unifies with the UTF-8 code of the next character from `Stream_or_alias`. The position of the stream is advanced. `Char` is unified with -1 if an end of file condition is detected.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`
- `Code` is not a variable or character code
  - `domain_error(character_code_or_variable,Code)`

`get_code(?Code)` ISO

Behaves as `get_code/2`, but reads from the current input stream [2].

Error Cases

- `Code` is not a variable or character code
  - `domain_error(character_code_or_variable,Code)`

`get_byte(+Stream_or_alias,?Byte)` ISO

`Byte` unifies with the value of the the next byte from `Stream_or_alias`. The position of the stream is advanced. `Char` is unified with -1 if an end of file condition is detected. If reading from ASCII text, `get_byte/2` will have the same behavior as `get_code/2`, but in general `get_code/2` may return multi-byte characters

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.

[2] The obsolescent predicate `get0/1` is defined as `get_code/1`.

- `domain_error(stream_or_alias,Stream_or_alias)`

- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`
- `Code` is not a variable or character code
  - `domain_error(character_code_or_variable,Code)`

`get_byte/1` ISO

Behaves as `get_byte/2`, but reads from the current input stream [3].v

Error Cases

- `Code` is not a variable or `Code` is not a proper value for a byte
  - `domain_error(byte_code_or_variable,Code)`

`peek_char(+Stream_or_alias,?Char)` ISO

Unifies `Char` with the next UTF-8 character from `Stream_or_alias`. The position in `Stream_or_alias` is unchanged. `Char` is unified with the atom `end_of_file` if an end of file condition is detected.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`
- `Char` is not a variable or character.
  - `domain_error(character_or_variable,Char)`

`peek_char(?Char)` ISO

Behaves as `peek_char/2`, but the current input stream is used.

Error Cases

- `Char` is not a variable or character.
  - `domain_error(character_or_variable,Char)`

[3]The obsolescent predicate `get0/1` is defined using `get_byte/1`, but returns the next byte that does not match an ASCII whitespace character.

`peek_code(+Stream_or_alias,?Code)` ISO

Unifies `Code` with the next UTF-8 code from `Stream_or_alias`. The position in `Stream_or_alias` is unchanged. `Code` is unified with -1 if an end of file condition is detected.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`
- `Code` is not a variable or character.
  - `domain_error(character_code_or_variable,Code)`

`peek_code(?Code)` ISO

Behaves as `peek_code/2`, but the current input stream is used.

Error Cases

- `Char` is not a variable or character.
  - `domain_error(character_code_or_variable,Code)`

`peek_byte(?Byte)` ISO

Unifies `Byte` with the next byte from `Stream_or_alias`. The position in `Stream_or_alias` is unchanged. `Code` is unified with -1 if an end of file condition is detected.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`
- `Code` is not a variable or character.
  - `domain_error(byte_code_or_variable,Code)`

`peek_byte(?Byte)` ISO

Behaves as `peek_byte/2`, but the current input stream is used.

Error Cases

- `Char` is not a variable or character.
  - `domain_error(byte_code_or_variable,Code)`

`put_char(+Stream_or_alias,+Char)` ISO

Writes a UTF-8 character `Char` to `Stream_or_alias`.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`
- `Char` is a not a character
  - `type_error(character,Char)`

`put_char(+Char)` ISO

Puts a UTF-8 character `Char` to the current output stream.

Error Cases

- `Code` is a not a character.
  - `type_error(character,Char)`

`put_code(+Stream,+Code)` ISO

Puts the character for the UTF-8 code `Code` to `Stream_or_alias`.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream

  - existence_error(stream,Stream_or_alias)
- Code is a not a character code
  - type_error(character_code,Code)

put_code(+Code) ISO

Puts the character for the UTF-8 code Code to the current output stream [4].

Error Cases

- Code is a not a character code.
  - type_error(character_code,Code)

nl ISO

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

nl(+Stream_or_alias) ISO

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

Error Cases

- Stream_or_alias is a variable
  - instantiation_error
- Stream_or_alias is neither a variable nor a stream term nor an alias.
  - domain_error(stream_or_alias,Stream_or_alias)
- Stream_or_alias is not associated with an open stream
  - existence_error(stream,Stream_or_alias)

tab(+N)

Puts N spaces to the current output stream.

Error Cases

- Code is a not a positiveInteger
  - domain_error(positiveInteger,Code)

[4]The obsolescent predicate put/1 is defined as put_code/1.

6.1.6 Term I/O

Beginning with Version 3.8, XSB automatically supports Unicode in the form of UTF-8 characters for reading and writing.

`read(?Term)` ISO

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 presence 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.

In Version 3.8, `read/[1,2]` are non ISO-compliant in how they handle syntax errors or their behavior when encountering an end of file indicator.

`read(+Stream_or_alias, ?Term)` ISO

`read/2` has the same behavior as `read/1` but the input stream is explicitly designated by `Stream_or_alias`.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open stream
  - `existence_error(stream,Stream_or_alias)`

`read_canonical(-Term)`

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.

`read_canonical(+Stream_or_alias),-Term)`

Behaves as `read_canonical/1`, but reads from `Stream_or_alias`.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`

`read_term(?Term,?OptionsList)` ISO

A term is read from the current input stream as in `read/1`; but `OptionsList` is a (possibly empty) list of *read options* that specifies additional behavior. The read options include

- `variables(Vars)`: once a term has been read, `Vars` is a list of the variables in the term, in left-to-right order.
- `variable_names(VN_List)`: once a term has been read `VN_List` is a list of non-anonymous variables in the term. The elements of the list have the form `A = V` where `V` is a non-anonymous variable of the term, and `A` is the string used to denote the variable in the input stream.
- `singletons(VS_List)`: once a term has been read `VN_List` is a list of the non-anonymous `singleton` variables in the term. The elements of the list have the form `A = V` where `V` is a non-anonymous variable of the term, and `A` is the string used to denote the variable in the input stream.

Error Cases

- `OptionsList` is a variable, or is a list containing a variable element.
  - instantiation_error
- `OptionsList` contains a non-variable element `O` that is not a read option.
  - `domain_error(read_option,O)`

`read_term(+Stream_or_alias, ?Term,?OptionsList)` ISO

`read_term/3` has the same behavior as `read_term/2` but the input stream is explicitly designated using the first argument.

Error Cases are the same as `read_term/2`, but with the additional errors that may arise in stream checking.

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open stream
  - `existence_error(stream,Stream_or_alias)`

`write_term(?Term,+Options)` ISO

Outputs `+Term` to the current output stream. `Stream` (`write_term/3`) according to the list of write options, `Options`. The current set of write options which form a superset of the ISO-standard write options, are as follows:

- `quoted(+Bool)`. If `Bool = true`, then atoms and functors that can't be read back by `read/1` are quoted, if `Bool = false`, each atom and functor is written as its unquoted name. Default value is `false`.
- `ignore_ops(+Bool)`. If `Bool = true` each compound term is output in functional notation; curly brackets and list braces are ignored, as are all explicitly defined operators. If `Bool = false`, curly bracketed notation and list notation is enabled when outputting compound terms, and all other operator notation is enabled. Default value is `false`.
- `numbervars(+Bool)`. If `Bool = true`, a term of the form `'$VAR'(N)` where `N` is an integer, is output as a variable name consisting of a capital letter possibly followed by an integer. A term of the form `'$VAR'(Atom)` where `Atom` is an atom, is output as itself (without quotes). Finally, a term of the form `'$VAR'(String)` where `String` is a character string, is output as the atom corresponding to this character string. If `bool` is `false` this cases are not treated in any special way. Default value is `false`.
- `max_depth(+Depth)`. `Depth` is a positive integer or zero. If positive, it denotes the depth limit on printing compound terms. If `Depth` is zero, there is no limit. Default value is `0` (no limit).
- `priority(+Prio)` `Prio` is an integer between 1 and 1200. If the term to be printed has higher priority than `Prio`, it will be printed parenthesized. Default value is 1200 (no term parenthesized).

From the following examples it can be seen that `write_term/[2,3]` can duplicate the behavior of a number of other I/O predicates such as `write/[1,2]`, `writeq/[1,2]`, `write_canonical/[1,2]`, etc.

```
| ?- write_term(f(1+2,'A',"string",'$VAR'(3),'$VAR'('Temp'),(multifile foo)),[]).
f(1 + 2,A,"string",$VAR(3),$VAR(Temp),(multifile foo))
yes

| ?- write_term(f(1+2,'A',"string",'$VAR'(3),'$VAR'('Temp'),(multifile foo)),
                [quoted(true)]).
f(1 + 2,'A',"string",'$VAR'(3),'$VAR'('Temp'),(multifile foo))
yes

| ?- write_term(f(1+2,'A',"string",'$VAR'(3),'$VAR'('Temp'),(multifile foo)),
                [quoted(true),ignore_ops(true),numbervars(true)]).
f(+(1,2),'A','.'(115,'.'(116,'.'(114,'.'(105,'.'(110,'.'(103,[])))))),D,Temp,(multifile f
yes

| ?- write_term(f(1+2,'A',"string",'$VAR'(3),'$VAR'('Temp'),(multifile foo)),
                [quoted(true),ignore_ops(true),numbervars(true),priority(1000)]).
f(+(1,2),'A','.'(115,'.'(116,'.'(114,'.'(105,'.'(110,'.'(103,[])))))),D,Temp,multifile(fo
yes
```

Error Cases

- `Options` is a variable
  - `instantiation_error`
- `Options` neither a variable nor a list
  - `type_error(list,Options)`
- `Options` contains a variable element, `O`
  - `instantiation_error`
- `Options` contains an element `O` that is neither a variable nor a write option.
  - `domain_error(write_option,O)`

ISO Compatibility Note: In Version 3.8, `write_term/[2,3]` do not properly handle operators.

`write_term(+Stream_or_alias,?Term,+Options)` ISO

Behaves as `write_term/2`, but writes to `Stream_or_alias`.

Error Cases are the same as `write_term/2` but with these additions.

- `Stream_or_alias` is a variable

  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open output stream
  - `existence_error(stream,Stream_or_alias)`

`write(?Term)` ISO

Semantically, `write/1` behaves as if `write_term/1` were invoked using `quoted(false)`, `ignore_ops(false)`, and `numbervars(false)`. Attributed variables are written according to the value of the Prolog flag `write_attributes` (cf. `current_prolog_flag/2`).

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 token).

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

HiLog (or Prolog) 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.

`write/1` treats terms of the form `'$VAR'(N)`, which may be generated by `numbervars/[1,3]` specially: it writes `'A'` if `N`=0, `'B'` if `N`=1, ..., `'Z'` if `N`=25, `'A1'` if `N`=26, etc. `'$VAR'(-1)` is written as the anonymous variable `'_'`.

`write(+Stream_or_alias, ?Term)` ISO

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

Error Cases are the same as `read_term/2`, but with the additional errors that may arise in stream checking.

- `Stream_or_alias` is a variable
  - `instantiation_error`

- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open output stream
  - `existence_error(stream,Stream_or_alias)`

`writeq(?Term)` ISO

Acts as `write_term/1` when defined with the options `quoted(true)`, `numbervars(true)`, and `ignore_ops(false)`. In other words, atoms and functors are quoted whenever necessary to make the result acceptable as input to `read/1` `writeq/1` also treats terms of the form `'\VAR'(N)` specially, writing `A` if `N`= 0, etc., and output is in accordance with current operator definitions. `writeq/1` always succeeds without producing an error.

`writeq(+Stream_or_alias, ?Term)` ISO

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

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open output stream
  - `existence_error(stream,Stream_or_alias)`

`write_canonical(?Term)` ISO

This predicate is provided so that the HiLog term `Term`, if written to a file, can be read back using `read_canonical/[1,2]` or `read/[1,2]` 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`. Except for list-notation (`[]`) and infix comma-list notation, operator declarations are not taken into consideration, so that apart from these exceptions compound terms are 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,[]))`).

Finally, `write canonical/2` writes attributed variables as simple variables.

ISO Compatibility Note: In XSB, list notation and infix comma-list notation are considered canonical both for reading and writing. We find that this improves readability, and that these operators are so standard that there is little likelihood that they will not be in effect by any Prolog reader. We therefore deviate from the ISO standard definition of canonical in these cases.

`write_canonical(+Stream_or_alias, ?Term)` ISO

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

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open output stream
  - `existence_error(stream,Stream_or_alias)`

`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)`.

`write_prolog(?Term)` HiLog

`write_prolog(+Stream_or_alias,?Term)` HiLog

`write_prolog/1` acts as `write/1` except that any proper HiLog term `Term` is written using Prolog syntax – i.e. as a term whose outer functor is apply. `write_prolog/1` outputs `Term` according to the operator declarations in force. Because of this, it differs from `write_canonical/1` described above, despite the fact that both predicates write HiLog terms as Prolog terms.

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

Examples:

```
| ?- write_prolog(X(a,1+2)).
apply(_h120,a,1 + 2)

yes
| ?- write(X(a,1+2)).
_h120(a,1 + 2)

yes
| ?- write_canonical(X(a,1+2)).
apply(_h120,a,+(1,2))

yes
```

`numbervars(+Term, +FirstN,?LastN,+Options)` module: `num_vars`

This predicate provides a mechanism for grounding a (HiLog) term so that it may be analyzed. Each variable in the (HiLog) term `Term` is instantiated to a term of the form `'$VAR'(N)`, where `N` is an integer starting from `FirstN`. `FirstN` is used as the value of `N` for the first variable in `Term` (starting from the left). The second distinct variable in `Term` is given a value of `N` satisfying `"N is FirstN + 1"` and so on. The last variable in `Term` has the value `LastN-1`.

In `numbervars/4`, `Options` can be used to indicate the action to take upon encountering an attributed variable. Currently, `Options` must be either be the empty list, or the list `[attvar(Action)]` or the term `attvar(Action)`, where `Action` is

- `error` Throw a type error if an attributed variable is encountered.
- `bind` Bind attributed variables by unifying them with terms of the form `'$VAR'(N)`.
- `skip` Skip over attributed variables, performing no action on these variables.

Error Cases

- `Options` is a variable
  - `instantiation_error`
- `Options` is not an empty list, the list `[attvar(Action)]` or the term `attvar(Action)` where `Action` is one of `bind`, `error` or `skip`:
  - `domain_error`

`numbervars(+Term, +FirstN, ?LastN)` module: `num_vars`

Acts as `numbervars(+Term, +FirstN, ?LastN,attvar(error))`.

`numbervars(+Term)` module: `num_vars`

This predicate is defined as: `numbervars(Term, 0, _)`. It is included solely for convenience.

`unnumbervars(+Term, +FirstN, ?Copy)` module: `num_vars`

This predicate is a partial inverse of predicate `numbervars/3`. It creates a copy of Term in which all subterms of the form `'$VAR'(<int>)` where `<int>` is not less than `FirstN` are uniformly replaced by variables. `'$VAR''` subterms with the same integer are replaced by the same variable. Also a version `unnumbervars/2` is provided which calls `unnumbervars/3` with the second parameter set to 0.

Term Writing to Designated I/O Streams

While XSB has standard I/O streams for errors, warnings, messages, and feedback (cf. Section 6.1.1), the predicates above write to `STDOUT` which is the standard output for the process. Most of the time there is no issue with this as these streams are aliased to `STDOUT`. However in a number of circumstances, `STDOUT` may be redirected: a user may have invoked `tell/1`, XSB may be invoked through C or interprolog, etc. In such cases, it may be useful to ensure that output goes to one of the other I/O streams.

`error_write(?Message)` module: `standard`
`error_writeln(?Message)` module: `standard`

These predicates output `Message` to XSB's `STDERR` stream, rather than to XSB's `STDOUT` stream, as does `write/1` and `writeln/1`. In addition, if `Message` is a list or comma list, the elements in the comma list are output as if they were concatenated together. Each of these predicates must be imported from the module `standard`.

`console_write(?Message)` module: `standard`
`console_writeln(?Message)` module: `standard`

As above, but writes to `STDFDBK`, the console feedback stream.

`warning(?Message)` module: `standard`

By default, this predicate outputs `Message` to XSB's `STDWARN` stream, rather than to XSB's `STDOUT` stream, as does `write/1` and `writeln/1`. In addition, if `Message` is a list or comma list, the elements in the comma list are output as if they were concatenated together. Each of these predicates must be imported from the module `standard`.

The default behavior for warnings can be altered by setting the value of the Prolog flag `warning_action` to either `silent_warning` which performs no action when `warning/1` is called. or `error_warning` which throws a miscellaneous exception when `warning/1` is called (WARNING: this includes compiler warnings). The default behavior can be restored by setting `warning_action` to `print_warning`.

`message(?Message)` module: `standard`
`messageln(?Message)` module: `standard`

As above, but writes to `STDMSG` the standard stream for messages.

6.1.7 Special I/O

`fmt_read(+Fmt,-Term,-Ret)`

`fmt_read(+Stream,+Fmt,-Term,-Ret)`

These predicates provides a routine for reading data from the current input file (which must have been already opened by using `see/1`) according to a C format, as used in the C function `scanf`. `Fmt` must be a string of characters (enclosed in ") representing the format that will be passed to the C call to `scanf`. See the C documentation for `scanf` for the meaning of this string. The usual alphabetical C escape characters (*e.g.*, $\backslash n$) are recognized, but not the octal or the hexadecimal ones. Another difference with C is that, unlike most C compilers, XSB insists that a single `%` in the format string signifies format conversion specification. (Some C compilers might output `%` if it is not followed by a valid type conversion spec.) So, to output `%` you must type `%%`. Format can also be an atom enclosed in single quotes. However, in that case, escape sequences are not recognized and are printed as is.

`Term` is a term (*e.g.*, `args(X,Y,Z)`) whose arguments will be unified with the field values read in. (The functor symbol of `Term` is ignored.) Special syntactic sugar is provided for the case when the format string contains only one format specifier: If `Term` is a variable, `X`, then the predicate behaves as if `Term` were `arg(X)`.

If the number of arguments exceeds the number of format specifiers, a warning is produced and the extra arguments remain uninstantiated. If the number of format specifiers exceeds the number of arguments, then the remainder of the format string (after the last matching specifier) is ignored.

Note that floats do not unify with anything. `Ret` must be a variable and it will be assigned a return value by the predicate: a negative integer if end-of-file is encountered; otherwise the number of fields read (as returned by `scanf`.)

`fmt_read` cannot read strings (that correspond to the `%s` format specifier) that are longer than 16K. Attempting to read longer strings will cause buffer overflow. It is therefore recommended that one should use size modifiers in format strings (*e.g.*, `%2000s`), if such long strings might occur in the input.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open output stream
  - `existence_error(stream,Stream_or_alias)`

If the number of arguments in `Term` is greater than the number of conversion specifiers in `Fmt` no error is thrown, but a warning is issued.

`fmt_write(+Fmt,+Term)`

`fmt_write(+Stream_or_alias,+Fmt,+Term)`

These predicates provide routines for writing formatted data to a given output stream (`fmt_write/3`) or the current output stream (`fmt_write/2`).

`Fmt` should be a Prolog character list (string) or atom. A Prolog character list is preferred, as space can be more easily reclaimed for character lists than for atoms. `Term` is a Prolog term (*e.g.*, `args(X,Y,Z)`) whose arguments will be output. The number of arguments in `Term` should equal the number of conversion specifiers in `Fmt`. The functor symbol of `Term` is ignored [5].

Allowable syntaxes for `Fmt` reflect the syntax of the C function `printf()` on a given platform, with the following exceptions

- The usual alphabetical C escape characters (*e.g.*, $\backslash n$) are recognized, but not the octal or the hexadecimal ones.
- `%S` is supported, in addition to the usual C conversion specifiers. The corresponding argument can be any Prolog term. This provides an easy way to print the values of Prolog variables, etc.
- `%!` is supported and indicates that the corresponding argument is to be ignored and will generate nothing in the output.

[5] In the case where `Fmt` contains only a single conversion specifier, `Term` may be a string, integer or a float, and is considered to be equivalent to specifying `arg(Term)`.

- A single % in the format string must be followed by a conversion operator (e.g. d, s, etc.). (Some C compilers output % if the percentage character is not followed by a valid type conversion spec.) However, to output %, fmt_write must contain %%.

Example

```
| ?- fmt_write("%d %f %s %S \n",args(1,3.14159,ready,hello(world))).
1 3.141590 ready hello(world)

yes
```

XSB also offers an alternate version of formatted output in the format library described in volume 2. While not as efficient as fmt_write/[2,3], the format library is more compatible with the formatted output found in other Prologs.

Error Cases

- Stream_or_alias is a variable
  - instantiation_error
- Stream_or_alias is neither a variable nor a stream term nor an alias.
  - domain_error(stream_or_alias,Stream_or_alias)
- Stream_or_alias is not associated with an open output stream
  - existence_error(stream,Stream_or_alias)
- Fmt is uninstantiated or not a character string or atom
  - type_error('character string or atom',Fmt)
- A format specifier in Fmt and its corresponding argument in Term are of incompatible types.
  - misc_error
- Term contains fewer arguments than Fmt has format specifiers or Term is uninstantiated
  - misc_error

If the number of arguments in Term is greater than the number of conversion specifiers in Fmt no error is thrown, but a warning is issued.

Caution for 64-bit Platforms As discussed, fmt_write/[2,3] calls printf() and inherits the flexibility of that function, but also its "features". One of

these features is that in most 64-bit platforms, large integers that behave perfectly well otherwise are not printed out properly by `printf()` with the `%d` format – rather another format string needs to be used (such as `%ld` on Linux). `fmt_write/[1,2]` recognizes the `%ld` option and passes it onto `fprintf()`, but the proper format string for 64-bit integers may be different on other platforms.

`fmt_write_string(-String,+Fmt,+Term)`

This predicate works like the C function `sprintf`. It takes the format string and substitutes the values from the arguments of `Term` (*e.g.*, `args(X,Y,Z)`) for the formatting instructions %s, %d, etc. Additional syntactic sugar, as in `fmt_write`, is recognized. The result is available in `String`. `Fmt` is a string or an atom that represents the format, as in `fmt_write`.

If the number of format specifiers is greater than the number of arguments to be printed, an error is issued. If the number of arguments is greater, then a warning is issued.

`fmt_write_string` requires that the printed size of each argument (*e.g.*, X,Y,and Z above) must be less than 16K. Longer arguments are cut to that size, so some loss of information is possible. However, there is no limit on the total size of the output (apart from the maximum atom size imposed by XSB).

`file_read_line_list(-String)`

A line read from the current input stream is converted into a list of character codes. This predicate avoids interning an atom as does `file_read_line_atom/3`, and so is recommended when speed is important. This predicate fails on reaching the end of file.

`file_read_line_list(Stream_or_alias,-CharList)`

Acts as does `file_read_line_list`, but uses `Stream_or_atom`.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`

`file_read_line_atom(-Atom)`

Reads a line from the current (textual) input stream, returning it as `Atom`. This predicate fails on reaching the end of file.

`file_read_line_atom(+Stream_or_alias,-Atom)`

Like `file_read_line_atom/1` but reads from `Stream_or_alias`. **Error Cases**

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`

`file_write_line(+String, +Offset)` module: `file_io`
`file_write_line(+Stream_or_alias, +String, +Offset)` module: `file_io`

These predicates write `String` beginning with character `Offset` to the current output stream. `String` can be an atom or a list of UTF-8 character codes. This does *not* put the newline character at the end of the string (unless `String` already had this character). Note that escape sequences, like `\n`, are recognized if `String` is a character list, but are output as is if `String` is an atom.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`
- `String` is neither a Prolog character list not an atom
  - `misc_error`

`file_getbuf_list(+Stream_or_alias, +BytesRequested, -CharList, -BytesRead)` module: `file_io`

Read `BytesRequested` bytes from file represented by `Stream_or_alias` (which must already be open for reading) into variable `String` as a list of character codes. This is analogous to `fread` in C. This predicate always succeeds. It does not distinguish between a file error and end of file. You can determine if either of these conditions has happened by verifying that `BytesRead` < `BytesRequested`.

`file_getbuf_list(+BytesRequested, -String, -BytesRead)` module: `file_io`

Like `file_getbuf_list/3`, but reads from the currently open input stream (*i.e.*, with `see/1`).

`file_getbuf_atom(+Stream_or_alias, +BytesRequested, -String, -BytesRead)` module: `file_io`

Read `BytesRequested` bytes from file represented by `Stream_or_alias` (which must already be open for reading) into variable `String`. This is analogous to `fread` in C. This predicate always succeeds. It does not distinguish between a file error and end of file. You can determine if either of these conditions has happened by verifying that `BytesRead` < `BytesRequested`.

Note: although XSB has an atom table garbage collector, this predicate is inefficent to read large files. It is usually best to use `read_getbuf_list` or another predicate in such a case.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`
- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`

`file_getbuf_atom(+BytesRequested, -String, -BytesRead)` module: `file_io`

Like `file_getbuf_atom/4`, but reads from the currently open input stream.

`file_putbuf(+Stream_or_alias, +BytesRequested, +String, +Offset, -BytesWritten)` module: `file_io`

Write `BytesRequested` bytes into file represented by I/O port `Stream_or_alias` (which must already be open for writing) from variable `String` at position `Offset`. This is analogous to C `fwrite`. The value of `String` can be an atom or a list of UTF-8 characters.

Error Cases

- `Stream_or_alias` is a variable
  - `instantiation_error`

- `Stream_or_alias` is neither a variable nor a stream term nor an alias.
  - `domain_error(stream_or_alias,Stream_or_alias)`
- `Stream_or_alias` is not associated with an open input stream
  - `existence_error(stream,Stream_or_alias)`

`file_putbuf(+BytesRequested, +String, +Offset, -BytesWritten)` module: `file_io`

Like `file_putbuf/3`, but output goes to the currently open output stream.

6.2 Interactions with the Operating System

XSB provides a number of facilities for interacting with the UNIX and Windows operating systems. This section describes basic facilities for invoking shell commands and file manipulation. Chapter 1 of Volume 2 discusses more advanced commands for process spawning and control, along with interprocess communication.

`shell(+SystemCall)`

Calls the operating system with the atom `SystemCall` as argument, using the `libc` function `system()`. The predicate succeeds if `SystemCall` is executed successfully; otherwise it fails. As a convenience, the user can also supply `SystemCall` either as an atom or as a list of atoms. If a list of atoms is used, elements of the list will be concatenated together to form the system call.

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 `system()` (and `shell/1`) executes by forking off a shell process. Thus it cannot be used, for example, to change the working directory of the program. For that reason the standard predicate `cd/1` described below should be used.

Error Cases

- `SystemCall` is a variable

  - instantiation_error
- SystemCall is neither an atom nor a list
  - type_error(atom_or_list,SystemCall)
- SystemCall is longer than the maximum command length allowed by shell/1
  - resource_error(memory)

shell(+SystemCall, -Result)

As with shell/1, this predicate calls the operating system with the atom SystemCall as argument, using the libc function system() using the same forms of input. shell/2 always succeeds instantiating Result to the exit code of system(). Thus Result will be 0 if SystemCall executed properly, and non-0 otherwise: the specific return values of system() may be platform-dependent.

Error Cases

- SystemCall is a variable
  - instantiation_error
- SystemCall is neither an atom nor a list
  - type_error(atom_or_list,SystemCall)
- Result is not a variable
  - type_error(variable,Result)
- SystemCall is longer than the maximum command length allowed by shell/2
  - resource_error(memory)

shell_to_list(+SystemCall,-StdOut,-ErrOut,-Result)

shell_to_list(+SystemCall,-StdOut,-Result)

Behaves as shell/2 in its 1st and 4th arguments, and like shell/2 always succeeds. Both StdOut and ErrOut are lists of lists: each element of the outer list corresponds to a line of output from SystemCall, while each element of an inner list corresponds to a token in that line. shell_to_list/3 is thus a sort of Prolog analog of the shell command designated by SystemCall.

Examples (from OSx):

```
?- shell_to_list(sw_vers,Stdout,Ret).
```

```
Stdout = [[ProductName:,Mac,OS,X],[ProductVersion:,10.4.9],[BuildVersion:,8P2137]]
Ret = 0

?- shell_to_lists('gcc -c nofile.c',StdOut,StdErr,Ret).

Stdout = []
StdErr = [[i686-apple-darwin8-gcc-4.0.1:,nofile.c:,No,such,file,or,directory]]
Ret = 256
```

Error cases are as with `shell/2`

`datime(?Date)` module: `standard`

Unifies `Date` to the current UTC date, returned as a Prolog term, suitable for term comparison. Note that `datime/1` must be explicitly imported from the module `standard`.

Example:

```
> date
Mon Aug  9 16:19:44 EDT 2004
> xsb
XSB Version 2.6 (Duff) of June 24, 2003
[i686-pc-cygwin; mode: optimal; engine: slg-wam; gc: indirection; scheduling: loca

| ?- import datime/1 from standard

yes
| ?- datime(F).
F = datime(2004,8,9,20,20,23)

yes
```

`local_datime(?Date)` module: `standard`

Acts as `datime/1`, but returns the local, rather than the UTC date.

`epoch_seconds(-Seconds)` module: `machine`
`epoch_milliseconds(-Seconds,-Milliseconds)` module: `machine`

Returns the number of seconds since the beginning of the POSIX/UNIX epoch (January 1, 1970) [6]. May cause overflow on 32-bit platforms. `epoch_milliseconds/2` returns both the number of seconds and the number of additional milliseconds since the last whole second.

[6] Uses the Posix call `time(0)`, so the number of seconds will be returned on non-Unix platforms, such as Microsoft.

`sleep(+Seconds)` module: `shell`

Put XSB to sleep for a given number of seconds.

Error Cases

- `Seconds` is a variable
  - `instantiation_error`.
- `Seconds` is not an integer
  - `type_error(integer, Seconds)`.

`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 a permission error.

Error Cases

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

`type_error Dir` is not an atom.

`getenv(+VarName,-VarVal)` module: `machine`

Unifies `VarVal` with the value of `VarName` in the current shell. If `VarName` is not an environment varible, the predicate fails.

Example:

```
:- import getenv/2 from machine.

yes
| ?- getenv('HOSTTYPE',F).

F = intel-pc
```

`putenv(+String)` module: `machine`

If `String` is of the form `VarName=Value`, inserts or resets the environment variable `VarName`. If `VarName` does not exist, it is inserted with `VarVal`. If the `VarName` does exist, it is reset to `VarVal`. `putenv/2` always succeeds.

Exceptions:

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

`type_error VarName` or `VarVal` is not an atom or a list of atoms.

6.2.1 The `path_sysop/2` interface

In addition, XSB provides the following unified interface to the operations on files. All these calls succeed iff the corresponding system call succeeds. These calls work on both Windows and Unixes unless otherwise noted.

`path_sysop(isplain, +Path)`
: Succeeds, if `Path` is a plain file.

`path_sysop(isdir, +Path)`
: Succeeds, if `Path` is a directory.

`path_sysop(rename, +OldPath, +NewPath)`
: Renames `OldPath` into `NewPath`.

`path_sysop(copy, +FromPath, +ToPath)`
: Copies `FromPath` into `ToPath`.

`path_sysop(rm, +Path)`
: Removes the plain file `Path`.

`path_sysop(rmdir, +Path)`
: Deletes the directory `Path`, succeeding only if the directory is empty.

`path_sysop(rmdir_rec, +Path)`
: Deletes the directory `Path` along with any of its contents.

`path_sysop(link, +SrsPath, +DestPath)`
: Creates a hard link from `SrsPath` to `DestPath`. UNIX only.

`path_sysop(cwd, -Path)`
: Binds `Path` to the current working directory.

`path_sysop(chdir, +Path)`
: Changes the current working directory to `Path`.

`path_sysop(mkdir, +Path)`
: Creates a new directory, `Path`.

`path_sysop(exists, +Path)`
: Succeeds if the file `Path` exists.

`path_sysop(readable, +Path)`
: Succeeds if `Path` is a readable file.

`path_sysop(writable, +Path)`
: Succeeds if `Path` is a writable file.

`path_sysop(executable, +Path)`
: Succeeds if `Path` is an executable file.

`path_sysop(modtime, +Path, -Time)`
: Returns a list that represents the last modification time of the file. Succeeds if file exists. In this case, `Time` is bound to a list `[high,low]` where `low` is the least significant 24 bits of the modification time and `high` is the most significant bits (25th) and up. `Time` represents the last modification time of the file. The actual value is thus $\texttt{high} * 2^{24} + \texttt{low}$, which represents the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).

`path_sysop(newerthan, +Path1, +Path2)`
: Succeeds is the last modification time of `Path1` is higher than that of `Path2`. Also succeeds if `Path1` exists but `Path2` does not.

`path_sysop(size, +Path, -Size)`
: Returns a list that represents the byte size of `Path`. Succeeds if the file exists. In this case `Size` is bound to the list of the form `[high,low]` where `low` is the least significant 24 bits of the byte-size and `high` is the most significant bits (25th) and up. The actual value is thus $\texttt{high} * 2^{24} + \texttt{low}$.

`path_sysop(tmpfilename, -Name)`
: Returns the name of a new temporary file. This is useful when the application needs to open a completely new temporary file.

`path_sysop(extension, +Name, -Ext)`
: Returns file name extension.

`path_sysop(basename, +Name, -Base)`
: Returns the base name of the file name (*i.e.*, the name sans the directory and the extension).

`path_sysop(dirname, +Name, -Dir)`
: Returns the directory portion of the filename. The directory is slash or backslash terminated.

`path_sysop(isabsolute, +Name)`
: Succeeds if `Name` is an absolute path name. File does not need to exist.

`path_sysop(expand, +Name, -ExpandedName)`

Binds `ExpandedName` to the expanded absolute path name of `Name`. The file does not need to exist. Duplicate slashes, references to the current and parent directories are factored out.

6.3 Evaluating Arithmetic Expressions through `is/2`

Before describing `is/2` and the expressions that it can evaluate, we note that in Version 3.8 of XSB, integers in XSB are represented using a single word of 32 or 64 bits, depending on the machine architecture. Floating point values are, by default, stored as word-sized references to double precision values, regardless of the target machine. Direct (non-referenced, tagged) single precision floats can be activated for speed purposes by passing the option –enable-fast-floats to the configure script at configuration time. This option is not recommended when any sort of precision is desired, as there may be as little as 28 bits available to represent a given number value under a tagged architecture.

All of the evaluable functors described below throw an instantiation error if one of their evaluated inputs is a variable, and an `evaluation(undefined)` error if one of their evaluated inputs is instantiated but non-numeric. With this in mind, we describe below only their behavior on correctly typed input.

ISO Compatibility Note: In addition, evaluation of arithmetic expressions through `is/2` does not check for overflow or underflow. As a result, XSB's floating point operations do not conform to IEEE floating point standards, and deviates in this regard from the ISO Prolog standard (see [37] Section 9) We hope to fix these problems in a future release [7].

`is(?Result,+Expression)` ISO

`is(Result,Expression)` is true iff the result of evaluating `Expression` as a sequence of evaluable functors unifies with `Result`. As mentioned in Section 3.10.6, `is/2` is an inline predicate, so calls to `is/2` within compiled code will not be visible during a trace of program execution.

Error Cases

`instantiation_error` `Expression` contains an uninstantiated value

[7] We also note that the ISO Prolog evaluable functors`float_integer_part/1` (which can be obtained via `truncate/1`), `float_fractional_part/1` (which can be obtained via `X - truncate(X)`), and bitwise complement (which is implementation dependent in the ISO standard) are not implemented in Version 3.8.

`domain_error(`$< function >, < value >$ `Expression` contains a $function$ applied to $value$, but $value$ is not part of the domain of $function$.

For `is/2` the action for the above error cases can be altered so that the `is/2` literal is treated as having a truth value of *undefined* in the well-founded semantics. This is done via the Prolog flag `exception_action`.

6.3.1 Evaluable Functors for Arithmetic Expressions

`+(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns `Number1 + Number2`, performing any necessary type conversions.

`-(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns `Number1 - Number2`, performing any necessary type conversions.

`*(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns `Number1 * Number2` (i.e. multiplies them), performing any necessary type conversions.

`/(+Expr1,Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns `Number1 / Number2` (i.e. divides them), performing any necessary type conversions.

`div(+Expr1,Expr2)` ISO

`//(+Expr1,Expr2)` Evaluable Functor

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns `Number1 // Number2` (i.e. integer division), performing any necessary type conversions, and rounding to 0 if necessary.

Example:

```
| ?- X is 3/2.

X = 1.5000

yes
| ?- X is 3 // 2.

X = 1

yes
```

```
| ?- X is -3 // 2.

X = -1

yes
```

`-(+Expr1)` Evaluable Functor (ISO)

If `+Expr` evaluates to `Number`, returns `-Number1`, performing any necessary type conversions.

`'∧'(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns the bitwise conjunction of `Number1` and `Number2`.

`'∨'(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns the bitwise disjunction `Number1` and `Number2`.

`'»'(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns the logical shift right of `Number1`, `Number2` places.

`'«'(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns the logical shift left of `Number1`, `Number2` places.

`xor(+Expr1,+Expr2)` ISO

`'><'(+Expr1,+Expr2)` Evaluable Functor

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns the bitwise exclusive or of `Number1` and `Number2`.

`min(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns the minimum of the two.

`max(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1`, and `Expr2` evaluates to `Number2`, returns the maximum of the two.

`ceiling(+Expr)` Evaluable Functor (ISO)

If `+Expr` evaluates to `Number`, `ceiling(Number)` returns the integer ceiling of `Number` if `Number` is a float, and `Number` itself if `Number` is an integer.

`float(+Expr)` Evaluable Functor (ISO)

If `+Expr` evaluates to `Number`, `float(Number)` converts `Number` to a float if `Number` is an integer, and returns `Number` itself if `Number` is a float.

`floor(+Expr)` Evaluable Functor (ISO)

If `+Expr` evaluates to `Number`, `floor(Number)` returns the integer floor of `Number` if `Number` is a float, and `Number` itself if `Number` is an integer.

`mod(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1` and `Expr2` evaluates to `Number2` where `Number2` is not 0, `mod(Number1,Number2)` returns

$$Number1 - (\lfloor (Number1/Number2) \rfloor) \times Number2)$$

`rem(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `+Expr1` evaluates to `Number1` and `Expr2` evaluates to `Number2` where `Number2` is not 0, `rem(Number1,Number2)` returns

$$Number1 - (Number1//Number2) \times Number2)$$

Example:

```
| ?- X is 5 mod 2.

X = 1

yes
| ?- X is 5 rem 2.

X = 1

yes
| ?- X is 5 mod -2.

X = -1

yes
| ?- X is 5 rem -2.

X = 1

yes
```

`round(+Expr)` Evaluable Functor (ISO)

If `+Expr` evaluates to `Number`, `round(Number)` returns the nearest integer to `Number` if `Number` is a float, and `Number` itself if `Number` is an integer.

`^/2` Evaluable Functor (ISO)

If `Expr1` and `Expr2` both evaluate to numbers, the infix function `^/2` raises `Expr1` to the `Expr2` power. If `Expr1` and `Expr2` both evaluate to integers, an integer is returned; otherwise a float is returned.

`'**'(+Expr1,+Expr2)` Evaluable Functor (ISO)

If `Expr1` and `Expr2` both evaluate to numbers, the infix function `**/2` raises `Expr1` to the `Expr2` power. A floating-point number is always returned.

`sqrt(+Expr)` Evaluable Functor (ISO)

If `+Expr` evaluates to `Number`, `sqrt(Number)` returns the square root of `Number`.

`truncate(+Expr)` Evaluable Functor (ISO)

If `+Expr` evaluates to `Number`, `truncate(Number)` truncates `Number` if `Number` is a float, and returns `Number` itself if `Number` is an integer.

`sign(+Expr)` Evaluable Functor (ISO)

If `+Expr` evaluates to `Number`, `sign(Number)` returns `1` if `Number` is greater than 0, `0` if `Number` is equal to 0, and `-1` if `Number` is less than 0.

`pi` Evaluable Functor (ISO)

Evaluates to π within an arithmetic expression.

`e` Evaluable Functor

Evaluates to e, the base of the natural logarithm, within an arithmetic expression. (Use exp(1) for ISO compatibility.)

`epsilon` Evaluable Functor

Evaluates to *epsilon*, the difference between the float 1.0 and the first larger floating point number.

Mathematical Functions from `math.h`

XSB also allows as evaluable functors, many of the functions from the C library `math.h`. Functions included in XSB Version 3.8 are `cos/1` (ISO), `sin/1` (ISO), `tan/1` (ISO), `acos/1` (ISO), `asin/1` (ISO), `atan/1` (ISO), `log/1` (natural logarithm) (ISO), `log10/1`, and `atan/2` (ISO) (also available as `atan2/2`). For their semantics, see documentation to `math.h`.

6.4 Convenience

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

`true` ISO
: Always succeeds.

`otherwise`
: Same as `true/0`.

`fail` ISO
: Always fails.

`false` ISO
: Same as `fail/0`.

6.5 Negation and Control

`'!'/0` ISO
: 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.

`\+ +P` ISO
: If the goal `P` has a solution, fails, otherwise it succeeds. Equivalently, it is true iff `call(P)` (see Section 6.11) is false. Argument `P` must be ground for sound negation as failure, although no runtime checks are made.

Error Cases

`instantiation_error` `P` is not instantiated.

`type_error(callable,P)` `P` is not callable.

`fail_if(+P)`

`not +P`
: Like `\+/1` and provided for compatibility with legacy code. Compilation of `\+/1` and `fail_if/1` is optimized by XSB's compiler, while that of `not/1` is not

– therefore the first two syntactical forms are preferred in terms of efficiency, while `\+/1` is preferred in terms of portability.

All error cases are the same as `call/1` (see Section 6.11).

`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 3.8 see [68, 70]. Chapter 5 contains further discussion of the functionality of `tnot/1`.

Error Cases

- `P` is not ground (floundering occurs)
  - `instantiation_error`
- `P` is not callable
  - `type_error(callable,P)`
- `P` is not a call to a tabled predicate
  - `table_error`

`not_exists(+P)` Tabling

If `+P` is a tabled predicate, `not_exists/1` acts as `tnot/1` but permits variables in its subgoal argument The semantics in the case of unbound variables is as follows:

```
... :- ..., not_exists(p(X)), ...
```

is equivalent to

```
... :- ..., tnot(pp), ...
pp :- p(X).
```

where `pp` is a new proposition. Thus, the unbound variable X is treated as `tnot`($\exists$`X`(`p`(`X`))).

If `+P` is a non-tabled predicate `not_exists/1` ensures that `+P` is ground and called via a tabled predicate so that `not_exists/1` can be used with non-tabled predicates as well, regardless of whether `+P` is ground or not [8].

`not_exists/1` uses an auxiliary tabled predicate, `tunnumcall/1` in its execution. Therefore to reclaim space at the predicate or call level (e.g. using

[8] In previous versions of XSB, `not_exists/1` was called `sk_not/1`.

`abolish_table_pred/1` or similar predicates), `tunnumcall/1` must be explicitly abolished.

Error Cases

- `P` is not instantiated
  - `instantiation_error`
- `P` is not callable
  - `type_error(callable,P)`

`u_not(+P)` module: `tables`

If `P` is ground (or cyclic), `u_not(P)` is equivalent to `tnot(P)`; but `u_not/1` provides a different semantics than `tnot/1` or `not_exists/1` if `P` is non-ground. In this latter case, `u_not(P)` applies SLG delay to the goal `P`, explicitly indicating that the default negation of `P` is floundered. This action is safe because any answer that relies on *not P* will be undefined, rather than true or false. A current limitation of `u_not/1` is that while floundering correctly causes a literal to be delayed, no simplification is ever performed if the delayed literal ever becomes ground (see the example below). `u_not/1` thus provides an informationally sound but incomplete semantics for floundering.

Thus, the use of `tnot/1`, `not_exists/1`, or `u_not/1` depends on two conditions. `not_exists/1` is the only one of these predicates that allows `P` to be a non-tabled predicate. However as mentioned, their main difference is in handling non-ground negative subgoals. If an error should be thrown for a non-ground negative subgoal, `tnot/1` should be used; if it is semantically correct to skolemize if `P` is not ground, `not_exists/1` should be used; if it is semantically correct to treat the truth value of the negative subgoal as undefined, `u_not/1` should be used. From the perspective of performance, `tnot/1` is fastest followed by `u_not/1` and then `not_exists/1`.

The following examples should clarify the behavior of `u_not/1`. For the program fragment:

```
:- table p/1,q/1.
p(1):- u_not(q(X)).
q(1).
```

the goal `p(V)` returns

```
V = 1 undefined
```

Examining this answer shows the following:

```
| ?- get_residual(p(1),Res).

Res = [floundered(q(_h258))].
```

The program fragment

```
:- table r/1,q/1.
r(1):- u_not(q(X)),s(X).
q(1).
s(1).
```

shows a limitation in the current implementation of `u_not/1`. The goal `r(V)` returns

```
V = 1 undefined
```

as before. However, examining the answer shows

```
| ?- get_residual(r(1),Res

Res = [floundered(q(1))]
```

Note that the binding *X=1* is propagated to the delayed literal after the resulution of `s(X)`. However, the call `tnot(q(1))` is not made once X is bound, so that the delayed literal does not fail.

Error Cases are the same as for `tnot/1`.

`P -> Q ; R` ISO

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` ISO

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.
```

`between(+L,+U,B)` module: `basics`

For `L` and `U` integers, with `L` less than or equal to `U`, successive calls to `between/3` unify `B` with all integers between `L` and `U` inclusively. If `L` is less than `U` the predicate fails.

Error Cases:

- `L` (or `U`) is a not an integer
  - `type_error(integer,L)`

`(do_all +Goal)`

Defines a failure driven loop, as if defined by:

```
(do_all Goal) :- (Goal, fail ; true).
```

The control operator, `do_all/1` is defined as a prefix operator with precedence 1150.

`(+CGoal do_all +Goal)`

Defines a failure driven loop, as if defined by:

```
(CGoal do_all Goal) :-
            common_vars(CGoal,Goal,CommonVars),
            findall(CommonVars,CGoal,Vals),
            sort(Vals,UniqueVals),
            (basics:member(CommonVars,UniqueVals),
             call(Goal),
             fail
             ;
             true
            ).
```

where `common_vars/3` collects the variables that occur both in `P` and `Q`. The control operator, `do_all/2` is defined as an infix operator with precedence 1150.

6.6 Unification and Comparison of Terms

The predicates described in this section allow unification and comparison of terms [9].

Like most Prologs, default unification in XSB does not perform a so-called *occurs check* — it does not handle situations where a variable `X` may be bound to a structure containing `X` as a proper subterm. For instance, in the goal

```
X = f(X) % incorrect!
```

`X` is bound to `f(X)` creating a term that is either cyclic or infinite, depending on one's point of view. Prologs in general perform unification without occurs check since without occurs check unification is linear in the size of the largest term to be unified, while unification with occurs check may be exponential in the size of the largest term to be unified. Most Prolog programmers will rarely, need to concern themselves with cyclic terms or unification with occurs check. However, unification with occurs check can be important for certain applications, in particular when Prolog is used to implement theorem provers or sophisticated constraint handlers. As a result XSB provides an ISO-style implementation of the predicate `unify_with_occurs_check/2` described below, as well as a Prolog flag `unify_with_occurs_check` that changes the behavior of unification in XSB's engine.

As opposed to unification predicates, term comparison predicates described below 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:

- ordering of variables is based on their address within the SLG-WAM — the order is *not* related to the names of variables. Thus note that two variables are identical only if they share the same address – only if they have been unified or are the same variable to begin with. As a corollary, note that two anonymous variables will not have the same address and so will not be considered identical terms. As with most WAM-based Prologs, the order of variables may change as variables become bound to one another. If the order is expected to be invariant across variable bindings, other mechanisms, such as attributed variables, should be used.
- 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

[9] Arithmetic comparison predicates that may evaluate terms before comparing them are described in Section 6.3.1.

of their numerical values. If comparison is needed, a conversion should be performed (e.g. through `float/1`).

- atoms are put in alphabetical (i.e. UTF-8) 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 unification and comparison of arbitrary terms are:

`X = Y` ISO

Unifies `X` and `Y` without occur check.

`unify_with_occurs_check(One,Two)`

Unifies `One` and `Two` using an occur check, and failing if `One` is a proper subterm of `Two` or if `Two` is a proper subterm of `One`.

Example:

```
| ?- unify_with_occurs_check(f(1,X),f(1,a(X))).

no
| ?- unify_with_occurs_check(f(1,X),f(1,a(Y))).

X = a(_h165)
Y = _h165

yes
| ?- unify_with_occurs_check(f(1,a(X)),f(1,a(X))).

X = _h165

yes
```

`T1 == T2` ISO

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 goal:

```
| ?- 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.

`X \= Y` ISO

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

`T1 \== T2` ISO

Succeeds if the terms currently instantiating `T1` and `T2` are not literally identical.

`Term1 ?= Term2`

Succeeds if the equality of `Term1` and `Term2` can be compared safely, i.e. whether the result of `Term1 = Term2` can change due to further instantiation of either term. It is specified as by `?=(A,B) :- (A==B ; A B̄).`

`unifiable(X, Y, -Unifier)` module: `constraintLib`

If `X` and `Y` can unify, succeeds unifying `Unifier` with a list of terms of the form `Var = Value` representing a most general unifier of `X` and `Y`. `unifiable/3` can handle cyclic terms. Attributed variables are handles as normal variables. Associated hooks are not executed [10].

`T1 @< T2` ISO

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

`T1 @> T2` ISO

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

`T1 @=< T2` ISO

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

`T1 @>= T2` ISO

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

`T1 @= T2`

Succeeds if `T1` and `T2` are identical variables, or if the main structure symbols of `T1` and `T2` are identical.

[10] In Version 3.8, `unifiable/3` is written as a Prolog predicate and so is slower than many of the predicates in this section.

`compare(?Op, +T1, +T2)` ISO

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.

`ground(+X)` ISO

Succeeds if `X` is currently instantiated to a term that is completely bound (has no uninstantiated variables in it); otherwise it fails. While `ground/1` has no associated error conditions, it is not safe for cyclic terms: if cyclic terms may be an issue use `ground_or_cyclic/1`.

`ground_and_acyclic(+X)`

`ground_or_cyclic(+X)`

`ground_or_cyclic/1` succeeds if `X` is currently instantiated to a term that is completely bound (has no uninstantiated variables in it) *or* is a cyclic term; otherwise it fails. Alternately, `ground_and_acyclic/1` succeeds if `X` is currently instantiated to an acyclic term that is completely bound (has no uninstantiated variables in it). Neither predicate has no associated error conditions.

Both predicates are written to be as efficient as possible, and each requres a single traversal of a term, regardless of whether the term is ground, nonground or cyclic. However, due to the nature of checking for cyclicity, these predicates are somewhat slower than the unsafe `ground/1`.

`subsumes(?Term1, +Term2)` module: `subsumes`

Term subsumption is a sort of one-way unification. Term `Term1` and `Term2` unify if they have a common instance, and unification in Prolog instantiates both terms to that (most general) common instance. `Term1` subsumes `Term2` if `Term2` is already an instance of `Term1`. For our purposes, `Term2` is an instance of `Term1` if there is a substitution that leaves `Term2` unchanged and makes `Term1` identical to `Term2`. Predicate `subsumes/2` does not work as described if `Term1` and `Term2` share common variables.

`subsumes_chk(+Term1, +Term2)` module: `subsumes`

`subsumes_term(+Term1, +Term2)` ISO

The `subsumes_chk/2` predicate is true when `Term1` subsumes `Term2`; that is, when `Term2` is already an instance of `Term1`. This predicate simply checks for subsumption and does not bind any variables either in `Term1` or in `Term2`. `Term1` and `Term2` should not share any variables.

Examples:

```
| ?- subsumes_chk(a(X,f,Y,X),a(U,V,b,S)).

no
| ?- subsumes_chk(a(X,Y,X),a(b,b,b)).

X = _595884
Y = _595624
```

`variant(?Term1, ?Term2)` module: `subsumes`

This predicate is true when `Term1` and `Term2` are alphabetic variants. That is, you could imagine that `variant/2` as being defined like:

```
variant(Term1, Term2) :-
     subsumes_chk(Term1, Term2),
     subsumes_chk(Term2, Term1).
```

but the actual implementation of `variant/2` is considerably more efficient. However, in general, it does not work for terms that share variables; an assumption that holds for most (reasonable) uses of `variant/2`.

`check_variant(?Term1)` module: `tables`
`check_variant(+Term1,+DontCares)` module: `tables`

`check_variant/[1,2]` provide efficient means of checking whether the variant of a term has been asserted to a trie indexed predicate. A call `?- check_variant(Term)` thus succeeds if a variant of `Term` has been trie indexed and asseerted, and fails otherwise; the check performs no unification, and no backtracking is possible.

`check_variant/2` allows the user to specify that the last n arguments of `Term` are not to be checked for variance. This `check_variant(Term,N)` succeeds of there is a trie indexed term whose first $arity - n$ arguments are variants of those in term.

These predicates exploit the trie data structure to obtain their efficiency; as a result our implementation does not allow don't care arguments apart from the

final n arguments. ***More importantly, for efficiency, no check is made to determine whether a predicate has been trie-indexed.*** If unsure, the user should call `current_index/2`.

Example 6.6.1 `?- import check_variant/1 from tables.`

```
yes
 ?- index(cmp/3, trie).

yes
| ?- assert(cmp(a,b,c)),assertcmp(d,e,f)).

yes
| ?- check_variant(cmp(a,b,c)).

yes
| ?- check_variant(cmp(a,b,1)).

no
| ?- check_variant(cmp(a,b,X)).

no
| ?- check_variant(cmp(a,b,X),1).

X = _h183
```

Error Cases

`type_error` Argument 1 of `check_variant/[1,2]` is not a callable structure.

`type_error` Argument 2 of `check_variant/[2]` is not an integer

6.6.1 Sorting of Terms

Sorting routines compare and order terms without instantiating them. Users should be careful when comparing the value of uninstantiated variables. The actual order of uninstantiated variables may change in the course of program evaluation due to variable aliasing, garbage collection, or other reasons.

`sort(+L1, ?L2)` ISO

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.

`type_error` Argument 1 of `sort/2` is a non-variable, non-list term.

`keysort(+L1, ?L2)` ISO

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 $\mathcal{O}(n \log n)$ 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 in the presence of uninstantiated variables..

Example:

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

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

yes
```

Error Cases:

`instantiation_error` `L1` is a variable or is not a proper list.

`type_error` `L1` is a non-variable, non-list term.

`domain_error(key_value_pair,Element)` `L1` contains an element `Element` that is not of the form `Key-Value`.

`parsort(+L1, +SortSpec, +ElimDupl, ?L2)` module: `machine`

`parsort/4` is a very general sorting routine. The list `L1` may consist of elements of any form. `SortSpec` is the atom `asc`, the atom `desc`, or a list of terms of the form `asc(I)` or `desc(I)` where `I` is an integer indicating a sort argument position. The elements of list `L1` are sorted into order according to the sort specification. `asc` indicates ascending order based on the entire term; `desc` indicates descending order. For a sort specification that is a list, the individual elements indicate subfields of the source terms on which to sort. For example, a specification of `[asc(1)]` sorts the list in ascending order on the first subfields of the terms in the list. `[desc(1),asc(2)]` sorts into descending order on the first subfield and within equal first subfields into ascending order on the second subfield. The order is determined by the standard predicate `compare`. If `ElimDupl` is nonzero, merging of multiple occurring elements takes place (i.e., duplicate (whole) terms are eliminated in the output). If `ElimDupl` is zero, then no merging takes place. A `SortSpec` of `[]` is equivalent to "asc". The time to perform the sorting is $O(nlogn)$ where n is the length of list `L1`. The sorting of elements in `L1` is not guaranteed to be stable. `parsort/4` must be imported from module `machine`.

Example:

```
| ?- parsort([f(3,1),f(3,2),f(2,1),f(2,2),f(1,3),f(1,4),f(3,1)],
     [asc(1),desc(2)],1,L).

L = [f(1,4),f(1,3),f(2,2),f(2,1),f(3,2),f(3,1)];

no
```

Error Cases:

`instantiation_error` `L1` is a variable or not a proper list.

6.7 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)` ISO

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)` ISO

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)` ISO

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)` ISO

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

`float(?X)` ISO

float/1 Same as `real/1`. Succeeds if `X` is currently instantiated to a floating point number; otherwise it fails.

`real(?X)`

Succeeds if `X` is currently instantiated to a floating point number; otherwise it fails. This predicate is included for compatibility with earlier versions of XSB.

`number(?X)` ISO

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

`atomic(?X)` ISO

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)` ISO

Succeeds if `X` is currently instantiated to a compound term (with arity greater that zero), i.e. to a non-variable 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 previous versions.

`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)`
: Works as above, but also returns the length of that string in the second argument, which must be a variable.

`is_attv(+Term)`
: Succeeds is `Term` is an attributed variable, and fails otherwise.

`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
```

`is_number_atom(?X)`

: Succeeds if `X` is an atom (e.g. `'123'`) (as opposed to a number `123`) which can be converted to a numeric atom (integer or float) and fails otherwise. In particular, if `is_number_atom(X)` succeeds, then

```
| ?- atom_codes(X,Codes),number_codes(N,Codes).
```

will succeed.

`callable(?X)` ISO
: 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 – i.e. a HiLog term that is not a Prolog term; otherwise the predicate 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)` ISO

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.

Error Cases

`atom_or_variable` `Functor` is not an atom or variable.

`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)` ISO

Unifies `Argument` with the `Index`th argument of `Term`, where the index is taken to start at 1. In accordance with ISO semantics, `Index` must be instantiated to a non-negative integer, and `Term` to a compound term, otherwise an error is thrown as described below. If `Index` is `0` or a number greater than the arity of `Term`, the predicate 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
```

Error Cases

- `Index` is a variable
  - `instantiation_error`
- `Index` neither a variable nor an integer
  - `type_error(integer,Index)`
- `Index` is less than 0
  - `domain_error(not_less_than_zero,Index)`
- `Term` is a variable
  - `instantiation_error`
- `Term` neither a variable nor a compound term
  - `type_error(integer,Index)`

`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 =..  ?List` ISO

Given proper instantiation of the arguments, `=../2` (pronounced *univ*) succeeds when (1) `Term` unifies with a compound Prolog or HiLog term and `List` unifies with a list whose head is the functor of `Term` and whose tail is a list of the arguments of `Term`; or (2) when `Term` unifies with an atomic term and `List` unifies with a list whose only element is `Term`. More precisely,

- If initially `Term` is uninstantiated, then `List` 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` throws the appropriate error message.

Examples:

```
| ?- X - 1 =.. L.
```

```
X = _h112
L = [-,_h112,1]

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

| ?- T =.. [1,2].
++Error[XSB/Runtime/P]: [Type (1 in place of atomic)] in arg 2 of predicate =../2

| ?- T =.. [a+b,2].
++Error[XSB/Runtime/P]: [Type (a + b in place of atomic)] in arg 2 of predicate =../2

| ?- X =.. [foo|Y].
++Error[XSB/Runtime/P]: [Instantiation]  in arg 2 of predicate =../2
```

Error Cases

- `Term` is a variable and `List` is a variable, a partial list, a or a list whose head is a variable
  - `instantiation_error`
- `List` is neither a variable nor a non-empty list
  - `type_error(list, H)`
- `List` is a list whose head `H` is neither an atom nor a variable, and whose tail is not the empty list
  - `type_error(atomic, H)`
- `Term` is a variable and the tail of `List` has a length greater than XSB's maximum arity for terms (65535)
  - `representation_error(max_arity)`

`?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...
```

Error Cases

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

`copy_term(+Term, -Copy)` ISO

Makes a `Copy` of `Term` in which all variables have been replaced by brand new variables which occur nowhere else. Variable attributes are also copied. 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)
```

`copy_term_nat(+Term, -Copy)` module: `basics`

Behaves as `copy_term/2`, however it replaces attributed variables with non-attributed variables in the copy.[11]

`term_variables(+Term,-Variableust)` ISO

Collects the variables in `Term` into the list `VariableList`. The variables are in the order of their first occurrences in a depth-first traversal of `Term`.

`term_depth(+Term, -Depth)`

`term_depth/2` provides an efficient way to find the maximal depth of a term. Term depth is defined recursively as follows:

- The depth of a structure is defined as 1 + the maximal depth of any argument of that structure.
- The depth of an attributed variable is the depth of the attribute structure associated with that variable.
- The depth of a list `[H|T]` is defined as 1 + the maximal depth of `H` and `T`.
- The depth of any other element is 1.

[11] The name of this predicate was chosen for consistency with SWI Prolog, and stands for *copy_term no attributes*.

Note that according to this definition, the depth of the list `[a,b]` is 3, since the list is equivalent to the structure `.(a,.(b,[]))` whose depth is 3.

`term_depth/2` does not check for cyclic structures, so it must be ensured that `Term` is acyclic.

`term_size(+Term, -Size)`

`term_size/2` provides an efficient way to find the total number of constituents of a term. Term size is defined recursively as follows:

- The size of an attributed variable is 1 (the variable size) + the size of the attribute structure.
- The size of a non-compound term is 1.
- The size of a compound term is defined as 1 + the sum of the sizes of all arguments of that term.
- The size of a list `[H|T]` is defined as the size of the term `'.'(H,T)`.

`term_size/2` does not check for cyclic structures, so it must be ensured that `Term` is acyclic.

`intern_term(+Term,-InternedTerm)` module: `machine`

`intern_term` makes an "interned" version of its first argument and returns that term in its second argument. The terms are equal terms (i.e., `Term == InternedTerm` would succeed.) The interned term has all its ground subterms represented (uniquely) in a global space. Subterms that contain variables are not copied but remain on the heap. The interned representation of ground terms can save space and/or time in some situations. Note that already interned subterms or `Term` do not need to be traversed in this operation.

6.8 Cyclic Terms

6.8.1 Unification with and without Occurs Check

Cyclic terms are created when Prolog unifies two terms whose variables have not been standardized apart: for instance

$$X = f(X)$$

will produce the cyclic term f(f(f(f(f(f(...)))))) – in other words, a term with an "infinite" depth. Note that according to the mathematical definition of unification,

`X` should not unify with a term containing itself. There are two reasons why XSB (along with virtually all other Prologs) has this *default* behavior.

- The default unification algorithm, when it unifies a variable V with a term T, does not check for the occurrence of V in `T`, in other words it does not perform an *occurs check*. Unification without an occurs check is linear in the size of the smaller of the terms to be unified. Unification with occurs check is (essentially) linear in the size of the larger term. Since unification is often used to assign a value to a variable, it is important in a programming language that assignment be constant time, and not linear in the size of the term being assigned.
- Some programs purposefully construct cyclic terms: this occurs with various constraint libraries such as CHR. These libraries do not perform as expected when a mathematically correct unification algorithm is used.

XSB provides two mechanisms for overriding this default behavior for unification.

- First, there is a Prolog flag `unify_with_occurs_check` which when set to `on` ensures that all unification is mathematically correct. Care should be taken when using this flag, for the above two reasons.
- For more detailed usages, the ISO predicate `unify_with_occurs_check/2` can be used syntactically rather than Prolog's default unification operator `=/2`.

6.8.2 Cyclic Terms

Fortunately, the creation of cyclic terms is uncommon for most types of programming; even when cyclic terms arise they can often be avoided by the proper use of `copy_term/2` or other predicates. Nevertheless cyclic terms do arise when XSB is used for meta-programming or if XSB is used as the basis of a high-level knowledge representation language such as Flora-2 or Silk. It is important that XSB's behavior be *cycle-safe* in the sense that the creation of cyclic terms per se will not create infinite loops in XSB's tabling or XSB's built-ins. Like some other Prologs, XSB supports unification of cyclic terms. In addition, most predicates like `functor/3`, or `=../2` that either take non-compound terms or that do not require term traversal are cycle-safe. A few built-ins that require term-traversal are "safe" for cyclic terms. For instance writing in XSB is subject to a depth check, which terminates for cyclic terms. Most importantly, the XSB heap garbage collector is guarenteed to be safe for cyclic terms.

Variant tabling can also handle cyclic terms if the proper flags are set. These flags are `max_table_subgoal_depth` which determines the maximal "reasonable" depth of a subgoal; and `max_table_answer_depth`, `max_table_answer_list_depth` which determine the maximal "reasonable" depth for non-list terms or lists (respectively) in answers. These last two flags also determine a "reasonable" depth for interned tries. Each of these depth flags have an associated answer flag: `max_table_subgoal_action`, `max_table_answer_action` and `max_table_answer_list_action` respectively. The actions can be of three types: `error` which throws an error if a term with a certain depth is encountered as a tabled subgoal or answer (regardless of whether that term is tabled); `failure` which causes failure for these cases; and `fail_on_cycles` which fails on cyclic terms, and otherwise throws an error for a term of a certain depth [12].

While the above operations cycle-safe, cyclic terms can cause problems in XSB for built-ins or predicates that require term traversal. For instance the library predicates `length/2` and `append/2` currently go into infinite loops with cyclic terms; unless otherwise specified it is the user's responsibility to check library predicates (as opposed to standard built-ins) for acyclicity using `is_acyclic/1` or `is_cyclic/1`. In addition the following XSB built-ins are `not` cycle-safe:

- `bagof/3`, `copy_term/2`, `ground/1` `numbervars/[1,3,4]`, `setof/3`, `subsumes/2`, `subsumes_chk/2`, `term_depth/2`, `term_size/2`, term_to_atom/[2,3], `term_to_codes/[2,3]`, `term_variables/2`, `unifiable/2` and `variant/2` [13].
- Various table inspection built-ins based on `get_call/2` or similar routines (including `get_residual/2`).

Arguably, programs should not intentionally create cyclic terms, and the above flags, as well as the following predicates, can help debug when cyclic terms are created.

`is_cyclic(?X)`

Succeeds if `X` is a cyclic term.

`is_acyclic(?X)`

`acyclic_term(?X)` ISO

Succeeds if `X` is not a cyclic term.

[12] We hope to efficiently integrate cycle checking into XSB's subsumptive tabling in the reasonably near future.

[13] The predicate `ground_or_cyclic/1` is safe for cyclic terms.

6.9 Manipulation of Atomic Terms

This section lists some of XSB's standard predicates for manipulating atomic terms. See also in Volume 2, Section 1.5 for other library predicates. Section 7 for wildcard matching, and Section 8 for an interfae to the PCRE library.

`atom_codes(?Atom, ?CharCodeList)` ISO

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 UTF-8 codes representing the *"name"* of that atom. In that case, `CharList` is exactly the list of UTF-8 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 UTF-8 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.

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]).
++Error[XSB/Runtime/P]: [Instantiation]  in arg 2 of predicate atom_codes/2
Forward Continuation...

| ?- atom_codes(431, L).
++Error[XSB/Runtime/P]: [Type (431 in place of atom)] in arg 1 of predicate
```

```
atom_codes/2
Forward Continuation...

| ?- atom_codes(X, [52,300,49]).
[Representation (300 is not character code)] in arg 2 of predicate
atom_codes/2
Forward Continuation...
```

Error Cases

- `Atom` is a variable and `CharCodeList` is a partial list or a list with an element which is a variable
  - `instantiation_error`
- `Atom` is neither a variable nor an atom
  - `type_error(atom, Atom)`
- `Atom` is a variable and `CharCodeList` is neither a list nor a partial list
  - `type_error(list, CharCodeList)`
- `Atom` is a variable and an element `E` of `CharCodeList` is neither a variable nor a character code
  - `representation_error(character_code, E)`

`number_codes(?Number, ?CharCodeList)` ISO

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 UTF-8 (= 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 UTF-8 (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.

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).
++Error[XSB/Runtime/P]: [Type (foo in place of
  number)] in arg 1 of predicate
number_codes
Forward Continuation...
```

Error Cases

- `Number` is a variable and `CharCodeList` is a partial list or a list with an element which is a variable
  - `instantiation_error`
- `Number` is neither a variable nor a number
  - `type_error(number, Number)`
- `Number` is a variable and `CharCodeList` is neither a list nor a partial list
  - `type_error(list, CharCodeList)`
- `Number` is a variable and an element `E` of `CharCodeList` is neither a variable nor a character code
  - `representation_error(character_code, E)`

`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 UTF-8 codes representing the *"name"* of the constant. In that case, `CharList` is exactly the list of UTF-8 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 UTF-8 character codes. In that case, `name/2` will convert a list of UTF-8 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.

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[XSB/Runtime/P]: [Miscellaneous] Predicate name/2 for reals is not imple
Aborting...
```

- `Constant` is a variable and `CharCodeList` is a partial list or a list with an element which is a variable
  - `instantiation_error`
- `Constant` is neither a variable nor atomic
  - `type_error(atomic, Constant)`
- `Constant` is a variable and `CharCodeList` is neither a list nor a partial list
  - `type_error(list, CharCodeList)`
- `Constant` is a variable and an element `E` of `CharCodeList` is neither a variable nor a character code
  - `representation_error(character_code, E)`

`atom_chars(?Number, ?CharList)` ISO

Like `atom_codes/2`, but the list returned (or input) is a list of characters *as*

atoms rather than UTF-8 codes. For instance, `atom_chars(abc,X)` binds `X` to the list `[a,b,c]` Instead of `[97,98,99]`.

Error Cases

- `Atom` is a variable and `CharList` is a partial list or a list with an element which is a variable
  - `instantiation_error`
- `Atom` is neither a variable nor an atom
  - `type_error(atom, Atom)`
- `Atom` is a variable and `CharList` is neither a list nor a partial list
  - `type_error(list, CharList)`
- An element `E` of `CharList` is not a single-character atom
  - `type_error(character, E)`
- `Atom` is a variable and an element `E` of `CharCodeList` is not a single-character atom
  - `representation_error(character, E)`

`number_chars(?Number, ?CharList)` ISO

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

Error Cases

- `Number` is a variable and `CharList` is a partial list or a list with an element which is a variable
  - `instantiation_error`
- `Number` is neither a variable nor a number
  - `type_error(number, Number)`
- `Number` is a variable and `CharList` is neither a list nor a partial list
  - `type_error(list, CharList)`
- An element `E` of `CharList` is not a single-character atom
  - `type_error(character, E)`
- `CharList` is a list of single-character atoms but is not parsable as a number (by XSB)

- `syntax_error(CharList)`

`number_digits(?Number, ?DigitList)`

Like `number_codes/2`, but the list returned (or input) is a list of digits *as numbers* rather than UTF-8 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]`.

Error cases are the same as `number_chars/2`.

`char_code(?Character, ?Code)` ISO

The standard predicate `char_code/2` is true if `Code` is the current code for `Character`. In XSB it is defined as `atom_codes(Character,[Code])`.

`atom_length(+Atom1,?Length)` ISO

This standard predicate succeeds if `Length` unifies with the length of (the name of) `Atom`.

Example

```
|?- atom_length(trilobyte,L).

L = 9
```

Error Cases

- `Atom` is a variable
  - `instantiation_error`
- `Atom` is neither a variable nor an atom
  - `type_error(atom,Atom)`
- `Length` is neither a variable nor an integer
  - `type_error(integer,Length)`

`concat_atom(+AtomList,?Atom)` module: `string`

If `Atom` is a variable, then `AtomList` must be a list structure containing atoms, integers and/or floats. This predicate flattens `AtomList` and concatenates the atoms and integers into a single atom, returned in `Atom`. Integers and floats are converted to character strings using `number_codes/2`.

If `Atom` is an atom, then `AtomList` must be a list containing atoms, and/or variables. In this case `atom_codes` binds the variables in the list to atoms in

such a way that the atoms of `AtomList` concatenate to the atom `Atom`. For example, `concat_atom([X,abb,Y,cc],aabbabbdefcc)` will succeed twice, first binding X to `a` and Y to `abbdef`, and then binding X to `aabb` and Y to `def`.

This is a somewhat more general predicate than the ISO `atom_concat/2` described below, and can be more efficient if numerous atoms are to be concatenated together.

`concat_atom(+AtomList,+Sep,?Atom)` module: `string`

`AtomList` must be a list containing atoms, integers and/or floats, and `Sep` must be an atom. This predicate concatenates the atoms and integers into a single atom, separating each by `Sep`, return the resulting atom in `Atom`. Integers and floats are converted to character strings using `number_codes/2`.

This is a somewhat more general predicate than the ISO `atom_concat/2` described below, and can be more efficient if numerous atoms are to be concatenated together.

`atom_concat(Atom1,Atom2,Atom3)` ISO

- **Usage:** `atom_concat(?Atom,?Atom,+Atom)`
- **Usage:** `atom_concat(+Atom,+Atom,-Atom)`

Succeeds if `Atom12` is the concatenation of `Atom1` and `Atom2`.

Examples

```
| ?- atom_concat(hello,world,F).

F = hello world

| ?- atom_concat(X,Y,'hello world').

X =
Y = hello world;

X = h
Y = ello world
```

The last query will re-succeed for all combinations of atoms that produce `hello world`.

Error Cases

- `Atom1` and `Atom3` are both variables
  - `instantiation_error`
- `Atom2` and `Atom3` are both variables
  - `instantiation_error`
- `Atom1` is neither a variable nor an atom
  - `type_error(atom,Atom1)`
- `Atom2` is neither a variable nor an atom
  - `type_error(atom,Atom2)`
- `Atom3` is neither a variable nor an atom
  - `type_error(atom,Atom3)`

`sub_atom(+Atom,?LeftLength,?CenterLength,?RightLength,?CenterAtom` ISO

Succeeds if `Atom` can be broken into three pieces: A left atom of length `LeftLength`, a center atom `CenterAtom` of length `CenterLength` and a right atom of length `RightLength`. If sufficient arguments are uninstantiated to produce `CenterAtom` in non-deterministic starting positions, the predicate will backtrack through all center atoms for which the left atom length is the smallest , up to those whose left atom length is greatest (see examples below).

Examples

```
| ?- sub_atom(trilobyte,5,4,RL,CA).

RL = 0
CA = byte
| ?- sub_atom(trilobyte,1,CL,2,CA).

CL = 6
CA = riloby
| ?- sub_atom(trilobyte,LL,6,RL,riloby).

LL = 1
RL = 2
| ?- sub_atom(trilobyte,RL,4,LL,CA).

RL = 0
LL = 5
CA = tril;
```

```
RL = 1
LL = 4
CA = rilo;

RL = 2
CL = 3
CA = ilob
| ?- sub_atom(trilobyte,LL,CL,RL,CA).

LL = 0
CL = 0
RL = 9
CA = ;

LL = 0
CL = 1
RL = 8
CA = t;

LL = 0
CL = 2
RL = 7
CA = tr;

: /* after more backtracking */

LL = 0
CL = 9
RL = 0
CA = trilobyte;

LL = 1
CL = 0
RL = 8
CA = ;

Ll = 1
CL = 1
RL = 7
CA = r;
```

Error Cases

- `Atom` is a variable
  - `instantiation_error`
- `Atom` is neither a variable nor an atom
  - `type_error(atom, Atom)`
- `CenterAtom` is neither a variable nor an atom
  - `type_error(atom, CenterAtom)`
- `LeftLength` is neither a variable nor an integer
  - `type_error(integer, LeftLength)`
- `CenterLength` is neither a variable nor an integer
  - `type_error(integer, CenterLength)`
- `RightLength` is neither a variable nor an integer
  - `type_error(integer, RightLength)`
- `LeftLength` is an integer that is less than zero
  - `domain_error(not_less_than_zero, LeftLength)`
- `CenterLength` is an integer that is less than zero
  - `domain_error(not_less_than_zero, CenterLength)`
- `RightLength` is an integer that is less than zero
  - `domain_error(not_less_than_zero, RightLength)`

`string_substitute(+InpStr, +SubstrList, +SubstitutionList, -OutStr)` module: `string`

`InputStr` can an atom or a list of characters. `SubstrList` must be a list of terms of the form `s(BegOffset, EndOffset)`, where the name of the functor is immaterial. The meaning of the offsets is the same as for `substring/4`. (In particular, negative offsets represent offsets from the first character past the end of `String`.) Each such term specifies a substring (between `BegOffset` and `EndOffset`; negative `EndOffset` stands for the end of string) to be replaced. `SubstitutionList` must be a list of atoms or character lists.

Offsets start from 0, as in C/Java.

This predicate replaces the substrings specified in `SubstrList` with the corresponding strings from `SubstitutionList`. The result is returned in `OutStr`. `OutStr` is a list of characters, if so is `InputStr`; otherwise, it is an atom.

If `SubstitutionList` is shorter than `SubstrList` then the last string in `SubstitutionList` is used for substituting the extra substrings specified in `SubstitutionList`. As a special case, this makes it possible to replace all specified substrings with a single string.

As in the case of `re_substring/4`, if `OutStr` is an atom, it is not interned. The user should either intern this string or convert it into a list, as explained previously.

The `string_substitute/4` predicate always succeeds.

Here are some examples:

```
| ?- string_substitute('qaddf', [s(2,4)], ['123'] ,L).

L = qa123f

| ?- string_substitute('qaddf', [s(2,-1)], ['123'] ,L).

L = qa123

| ?- string_substitute("abcdefg", [s(4,-1)], ["123"],L).

L = [97,98,99,100,49,50,51]

| ?- string_substitute('1234567890123', [f(1,5),f(5,7),f(9,-2)], ["pppp", lll],X).

X = 1ppppllll89lll

| ?- string_substitute('1234567890123', [f(1,5),f(6,7),f(9,-2)], ['---'],X).

X = 1---6---89---
```

`term_to_atom(+Term,-Atom,+Options)` module: `string`

Converts `+Term` to an atomic form according to a list of write options, `Options`, that are similar to those used by `write_term/[2,3]`. The various options of `term_to_atom/[2,3]` are especially useful for the interface from C to XSB (see *Calling XSB from C* in Volume 2 of this manual).

- `quoted(+Bool)`. If `Bool = true`, then atoms and functors that can't be read back by `read/1` are quoted, if `Bool = false`, each atom and functor is written as its unquoted name. Default value is `false`.

- `ignore_ops(+Bool)`. If `Bool = true` each compound term is output in functional notation; list braces are ignored, as are all explicitly defined operators. If `Bool = canonical`, bracketed list notation is used. Default value is `canonical`. The corresponding value of `false`, that would enable operator precedence, is not yet implemented.
- `numbervars(+Bool)`. If `Bool = true`, a term of the form `'$VAR'(N)` where `N` is an integer, is output as a variable name consisting of a capital letter possibly followed by an integer. A term of the form `'$VAR'(Atom)` where `Atom` is an atom, is output as itself (without quotes). Finally, a term of the form `'$VAR'(String)` where `String` is a character string, is output as the atom corresponding to this character string. If `bool` is `false` this cases are not treated in any special way. Default value is `false`.

Error Cases

- `Options` is a variable
  - `instantiation_error`
- `Options` neither a variable nor a list
  - `type_error(list,Options)`
- `Options` contains a variable element, `O`
  - `instantiation_error`
- `Options` contains an element `O` that is neither a variable nor a write option.
  - `domain_error(write_option,O)`

Examples:

```
| ?- term_to_atom(f(a,1,X,['3cpio',d(3),'$VAR'("Foo")]),F,[]).

X = _h131
F = f(a,1,_h0,[3cpio,d(3),$VAR([70,111,111])])

yes
| ?- term_to_atom(f(a,1,X,['3cpio',d(3),'$VAR'("Foo")]),F,[numbervars(true)]).

X = _h131
F = f(a,1,_h0,[3cpio,d(3),Foo])

yes
| ?- term_to_atom(f(a,1,X,['3cpio',d(3),'$VAR'("Foo")]),F,[numbervars(true),quoted(true)]).

X = _h131
```

```
F = f(a,1,_h0,['3cpio',d(3),Foo])

yes
| ?- term_to_atom(f(a,1,X,['3cpio',d(3),'$VAR'("Foo")]),F,[numbervars(true),quoted(true),ignore_op

X = _h131
F = f(a,1,_h0,'.'('3cpio','.'(d(3),'.'(Foo,[]))))

yes
```

`term_to_atom(+Term,-Atom)` module: `string`

This predicate converts an arbitrary Prolog term `Term` into an atom, putting the result in `Atom`. It is defined using the default options for `term_to_atom/3`, e.g. `ignore_ops(canonical)`, `quoted(false)`, and `numbervars(false)`.

`term_to_codes(+Term,-CodeList,+OptionList)` module: `string`

This predicate is used in the definition of `term_to_atom/3` but only converts a term into a list of UTF-8 codes, and does not intern the list as an atom. Allowed values for `OptionList` and error cases are the same as in `term_to_atm/3`.

`term_to_codes(+Term,-CodeList)` module: `string`

This predicate converts a term to a list of UTF-8 codes. It is defined using the default options for `term_to_atom/3`, e.g. `ignore_ops(canonical)`, `quoted(false)`, and `numbervars(false)`.

`gc_atoms`

Explicitly invokes the garbage collector for atoms that are created, but no longer needed. By default, `gc_atoms/1` is called automatically, unless the Prolog_flag `atom_garbage_collection` is set to `false`, or if more than one thread is active. However there are reasons why a user may need to invoke atom table garbage collection. First, in Version 3.8, if atom table garbage collection is invoked automatically, it occurs periodically on heap garbage collection, or if numerous asserts and retracts have taken place. These heuristics overlook certain cases where numerous atoms may be created without invoking the garbage collector – e.g. through repeated uses of `format_write_string/3`. In addition if user-defined C code contains pointers to XSB's atom table, atom table garbage collection will be unsafe, as Version 3.8 of XSB does not detect such pointers in external code. In such cases, atom table garbage collection should be turned off via the Prolog flag `atom_garbage_collection`, and reinvoked at a point where the external pointers are no longer used.

6.10 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 ISO-standard predicates such as `setof/3`, `bagof/3`, and `findall/3` to 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 answer subsumption to produce a powerful aggregation facility as discussed in Section 5.4

`setof(?Template, +Goal, ?Set)` ISO

This predicate may be read as "`Set` is the set of all instances of `Template` 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 6.6). If there are uninstantiated variables in `Goal` which do not also appear in `Template`, 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).

Error cases are the same as predicate `call/1` (see Section 6.11).

Example: Consider the following predicate:

```
p(red,high,1).
p(green,low,2).
p(blue,high,3).
p(black,low,4).
p(black,high,5).
```

The goal `?- setof(Color,Height^Val^p(Color,Height,Val),List)` returns a single solution:

```
Color = _h73
Height = _h87
Val = _h101
L = [black,blue,green,red]
```

If `Height` is removed from the sequence of existential variables, so that the goal becomes:

```
?- setof(Color,Val^p(Color,Height,Val),List)
```

the first solution is:

```
Color = _h73
Val = _h87
Height = high
L = [black,blue,red];
```

upon backtracking, a second solution is produced:

```
Color = _h73
Val = _h87
Height = low
L = [black,green]
```

`bagof(?Template, +Goal, ?Bag)` ISO

This predicate has the same semantics as `setof/3` except that the third argument returns an unsorted list that may contain duplicates.

Error Cases are the same as predicate `call/1` (see Section 6.11).

Example: For the predicate `p/3` in the example for `setof/3`, the goal `?- bagof(Color,Height^Val^p(Color,Height,Val),L)` returns the single solution:

```
Color = _h73
Height = _h87
Val = _h101
L = [red,green,blue,black,black];
```

If `Height` is removed from the sequence of existential variables, so that the goal becomes: `?- bagof(Color,Val^p(Color,Height,Val),List)`, the first solution is:

```
Color = _h73
Val = _h87
Height = high
L = [red,blue,black];
```

upon backtracking, a second solution is produced:

```
Color = _h73
Val = _h87
Height = low
L = [green,black];
```

`findall(?Template, +Goal, ?List)` ISO

Similar to predicate `bagof/3`, except that variables in `Goal` that do not occur in `Template` are treated as existential, and alternative lists are not returned for different bindings of such variables. Note that this means that `Goal` should not contain existential 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.

Error cases are the same as `call/1` (see Section 6.11).

Example: For the predicate `p/3` in the example for `setof/3`, the goal `findall(Color,p(Color,Height,Val),L)` returns a single solution:

```
Color = _h73
Height = _h107
Val = _h121
F = [red,green,blue,black,black]
```

`findall(?Template, +Goal, ?List,?Tail)`

Acts as `findall/3`, but returns the result as the difference-list `Bag-Tail`. In fact, the 3-argument version is defined in terms of the 4-argument version:

```
findall(Templ, Goal, Bag) :- findall(Templ, Goal, Bag, [])
```

Error cases are the same as `findall/3` (or `call/1`).

`tfindall(?Template, +Goal, ?List)` Tabling

Like `findall/3`, `tfindall/3` treats all variables in `Goal` that do not occur in `Template` as existential. However, in `tfindall/3`, the `Goal` must be a call to a single tabled predicate.

`tfindall/3` allows `findall` functionality to be used safely with tabling by throwing an error if it is called recursively. Its use can be seen by considering the following series of programs.

```
p1(X):- findall(Y,p1(Y),X).
```

When executing the goal `p(X)`, XSB will throw an error when it reaches the maximum number of recursive invocations of findall.

Next, consider the program

```
:- table t/1.
t(X):- findall(Y,t(Y),X).
t(a).
```

The query `t(X)` will terminate without error, but will return two answers: `X = []` and `X = a`. These answers are hard to defend semantically, since there is an implicit domain closure axiom in findall-like predicates. On the other hand, for the program

```
:- table t2/1.
t2(X):- tfindall(Y,t2(Y),X).
t2(a).
```

the query `t2(X)` will throw a table error, indicating that a call to `tfindall/3` is apparently non-stratified footnoteDetection of non-stratification is based on the approximate detection of dependencies among subgoals maintained by XSB. This approximation is quite close for local evaluation, but is less close for batched evaluation.. Other behavior for tabled aggregation is provided by answer subsumption as discussed in Section 5.4

Other 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
```

Error cases include those of `findall/3` (see above), along with

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

`table_error` A call to `tfindall/3` is apparently non-stratified

`X ^ Goal` ISO

Within `setof/3`, `bagof/3` and the like, the `^ /2` operator means there exists an `X` such that `Goal` is true.

`excess_vars(+Term, +ExistVarTerm, +AddVarList, -VarList)` module: `setof`

Returns in `VarList` the list of (free) variables found in `Term` concatenated to the end of `AddVarList`. (In normal usage `AddVarList` is passed in as an empty list.) `ExistVarTerm` is a term containing variables assumed to be quantified in `Term`

so none of these variables are returned in the resulting list (unless they are in `AddVarList`.) Subterms of `Term` of the form `(VarTerm ^ SubTerm)` are treated specially: all variables in `VarTerm` are assumed to be quantified in `SubTerm`, and so no occurrence of these variables in `SubTerm` is collected into the resulting list.

Error Cases

`type_error AddVarList` is not a list of variables

`memory` Not enough memory to collect the variables.

`find_n(+N,?Template, +Goal, ?List)` module: `setof`

Acts as `findall/3` but returns only the first `N` bindings of `Template` to `List`.

6.11 Meta-Predicates

`call(#X)` ISO

If `X` is a non-variable 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.

Error Cases

`instantiation_error X` is a variable

`type_error(callable,X) X` is not callable.

`#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.

`call(Goal,Arg,...)` ISO

`call(Goal,Arg)` where `Goal` is an N-ary callable term first constructs a new N+1-ary term `NewGoal` with the same functor and first N arguments as `Goal` and with `Arg` as its N+1th argument, and then calls `NewGoal`. As an example,

```
call(member(X),[a,b,c])
```

is equivalent to `call(member(X,[a,b,c])`. `Goal` must be a callable term, but can be prepended by a module name using the `:/2` symbol. `call(Goal,Arg1,Arg2,...)` will act similarly. Note that `Goal` should usually be atomic – if the outer functor of `Goal` is, say, `,/2`, `call/[2-10]` will try to add the extra argument(s) to the comma functor, which is generally not the intended behavior.

While meta calls are generally fast in XSB, the extra term manipulation of `call/[2-10]` makes it somewhat slower than `call/1`.

`call_tv(#Goal,-TV)`

Calls `Goal` just as with `call/1`, and if `Goal` does not fail, instantiates `TV` with either `true` or `undefined`, depending on the truth value of `Goal` ***at the current stage of the evaluation.*** `Goal` need not be tabled itself. Note that `Goal` might succeed with truth value `undefined` before succeeding with truth value `true`.

Since `call_tv/2` is a meta-preducate that actually calls `Goal`, `call_tv/2` will have the same truth value as `Goal`. In other words, if `Goal` fails, `call_tv/2` will fail; if `Goal` succeeds unconditionally (is true in the well-founded semantics), `call_tv/2` will succeed unconditionally; and if `Goal` succeeds conditionally (is neither true nor false in the well-founded semantics) `call_tv/2` will itself succeed conditionally. An alternative approach is provided by `truth_value/2`.

Examples

The following example shows that `call_tv/2` propagates the truth value of `Goal`:

```
| ?-  call_tv(undefined,_TV),writeln(has_value(_TV)).
has_value(undefined)

undefined
```

The second example shows that `call_tv/2` shows the truth value of `Goal` alone, regardless of where in larger derivation `Goal` is called.

```
call_tv(undefined,_TV1),writeln(call1(_TV1)),call_tv(true,_TV2),writeln(call2(_TV2)).
call1(undefined)
```

```
call2(true)

undefined   << top-level conjunctive query is undefined.
```

Error cases are the same as `call/1`.

`truth_value(#Goal,?TruthValue)` module: `tables`

`truth_value(Goal,TruthValue)` succeeds only if `TruthValue` is the truth value of `Goal` in the well-founded model of the program. The predicate acts as follows.

1. `Goal` is executed;
   (a) If `Goal` is incomplete a permission error is thrown.
   (b) Otherwise if `Goal` is complete
      i. If `Goal` has no answers, `TruthValue` is unified with `false`.
      ii. Otherwise, `truth_value/2` backtracks through all answers for `Goal`, setting `TruthValue` to `true` or `undefined` as appropriate.

Examples Consider the program

```
:- table p/2.              :- table p/1.
p(a,b).                    p(F):- truth_value(p(_),F).
p(a,c):- undefined.
```

The goal `?- truth_value(p(X,Y),TV)` will succeed twice, setting `TV` once to `true` and once to `undefined`. On the other hand, the goal `?- truth_value(p(c,Y),TV)` will set `TV` to `false` since the goal `p(c,Y)` has no answers of any kind. The goal `?- truth_value(p(Y),TV)` will throw a permission error, since proving the goal relies on a call to `truth_value(p(Y),_)`.

Error Cases Error cases are the same as those for `call/1` with the addition

- `Goal` is incomplete after being called.
  - `permission_error(obtain_models_truth_value,incomplete_subgoal,Goal)`

`once(#X)` ISO

`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.

Error cases are the same as `call/1`.

`forall(Generate,Test)`

`forall(Generate, Test)` is true iff for all possible bindings of `Generate`, the goal `Test` is true. Procedurally, abstracting error checking, the predicate shall behave as being defined by `\+ (call(Generator), \+ call(Test))`.

Error cases are the same as `call/1`.

`table_once(#X)` Tabling

`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.

`call_cleanup(#Goal,#Handler)`

`call_cleanup(Goal, Cleanup)` calls `Goal` just as if it were called via `call/1`, but it is ensures that `Handler` will be called after `Goal` finishes execution. `call_cleanup/2` is thus useful when `Goal` uses a resource, (such as a stream, mutex, database cursor, etc.) that should be released when `Goal` finishes execution.

More precisely, `Goal` finishes execution either 1) by failure, 2) by determining that the success of `Goal` is deterministic, 3) when an error is thrown and not handled by `Goal` or one of its subgoals; or 4) when `Goal` is cut over. In all of these cases, `Handler` will be called and will succeed non-deterministically. We illustrate these cases through examples.

- Failure of `Goal`:

  ```
  ?- call_cleanup(fail,writeln(failed(Goal))).
  ```

  In this case, `Goal` has no solutions, and the handler is invoked when the engine backtracks out of `Goal`.

- Deterministic success of `Goal`. Assume that `p(1)` and `p(2)` have been asserted. Then

  ```
  ?- call_cleanup((p(X),writeln(got(p(X)))),writeln(handled(p(X)))).
  got(p(1))

  X = 1;
  got(p(2))
  handled(p(2))
  ```

```
X = 2;

no
```

Note that `Handler` is called only after the last solution of the goal `p(X)` has been obtained. XSB decides to call `Handler` only when it can be determined that the success of `Goal` has left no choice points. In such a case, the final solution has been obtained for `Goal`. Of course, it may be that a solution S to `Goal` leaves a choice point but the choice point will produce no further solutions for `Goal`. XSB will not call `Handler` in this case, rather it will wait until there are no choice points left for `Goal`.

- An uncaught error E is thrown out of `Goal`. In this case, `Handler` will be called, and then, if E is uncaught, E will be rethrown. This is illustrated in the following example (Error handling is discussed further in Section 12.3.2):

```
?- catch(call_cleanup(throw(my_error),writeln(invoking_handler)),
         Ball,
         write(Ball)).
invoking_handler
my_error
yes
```

Of course, `Handler` itself can be wrapped in a `catch/3` so that any errors will be caught by `call_cleanup/2`.

- Choice points for `Goal` are removed via a cut. Consider an example in which `p/1` has the same extension as above (`p(1),p(2)`:

```
call_cleanup(p(X),writeln(handled_1)),!.
handled_1

X = 1

yes
```

The handler is invoked immediately when the choice point laid down by `p(X)` is cut over – before returning to the command line. If a cut cuts over more than goal to be cleaned, more than one handler will be executed:

```
?-call_cleanup(p(X),writeln(handled_4_1)),
    call_cleanup(p(Y),writeln(handled_4_2)),
```

```
    call_cleanup(p(Z),writeln(handled_4_3)),
    !.
handled_4_3
handled_4_2
handled_4_1

X = 1
Y = 1
Z = 1
```

`call_cleanup/2` is thus an extremely powerful and flexible mechanism when used in a simple manner. While `Handler` is "guaranteed" to be invoked whenever `Goal` finishes execution [14], it may be difficult to predict when `Handler` will be invoked, as `Handler` may be invoked because of deeply non-local cuts over `Goal`, and even when such cuts are not present, the invocation depends on XSB determining when the last solution for `Goal` has been obtained. Baroque usages, such as invoking `call_cleanup/2` and cuts in the handler are supported, but may lead to code that is difficult to debug, since handlers may be invoked based on the state of XSB's choice point stack.

Error Cases

`Goal` is a variable

- `instantiation error`

`Goal` is neither a variable nor a callable term

- `type error(callable, Goal)`

`Handler` is a variable

- `instantiation error`

`Handler` is neither a variable nor a callable term

- `type error(callable, Handler)`

6.11.1 Timed Calls and Co-routining

When XSB is used in multi-threaded mode, one XSB thread t_1 may send a signal goal G to another thread t_2, which causes t_2 to interrupt its computation, execute G, and then resume its computation. (Section 7). `timed_call/2` and related predicates provide similar functionality within the single-threaded engine, a useful addition

[14] In fact we don't guarantee anything, see XSB's license.

since not all of tabling features are currently available in the multi-threaded engine. `timed_call/2` invokes a *base goal* which is interrupted either once or repeatedly at specified time intervals; when interrupted the engine invokes a handler that can implement a fairly general co-routine. In this way, monitors, logs, specialized debuggers – even adaptive behavior – can be implemented for large and complex tabled evaluations.

Within the timed call paradigm, it is important that the execution of the handler be independent of the base goal. For instance, any tables called by the handler should not depend on incomplete tables called by the base goal. On the other hand, the handler may inspect and analyze the evaluation state of the base goal through XSB's growing set of inspection routines (Section 10.3).

Interpreter Indicess

One use of timed calls is to have the handler suspend the base computation, and then start an interpreter that can be used to execute queries about the base computation. In the classic command-line interpreter of XSB, this is done by `break/0`. However, XSB can be called in a variety of ways other than the command-line-interpreter. It can be invoked as an executable via its `-e` option, called from a socket, called directly from C, or called from Java via Interprolog or some other bridge. To represent the generality of the interpreters that may be used, we make use of the notion of *interpreter indices* in this section and in Section 10.3, which discusses inspection predicates. Conceptually, a computation starts at index 0 – even if XSB was not invoked with a command-line interpreter. The handler executes at index 1, which might or might not be associated with a new command-line interpreter. In principle, an index N interpreter is invoked through the handler of a suspended goal that was executed at level $N-1$.

Timed Call and related predicates

`timed_call(#Goal,#Options)`

This predicate calls `Goal` and may interrupt `Goal` to call handlers as specified in `Options`. In the case where a handler succeeds, the execution of `Goal` will be continued; if the handler fails, `Goal` will fail; and if a handler throws an uncaught exception the execution of `Goal` may be aborted. In these ways `timed_call/2` can be used to allow co-routining of `Goal` with a repetition handler and/or timing `Goal` out with a (separate) handler.

`Options` is a list that may contain the following terms

- The term `max(+MaxInterval,#MaxHandler)` specifies that `MaxHandler` will be called when it is determined that the total elapsed time to execute `Goal` exceeds `MaxInterval` milliseconds. As a use case, if `MaxHandler` throws an exception, `Goal` can be aborted; and if `MaxHandler` fails, `Goal` will fail.
- The term `repeating(+RepInterval,#RepHandler)` specifies that `RepHandler` will be repeatedly called whenever `Goal` has executed an additional `RepInterval` milliseconds of elapsed time. The time taken to execute `RepHandler` is not counted as part of `RepInterval` milliseconds (or that of `MaxInterval`, if a maximum handler is also specified).
- The term `nesting` indicates that nested timed calls should be allowed within the same interpreter index. In this case, the nested timed call is simply treated as a call to `Goal`: in other words the interval(s) and handlers for the nested call are ignored. Otherwise, if `Options` does not contain the term `nesting` an attempt to nest calls will raise a permission exception.

Executing a timed call for `Goal` is more expensive than simply calling `Goal`, so it should not be used for frequent calls to goals that whose derivation is simple.

`timed_call/2` is based on XSB's internal interrupt mechanism, used for attributed variable handlers and thread signalling. As such, the ability to execute complex actions upon interrupt and then to resume `Goal` is very robust. However, checks for interrupts are only made whenever XSB's SLG-WAM engine is executing. Because of this, if XSB is suspended on I/O, calling a C or java function, in a C-implemented built-in, or otherwise outside of its virtual machine, the interrupt will not be executed until computation returns to XSB's virtual machine.

`timed_call/2` is integrated with XSB's break levels with a different timed call possible at each break level. In this way, a handler can call a break statement, and the base computation inspected using one of XSB's built-ins. As mentioned, the time that is spent in a break level is not counted as part of the repetition or maximum intervals associated with the base goal.

`timed_call/3` is not implemented for the multi-threaded engine but its functionality is easily duplicated using thread signalling (Section‘7.5).

Examples Consider the simple (and non-tabled) program fragment

```
loop :- loop.
```

which goes into an infinite loop on the query `?- loop.` However, the query

```
timed_call(loop,[max(5000,abort)]).
```

will interrupt `loop` and abort its computation after 5000 milliseconds. Alternately, the query

```
timed_call(loop,[max(5000,fail)]).
```

will fail the query. Finally, the query

```
timed_call(loop,[repeating(500,statistics)]).
```

will interrupt the computation every 500 milliseconds, print out statistics, and resume the computation where it left off.

These approaches can be combined:

```
timed_call(loop,[repeating(500,statistics),max(5000,break)]).
```

will interrupt the computation every 500 milliconds to print statistics, and then will enter a break after 5000 milliseconds, so that the state of computation can be explored after 5000 milliseconds. Handlers can be quite complex, and can support UI-based monitors, and even analysis routines that may modify the parameters of the computation when possible (e.g., by changing one form of tabling to another, when permitted).

Error Cases Error cases are the same as `call/1` for the first argument of `timed_call/3` and for handlers. In addition `timed_call/1` also throws these other errors.

`Options` is non-ground

- `instantiation_error`

`Options` is not a list

- `type_error`

`Interval` as contained in the first argument of `max/2` or `repeating/2` is not a positive integer

- `type_error`

`Options` contains neither a term of the form `max/2` nor of the form `repeating/2`

- `misc_error`

A call `C` to `timed_call/3` is made within the scope of some other call to `timed_call/3`

- `permission_error(nested_call,predicate,C)`

`timed_call/3` is called from the multi-threaded engine

- `misc_error`

`current_timed_call(?Index,?DisplayOptions)`

If there is an active timed call, invoked by `timed_call(Goal,Options)`, at interpreter index `Index`, this predicate returns information representing `Options`. `DisplayOptions` differs from `Options` only in the following case. If `Options` contains a term `max(Interval,Handler)`, `DisplayOptions` will contain a corresponding term `max(Used/Interval,Handler)`, indicating both the original interval (in milliseconds) and the number of milliseconds used so far.

Example

As above, assume that the following goal is called from the command-line interpreter:

```
?- timed_call(loop,[max(5000,abort)]).
```

After a few seconds, the user interrupts the goal with a Ctrl-C, sending XSB into a break level. At that point the goal

```
1:  ?- current_timed_call(Index,Options).
```

succeeds with `Index = 0` and `Options = [max(3539 / 5000,abort)]`.

`timed_call_modify(#NewOptions)`

When called from interpreter index $N > 1$, this predicate modifies the behavior of the suspended derivation.

- If current state is in the scope of the suspended goal `timed_call(Goal,OldOptions)` at interpreter index $N - 1$, `Goal` is made to behave as if it had been called with `timed_call(Goal,NewOptions)`. I.e., `NewOptions` rather than `OldOptions` takes effect as soon as the suspended goal is resumed. If `NewOptions` is the empty list, this has the effect of removing any interrupts that would be due to the timed call.
- If the current suspended state is *not* in the scope of a timed call as described above, the *top-level* goal G_T of interpreter level $N - 1$ is made to behave as if it had been called as `timed_call(`G_T`,NewOptions)`.

Example This rather fanciful example shows the essential points about how `timed_call_modify/1` can be used in practice. Suppose a user sets up a monitor for the infinite loop program (introduced above) using the goal:

```
timed_call(loop,[repeating(500,writeln(interruption_interval(500)))]).
```

which produces the output

```
interruption_interval(500)
interruption_interval(500)
interruption_interval(500)
:
```

At this point, the user realizes that too much information is being printed out, and decides to back off somewhat. The user obtains a break level by executing Ctrl-C as before, and calls the goal:

```
1: ?- timed_call_modify([repeating(1000,writeln(new_interruption_interval(1000)))]).
```

When the break is exited, information is presented in an undoubtedly more useful manner:

```
1: ?- [ End break (level 1) ]
new_interruption_interval(1000)
new_interruption_interval(1000)
new_interruption_interval(1000)
```

Additions and deletions of timed call parameters is done in a similar manner.

Error Cases Error cases are the same those for the options list of `timed_call/2` and for handlers. In addition `timed_call/1` also throws the following error.

- `timed_call_modify/1` is called in the top-level interpreter.
  - `permissions_error`

`timed_call_cancel`

When called at interpreter index N removes any handlers for a timed call invoked at level $N-1$. That is, if `Goal` was called at interpreter level N via `timed_call/2`, cancelling will remove any repeating or maximum interrupts. When interpreter level $N-1$ is later resumed, `Goal` will continue execution as if it were called via the normall calling mechanism. If there is no timed call active at level $N-1$, the predicate succeeds with no effects.

`bounded_call(#Goal,+MaxMemory,+MaxCPU,#Handler)` module: `standard`

`bounded_call(#Goal,+MaxMemory,+MaxCPU)` module: `standard`

These predicates call `Goal` and check once per second whether the total CPU time to execute `Goal` is greater than `MaxCPU` seconds, and whether the total memory taken by XSB is greater than `MaxMemory` bytes. Under `bounded_call/4` if either of these conditions arise, `Handler` is called; under `bounded_call/3` a resource exception is thrown for memory or CPU time.

These predicates are implemented directly using `timed_call/3` and inherit the advantages and limitations of that predicate. As an advantage, the ability to execute complex actions upon interrupt and to resume is very robust. However, checks for interrupts are only made whenever XSB's SLG-WAM engine is executing. Because of this, if XSB is suspended on I/O, calling a C or java function, in a C-implemented built-in, or otherwise outside of its virtual machine, the interrupt will not be executed until computation is back within XSB's virtual machine.

`Handler` cannot cause `timed_call/3` to be executed as a subgoal; but otherwise `Handler` has no restrictions.

`bounded_call/[3,4]` is not yet implemented for the multi-threaded engine but its functionality is easily duplicated using thread signalling (Section‘7.5).

Error Cases Error cases are the same as in `call/1` for the first argument of `bounded_call/3`, and are the same as that of `timed_call` for `Handler`.

`MaxCPU` or `MaxMemory` is not an integer

- `type_error(integer)`

`MaxCPU` or `MaxMemory` is not a positive integer

- `domain_error(positive_integer)`

6.12 Information about the System State

Various aspects of the state of an instance of XSB — information about what predicates, modules, or dynamic clauses have been loaded, their object files, along with other information can 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)` ISO

Succeeds iff stream `Stream` is the current input stream, or procedurally unifies `Stream` with the current input stream.

Error Cases

- `Stream` is neither a variable nor a stream identifier
  - `domain_error(stream_or_variable,Stream))`

`current_output(?Stream)` ISO

current_output/1 Succeeds iff stream `Stream` is the current output stream, or procedurally unifies `Stream` with the current output stream.

Error Cases

- `Stream` is neither a variable nor a stream identifier
  - `domain_error(stream_or_variable,Stream))`

ISO Compatibility Note: In XSB `current_input/1` does not throw an error if `Stream` is not a current input stream, but quietly fails instead.

`current_prolog_flag(?Flag_Name, ?Value)` ISO

`current_prolog_flag/2` allows the user to examine both dynamic aspects of XSB along with certain non-changeable ISO flags and non-changeable Prolog-commons flags. Calls to `current_prolog_flag/2` will unify against ISO, Prolog-commons, and XSB-specific flags.

ISO and Prolog-commons flags are as follows:

- `bounded` Indicates whether integers in XSB are bounded. This flag always has the value `true`
- `min_integer, max_integer` The minimum and maximum integers available in the current XSB configuration (differs between 32- and 64-bits).
- `max_arity` Indicates the maximum arity of terms in XSB. This flag always has the value `65535` (2^{16}
- `integer_rounding_function` This flag always has the value `toward_zero`
- `debug` Indicates whether trace or debugging is turned `on` or `off`
- `unknown` Indicates the behavior to be taken when calling an unknown predicate. The value can be set to `error`, `fail`, `warning`, `unknown` or `user_hook`. The default setting is `error`.

  The first three values respectively indicate that calls to unknown predicates should throw an existence error, fail, or produce a warning message to the `user_warning` stream and then throw an existence error.

  The value is `undefined` then a call G to an unknown predicate succeeds with value *undefined*, and the delay literal `wcs_undefined(G)` is added to the delay list.[15]

  The value is `user_hook` allows a hook to be user-specified. The hook must be specified as follows. In `usermod` the fact

  `unknown_predicate_hook(Goal)`

  should be asserted, where `Goal`= $Predicate(Arg)$. When handling a call of the form G_1, where G_1 refers to an unknown predicate, `Goal` will be unified with G_1 and then $Goal\theta_{G_1}$ will be called.

  Example

  Suppose the following code has been compiled during an XSB session.

```
:- import misc_error/1 from error_handler.

my_unknown_predicate_hook(Goal):-
```

[15] This action is part of a semantics for Prolog sometimes called the *Weak Completion Semantics.*

```
        writeln(this_is_my_undefined_warning_about(Goal)),
        misc_error(unknown_predicate).
```

and the following fact asserted into usermod.

```
unknown_predicate_hook(my_unknown_predicate_hook(_X)
```

Then XSB will have the following behavior when calling the following unknown predicate:

```
| ?- foo(X).
this_is_my_undefined_warning_about(foo(A))
++Error[XSB/Runtime/P]: [Miscellaneous] unknown_predicate
Forward Continuation...
:
```

- `double_quotes` Indicates that double-quoted terms in XSB represent lists of character codes. Value is `codes`
- `dialect` indicates the implementation of Prolog that is running. Using this flag, applications intended to run on more than one Prolog can take actions that conditional on the executing Prolog. The value is `xsb`.
- `version_data` indicates the version of XSB that is running. Using this flag, applications intended to run on more than one Prolog can take actions that conditional on the executing Prolog. The value is `xsb(`$\langle Major\_version\# \rangle$`,`$\langle Minor\_version\# \rangle$`,`$\langle Patch\_version\# \rangle$`,_)`.

ISO Compatibility Note: The ISO flags `char_conversion` is not available – XSB does not use character conversion. XSB reads double quoted strings as lists of character codes, so that the value of the flag `double_quotes` is always `codes`, and this flag is not settable.

Non-standard flag names may be specific to XSB or may be common to XSB and certain other Prolog. These flag names are:

- `backtrace_on_error` The flag is `on` iff system-handled errors automatically print out the trace of the execution stack where the error arose, `off` otherwise. Default is `on`. In the multi-threaded engine, this flag is thread-specific and controls whether the backtrace for a current execution will be printed to `STDERR`.
- `dcg_style` the DCG style currently used; `xsb` or `standard` (standard is used in Quintus, SICSTUS, etc.). See Section 11.4 for more details. Default is `xsb`. This flag affects all threads in the process.

- `heap_garbage_collection` Values: `indirection` or `none`. Indicates the heap garbage collection strategy that is currently being employed (see also Section 3.7). Default is `indirection`. This flag is private to each thread.
- `heap_margin` Specifies the size *in bytes* of the margin used to determine whether to perform heap garbage collection or reallocation of the environment stack. The default is 8192 (8K) bytes for 32-bit platforms 16384 (16K) for 64-bit platforms. Setting this field to a large value (e.g. in the megabyte range) can cause XSB to be more aggressive in terms of expanding heap and local stack and to do fewer heap garbage collections than with the default value. However `heap_margin` should never be set *lower* than its default, as this may prevent XSB from properly creating large terms on the heap.
- `clause_garbage_collection` Values: `on` if garbage collection for retracted clauses is allowed, and off otherwise. Default is `on`. This flag is private to each thread.
- `atom_garbage_collection` Values: `on` if garbage collection for atomic constants is allowed, and off otherwise. Default is `on`. This flag is global for all threads (currently, string garbage collection will only be invoked if there is a single active thread.)
- `table_gc_action` The setting `abolish_tables_transitively` causes predicates or subgoals that depend on a conditional answer of an abolished table to be abolished automatically; the setting `abolish_tables_singly` not does not cause this action. The distinction is important, since if table T_1 depends on table T_2, and T_2 is abolished but T_1 is not, then predicates that introspect the dependencies of T_1 could cause memory violations (e.g., `get_residual/2`). Default is `abolish_tables_transitively`. This flag affects all threads in the process.
- `goal` The goal passed to XSB on command line with the '-e' switch; or '`true.`' if nothing is passed. This flag may be examined, but not set.
- `tracing` Values: `on` iff trace mode is on; `off` otherwise. This flag affects all threads in the process.
- `write_depth` The depth to which a term is written by `write`-like predicates. Default is 64. This flag affects all threads in the process.
- `warning_action` The action to take on warnings: the default value `print_warning` prints a warning message to the `user_warning` stream when `warning/1` is called; `silent_warning` silently succeeds when `warning/1` is called; and `error_warning/1` throws a miscellaneous exception.

- `write_attributes` Determines the action to take by `write/1` when it writes an attributed variable. By default *write/1* portrays attributed variables using module-specific routines (cf. Volume 2 of this manual) as $Variable\{Module : PA\_Output\}$ where PA_Output is the output of the `portray_attrubutes/2` clause for $Module$. However the value `ignore` causes an attributed variable to be written simply as a variable; and `dots` causes $Variable\{< module\_name >: ...\}$ to be written. Finally, the value `write` causes a variables attribute to be written as a term [16]. The default behavior is set to the value `portray`.
- `unify_with_occurs_check` If set to `on`, perform all unification using an occurs check, which makes unification mathematically correct, at the cost of increasing its computational complexity. Without the occurs check, the unification

  $$X = f(X)$$

  will produce a cyclic term `X = f(f(f(f(...))))`; with the occurs check this unification will fail. Setting the flag to `on` may slow down programs, perhaps drastically, and may be incompatible with some constraint libraries such as CHR. An alternate to this flag is the ISO predicate `unify_with_occurs_check/2`: see Section 6.8 for further discussion. The default for this flag is `off`.
- `character_set` If set to `utf_8`, interprets input/output byte sequences as UTF-8 encodings of unicode code points; if set to `cp1252` then interprets bytes using the Windows Code Page 1252; if set to `latin_1`, then input/output bytes are interpreted as directly representing unicode code points. Default for UNIX-style systems is `utf_8` and for Windows-style systems is `cp1252`, but the flag (and character sets) may be changed at any time. (See section "Character Sets in XSB" in XSB User Manual Volume 2 for more details.)
- `errors_with_position` If set to on, then the Prolog `read` predicates, when they encounter a syntax error in the term being read, will throw a syntax error which contains a pair `ErrorMessage-ErrorPosition`. `ErrorPosition` is an integer indicating the position in the file at which the syntax error was detected. If set to off, then the read predicates will simply throw the syntax error message.
- `exception_action` If set to `iso` then ISO-style exceptions will be thrown whenever an error condition arises. However, if `exception_action` is set

[16] When writing an attribute, any attributed variables in the attribute itself are written just as variables with their attributes ignored.

to `undefined_truth_value` then certain goals will succeed with an undefined truth value rather than throwing an error. When this occurs, a literal is added to the delay list of the current evaluation. Later, it can be determined whether an undefined answer depends on an exceptional condition through `explain_u_val/[3.6]`, `get_residual_sccs/[3,5]` or via a justification system that depends on these predicates. The default for this flag is `iso`

- `exception_pre_action` If set to `print_incomplete_tables`, then the incomplete subgoals are printed *before* throwing an exception. The execution of this action causes the stack of incomplete tables to be printed to a temporary file in `$XSBDIR/etc`. The file can be obtained via the predicate `get_scc_dumpfile/1`; later, information in the file can be used to help understand the context in which the exception arose. The file will be created only if an exception is thrown over at least one incomplete table. The default for this flag is `off`.

  Use of this flag may be seen as an aid to analyzing tabling behavior when XSB is part of a running system; for interactive analysis inspection predicates may be more useful (cf. Section 10.3).
- `max_tab_usage` If set to `on`, maintains the maximal table usage (in bytes) for display in `statistics/[0,1]`. This information can be useful if a program performs various types of table abolishes. Setting this flag to `on` may slightly slow down computation. Defailt is `off`.

Tripwire Flags The following flags that pertain to tripwires (cf. Setion 10.3.4) are not currently implemented in the multi-threaded engine. Each tripwire has one flag that sets a limit on some aspect of derivation along with an action of what to do in such a case.

- `max_table_subgoal_size` A limit set on the size of a subgoal argument that can be added to a table: if the limit is reached, an action is taken as indicated in the following flag. To understand the use of this flag, consider that if a predicate such as

  ```
  p(X):- p(f(X)).
  ```

  is tabled, it can create subgoals of unbounded size. When the limit is set to 0, this tripwire is disabled. The default value is `0`.
- `max_table_subgoal_size_action` The action to take whenever a tabled subgoal of limit size is encountered. When the maximum subgoal size is reached, XSB can

1. Throw a miscellaneous error, set using the value `error`. This is the default action.
2. Apply subgoal abstraction, using the value `abstract`.
3. Suspend the computation and throw it into a break-level CLI, using the value `suspend`

- `max_incomplete_subgoals` A limit set on the maximum number of tabled subgoals that can be incomplete at one time. If the limit is reached, an action is taken as indicated in the following flag. Note that subgoals are usually completed during the course of a derivation, so a large number of incomplete subgoals may indicate unfounded recursion or some other mis-specification in a program. When the limit is set to 0, this tripwire is disabled. The default value is `0`.
- `max_incomplete_subgoals_action` The action to take whenever the limit number of incomplete subgoals is encountered. XSB can
  1. Throw a miscellaneous error, set using the value `error`. This is the default action.
  2. Suspend the computation and throw it into a break-level CLI, using the value `suspend`.
- `max_sccs_subgoals` A limit set on the maximum number of incomplete tabled subgoals *that are mutually recursive*. If the limit is reached, an action is taken as indicated in the following flag. Note that a large number of mutually recursive subgoals may indicate a mis-specification in a program, such as an unintended expansion of the search space via meta-predicates or HiLog. When the limit is set to 0, this tripwire is disabled. The default value is `0`.
- `max_sccs_subgoals_action` The action to take whenever the limit number of incomplete subgoals within a single SCC is encountered. XSB can
  1. Throw a miscellaneous error, set using the value `error`. This is the default action.
  2. Suspend the computation and throw it into a break-level CLI, using the value `suspend`.
- `max_table_answer_size` A limit set on the size of an answer argument that can be added to a table: if the limit is reached, an action is taken as indicated in the following flag. To understand the use of this flag, consider the program fragment:

```
:- table p/1.
p(f(X)):- p(X).          p(a).
```

is tabled, the model for the goal `?- p(X)` is infinite, so that this program will not terminate. When the size is set to 0, this this tripwire is disabled. The default value is `0`.

- `max_table_answer_size_action` The action to take when a tabled answer of maximum size is encountered. When the maximum answer size is reached, XSB can
  1. Throw a miscellaneous error, set using the value `error`. This is the default action.
  2. Apply answer abstraction through radial restraint, using the value `abstract`.
  3. Suspend the computation and throw it into a break-level CLI, using the value `suspend`
- `max_answers_for_subgoal` A limit set on the number of answers that any single tabled subgoal should have: if the limit is reached, an action is taken as indicated in the following flag. Note that in a program with a large number of constant or functor symbols, it is possible to construct many answers of a fixed size; and if too many such answers are added for a given subgoal, it may indicate a program mis-specification. When the size is set to 0, this tripwire is disabled. The default value is `0`.
- `max_answers_for_subgoal_action` The action to take when a the number of answers for a given subgoal exceeds the limit set in the previous flag. XSB can
  1. Throw a miscellaneous error, set using the value `error`. This is the default action.
  2. Suspend the computation and throw it into a break-level CLI, using the value `suspend`
- `max_memory` The maximum amount of memory that an XSB thread (in the single-threaded engine) or all XSB threads (in the multi threaded engine) can use for their combined execution stacks, program space, tables, or any other purpose. If a query exceeds this amount, XSB will abort the query with a resource exception and then try to reclaim space used by the query. As with other flags, this flag can be set during an XSB session.

  The maximum amount can be set in two ways. If given a floating point number $F, 0 \leq F \leq 1$, the maximum will be set to F times the total amount of RAM for the machine on which XSB is executing. If given an integer I, the maximum will be set to I kilobytes. The value of 0 effectively disables the flag, allowing XSB to allocate as much memory as

the underlying OS will grant. The default value is 0, so that the flag is disabled by default.

Flags Pertaining to Multi-Threading The following flags affect only the multi-threaded engine.

- `thread_glsize` In the multi-threaded engine, the initial size, in kbytes, of the global and local stack area of a newly created thread if no such option is explicitly passed. By default this is 768 (or 1536 for 64-bit configurations), or whatever was passed in if the command-line option `-m` was used, but that value may be modified at any time by resetting the flag. This flag affects a thread created by any thread in the process.
- `thread_tcpsize` In the multi-threaded engine, the initial size, in kbytes, of the trail and choice point area of a newly created thread if no such option is explicitly passed. By default this is 768 (or 1536 for 64-bit configurations), or whatever was passed in if the command-line option `-c` was used, but that value may be modified at any time by resetting the flag. This flag affects a thread created by any thread in the process.
- `thread_complsize` In the multi-threaded engine, the initial size, in kbytes, of the completion stack area of a newly created thread if no such option is explicitly passed. By default this is 64 (or 128 for 64-bit configurations), or whatever was passed in if the command-line option `-0` was used, but that value may be modified at any time by resetting the flag. This flag affects a thread created by any thread in the process.
- `thread_pdlsize` In the multi-threaded engine, the initial size, in kbytes, of the unification stack area of a newly created thread if no such option is explicitly passed. By default this is 64 (or 128 for 64-bit configurations), or whatever was passed in if the command-line option `-m` was used, but that value may be modified at any time by resetting the flag. This flag affects a thread created by any thread in the process.
- `thread_detached` In the multi-threaded engine, this specifies whether threads are to be created as detached or joinable if no explicit option is passed. A value of `true` indicates that threads are to be created as detached, and `false` as joinable. If this flag is not set, its default is `false`.
- `max_threads` In the multi-threaded engine, the maximum number of valid threads. By default this is 1024 and this value may not be reset at runtime, but it may be set by the command-line option `-max_threads`. This option

is settable only by a command-line argument, and has no effect in the single-threaded engine.

- `max_queue_size` In the multi-threaded engine, the default maximum number of terms a message queue contains before writes to the message queue block. By default this is 1000. If set to 0, queues by default will be unbounded. This option has no effect in the single-threaded engine.
- `shared_predicates` In the multi-threaded engine, indicates whether predicates are considered thread-shared by default – that is, whether tables or dynamic predicates are shared among threads. By default this is false, and predicates are considered thread-private by default. This option is settable only by a command-line argument, and has no effect in the single-threaded engine.

. Note that the above non-ISO flags are used only for dynamic XSB settings, *i.e.*, settings that might change between sessions (via command line arguments) or within the same session (via modifiable flags). For static configuration information, the predicate `xsb_configuration/2` should be used.

Error Cases

- `Flag_Name` is neither a variable nor an atom.
  - `domain_error(atom_or_variable,Flag_Name)`

`set_prolog_flag(?Flag_Name, ?Value)` ISO

`set_prolog_flag/2` allows the user to change settable prolog flags. Currently the only settable ISO flag is the `unknown` flag. Setting the flag `unknown` to `fail` results in calls to undefined predicates to quietly fail. Setting it to `warning` causes calls to undefined predicates to generate a warning (to `STDWARN`) and then fail. Setting it to `error` (the default) causes calls to undefined predicates to throw an existence error.

Dynamic XSB settings can also be changed, as described in `current_prolog_flag/2`.

Error Cases

- `Flag_Name` or `Value` is a variable.
  - `instantiation_error`
- `Flag_Name` is not the name of a recognized Prolog flag.
  - `domain_error(prolog_flag,Flag_Name)`

`current_predicate(?Predicate_Indicator)` ISO

`current_predicate/1` can be used to backtrack through indicators for loaded user or system predicates. If `Predicate_Indicator` unifies with `Module:F/A` all loaded predicates unifying with this indicator is returned. If `Predicate_indicator` is `F/A`, `current_predicate/1` behaves as if it were called with the form `usermod:F/A`. Unlike `current_functor/1` `current_predicate/1` does not return indicators for predicates that have been imported but not actually loaded into code space. For more detailed analysis of predicate properties, the predicate `predicate_property/2` can be used.

As an example 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, 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 `usermod`, use:

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

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

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

Error Cases

- `Predicate_indicator` is neither a variable nor a predicate indicator
  - `type_error(predicate_indicator,Predicate_indicator))`

ISO Compatibility Note: In XSB, `current_predicate` will backtrack through system predicates as well as user predicates.

`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. For more detailed analysis of module properties, the predicate `module_property/2` can be used.

Note that predicate `current_module/1` succeeds for a given module even if that module 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 end with the object file extension for the system (by default, `.xwam`). It is possible for a current module to have no associated file name (as is the case for `"usermod"`), 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 3.6). 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_functor(?Predicate_Indicator)`

`current_predicate/1` can be used to backtrack through indicators for all non-atomic terms occurring in loaded modules. If `Predicate_Indicator` unifies with `Module:F/A` all term indicators unifying with `F/A` in a module unifying with `Module` are returned. If `Predicate_indicator` is `F/A`, `current_predicate/1` behaves as if it were called with the form `usermod:F/A`. Unlike `current_predicate/1` `current_functor/1` returns not only structures occurring in predicates but predicates that are imported into loaded modules but are not yet themselves loaded.

As an example, 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.
```

There are no error conditions associated with this predicate; if its argument is not a predicate indicator the predicate simply fails.

`current_index(Functor/Arity,IndexSpec)`

XSB has a variety of ways to index dynamic predicate including alternate argument indexing, multiple argument indexing, star-indexing, and tries, as discussed in Section 6.14. In addition XSB allows a choice of which argument to

index for compiled predicates as well. `current_index/2` returns the index specification for each functor/arity pair unifying with `Functor/Arity` and visible from the calling context of `current_index/2`.

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

`predicate_property(?Term_Indicator, ?Property)`
: The standard predicate `predicate_property/2` can be used to find the properties of any predicate that 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 not a variable, and is a structure or atom, then `Property` is successively unified with the various properties associated with `Term_Indicator`. If `Term_Indicator` is not a known to the system, the call succeeds with `Property` successively unified to `exported` and `unclassified`. These properties can be considered as a default for any structure or atom.
  - If `Property` is bound to a valid predicate property, then `predicate_property/2` successively unifies `Term_Indicator` with the skeletal specifications of all predicates known to the system 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 not a term indicator, or if `Property` is not a valid predicate property, the call 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. Double lines show property categories, and a predicate can have at most one property of each category.

- *Execition Type* which is one of
  - 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 `usermod` 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 `usermod`.
  - `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.
- *Visibility Type* which can be one of
  - `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`.
- *Tabling Call Behavior* which can be one of
  - `tabled(variant)` The predicate has been declared tabled and to use call variance.
  - `tabled(subsumptive)` The predicate has been declared tabled and to use call subsumption
  - `tabled(default)` The predicate has been declared tabled and to use the default tabling strategy of the session, which can be either call variance or call subsumption.
- *Incremental Tabling Behavior* which can be one of

  - `incremental` The predicate was declared as either incremental dynamic or as incremental tabled; or
  - `opaque` The predicate was declared as opaque to incremental updates.
- `spied` The predicate symbol has been declared spied (either conditionally or unconditionally).
- `shared` The predicate has been declared shared in the multi-threaded engine. This means that any dynamic code or tables for this predicate will be shared among threads, but it does not affect static, non-tabled code.
- `built_in` The predicate symbol has the same Functor and Arity as one of XSB's standard predicates, and is available tothe user without needing to load a file or import th epredicate from a module.
- `meta_predicate(Template)` The predicate is a meta-predicate. This property provides compatibility with other Prolog compilers and with forthcoming ISO Prolog standards.

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 `usermod`.

`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, 'i386-apple-darwin8.9.1').
%% configuration is usualy the same as architecture, but it can also
%% contain special tags, {\it e.g.}, i386-apple-darwin8.9.1-dbg, for a verion
%% built with debugging enabled.
xsb_configuration(configuration, 'i386-apple-darwin8.9.1-dbg').
xsb_configuration(host_os, 'darwin8.9.1').
xsb_configuration(os_version, '8.9.1').
xsb_configuration(os_type, 'darwin').
xsb_configuration(host_vendor, 'apple').
xsb_configuration(host_cpu,  'i386').
xsb_configuration(compiler, 'gcc').
xsb_configuration(compiler_flags, '-faltivec -fPOC -Wall -pipe -g').
xsb_configuration(loader_flags, '-g -lm ').
xsb_configuration(compile_mode, 'debug').
%% The type of XSB engine configured.
xsb_configuration(scheduling_strategy, '(local)').
xsb_configuration(engine_mode, 'slg-wam').
xsb_configuration(word_size, '32').
%% The following is XSB release information
xsb_configuration(major_version, '3').
```

```
xsb_configuration(minor_version, '3').
xsb_configuration(patch_version, '1').
xsb_configuration(beta_version, '').
xsb_configuration(version, '3.3.1').
xsb_configuration(codename, 'Pignoletto').
xsb_configuration(release_date, date(2011, 04, 12)).
%% Support for other languages
xsb_configuration(perl_support, 'yes').v
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) :- ...
```

`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, ?Specifier, ?Name)` ISO

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 `Specifier` 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` must be an integer in the range from 1 to 1200.

`Specifier` must be one of the atoms:

```
xfx xfy yfx fx fy hx hy xf yf
```

`Name` it must be an atom.

Error Cases

- `Precedence` is neither a variable nor an integer in the range from 1 to 1200.
  - `domain_error(operator_priority,Precedence)`
- `Specifier` is neither a variable nor an operator specifier of the types above.
  - `domain_error(operator_specifier,Specifier)`
- `Name` is neither a variable nor an atom.
  - `domain_error(atom_or_variable,Name)`

`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`.

6.13 Execution State

`break`

Causes the current execution to be suspended at the beginning of the next call. The interpreter then suspends the current computation, 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 down to break level 2, and so on. As long as the current computation occurs 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 [17].

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:  ?-
```

It is important to note that when XSB is interrupted via a ctrl-C, the current computation is suspended, and a new break level is entered. Any incomplete tables in the suspended computation can be examined, making break levels useful for for analyzing tabled computations under execution, as described in Section 10.3. However, it is also important to note that executing a tabled predicate during a break point throws an exception if there are incomplete tables that are suspended.

`halt` ISO

halt/0 Exits the XSB session regardless of the break level. On exiting the system cpu and elapsed time information is displayed.

`halt(Code)` ISO

halt/1 Exits the XSB session regardless of the break level, sending the integer `Code` to the parent process. Normally `0` is considered to indicate normal termination, while other exit codes are used to report various degrees of abnormality.

Error Cases

[17] If only the break-level computation should be aborted, the predicate `abort_level/[0,1]` can be called. This should rarely be needed, except if a specialized intepreter is written using XSB.

- `Code` is not an integer
  - `type_error(Integer,Code)`

`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 >
```

`trimcore` module: `machine`

A call to `trimcore/0` reallocates an XSB thread's execution stacks (and some tabling stacks) to their initial allocation size, the action affecting only the memory areas for the calling thread. When XSB is called in standalone or server mode, `trimcore/0` is automatically called when the top interpreter level is reached. When XSB is embedded in a process, `trimcore/0` is called at the top interpreter level for any thread created through xsb_ccall_thread_create() (see Volume 2, Chapter 3 *Embedding XSB in a Process*).

`gc_heap`

Explicitly invokes the garbage collector for a thread's heap. By default, heap garbage collection is called automatically for each thread upon stack expansion, unless the Prolog flag `heap_garbage_collection` is set to `none`. Automatic heap garbage collection should rarely need to be turned off, and should rarely need to be invoked manually.

`statistics`

Outputs time and memory usage information to the current output stream. This information is fairly detailed so the best way to explain is through an example. Figure 6.13 shows the output during a large and heavily tabled program written by XSB users.

Statistics first displays information about memory, then information about tabling operations as well as time.

- The first subsection of memory statistics, `permanent_space`, summarizes information about certain kinds of memory that is not generally under user control and is generally process-level tather than thread-specific. Allocated memory is broken into different classes including the following.
  - `atoms` Space used to maintain information about all predicates and structures.
  - `string` Space used to maintain information about all atomic constants in XSB.
  - `asserted` Space allocated for dynamic code.
  - `static` Space allocated for static code.
  - `foreign` Space allocated for foreign predicates.
  - `findall` Space allocated for buffers to support `findall/3` and similar predicates.
  - `profiling` Space used to maintain profiling information, if XSB is called with profiling on. (Not shown in Figure 6.13.)
  - `mt-private` Private space used by threads. (Only for the MT engine and not shown in Figure 6.13.)
  - `buffer` Space used for buffers used by forest logging, message queues and other libraries. (Not shown in Figure 6.13.)
  - `hash` Space used for hash-tables not otherwise classified, such as the storage library. (Not shown in Figure 6.13.)
  - `interprolog` space allocated for the InterProlog XSB/Java interface. (Not shown in Figure 6.13.)
  - `thread` In the MT engine, space allocated for the thread table, mutex array, and other global structures. (Not shown in Figure 6.13.)
  - `other` Other unclassified memory (usually this is a small amount).
- The next section summarizes information about XSB's main stacks. In the MT-engine this information is specific to the calling thread.
  - Global stack (heap) and local (environment) stack (see e.g. [1]) for the calling thread. Memory for these two WAM stacks is allocated as a single unit (per thread) so that each stack grows together; information is provided on the current allocation for the stacks as well as on the stack sizes themselves. (See Section 3.7.3 for initialization details.)
  - Trail and choice point stack (see e.g. [1]) for the calling thread. Memory for these two WAM stacks is allocated as a single unit (per thread) so that each stack grows together; information is provided on the cur-

```
| ?- statistics.
 Memory (total)        10674384280 bytes:     10543481920 in use,       130902360 free
   permanent space        58486184 bytes:        58486184 in use,               0 free
     atom                                         383976
     string                                      1736104
     asserted                                   53237408
     compiled                                    1267816
     findall                                     1060560
     buffer                                       787008
     other                                         13312
   glob/loc space        268435456 bytes:       141921760 in use,       126513696 free
     global                                     140607272
     local                                        1314488
   trail/cp space          8388608 bytes:         4631208 in use,         3757400 free
     trail                                         303176
     choice point                                 4328032
   SLG unific. space        131072 bytes:               0 in use,          131072 free
   SLG completion           262144 bytes:               0 in use,          262144 free
   SLG table space     10338680816 bytes:     10338442512 in use,          238304 free
   Incr table space                           620778800 in use

 Tabling Operations
   0 subsumptive call check/insert ops: 0 producers, 0 variants,
   0 properly subsumed (0 table entries), 0 used completed table.
   0 relevant answer ident ops.  0 consumptions via answer list.
   1251821353 variant call check/insert ops: 979496 producers, 1250841857 variants.
   11570194 answer check/insert ops: 97921 unique inserts, 11472273 redundant.
        4 DEs in the tables (space: 98312 bytes allocated,   200 in use)
        4 DLs in the tables (space: 49160 bytes allocated,   104 in use)

 Total number of incremental subgoals created: 1089296
 Currently 799432 incremental subgoals, 11240813 dependency edges

6 heap (6 string) garbage collections by sliding: collected 1202735 cells in 0.539478 secs
```

Figure 6.1: Statistics output from a large and heavily tabled program

rent allocation for the stacks as well as on the stack sizes themselves. (See Section 3.7.3 for initialization details.)

- SLG unification stack for the calling thread. This stack is used as a space to copy terms from the execution stacks into table space, or back out. This stack is not be reallocated unless extremely large terms are tabled.
- SLG completion stack for the calling thread. The completion stack is used to perform incremental completion for sets of mutually dependent tabled subgoals. One completion stack frame is allocated per tabled subgoal [68] but the size of these frames is version-dependent.
- Overall space used for tabling, followed by the amout of that overall space used for incremental tabling. (Generally speaking, this is the space that is used to contruct the IDG.)

- Information about the number of tabling operations performed in the session by any thread. Global counts (per-thread) are given first, followed by a few breakdowns of these counts by different types of tables.
  - The global information starts with the total number of calls to tabled predicates. Next is the total number or answer check/insert operations, followed by a breakdown into the number of unique answers generated, and the number of answers that were redundant when they were generated. Finally comes information about conditional answers, including the number of delay lists, and the total number delay elements: i.e., literals contained in delay lists. (See Section 5.3.2 for a general discussion of delay representation.)
  - Next comes a breakdown of the global numbers according to whether call subsumption or call variance is used.
    - Call Variance Subgoal Operations. For call variance the number of subgoal check/insert operations is given along with the unique number of subgoals encountered (`producers`) and the number of redundant consumer encountered (`variants`).
    - Call Subsumption Subgoal Operations. The total number calls to predicates that use call subsumption is given first. It is followed by the number of `producers` – that is, the number of distinct tables that have been created for these predicates. Next is `variants`, the number of repeated non-subsumed calls to these tables while the tables are non-completed and completed.
    - Call Subsumption Answer Consumption. In call subsumptive tabling, answer lists are copied from producer subgoals to subsumed con-

sumer subgoals (this operation is not required in variant tabling). The number of `answer ident` operations represents the number of times this copy is done. In addition, the number of consumptions performed by all consuming subsumptive table entries is also given. Note that these counts indicate the number of times an answer is consumed by a call subsumption table, while the overall count of answers produced is provided above.

– Finally, if incremental tabling is used, the total number of producers subgoals for incremental tables is given. This is followed by a measure of the incremental dependency graph (IDG) in its current state. The number of nodes in the IDG (incremental subgoals) is given, followed by the number of dependency edges.

- Garbage Collection Information. Time spent garbage collecting by the calling thread and number of heap cells collected.
- Information about process CPU and clock time, as well as the number of active threads.

`statistics(+Key)`

`statistics/1` allows the user to output detailed statistical information about the atom and symbol tables, as well as about table space. The following calls to `statistics/1` are supported:

- `statistics(reset)` Resets the CPU time as well as counts for various tabling operations.
- `statistics(atom)` Outputs statistics about both the atom and symbol tables. An example is:

```
| ?- statistics(atom).

Symbol table statistics:
------------------------
Table Size: 8191
Total Symbols: 1188
            used buckets:              1088  (range: [0, 8174])
            unused buckets:            7103
            maximum bucket size:       3  (#: 18)

String table statistics:
------------------------
Table Size: 16381
Total Strings: 1702
            used buckets:              1598  (range: [0, 16373])
            unused buckets:            14783
```

```
maximum bucket size:       3  (#: 2318)
```

- `statistics(summarize_idg)` Outputs a simple but sometimes useful summary of the IDG. This summary consists of counts of IDG nodes grouped at a predicate level. Counts are displayed for both tabled subgoals for incremental predicates along with subgoals to dynamic incremental predicates.
- `statistics(table)` Outputs *very* detailed statistics about table space, including breakdowns into variant and subsumptive call- and answer- trie nodes and hash tables; answer return list nodes, and structures for conditional answers (cf. [68, 64, 40, 18]). In the multi-threaded engine, these data structures are reported both for shared tables and for private tables of the calling thread.

  While this option is intended primarily for developers, it can also provide valuable information for the serious user of tabling.

Error Cases

- `Key` not a valid atom for input to `statistics/1`
  - `domain_error(statisticsInputDomain,Key))`

`statistics(?Key,-Result)`

`statistics/2` allows a user to determine information about resources used by XSB. Currently `statistics/2` unifies `Key` with

- `runtime`, which instantiates `Result` to the structure `[TotalCPU,IncrCPU]` where `TotalCPU` is the total (process-level) CPU time at the time of call, and `IncrCPU` is the CPU time taken since the last call to `statistics/2`. Times are measured in seconds. The process-level CPU time includes time taken for system calls, as well as time taken for garbage collection and stack-shifting. Note that in the multi-threaded engine, `statistics/2` measures the time for all threads.
- `walltime`, which instantiates `Result` to the list `[TotalTime,IncrTime]` where `TotalTime` is the total elapsed time at the time of call, and `IncrTime` is the elapsed time taken since the last call to `statistics/2`. Times are measured in seconds.
- `total_memory` which instantiates `Result` to the list `[Alloc,Used]`. In the single-threaded engine, `Alloc` is the total table space allocated and `Used` is the total table space used, both in bytes. In the multi-threaded engine, both refer to table space *private* to the calling thread.

- `tablespace` which instantiates `Result` to the list `[Alloc,Used]`. In the single-threaded engine, `Alloc` is the total table space allocated and `Used` is the total table space used, both in bytes. In the multi-threaded engine, both refer to table space *private* to the calling thread.
- `shared_tablespace` which instantiates `Result` to the list `[Alloc,Used]`. In the multi-threaded engine, `Alloc` is the total space allocated for *shared* tables and `Used` is the total table space used, both in bytes. An error is thrown if this option is called by the single-threaded engine.
- `trie_assert` which instantiates `Result` to the list `[Alloc,Used]`. In the single-threaded engine, `Alloc` is the total space allocated for trie-asserted facts and interned tries; `Used` is the total space used for these purposes, both in bytes.
- `heap` which instantiates `Result` to the total number of bytes used by XSB's heap. In the multi-threaded engine, the number refers only to the heap of the calling thread.
- `local` which instantiates `Result` to the total number of bytes used by XSB's local (environment) stack. In the multi-threaded engine, the number refers only to the local stack of the calling thread.
- `trail` which instantiates `Result` to the total number of bytes used by XSB's trail stack. In the multi-threaded engine, the number refers only to the trail stack of the calling thread.
- `choice_point` which instantiates `Result` to the total number of bytes used by XSB's choice point stack. In the multi-threaded engine, the number refers only to the choice point stack of the calling thread.
- `incomplete_tables` which instantiates `Result` to a list containing the following elements (in order):-
  - The number of incomplete tables in XSB's completion stack, i.e., the number of subgoals currently under evaluation.
  - The number of SCCs currently under evaluation in XSB's completion stack.

  In the multi-threaded engine, both of these numbers refer to the completion stack of the calling thread, which may contain both thread-private and thread-shared tables.
- `atoms` which instantiates `Result` to the number of bytes taken by atoms in the atom table.

- `idg` which instantiates `Result` to a a list containing (in order) the number of nodes and the number of edges currently in the incremental dependency graph (IDG).
- `table_ops` which instantiates `Result` to a list containing the following elements (in order):
  - The total number of calls to subgoals that are tabled using call subsumption.
  - The total number of distinct tables created using call subsumption (i.e., the total number of distinct calls to subgoals that are tabled using call subsumption).
  - The total number of calls to subgoals that are tabled using call variance.
  - The total number of distinct tables created using call variance (i.e., the total number of distinct calls to subgoals that are tabled using call variance).
  - The total number of check/insert operations for all answers (whether they are in subsumptive or variant tables).
  - The number of distinct answers added (whether they are in subsumptive or variant tables).

Example An example of using `statistics/2` to check CPU time is as follows:

```
?- statistics(runtime,[BeforeCumu,BeforeIncr]),spin(100000000),
   statistics(runtime,[AfterCumu,AfterIncr]).

BeforeCumu = 5.0167
BeforeIncr = 5.0167
AfterCumu = 9.6498
AfterIncr = 4.6331
```

Note that `statistics/2` can provide either cumulative or incremental times; here

$$AfterCumu - BeforeCumu = AfterIncr$$

Checking wall time is done similarly.

```
?- statistics(walltime,Before),sleep(1),statistics(walltime,After).

Before = [35.0651,35.0651]
After = [36.0652,1.0001]
```

Error Cases

- `Key` not a valid atom for input to `statistics/1`
  - `domain_error(statisticsInputDomain,Key))`

`time(+Goal)`

Prints both the CPU time and wall time taken by the execution of `Goal`. Any choice-points of `Goal` are discarded. The definition of predicate is based on the SWI-Prolog definition (minus reporting the number of inferences, which XSB does not currently support). This predicate is also found on other Prolog compilers such as YAP.

6.14 Asserting, Retracting, and Other Database Modifications

XSB provides an array of features for modifying the dynamic database. As a default, using `assert/1`, clauses can be asserted using first-argument indexing in a manner that is now standard to Prolog implementations. However, a variety of other behaviors 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, and 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 after it is retracted. Furthermore, the size of any index table expands dynamically as clauses are asserted.

All dynamic predicates are compiled into SLG-WAM code, however the manner of their compilation may differ, and the differences in compilation affect the semantics for the predicate. If a dynamic predicate P/n is given an indexing directive of `trie`, clauses for P/n will be compiled using trie instructions; otherwise clauses for P/n will be compiled into SLG-WAM instructions along the lines of static predicates.

Consider first dynamic predicates that use any indexing other than `trie` – including multiple or joint indices and star indexing. XSB asserts WAM code for such clauses so that that the execution time of dynamic code is similar to compiled code for unit and binary clauses. Furthermore, tabling can be used by explicitly declaring a predicate to be both dynamic and tabled. In Version 3.8, when the clause of a dynamic predicate is asserted as WAM code, the *"immediate semantics"* rather than the ISO Semantics of assert/retract [51]. The immediate semantics allows assert and retract to be fast and spatially efficient, but requires that significant care must be taken when modifying the definition of a predicate which is currently being executed.

If a dynamic predicate is given an indexing directive of `trie`, clauses of the predicate are compiled (upon a call `assert/1`) using trie instructions as described in [64]. Creation of trie-based dynamic code is significantly faster than creation of other dynamic code, and execution time may also be faster. However, trie-based predicates can only be used 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 289) 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.

`asserta(+Clause)` ISO

If the index specification for the predicate is not `trie`, 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`.

Note that because of the precedence of `:-/2`, asserting a clause containing this operator requires an extra set of parentheses: `assert((Head :- Body))`.

Error Cases

- `Clause` is not instantiated
  - `instantiation_error`
- `Clause` is not a callable clause.
  - `domain_error(callable,Clause)`
- `Clause` has a head that is a static built-in
  - `permission_error(modify,builtin,Clause)`
- `Clause` has a head that is a static user predicate
  - `permission_error(modify,static,Clause)`

`assertz(+Clause)` ISO

If the index specification for the predicate is not `trie`, 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`. Error cases are as with `asserta/1`.

Note that because of the precedence of `:-/2`, asserting a clause containing this operator requires an extra set of parentheses: `assert((Head :- Body))`.

`assert(+Clause)`

If the index specification for the predicate is not `trie`, this predicate adds a dynamic clause, `Clause`, to the database *after* any other clauses for the same predicate currently in the database (acting as `assertz/1`). If the index specification for the predicate is `trie`, the clause is asserted arbitrarily within the trie. Error cases are as with `assertz/1`.

Note that because of the precedence of `:-/2`, asserting a clause containing this operator requires an extra set of parentheses: `assert((Head :- Body))`.

`assert(+Clause,+AorZandVar,+Index)`

This is a lower-level interface to (non-trie-indexed) assert. It is normally not needed except in one particular situation, when assert aborts because it needs too many registers. In this case, this lower-level assert may allow the offending clause to be correctly asserted.

The default implementation of non-trie-indexed assert generates code with a single pass through the asserted term. Because of this, it cannot know when it has encountered the final occurrence of a variable, and thus it can never release (and thus re-use) registers that are used to refer to variables. Since there is a limit of 255 registers in the XSB virtual machine, asserting a clause with more than this many distinct variables results in an error. There is an alternative implementation of assert that initially traverses the clause to determine the number of occurrences of each variable and thus allows better use of registers during code generation.

`Clause` is the clause to assert. `AorZandVar` is an integer whose lower 2 bits are used: The low-order bit is 0 if the clause is to be added as the first clause, and 1 if it is to be added as the last clause. If the second bit (2) is on, then the clause is traversed to count variable occurrences and so improve register allocation for variables; if it is 0, the default one-pass code-generation is done. So, for example, if `AorZandVar` is 3, then the clause will be asserted as the last one in the predicate and the better register allocation will be used. `Index` indicates the argument(s) on which to index.

`retract(+Clause)` ISO

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))`.

The technical details on space reclamation are as follows. When retract is called, a check is made to determine whether it is safe to reclaim space for that clause. Safety is ensured when:

- A check is made of the choice point stack indicating that no choice point will backtrack into space that is being reclaimed; AND
  - The predicate is thread-private; OR
  - there is a single active thread
- AND if the predicate is tabled, there is no incomplete table for that predicate.

If it is safe to reclaim space for the clause, space is reclaimed immediately. Otherwise the clause is marked so that its space may later be reclaimed through garbage collection. (See `gc_dynamic/1`).

Error Cases

- `Clause` is not instantiated
  - `instantiation_error`
- `Clause` is not a callable clause.
  - `domain_error(callable,Clause)`
- `Clause` has a head that is a static built-in
  - `permission_error(modify,builtin,Clause)`
- `Clause` has a head that is a static user predicate
  - `permission_error(modify,static,Clause)`

`retractall(+Head)` ISO

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. Conditions for space reclamation and error cases are as with `retract/1`.

`abolish(+PredSpec)` ISO

Removes all information about the specified predicate. `PredSpec` is of the form `Pred/Arity`. Everything about the abolished predicate is completely forgotten by the system (including the `dynamic or static` property, whether the predicate is tabled, and whether the predicate is thread-shared or thread-private) [18]. Any completed tables for the predicate are also removed.

It is an error to abolish a predicate when there is more than 1 active thread, regardless of whether the predicate is thread-private or thread-shared. The reason for this is that, even if `PredInd` denotes a thread-private predicate, one thread may be making use of `PredInd` as another thread abolishes it. `abolish/1` throws an error in such a case to prevent such a semantic inconsistency. Similarly, if there is a non-completed table for `PredInd`, an error is thrown to prevent incompleteness in the tabled computation.

ISO Compatibility Note: Version 3.8 of XSB allows static predicates to be abolished and their space reclaimed. Such space is reclaimed immediately, and unlike the case for abolished static code, no check is made to ensure that XSB's choice point stack is free of choice points for the abolished static predicate. Abolishing static code is thus dangerous and should be avoided unless a user is certain it is safe to use.

Error Cases

- `PredInd`, `Pred` or `Arity` is not instantiated
  - `instantiation_error`
- `Arity` is not in the range $0..2^{16}$ (`max_arity`)
  - `domain_error(arity_indicator,Arity)`
- `PredInd` indicates a static built-in
  - `permission_error(modify,builtin,Predind)`
- `abolish/1` is called when there is more than 1 active thread.
  - `misc_error`
- `PredInd` has a non-completed table in the current thread.
  - `table_error`
- There are active backtrack points to a (dynamic) clause for `PredInd` [19].

[18] For compatibility with older Prologs, there is also an `abolish/2` which takes `Pred` and `Arity` as its two arguments.

[19] XSB throws an error in this case because garbage collection for abolished predicates has not been implemented (unlike for retract(all) and various table abolishes). Besides, you shouldn't be abolishing a predicate that you could backtrack into. What were you thinking?

- `misc_error`

`clause(+Head,?Body)` ISO

Returns through backtracking all dynamic clauses in the database whose head matches `Head` and Body matches `Body`. For facts the `Body` is `true`. `clause/2` works properly for all dynamically *asserted* clauses, even if they are trie-indexed; however `clause/2` does not access trie-inserted terms. In the multi-threaded engine, when a thread T calls `clause/2` it accesses both thread-shared dynamic code and thread-private dynamic code for T.

Error Cases

- `Head` is not instantiated
  - `instantiation_error`
- `Head` (or `Body`) is not a callable clause.
  - `domain_error(callable,Head)`
- `Head` is a static built-in
  - `permission_error(access,builtin,Head)`
- `Head` is a static user predicate
  - `permission_error(access,static,Clause)`

`gc_dynamic(-N)`

Invokes the garbage collector for dynamic clauses that have been retracted, or whose predicate has been abolished. When called with more than 1 active thread, `gc_dynamic/1` will always perform garbage collection for that thread's private retracted clauses; however in Version 3.8, it will only perform garbage collection for retracted thread-shared clauses if there is a single active thread. `N` is the number or shared and/or private frames left to be collected – if `N` is unified to 0, then all possible garbage collecting has been performed. `N` is unified to -1 garbage collection was not attempted (due to multiple active threads).

By default, `gc_dynamic/1` is called automatically at the top level of the XSB interpreter, when abolishing a predicate, and when calling retractall for an "open" term containing no variable bindings.

`index(+PredSpec, +IndexSpec)`

In `index(PredSpec, IndexSpec)`, `PredSpec` is a predicate indicator or term indicator, and `IndexSpec` is a form of index specification as described below.

In general, XSB supports hash-based indexing on various arguments of clauses, on combinations of arguments, as well as within the arguments of a clause. The

availability of various kinds of indexing depends on whether code is static (e.g. compiled) or dynamic (e.g. asserted, loaded with `load_dyn/1` and so on). Index directives can be given to the compiler as part of source code or executed during program execution (analogously to `op/3`). When executed during program execution, `index/2` does *not* re-index an already existing predicate; however for dynamic predicates `index/2` does affect the index for clauses asserted after the directive has been given.

- *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), a wide variety of indexing techniques are possible. We discuss their syntax first, and then their semantics. For dynamic predicates then, `IndexSpec` can be either an *indexing element* or a list of indexing elements. Each indexing element defines a separate index and specifies an argument or group of arguments that make up the search key of that index. Thus an indexing element consists of one or more *argument indicators* joined together by `+/2`. An argument indicator is may be an integer (`ArgNo`) indicating an argument number (starting from 1) to use in the index, or it may have the form `*(ArgNo)`. If `ArgNo` is an integer, only the main functor symbol of argument `ArgNo` will participate in the index. When annotated with the asterisk, the first 5 fields of argument `ArgNo` (in a depth-first traversal of the term) will be used in the index. If there are fewer than 5, they all will be used. If any of the first 5 is a variable, then the index cannot be used. An index is usually on a single argument, in which case the indexing element consists of a single argument indicator. If an indexing element contains more than one argument specifier, then a joint index is specified i.e. an index will be constructed so that the values of each argument indicator are to be concatenated to create the search key of the index.
    Examples help clarify this. `index(p/3,[2,1])` indicates that clauses asserted for the predicate `p/3` should be indexed on both the second and the first argument. A query Q to `p/3` will first use the second argument index to `p/3` if the second argument of Q is non-variable, and will use the main functor of the second argument. Otherwise, if the second argument of Q is a variable, but not the first argument,

the first argument index of `p/3` will be used. If both arguments in Q are variables, no index will be used and Q will backtrack through all clauses for `p/3`.
`index(p/3,[*(2),1])` would result in similar behavior as the previous example, but the first index to be tried (on the second argument) would be built using more of the term value in that second argument position (not just the main functor symbol.)
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 non-variable and if so, use an index based on both those values. Otherwise, it would see if the first argument is non-variable and if so, use an index based on it. Otherwise, it would see if the fourth argument is non-variable 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. In each of these cases, the indexes are built using only the main functor symbol in the indicated argument position. (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.)
If we want to use similar indexing on `p/5` of the previous example, except say argument 1 takes on complex term values and we want to index on more of those terms, we might specify the index as `index(p/5,[*(1)+2,*(1),4])`.

- *Trie-based Indexing* If `Predspec` is dynamic, the executable directive `index(Predspec,trie)` causes clauses for `Predspec` to be asserted using tries (see [64], 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, a 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, trie indexing 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.

If in doubt what indexing is being used for a predicate, a call to `current_index/2` can be made.

Error Cases

- `PredSpec` or `IndexSpec` is a variable
  - `instantiation_error`
- `PredSpec` is neither a variable, a predicate indicator, nor a callable term.
  - `type_error(predicate_indicator_or_callable,PredSpec)`
- `IndexSpec` is not ground
  - `instantiation_error`
- `IndexSpec` is neither a properly formed indexing element nor a list of indexing elements
  - `domain_error(indexing_element,IndexSpec)`
- `IndexSpec` is a list containing an element `IndexElt` that not a properly formed indexing element
  - `domain_error(indexing_element,IndexElt)`
- `PredSpec` represents a predicate that has been previously defined to be static
  - `permission_error(modify,static_predicate)`

`dynamic(+Operations)` ISO

`dynamic/1` can be used either as a compiler declaration or as an executable directive. Used as a compiler declaration, it indicates that all clauses for each predicate denoted by the command are dynamic – clauses for these predicates can be asserted or retracted. Without this declaration compiled clauses will be treated as static. Executed as a directive in a state of execution where no clauses exist for each denoted predicate `dynamic/1` ensures clauses for the affected predicates are to be treated as dynamic. If `PredSpec` contains a predicate that is defined as static or as foreign code, a permission error will be thrown. `Operations` can take one of two forms:

1. `Operations` is a predicate indicator, a callable term, or a comma-list of predicate indicators or callable terms.
2. `Operations` has the form `Predspec as Options` where

- `PredSpec` is a predicate indicator, a callable term, or comma-list of predicate indicators or callable terms.
- `Options` is either a dynamic_option or a list of dynamic_options. These dynamic options control the attributes of a dyamic predicate. In Version 3.8, the following dynamic options are supported
  - `intern` which causes every clause for this predicate, before being asserted, to force all its ground subterms to be interned into a global table.
  - `tabled` which causes the dynamic predicate to be tabled. The declaration/directive `dynamic p/n as tabled` has the same effect as `table p/n as dynamic`.
  - `variant` which causes the table evaluation method of the predicate(s) to use call variance.
  - `incremental` which allows (incremental) tables that are based on the dynamic predicate to be automatically updated when clauses are asserted or retracted.
  - `opaque`. This option is essencially the same as non-incremental dynamic code, *except* that `opaque` predicates can be made `incremental` by a later `dynamic/1` directive, and `incremental` predicates can be made `opaque` by a `dynamic/1` directuve.
  - `private` which causes the predicate(s) to be treated as thread private.
  - `shared` which causes the predicate(s) to be treated as thread shared.

If the directive

`dynamic` p/n`.`

is executed, its behavior is as follows:

- If p/n is already dynamic, the directive has no effect, regardless of wither p/n is tabled, incremental or opaque, private or shared.
- If p/n has *not* already been defined, the directive makes p/n non-tabled, non-incremental, and to use the default thread sharing strategy (`private` unless XSB is called with `-shared_predicates`).

If the directive

`dynamic` *PredList* `as` *Options*`.`

is executed, various checks are performed on *Options*. These checks are (mostly) performed before any predicates are declared as dynamic or options changed,

and reduce the possibility of leaving some p/n in *PredList* with inconsistent attributes.

- If a dynamic predicate in `Predlist` is declared as `incremental` it may be changed to `opaque` at any time; similarly, a dynamic predicate that is `opaque` may be changed to `incremental`
- Otherwise, an attempt to change an attribute of p/n in *PredList* – i.e. whether p/n is tabled or not, incremental/opaque or not, and thread-private or thread-shared – will throw a permission error.

In addition, regardless of the state of predicates in *PredList*, if options contains an inconstent set of declarations, a domain error will be thrown. `Options` is inconsistent in the following cases:

- `Options` contains `tabled` or `variant` and `opaque` or `incremental`. Tabled dynamic incremental code is not yet supported in XSB.
- `Options` contains both `private` and `shared`
- `Options` contains both `incremental` and `opaque`
- `Options` contains `intern` and (`dynamic` or `subsumptive` or `incremental` or `opaque`)

Error Cases

Error cases are summarized as follows. Let `Operations` be of the form `PredSpec` or `PredSpec` as `Options`. Then if

- `PredSpec` or is a variable or a comma list containing a variable
  - `instantiation_error`
- An element of `PredSpec` is neither a variable nor a comma list
  - `type_error(callable,PredSpec)`
- A predicate in `PredSpec` has been previously defined to be static or foreign
  - `permission_error(modify,static_predicate)`
- `Options` is a variable or a list containing a variable
  - `instantiation_error`
- `Options` contains an element `Option` that isn't a dynamic option (as described above)
  - `domain_error(dynamic_option,Option)`

- `Options` contains inconsistnet elements (as described above)
  - `table_error`
- An option in `Options` would modify a predicate in predspec in a manner that is not allowed (as described above)
  - `permission_error`

In addition, if a predicate `p/n` was declared to be dynamic and a file containing clauses for `p/n` is later consulted, a permission error will be thrown.

6.14.1 Reading Dynamic Code from Files

Several built-in predicates are available that can assert the contents of a file into XSB's database. These predicates are useful when code needs to be dynamic, or when the they contain a large number of clauses or facts. Configured properly, files containing millions of facts can be read and asserted into memory in under a minute, making XSB suitable for certain kinds of in-memory database operations [20].

Each of the predicates in this section allow loading from files with proper prolog extensions, and makes use of the XSB library paths. See Sections 3.6 and 3.3 for details.

`load_dyn(+FileName)`

Asserts the contents of file `FileName` into the database. All existing clauses of the predicates in the file that already appear in the database, are retracted, unless there is a `multifile/1` declaration for them. An indexing declaration of a predicate `p/n` in `FileName` will be observed as long as the declarations occur before the first clause of `p/n`. Clauses in `FileName` must be in a format that `read/1` will process. So, for example, operators are permitted. Modules (files containing export statements) can be loaded and all terms are treated as they are in the compiler.

Dynamically loaded files can be filtered through the XSB preprocessor. To do this, put the following in the source file:

```
:- compiler_options([xpp_on]).
```

[20] In Version 3.8, loading code dynamically can also be useful when the clauses contain atoms whose length is more than 255 that cannot be handled by the XSB compiler.

Of course, the name `compiler_options` might seem like a misnomer here (since the file is not being compiled), but it is convenient to use the same directive both for compiling and loading, in case the same source file is used both ways.

Error Cases

- `FileName` is a variable
  - `instantiation_error`
- `FileName` is not an atom.
  - `type_error(atom,Filename)`
- `FileName` has been loaded previously in the session *and* there is more than one active thread.
  - `misc_error`

`load_dyn(+FileName,+Dir)`

Asserts the contents of file `FileName` into the database. `Dir` indicates whether `assertz` or `asserta` is to be used. If `Dir` is `z`, then `assertz` is used and the behavior of `load_dyn(FileName)` is obtained. If `Dir` is `a`, then `asserta` is used to add the clauses to the database, and clauses will be in the reverse order of their appearance in the input file. `asserta` is faster than `assertz` for predicates such that their indexing and data result in many hash collisions. `Dir` is ignored for facts in `FileName` that are trie-indexed.

Error Cases

- `FileName` is a variable
  - `instantiation_error`
- `FileName` is not an atom:
  - `type_error(atom,FileName)`
- `Dir` is not equal to `a` or `z` [21]:
  - `domain_error(a_or_z,Dir)`
- `FileName` has been loaded previously in the session *and* there is more than one active thread.
  - `misc_error`

[21] For backward compatibility, `0` and `1` are also allowed.

`load_dync(+FileName)`

Acts as `load_dyn/1`, but assumes that facts are in "canonical" format and is much faster as a result. In XSB, a term is in canonical format if it does not use any operators other than list notation and comma-list notation. This is the format produced by the predicate `write_canonical/1`. (See `cvt_canonical/2` to convert a file from the usual `read/1` format to `read_canonical` format.) As usual, clauses of predicates are not retracted if they are compiled instead of dynamically asserted. All predicates are loaded into `usermod`. `:- export` declarations are ignored and a warning is issued.

Notice that this predicate can be used to load files of Datalog facts (since they will be in canonical format). This predicate is significantly faster than `load_dyn/1` and should be used when speed is important. (See `load_dync/2` below for further efficiency considerations.) A file that is to be dynamically loaded often but not often modified by hand should be loaded with this predicate.

As with `load_dyn/1`, the source file can be filtered through the C preprocessor. However, since all clauses in such a file must be in canonical form, the `compiler_options/1` directive should look as follows:

```
:-(compiler_options('.'(xpp_on,[]))).
```

Error Cases

- `FileName` is a variable
  - `instantiation_error`
- `FileName` is not an atom.
  - `type_error(atom,FileName)`
- `FileName` has been loaded previously in the session *and* there is more than one active thread.
  - `misc_error`

`load_dync(+FileName,+Dir)`

Acts as `load_dyn/2`, but assumes that facts are in "canonical" format. `Dir` is ignored for trie-asserted code, but otherwise indicates whether `assertz` or `asserta` is to be used. If `Dir` is `z`, then `assertz` is used and the exact behavior of `load_dync(FileName)` is obtained. If `Dir` is `a`, then `asserta` is used to add the clauses to the database, and clauses will end up in the reverse order of their appearance in the input file.

Setting `Dir` to `a` for non trie-asserted code can sometimes be *much* faster than the default of `z`. The reason has to do with how indexes on dynamic code are represented. Indexes use hash tables with bucket chains. No pointers are kept to the ends of bucket chains, so when adding a new clause to the end of a bucket (as in `assertz`), the entire chain must be run. Notice that in the limiting case of only one populated bucket (e.g., when all clauses have the same index term), this makes assertz-ing a sequence of clauses quadratic. However, when using `asserta`, the new clause is added to the beginning of its hash bucket, and this can be done in constant time, resulting in linear behavior for asserta-ing a sequence of clauses.

Error Cases

- `FileName` is a variable
  - `instantiation_error`
- `FileName` is not an atom:
  - `type_error(atom,FileName)`
- `Dir` is not instantiated to `a` or `z` [22]:
  - `domain_error(a_or_z,Dir)`
- `FileName` has been loaded previously in the session *and* there is more than one active thread.
  - `misc_error`

`ensure_loaded(+FileName,+Action)`

This predicate does nothing if `FileName` has been loaded or consulted into XSB, and has not changed since it was loaded or consulted. Otherwise

- If `Action` is instantiated to `dyn` the behavior is as `load_dyn/1` (or `load_dyn(FileName,z)`).
- If `Action` is instantiated to `dyna` the behavior is as `load_dyn(FileName,a)`.
- If `Action` is instantiated to `dync` the behavior is as `load_dync/1` (or `load_dync(FileName,z)`).
- If `Action` is instantiated to `dynca` the behavior is as `load_dync(FileName,a)`.
- If `Action` is instantiated to `consult`, `FileName` is consulted (action is the same as `ensure_loaded/1`).

Error Cases

[22] For backward compatibility, `0` and `1` are also allowed.

- `FileName` is not instantiated:
  - `instantiation_error`
- `FileName` is not an atom:
  - `type_error(atom,FileName)`
- `Action` is not a valid load action as described above
  - `domain_error(loadAction,Action)`

`cvt_canonical(+FileName1,+FileName2)` module: `consult`

Converts a file from standard term format to "canonical" format. The input file name is `FileName1`; the converted file is put in `FileName2`. This predicate can be used to convert a file in standard Prolog format to one loadable by `load_dync/1`.

6.14.2 The `storage` Module: Associative Arrays and Backtrackable Updates

XSB provides a high-level interface that allows the creation of "objects" that efficiently manage the storage of facts or of associations between keys and values. Of course, facts and associative arrays can be easily managed in Prolog itself, but the `storage` module is highly efficient and supports the semantics of backtrackable updates as defined by Transaction logic [6] in addition to immediate updates. The semantics of backtrackable updates means that an update made by the storage module may is provisional until the update is committed. Otherwise, if a subgoal calling the update fails, the change is undone. The commit itself may be made either by the predicate `storage_commit/1`, or less cleanly by cutting over the update itself.

A storage object O is referred to by a name, which must be a Prolog atom. O can be associated either with a set of facts or a set of *key-value pairs*. Within a given storage object each key is associated with a unique value: however since keys and values can be arbitrary Prolog terms, this constraint need not be a practical restriction. A storage object O is created on demand, simply by calling (a backtrackable or non-backtrackable) update predicate that refers to O. However to reclaim O's space within a running thread, the predicate `storage_reclaim_space/1` must be called. Both backtackable and non-backtrackable updates can be made to the same storage object, although doing so may not always be a good programming practice.

If multiple threads are used, each storage object is private to a thread, and space for a storage object is reclaimed upon a thread's exit. Thread-shared storage objects may be supported in future versions.

All the predicates described in this section must be imported from module `storage`.

Non-backtrackable Storage

`storage_insert_keypair(+StorageName,+Key, +Value, ?Inserted)`
: Insert the given Key-Value pair into `StorageName`. If the pair is new, then `Inserted` unifies with `1`. If the pair is already in `StorageName`, then `Inserted` unifies with `0`. If `StorageName` already contains a pair with the given key that is associated with a *different* value, then `Inserted` unifies with `-1`. The first argument, `StorageName`, must be an atom naming the storage to be used. Different names denote different storages. In all cases the predicate succeeds.

`storage_delete_keypair(+StorageName, +Key, ?Deleted)`
: Delete the key-value pair with the given key from `StorageName`. If the pair was in `StorageName` then `Deleted` unifies with `1`. If it was *not* in `StorageName`s then `Deleted` unifies with `0`. The first argument, `StorageName`, must be an atom naming the storage object to be used. Different names denote different storages. In both cases the predicate succeeds.

`storage_find_keypair(+StorageName, +Key, ?Value)`
: If `StorageName` has a key pair with the given key, then `Value` unifies with the value stored in `StorageName`. If no such pair exists in the database, then the goal fails.

  Note that this predicate works with non-backtrackable associative arrays described above as well as with the backtrackable ones, described below.

`storage_insert_fact(+StorageName, +Fact, ?Inserted)`
: Similar to keypair insertion, but this primitive inserts facts rather than key pairs.

`storage_delete_fact(+StorageName, +Fact, ?Inserted)`
: Similar to key-pair deletion, but this primitive deletes facts rather than key pairs.

`storage_find_fact(+StorageName, +Fact)`
: Similar to key-pair finding, but this primitive finds facts facts rather than key pairs.

Backtrackable Updates

`storage_insert_keypair_bt(+StorageName, +Key, +Value, ?Inserted)`
: Calling this predicate inserts a key pair into the trie represented by `StorageName`, similarly to `storage_insert_keypair/4`, and the key-value pair can then be queried via `storage_find_keypair/3`, just as with the non-backtrackable updates described above. In addition, the key-value pair can be removed from `StorageName` by explicit deletion. However, the key pair will be removed from `StorageName` upon failing over the insertion goal *unless* a commit is made to `StorageName` through the goal `storage_commit(StorageName)`. The exact semantics is defined by Transaction Logic [6].

  Note it is the update itself that is backtrackable, not the key-value pair. Hence, a key-pair may be (provisionally) inserted by a backtrackable update and deleted by a non-backtrackable update, or inserted by a non-backtrackable update and (provisionally) deleted by a backtrackable update. Of course, whether such a mixture makes sense would depend on a given application.

`storage_delete_keypair_bt(+StorageName, +Key, ?Deleted)`
: Like `storage_delete_keypair/3`, but backtrackable as described for the predicate `storage_insert_keypair_bt/4`.

`storage_insert_fact_bt(+StorageName, +Goal)`
: Like `storage_insert_fact/2`, but backtrackable.

`storage_delete_fact_bt(+StorageName, +Goal)`
: This is a backtrackable version of `storage_delete_fact/2`.

`storage_commit(+StorageName)`
: Commits to `StorageName` any backtrackable updates since the last commit, or since initialization if no commit has been made to `StorageName`. If `StorageName` does not exist, the predicate silently fails.

Reclaiming Space

`storage_reclaim_space(+StorageName)`
: This is similar to `reclaim_space/1` for `assert` and `retract`, but it is used for storage managed by the primitives defined in the `storage` module. As with `reclaim_space/1`, this goal is typically called just before returning to the top level.

6.15 Tabling Declarations and Builtins

In XSB, tables are designed so that they can be used transparently by computations. However, it is necessary to first inform the system of which predicates should be evaluated using tabled resolution (Section 3.10.2) along with the properties to be used, such as call variance or call subsumptiion (Chapter 5). Further, it is often useful to be able to explicitly inspect a table, or to alter its state. The predicates described in this section are provided for these purposes. In order to ground the discussion of these predicates, we continue our overview of tables and table creation from Chapter 5. For a detailed description of the implementation of table access routines in XSB, the reader is referred to [64, 40, 19, 82] and other papers listed in the bibliography.

Tables and Table Entries

Abstractly, a table $\mathcal{T}$ can be seen as a triple $\langle S, \mathcal{A}, Status \rangle$ where S is a subgoal, $\mathcal{A}$ is its associated answer set, and *Status* its status — whether the table is `complete` or `incomplete`, along with tabling properties it uses (e.g., incremental or non-incremental, cf. Chapter 5 for a discussion of tabling properties). XSB's table inspection built-ins sometimes use a *TableEntryHandle* to efficiently access $\mathcal{T}$ and a *ReturnHandle* to access $\mathcal{A}$. Often is is useful to access or manipulate the set of all (subgoal-level) tables for some tabled predicate `p/n`. We thus sometimes abuse terminology slightly by referring to this set as a *predicate-level* table.

At execution time, invocation of a tabled subgoal S leads to the classification of S according to the properties associated with its predicate, as well as its possible creation of a table for S. Each occurrence of a subgoal that is not yet completely evaluated can be classified as either (a) a *generator*, of answers or (b) a *consumer* of those answers.

Skeletons and Predicate Specifications

A *skeleton* for a functor `f/n` is a structure of the form `f(`Arg_1`,...,`Arg_n`)` where each Arg_i is a distinct variable. Similarly the skeleton of a term is the skeleton formed from the principal functor of the term, so that skeletons from the terms `f(1,2)` and `f(A,B)` are the same. A *return skeleton* is a specific application of this notion to answers. From it, one may discern the size of the template for a given subgoal. Below, we assume that a predicate specification for a predicate `p` and arity `n`, represented as `PredSpec` below, can be given either using the notation `p/n` or as a skeleton,

`p(`t_1`,...,`t_n`).`

6.15.1 Declaring and Modifying Tabled Predicates

`table(+Operations)` Tabling

`table/1` can be used either as a compiler declaration or as an executable directive. Used as a compiler declaration, it indicates that each predicate denoted by the command is to be compiled using (a particular form of) tabling, and may indicate that the predicate itself is dynamic or thread-shared or thread-private. Executed as a directive in a state of execution where no clauses exist for each denoted predicate `table/1` ensures that any clauses asserted for each predicate use tabling and may indicate the mode of tabling to be used. The parameter `Operations` can take one of three forms:

1. `Operations` is a predicate indicator, a skeleton, or a comma-list or list of predicate indicators or skeletons.
2. `Operations` is a term indicating that a predicate is to be tabled with a particular form of answer subsumption (cf. Section 5.4).
3. `Operations` has the form `Predspec as Options` where
   - `PredSpec` is a predicate indicator, a skeleton, or a comma-list or list of predicate indicators or skeletons.
   - `Options` is either a table option or a list of table options. In Version 3.8, the following table options are supported
     - `dynamic` or `dyn` which causes the predicate(s) to be treated as dynamic in addition to being tabled, and is equivalent to `?- dynamic PredSpec` [23]
     - `subsumptive` which causes the table evaluation method of the predicate(s) to use call subsumption.
     - `variant` which causes the table evaluation method of the predicate(s) to use call variance.
     - `intern` which causes all ground subterms of subgoals and answers entered into the table for the predicate(s) to be interned.
     - `incremental` which causes the table evaluation method of the predicate(s) to be incremental.

[23] Because dynamic is an operator, the declaration requires parentheses, e.g.: `table p/n as (dynamic).`

- `opaque` which indicates that the tables predicate is used in the definition of an incremental table, but are not to be incrementally maintained themselves.
- `private` which causes the predicate(s) to be treated as thread private in addition to being tabled.
- `shared` which causes the predicate(s) to be treated as thread shared in addition to being tabled.
- `subgoal_abstract(n)` which enables size-based subgoal abstraction for the predicate(s).
- `answer_abstract(n)` which enables depth-n answer abstraction for the predicate(s).

If the directive

`table` *PredList* `as` *Options*`.`

is executed, various checks are performed on *Options*. These checks are (mostly) performed before any predicates are declared as dynamic or options changed, and reduce the possibility of leaving some p/n in *PredList* with inconsistent attributes, which could cause an error to be thrown during program execution.

- If a predicate in `Predlist` has been declared as `incremental` it may be changed to `opaque` at any time; similarly, a predicate that is `opaque` may be changed to `incremental`
- If a predicate in `Predlist` has been declared to use call variance it may be changed to use call subsumption at any time; similarly, a predicate that uses call subsumption may be changed to use call variance.
- Otherwise, an attempt to change an attribute of p/n in *PredList* – i.e. whether p/n is tabled or not, dynamic or not and thread-private or thread-shared – will throw a permission error.

In addition, regardless of the state of predicates in *PredList*, if options contains an unsupported set of declarations, a permission error will be thrown (see Table 5.1 for a list of supported and non-supported combinations of tabling modes and predicate properties). `Options` throws a table error in the following cases:

- `Options` contains `dynamic` and either `opaque` or `incremental`. Tabled dynamic incremental code is not yet supported in XSB.
- `Options` contains (`incremental` or `opaque`) and (`subsumptive` or `shared`)
- `Options` contains `subsumptive` and (`variant` or `shared` or `subgoal_abstract/1` or `answer_abstract/1`)

- `Options` contains `intern` and (`dynamic` or `subsumptive` or `approximate` or `incremental` or `opaque` or `answer_abstract` or `subgoal_abstract`)
- `Options` contains both `private` and `shared`
- `Options` contains both `incremental` and `opaque`

Error Cases

Error cases are summarized as follows. Let `Operations` be of the form `PredSpec` or `PredSpec` as `Options`. Then if

- `PredSpec` or is a variable or a comma list containing a variable
  - `instantiation_error`
- An element of `PredSpec` is neither a variable nor a predicate indicator, nor a skeleton.
  - `type_error(callable,PredSpec)`
- A predicate in `PredSpec` has been previously defined to be static or foreign and `Options` contains `dynamic` or `dyn`
  - `permission_error(modify,static_predicate)`
- `Options` is a variable or a list containing a variable
  - `instantiation_error`
- `Options` contains an element `Option` that isn't a table option (as described above)
  - `domain_error(table_option,Option)`
- `Options` contains a non-supported combination of elements (as described above)
  - `permission_error`
- An option in `Options` would modify a predicate in `PredSpec` in a manner that is not allowed (as described above)
  - `permission_error`

6.15.2 Predicates for Table Inspection

Often, the higher level inspection predicates described in Section 10.3 are the best bet for analyzing tables and other aspects of the state of computation. However, for some purposes, a finer level of control is needed, which these predicates provide. In

this section we describe inspection predciates that can be used to quickly examine a collection of tables. In the next section, we describe lower-level inspection predicates that are special-purpose, and may not be needed by most users.

For explanitory purposes, we maintain two running examples in this section and the next. The first uses tabling based on call variance:

Call Variance Example

Program

```
:- table p/2 as variant.

p(1,2).
p(1,3).
p(1,_).
p(2,3).
```

Table

Subgoal	Answer Set	Status
p(1,Y)	p(1,2) p(1,3) p(1,Y)	complete
p(X,3)	p(1,3) p(2,3)	complete

and the second uses tabling based on call subsumption::

Call Subsumption Example

Program

```
:- table q/2 as subsumptive.
q(a,b).
q(b,c).
q(a,c).
```

Table

Subgoal	Answer Set	Status
q(X,Y)	q(a,b) q(b,c) q(a,c)	complete
q(a,Y)	*q(a,b)* *q(a,c)*	complete
q(X,c)	*q(b,c)* *q(a,c)*	complete

Note that in the call subsumption example, the subgoals `q(a,Y)` and `q(X,c)` are subsumed by, and hence obtain their answers from, the subgoal `q(X,Y)`.

`get_calls_for_table(+PredSpec,?Call)` Tabling

Identifies through backtracking all tabled subgoals whose predicate is that of `PredSpec` and that unify with `Call`. `PredSpec` is left unchanged while `Call` contains the unified result. Its behavior is shown in Example 6.15.1.

Example 6.15.1 (get_calls_for_table/2)

Variant Predicate

```
|?- get_calls_for_table(p(1,3),Call).

Call = p(_h142,3);

Call = p(1,_h143);

no
| ?- get_calls_for_table(p/2,Call).

Call = p(_h137,3);

Call = p(1,_h138);

no
```

Subsumptive Predicate

```
| ?- get_calls_for_table(q(X,Y),Call).

X = _h80
Y = _h94
Call = q(a,_h167);

X = _h80
Y = _h94
Call = q(_h166,c);

X = _h80
Y = _h94
Call = q(_h166,_h167);

no
```

`get_returns_for_call(+Subgoal,?AnswerTerm)` Tabling

Succeeds through backtracking for each answer of the subgoal `Subgoal` which unifies with `AnswerTerm`. Fails if `Subgoal` is not a tabled subgoal or `AnswerTerm` does not unify with any of its answers or if `Subgoal` has no answers.

The answer is created in its entirety, including fresh variables so that `Subgoal` is *not* further instantiated. Of course the user may unify `Subgoal` with its answer if desired. Example 6.15.2 illustrates its behavior.

Example 6.15.2 (get_returns_for_call/2)

Variant Predicate

```
| ?- get_returns_for_call(p(1,Y),
                          AnsTerm).

Y = _h88
AnsTerm = p(1,_h161);

Y = _h88
AnsTerm = p(1,3);

Y = _h88
AnsTerm = p(1,2);

no
| ?- get_returns_for_call(p(X,Y),
                          AnsTerm).

no
| ?- get_returns_for_call(p(1,2),
                          AnsTerm).

no
```

Subsumptive Predicate

```
| ?- get_returns_for_call(q(a,Y),
                          AnsTerm).

Y = _h88
AnsTerm = q(a,c);

Y = _h88
AnsTerm = q(a,b);

no
| ?- get_returns_for_call(q(X,c),
                          AnsTerm).

X = _h80
AnsTerm = q(b,c);

X = _h80
AnsTerm = q(a,c);

no
```

`get_residual(#CallTerm,?DelayList)` Tabling

`variant_get_residual(#CallTerm,?DelayList)` Tabling

`get_residual/2` backtracks through the answers to each *completed* subgoal in the table that unifies with `CallTerm`. For each such answer A, `CallTerm` is unified with A, and `DelayList` with a delay list of A if A is conditional, and otherwise with the empty list.

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 Chapter 5) `get_residual/2` is useful to examine portions of the residual program. Example 6.15.3 illustrates such a use.

Example 6.15.3 (get_residual/2) *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)).
```

Subgoal	Answers
p(1,X)	*p(1,2)* *p(1,3):- tnot(p(2,3))*
p(1,3)	*p(1,3):- tnot(p(2,3))*
p(2,3)	*p(2,3):- tnot(p(1,3))*

the completed subgoals are `p(1,X)`, `p(1,3)`, *and* `p(2,3)`. *Calls to* `get_residual/2` *will act as follows*

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

X = 1       % from subgoal p(1,X)
Y = 2
List = [];

X = 1       % from subgoal p(1,X)
Y = 3
List = [tnot(p(2,3))];

X = 1       % from subgoal p(1,3)
Y = 3
List = [tnot(p(2,3))];

X = 2       % from subgoal p(2,3)
Y = 3
List = [tnot(p(1,3))];

no
```

For other purposes, it may be desired to examine the answers for a particular subgoal, rather than for all subgoals that unifiy with `CallTerm`. In this case, `variant_get_residual/2` can be used, which backtracks through all answers for `CallTerm` if `CallTerm` is a tabled subgoal with answers, and fails otherwise. For the above example, `variant_get_residual/2` behaves as follows:

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

no
| ?- variant_get_residual(p(1,Y),List).

X = 1       % from subgoal p(1,X)
```

```
Y = 2
List = [];

X = 1       % from subgoal p(1,X)
Y = 3
List = [tnot(p(2,3))];

no
```

Error Cases

- `CallTerm` is not a callable term
  - `type_error(callable_term,CallTerm)`
- `CallTerm` does not correspond to a tabled predicate
  - `permission_error(table access,non-tabled predicate,CallTerm)`

`table_state(+Subgoal,?Strategy,?CallType,?AnsSetStatus)` Tabling

`table_state(+TableEntryHandle,?Strategy,?CallType,?AnsSetStatus)` Tabling

May succeed whenever `Subgoal` is a subgoal in the table, or `TableEntryHandle` is a valid reference to a table entry. In either case, certain arguments 2 through 4 unify with constants representing properties of the table. Taken together, these properties provide a detailed description of current state of the given subgoal within an evaluation. The combinations valid in the current version of XSB and their specific meaning is given in the following table. Notice that not only can these combinations describe the characteristics of a subgoal in the table, but they are also equipped to predict how `CallTerm` would have been treated had it been called at that moment.

Strategy	CallType	AnsSetStatus	Description
variant	producer	complete	Self explanatory.
		incremental_needs_reeval	An incremental table that has been invalidated, and is therefore inconsistent with a KB and needs recomputation (which will be lazily done).
		incomplete	Self explanatory.
	no_entry	undefined	The call does not appear in the table.
subsumptive	producer	complete	Self explanatory.
		incomplete	Self explanatory.
	subsumed	complete	The call is in the table and is properly subsumed by a completed producer.
		incomplete	The call is in the table and is properly subsumed by an incomplete producer.
	no_entry	complete	The call is not in the table, but if it were to be called, it would consume from a completed producer.
		incomplete	The call is not in the table, but if it had been called at this moment, it would consume from an incomplete producer.
		undefined	The call is not in the table, but if it had been called at this moment, it would be a producer.
undefined	undefined	undefined	The given predicate is not tabled.

`get_scc_dumpfile(-Filename)` module: `tables`

If the Prolog flag `exception_pre_action` is set to `print_incomplete_tables` (its default setting is `none`), then when an exception is thrown, incomplete tables and their SCC information are printed to an "SCC dumpfile". Note that the information is output for the state of execution where the error was thrown, and so is more informative than an action taken when the error is caught. (No file is generated unless the exception is thrown over at least one incomplete table.) Creation of an SCC dumpfile can triggered by any error condition, rather than by tha more restricted set of tripwire conditions (cf. Section 10.3.4) and so provide a complementary functionality.

This predicate returns the name of the last such file generated and fails if there is no such file. Files are written to the `$XSBDIR/etc` directory with the prefix `scc_dump_`. Users are responsible for removing these files.

Note that XSB backtraces (Section 12.5) provide information about the context in which an exception is thrown, but the SCC dumpfile provides explicit SCC

information along with argument. values for tabled predicates.

Error Cases

- `Filename` is a not a variable
  - `instantiation_error`

6.15.3 Predicates for Table Inspection: Lower-level

In this section, the user should be aware that skeletons that are dynamically created (e.g., by `functor/3`) are located in `usermod` (refer to Section 3.4.7). 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`.

Answers, Returns, and Return Templates

Given a table entry $(S, \mathcal{A}, Status)$, the vector of variables in S is sometimes called the *substitution factor* of S. The order of arguments in the substitution factor corresponds to the order of distinct variables in a left-to-right traversal of S. Each answer in $\mathcal{A}$ substitutes values for the variables in the substitution factor of S; this substitution is sometimes called an *answer substitution*. The table inspection predicates allow access to substitution factors and answer substitutions through a family of terms called *return templates* and whose principle functors have the form `ret/n`, where `n` is the size of the substitution factor.

Example 6.15.4 *Let* $S =$ `p(X,f(Y))` *be a tabled subgoal. Using a return template, the substitution factor can be depicted as* `ret(X,Y)`*, while the answer substitution* `{X=a,Y=b}` *is depicted as* `ret(a,b)`*. Note that the application of the answer substitution to the generator subgoal yields the answer* `p(a,f(b))`*.*

To take a slightly more complex example, consider the subgoal `q(X)` *where* `X` *is an attributed variable whose attribute is* `f(Z,Y,Y)`*. In this case the substitution factor is* `ret(X,Z,Y)`*.* □

XSB overloads return templates to maintain substitutions between generator subgoals and consuming subgoals when call subsumption is used. The return template for a consuming subgoal is a substitution that maps variables of its generator to subterms of the consuming subgoal. This template can then be used to select answers from the generator that unify with the consuming call.

Example 6.15.5 *Let* `p/2` *of the previous example be evaluated using call subsumption and let the subgoal* $S =$ `p(A,f(B))` *be present in its table. Further, let* S_1*:* `p(A,f(B))` *and* S_2*:* `p(g(Z),f(b))` *be two consuming subgoals of* S*. Then the* return template *of* S_1 *is* `ret(A,B)` *and that of* S_2 *is* `ret(g(Z),b)`*.* S_1*, being a variant of* S*, selects answers for* S *such that* `{X=A,Y=B}`*, i.e., all answers of* S*.* S_2*, on the other hand, selects only* relevant *answers of* S*, those that satisfy* `{X=g(Z),Y=b}`*.* □

Description of Low-level Inspection Predicates

`get_call(+CallTerm,-TableEntryHandle,-ReturnTemplate)` Tabling

If call variance is used for the predicate corresponding to `CallTerm`, then this predicate searches the table for an entry whose subgoal is a *variant* of `CallTerm`. If subsumption is used, then this predicate searches for some entry that subsumes (properly or not) `CallTerm`. In either case, should the entry exist, then the handle to this entry is assigned to the second argument, while its return template is constructed in the third argument. These latter two arguments must be uninstantiated at call time. Example 6.15.6 illustrates its behavior.

Error Cases

- `CallTerm` is not a callable term
  - `type_error(callable_term,CallTerm)`
- `CallTerm` does not correspond to a tabled predicate
  - `permission_error(table access,non-tabled predicate,CallTerm)`

Example 6.15.6 (get_call/2)

Variant Predicate

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

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

Y = _h92
Ent = 136039108
Ret = ret(_h92);

no
| ?- get_call(p(X,3),Ent,Ret).

X = _h84
Ent = 136039156
Ret = ret(_h84);

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

no
```

Subsumptive Predicate

```
| ?- get_call(q(X,Y),Ent,Ret).

X = _h80
Y = _h94
Ent = 136043988
Ret = ret(_h80,_h94);

no
| ?- get_call(q(a,Y),Ent,Ret).

Y = _h88
Ent = 136069412
Ret = ret(a,_h88);

no
| ?- get_call(q(X,c),Ent,Ret).

X = _h80
Ent = 136069444
Ret = ret(_h80,c);

no
```

`get_calls(#Subgoal,-TableEntryHandle,-ReturnTemplate)` Tabling

Identifies through backtracking each tabled subgoal S that unifies with `Subgoal`. For each such S, the handle to the table entry is assigned to the second argument, and its return template is constructed in the third. These latter two arguments must be uninstantiated at call time. The error terms are the same as for `get_calls/1`. Example 6.15.7 illustrates its behavior.

Example 6.15.7 (get_calls/3)

Variant Predicate

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

X = _h80
Y = 3
Ent = 136039156
Ret = ret(_h80);

X = 1
Y = _h94
Ent = 136039108
Ret = ret(_h94);

no
| ?- get_calls(p(X,3),Ent,Ret).

X = _h80
Ent = 136039156
Ret = ret(_h80);

X = 1
Ent = 136039108
Ret = ret(3);

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

Ent = 136039156
Ret = ret(1);

Ent = 136039108
Ret = ret(3);

no
```

Subsumptive Predicate

```
| ?- get_calls(q(X,Y),Ent,Ret).

X = a
Y = _h94
Ent = 136069412
Ret = ret(a,_h94);

X = _h80
Y = c
Ent = 136069444
Ret = ret(_h80,c);

X = _h80
Y = _h94
Ent = 136043988
Ret = ret(_h80,_h94);

no
| ?- get_calls(q(a,Y),Ent,Ret).

Y = _h88
Ent = 136069412
Ret = ret(a,_h88);

Y = c
Ent = 136069444
Ret = ret(a,c);

Y = _h88
Ent = 136043988
Ret = ret(a,_h88);

no
```

`get_returns(+TableEntryHandlex,#ReturnTemplate)` Tabling

Backtracks through the answers for the subgoal whose table entry is referenced through the first argument, `TableEntryHandle`, and instantiates `ReturnTemplate` with the variable bindings corresponding to the answer.

The supplied values for the entry handle and return skeleton should be obtained from some previous invocation of a table-inspection predicate such as `get_call/3` or `get_calls/3`. Its behavior is illustrated in Example 6.15.8.

Example 6.15.8 get_returns/2

Variant Predicate

```
| ?- get_calls(p(X,3),Ent,Ret),
     get_returns(Ent,Ret).

X = 2
Ent = 136039156    % p(X,3)
Ret = ret(2);

X = 1
Ent = 136039156
Ret = ret(1);

X = 1
Ent = 136039108    % p(1,Y)
Ret = ret(3);

X = 1
Ent = 136039108
Ret = ret(3);

no
```

Subsumptive Predicate

```
| ?- get_calls(q(a,c),Ent,Ret),
     get_returns(Ent,Ret).

Ent = 136069412    % q(a,Y)
Ret = ret(a,c);

Ent = 136069444    % q(X,c)
Ret = ret(a,c);

Ent = 136043988    % q(X,Y)
Ret = ret(a,c);

no
| ?- get_calls(q(c,a),Ent,Ret),
     get_returns(Ent,Ret).

no
```

`get_returns_and_tvs(+TableEntryHandle,#ReturnTemplate,-TruthValue)` Tabling

Identical to `get_returns/2`, but also obtains the truth value of a given answer, setting `TruthValue` to `t` if the answer is unconditional and to `u` if it is conditional. If a conditional answer has multiple delay lists, this predicate will succeed only once, so that using this predicate may be more efficient than `get_residual/2` (although less informative).

`get_returns(+TableEntryHandle,#ReturnSkeleton,-ReturnHandle)` Tabling

Functions identically to `get_returns/2`, but also obtains a handle to the answer given in the second argument.

6.15.4 Abolishing Tables and Table Components

The following predicates are used to *abolish* tables: to ensure that they are not used by new computations and to reclaim their space when it is safe to do so. The use of the word "tables" in this section is rather unspecific. For the purpose of deletion a table can either refer to a single subgoal and its answers, or to all subgoals and answers for a tabled predicate. Predicates are provided to abolish tables not only for particular predicates and subgoals, but for all tabled predicates, all tabled predicates in a module, and in the multi-threaded engine all thread-private tabled predicates or all thread-shared tabled predicates. Overall, these predicates share similar characteristics.

Abolishing a Table that is being Computed An incomplete tabled subgoal S may not be directly abolished by the user. This restriction is made since if S is incomplete there may be pointers to S from various elements of the current execution environment, and removing all of these pointers may be difficult to do. (Not to mention that abolishing an incomplete table has a murky semantics.) Accordingly, calling an `abolish_xxx` predicate on an incomplete table raises an error.

However, note that incomplete tables may be abolished *automatically* by XSB on exceptions, and when the interpreter level is resumed. Because tabled computation is more complex than Prolog computation, error handling must be correspondingly more complex. Suppose an exception is thrown over some incomplete table, so that the system looks for some `catch/3` or similar call that will catch the error. In order to ensure safe space reclamation, XSB looks for the catcher C that is nearest to the throw, but is also *between* SCCs. Both XSB's command line interpeter and the interpreter XSB uses when embedded in a process use a top-level `catch/3` goal, which is considered to be "between" SCCs, so that a thrown error will eventually be caught.

Because of the complexity of error handling in tabled computations, it is usually best to ensure that user-level catches are close to where an exception may be thrown so that there is no goal to an incomplete table between the thrower and catcher. In such a case XSB's error handling mechanism conforms to the ISO standard for Prolog.

View Consistency and Table Garbage Collection If one of the table abolish predicates is called when the current execution environment contains a failure continuation (i.e., a choice point) to an answer A in a *completed* table

T, space for T is not immediately reclaimed. Rather the space for T will be reclaimed by the *table garbage collector* at a later point. More precisely, if the current global tabling envonment (including suspended states) has either

- a choice point that points to an answer A in T;
- or a (heap) delay list that points to a subgoal S in T

we say that T is *active*. Also, since tables can be abolished and rederived during the course of an evaluation, the table deletion system marks the tables with versions. Accordingly, if a tabled predicate $P_{version}$ or subgoal $S_{version}$ to be abolished is active in the current environment, reclamation of space for that version of P or S will be delayed until no answers for $P_{version}$ or $S_{version}$ are active. Meanwhile the older version of table will be available for backtracking, ensuring view consistency for the choice points. New calls to P or S, however, will force rederivation of a new table version, rather than using the abolished information.

Maintenance of the Residual Program When conditional answers are present, abolishing a specific table or call may lead to semantic or implementational complications. Consider the conditional answer `r(a,b):- undef|` from Figure 6.2. If the predicate `r/2` (or subgoal `r(a,X)`) is abolished and later rederived, the rederivation of `r(a,X)` might have different semantics than the original derivation (e.g. if `undef` depended on a database predicate whose definition has changed). From an implementation perspective, if space for `r(a,X))` is reclaimed, then the call `get_residual(p(a,X),Y)` may core dump, even if there are no choice points for completed tables anywhere in the choice point stack. To address this problem, by default abolishing a subgoal S (predicate P) will abolish all subgoals (predicates) that (transtively) depend on S (P) [24]. In this case the goal `abolish_table_call(r(a,X))` would cause the deletion of `p(a,X)` while the goal `abolish_table_pred(r/2)` would cause the deletion of `p/2`, since there are tabled subgoals of `p/2` that depend on `r/2`. Only dependencies from subgoals or answers to the answers that are conditional on them are taken into account for table deletion: thus the deletion `r(a,X)` deletes `p(a,X)`, but not `undef`.

Users with programs that give rise to conditional answers in completed tables are encouraged to maintain this default behavior. However the default behavior may be changed either by setting a Prolog flag:

[24] Dao Tran Minh contributed to implementing this functionality.

Program

```
:- table p/2, r/2.
p(X,Y):- r(X,Y).

r(a,b):- undef.
r(a,c):- undef.
r(a,d):- undef.
r(a,e):- undef.

:- table  s/0, t/0.
s:- tnot(t).

t:- tnot(undef).

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

Table

Subgoal	Answer Set	Status
p(a,X)	p(a,b):- r(a,b)| p(a,c):- r(a,c)|	complete
p(b,X)	p(b,d):- r(b,d)| p(b,d):- r(b,e)|	complete
r(a,X)	r(a,b):- undef| r(a,c):- undef|	complete
r(b,X)	r(b,d):- undef| r(b,d):- undef|	complete
s	s:- tnot(t)|	complete
t	t:- tnot(undef)|	complete
undef	undef:- tnot(undef)|	complete

Figure 6.2: Example for Deleting Tables (Call-Variance)

```
?- set_prolog_flag(table_gc_action,abolish_tables_singly).
```

or by calling a 2-ary abolish command with `abolish_tables_singly` in the options list.

Abolishing Incremental Tab;es In XSB, incremental tables react to changes in underlying dynamic predicates and/or external events (cf. Section 5.6). To support this, XSB maintains an Incremental Dependency Graph (IDG) among incrementally tabled subgoals and incremental dynamic predicates. When an incremental table T is abolished, the IDG needs to be restructured. Fortunately, with lazy incremental tabling as used by XSB, the only maintenance needed for the IDG outside of T is to delete direct links between other IDG tables and T. In addition, all tables that depend on T are incrementally invalidated. As a result, if some T' which had previously depended on T is called after T was abolished, T' will be seend to be incrementally invalid and will be recomputed. This recomputation will re-insert T into the IDG in a manner that reflects the new state of the program.

Multiple Threads In the multi-threaded engine abolishing tables private to a thread behaves exactly as in the sequential engine, regardless of whether the tables are complete or incomplete, or contain conditional answers. In addition, when a thread T exits (by normal termination or via an exception), tables private to T are abolished automatically and their space reclaimed, as are any incomplete shared tables owned by T in local evaluation. Shared tables can be abolished by the user at any time, but their space will not be reclaimed until there is a single active thread.

Table Deletion Predicates

`abolish_table_pred(+Pred)` Tabling

Invalidates all tabled subgoals for the predicate denoted by the predicate or term indicator `Pred`. If any subgoal for `Pred` contains an answer A that is active in the current enviornment, `Pred` space reclamation for the `Pred` tables will be delayed until A is no longer active; otherwise the space for the `Pred` tables will be reclaimed immediately.

If `Pred` has a subgoal that contains a conditional answer, the default behavior will be to transitively abolish any tabled predicates with subgoals having answers that depend on any conditional answers of S. This default may be changed either by setting a Prolog flag:

```
?- set_xsb_flag(table_gc_action,abolish_tables_singly).
```

or by calling `abolish_table_pred/2` with the appropriate option. If the transitive abolishes are turned off, and `Pred` contains a conditional answer, the warning

```
abolish_table_pred/[1,2] is deleting a table with conditional answers:
delay dependencies may be corrupted.
```

will be issued.

In the multi-threaded engine, if `Pred` is shared, reclamation for `Pred` will be delayed until there is a single active thread and no answer in `Pred` is active in the current execution environment. Otherwise, the behavior of `abolish_table_pred/1` is the same as in the sequential engine.

Finally, `abolish_table_pred/1` will throw an error if the predicate to be abolished is incremental. Until `abolish_table_pred/[1,2]` is extended to support incremental tables, use `abolish_table_call/[1,2]` or `abolish_all_tables/0`.

Error Cases

- `Pred` is not instantiated
  - `instantiation_error`
- `PredSpec` is not a predicate_indicator or a term_indicator
  - `domain_error(predicate_or_term_indicator,Pred)`
- `PredSpec` does not indicate a tabled predicate
  - `table_error`
- `PredSpec` indicates an incrementally tabled predicate.
  - `permission_error`
- There is currently an incomplete table for an atomic subgoal of `Pred`.
  - `permission_error`

`abolish_table_pred(+CallTerm,+Options)` Tabling

Behaves as `abolish_table_pred/1`, but allows the default `table_gc_action` to be over-ridden with a flag, which can be either `abolish_tables_transitively` or `abolish_tables_singly`.

Error Cases Error cases are the same as `abolish_table_pred/1` but with the additions:

- `Options` is a variable, or contains a variable as an element
  - `instantiation_error`
- `Options` is not a list
  - `type_error(list,Options)`
- `Options` contains an option `O` that is not a table abolish option.
  - `domain_error([abolish_tables_transitively, abolish_tables_singly,O)`

`abolish_table_subgoals(+Subgoal)` Tabling

Invalidates the table for any subgoal that unifies with `Subgoal`. If a subgoal S unifying with `Subgoal` contains an answer A that is active in the current enviornment, the table entry for S will not be reclaimed until A is no longer active; otherwise the space for S will be reclaimed immediately.

If S contains a conditional answer, the default behavior will be to transitively abolish any subgoals that depend on any conditional answers of S. This default may be changed either by setting an XSB flag:

```
?- set_xsb_flag(table_gc_action,abolish_tables_singly).
```

or by calling `abolish_table_call/2` with the appropriate option. If the transitive abolishes are turned off, and S contains a conditional answer, the warning

`abolish_table_call/1 is deleting a table with conditional answers: delay dependencies may be corrupted.`

will be issued.

In the multi-threaded engine, if S is a subgoal for a predicate that is shared, reclamation for S will be delayed until there is a single active thread and no answer in S is active in the current execution environment. Otherwise, the behavior of `abolish_table_call/1` is the same as in the sequential engine on tabled predicates that are thread-private.

For incremental tables, `abolish_table_call/[1,2]` not only deletes the table structures for `Subgoal`, but pointers to `Subgoal` in the Incremental Dependency Graph (IDG), after invalidating all subgoals that depend on `Subgoal`. The node and edges for `Subgoal` will be reinserted into the IDG when `Subgoal` is re-evaluated, either lazily or by an explicit update command.

Error Cases

- The term spec `Subgoal` does not correspond to a tabled predicate:
  - `table_error`
- The term spec `Subgoal` unifies with a tabled subgoal that is incomplete:
  - `permission_error`
- The term spec `Subgoal` is a cyclic term::
  - `table_error`

`abolish_table_subgoals(+Subgoal,+Options)` Tabling

Behaves as abolish_table_subgoals/1, but allows the default `table_gc_action` to be over-ridden with a flag, which can be either `abolish_tables_transitively` or `abolish_tables_singly`.

Error Cases Error cases are the same as `abolish_table_call/1` but with the additions:

- `Options` is a variable, or contains a variable as an element
  - `instantiation_error`
- `Options` is not a list
  - `type_error(list,Options)`
- `Options` contains an option `O` that is not a table abolish option.
  - `domain_error([abolish_tables_transitively, abolish_tables_singly,O)`

`abolish_table_subgoal(+Subgoal)` Tabling

`abolish_table_subgoal(+Subgoal,+Options)` Tabling

These predicates behave as `abolish_table_subgoals/[1,2]`. However rather than abolishing all tables whose subgoal *unifies* with `Subgoal` they only abolish the table whose subgoal is a *variant* of `Subgoal`, if such a table exists.

`abolish_all_tables` Tabling

In the single-threaded engine, removes all tables presently in the system and frees all the memory held by XSB for these structures. Predicates that have been declared tabled remain so, but information in their table is deleted. `abolish_all_tables/0` works directly on the memory structures allocated for table space. This makes it very fast for abolishing a large amount of tables, and to maintain its speed it throws an error if the current execution enviornment contains any incomplete tables, or any active completed tables. `abolish_all_tables/0` can be used regardless of whether there are incremental tables, or tables that use call or answer subsumption.

In the multi-threaded engine `abolish_all_tables/0` additionally raises an error unless it is called when there is a single active thread. In that case, all shared tables are abolished as well as all private tables for the main thread.

Error Cases

- There are incomplete tables at the time of the predicate's call;
  - `permission_error`
- The current execution environment has an active completed table T
  - `permission_error`
- (Multi-threaded engine only) More than one thread is active:
  - `table_error`

`abolish_nonincremental_tables` Tabling

Abolishes all tabled calls for predicates that are *not* declared to be incremental. [25] This predicate allows XSB to function in a manner similar to that of a deductive database: incremental tables will be automatically updated when the data they depends on changes; while non-incremental tables, which may have become invalid, can be abolished. As currently implemented, `abolish_nonincremental_tables/1` traverses through each nonincremental tabled predicate, *Pred*, and if *Pred* has any incomplete subgoals, a permission error will be thrown. However, unlike

[25] Calls for predicates that are declared as opaque are considered to be non-incremental.

with `abolish_all_tables/0` no errors will be thrown if there are active completed tables: rather these tables will be marked for deletion and their space later garbage collected. In addition, no error will be thrown if there are incomplete incremental subgoals.

Error Cases

- There are incomplete nonincremental tables at the time of the predicate's call;
  - `permission_error`

`abolish_all_private_tables` Tabling

In the multi-threaded engine, removes all tables private to the thread and frees all the memory held by XSB for these structures, including space for conditional answers. Predicates that have been declared tabled remain so, but information in their table is deleted. Like `abolish_all_tables/0`, `abolish_all_private_tables/0` works directly on the memory structures allocated for table space. This makes it very fast for abolishing a large amount of tables, and to maintain its speed it throws an error if the current execution enviornment contains any incomplete tables, or any active completed tables. `abolish_all_private_tables/0` can be used regardless of whether there are incremental tables, or tables that use call or answer subsumption.

Error Cases

- There are incomplete tables at the time of the predicate's call;
  - `permission_error`
- The current execution environment for the thread has an active private table T for the current thread
  - `table_error`

`abolish_all_shared_tables` Tabling

In the multi-threaded engine, removes all tables private to the thread and frees all the memory held by XSB for these structures, including space for conditional answers. Predicates that have been declared tabled remain so, but information in their table is deleted. `abolish_all_private_tables/0` works directly on the memory structures allocated for table space. This makes it very fast for abolishing a large amount of tables, and to maintain its speed it throws an error if the current execution enviornment contains any incomplete tables, or any active completed tables. `abolish_all_private_tables/0` can be used regardless of whether there are incremental tables, or tables that use call or answer subsumption. In addition, `abolish_all_shared_tables/0`

raises an error unless it is called when there is a single active thread. If called with a single active thread, all shared tables are abolished, but private tables for the main thread are unaffected.

Error Cases

- There are incomplete tables at the time of the predicate's call;
  - `permission_error`
- The current execution environment has an active table T
  - `permission_error`
- More than one thread is active:
  - `table_error`

`abolish_module_tables(+Module)` Tabling

Given a module name (or the default module, `usermod`), this predicate abolishes all tables for each tabled predicate in `Module`. It is implemented using a series of calls to `abolish_table_pred/1` and so inherits the behavior of that predicate.

`gc_tables(-Number)` Tabling

When a tabled subgoal or predicate is abolished, reclamation of its space may be postponed if the subgoal or predicate has an answer that is active in the current environment. A garbage collection routine is called at various points in execution to check which answers are active in the current environment, and to reclaim the space for subgoals and predicates with no active answers. In particular, space for all abolished tables is reclaimed whenever the engine re-executes the main command-line or C thread interpreter code. However in rare situations this strategy may not be adequate. For this reason, the user can explicitly call the table garbage collector to reclaim space for any deleted tabled predicates or subgoals that no longer have active answers.

`gc_tables/1` always succeeds, unifying `Number` to -1 if garbage collection was not attempted (due to multiple active threads) and otherwise to the number of tables still unreclaimed at the end of garbage collection.

Error Cases

- `Number` is not a variable
  - `type_error(variable)`

`delete_return(+TableEntryHandle,+ReturnHandle)` Tabling

Removes the answer indicated by `ReturnHandle` from the table entry referenced by `TableEntryHandle`. The value of each argument should be obtained from some previous invocation of a table-inspection predicate.

This predicate is low-level so no error checking is done. In Version 3.8, this predicate does not reclaim space for deleted returns, but simply marks the returns as invalid.

Warning: While useful for purposes such as tabled aggregation, `delete_return/2` can be difficult to use, both from an implementation and semantic perspective.

`invalidate_tables_for(+DynamicPredGoal,+Mode)` Tabling

Note that using incremental tabling provides a simpler and much more powerful approach to maintaining dependencies of tables on dynamic code. `invalidate_tables_for/2` *should only be used in cases where incremental tabling is not available (e.g., subsumptive tabling).*

This predicate supports invalidation of tables. Tables may become invalid if dynamic predicates on which they depend change, due to asserts or retracts. By default XSB does not change or delete tables when they become invalid; it is the user's responsibility to know when a table is no longer valid and to use the `abolish_table_*` primitives to delete any table when its contents become invalid.

This predicate gives the XSB programmer some support in managing tables and deleting them when they become invalid. To use this predicate, the user must have previously added clauses to the dynamic predicate, `invalidate_table_for/2`. That predicate should be defined to take a goal for a dynamic predicate and a mode indicator and abolish (some) tables (or table calls) that might depend on (any instance of) that fact.
`invalidate_tables_for(+DynamicPredGoal),+Mode` simply backtracks through calls to all unifying clauses of
`invalidate_table_for(+DynamicPredGoal,+Mode)`. The `Mode` indicator can be any term as long as the two predicates agree on how they should be used. The intention is that `Mode` will be either 'assert' or 'retract' indicating the kind of database change being made.

Consider a simple example of the use of these predicates: Assume the definition of tabled predicate `ptab/3` depends on dynamic predicate `qdyn/2`. In this case, the user could initially call:

```
:- assert((invalidate_table_for(qdyn(_,_),_) :-
                abolish_table_pred(ptab(_,_,_)))).
```

to declare that when `qdyn/2` changes (in any way), the table for `ptab/3` should be abolished. Then each time a fact such as `qdyn(A,B)` is asserted to, or retracted from, `qdyn/2`, the user could call

```
:- invalidate_table_for(qdyn(A,B),_).
```

The user could use the hook mechanisms in XSB (Chapter 9) to automatically invoke
`invalidate_tables_for` whenever `assert` and/or `retract` is called.

6.15.5 Indexing using Tables

Tables are implemented in XSB using Tries, which provide powerful indexing capabilities. By default every table has a trie-index based on the left-to-right ordering of its arguments. This means that any lookup to a table which is bound on an initial sequence of the tabled predicate's arguments is fully indexed. But if an initial sequence of a table lookup is unbound and only a later argument is bound, indexing can be very poor. XSB allows the user to provide `table_index` declarations to improve indexed access to tables. In fact a `table_index` can be used to provide very powerful and general indexed access to any predicate.

`table_index(+PredSpec,+IndexSpec)` Tabling

`table_index` is a compiler directive that causes the compiler to generate code to create tables to provide the requested indexing and is normally used with subsumptive tables. `PredSpec`, of the form `PredName/Arity`, specifies the predicate to be indexed (and subsumptively tabled.) `IndexSpec` is a list of index specifications. Each index specification is a `+`-term that indicates a set of argument positions for which indexes are required. For example, `[1+2+3, 1, 2+3, 2]` indicates that four indexes are desired; a multiple argument index on arguments 1, 2, and 3; a single argument index on argument 1; a multi-argument index on 2 and 3; and a single argument index on argument 2. In this case, for a call to the indicated predicate, if arguments 1, 2, and 3 are bound, that index will be used; if not, but argument 1 is bound, then that index will be used; if not but arguments 2 and 3 are bound, then that index will be used, and finally if argument 2 is bound, then that index will be used. If none of these situations obtain, then a `table_error` will be thrown.

The declared indexes should describe modes under which the predicate will be called. Since the transformation assumes that the predicate will be called in all indicated modes, it will normally abstract all calls to a more general call. So any call will be abstracted to the call to the base predicate that is bound only on argument positions that appear in *all* indexes. For the example above the first call will result in a completely open call to the base predicate, since the intersection of the four indexes is empty. An index of `0` can be used (as the last index in the index specification) to indicate

that any call is permitted, and this will always ensure the fully open call is made initially. For example an index specification of `[1,0]` causes the first call to be abstracted to the fully open call and the subsumptive table to be completely filled. Then every subsequent call will use that constructed table. A call to a `table_index`-ed predicate that is not of a mode explicitly declared in the index specification may not be optimally indexed.

As mentioned above, a predicate indicated as `table_index`-ed should **not** be declared as (normally) tabled. The indexes are created using subsumptive tables. An attempt is made to use the smallest number of tables as possible, but each index does take (perhaps significant) memory. For example the above four indexes can be accommodated using only two subsumptive tables: with argument orders of `[1,2,3,4]` and `[2,3,1,4]`.

The tables of table-indexed predicates can be abolished by `abolish_table_pred`, which will abolish all the generated subsumptive tables. But the tables cannot be removed by `abolish_table_subgoals/1` and its finer variants. Also these predicates cannot be declared as `incremental`.

Chapter 7

Multi-Threaded Programming in XSB

id with Version 3.0, XSB supports the use of POSIX threads to perform separable computations, and in certain cases to parallelize them. POSIX threads have a simple and clear API, and are available on all Unixes and by using open-source libraries, on Windows as well (see Section 7.8 to configure under Windows). This chapter introduces how to program with threads in XSB through a series of examplesi sections discuss performance aspects of our implementation as well as describing relevant predicates. A general knowledge of multi-threaded programming is assumed, such as can be found in [48, 9].

7.1 Getting Started with Multi-Threading

In Version 3.8 the default configuration of XSB does not include multi-threading. This is partly because multi-threading is new, and despite our efforts, the multi-threaded engine may contain bugs not present in the single-threaded engine. However the main reason is because in Version 3.8, not all libraries and packages have yet been made thread-safe so that not all configurations are supported with multi-threading. Both the XSB-calling-C and the C-calling-XSB interfaces are supported in the multi-threaded engine. All XSB libraries have been ported to the multi-threaded engine *except* the profiling library and the `string` library (which is not yet thread-safe). The packages `ODBC` and `CHR`, `FLORA-2`, and `regmatch` are supported by the multi-threaded engine, but the packages `dbdrivers`, `xpath`, `interprolog`, `smodels`, `perlmatch`, `libwww` and `posix` are not yet fully supported. We note, however that all basic/ISO

Prolog functionality is thread-safe (at least, as far as we know :-).

With this in mind, making the multi-threaded engine is simple: configure and make XSB as in Chapter 3, but include the command `-enable-mt`. When you invoke the newly made configuration of XSB you should see `engine: multi-threading` in the configuration list below the banner rather than `engine: slg-wam` as in the sequential engine.

Hello World for Beginners We naturally start with a program to print "hello world". Within the multi-threaded engine, import `thread_create/2` from the module `thread`, and type the command

```
?- thread_create(writeln('hello world'),Id)
```

you should see something like

```
Id = 1hello world
```

while the output is a little ugly, the "hello world" program does illustrate simple multi-threading at work. The calling thread (i.e. the thread controlling the command-line interpreter which we call T_{prompt}) executes the predicate `thread_create/2` which creates a thread T_{child} and immediately returns with the *XSB thread id* of the created thread. Meanwhile, T_{child} initializes its stacks and other memory areas and executes the goal `writeln('hello world')`. T_{child} and T_{prompt} share most of their process-level information: in particular they share a common I/O stream for standard output, leading to the output above. What is happening may be seen a little more easily by executing the command

```
?- thread_create((sleep(1),writeln('hello world')),Id)
```

In this case the interpreter reports that `F` is bound to a thread id, then about a second later `writeln/1` is executed.

The simple "hello world" program illustrates a couple of points. First, it is easy to create a thread in XSB and have that thread do work. Second, it can be tricky to coordinate actions among threads. We'll explore these two themes in more detail, but first suppose we are determined to extend out multi-threaded program so that it produces good output. One way to do this is to *join* T_{prompt} and T_{child} as follows

```
?- thread_create(writeln('hello world'),Id),
```

```
   thread_join(Id,ExitCode).
hello world

Id = 1
ExitCode = true
```

In this case, as soon as T_{prompt} has issued a command to create T_{child}, it executes `thread_join/2`. This latter predicate makes a system call to the underlying operating system to suspend T_{prompt} until T_{child} has exited. `thread_join/2` returns a status term indicating whether the goal to thread `Id` succeeded, failed, exited with an error term, or was cancelled (in this case `Id` succeeded).

So far, we've introduced a few concepts that have not been fully discussed. First is the concept of an *XSB thread id*: XSB manages up to M active threads using XSB thread ids. The default for M in Version 3.8 is 1024, but M can be reset via the `max_threads` command line option to XSB (cf. Section 3.7). Once XSB is initialized, the maximum number of threads for an XSB session can be obtained at run time via the Prolog flag `max_threads` (cf. Section 6.12). It should be noted that the XSB thread id of a thread is different from the identifier of the underlying Pthread. An XSB thread id is a Prolog term, and unlike POSIX thread ids, XSB thread ids can be compared for equality using unification. The actual form of an XSB thread id, however, is subject to change between versions, so programs should not make use of the exact form of an XSB thread id. In the multi-threaded engine, the XSB thread id of any thread can be queried using the predicate `thread_self/1`.

7.2 Communication among Threads

Example 7.2.1 *Consider the program fragment*

```
:- dynamic p/1.

test:- thread_create(assert(p(1)),_X).
```

If you type the goal `?- test` *and then the goal* `?- p(X)`*, the call* `p(X)` *will fail.*

This illustrates an important point about dynamic and tabled predicates in the multi-threaded engine: by default clauses for a dynamic predicate `p/n` are private to the thread that asserts them; and by default tables created in an evaluation of a goal for `p/n` are private to the thread that evaluates the goal. This behavior contrasts to

that of static code which is always shared between threads. In the example above, to allow `p(1)` to be visible to various threads, `p/1` must be declared to be shared with the following declaration.

```
:- table p/1 as shared.
```

or

```
:- dynamic p/1 as shared.
```

Alternately, dynamic and tabled predicates can be made thread-shared by default by invoking XSB with the command-line argument `-shared_predicates`, in which case a predicate may be declared thread-private through the declaration

```
:- table p/1 as private.
```

or

```
:- dynamic p/1 as private.
```

The ability to share dynamic code between predicates provides an extremely powerful mechanism for threads to communicate. So why does XSB make dynamic predicates thread-private by default? The main reason for this is that if dozens or hundreds of threads are running concurrently, shared dynamic code becomes an expensive synchronization point. Code for shared predicates must be more heavily mutexed than code for private predicates. In the case of dynamic code, XSB does not always immediately reclaim the space of retracted clause, to avoid the possibility of some computation backtracking into a clause that has been reclaimed. Rather, (like most Prologs), XSB may decide to garbage collect the space of the retracted clauses at a later time. While clause garbage collection is simple enough to implement for a single thread, garbage collecting clauses for shared dynamic predicates is difficult to do when multiple threads are active. Accordingly, in Version 3.8, space for shared dynamic clauses is not reclaimed until there is a single active thread. However for *thread-private* dynamic predicates, there is no problem in reclaiming space when multiple threads are active: from the engine's perspective garbage collection is no different than in the sequential case. Thus one set of reasons for making dynamic predicates private by default are based on efficiency [1].

[1] Future versions may offer more powerful garbage collectors for shared predicates.

The second reason for making dynamic predicates thread-private by default is semantic. Suppose thread T_1 starts a tabled computation that depends on the dynamic shared predicate `p/1`. While T_1 is computing the table, thread T_2 asserts a clause to `p/1`. T_1's table is likely to be inconsistent, leading to the problem of *read consistency* of any table that depends on thread-shared dynamic predicates. In Version 3.8, users are responsible for ensuring read consistency of any tables that depend on shared dynamic data. Future versions of XSB are intended to allow more sophisticated mechanisms for read consistency.

Not only can tables depend on thread-shared or thread-private dynamic data, but the tables themselves may be thread-shared or thread-private. Like dynamic code, the declaration `table Predspec as shared` allows sharing of tables for a predicate evaluated with call-variance to be shared among threads [2]. To some extent, tabling considerations for making a predicate thread-shared or thread-private are like those of dynamic code. Thread-private tables require fewer synchronization points overall. The situation for reclaiming space for abolished tables is analogous to reclaiming space for retracted dynamic clauses: the garbage collector treats abolished tables for thread-private predicates as in the sequential case, while space for shared tables is not reclaimed until there is a single active thread. However the precise semantics of how tabling information is shared depends on whether the multi-threaded engine is configured with the default local evaluation or with batched evaluation. As discussed in Chapter 5, local evaluation is so-named because computation always takes place in the SCC most recently created, and no answer is returned outside of an SCC until the SCC has been completely evaluated. Within this scheduling strategy it is not often useful to share answers between tables that have not been completed – as local evaluation would allow these answers to be returned only if the tables were in the same SCC. This leads to a concurrency semantics called *Shared Completed Tables* [54, 55, 57]. Shared Completed Tables can in fact be supported by a relatively simple algorithm for optimistic concurrency control. If goals to two mutually dependent tables $Table_a$ and $Table_b$ are called concurrently by two different threads, $Thread_a$ and $Thread_b$, nothing is done until it is detected that $Table_a$ and $Table_b$ are both incomplete and are contained in the same SCC of the table dependency graph. At that time, one of the threads (e.g. $Thread_a$) takes over recomputation of all tables in the SCC, and when the SCC is completed, any remaining answers are returned to other threads that had invoked goals in the SCC. While $Thread_a$ is completing this computation, $Thread_b$ suspends until the SCC is complete. Thus the semantics of Shared Completed Tables supports concurrency for the well-founded semantics, but only supports the most coarse-grained parallelism.

[2] In Version 3.8, tabled predicates using call-subsumption are always private; an attempt to make such a predicate thread-shared throws an exception.

Batched evaluation, on the other hand, allows answers to be returned outside of an SCC before that SCC has been completed. Concurrency control for batched evaluation is similar to that for local evaluation, except in the following case. Assume as before that $Table_a$, first called by $Thread_a$, and $Table_b$ first called by $Thread_b$ are determined to be in the same SCC, and that $Thread_a$ takes over computation of subgoals in the SCC. Now, $Thread_b$, rather than suspending, may continue work. In particular, $Thread_b$ can return any answers in $Table_b$ that it finds whenever it finds them, regardless of whether they have been produced by $Thread_b$ (before $Thread_a$ took over the SCC) or by $Thread_a$ (afterwards). We call this type of concurrency semantics, *Table Parallelism*. Table Parallelism can be used to program producer-consumer examples, as well as to implement Or- and And- parallelism. Table Parallelism was first introduced in [28], but the mechanism now used for implementing Table Parallelism differs significantly from what was described there. In Version 3.8 of XSB, the implementation of Table Parallelism is experimental: in particular, it does not yet support tabled negation.

As mentioned, for either semantics of shared tables, in Version 3.8, users of thread-shared tables are responsible for ensuring read consistency. Note that, in principle, thread-shared tables may depend on thread-private tables and vice-versa. Either type of table may depend on thread-private or thread-shared dynamic code. In addition, a predicate may be *both* dynamic and tabled, and its clauses and tables may be either thread-private or thread-shared.

7.3 Thread Statuses: Joinable and Detached Threads

So far we have assumed that the goal called in `thread_create/2` terminates normally — by success or failure. But what if a thread throws an error while executing a goal? How long should error information for a thread persist, and how can it be checked?

Our approach relies on the semantics of Pthreads, which can be either *joinable* or *detached*. Within this framework, we consider a thread to be *valid* if it has not yet terminated, or if it is joinable and has not yet been joined. After a joinable Pthread T_{dead} has terminated, status information about T_{dead} persists until some other thread joins it — at which time the information is removed. On the other hand, if T_{dead} is detached, status information is removed as soon as T_{dead} terminates. Reclamation of thread status information may be contrasted to that of thread-specific data structures such as stacks. Upon normal or exceptional termination of T_{dead}, any memory automatically allocated in the process of initializing T_{dead}'s, or executing its goal – including stacks, private dynamic code, private tables is reclaimed. In addition,

any mutexes held by T_{dead}, are released. On the other hand, XSB-specific *status* information about threads follows the Pthread model: by default, error information is available when joining a joinable thread, but not otherwise [3].

Example 7.3.1 *Suppose the goal*

```
?- thread_create(functor(X,Y,Z),F).
```

is executed. By default, this will produce the result

```
X = _h113
Y = _h127
Z = _h141
F = 1++Error[XSB/Runtime/P]: [Instantiation]  in arg 2 of predicate functor/3
```

In fact, the variable bindings are output to `STDOUT`*, while the error message*

```
++Error[XSB/Runtime/P]: [Instantiation]  in arg 2 of predicate functor/3
```

is output to `STDERR`*, and may be redirected. The call*

```
?- thread_join(2,Error).
```

returns

```
Error = exception(error(instantiation_error, in arg 2 of predicate functor/3,
              [[Forward Continuation...,... standard:call/1,... standard:catch/3],
                Backward Continuation...]))
```

In other words, `Error` *is instantiated to a* `exception/1` *structure, containing a standard XSB error term (including backtrace).*

The error term in the above example is one example of a *thread status* term. In XSB, these thread statuses are as follows.

- `running` The thread is still executing

[3] This behavior can, of course, be overridden by embedding goals within `catch/3` and handling errors separately, or simply by adding a default user error handler: see Chapter 12 for details.

- `true` The thread has exited and successfully evaluated its goal.
- `false` The thread has exited and failed its goal.
- `exception(Exception)` The thread has been terminated due to an uncaught exception, represented by the term `Exception` which is a standard XSB error term.
- `cancelled(Exception)` The thread has been terminated due to a thread cancellation, represented by the term `Exception` which is a standard XSB error term.
- `exited(ExitTerm)` The thread has been terminated using the predicate `thread exit/1` with `ExitTerm` as its argument.

Any of these statuses except `running` may be returned by `thread_join/2`. In Prolog, the statuses of exited threads provide much more information than C exit codes.

As with pthreads, XSB threads are created as joinable by default, but can be created as detached using an option in `thread_create/3`. Alternatively, a thread created as joinable can be made detached by `thread_detach/1`. All of the predicates mentioned in this section are fully described in Section 7.9.

7.4 Prolog Message Queues

While Prolog predicates can communicate through shared dynamic code and tables, message queues provide a useful mechanism for one thread to pass a command to another or to synchronize on the return of data. A Prolog message queue contains an arbitrary Prolog Term, and unification may be used to obtain a term from a queue. More specifically, when a producer writes $Term$ into a queue, the term is copied into the queue so that no binding are shared between $Term$ and the producer's stacks. $Term$ may include structures or lists and need not be bound, and any variable bindings within $Term$ are preserved. When a consumer T_{cons} accesses the queue it provides a goal G and traverses the queue until it finds a term in the queue that unifies with G. If T_{cons} finds a term in the queue that unifies with G, it removes it from the queue and continues in its computation. If there is no term in the queue that unifies with G, T_{cons} will suspend until at least one other term is added to the queue. When it awakens it will retraverse the queue from the beginning to find a term that unifies with G [4]. Because of the behavior of message queues, it is usually

[4]Note that this traversal is necessary since the position of T_{cons} may in the queue may not be valid due to the addition and deletion of terms by other threads.

good programming practice to ensure that terms written into the queue will unify with the goals of consumers. This can usually be done by abstracting a consumers goal (say to a variable, `X`) or by splitting one "multiplexed" queue into two separate queues.

A Prolog message queue can be *public* or *private*: a public message queue can have any number of readers and writers. In addition, each thread T also has a private message queue Q_T: any thread can write to Q_T but only T can read from it. The following example illustrates how to use private message queues:

```
test_private:-
     thread_id(Tid),
     thread_create(child(Tid),Id),
     thread_get_message('Mom Im home'(ChildId)),
     thread_send_message(ChildId,'Im in the kitchen'),
     thread_join(Id,_).

child(Parent):-
     thread_self(Id),
     thread_send_message(Parent,'Mom, Im home'(Id)),
     thread_get_message('Im in the kitchen').
```

If `?- test` is called by T_{parent}, it will obtain its own thread id, create a new thread T_{child} to execute `child/1`, wait for a message that T_{child} is operational using `thread_get_message/1`, send a message to T_{child} using `thread_send_message/2` and then wait for T_{child} to terminate. When it is created, T_{child} immediately sends a message to its parent, waits for a message back from its parent, and terminates.

It is illustrative to compare

```
test_public:-
     message_queue_create(Qid)
     thread_create(child(Qid),Id),
     thread_get_message(Qid,'Mom Im home'(ChildQ)),
     thread_send_message(ChildQ,'Im in the kitchen'),
     thread_join(Id,_),
     message_queue_destroy(Qid).

child(ParentQ):-
     message_queue_create(Qid),
     thread_send_message(ParentQ,'Mom, Im home'(Qid)),
```

```
thread_get_message(Qid,'Im in the kitchen'),
message_queue_destroy(Qid).
```

`test_public` is essentially the same program as `test_private`, but uses public message queues, rather than private queues. The public queues must be explicitly created and destroyed, and they are referred to via a queue id (or alias) rather than via a thread id (or alias). Like thread ids, queue ids in XSB are integers, but a user should not depend on their precise form: aliases should be used if a user wants control of queue or thread identifiers.

Thus, apart from who can read from them, private and public message queues have essentially the same behavior. In addition, any queue can be created with a bound, *size* on the number of messages (terms) it contains. If *size* is 0, the queue is taken to be unbounded. If a bounded queue already contains *size* elements, the producer will suspend until one or more elements are removed from the queue. For public queues, a size argument can be passed using the predicate `message_queue_create/2` (See Section 7.9). For private queues, and for public queues created with `message_queue_create/1`, the value for *size* is taken from the settable Prolog flag `max_queue_terms`. The default value for `max_queue_terms` is currently 100.

7.5 Thread Cancellation and Signalling

There may be a number of situations in which it is useful to give one thread the ability to cancel the execution of another thread. Within the semantics of pthreads, this is called *thread cancellation*. At the C level, thread cancellation can be tricky, as mutexes must be released, allocated memory freed, and so on. Accordingly, the predicate `thread_cancel/1` cancels XSB threads by acting purely within the SLG-WAM engine. When thread T_1 interrupts thread T_2, T_1 writes to the thread-specific XSB interrupt vector in T_2. Later, when T_2 checks its interrupt vector, it throws a cancellation error, which causes it to clean up its mutexes, memory, private tables and dynamic code, and then exit.

Thread cancellation is just a special case of Prolog thread signalling, in which one thread can signl another thread to interrupt what it is doing and execute a goal [5]. The following code provides an example of thread signalling.

```
test_signal:-
```

[5] Prolog thread signalling should be distinguished from signalling at the OS level where functions such as `pthread_kill()` or `kill()` are used.

```
     thread_self(Tid),
     thread_create(child(Tid),T1,[]),
     thread_get_message('Im alive'),
     thread_signal(T1,writeln('Excuse me, but did you just kick me?')),
     thread_join(T1,_Ball),
     writeln(test5_ok).

child(Tid):-
     thread_send_message(Tid,'Im alive'),
     loop.

loop:- loop.
```

`test_signal` begins like `test_private`, but rather than waiting for a signal from its parent, the child goes into an infinite loop. The signal interrupts the child, which writes out a message and returns to the infinite loop.

Thread signals may be any callable Prolog term. As with private message queues, each thread is created with its own private signal queue (there are no public signal queues). In XSB, threads handle Prolog signal interrupts (including cancellation messages) at the same time as attributed variable interruptions. This means that Prolog signal interrupts will be handled very quickly if SLG-WAM code is being executed. On the other hand, if a thread executing a built-in to, e.g. waiting on a mutex, the thread may be immediately awakened to process the signal, but not always: if a thread is waiting for input on a stream or socket, the thread may not handle the signal interrupt until the input is received. Furthermore, in a very few critical sections of code, thread signal handling may be distabled. However, the thread is guarenteed to handle the signal interrupt or cancellation message very shortly after it finishes the built-in.

So, while thread cancellation and signalling is useful, it must be used with a certain amount of care. Any thread can signal any other thread, and any thread can cancel any other thread, with the exception that the *main* thread, which controls the console (or interface to C or interprolog) cannot be cancelled. The main thread always has XSB thread id `0` in both the single-threaded and multi-threaded systems, and has the thread alias `main`.

7.6 Performance and other Considerations

For running programs that do not use multiple threads, the multi-threaded engine has a minimal overhead compared to the single-threaded engine. Times for single-threaded execution of Prolog or tabled programs range from about 10–20% slower to 10–20% *faster* for the multi-threaded engine compared to the single-threaded engine. Speedups for running multiple threads on multiple processors depends heavily on the applications run and on the underlying operating system.

The size of a given thread may be a consideration for multi-threaded applications, especially on a 32-bit platform (the multi-threaded engine has been tested on both 32-bit and 64-bit platforms). Each thread has an area of thread-private variables that are "global" to its own virtual machine. This area, called the *thread context*, which accounts for about 4 Kbytes of space. Much larger are the various stacks used by the threads for tabled and Prolog execution. Almost all of XSB's memory areas are fully expandable, and the initial size of the execution stacks may be set explicitly as options in `thread_create/3`. Explicitly setting a default thread stack size for an XSB thread to be smaller than the default process stack size may be useful for applications that have a large number of concurrently running threads.

Other performance considerations involve the contention by threads for shared resources. As discussed above, contention may arise when creating or abolishing tables, or when asserting or retracting dynamic code — however in either case thread-private predicates give rise to less contention than thread-shared predicates. In terms of I/O, each XSB stream up to the maximum number of file descriptors has its own mutex; as a result threads writing to different streams will not contend for I/O. Thus, in multi-threaded applications, it may be more efficient to open and close streams and access these streams explicitly, than to redirect standard input or standard output through `see/1` and `tell/1`.

7.7 Examples of Multi-Threaded Programs in XSB

Figure 7.1 shows an example of a multi-threaded goal server in XSB, which makes use of XSB's socket library (see Volume 2 of this manual) [6]. The server listens for requests from clients using `socket_accept/2` and spawns a thread to handle each request via the goal `accept_client/2` which actually calls the goals. The goals executed by the server could be tabled and take advantage of the shared table implementation, shared dynamic code, or any other mechanism in XSB. Halting of the server is done by the

[6] Material in this section is based on [54].

thread cancellation mechanism, and a shared dynamic predicate is used to make the server's thread identifier known to the other threads. Note that this is the reason a specific thread was created to execute `server_loop`, as the main thread cannot be canceled.

Figure 7.2!la uses a multi-threaded execution model to compute a series of prime numbers in parallel [7], The master thread partitions the work and creates two worker threads. The worker threads each compute its portion of the interval and return their results to the master through a message queue.

Notice how the `primes/2` predicate uses difference lists to avoid the use of the append predicate[8], and while threads don't share variables, the bindings of the terms in the messages are correctly handled, allowing Prolog's unification to assume its full power. Although only two threads are used, the program could easily be extended to use an arbitrary number of threads

7.8 Configuring the Multi-threaded Engine under Windows

Libraries for pthreads are included on most versions of Unix and Linux. Windows also supports multi-threading, but with a somewhat different semantics and API than that of pthreads. To run multi-threaded XSB under Windows, a library must be included to translate the Pthread library, used by XSB, to the native thread API of Windows.

Different libraries are available for this purpose. Internally, the multi-threaded engine has been tested using the Win32 pthreads interface, available via `http://sourceware.org/pthreads-w` but other libraries may also work, including Pthread library included with Cygwin. To install the sourceware library, let `$XSBENV` be the parent directory of `$XSBDIR` the root directory of XSB – i.e. `$XSBENV` is the directory into which XSB is installed.

- Download a version such as pthreads-2005-01-25.exe or later, and extract it into `$XSBENV`
  pthreads. Add `$XSBENV\pthreads\Pre-built\lib` to your system path
- To configure with windows enter the commands:

[7] This example was inspired by a similar example for multi-threaded computation of primes in from Logtalk [59]

[8] For a description on how to program with difference lists see a Prolog programming text, such as[77]).

```
:- dynamic server_id/1 as shared.

server :-
   socket(SockFD),
   socket_set_option( SockFD, linger, SOCK_NOLINGER ),
   xsb_port(XSBport),
   socket_bind(SockFD, XSBport),
   socket_listen(SockFD,Q_LENGTH),
   thread_create( server_loop(SockFD), Id, [] ),
   assert( server_id(Iden) ),
   thread_join( Iden ).

server_loop(SockFD) :-
   socket_accept(SockFD, SockClient),
   thread_create( attend_client(SockClient) ),
   server_loop(SockFD).

attend_client(SockClient) :-
   socket_recv_term(SockClient, Goal),
   (  Goal == stop ->
      retract(server_id( Server )),
      thread_cancel( Server ),
      socket_close( SockClient ),
      thread_exit
   ;  true
   ),
   (  is_valid(Goal) ->
      call(Goal),
      socket_send_term(SockClient, Goal),
      fail,
   ;  socket_send_term(SockClient, invalid_goal(Goal))
   ),
   socket_send_term(SockClient, end),
   socket_close(SockClient).
```

Figure 7.1: A multi-threaded goal server in XSB

```
prime(P, I) :- I < sqrt(P),!.
prime(P, I) :- Rem is P mod I, Rem = 0, !, fail.
prime(P, I) :- I1 is I - 1, prime(P, I1).

prime(P) :- I is P - 1, prime(P, I ).

list_of_primes(I, F, Tail, Tail) :- I > F, !.
list_of_primes(I, F, [I|List], Tail) :-
    prime(I), !,
    I1 is I + 1, list_of_primes(I1, F, List, Tail).
list_of_primes(I, F, List, Tail) :-
    I1 is I + 1, list_of_primes(I1, F, List, Tail).

partition_space(N, H, H1) :-
    H is N//2, H1 is H + 1.

worker( Q, Iden, I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    thread_send_message( Q, primes(Iden,List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    message_queue_create(Q),
    thread_create( worker(Q, p1, 1, H, L, L1) ),
    thread_create( worker(Q, p2, H1, N, L1, []) ),

    thread_get_message( Q, primes(p1,L,L1) ),
    thread_get_message( Q, primes(p2,L1,[]) ).
```

Figure 7.2: A multi-threaded program to calculate prime numbers in XSB

```
sh configure --enable-mt --with-wind \
--with-includes='c:\XSBSYS\XSBENV\pthreads\Pre-built\include \
--with-static-libraries='c:\XSBSYS\XSBENV\pthreads\Pre-built\lib

makexsb_wind
```

Note that the Unix `sh` shell must be available in order to reconfigure.

- To configure with cygwin enter the commands:

```
sh configure --enable-mt \
  --with-includes='/cygdrive/c/XSBSYS/XSBENV/pthreads/Pre-built/include' \
  --with-static-libraries='/cygdrive/c/XSBSYS/XSBENV/pthreads/Pre-built/lib'

sh makexsb --config-tag=mt
```

7.9 Predicates for Multi-Threading

The predicates described in this section do not address tabling or dynamic code. With only a few minor deviations the provisional working standard described in [39] is supported. As a result, these predicates are substantially the same as those in SWI, YAP, and other Prologs. In the single-threaded engine, semantically correct calls to these predicates will give a miscellaneous error.

`thread_create(+Goal,ThreadId,+OptionsList)`

When called from thread T, this predicate creates a new XSB thread T_{new} to execute `Goal`. When goal either succeeds, throws an unhandled error, exits, or fails, T_{new} exits, but `thread_create/2` will succeed immediately, binding `ThreadId` to the XSB thread id of T_{new}. `Goal` must be callable, but need not be fully instantiated. No bindings from `Goal` are passed back from T to T_{new}, so communication between T_{new} and T must be through tables, asserted code, message queues or other side effects.

`OptionList` allows optional parameters in the configuration for the initial size of XSB stacks, for aliases, and to indicate whether T_{new} is to be created as detached. Note that XSB threads allow automatic stack allocation, so that the size options may be most useful for (32-bit) applications with very large numbers of threads. In this case, setting initial stack sizes to be small may allow more threads to be created on a given hardware platform. Also note that

only XSB stacks are affected, the stack size of the underlying Pthread remains unaltered.

- `glsize(N)`: create thread with global (heap) plus local stack size initially set to `N` kbytes. If not specified, the default size is used. The default size can be set at the command line (cf. Section 3.7), and altered at run time by the Prolog flag `thread_glsize` (cf. Section 6.12).
- `tcpsize(N)`: create thread with trail plus choice point stack size initially set to `N` kbytes. If not specified, the default size is used (cf. Section 3.7). The default size can be set at the command line (cf. Section 3.7), and altered at run time by the Prolog flag `thread_tcpsize` (cf. Section 6.12).
- `complsize(N)`: create thread with completion stack size initially set to `N` kbytes. If not specified, the default size is used (cf. Section 3.7). The default size can be set at the command line (cf. Section 3.7), and altered at run time by the Prolog flag `thread_complsize` (cf. Section 6.12).
- `pdlsize(N)`: create thread with `N` kbytes of unification stack. If not specified, the default size is used (cf. Section 3.7). The default size can be set at the command line (cf. Section 3.7), and altered at run time by the Prolog flag `thread_pdlsize` (cf. Section 6.12).
- `detached(Boolean)`: if `Boolean` is true, creates detached thread. If `Boolean` is false, the thread created will be joinable, while if no option is given the default will be used. In Version 3.8 threads are created joinable by default, but this default can be altered at run time by the Prolog flag `thread_default` (cf. Section 6.12).
- `on_exit(Handler)`: Ensures that `Handler` is called whenever the thread exits: whether that exit arises from success of `Goal`, failure, throwing an error that is unhandled in the user's program, or an explicit call to `thread_exit/1`.
- `alias(Alias)`: Allow thread `ThreadId` to be referred to via `Alias` in all standard thread predicates. `Alias` remains active for `ThreadId` until it is joined. Note that the main XSB thread has alias `main`.

Finally, each thread is created with a signal queue and a private message queue, so these queues do not need to be explicitly created. Their size is obtained through the settable Prolog flag `max_queue_terms`.

Error Cases

- `Goal` is a variable

  - instantiation_error.
- Goal is not callable
  - type_error(callable,Goal).
- ThreadId is not a variable
  - type_error(variable,ThreadId)
- OptionList is a partial list or contains an option that is a variable
  - instantiation_error
- OptionList is neither a list nor a partial list
  - type_error(list,OptionsList)
- OptionList contains an option, Option not described above
  - domain_error(thread_option,Option)
- An element of OptionsList is alias(A) and A is already associated with an existing thread, queue, mutex or stream
  - permission_error(create,alias, A)
- An element of OptionsList is alias(A) and A is not an atom
  - type_error(atom,A)
- An element of OptionsList is on_exit(Handler) and Handler is not callable
  - type_error(callable,Handler).
- No more system threads are available (EAGAIN)
  - resource_error(system threads)

thread_create(+Goal,-ThreadId)
: Acts as thread_create(Goal,ThreadId,[]).

thread_create(+Goal)
: Acts as thread_create(Goal,_,[detached(true)]).

thread_join(+Threads_or_aliases,-ExitDesignators)
: When thread_join/2 is called by thread T, Threads_or_aliases must be instantiated to either 1) an XSB thread id or alias; or 2) a list where each element is an XSB thread id or an alias; ExitDesignators must be uninstantiated. The action of the predicate is to suspend T until all of the threads denoted by Threads_or_aliases have exited. At this time, any remaining resources for the threads in ThreadIds will have been reclaimed. Upon success ExitDesignators

is either a the thread status of the associated thread (see page 332) or a list of such elements.

Error Cases

- `Thread_or_Aliases` is not instantiated
  - `instantiation_error`
- `Threads_or_aliases` is not a list of XSB thread ids or aliases
  - `domain_error(listof(thread_or_alias),ThreadIds)`
- `ExitDesignators` is not a variable
  - `type_error(variable,ExitDesignatorst)`
- `ThreadId` does not correspond to a valid thread
  - `existence_error(valid_thread,ThreadId)`
- `ThreadId` does not correspond to a joinable thread (i.e. `ThreadId` is detached).
  - `permission_error(join,non_joinable_thread,ThreadId)`

`thread_exit(+ExitTerm)`

Exits a thread T with `ExitTerm` after releasing any mutexes held by T, freeing any thread-specific memory allocated for T (we hope), as well as calling any exit handlers for T. `ExitTerm` will be used if the caller of T joins to T, but will be ignored in other cases. There is no need to call this routine on normal termination of a thread as it is called implicitly on success or (final) failure of a thread's goal.

Error Cases

- `ExitCode` is a variable
  - `instantiation_error`

`thread_self(?ThreadId_or_Alias)`

If `ThreadId` is an atom, unifies `ThreadId_or_Alias` with an alias of the calling thread. Otherwise, unifies `ThreadId_or_Alias` with the XSB thread id of the calling thread. There are no error conditions.

`thread_detach(+Thread_or_Alias)`

Detaches a joinable thread denoted by `Thread_or_Alias` so that all resources will be reclaimed upon its exit. The thread denoted by `ThreadId` will no longer be joinable, once it is detached. If `Thread_or_Alias` has already exited, all resources used by `Thread_or_Alias` are removed from the system.

Error Cases

- `Thread_or_Alias` is a variable
  - `instantiation_error`
- `Thread_or_Alias` is not a thread id or alias
  - `domain_error(thread_or_alias,Thread_or_Alias)`
- `Thread_or_Alias` does not correspond to a valid thread
  - `existence_error(valid_thread,Thread_or_alias)`
- `Thread_or_Alias` is active but not joinable
  - `permission_error(thread_detach,thread,Thread_or_Alias)`

`thread_cancel(+Thread_or_Alias)`

Cancels the XSB thread denoted by `Thread_or_Alias`. The cancellation does not use Pthread cancellation mechanisms, rather it uses XSB's interrupt mechanism to set `Thread_or_Alias`'s interrupt vector [9]. When this interrupt vector is checked, `Thread_or_Alias` will throw a thread cancellation error, which can be caught within `Thread_or_Alias` like any other error. However, the default behavior is for `Thread_or_Alias` to exit with an exit ball indicating that it has been cancelled.

As noted above, an executing thread that is cancelled will exit very shortly after the `thread_cancel/1` predicate is called. Blocked threads, however, are not always guarenteed to exit when cancelled. Currently a blocked thread may be cancelled

- when it is waiting to read or write a message on a queue
- when it is executing `thread_sleep/1`

On the other hand, a blocked thread may not be cancelled while it is waiting to read from a stream or waiting for a mutex.

During critical operations a thread may want to prevent itself from being cancelled. This can be done by If `?- thread_cancel(T)` is called for a thread `T` for which cancelling has been disabled, `T` will be cancelled immediately after `T` re-enables cancellation through calling the predicate `thread_enable_cancel/0`.

The main XSB thread cannot be cancelled; apart from that any thread can cancel any other thread.

Error Cases

[9] This interrupt vector is checked upon every it is checked on every SLG-WAM call and execute instruction.

- `Thread_or_Alias` is not instantiated
  - `instantiation_error`
- `Thread_or_Alias` is not a thread id or alias
  - `domain_error(thread_or_alias,Thread_or_Alias)`
- `Thread_or_Alias` does not correspond to valid thread
  - `existence_error(valid_thread,Thread_or_Alias)`
- `Thread_or_Alias` denotes the main thread.
  - `permission_error(cancel,main_thread,Thread_or_Alias)`

`thread_signal(Thread_or_Alias,Goal)`

`thread_signal(ThreadOrAlias, Goal)` interrupts thread `ThreadOrAlias` so that it executes `Goal` at the first opportunity. Specifically, once `Goal` is placed onto the signal queue of `ThreadOrAlias` and the interrupt vector of `ThreadOrAlias` is adjusted, `thread_signal/2` succeeds. `ThreadOrAlias` handles the interrupt asynchronously, and if the interrupt is handled while `ThreadOrAlias` is executing a goal with continuation C, all solutions for `Goal` will be obtained, and the failure continuation of `Goal` will be C. If `Goal` throws an exception E, the continuation will be the handler for E.

For blocked threads, signalling works much like cancellation (described above), and a blocked thread will handle a signal whenever it can be cancelled. However, the thread does not return to the blocking operation *after* the signal – rather it will execute the signal and then execute the continuation to be taken after the blocking operation.

Error Cases

- `Thread_or_Alias` is not instantiated
  - `instantiation_error`
- `Thread_or_Alias` is not a thread id or alias
  - `domain_error(thread_or_alias,Thread_or_Alias)`
- `Thread_or_Alias` does not correspond to valid thread
  - `existence_error(valid_thread,Thread_or_Alias)`
- `Goal` is not instantiated
  - `instantiation_error`
- `Goal` is not callable
  - `type_error(callable,Goal)`

`thread_disable_cancel` module: `thread`

Disables the calling thread from being cancelled, so that it can be ensured that critical operations can run to completion. This predicate always succeeds.

`thread_enable_cancel` module: `thread`

Enables the calling thread to be cancelled. By default, threads may be cancelled, so this predicate needs to be called if `thread_disable_cancel/0` has been previously called. This predicate always succeeds.

`thread_yield`

Make the calling thread ready to be run *after* other threads of the same priority. This predicate relies on the real-time extensions to pthreads specified in POSIX 1b, and may not be available on all platforms.

Error Cases

- The current platform does not support POSIX real-time extensions
  - `misc_error`

`thread_property(?ThreadOrAlias,?Property)`

If `ThreadOrAlias` is instantiated, unifies `Property` with current properties of the thread that unify with `Property`; if `ThreadOrAlias` is a variable, backtracks through all the current threads whose properties unify with `Property`. Note that there is no guarantee that that the information returned will be valid, due to concurrency issues.

Currently `Property` can have the form

- `detached(Bool)`: if `Bool` is true the thread is detached, otherwise it is joinable.
- `alias(Alias)`: if the thread has an alias `Alias`
- `status(Status)`: see Section 7.3 for thread statuses that are currently supported.

Example: The following predicate may be used to clear resources from the thread table, although due to concurreny reasons, non-running threads may remain in the thread table after this predicate terminates.

```
clear_thread_table:-
    thread_property(Tid,status(S)),
    \+ (S = running),
    thread_join(Tid),
```

```
    fail.
clear_thread_table.
```

Error Cases

- `ThreadOrAlias` is neither a variable nor an XSB thread id nor an alias
  - `domain_error(thread_or_alias, ThreadOrAlias)`
- `ThreadOrAlias` is not associated with a valid thread
  - `existence_error(thread, ThreadOrAlias)`

`thread_sleep(+Seconds)`

Causes the calling thread to sleep approximately `Seconds` before resuming. A thread may be cancelled while sleeping. However, a sleeping thread that is signaled will execute the signaled goal and resume execution *without* returning to sleep.

Error Cases

- `Seconds` is a variable
  - `instantiation_error`.
- `Seconds` is not a number
  - `type_error(number, Seconds)`.

7.9.1 Predicates for Thread Synchronization and Communication

Threads can communicate to some extent through shared tables and dynamic code. However, it is often useful to use message queues as a synchronizable form of communication. Similarly, while the XSB engine itself is thread-safe, thread synchronization may be needed when calling a package that is not itself thread safe (see the beginning of this chapter for a list of which packages are and are not thread-safe). Synchronization may also be needed to protect data accessed by foreign function calls, or to coordinate responses to external events.

Prolog Message Queues

As described previously, each thread is created with a private message queue that is readable only by itself. The following predicates are used to communicate using private and public message queues.

`message_queue_create(-Queue,+Options)`

Creates a new public message queue with identifier `Queue`. `Options` allows optional parameters to be passed for the maximum number of terms in the queue, and for aliases of the queue.

- `max_terms(N)`: create queue so that it can contain at most `N` terms before writes to the queue block. If not specified, the default size is used. This default can be queried and altered at run time via the Prolog flag `queue_max_terms`. (cf. Section 6.12). If the flag `queue_max_terms` is set to 0, the queue size will be bounded only by available memory.
- `alias(Alias)`: Allow queue `Queue` to be referred to via `Alias` in all standard queue predicates. `Alias` remains active for `Queue` until it is destroyed.

Error Cases

- Queue is not a variable
  - `type_error(variable,Queue)`
- Options is a partial list or a list with an element that is a variable
  - instantiation error
- Options is neither a partial list or a list
  - type error(list, Options)
- `Options` contains an option, `Option` not described above
  - `domain_error(queue_option,Option)`
- An element of `Options` is `alias(A)` and `A` is already associated with an existing thread, queue, mutex or stream
  - `permission_error(create,alias, A)`
- An element of `Options` is `alias(A)` and `A` is not an atom
  - `type_error(atom,A)`

`message_queue_detroy(+Queue_or_Alias)`

Destroys a public message queue with alias or id `Queue_or_alias`, as created by `message_queue_create/[1,2]`. If any threads are currently waiting on `Queue_or_Alias` to read or write a term, they will be awakened and will throw an existence error.

Error Cases

- `Queue_or_Alias` is a variable

  - instantiation_error
- Queue_or_Alias is not a queue id or alias
  - domain_error(queue_or_alias,Queue_or_Alias)
- Queue_or_Alias denotes a private message queue or signal queue rather than a public message queue
  - permission_error(destroy,private_signal_or_message_queue,Queue_or_Alias)
- Queue_or_alias is not the queue name or alias of a public message queue.
  - existence_error(message_queue, Queue_or_Alias)

thread_send_message(+Queue_or_Alias,#Message)

Queue_or_alias may either be a queue id or alias, or a thread id or alias in which latter case the private queue for a thread is used. If there are fewer terms on Queue_or_Alias than the queue's maximum allowed number thread_send_message/2 puts Message onto Queue_or_Alias, and returns immediately. Otherwise, the calling thread suspends until there are fewer elements on Queue_or_Alias than the queue's maximum allowed number, when the thread will be awakened to put Message onto the queue.

Error Cases

- Queue_or_Alias is a variable
  - instantiation_error
- Queue_or_Alias is not a queue id, queue alias, thread id, or thread alias.
  - domain_error(queue_or_alias,Queue_or_Alias)

thread_get_message(+Queue_or_Alias,?Message)

If there are terms on Queue_or_Alias thread_get_message/2 traverses Queue_or_Alias to obtain the first term T that unifies with Message. If T exists, the predicate returns with Message bound to the most general unifier of Message and T. If there are no terms on Queue_or_Alias or if no terms unify with Message, the calling thread suspends until at least one term is added to Queue_or_Alias. When the thread awakes, it will recheck Queue from its beginning for a term that unifies with Message.

Error Cases

- Queue_or_Alias is a variable
  - instantiation_error
- Queue_or_Alias is not a queue id or alias

  - `domain_error(queue_or_alias,Queue_or_Alias)`
  - `existence error(queue, Queue_or_Alias)`

`thread_get_message(?Message)`
Acts as `thread_get_message/2`, but on a thread's private queue.

`thread_peek_message(+Queue_or_Alias,?Message)`
If there are terms on `Queue_or_Alias` `thread_peek_message/2` traverses `Queue_or_Alias` to obtain the first term T that unifies with `Message`. If T exists, the predicate returns with `Message` bound to the most general unifier of `Message` and T. If there are no terms on `Queue_or_Alias` or if no terms unify with `Message`, the predicate fails.

Error Cases

- `Queue_or_Alias` is a variable
  - `instantiation_error`
- `Queue_or_Alias` is not a queue id or alias
  - `domain_error(queue_or_alias,Queue_or_Alias)`
- `Queue_or_Alias` is not associated with a current queue
  - `existence error(queue, Queue_or_Alias)`

`thread_peek_message(?Message)`
Acts as `thread_peek_message/2`, but on a thread's private queue.

User-defined Mutexes

Usually, running multi-threaded evaluations does not requre a user to set any mutexes – necessary mutexes are handled by XSB itself (we hope), and programs can often be written so that user-level locking is unnecessary. However, under certain conditions, locking is useful or even necessary: for instance, a user may need to set a lock so that a set of shared dynamic facts cannot be accessed when it is updated.

One of the simplest and most powerful primitives for locking are mutexes. The mutexes provided by the following predicates are *recursive*: if a thread T locks a recursive mutex M, any calls to `mutex_lock(M)` made by T will immediately succeed without suspending while M is locked. Other threads that attempt to lock M will suspend until M is unlocked. To unlock M after n calls to `mutex_lock(M)`, T must make n calls to `mutex_unlock(M)`.

When using mutexes in XSB, programmers must not only avoid explicitly creating deadlocks, but must also ensure that a mutex is unlocked when leaving a critical area, and destroyed when it is no longer needed. Making sure that this happens for successful goals, for failed goals and for goals that raise exceptions can sometimes be complicated. The predicate `with_mutex/2` handles all of these cases. We recommend using it if possible, and making use of lower-level calls to `mutex_lock/1`, `mutex_unlock/1` and `mutex_trylock/1` only in rare cases when `with_mutex/2` is not applicable.

`with_mutex(+Mutex,?Goal)`

Locks a current mutex or alias`Mutex`, executes `Goal` deterministically, then unlocks `Mutex`. If `Goal` leaves choice-points, these are destroyed. `Mutex` is unlocked regardless of whether `Goal` succeeds, fails or raises an exception. Any exception thrown by `Goal` is re-thrown after the mutex has been successfully unlocked.

Error Cases

- `Mutex` is a variable
  - `instantiation_error`
- `Mutex` is not a mutex id or alias
  - `domain_error(mutex_or_alias,Mutex_or_Alias)`
- `Mutex` is not associated with a current mutex.
  - `existence_error(mutex,Mutex)`
- Locking `Mutex` would give rise to a deadlock [10]
  - `permission_error(mutex,lock,Mutex)`
- `Goal` is a variable
  - instantiation error
- `Goal` is neither a variable nor a callable term
  - type error(callable, Goal)

`mutex_create(?Mutex)`

Creates a new recursive user mutex with identifier `Mutex`. `Options` allows optional parameters to be passed, currently only for aliases of the mutex.

- `alias(Mutex)`: Allow queue `Mutex` to be referred to via `Mutex` in all standard queue predicates. `Mutex` remains active for `Mutex` until it is destroyed.

[10] This error case handles the `EDEADLK` return code on MacOS X, and other platforms.

Error Cases

- Mutex is not a variable
  - `type_error(variable,Mutex)`
- Options is a partial list or a list with an element that is a variable
  - instantiation error
- Options is neither a partial list or a list
  - type error(list, Options)
- `Options` contains an option, `Option` not described above
  - `domain_error(mutex_option,Option)`
- An element of `Options` is `alias(A)` and `A` is already associated with an existing thread, queue, mutex or stream
  - `permission_error(create,alias, A)`
- An element of `Options` is `alias(A)` and `A` is not an atom
  - `type_error(atom,A)`

`mutex_destroy(+Mutex)`

Destroys a current unlocked mutex with alias or id `Mutex` along with any memory it uses.

Error Cases

- `Mutex` is a variable
  - `instantiation_error`
- `Mutex` is not a mutex id or alias
  - `domain_error(mutex_or_alias,Mutex_or_Alias)`
- `Mutex` is not associated with a current mutex.
  - `existence_error(mutex,Mutex)`
- `Mutex` is locked
  - `permission_error(mutex,destroy,Mutex)`

`mutex_lock(+Mutex)`

`mutex_lock(Mutex)` locks a (recursive) mutex with alias or id `Mutex`. Locking and unlocking mutexes should be paired carefully in order to avoid deadlocks. In particular, a programmer needs to ensure that mutexes are properly unlocked even if the protected code fails or raises an exception.

Error Cases

- `Mutex` is a variable
  - `instantiation_error`
- `Mutex` is not a mutex id or alias
  - `domain_error(mutex_or_alias,Mutex_or_Alias)`
- `Mutex` is not associated with a current mutex.
  - `existence_error(mutex,Mutex)`
- Locking `Mutex` would give rise to a deadlock [11]
  - `permission_error(mutex,lock,Mutex)`

`mutex_trylock(+Mutex)`

Works as `mutex_lock/1` but fails immediately if `Mutex` is held by another thread, rather than suspending the calling thread.

Error Cases

- `Mutex` is a variable
  - `instantiation_error`
- `Mutex` is not a mutex id or alias
  - `domain_error(mutex_or_alias,Mutex_or_Alias)`
- `Mutex` is not associated with a current mutex.
  - `existence_error(mutex,Mutex)`

`mutex_unlock(+Mutex)`

Unlocks the mutex with alias or id `Mutex` when called by the same thread that locked `Mutex`.

Error Cases

- `Mutex` is a variable
  - `instantiation_error`
- `Mutex` is not a mutex id or alias
  - `domain_error(mutex_or_alias,Mutex_or_Alias)`
- `Mutex` is not associated with a current mutex.
  - `existence_error(mutex,Mutex)`
- `Mutex` is not held by the calling thread

[11] This error case handles the `EDEADLK` return code on MacOS X, and other platforms.

- `permission_error(unlock,mutex,Mutex)`

`mutex_unlock_all`

`mutex_unlock_all/0` unlocks all user mutexes owned by the current thread. It has no error cases.

`mutex_property(?MutexOrAlias,?Property)`

If `MutexOrAlias` is instantiated, unifies `Property` with current properties of the mutex; if `MutexOrAlias` is a variable, backtracks through all the current mutexes whose properties unify with `Property`. Note that there is no guarantee that that the information returned will be valid, due to concurrency issues.

Currently `Property` can have the form

- `alias(Alias)`: if the mutex has an alias `Alias`
- `status(Status)`. If the mutex is locked, `Status` will be a term of the form `locked(ThreadId,NumLocks)` where `ThreadId` is the thread id of the owner of the lock, and `NumLocks` is the number of times the mutex has been locked by the current owner (recall that user-defined mutexes are recursive and must be unlocked as many times as they have been locked in order to be freed). If the mutex is unlocked, `Status` will be a term of the form `unlocked`.

Example: The query

```
?- mutex_property(M,status(_)).
```

can be used to enumerate all active user-defined mutexes.

Error Cases

- `MutexOrAlias` is neither a variable nor an XSB mutex id nor an alias
  - `domain_error(mutex_or_alias, MutexOrAlias)`
- `MutexOrAlias` is not associated with an active mutex
  - `existence_error(mutex, MutexOrAlias)`
- `Property` is neither a variable nor a valid mutex property
  - `domain_error(mutex_property, Property)`

Chapter 8

Storing Facts in Tries

XSB offers a mechanism by which large numbers of facts can be directly stored and manipulated in *tries*, which can either be private to a thread or shared among threads. The mechanism described in this chapter is in some ways similar to trie-indexed asserted code as described in Section 6.14, but allows creation of tries that are shared between threads, and of associative tries that support efficient memory management [1].

When stored in a trie, facts are compiled into trie-instructions similar to those used for XSB's tables. For instance set of facts

```
{ rt(a,f(a,b),a), rt(a,f(a,X),Y), rt(b,V,d) }
```

would be stored in a trie as shown in Figure 8, where each node corresponds to an instruction in XSB's virtual machine. Using a trie for storage has the advantage that discrimination can be made on a position anywhere in a fact, and directly inserting into or deleting from a trie is 4-5x faster than with standard dynamic code. In addition, in trie-dynamic code, there is no distinction between the index and the code itself, so for many sets of facts trie storage can use much less space than standard dynamic code. For instance, Figure 8 shows how the prefix `rt(a,f(a,...` is shared for the first two facts. However, trie storage comes with tradeoffs: first, only facts can be stored in a trie; second, unlike standard dynamic code, no ordering is preserved among the facts; and third, duplicate facts are not supported.

In Version 3.8 of XSB, tries that store facts may have the following forms:

[1] For nearly all purposes, the predicates in this chapter replace the low-level API for interned tries in previous versions, which included `trie_intern`, `trie_unintern`, `trie_interned` etc. However that API continues to be supported for low-level systems programming.

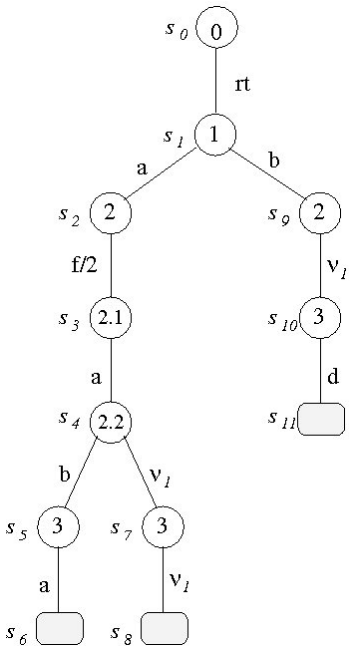


Figure 8.1: Terms Stored as a Trie

- *Private, general* tries allow arbitrary terms to be inserted in a trie. These tries are thread-private so that inserting a term in a trie Tr in one thread will not be visible to another thread. Although such tries are general, they have limitations in memory reclamation in Version 3.8 of XSB. If a term is deleted from Tr, memory will be reclaimed if it is safe to do so at the time of deletion [2]; otherwise the space will not be reclaimed until all terms in Tr are removed by truncating Tr or until the thread exits.

- *Private, associative* Associative tries are more restricted than general tries: an associative trie combines a *key* which can be any ground term, with a *value* which can be any term. Memory for deleted key-value pairs in an associative trie is always immediately reclaimed, and insert or delete operations can be faster for an associative trie than for a general trie. These tries are private to a thread, and in addition to reclaiming memory when a term is deleted, memory is reclaimed when the trie is truncated or dropped, and when the thread exits.

- *Shared, associative* tries are associative tries that are shared among threads. Memory for deleted key-value pairs is always immediately reclaimed, and when the trie is truncated or dropped.

[2] That is, if no choice points are around that may cause backtracking into Tr.

8.1 Examples of Using Tries

A handle for a trie can be obtained using the `trie_create/2` predicate. Terms can then be inserted into or deleted from that trie, and terms can be unified with information in the trie, as shown in the following example:

Example 8.1.1 First, we create a private general trie:

```
| ?- trie_create(X,[type(prge)]).
X = 1

yes
```

Next, we insert some terms into the trie

```
| ?- trie_insert(1,f(a,b)), trie_insert(1,[a,dog,walks]).

yes
```

Now we can make arbitrary queries against the trie

```
| ?- trie_unify(1,X).

X = [a,dog,walks];

X = f(a,b);

no
```

Above, a general query was made, but the query could have been any Prolog term. Now we delete a term, and see what's left.

```
| ?- trie_delete(1,f(X,B)).

X = a
B = b

yes
| ?- trie_unify(1,X).

X = [a,dog,walks];

no
```

The behavior of general tries can be constrasted with that of associative tries as seen in the next example.

Example 8.1.2 Now we start by creating a shared associative trie, with abbreviation `shas` using the multi-threaded engine

```
| ?- trie_create(X,[type(shas),alias(foo)]).
X = 1048577

yes
```

This time we used an alias so now we can use `foo` to refer to insert a couple of key-value pairs into the trie (we could also use the trie handle itself)

```
| ?- trie_insert(foo,pair(sentence(1),[a,dog,walks])),
     trie_insert(foo,pair(sentence(2),[a,man,snores])).

yes
```

However, inserting a general term into an associative trie throws an error

```
| ?- trie_insert(foo,f(a,b)).
++Error[XSB/Runtime/P]: [Domain (f(a,b) not in domain pair/2)]
in arg 2 of predicate trie_insert/2
(Inserted term must be key-value pair in trie 1048577)
```

Finally, in an associative trie, if we insert a value for a key that is already in the trie, it will *update* the value for that key.

```
| ?- trie_insert(foo,pair(sentence(1),[a,dog,snoress])).

yes
| ?- trie_unify(foo,pair(sentence(1),X)).
X = [a,dog,snores]

yes
```

8.2 Space Management for Tries

When creating or adding terms to an interned trie, XSB manages all space necessary for the terms and their indexes. However, when removing a term from a trie an

issue may arise if there is a possibility of backtracking into the term to be removed; this issue also arises for retracting dynamic code. In the sequential engine and in private tries XSB's dynamic clause garbage collector handles space reclamation when terms are removed from a trie through `trie_delete/2` or similar low-level predicates. However, in the case of `trie_truncate/1` or `trie_drop/1`, an exception is thrown if there are active choice points to terms in a trie that is to be truncated or dropped.

In the multi-threaded engine the space reclamation problem becomes even more difficult for tries that can be shared among threads. In this case, no garbage collection is performed until there is a single active thread.

These space reclamation issues arise for non-associative tries only. Associative triesessentially contain key-value pairs, and so may have their space reclaimed upon deletion of a term, or upon truncation or dropping their trie, regardless of the number of active threads [3].

8.3 Predicates for Tries

The following subsections describe predicates for inserting terms into a trie, deleting terms from a trie, and unifying a term with terms in a trie, predicates for creating, dropping, and truncating tries, as well as predicates for bulk insertes into and deletes from a trie. These predicates can apply to any type of trie, and perform full error checking on their call arguments. As such, they are safer and more general than the lower-level trie predicates described in Chapter 1 of Volume 2 of this manual. Use of the predicates described here is recommended for applications unless the need for speed is paramount.

`trie_create(-TrieId,+OptionList)` module: `intern`

`OptionList` allows optional parameters in the configuration of a trie to indicate its type and whether an alias should be used. In the present version, `OptionList` may contain the following terms

- `type(Type)` where `Type` can be one of
  - `prge` (private, general) maintains information that is accessable only to the calling thread. No other restrictions are made for accessing information in a private trie. In the single-threaded engine, tries are private by default.

[3] Future versions of XSB may extend garbage collection to handle trie truncation, trie dropping and better space reclamation in the multi-threaded engine.

  - `pras` (private, associative) creates a private trie that maintains key-value pairs in a manner similar to an associative array, using the term `pair(Key,Value)`. Each key must be ground, and there may be only one value per key.
  - `shas` (shared associative) creates a shared trie that maintains key-value pairs in a manner similar to an associative array, using the term `pair(Key,Value)`. Each key must be ground, and there may be only one value per key. This option is available only in the multi-threaded engine

- `alias(Alias)`: Allow trie `TrieId` to be referred to via `Alias` in all standard trie predicates. `Alias` remains active for `TrieId` until it is dropped.
- `incremental`: Allows tables that depend on trie `TrieId` to be automatically updated as information in `TrieId` changes (cf. Section 5.6.3).
- `nonincremental`: Specifies that tables that depend on trie `TrieId` should not be automatically updated as information in `TrieId` changes (cf. Section 5.6.3).

Error Cases

- `TrieId` is not a variable
  - `type_error(variable,TrieId)`
- `OptionList` is a partial list or contains an option that is a variable
  - `instantiation_error`
- `OptionList` is neither a list nor a partial list
  - `type_error(list,OptionsList)`
- `OptionList` contains an option, `Option` not described above
  - `domain_error(trie_option,Option)`
- An element of `OptionsList` is alias(A) and A is already associated with an existing thread, queue, mutex or stream
  - `permission_error(create,alias, A)`
- An element of `OptionsList` is alias(A) and A its not an atom
  - `type_error(atom,A)`

`trie_insert(+TrieIdOrAlias,Term)` module: `intern`

Inserts `Term` into the trie denoted by `TrieIdOrAlias`. If `TrieIdOrAlias` denotes an associative trie, `Term` must be of the form `pair(Key,Value)` where `Key`

is ground. If `TrieIdOrAlias` is a general trie and already contains `Term`, the predicate fails (as the same term cannot be inserted multiple times in the same trie). Similarly, if `TrieIdOrAlias` is an associative trie and already contains a value for `Key` the predicate fails.

Insertion of tries can be controlled by the flags `max_answer_term_depth`, `max_answer_list_depth`, `max_answer_term_action`, and `max_answer_list_action`, which are also used to control additions of answers to tables. Using these flags, if a term to be inserted is cyclic and exceeds a stated depth, trie insertion may either fail or throw an error depending on the associated action: see pg. 253.

Error Cases

- `TrieIdOrAlias` is a variable
  - `instantiation_error`.
- `TrieIdOrAlias` is not a trie id or alias
  - `domain_error(trie_id_or_alias,TrieIdOrAlias)`
- `TrieIdOrAlias` denotes an associative array, and `Term` does not unify with `pair(_,_)`
  - `domain_error(pair/2,Term)`
- `TrieIdOrAlias` denotes an associative array, `Term = pair(Key,Value)` but `Key` is not ground
  - `misc_error`
- `Key` or `Value` is a cyclic term, or exceeds the depth
  - `misc_error`

`trie_unify(+TrieIdOrAlias,Term)` module: `intern`

Unifies `Term` with a term in the trie denoted by `TrieIdOrAlias`. If `TrieIdOrAlias` denotes a general trie, successive unifications will succeed upon backtracking. If `TrieIdOrAlias` denotes an associative trie, `Term` must be of the form `pair(Key,Value)` where `Key` is ground.

Error Cases

- `TrieIdOrAlias` is a variable
  - `instantiation_error`.
- `TrieIdOrAlias` is not a trie id or alias
  - `domain_error(trie_id_or_alias,TrieIdOrAlias)`

- `TrieIdOrAlias` denotes an associative array, and `Term` does not unify with `pair(_,_)`
  - `domain_error(pair/2,Term)`
- `TrieIdOrAlias` denotes an associative array, `Term = pair(Key,Value)` but `Key` is not ground
  - `misc_error`

`trie_delete(+TrieIdOrAlias,Term)` module: `intern`

Deletes a term unifying with `Term` from the trie denoted by `TrieIdOrAlias`. `TrieIdOrAlias` denotes a general trie, all such terms can be deleted upon backtracking. If `TrieIdOrAlias` denotes an associative trie, `Term` must be of the form `pair(Key,Value)` where `Key` is ground. In either case, if `TrieIdOrAlias` does not contain a term unifying with `Term` the preicate fails.

Error Cases

- `TrieIdOrAlias` is a variable
  - `instantiation_error`.
- `TrieIdOrAlias` is not a trie id or alias
  - `domain_error(trie_id_or_alias,TrieIdOrAlias)`
- `TrieIdOrAlias` denotes an associative array, and `Term` does not unify with `pair(_,_)`
  - `domain_error(pair/2,Term)`
- `TrieIdOrAlias` denotes an associative array, `Term = pair(Key,Value)` but `Key` is not ground
  - `misc_error`

`trie_truncate(+TrieIdOrAlias)` module: `intern`

Removes all terms from `TrieIdOrAlias`, but does not change any of its properties (e.g. the type of the trie or its aliases).

@@

Error Cases

- `TrieIdOrAlias` is a variable
  - `instantiation_error`.
- `TrieIdOrAlias` is not a trie id or alias
  - `domain_error(trie_id_or_alias,TrieIdOrAlias)`

- There are active failure continuations to terms in `TrieIdOrAlias`
  - `miscellaneous_error`

`trie_drop(+TrieIdOrAlias)` module: `intern`

Drops `TrieIdOrAlias`. `trie_drop/1` not only removes all terms from `TrieIdOrAlias`, but also removes information about its type and any aliases the trie may have.

Error Cases

- `TrieIdOrAlias` is a variable
  - `instantiation_error`.
- `TrieIdOrAlias` is not a trie id or alias
  - `domain_error(trie_id_or_alias,TrieIdOrAlias)`
- There are active failure continuations to terms in `TrieIdOrAlias`
  - `miscellaneous_error`

`trie_bulk_insert(+TrieIdOrAlias,+Generator)` module: `intern`

Used to insert multiple terms into the trie denoted by `TrieIdOrAlias`. `Generator` must be a callable term. Upon backtracking through `Generator` its first argument should successively be instantiated to the terms to be interned in `TrieIdOrAlias`. When inserting many terms into a general trie, `trie_bulk_insert/2` is faster than repeated calls to `trie_insert/2` as it does not need to make multiple checks that the choice point stack is free of failure continuations that point into the `TrieIdOrAlias` trie. For associative tries, `trie_bulk_insert/2` can also be faster as it needs to perform fewer error checks on the arguments of the insert.

Example 8.3.1 Given the predicate

```
bulk_create(p(One,Two,Three),N):-
     for(One,1,N),
     for(Two,1,N),
     for(Three,1,N).
```

and a general trie `Trie`, the goal

```
?- trie_bulk_insert(Trie,bulk_create(_Term,N))
```

will add N^3 terms to `Trie`.

Error Cases

- `TrieIdOrAlias` is a variable
  - `instantiation_error`.
- `TrieIdOrAlias` is not a trie id or alias
  - `domain_error(trie_id_or_alias,TrieIdOrAlias)`
- `Generator` is not a compound term
  - `type_error(compound,Generator)`
- `TrieIdOrAlias` denotes an associative array, and `Generator` does not unify with `pair(_,_)`
  - `domain_error(pair/2,Term)`
- `TrieIdOrAlias` denotes an associative array, and `Generator` succeeds with a term that unifies with `pair(Key,Value)` and `Key` is not ground
  - `misc_error`
- `Key` or `Value` is a cyclic term
  - `misc_error`

`trie_bulk_delete(+TrieIdOrAlias,Term)` module: `intern`

Deletes all terms that unify with `Term` from `TrieIdOrAlias`. If `TrieIdOrAlias` denotes an associative trie, the key of the key value pair need *not* be ground.

Example 8.3.2 For the trie in the previous example, the goal

```
?- trie_bulk_delete(Trie,p(1,_,_))
```

will delete the N^2 terms that unify with `p(1,_,_)` from `TrieIdOrAlias`.

Error Cases

- `TrieIdOrAlias` is a variable
  - `instantiation_error`.
- `TrieIdOrAlias` is not a trie id or alias
  - `domain_error(trie_id_or_alias,TrieIdOrAlias)`

`trie_bulk_unify(+TrieIdOrAlias,#Term,-List)` module: `intern`

Returns in `List` all terms in `TrieIdOrAlias` that unify with `Term`. If `TrieIdOrAlias` denotes an associative trie, the key of the key value pair need *not* be ground.

This predicate is useful for two reasons. First, it provides a safe way to backtrack through an associative trie while maintaining the memory management and concurrency properties of associative tries. Second, it enforces read consistency for `TrieIdOrAlias`, regardless of whether the trie is private or shared, general or associative.

Example 8.3.3 Continuing from Example 8.3.2 the goal

```
?- trie_bulk_unify(Trie,X),List
```

will return the the $N^3 - N^2$ terms still in `TrieIdOrAlias`.

Error Cases

- `TrieIdOrAlias` is a variable
  - `instantiation_error`.
- `TrieIdOrAlias` is not a trie id or alias
  - `domain_error(trie_id_or_alias,TrieIdOrAlias)`
- `List` is not a variable
  - `type_error(variable,List)`.

`trie_property(?TrieOrAlias,?Property)` module: `intern`

If `TrieOrAlias` is instantiated, unifies `Property` with current properties of the trie; if `TrieOrAlias` is a variable, backtracks through all the current tries whose properties unify with `Property`. In the MT engine, `thread_property/2` accesses only tries private to the calling thread and shared tries; however note that there is no guarantee that that the information returned about shared tries will be valid, due to concurrency issues [4].

Currently `Property` can have the form

- `type(Type)`: where `Type` is the type of the trie.
- `alias(Alias)`: if the trie has an alias `Alias`

Error Cases

[4] `trie_property/2` is not yet implemented for shared tries.

- `TrieOrAlias` is neither a variable nor an XSB trie id nor an alias
  - `domain_error(trie, TrieOrAlias)`
- `TrieOrAlias` is not associated with a valid trie
  - `existence_error(trie, TrieOrAlias)`

8.4 Low-level Trie Manipulation Utilities

The previous sections indicate how tries can be used as an efficient mechanism to store thread-private and thread-shared terms. In this section we describe lower-level trie manipulation predicates that are suitable for implemeting XSB libraries [5]. As with other tries, these utilities are suitable for storing terms rather than exectuable clauses, use a set based semantics, and do not maintain an ordering among these terms. In addition

- These predicates create and maintain thread-private, general tries.
- These predicates do not always perform error checking. If not explicitly specified in the description of the predicate, errors returned may be confusing, and calling with improper arguments may even cause memory violations.
- For historical reasons, the ordering of arguments in these predicates is not consistent.

Despite (and sometimes because of) these limitations, the trie manipulation facilities can be extremely fast, so that interning and uninterning terms in a trie may be much faster than assert and retract in XSB or in any other Prolog.

8.4.1 A Low-Level API for Interned Tries

`new_trie(-Root)` module: `intern`

`Root` is instantiated to a handle for a new private, general trie.

`trie_intern(+Term,+Root)` module: `intern`
`trie_intern(+Term,+Root,-Leaf,-Flag,-Skel)` module: `intern`

`trie_intern/2` effectively asserts `Term` by interning into the trie designated by

[5] Flora-2, XASP, XSB's `storage` library and others use these predicates.

Root. If a variant of Term is already in Root the predicate succeeds, but a new copy of Term is not added to the trie.

trie_intern/5 acts as trie_intern/2 but returns additional information: Leaf is the handle for the interned Term in the trie. Flag is 1 if the term is "old" (already exists in the trie); it is 0, if the term is newly inserted. Skel represents the collection of all the variables in Term. It has the form ret(V1,V2,...,VN), exactly as in get_calls (see Vol. 1 of the XSB manual).

Error Cases

- Root is uninstantiated
  - instantiation_error
- Root is instantiated, but not an integer (trie handle)
  - type_error(integer,Root)

trie_interned(?Term,+Root) module: intern
trie_interned(?Term,+Root,?Leaf,-Skel) module: intern

trie_interned/2 backtracks through the terms that unify with Term and that are interned into the trie represented by the handle Root. Term may be free, or partially bound.

If Leaf is a free variable, trie_interned/5 works as trie_interned/2: it backtracks through the terms that unify with Term interned into the trie represented by the handle Root. In addition it returns Leaf as the handle for each such term and returns in Skel the collection of all the variables in Term using the form ret(V1,...,Vn). Otherwise, if Leaf is bound, trie_interned/5 will unify Term with the term in the trie designated by Leaf, returning a vector of variables in Skel.

Error Cases

- Root is uninstantiated
  - instantiation_error
- Root is instantiated, but not an integer (trie handle)
  - type_error(integer,Root)

trie_unintern(+Root,+Leaf) module: intern
trie_unintern_nr(+Root,+Leaf) module: intern

trie_unintern(+Root,+Leaf) deletes a term from a trie using the handle Leaf, as obtained from trie_intern/[2,4] or trie_interned/[2,4]. Space is reclaimed for the term only if it is safe to do so – if there are no failure continuations that may consume the term (cf. Section 8.2).

`trie_unintern_nr/2` does not perform space reclamation and as a result requires no garbage collection – it simply marks a term as "deleted". This makes `trie_unintern_nr/2` suitable if trie garbage collection may be an issue, and also allows it to be used in libraries that support backtrackable updates, such as XSB's `storage` library.

Error Cases

- `Root` or `Leaf` is uninstantiated
  - `instantiation_error`
- `Root` or `Leaf` is instantiated, but not an integer (trie handle or trie leaf)
  - `type_error(integer,Root)` or `type_error(integer,Leaf)`

`reclaim_uninterned_nr(+Root)` module: `intern`

Runs through the chain of leaves of the trie `Root` and deletes the terms that have been marked for deletion by `trie_unintern_nr/2`. This can be viewed either as a garbage collection step or as a commit.

Error Cases

- `Root` is uninstantiated
  - `instantiation_error`
- `Root` is instantiated, but not an integer (trie handle)
  - `type_error(integer,Root)`

`unmark_uninterned_nr(+Root,+Leaf)` module: `intern`

The term pointed to by `Leaf` should have been previously marked for deletion using `trie_unintern_nr/2`. This term is then "unmarked" (or undeleted) and becomes again a normal interned term.

Error Cases

- `Root` or `Leaf` is uninstantiated
  - `instantiation_error`
- `Root` or `Leaf` is instantiated, but not an integer (trie handle or trie leaf)
  - `type_error(integer,Root)` or `type_error(integer,Leaf)`

`delete_trie(+Root)` module: `intern`

Deletes all the terms in the trie pointed to by `Root`. Garbage collection ensures that space reclamation is performed only if it is safe to do so.

Error Cases

- `Root` is uninstantiated
  - `instantiation_error`
- `Root` is instantiated, but not an integer (trie handle)
  - `type_error(integer,Root)`
- Failure continuations point to one or more nodes in the trie with root `Root`
  - `misc_error`

Chapter 9

Hooks

Sometimes it is useful to let the user application catch certain events that occur during XSB execution. For instance, when the user asserts or retracts a clause, etc. XSB has a general mechanism by which the user program can register *hooks* to handle certain supported events. All the predicates described below must be imported from `xsb_hook`.

9.1 Adding and Removing Hooks

A hook in XSB can be either a 0-ary predicate or a unary predicate. A 0-ary hook is called without parameters and unary hooks are called with one parameter. The nature of the parameter depends on the type of the hook, as described in the next subsection.

`add_xsb_hook(+HookSpec)` module: `xsb_hook`

This predicate registers a hook; it must be imported from `xsb_hook`. `HookSpec` has the following format:

```
hook-type(your-hook-predicate(_))
```

or, if it is a 0-ary hook:

```
hook-type(your-hook-predicate)
```

For instance,

```
:- add_xsb_hook(xsb_assert_hook(foobar(_))).
```

registers the hook `foobar/1` as a hook to be called when XSB asserts a clause. Your program must include clauses that define `foobar/1`, or else an error will result.

The predicate that defines the hook type must be imported from `xsb_hook`:

```
:- import xsb_assert_hook/1 from xsb_hook.
```

or `add_xsb_hook/1` will issue an error.

`remove_xsb_hook(+HookSpec)` module: `xsb_hook`

Unregisters the specified XSB hook; imported from `xsb_hook`. For instance,

```
:- remove_xsb_hook(xsb_assert_hook(foobar(_))).
```

As before, the predicate that defines the hook type must be imported from `xsb_hook`.

9.2 Hooks Supported by XSB

The following predicates define the hook types supported by XSB. They must be imported from `xsb_hook`.

`xsb_exit_hook(_)` module: `xsb_hook`

These hooks are called just before XSB exits. You can register as many hooks as you want and all of them will be called on exit (but the order of the calls is not guaranteed). Exit hooks are all 0-ary and must be registered as such:

```
:- add_xsb_hook(xsb_exit_hook(my_own_exit_hook)).
```

`xsb_assert_hook(_)` module: `xsb_hook`

These hooks are called whenever the program asserts a clause. An assert hook must be a unary predicate, which expects the clause being asserted as a parameter. For instance,

```
:- add_xsb_hook(xsb_assert_hook(my_assert_hook(_))).
```

registers `my_assert_hook/1` as an assert hook. One can register several assert hooks and all of them will be called (but the order is not guaranteed).

`xsb_retract_hook(_)` module: `xsb_hook`

These hooks are called whenever the program retracts a clause. A retract hook must be a unary predicate, which expects as a parameter a list of the form `[Head,Body]`, which represent the head and the body parts of the clause being retracted. As with assert hooks, any number of retract hooks can be registered and all of them will be called in some order.

Chapter 10

Debugging and Profiling

10.1 Prolog-style Tracing and Debugging

XSB supports a version of the Byrd four-port debugger for interactive debugging and tracing of Prolog code. In this release (Version 3.8), it does not work very well when debugging code involving tabled predicates [1]. If one only creeps (see below), the tracing can provide some useful information. For programs that involve large amounts of tabling forest-view tracing can be used (Section 10.2). To turn on tracing, use `trace/0`, `trace/1`, or `trace/2`. To turn tracing off, use `notrace/0`.

`trace`

`notrace`

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).

When debugging interactively, a message may be printed and tracer stopped 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.)

[1] The current version of XSB's Prolog debugger does not include exceptions as a debugging port.

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.

`trace(+Filename,+option)`

`trace/2` is like `trace/0` except that it is non-interactive and dumps trace information into a log file, `Filename`. Currently the only supported option is `log`. However, the log is written in the form of Prolog facts, which can be loaded queried. The format of the facts is:

```
xsb_tracelog(CallId,CallNum,PortType,ParentCallNum,DepthOfCall,CurrentCall,Time)
```

where `CallId` is an identifier generated when XSB encounters a new top-level call. This identifier remains the same for all subgoals called while tracing that top-level call.

- `CallNum` is a generated number to show the nesting of the calls being traced. It is the same number that the user sees when tracing interactively.
- `PortType` is `'Call'`, `'Redo'`, `'Exit'`, or `'Fail'`.
- `ParentCallNum` is the call number of the parent call.
- `DepthOfCall` is the nesting depth of the current call with respect to its ancestor calls.
- `CurrentCall` is the call being traced
- `Time` is the CPU time it took to execute `CurrentCall`. On `'Call'` and `'Redo'`, `Time` is always 0 — it has a meaningful value only for the `'Exit'` and `'Fail'` log entries.

It should be noted that when calls are delayed due to the well-founded negation computation of because of the `when/2` primitive, the parent call might be off in some cases. However, the parent property repairs itself for subsequent calls.

'The name of the predicate (`xsb_tracelog`) used for logging can be changed by asserting it into the predicate `debug_tracelog_predicate/1`, which should be imported from `usermod`. For instance,

```
:- import debug_tracelog_predicate/1 from usermod.
?- assert(debug_tracelog_predicate(foobar)).
```

`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 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/0`, `trace/1`, `trace/2`, 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.

10.1.1 Control of Prolog-Style Tracing and Debugging

XSB debugger also provides means for the low-level control of what must be traced. Normally, various standard 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.

Finally, XSB also provides a facility for low-level tracing of Prolog execution. This can be activated by invoking the emulator with the `-T` option (see Section 3.7),

or through the predicate `trace/0`. It causes trace information to be printed out at *every* Prolog call (including those to system predicates, and tabled predicates). While this method can occasionally be useful, its use is limited. For tabled executions the techniques in the following sections are much more appropriate. Ever for Prolog programs, the volume of such trace information can become very large very quickly, so this method of tracing is only recommended for situations where no other debugging method is useful.

10.2 Trace-based Execution Analysis through Forest Logging

The tracing and debugging described in previous sections has proven useful for Prolog programs for 30 or more years. However, when tabling is added to Prolog, things change. First, as described in Chapter 5, tabling can be used to find the least fixed point of mutually recursive predicates. Operationally, this requires the ability to suspend one computation path and to resume another. Second, the addition of tabled negation for the well-founded semantics requires the ability to delay negative goals whose only proof may be involved in a loop through negation and to simplify these goals once their truth value has become known. Furthermore, a tabled subgoal has different states: it may be *new*; it may be *incomplete* so that new answers might be derived for it; or *completed* (completely evaluated) so that the answers may simply be read from the table. In short, tabling, which can execute much more general programs than Prolog and which can use the stronger well-founded semantics, requires a more complex set of operations than Prolog's SLDNF. Accordingsly, debugging and tracing is correspondingly more complex. Thus, while Prolog's 4-port debugger may be useful for programs that involve just a few tabled predicates, it may not be useful for programs that heavily use tabling for complex recursions, non-monotinic reasoning or other purposes.

There is currently no standard approach to debugging tabled programs. One possible approach would be to extend the 4-port debugger to include other ports for tabling operations. Such extensions have not yet been explored, and whether the paradigm of n-port debugging can be extended to full tabling so that it can be useful to programmers is an open question. Another approach would be use the declarative approach of *justification* [35, 61] to explain why derivations were or were not made. XSB does in fact have a justification package but it is not currently robust enough to be recommended for general use. Below we present the `logforest` approach [83]

10.2.1 Tracing a tabled evaluation through forest logging

While the operations used for tabling are more complex than those of SLDNF, they have a clear formal operational semantics through SLG and the forest-of-trees model. We recall this model briefly below for a definite program but assume a background knowledge of tabled logic programming (see, for instance [85]).

Example 10.2.1 Figure 10.1 shows a program fragment along with an SLG forest for the query `?- reach(1,Y)` to the the right-recursive tabled predicate `reach/1`. An SLG forest consists of an SLG tree for each tabled subgoal S: this tree has root S :- S. In a definite program an SLG tree represents resolution of program clauses and answers to prove S. In Figure 10.1 each non-root node of the form $K.N$ where $N = (S$:- $Goals)\theta$ is a clause in which the bindings to a subgoal S are maintained in $S\theta$, the goals remaining to prove S are in $Goals\theta$, and the order of creation of N within the tabled evaluation is represented by a number, K (local scheduling is used in this example). Children of a root node are obtained through resolution of a tabled subgoal against program clauses. Children of non-root nodes are obtained through answer clause resolution, if the left most selected literal is tabled (e.g. children of node 3 or 11 in the tree for `reach(1,Y)`), or through program clause resolution if the leftmost selected literal is not tabled (e.g. children of nodes 2 and 18 in the tree for `reach(1,Y)`). Nodes that have empty *Goals* are termed *answers*. Note that the evaluation keeps track of each tabled subgoal S that it encounters. Later if S is selected again, resolution will use answers rather than program clauses; if no answers are available, the computation will *suspend* at that point and the evaluation will backtrack to try to derive answers using some other computation path. Once more answers have been derived, the evaluation *resumes* the suspended computation. Similarly, once the computation has backtracked through all answers available for S in the current state, the computation path will suspend, and resume after further answers are found. Thus a tabled evaluation is a fixed point computation for a set of interdependent subgoals. When it is etermined that a (perhaps singleton) set of subgoals can produce no more answers, the subgoals are completed.

The forest logging approach (`logforest`) allows one to run a tabled query and produce a log that can be interpreted as (a partial image of) an SLG forest. The log can then used to analyze program correctness, to optimize performance and so on. Because `logforest` produces a log, it superficially resembles the non-interactive trace described earlier in this chapter. However,

- `trace/1` produces a Prolog-style trace that takes little account of tabling. `logforest` structures its output according to the forest-of-trees model, and

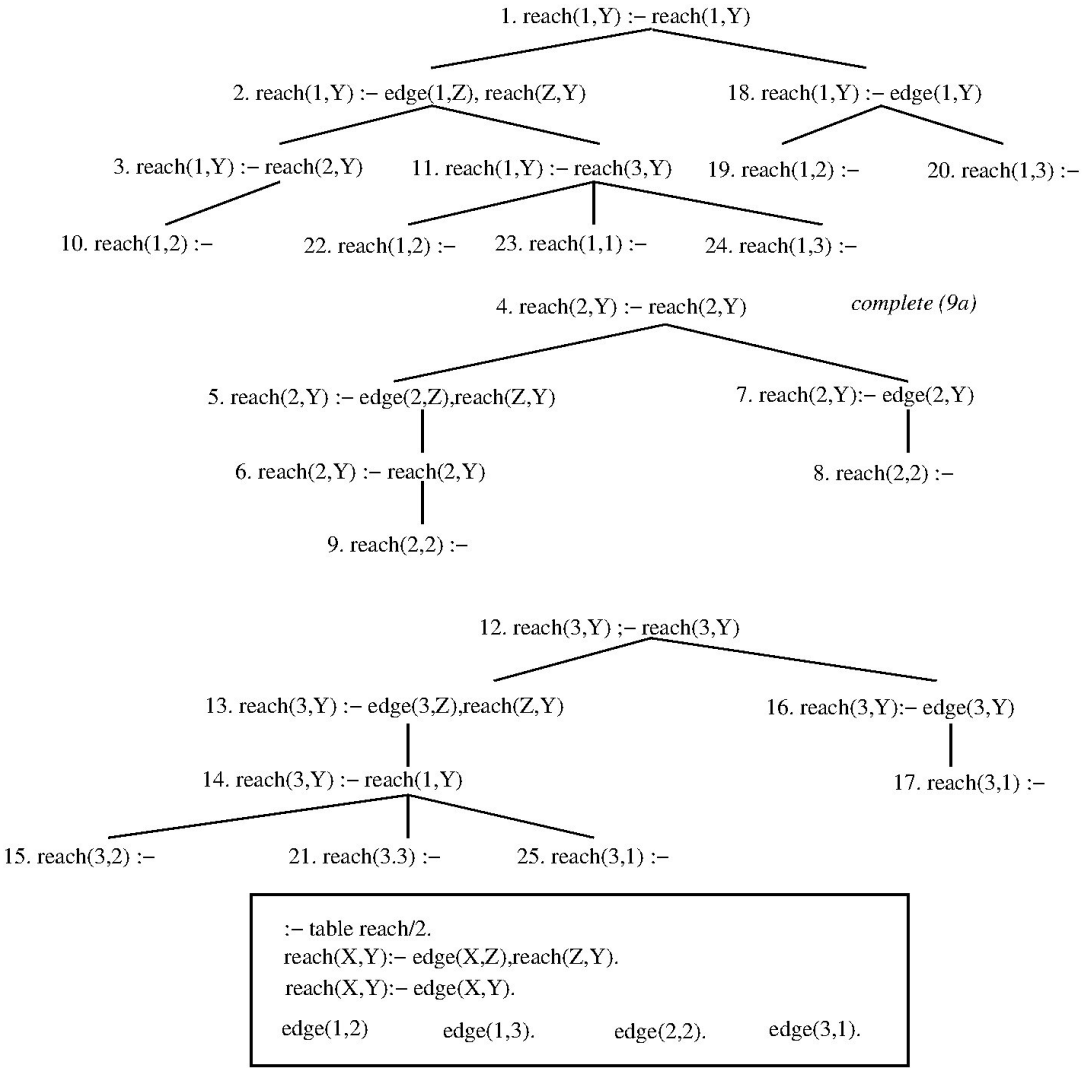


Figure 10.1: A program P_{Rrec} and SLG forest for (local) evaluation of `?- reach(1,Y)`

takes little account of program clause resolution.

- `logforest` is implemented in C for efficiency, while `trace/1` is built on top of XSBs interactive debugger. Unlike `trace/1`, `logforest` can therefore to produce logs for very large evaluations with little overhead.

Currently, `logforest` captures the following actions.

- *A call to a tabled subgoal* If a positive call to a tabled subgoal S_1 is made from a tree for S_2 a Prolog-readable fact of the form `tc(S1,S2,Stage,Counter)` is logged, where *Counter* is the ordinal number of the fact, and `Stage` is
  - `new` if S_1 is a new subgoal
  - `cmp` if S_1 is not a new subgoal and has been completed
  - `incmp` if S_1 is not a new subgoal but has *not* been completed
  - `reeval` if S_1 is an incremental subgoal being re-evaluated.

  If the call is negative a fact of the form `nc(S1,S2,Stage,Counter)` is logged, where all arguments are as above.

  For instance, in the above example, node 3 would be represented as `tc(reach(2,Y),reach(1,Y),2)` (the reason for using the counter value of 2 rather than 3 is explained below). If S_1 is the first tabled subgoal in an evaluation, S_2 is the atom *null*.

- *Derivation of a new answer* When a new *unconditional* answer A is derived for subgoal S and added to the table (i.e. A is not already an answer for S) a fact of the form `na(A,S,Counter)` is logged. In the above example, the answer node 9 would be represented as `na([2],reach(2,_v1),4)` where the first argument is a list of substitutions for the variables *_v1,...,_vn* in S.

  When a new *conditional* answer A :- $D|$, with substitution A and delayed literals D. is derived for subgoal S and added to the table a fact of the form `nda(A,S,D,Counter)` is logged.

- *Return of an answer to a consuming subgoal* When an unconditional answer A is returned to a consuming subgoal S in a tree for S_T, a fact of the form `ar(A,S,ST,Counter)` is logged. A log entry is made only if the table for S is incomplete (see the explanation below).

  If the answer A is conditional, the fact has the form `dar(A,S,ST,Counter)`, where each argument is as above.

- *Delaying a selected negative literal.* If a selected negative literal L of a node N is delayed, because it is involved in a loop through negation, and N is in a tree for S_T, a fact of the form `dly(L,`S_T`,Counter)` is logged.

- *Subgoal completion*

  - When a set $\mathcal{S}$ of subgoals is determined to be completely evaluated and is completed, a fact of the form `cmp(S,SCCNum,Counter)` is logged for each $S \in \mathcal{S}$. Here $SCCNum$ is simply a number giving an ordinal value that can be used to group subgoals into mutually dependent sets of subgoals (here called *Strongly Conneced Components* or *SCCs*), i.e. the *SCCNum* of each $S \in \mathcal{S}$ has the same value, but that value is not used for a completion fact of any subgoal not in $\mathcal{S}$.
  - When a subgoal $\mathcal{S}$ is *early completed*, i.e. it is determined that no more answers for S are possile or are desired a fact of the form `cmp(S,ec,Counter)` is logged. If S belonged to a larger mutually dependent set $\mathcal{S}$ when it was early completed, S will also be included in the completion facts for $\mathcal{S}$.

- *Table Abolishes*

  - When a tabled subgoal S is abolished, a fact of the form `ta(subg(S),Counter)` is logged.
  - When all tables for a predicate p/n are abolished, a fact of the form `ta(pred(p/n),Counter)` is logged.
  - When all tables are abolished, a fact of the form `ta(all,Counter)` is logged.

- *Location of errors* Whenever an error is thrown and the execution is in a tree for a subgoal S, a Prolog-readable fact of the form `err(S,Counter)` is logged, where *Counter* is the ordinal number of the fact. The primary purpose of this fact is to indicate the nearest tabled call that gave rise to an uncaughterror.

`logforest` does *not* contain

- Information about the occurrence of program clause resolution either when used to produce children of tabled predicates, or when it is used to produce children whose nodes have a selected literal that is non-tabled.

- Information about the return of answers from completed tables. XSB uses a so-called *completed table optimization* which treats answer return from completed tables in a manner akin to program clause resolution.

The inclusion of the above two features in `logforest` would significantly slow down execution of XSB. However, future versions of `logforest` may include expanded logging features for negation, for call and answer subsumption and for incremental tabling [2].

Example 10.2.2 *The forest for* `reach(1,Y)` *in the foregoing example has the log file as shown in Table 10.1.*

`log_forest(+Call)` module: `tables`
`log_forest(+Call,+Options)` module: `tables`

These predicates turn on forest logging, call `Call`, then turn logging off when `Call` is finished. `Options` is a list of possible options.

- `Options` may contain the term `file(File)` which directs the logging to `File`; otherwise the log will be sent to standard output.
- `Options` may contain the term `level(Level)` where `Level` may be one of the following values.
  - `full` which means that all tabling actions are logged as described above.
  - `partial` which means that answer return operations are not logged.
  - `calls_only` which does not log answer return, new answer, nor simplification operations.

  The levels `partial` and `calls_only` both reduce the size of the log which can be useful for analyzing some computations.
- `Options` may containt the term `set_pred(PredSpec,Mode)` where `Mode` is `on` or `off`. This allows certain predicates not to be logged, a useful feature if only part of a program needs to be debuged

Error Cases

- `Options` is a variable, or contains a variable as an element
  - `instantiation_error`
- `Options` is not a list
  - `type_error(list,Options)`
- `Options` contains an option `O` that is not a forest logging option.
  - `domain_error(forest_logging_option,O)`

[2] Currently, attributes of attributed variables are not printed out.

Log File	Forest	Explanation
tc(reach(1,_v0),null,new,0)	node 1	
	node 2	created by program clause resol.
	node 3	created by program clause resol.
tc(reach(2,_v0),reach(1,_v0),new,1)	node 4	
	node 5	created by program clause resol.
	node 6	created by program clause resol.
tc(reach(2,_v0),reach(2,_v0),incmp,2)		repeated subgoal registered
	node 7	created by program clause resol.
	node 8	created by program clause resol.
na([2],reach(2,_v0),3)	node 8	registered as answer
ar([2],reach(2,_v0),reach(2,_v0),4)	node 9	created by answer resol.
cmp(reach(2,_v0),2,5)	9a	`reach(2,_v0)` completed
	node 10	created by return from completed table
na([2],reach(1,_v0),6)	node 10	registered as an answer
	node 11	created by program clause resol.
tc(reach(3,_v0),reach(1,_v0),new,7)	node 12	
	node 13	created by program clause resol.
	node 14	created by program clause resol.
tc(reach(1,_v0),reach(3,_v0),incmp,8)	node 14	repeated subgoal registered
ar([2],reach(1,_v0),reach(3,_v0),9)	node 15	created by answer resol.
na([2],reach(3,_v0),10)	node 15	registered as an answer
	node 16	created by program clause resol.
	node 17	created by program clause resol.
na([1],reach(3,_v0),11)	node 17	registered as an answer
	node 18	created by program clause resol.
	node 19	created by program clause resol. (repeated answer)
	node 20	created by program clause resol.
na([3],reach(1,_v0),12)	node 20	registered as an answer
ar([3],reach(1,_v0),reach(3,_v0),13)	node 21	created by answer return
na([3],reach(3,_v0),14)	node 21	registered as an answer
ar([2],reach(3,_v0),reach(1,_v0),15)	node 22	created by answer resol.
ar([1],reach(3,_v0),reach(1,_v0),16)	node 23	created by answer resol.
na([1],reach(1,_v0),17)	node 23	registered as an answer
ar([3],reach(3,_v0),reach(1,_v0),18)	node 24	created by answer resol.
ar([1],reach(1,_v0),reach(3,_v0),19)	node 25	created by answer resol.v
cmp(reach(1,_v0),1,20)		
cmp(reach(3,_v0),1,21)		

Table 10.1: Log file for computation in Figure 10.1

`load_forest_log(+File)` module: `tables`

The log produced by `log_forest/[1,2]` is a Prolog file that can be compiled and/or loaded dynamically just as any other Prolog file. However, for large logs (i.e. those of many megabytes) use of `load_dync/[1,2]` XSB commands can drastically reduce the time needed to load the file, while use of the proper `index/2` declarations can greately improve query time. The simple predicate, `load_forest_log/1` loads a log file and indexes needed arguments.

10.2.2 Analyzing the log; seeing the forest through the trees

As previously described, forest logging is based on the formal operational semantics of SLG, and as a result the log can be analyzed to query any result that can be modelled by the theory. But despite the power of forest logging, it can be difficult to use. Not all users have the background to fully understand the operational semantics of SLG. Even those users with a formal background may find it difficult to write efficient analysis routines for logs of large computations [3]. Accordingly, XSB provides routines that analyze logs and display information about a computation. These routines can answer many questions about a computation and can provide the starting point for further exploration. We introduce these routines via an extended example.

Example 10.2.3 This example arises from the actual use of forest logging to understand a Flora-2 computation [96], in which the Cyc reasoner (cf. http://www.cyc.com) was translated into Silk (cf. http://silk.semwebcentral.org) and used to answer various questions in biology. Silk itself compiles into Flora-2 which in turn compiles into XSB [4]. After translation, query answering took more resources than expected, and users wanted to determine why. Using the features of Version 3.8, the first step is to call `statistics/0` at the end of the computation. The statistics indicated that the computation took about 30 seconds of CPU time and 300 megabytes of table space, while XSB's trail had allocated over 1 gigabyte of space. The call to `statistics/0` also showed the following information:

```
8678944 variant call check/insert ops: 615067 producers, 8063877 variants.
317346 answer check/insert ops: 304899 unique inserts, 12447 redundant.
```

In other words, there were nearly 10 million tabled subgoals that were called, indicating that this computation was heavily tabled (a characteristic of most Flora-2

[3] I find it difficult myself!

[4] This example was run in 2012 using a 64-bit server with a large amount of RAM.

computations), It also shows that the average number of answers per tabled subgoal is rather small.

This basic information leads to several questions. Why were there so many tabled subgoals? Did the tabling have anything to do with the large amount of choice-point/trail space that was allocated? Which tabled subgoals had answers? How many times did a given tabled predicate call another tabled predicate?

Some of these questions can be answered by `table_dump/[2,3]`: particularly, what tabled subgoals were called, and which had answers. However `table_dump/[2,3]` cannot provide other information, such as the dependencies of given tabled subgoals on other tabled subgoals or the order in which operations occurred. From a formal perspective, `table_dump/[2,3]` does not allow a user to analyze an entire SLG forest: only the "table", i.e., the subgoals in the forest and the unordered set of its answers. The table omits any information about interior nodes or completion information, both of which are used to compute dependency information. Dependencies are useful in analyzing most computations, but is especially important in Flora-2 computations such as this one, that make heavy use of HiLog. This use of HiLog means that the dependencies of tabled predicates on one another is not at all obvious, and may not easily be determined by static analysis.

The next step, therefore, in analyzing this computation is to rerun it with forest logging. For this computation forest logging has no impact on memory usage, but increases the time of the computation from about 30 seconds to about 52 seconds — around 73% in this case. It is worthwhile noting that the actual overhead of forest logging varies depending on how heavily the computation is tabled. The log itself had slightly over 14 million entries which were loaded into XSB via `load_forest_log/1`. The log took about 140 seconds to load and about 7.8 Gbytes of space for the log facts and their multiple and trie indexes [5].

The easiest way to start the analysis is to ask the query `?- forest_log_overview`, which for this example gives:

```
There were 613496 subgoals in 463330 (completed) SCCs.
93918 subgoals were early-completed.
0 subgoals were not completed in the log.
There were a total of 8670043 tabled subgoal calls:
    613496 were calls to new subgoals
    4467747 were calls to incomplete subgoals
```

[5]The load time for this example, about 100,000 facts/second is typical for 2012 CPUs; the size of the loaded code is larger than usual, due in part to the expansion in the size of terms caused by the HiLog encoding.

```
    3588800 were calls to complete subgoals

Number of SCCs with 1 subgoals is 463322
Number of SCCs with 4 subgoals is 1
Number of SCCs with 7 subgoals is 1
Number of SCCs with 52 subgoals is 1
Number of SCCs with 110 subgoals is 4
Number of SCCs with 149671 subgoals is 1
```

The overview extends the information shown by `statistics/0`. First, the total number of completed and non-completed SCCs is given along with a count of how many of the completed subgoals were early completed. Information about non-completed SCCs is useful, since the forest log may be analyzed for a computation that does not terminate. Since this computation did terminate, all subgoals in the log were completed [6]. Note that there is also a breakdown of calls to tabled subgoals that distinguishes whether the tabled subgoal was new, completed, or incomplete. Recall that calls to completed tabled subgoals essentialy treat the answers in the table as facts, so that these calls are efficient. Making a call to an incomplete subgoals on the other hand means that the calling and called subgoals are mutually recursive [7] and execution of recursive sets of subgoals can be expensive, especially in terms of space.

Finally, the overview report provides the distributions of tabled subgoals across SCCs. While most of the SCCs were small there was a large one, with nearly 150,000 mutually dependent subgoals. Clearly the large SCC should be examimed. The first step is to obtain its index. The query

```
get_scc_size(SCC,Index)), Index > 1000.
```

returns the information that the index of the large SCC was 39. The query `analyze_an_scc(39,userout)` then provides the following information.

```
There are 149671 subgoals and 4461290 links (average of 30.8073 edges per subgoal)
      within the SCC

There are 2 subgoals in the SCC for the predicate backchainForbidden / 0
There are 2 subgoals in the SCC for the predicate
                  http://www.cyc.com/silk/implementation/transformationPredicate / 0
```

[6] The slight difference between the number of subgoals shown here and the number shown by `statistics/0` is due to the use of tabling in the Flora compiler.

[7] This statement is true in local evaluation but not in batched evaluation.

```
:
There are 15613 subgoals in the SCC for the predicate gpLookupSentence / 3
There are 15613 subgoals in the SCC for the predicate removalSentence / 3
There are 18770 subgoals in the SCC for the predicate forwardSentence / 3
There are 18771 subgoals in the SCC for the predicate lookupSentence / 3

Calls from assertedSentence/3 to lookupSentence/3 : 32
Calls from backchainForbidden/0 to 'http://www.cyc.com/silk/implementation/transformatio
:
Calls from transformationSentence/2 to sbhlSentence/3 : 5479
Calls from tvaSentence/3 to removalSentence/3 : 7695
```

It is evident from the first line in this report that the vast majority of the calls to incomplete tables during this computation occur in the SCC under investigation. Since information on incomplete tables is kept in XSB's choice point stack (cf. [68]), the evaluation of SCC 39 is the likely culprit behind the large amount of stack space required. The subgoals in the SCC are first broken out by their predicate name and arity, then the edges within the SCC are broken out by the predicates of their caller and called subgoals. At this point a programmer can review the various rules for `lookupSentence/3`, `forwardSentence/3` and other predicates to determine whether the recursion is intended and if so, whether it can be simplified.

Using abstraction in the analysis

Within the SCC analysis, information about a given tabled subgoal S was abstracted to the functor and arity of S. For this example, abstraction was necessary, as reporting 150,000 subgoals or 4,000,000+ would not provide useful information for a human being. However, it could be the case that seeing the tabled subgoals themselves would be useful for a smaller SCC. Even for an SCC of this size, different levels of abstraction could be useful: mode information or type information might be useful in a given circumstance.

Example 10.2.4 Making the call `?- analyze_an_scc(39,userout,abstract_modes(_,_))` applies the predicate `abstract_modes/2` to each term, producing an output of the form:

```
There are 149671 subgoals and 4461290 links (average of 30.8073 edges per subgoal)
           within the SCC
```

```
There are 3 subgoals in the SCC for the predicate backchainRequired(g,g)
There are 2 subgoals in the SCC for the predicate backchainForbidden(g,g)
:
There are 29254 subgoals in the SCC for the predicate gpLookupSentence(g,g)
There are 29254 subgoals in the SCC for the predicate removalSentence(g,g)

Calls from assertedSentence(g,g) to lookupSentence(g,g) : 10
Calls from assertedSentence(m,g) to lookupSentence(m,g) : 22
:
Calls from transformationSentence(m,g) to sbhlSentence(m,g) : 741
Calls from tvaSentence(g,g) to removalSentence(g,g) : 7695
```

`abstract_modes(In,Out)` simply goes through each argument of `In` and unifies the corresponding argument of `Out` with a `v` if the argument is a variable, a `g` if the argument is ground, and `m` otherwise.

`abstract_modes/2` is simply an example: any term-abstraction predicate may be passed into the last argument of `analyze_an_scc/3` [8].

Analyzing Negation

Many programs that use negation are stratified in such a way that they do not require the use of DELAYING and SIMPLIFICATION operations, and the routines described in the previous section are sufficient for these programs. However if a program does not have a two-valued well-founded model, a user would often like to understand why. Even in a program that is two-valued, the heavy use of DELAYING and SIMPLIFICATION can indicate that some rules may need to be optimized by having their literals reordered.

Example 10.2.5 Figure 10.2 shows a program with negation and illustrates SLG resolution for the query *p(c)* to the program. The nodes in Figure 10.2 have been annotated with the order in which they were created under local scheduling. In the formalism used by Figure 10.2, the symbol | in a node separates the unresolved goals to the right from the delayed goals to the left. In the evaluation state where nodes 1 through 10 have been created, *p(b)* has been completed, and *p(a)* and *p(c)* are in the same SCC. There are no more clauses or answers to resolve, but *p(a)* is involved

[8] Because of the special representation of Flora-2 terms, abstraction was used to produce the output of Example 10.2.4, while a more sophisticated version of `abstract_modes/2` was used in Example 10.2.4.

in a loop through negation in node 5, and nodes 2 and 10 involve *p(a)* and *p(c)* in a negative loop [9].

In situations such as this, where all resolution has been performed for nodes in an SCC, an evaluation may have to apply a DELAYING operation to a negative literal such as *not(p(a))*, in order to explore whether other literals to its right might fail. When multiple literals can be delayed, an arbitrary one is chosen to be delayed first. So the evaluation delays the selected literal of node 2 to generate node 12 producing a *conditional answer* – an answer with a non-empty delay list (cf. Section 5.3.2 for an overview of how XSB computes and allows inspection of delayed literals). Next, *not p(a)* in node 5 is delayed, failing that computation path, and *not p(c)* in node 10 is delayed to produce node 15 and failing the final computation path for *p(a)*. At this stage the SCC $\{p(a), p(c)\}$ is *completely evaluated* meaning that there are no more operations applicable for goal literals (as opposed to delay literals). Since *p(a)* is completely evaluated with no answers, conditional or otherwise, the evaluation determines it to be *failed* and a SIMPLIFICATION operation can be applied to the conditional answer of node 12, leading to the *unconditional* answer in node 17 and *success* of the literal *p(c)*.

As indicated previously, the forest log overview includes a total count of DELAYING and SIMPLIFICATION operations, as well as a count of conditional answers. In addition, SCC analysis counts negative as well as positive links within the SCC. The current version of forest logging also provides a means to examine the causes of answers that have an undefined truth value. Recall from Example 10.2.5 that there are two types of causes of an undefined truth value: either 1) a negative literal explicitly undergoes a DELAYING operation; or 2) a conditional answer may be used to resolve a literal. It can be shown that in local evaluation, a conditional answer A will never be returned out of an SCC if A is successful or failed in the well-founded model of a program. This means that if an answer for S is undefined, then it would be caused operationally by a DELAYING operation within the SCC of S or within some other SCC on which S depends. So to understand why an atom is undefined it can be useful understand the "root causes" of the delay: to examine SCCs in which DELAYING operations were executed and conditional answers were derived, but the answers could not be simplified.

Example 10.2.6 *As a use case, logging was made of execution of a Flora-2 program that tested out a new defeasibility theory. The forest log overview indicated that the top-level query was undefined:*

[9] In this example, we ignore the effects of early completion which would complete *p(b)* immediately upon creation of node 8, obviating the need to create node 9.

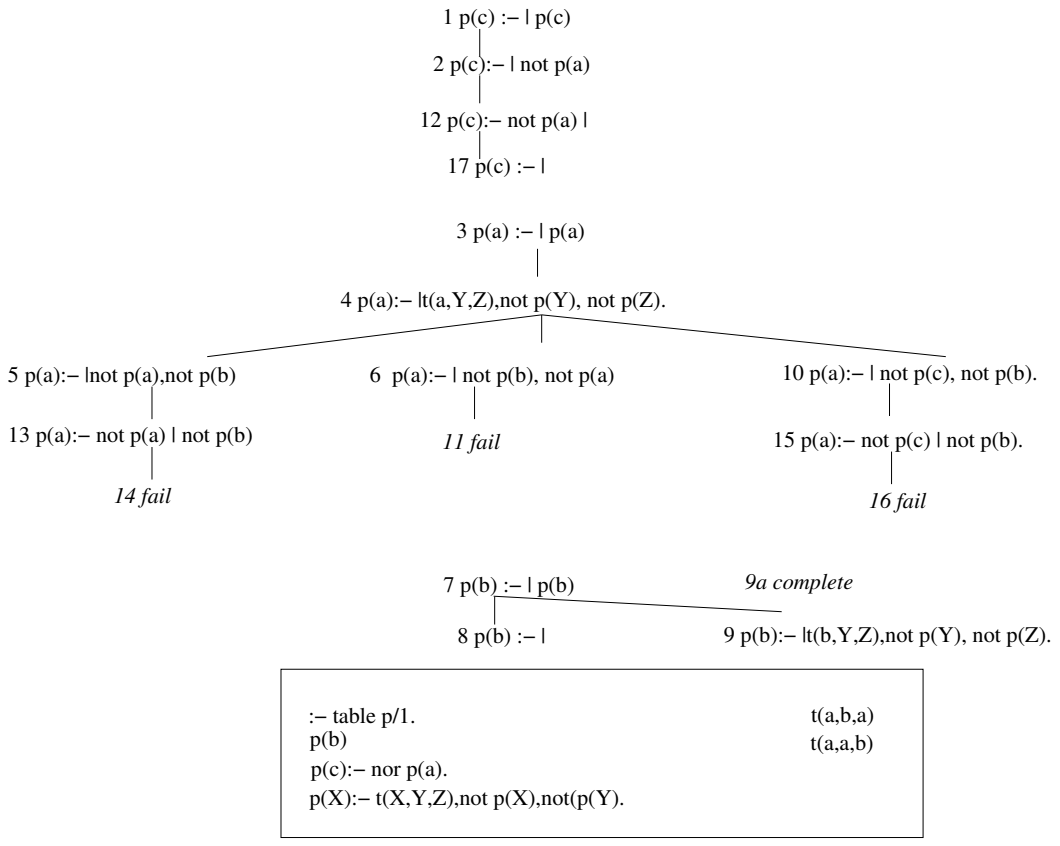


Figure 10.2: A Normal Program and SLG Forest for Evaluation of the Query *p(c)*

```
:
There were a total of 55 negative delays
There were a total of 0 simplifications
There were a total of 695 unconditional answers derived:
There were a total of 66 conditional answers derived:
```

The analysis predicate `three_valued_scc(List)` *produces a list of all SCC indices in which* DELAYING *caused the derivation of conditional answers. These SCCs can then be analyzed as discussed in the previous section.*

10.2.3 Discussion

Using log forest imposes a relatively minimal overhead on most computations, considering the information it can provide, and loading and analysis is relatively quick. For this example, the top level analysis took around 10 seconds, and analysing SCC 39 took about 20 seconds in Example 10.2.3 and about 60 seconds in Example 10.2.4. For more information, see [83].

10.2.4 Predicates for Forest Logging

`forest_log_overview` module: `tables`

Provides an overview of subgoals, calls, and SCCs in the forest log as indicated in Section 10.2.2.

`get_scc_size(?Index,?Size)` module: `tables`

This simple predicate determines the indices of SCCs whose size is `Size`, for use with `analyze_an_scc/[2,3]`.

`three_valued_sccs(List)` module: `tables`

If there are any SCCs in the log where delay is performed, causing conditional answers to be added that were not simplified into unconditional answers, unifies `List` with the index of all such SCCs.

`analyze_an_scc(+Index,+File)` module: `tables`
`analyze_an_scc(+Index,+File,+Abstraction)` module: `tables`

These predicates can be used to analyze the SCC indexed by `Index` in a forest log, as explained in Section 10.2.2. The output is written to `File`; calling the predicate with `File` set to `userout` causes the output to be written to the console. In `analyze_an_scc/2`, tabled subgoals are abstracted to predicate

indicators, in `analyze_an_scc/3`, a two-ary abstraction predicate in `usermod` is called.

Error conditions on `File` are the same as `tell/1`.

`abstract_modes(Term,AbstractedTerm)` module: `usermod`

`abstract_modes(In,Out)` simply goes through each argument of `Term` and unifies the corresponding argument of `Abstracted` with a `v` if the argument is a variable, a `g` if the argument is ground, and `m` otherwise.

To use this predicate, the file `term_abstract.P` must be loaded, via `ensure_loaded/1` or similar means.

`set_forest_logging_for_pred(+PredSpec,+Mode)` module: `tables`

If forest logging is active, this predicate allows any logging specific to the predicate or term indicater, `PredSpec`, to be turned on or off. Thus, for instance, tabled predicates in a pre-existing library need not clutter up the log.

Error Cases

- `PredSpec` is not a predicate or term indicator.
  - `type_error`
- `Mode` is not in the set {`on`,`off`}
  - `domain_error`

10.3 Inspecting a Tabled Derivation

As described in the previous section, Forest Logging is a powerful technique for understanding the operational aspects of a tabled derivation, and is based on the idea that a derivation is itself a mathematical entity that can be represented and analyzed. This basis allows Forest Logging to support various types of analysis including profiling the derivation, and understanding its termination properties [50, 49]. At the same time, Forest Logging may not always be convenient to use. Since it is a trace-based analysis a (sometimes very large) trace file must be created and loaded before being analyzed.

An alternate approach is to use *inspection predicates* – a term that loosely refers to predicates useful for understanding a tabled derivation. Most of these predicates can be used in two ways. First, they can inspect an on-going derivation that has been suspended through various means. Alternately, they may be used to retroatively inspect a derivation that has completed. In this section, we first describe two important sets

of interactive inspection predicates. First we describe the `table_dump` library which provides a flexible approch to inspecting tables (Section 10.3.1).[10] Next we discuss a set of predicates for inspecting various dependency graphs of a computation (Section 10.3.2. We then discuss how *tripwires* can automatically suspend a derivation for inspection at a point where the derivation begins to use too many resources, and so might be inefficient (Section 10.3.4).

10.3.1 Inspecting Tables with `table_dump`

`table_dump(+Term,+OptionList)` module: `dump_table`
`table_dump(+Stream,#Term,+OptionList)` module: `dump_table`

`table_dump/[2,3]` provides an easy method to view subgoals and answers that are present in a table. Given an input `Term`, `table_dump/[2,3]` provides information about all tabled subgoals that are subsumed by `Term`; if `Term` is a variable, information is provided about all tables.

The information can be provided at three levels of aggregation, and the form of the information is determined by the options in `OptionsList`.

- If the option `summary(true)` is set, the aggregate sum of subgoals and answers that are subsumed by `Term` is collected, along with the aggregate sum of calls *to* these subgoals. If `Term` is a variable this information is broken down by tabled predicates.
  - If `details(answers)` is set, a list is collected of every tabled subgoal S such that S is subsumed by `Term` along with the number of answers for each S along with a list of those answers and the truth value of each answer (`t` if true and `u` if undefined). If `Term` is a variable this information is broken down by tabled predicates.
  - If `details(subgoals)` is set, a list is collected of all subgoals S such that S is subsumed by `Term` along with the number of answers for each S. However, unlike the action for `details(answers)` the actual list of answers for S is not returned. If `Term` is a variable this information is broken down by tabled predicates.
  - If `details(false)` is set, no detail information is provided for the actual subgoals or their answers.
- If `OptionsList` contains the option `results(X)` for some variable `X`, `X` will be instantiated upon backtracking to all infomation collected about the tables.

[10] Other predicates for table inspection that are generally lower-level are described in Section 6.15.

- If the option `output(true)` is set, the information is written to `Stream` or to `userout` in Prolog-readable form.

If not otherwise specified the default options are `summary(true)`, `details(false)`, `output(true)`.

Example Consider the program:

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

:- table q/1.
q(1).              q(2).

:- table r/1.
r(a).

:- table s/2.
s(1,a).            s(2,b).          s(1,a1).            s(2,b1).
```

and suppose the top-level query `?- p(X,Y)` has been made. Then `table_dump/2` provides the following information **(reformatted for readability)**:

```
| ?- table_dump(_X,[summary(true)]).

summary = p(A,B) - subgoals(3) - total_times_called(4) - total_answers(7)

X = p(_h243,_h244);

summary = q(A) - subgoals(1) - total_times_called(1) - total_answers(2).

X = q(_h228)

yes
| ?- table_dump(_X,[details(answers)]).

summary = p(A,B) - subgoals(3) - total_times_called(4) - total_answers(7).
details = p(A,B) - subgoals(3) - details([
    p(C,D) - times_called(1) - answers(5) - [p(3,1)-t,p(3,2)-t,p(2,b)-t,
```

```
                                          p(1,b)-t,p(1,a)-t]          - completed,
    p(1,a) - times_called(2) - answers(1) - [p(1,a)-t]                - completed,
    p(2,b) - times_called(1) - answers(1) - [p(2,b)-t]                - completed]).

X = p(_h232,_h233);

summary = q(A) - subgoals(1) - total_times_called(1) - total_answers(2).
details = q(A) - subgoals(1) - details([
     q(B) - times_called(1) - answers(2) - [q(2)-t,q(1)-t] - completed]).

X = q(_h232)

yes
```

As the above example shows, each line of the summary has the form:

summary =
Pred/Goal - subgoals($N_{subgoals}$) - total_times_called(N_{called}) - total_answers($N_{answers}$)

where

- *Pred/Goal* is either a term indicator, if the `Term` argument of `table_dump/[2,3]` was a variable (to indicate there should be no filtering of tabled calls); or `Term` itself.
- $N_{subgoals}$ are the total number tabled subgoals that are subsumed by *Pred/Goal* (perhaps including *Pred/Goal* itself).
- N_{called} is the total number of times all subgoals subsumed by *Pred/Goal* have been called.
- $N_{answers}$ is the total number of answers currently derived by all subgoals subsumed by *Pred/Goal*.

Each line of details has the form:

Details =
Pred/Goal - subgoals($N_{subgoals}$) - details(List)

where *Pred/Goal* and $N_{subgoals}$ are as above. If `details(answers)` was an input option

List =
Subgoal - *times_called*(N_{called}) - *answers*($N_{answers}$) - *List_of_Answers* - *Status*

for each *Subgoal* in the table subsumed by *Pred/Goal*. N_{called} and $N_{answers}$ are as above, while *List_of_Answers* contains $A - TV$ for each answer A with truth value TV that is currently derived for *Subgoal*. On the other hand, if `details(subgoals)` was an input option

List =
Subgoal - *times_called*(N_{called}) - *answers*($N_{answers}$) - *Status*

where all elements are as before. Finally *Status* is

- `completed` if *Subgoal* has been completed; and
- `scc`(N_{SCC}) if *Subgoal* is incomplete. N_{SCC} is relative: if N_{SCC} is greater than M_{SCC} then N_{SCC} is a descendent of M_{SCC}: i.e., subgoals in SCC M_{SCC} depend on subgoals in SCC N_{SCC}. However, these numbers should only be used relatively: at a given state in the computation there may be fewer than M_{SCC} Sccs.[11]

Error Cases

- `OptionList` is a variable, or contains a variable as an element
  - `instantiation_error`
- `OptionList` is not a list
  - `type_error(list,OptionList)`
- `OptionList` contains an element, `O`, that is not a valid `table_dump_option`.
  - `domain_error(table_dump_option,O)`

10.3.2 Inspection Predicates for Dependency Graphs

Recall that Forest Logging is based on a representation of the tabling operations of an entire SLG evaluation, even those for completed tables. Maintaining such information within XSB's engine would be prohibitively expensive, which is why Forest Logging needs a trace. Nonetheless, XSB's engine does maintain certain information that

[11] XSB keeps track of SCCs through an algorithm similar to depth-first search: the numbers associated with subgoals are the depth-first numbers of the minimal back-dependency of a subgoal (cf. [68])

indicates critical aspects of a tabled derivation. As discussed in the previous section, the tables themselves can be viewed and can offer useful information. However, the tables don't provide information about how the different subgoals depend on one another, an aspect that is often central to optimizing a derivation.

However, such dependency information is available in some cases. For incremental tables, dependencies among subgoals may be obtained through the Incremental Dependency Graph (IDG). In addition, XSB maintains information about the dependencies among incomplete subgoals, and this information can be viewed through the Subgoal Dependency Graph (SDG).[12] As an separate matter, it can be difficult to understand why certain atoms are undefined from looking directly at the tables. For this, the Residual Dependency Graph (RDG) can be inspected.

In this section we first present an adjacency list format for representing dependency graphs in Prolog. We then consider predicates for obtaining information about each type of dependency graph. As dependency graphs may be too large for humans to productively read, we also present predicates that allow filtering, manipulation and summary of these graphs.

A Prolog Format for Dependency Graphs

Several of the inspection predicates produce a dependency graph in Prolog in the format of adjacency lists. This format also annotates information about each subgoal. Specifically, an adjacency list as used here is a list of terms of the form:

`subgoal(Vertex,SCCKey,SubgoalKey,CallsTo,Answers,PosEdges,NegEdges)` such that:

- `Vertex` is a vertex in the current state of a dependency graph. For the Subgoal Dependency Graph (SDG) and Incremental Dependency Graph (IDG), `Vertex` is a subgoal; for the Residual Dependency Graph (RDG) it is a subgoal/atom pair.
- `SCCKey` is a key of the SCC to which the vertex belongs.[13]
- `VertexKey` is a key that uniquely identifies `Vertex` and is either
  - An integer value that represents the handle to the table entry; or

[12] Maintenance of the Subgoal Dependency Graph is in fact necessary to ensure that all appropriate answers are returned to each incomplete subgoal.

[13] In general, no information can be inferred from the ordering of the returned SCC keys.

– An atom that represents a unique generated key for the vertex of the dependency graph, if a morphism has been applied to the dependency graph.

- `CallsTo` For the SDG and IDG the number of calls that have been made to the subgoal so far; for the RDG this value is set to 0.
- `Answers` For the SDG and IDG the number of distinct answers that the subgoal has so far; [14] for the RDG this value is set to 0.
- `PosEdges` is a list of keys for those vertices that `Vertex` positively directly affects. In the case of dependency graphs that do not have signed edges, all edge information is kept in this argument.
- `NegEdges` is a list of keys for those vertices that `Vertex` negatively directly affects. In the case of dependency graphs that do not have signed edges, no edge information is kept in this argument.

Predicates to Access the Subgoal Dependency Graph

The Subgoal Dependency Graph (SDG) has as vertices those tabled subgoals that are incomplete in the state of a suspended derivation. A *depends* edge exists from S_1 to S_2 iff a call is made to S_2 while computing answers for S_1, and if there are no intervening tabled subgoals between S_1 and S_2. An *affects* edge is the inverse of a depends edge. Edges in the SDG are signed indicating positve or negative dependence. A subgoal and its incident edges are removed from the SDG when the subgoal is completed.

The main predicate for accessing information about the SDG is `get_sdg_info/1`. Because it accesses the SDG, `get_sdg_info/1` returns information concerning incomplete subgoals *only*.

`get_sdg_info(-SDG)` module: `tables`

For a suspended derivation, returns information about the *Subgoal Dependency Graph* (SDG) as an adjacency list whose form is described in Section 10.3.2. If there are no incomplete tables in the current state, an empty list is returned.

This predicate has no error conditions.

Example 10.3.1 Consider the goal `?- q(3,3)` to the program:

[14] If the same answer was derived more than once, it is counted only one time.

```
:- import get_sdg_info/1 from tables.
:- import between/3 from basics.

:- table q/2 as incremental.
q(M,N):- between(1,N,X),
         (M = N,N = X -> break ; q(X,N)).
```

Execution of this query creates a number of tabled subgoals, but breaks before the initial goal is completely evaluated. The SDG at the time of the break is shown in Figure 10.3

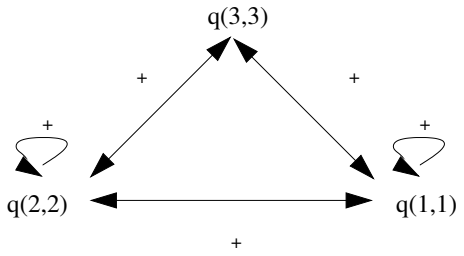


Figure 10.3: *SDG for* `?- q(3,3)` *when the derivation is suspended by* `break/0`c

This SDG can be produced as follows:

```
| ?- q(3,3).
[ Break (level 1) ]

1: ?- get_sdg_info(F).

F = [subgoal(q(2,3),1,140253373671912,3,0,
             [140253373671672,140253373671792,140253373671912],[]),
     subgoal(q(1,3),1,140253373671792,3,0,
             [140253373671792,140253373671672,140253373671912],[]),
     subgoal(q(3,3),1,140253373671672,3,0,
             [140253373671912,140253373671792],[])]
```

`get_sdg_subgoal_info(-SDG)` module: `tables`

Note that the size of the SDG, which includes dependency edges, may be quadratic in the number of incomplete subgoals. When only summary information about the subgoals in the subgoal dependency graph `get_sdg_subgoal_info/1` can be used, rather than `get_sdg_info/1`. As with the previous predicate, a list of terms of the form,

`subgoal(Vertex,SCCKey,_SubgoalKey,CallsTo,Answers,_PosEdges,_NegEdges)`

is returned, but `_SubgoalKey` is set to the atom `null`, and `_PosEdges` and `_NegEdges` are both set to the empty list.

This predicate has no error conditions.

Predicates to Access the Incremental Dependency Graph

The Incremental Dependency Graph (IDG) is used by XSB's incremental tabling subsystem to ensure that tables that depend on dynamic facts or rules are properly updated when the underlying dynamic code changes.

The Incremental Dependency Graph (IDG) has as vertices those subgoals whose predicate symbols are incrementally tabled, along with calls to dynamic predicates that are declared as incremental. A *depends* edge exists from S_1 to S_2 iff a call is made to S_2 while computing answers for S_1, and if there are no intervening tabled subgoals between S_1 and S_2. An *affects* edge is the inverse of a depends edge. Edges in the IDG are unsigned. XSB maintains both completed and incomplete subgoals in the IDG. (As long as the tables for these subgoals are not abolished.)

The main predicates for inspecting the IDG as a dependency graph are, desribed below. Additionally, Section 5.6.6 contains predicates for examining dependencies among individual subgoals, as well as returning information about whether a subgoal in the IDG needs to be updated or not.

`get_idg_info(+SubgoalList,-SDG)` module: `tables`
`get_idg_info(-SDG)` module: `tables`

Warning: this predicate is not yet implemented

Returns information about the *Incremental Dependency Graph* (IDG) as an adjacency list whose form is described in Section 10.3.2. If there is an empty IDG in the current state, an empty list is returned.

Recall from the previous section that if the SDG is accessed, information is returned about all completed subgoals. The IDG however may be both very large and disconnected. Accordingly, `get_idg_info/2` allows a list of subgoals to be specified, and returns information about all of the IDG that is connected to any subgoal in the list; note that the resulting dependency graph may also be disconnected. If `get_idg_info/1` is called, information is returned about the entire dependency graph.

Error Cases

- `SubgoalList` is a variable
  - `instantiation_error`
- `SubgoalList` is not a list
  - `type_error`
- `SubgoalList` contains a predicate that is not tabled
  - `permission_error`

Predicates to Access the Residual Dependency Graph

As discussed in Section 5.3.3, answers that are undefined in the well-founded semantics are stored in XSB along with their delay lists, forming a residual program. The residual program can also be represented as a Residual Dependency Graph (RDG). Using the RDG, a user may be able to determine why an answer A to a subgoal S was unexpectedly undefined either because that answer was involved in or depended on a loop through negation; or because the answer depended on some other answer that was undefined because of the use of bounded rationality (Section 5.5) or because of floundering and the use of `u_not/1`.

The representation of the RDG is slightly different from that of the other dependency graphs. The following example illustrates the reasons for this.

Example 10.3.2 Consider the program

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

to which the query ?- p(1,X) was made, generating the tables:

Subgoal	*Answers*
p(1,X)	p(1,2) p(1,3):- tnot(p(2,3))|
p(1,3)	p(1,3):- tnot(p(2,3))|
p(2,3)	p(2,3):- tnot(p(1,3))|
	p(2,3):- tnot(r(a))|
r(a)	r(a):- tnot(r(b))|
r(b)	r(b):- tnot(r(a))|

The residual dependency graph for this program and query would have a node for each subgoal/answer combination with an undefined truth value, and a dependency edge for nodes S_1/A_1 and S_2/A_2 if A_2 occurs in a literal in the delay list for S_1/A_1, and the original subgoal for A_2 was S_2 in the subcomputation for S_1. The edge also has a sign indicating whether A_2 occurs positively or negatively in the delay list for A_1. In this example, the residual dependency graph could be conceptually represented as

```
depends_on(p(1,X)/p(1,3),p(2,3)/p(2,3),-).
depends_on(p(1,3)/p(1,3),p(2,3)/p(2,3),-).
depends_on(p(2,3)/p(2,3),p(1,3)/p(1,3),-).
depends_on(p(2,3)/p(2,3),r(a)/r(a),+).
depends_on(r(a)/r(a),r(b)/r(b),-).
depends_on(r(b)/r(b),r(a)/r(a),-).
```

Thus, vertices of the RDG are subgoal/atom pairs, unlike in the other dependency graphs where they are simply subgoals. Summarizing, the RDG which has as vertices those pairs of subgoals and answer atoms, such that the truth value of the answer atom for that subgoal is *undefined* in the state of a suspended computation. A *depends* edge exists from V_1 to v_2 iff V_2 is a delay literal in a conditional answer for V_1. An *affects* edge is the inverse of a depends edge. Edges in the RDG are signed indicating positve or negative dependence.[15] A pair (S, A) and its incident edges are removed from the RDG when the truth value of A changes, and of course when S is abolished.[16]

Information about specific vertices and edges of the RDG can be obtained through predicates such as `get_residual/2` and `variant_get_residual/2`.

`get_rdg_info(+PairList,-SDG)` module: `tables`
`get_rdg_info(-SDG)` module: `tables`

Warning: this predicate is not yet implemented

Returns information about the *Residual Dependency Graph* (RDG) as an adjacency list whose form is described in Section 10.3.2. If there is an empty RDG in the current state, an empty list is returned.

[15] An alternative definition of the RDG has tabled subgoals as vertices, where subgoal S_1 depends on subgoal S_2 if some answer for S_1 depends on some answer for S_2. Such a representation can be obtained from `get_rdg_info/[1,2]` below by applying a morphism, as described in Section 10.3.2.

[16] The truth value of an atom for a given subgoal may change when a suspended state is further evaluated, so that depending when a computation is suspended, it is possible though rare that a given atom may have a definite truth value when associated with one subgoal, but the truth value may not have been propiaged to another subgoal. Note that the truth value of atoms may also change for completed subgoals when the ANSWER COMPLETION operation is lazily performed.

Recall from the previous section that if the SDG is accessed, information is returned about all completed subgoals. The RDG however may be both very large and disconnected. Accordingly, `get_rdg_info/2` allows a list of subgoal/atom pairs to be specified, and returns information about all of the RDG that is connected to any subgoal in the list; note that the resulting dependency graph may also be disconnected. If `get_rdg_info/1` is called, information is returned about the entire dependency graph.

Error Cases

- `PairList` is a variable
  - `instantiation_error`
- `PairList` is not a list
  - `type_error`
- `PairList` contains a predicate that is not tabled
  - `permission_error`

`get_residual_sccs(+Subgoal,+Answer,-SCCList)` module: `tables`
`get_residual_sccs(+Subgoal,+Answer,-SCCList,-DepList,-SignList)` module: `tables`

Warning: these predicates may be obsolescent.

The residual dependency graph can be constructed in a straightforward way from `variant_get_residual/2`. However `get_residual_sccs/[3,5]` provides an alternate view that is higher-level and much faster. Given a subgoal/answer pair as input, each of these predicates constructs SCC-based information about the residual dependency graph via structures of the form:

```
ret(Subgoal,Answer,SCCKey).
```

where `SCCKey` is a generated key for the SCC to which the Subgoal/Answer pair belongs. Two subgoal/answer pairs are in the same SCC iff they have the same `SCCKey`; however no other dependency information can be otherwise directly inferred from the index [17].

To obtain dependency information, `get_residual_sccs/5` also returns a list indicating the direct dependencies among the SCCs, along with a list indicating whether each SCC contains a negative edge. For Example 10.3.2, the SCC information would have a form such as:

[17] The actual number used for each SCC key depends on how RDG happens to be traversed; as a result it is best to rely on the key only as a "generated" name for each SCC.

```
[ ret(p(1,X),p(1,3),1), ret(p(1,3),p(1,3),2), ret(p(2,3),p(2,3),2),
  ret(r(a),r(a),3), ret(r(b),r(b),3) ]
```

The dependency list would have a form such as:

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

while the sign list would have a form such as:

```
[ sign(1,no_neg), sign(2,neg), sign(3,neg) ]
```

If it is necessary to know which subgoal(s) in `SCC1` directly depends on which subgoal(s) in `SCC2`, the information can be easily reconstructed from the output of `get_residual_sccs/[4,5]` using `variant_get_residual/2`. A similar approach can be used to determine the actual edges within a given SCC.

SCC detection is implemented using Tarjan's algorithm [87] in C working directly on XSB's data structures. The algorithm is $\mathcal{O}(|V|+|E|)$ where $|V|$ is the number of vertices and $|E|$ the number of edges in the dependency graph. As a result, `get_residual_sccs/3` provides an efficient means to materialize the high-level topography of the dependency graph [18].

`explain_u_val(+Subgoal,+Answer,-Reason)` module: `tables`

`explain_u_val(+Subgoal,+Answer,-Sccs,-Deps,-Signs,-Reason)` module: `tables`

The XSB predicate `explain_u_val(+Subgoal,+Answer,?Reason)` can be used to query why `Answer` is undefined when derived in an evaluation of `Subgoal`. `Reason` may be

- `negative_loops(cycle)` if the derivation of `Answer` involves a loop through though negation that includes `Answer` itself.
- `negative_loops(dependent)` if the derivation of `Answer` depends on an atom that is involved in a loop through though negation.
- `unsafe_negation` if the derivation of `Answer` depends on a negative subgoal that is non-ground (XSB does not automatically perform subgoal reordering). The action of making a non-ground subgoal undefined is performed by `u_not/1`.

[18] Currently, the materialization of dependency information between SCCs is implemented in a naive manner, so that `get_residual_sccs/6` is $\mathcal{O}(|V|^2)$.

- `bounded_rationality` if the derivation of answer depends on bounded rationality based on radial restraint [34].

These reasons are not exclusive, and complex derivations may well involve several of the above reasons.

`explain_u_val/[3,6]` is based on the structures returned by `get_residual_sccs/[3,5]`. While `get_residual_sccs/[3,5]` is reasonably fast, it can take a peceptable time to analyze large residual programs containing many thousands of SCCs. Accordingly, `explain_u_val/6` can reuse dependency structures returned by `get_residual_sccs/[3,5]`, which can be useful for justification systems and other applciations.

Example 10.3.3 After executing the query `p` to the program

```
:- table p/0, q/0, r/0, s/1.
p:- q,tnot p.                    p:- s(f(f(f(f(0))))).

q:- tnot r.                      r:- tnot q.

s(f(X)):- s(X).                  s(0).
```

where the bounded rationality size has been set to 3. The query `explain_u_val(p,P,Reason)` will bind `Reason` to `negative_loops(cycle)`, to `negative_loops(dependent)`, and to `bounded_rationality` (this ordering is not guaranteed).

Filtering, Manipulating, and Summarizing Dependency Graphs

`morph_dep_graph(+DG_In,+Morph,-DG_Out)` module: `tables`

This predicate takes as input `DG_In`, a dependency graph in adjacency list format and returns its image, `DG_Out`, under the graph homomorphism `Morph`. `Morph` is a predicate symbol that identifies a 2-ary predicate, `Morph(+In,-Out)` that is functional on `In` and that maps the Herbrand Base of the current program into itself. The syntax of `DG_In` and `DG_Out` is described at the beginning of Section 10.3.2.

To recall the definition of a graph homomorphism (cf. e.g., [36]) a functional notation is used for `Morph/2`. `DG_Out` is a graph such that the vertices of `DG_Out`, *vertices*(`DG_Out`) is the set:

$$\{morph(V)|V \in vertices(\texttt{DG\_In})\}$$

while the edges of `DG_Out`, $edges($`DG_Out`$)$ are the sets

$$\{\langle morph(V_1), morph(V_2)\rangle | \langle V_1, V_2\rangle \in edges(\texttt{DG\_In})\}$$

We adapt this definition to signed dependency graphs by mapping all positive adjacenct edges into a positive set, and negative adjacent edges into a negative set.

The power of `morph_dep_graph/3` arises when the numbers of vertices and edges of `DG_In` is large, and `morph/2` ensures that numbers for `DG_Out` are much smaller – thus allowing recognizable patterns to emerge.

For efficiency reasons, a special condition, $\mathcal{C}_1$, is assumed about `morph/2`: that if two elements of its range unify, then they must be identical. For instance, a morphism M_1 that reduced the maximum depth of each non-variable argument of a term by 1 would not fit this condition, since $M_1(f(a, g(h(b)))) = f(X_1, g(h(X_2)))$ while $M_1(f(a, g(b))) = f(X_1, g(X_2))$, which unify. On the other hand, a morphism that abstracts each argument to have a maximal fixed depth would fulfill the condition. In any case, as long as $\mathcal{C}_1$ is observed, `morph/2` may instantiated by an abstraction function as used elsewhere in this manual: i.e., a function such that $morph(Term)$ subsumes $Term$. However, other morphisms may also be useful as demonstrated in Example 10.3.4 below.

While the syntax of `SDG_Out` is the same as that of `SDG_In`, the meaning of the arguments differs slightly. `SDG_Out` is a list of terms of the form:

```
subgoal(MorphSubg,null,Key,CallsTo,Answers,PosKeyList,NegKeyList)
```

such that

- `MorphVert` is $morph($`Vertex`$)$ for one or more subgoals that are vertices of `DG_In`
- The second argument, which represents SCC information in the original dependency graph, is the atom `null` when a morphism is applied, since SCC information is not preserved in general.
- `Key` is an atom identifying `MorphVert`. Note that while each subgoal in `DG_In` corresponds to e.g., a tabled subgoal, a given subgoal image in `DG_Out` may not correspond to a tabled subgoal in the current state. Thus a table entry handle may not be available, so generated keys are used in `DG_Out`.
- `CallsTo` If `DG_In` originated from an SDG or IDG, `CallsTo` is the sum of the number of calls to every subgoal $S \in$ `DG_In` such that $morph(S) =$ `MorphVert`. Otherwise, `CallsTo` is 0.

- `Answers` If `DG_In` originated from an SDG or IDG, `Answers` is the sum of the number of answers for every subgoal $S \in$ `DG_In` such that $morph(S) =$ `MorphVert`. Otherwise, `Answers` is 0.
- `PosKeyList` is a list of the keys to those vertices adjacent to `MorphSubg` with positive sign as described above.
- `NegKeyList` is a list of the keys to those vertices adjacent to `MorphSubg` with negative sign as described above.

Example 10.3.4 Continuing Example 10.3.1, let `mymorph` identify the predicate

```
mymorph(Term,NewTerm):-
        Term =.. [F,A1,A2],
        map_arg_1(A1,NewA1),
        NewTerm =.. [F,NewA1,A2].

map_arg_1(2,1):- !.
map_arg_1(X,X).
```

Thus `mymorph/2` maps `q(2,1)` to `q(1,1)` and maps both `q(1,1)` and `q(3,1)` to themselves. Then if `DG_In` is instantiated to the SDG produced in Example 10.3.1, the goal `?- morph_dep_graph(DG_In,mymorph,DG_Out)` would produce:

```
SDG\_Out = [subgoal(morph80,q(1,3),6,0,[morph80,morph81],[]),
          subgoal(morph81,q(3,3),3,0,[morph80],[])
```

Error Cases

- `Morph` is not an atom
  - `type_error`
- `DG_Out` is not a variable
  - `type_error`
- `DG_In` is not an adjacency list as described above
  - `misc_error`

`dep_graph_scc_info(+SDG,-ListOut)` module: `tables`

Given an SDG representation in the adjacency list format described above, this predicate returns information about the SCCs that are currently under evaluation. Upon success `ListOut` will contain a term

`scc(SCCIndex,NumSubgoals,NumAnswers,NumPosEdges,NumNegEdges)`

for each SCC under evaluation, such that:

- `SCCIndex` is the index of the SCC
- `NumSubgoals` is the number of subgoals in the SCC
- `NumAnswers` is the total number of answers for all subgoals in the SCC
- `NumPosEdges` is the total number of positive edges within the SCC.
- `NumNegEdges` is the total number of negative edges within the SCC.

`print_sdg_info` module: `tables`

Prints the current SDG to `stdout` in a readable manner.[19]

`print_sdg_subgoal_info` module: `tables`

Prints summary subgoal information about the current SDG to `stdout` in a readable manner.

`print_dep_graph(+DG)` module: `tables`

Prints a dependency graph `DG` (whether its an SDG, IDG, or RDG) to `stdout` in a simple, but readable manner.

Example 10.3.5 *Continuing from Example 10.3.4, if*

```
SDG = [subgoal(morph80,q(1,3),6,0,[morph80,morph81],[]),
          subgoal(morph81,q(3,3),3,0,[morph80],[])
```

then `print_dep_graph(SDG)` *would output*

```
Subgoal: q(1,3)
        Number of calls to this subgoal 6; Number of answers 0
        Affects positively q(1,3) ; q(3,3)
Subgoal: q(3,3)
        Number of calls to this subgoal 3; Number of answers 0
        Affects positively q(1,3)
```

[19] This predicate, along with `print_sdg_subgoal_info/0` replaces the predicate `print_incomplete_tables/0`, which was included in previous releases.

10.3.3 Summary: Inspection Predicates

XSB provides a number of ways to inspect a tabled derivation, including directly through the tables, or through one of the dependency graphs: the IDG, RDG or SDG. Specifically, some useful inspection predicates available in XSB are:

- `statistics/[0,1,2]` (Section 6.13) is a highly useful general-purpose predicate that provides an important summary of how memory is used by the XSB process or thread, the amount of time used by the process or thread, along with various counts of tabling operations and measures of table space.
- `table_dump/[2,3]` (Section 10.3.1) allows directed and iterative inspection of the current set of tabled subgoals and their answers, at various levels of summary aggregation.
- Inspection of the Incremental Dependency Graph can be made via the predicate `get_idg_info/[1,2]` [20] together with predicates for dependency graph manipulation such as `morph_dep_graph/3` and `dep_graph_scc_info/3` (cf. Section 10.3.2). More targeted inspection of specific edges and dependencies of the Incremental Dependency Graph is supported through `incr_directly_depends/2` and `incr_trans_depends/2` (cf. Section 5.6.6).
- Inspection of the Subgoal Dependency Graph can be obtained through the predicate `get_sdg_info/1`, and its information analyzed through predicates for dependency graph manipulation such as `morph_dep_graph/3`, and `dep_graph_scc_info/2` (cf. Section 10.3.2). Note that `get_sdg_info/1` returns information concerning incomplete subgoals *only*.
- Inspection of the Residual Dependency Graph can be made via the predicate `get_rdg_info/[1,2]`,[21] together with dependency graph manipulation predicates such as `morph_dep_graph/3` and `dep_graph_scc_info/3`. (cf. Section 10.3.2). The predicates `get_residual/2` and `variant_get_residual/2` allow the residual program to be viewed as sets of clauses. Finally, `explain_u_val/3` can be used to indicate why a given atom has the truth value *undefined* rather than *true* or *false* (cf. Section 6.15.2).

All of these predicates, except for `get_sdg_info/1`, can be used to retrospecitively analyze any completed derivation, as long as the derivations tables have not been abolished. In addition, all of the predicates can be used to analyze an onoging derivation

[20] These predicates are not yet implemented: although `tt get_incr_sccs/[1,2]`, and `get_incr_sccs_with_deps/[2,3]` have been.

[21] This predicate is not yet implemented: although `tt get_residual_sccs/[1,2]` has been.

by suspending the derivation and then examining the computation from a subsidiary command-line interpreter. This can be especially important for long-running computaiotions or those that take a lot of space.

In XSB, a computation can be suspended in several ways, depending a user's tastes in and needs for debugging:

- By a call to `break/0`. This is usually best done by calling `break/0` as part of a handler for `timed_call/2`, but `break/0` can also be called explicitly from a program.
- By hitting ctrl-C if XSB is running in stand-alone mode
- By setting a *tripwire* as introduced below (Section 10.3.4).

10.3.4 Setting Tripwires on Tabled Derivations

A tripwire represents an unexpected property of a derivation: such as an excessive use of time or memory; an unexpected number or complexity of tabled subgoals or answers; or an unexpected number of mutually dependent tabled subgoals. Depending both on the class of a tripwire and on how XSB's flags are set, a tripwire may have different effects. Any tripwire may be treated as an error so that it throws an exception just as any other error. *Inspectable* tripwires may additionally be considered as inspection points, and when hit may suspend the derivation and create a break point.[22] In such a case, a short explanation will be made of how a tripwire was encountered, along with suggestions about how to further inspect the suspended derivation.[23] *Correctable* tripwires are a subset of inspectable tripwires for which an automatic action may be taken to remedy the situation, such as rewriting a subgoal or an answer whose size is greater than a given limit, by using subgoal abstraction or answer restraint.

Tripwires may be set in various ways: most can be set and viewed at a session level using Prolog flags, others can also be set at the predicate level via the `table/1` declaration, while still others can only be set by explicit programming. Tripwires thus

[22]Note that such a suspension makes available for inspection the state of the derivation at the point the tripwire was activated. If inspection points were implemented using ISO errors, state could only be made available at the point where the error was *caught*, whose state may differ greatly from the point where the error occurred (i.e., where the tripwire was hit).

[23] This is the default behavior for XSB: handling of tripwires can be overridden by the user, as explained later in this section.

represent a coordinated set of tools for understanding bounds on a tabled derivation, rather than a unified API.

For a tripwire `T` that can be set and viewed as a Prolog flag, the flag name has the form `tripwire(T)`, and this flag has two or more values. An *action*, designated `action(A)`, indicating the action to take such as `error`, `suspend`, or other actions; and one or more parameters, designated `limit(P)`. For example, if a user wants to be able to suspend and inspect a computation whenever it has an active recursive component (SCC) with over 100 subgoals, she can execute the following directives:

```
?- set_prolog_flag(tripwire(max_scc_subgoals),limit(100)).
?- set_prolog_flag(tripwire(max_scc_subgoals),action(suspend)).
```

We discuss various types of tripwires in turn, and provide informal guidelines for inspecting a derivation when a given tripwire has been hit.

Tripwires Based on Resource Limits

Hitting a resource tripwire reflects the fact that a derivation is taking more time or using more memory than expected. A resource tripwire is a user-imposed limit, rather than an external limit imposed by the platform or operating system, and thus differs from an ISO resource error.

Time-based resource tripwires can easily be programmed using a handler to `timed_call/2`. Time-based tripwires are inspectable, so such a handler might throw an error after a derivation has taken a certain amount of CPU time, or call `break/0` to implement periodic inspection points, or implement other periodic analytics or monitoring. The parameters for timed call can be changed whenever the timed call is suspended by `timed_call_modify/1`. See Section 6.11.1 for more details.

An inspectable memory-based resource tripwire can be set via the Prolog flag `tripwire(max_memory)`, so that the tripwire will be hit whenever XSB uses more than a given total amount of memory. This amount can be set either as an integer, representing an absolute number of kilobytes or as a floating point number indicating a percentate of the RAM of the platform upon which XSB is executing. Currently, a memory-based tripwire can only throw an error.

Guidelines for Analysis of Resource-based Tripwires

Note that the numbers and sizes below are for example purposes only. If memory limits are set to, say, a gigabyte or more of memory, and

time limits are set to several seconds, the numbers and sizes may be several orders of magnitude more than those shown below.

If a resource tripwire is hit, the best course of analysis usually starts with viewing the output of `statistics/[0,1]`.

- *Check that there are a large number of incomplete tables* This can be determined, for instance, using the output of `statistics/0`, by a line near the end of the memory table. [24] E.g.:

```
(501227 incomplete table(s) in 89 SCCs)
```

  - If there are a large number of incomplete tables, XSB's stask space is likely to be high also, since an incomplete table T needs to maintain many details of its derivation state to ensure all answers for T are returned to all calls to T. In this case, the predicates for analyzing the Subgoal Dependency Graph of the suspended derivation can be used (Section 10.3.2). *Note that, here and below, when there are a large number of incomplete tables, information returned by* `get_sdg_info/1`*, as well as by* `get_idg_info/1` *and* `get_rdg_info/1` *may need to be filtered or manipulated using the predicates in Section 10.3.2.*

- *Otherwise, Check whether there are a large number of completed subgoals.* If there are not many incomplete tables but the table space seems large, `statistics/[0,1,2]` indicates both the total number of tabled subgoals and the total number of answers. In this case. For instance, the beginning of the summary of tabling operations might contain information such as:

```
Tabling Operations
 12 subsumptive call check/insert ops: 9 producers, 3 variants,
 0 properly subsumed (0 table entries), 0 used completed table.
 0 relevant answer ident ops.  0 consumptions via answer list.
 1065417 variant call check/insert ops: 938125 producers, 127292 variants.
 46210 answer check/insert ops: 46210 unique inserts, 0 redundant.
```

  This indicates that there are 1,065,417 subgoals (complete or incomplete) tabled with call variance, and 12 subgoals tabled with call subsumption. Among all subgoals there are 46,210 ansewrs.

[24] This line is not printed out if there are no incomplete tables.

  - To understand details of overall table space usage, `table_dump/[2,3]` can be called to provide further information (Section 10.3.1).

- *Check whether the IDG is large.* In addition to simply having a large number of subgoals (incomplete or complete) and answers, the use of incremental dependency, which maintains an IDG, has an effect on memory. `statistics/[0,1]` indicates when incremental tabling is being used heavily, by a line towards the bottom of the output, such as:

  ```
  Currently 501688 incremental subgoals, 781432 dependency edges
  ```

  - When there are large numbers of incrementally tabled subgoals and dependency edges, `get_idg_info/1` (Section 10.3.2) can be used to obtain a global view of the IDG. Note that incremental tabling does require more memory for completed tables than non-incremental tabling, due to the need to retain the IDG so that tables can be updated when dynamic code changes.

- *Check whether there are a large number of answers whose truth value is* undefined. This is indicated, for instance, by lines at the end of the summary of tabling operations in `statistics/0`. [25] E.g.:

  ```
  80005 DEs in the tables (space: 3932480 bytes allocated, 3840560 in use)
  40005 DLs in the tables (space: 983200 bytes allocated, 960280 in use)
  ```

  If there is indication that there are a large number of answers with truth value *undefined*`get_rdg_info/[1]` and/or `explain_u_val/2` can be used to understand the dependencies among these answers.

Tripwires Based on Properties of a Tabled Derivation

Theoretically speaking, in a pure logic program if a tabled derivation is not terminating it is because there are an unbounded number of SLG trees, or because one or more of the SLG trees is of unbounded size. The former case indicates that a computation has an unbounded number of subgoals, while the latter indicates that one or more subgoals has an unbounded number of answers. In a similar manner, terminating but expensive derivations also may have too many subgoals, too many answers, or too many dependencies among incomplete subgoals. We consider these cases in turn.

- *There are too many tabled subgoals*

[25] These lines are not printed out if there are no incomplete tables.

– *There are a potentially unbounded number of tabled subgoals in a pure program.* In this case, there must be a potentially unbounded number of *distinct* tabled subgoals in the derivation. If this happens using ca'v'vll variance, this situation can sometimes be addressed by using call subsumption instead. However termination can *always* be ensured by using subgoal abstraction, as long as a derivation produces only a finite number of answers (cf. Section 5.5 and [67]). XSB allows subgoal abstraction to be applied based on term size either globally through the tripwire `max_table_subgoal_size`, or on a predicate-by-predicate basis through tabling directives. In other words, when a tabled subgoal, S_{big}, is called whose size is greater then the limit specified for its predicate, a tripwire is activated, and various actions can be specified. In many – perhaps most – cases, the best action is simply to abstract S_{big}. However it is also possible to suspend and inspect the derivation so that the causes that led to S_{big} can be analyzed. As a final alternative, an error can be thrown, which is the default action.

– *There are a potentially infinite number of subgoals in a program with arithmetic..* Due to the manner in which numbers are represented in XSB, XSB's size metric would permit a potentially infinite number of subgoals if numbers occurred within these subgoals. The tripwire `max_incomplete_subgoals` allows a limit to be set on the maximal number of incomplete subgoals. If a derivation exceeds that limit, the computation may be suspended, or an error thrown.

– *There are a finite but large number of tabled subgoals.* A separate problem from those above can happen as follows. If a program is written over a language that has a finite but large number of constant symbols, then a program that generates subgoals of the form

$$p(c_i, c_j, c_k, X)$$

will theoretically terminate, but may be too inefficient for practical purposes. This problem can be addressed by the tripwrite `max_incomplete_subgoals` just discussed, but it is often helpful to have a different size limit for cases where there are a large number of subgoals within the *same* SCC.

The tripwire `max_scc_subgoals` allows such a limit to be set on the maximal number of incomplete subgoals in any recursive component. If a derivation exceeds that limit, the computation may be suspended, or an error thrown. This situation is similar to that of simply having too many incomplete subgoals, but may suggest a different focus when analyzing a suspended computation. In addition, the number of dependencies can rise

quadratically with the absolute size of an SCC. As a result, it often makes sense to have different limit for the size of a single (incomplete) SCC and for the number of incomplete subgoals overall.

Guidelines for Analysis If the computation is producing too many tabled subgoals the suspension may have been triggered by one of the tripwires: `max_table_subgoal_size`, `max_incomplete_subgoals` or `max_scc_size`. In any of these cases, the suspension or error message will indicate the tripwire that has been hit. The number of incomplete subgoals can be seen from the output of `statistics/0`. The inspection predicates of Section 10.3.2 can be used to examine these subgoals and their dependencies.

- *There are too many tabled answers.* The approaches to this situation are similar in spirit to the cases of too many subgoals.
  - *One or more subgoals has an unbounded number of answers.* In this case, termination can be ensured by using radial restraint, which abstracts answers in a manner that is sound with respect to the well-founded semantics and can ensure that a derivation will produce only a finite number of answers (cf. Section 5.5 and[34]), XSB allows radial restraint to be applied based on term size in two ways. First, restraint can be applied globally through the tripwire `max_table_answer_size`. Second, XSB allows restraint to be declared at a predicate-by-predicate basis. In other words, when an answer, A_{big}, is to be added to a table, and its size is greater then the limit specified for its predicate, a tripwire is activated, and various actions can be specified. In many cases, the best action is simply to abstract A_{big}, which gives it the truth value *undefined* for that answer. However it is also possible for the derivation to be suspended so that the causes that led to A_{big} can be analyzed. As a final alternative an error can be thrown; this is the default action for XSB.
  - *There are a finite but large number of tabled answers.* As with subgoals, checking for the depth of an answer may not catch certain causes of inefficency. If a program is written over a language that has a large number of constant symbols, then a program that generates answers of the form

    $$p(c_i, c_j, c_k, c_l)$$

    will theoretically terminate, but may be too inefficient for practical purposes.

    To address such situations, XSB has the tripwire `max_answers_for_subgoal` which is hit if any subgoal has more than the specified number of answers.

If a derivation exceeds that limit, the subgoal may be eagerly completed while maintaining soundness, the the computation may be suspended, or an error thrown (cf. Section 5.5.3 for a discussion of how a subgoal may be completed early while preserving soundness).

Guidelines for Analysis If the computation is producing too many tabled answers, a suspension may be triggered by one of the tripwires: `max_table_answer_size`, or `max_answers_for_subgoal`. In any of these cases, the suspension (or error message) will indicate the subgoal whose number of answers hit the tripwire. The number and shape of answers for that subgoal and others can be viewed through the `table_dump` library (Section 10.3.1). If some answers are undefined, the predicates `get_rdg_info/1` and `explain_u_val/2` can be used to explore dependencies among the answers. In addition, dependency graph analysis based on the predicates `get_sdg_info/1` and `get_idg_info/1` can help locate areas of code that caused the profusion of answers.

The Suspend Action for Flag-Based Tripwires

If the action `suspend` is specified for a given flag-based tripwire T, (i.e., a tripwire other than one based on `timed_call/2`), then hitting T causes XSB to take a given action. By default, this action is to enter a break level from which the computation can be inspected. (This default action for `suspend` can be overridden.) A preamble to the break level is also presented that describes certian values pertaining to the state of the computation, along with an attempt to summarize what the situation means. The preamble for the tripwire `max_incomplete_subgoals` appears as follows.

```
There are currently 11 incomplete tabled subgoals, which exceeds the limit set by the
flag 'max_incomplete_subgoals'.  These subgoals are in 11 separate recursive components.

The number of incomplete recursive components is close to the number of incomplete
subgoals, and furthermore, nearly all of these recursive components are trivial.

This information indicates a likelihood that the program is performing some sort of
structural recursion using tabling (i.e., recursing through a list, performing numeric
iteration, etc.)

To remedy, check that the recursion is well-founded.  If so, consider executing the
recursion using a non-tabled predicate, or consider using hash-cons tabling to reduce
the space required for tables, then increase the value of the flag.
```

```
* To continue, reset the flag and then type Ctrl-d
* To abort the suspended derivation, enter the command 'abort/0'
* To inspect the incomplete tabled subgoals, enter the command 'print_sdg_info/0'
```

From within a break level thrown by a tripwire, a user can perform most queries and commands with two important exceptions.

- No tabled goals or subgoals may be executed. Of course once the break is exited, the original comptuation can be continued, and tabled subgoals can be executed as usual.
- Reconsulting a file containing code upon which the original query depends may throw an error.

There are situations where the action of breaking to allow a user can inspect a query isn't suitable – if XSB is embedded into a process, for instance, or is part of some other application. In such a case, the user can override XSB's default behavior by asserting into the `user` module a *tripwire handler*, which will be executed when a given tripwire is hit. For instance, the tripwire `max_table_subgoal_size` would use the tripwire `max_table_subgoal_size_user_handler/0`. Subject to the constraints above, the handler may perform any actions desired, including increasing the tripwire limit, adjusting its action, changing runtime tabling properties, or simply writing to a log.[26]

A handler that is called when a tripwire is hit has some resemblences to a handler called when an error is caught, but there are important differences. A condition triggering a tripwire might or might not reflect an error in a program. So the handler is invoked when the tripwire is hit, rather than after exiting portions of that derivation as is the case when an error is caught. If the user wants to exit some or all of a derivation when a tripwire is hit, it is simple to change the tripwire's action to error, and arrange to catch the error with a suitable handler.

Summary of Flag-Based Tripwires

Each of the tripwires described below can be queried and set via prolog flags. For instance the query

[26] As a further example, when XSB supports the language Ergo the tripwire may set on forest logging to support termination analysis via the Terminyzer tool.

```
| ?- current_prolog_flag(tripwire(max_table_subgoal_size),Property).

Property = limit(12)
Property = action(error);
```

indicates the current values of this tripwire. (Also see the description of Prolog flags in Section 6.12.)

- `max_table_subgoal_size`.
  - *Limit:* The maximum size of a given argument in a subgoal (cf. Section 5.5.1). A limit of `0` means that this tripwire is disabled.
  - *Possible actions:* `abstract`, `suspend` or `error`.
  - *Default.* Limit: `0`; Action: `error`.
  - *User handler for suspend*: `max_table_subgoal_size_handler`.
- `max_incomplete_subgoals`
  - *Limit:* The maximum number of subgoals that are incomplete at any given time. A limit of `0` means that this tripwire is disabled.
  - *Possible actions:* `suspend`, `error` or `warning`.
  - *Default.* Limit: `0`; Action: `error`.
  - *User handler for suspend*: `max_incomplete_subgoals_user_handler`.
- `max_scc_subgoals`
  - *Limit:* The maximum number of subgoals that are incomplete, and are in the same SCC at any given time. A limit of `0` means that this tripwire is disabled.
  - *Possible actions:* `suspend`, `error` or `warning`.
  - *Default.* Limit: `0`; Action: `error`.
  - *User handler for suspend*: `max_scc_subgoals_user_handler`.
- `max_table_answer_size`
  - *Limit:* The maximum size of a given argument of an answer (cf. Section 5.5.1). A limit of `0` means that this tripwire is disabled.

  - *Possible actions:* `abstract`, `suspend`, or `error`.
  - *Default.* Limit: `0`; Action: `error`.
  - *User handler for suspend*: `max_table_answer_size_user_handler`.

- `max_answers_for_subgoal`
  - *Limit:* The maximum number of answers for any single tabled subgoal. A limit of `0` means that this tripwire is disabled.
  - *Possible actions:* `suspend`, `error`, `warning` or `complete_soundly`.
  - *Default.* Limit: `0`; Action: `error`.
  - *User handler for suspend*: `max_answers_for_subgoal_user_handler`.

- `max_memory`
  - *Limit:* If an integer, the limit is the maximum amount of memory allowed for a computation, in kilobytes. If a floating-point number, the limit is the proportion of RAM for the machine on which XSB is running. A limit of `0` means that this tripwire is disabled.
  - *Possible actions:* `suspend` or `error`.
  - *Default.* Limit: `0`; Action: `error`.
  - *User handler for suspend*: `max_memory_user_handler`.

Chapter 11

Definite Clause Grammars

11.1 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 is executable because it is just a notational variant of a logic rule 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 sequence of symbols or character codes, analyses some initial portion of that list, and produces the remaining portion (possibly enlarged) as output for further analysis. In XSB, the exact form of this sequence is determined by whether XSB's *DCG mode* is set to use tabling or not, as will be discussed below. In either case, 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 normal rule during parsing. 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.

The syntax of DCGs is orthogonal to whether tabling is used for DCGs or not. An overview of DCG syntax 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 standard 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 UTF-8 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, Semicontext (formerly called push-back lists) is 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.

[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.

11.2 Translation of Definite Clause Grammar rules

In this section we informally describe the translation of DCG rules into normal rules in XSB. Each grammar rule is translated into a Prolog clause as it is consulted or compiled. This is accomplished through a general mechanism of defining the hook predicate `term_expansion/2`, by means of which a user can specify any desired transformation to be done as clauses are read by the reader of XSB's parser. This DCG term expansion is as follows:

A DCG rule such as:

```
p(X) --> q(X).
```

will be translated (expanded) into:

```
p(X, Li, Lo) :-
      q(X, Li, Lo).
```

If there is more than one non-terminal on the right-hand side, as in

```
p(X, Y) --> q(X), r(X, Y), s(Y).
```

the corresponding input and output arguments are identified, translating into:

```
p(X, Y, Li, Lo) :-
      q(X, Li, L1),
      r(X, Y, L1, L2),
      s(Y, L2, Lo).
```

Terminals are translated using the predicate `'C'/3` (See section 11.3 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.

Semicontext (or a push-back list, which is 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)
      ).
```

11.2.1 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 3.10.5) 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, the efficiency of tabling for DCGs depends on the representation of the input and output sequences used, a topic to which we now turn.

Consider the expanded DCG rule from the previous section:

```
p(X, S0, S) :-
      'C'(S0, go, S1),
      'C'(S1, to,S2),
      q(X, S2, S3),
      'C'(S3, stop, S).
```

In a Prolog system, each input and output variable, such as `S0` or `S` is bound to a variable or a difference list. In XSB, this is called *list mode*. Thus, to parse *go to lunch stop* the phrase would be presented to the DCG rule as a list of tokens `[go,to,lunch,stop]` via a call to `phrase/3` such as:

```
phrase(p(X),[go,to,lunch,stop]).
```

or an explicit call to `p/3`, such as:

```
p(X,[go,to,lunch,stop|X],X).
```

Terminal elements of the sequence are consumed (or generated) via the predicate `'C'/3` which is defined for Prolog systems as:

```
'C'([Token|Rest],Token,Rest).
```

While such a definition would also work correctly if a DCG rule were tabled, the need to copy sequences into or out of a table can lead to behavior quadratic in the length of the input sequence (See Section 5.2.5). As an alternative, XSB allows a mode of DCGs that defines `'C'/3` as a call to a Datalog predicate `word/3` :

```
'C'(Pos,Token,Next_pos):- word(Pos,Token,Next_pos).
```

assuming that each token of the sequence has been asserted as a `word/3` fact, e.g:

```
word(0,go,1).
word(1,to,2).
word(2,lunch,3).
word(3,stop,4).
```

The above mode of executing DCGs is called *datalog mode*.

`word/3` facts are asserted via a call to the predicate `tphrase_set_string/1`. Afterwards, a grammar rule can be called either directly, or via a call to `tphrase/1`. To parse the list `[go,to,lunch,stop]` in datalog mode using the predicate `p/3` from above, the call

```
tphrase_set_string([go,to,lunch,stop])
```

would be made, afterwards the sequence could be parsed via the goal:

```
tphrase(p(X)).
```

or

```
p(X,0,F).
```

To summarize, DCGs in list mode have the same syntax as they do in datalog mode: they just use a different definition of `'C'/3`. Of course tabled and non-tabled DCGs can use either definition of `'C'/3`. Indeed, this property is necessary for tabled DCG predicates to be able to call non-tabled DCG predicates and vice-versa. At the same time,tabled DCG rules may execute faster in datalog mode, while non-tabled DCG rules may execute faster in list mode.

Finally, we note that the mode of DCG parsing is part of XSB's state. XSB's default mode is to use list mode: the mode is set to datalog mode via a call to `tphrase_set_string/3` and back to list mode by a call to `phrase/2` or by a call to `reset_dcg_mode/0`.

11.3 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 non-variable 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.

`tphrase(+Phrase)`

This predicate succeeds if the current database of `word/3` facts can be parsed via a call to the term expansion of `+Phrase` whose input argument is set to `0` and whose output argument is set to the largest `N` such that `word(_,_,N)` is currently true.

The database of `word/3` facts is assumed to have been previously set up via a call to `tphrase_set_string/1` (or variant). If the database of `word/3` facts is empty, `tphrase/1` will abort.

`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 non-variable 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 default 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 augment the default transformations by asserting clauses for the predicate `term_expansion/2` to `usermod`. After `term_expansion(Term_a,Term_b)` is asserted, then if a consulted file contains a clause that unifies with `Term_a` the clause will be transformed to `Term_b` before further compilation. (`Term_b` can be a list of clauses, so `term_expansion` can transform a single clause into a sequence of clauses.) `expand_term/2` calls user clauses for `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.

Example: Suppose the following clause is asserted:

```
?- assert(term_expansion(foo(X),bar(X))).
```

and that the file `te.P` contains the clause `foo(a)` then the clause will automatically be expanded upon consulting the file:

```
| ?- [te].
[Compiling /Users/macuser/te]
[te compiled, cpu time used: 0.0170 seconds]
[te loaded]

yes
| ?- bar(X).

X = a

yes
| ?- foo(X).
++Error[XSB/Runtime/P]: [Existence (No procedure usermod : foo / 1 exists)] []
Forward Continuation...
```

However, `read/[1,2]` does not automatically perform term expansion

```
| ?- use_module(standard,[expand_term/2]).

yes
| ?- read(X),expand_term(X,Y).
```

```
foo(a).

X = foo(a)
Y = bar(a)

yes
```

`'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 preempted.

`tphrase_set_string(+List)`
This predicate

1. abolishes all tables;
2. retracts all `word/3` facts from XSB's store; and
3. asserts new `word/3` facts corresponding to `List` as described in Section 11.2.1.

implicitly changing the DCG mode from list to datalog.

`tphrase_set_string_keeping_tables(+List)` module: `dcg`
This predicate is the same as `tphrase_set_string`, except it does not abolish any tables. When using this predicate, the user is responsible for explicitly abolishing the necessary tables.

`tphrase_set_string_auto_abolish(+List)` module: `dcg`
This predicate is the same as `tphrase_set_string`, except it abolishes tables that have been indicated as dcg-supported tables by a previous call to `set_dcg_supported_table/1`.

`set_dcg_supported_table(+TabSkel)` module: `dcg`

This predicate is used to indicate to the DCG subsystem that a particular tabled predicate is part of a DCG grammar, and thus the contents of its table depends on the string being parsed. `TabSkel` must be the skeleton of a tabled predicate. When `tphrase_set_string_auto_abolish/1` is called, all tables that have been indicated as DCG-supported by a call to this predicate will be abolished.

`dcg(+DCG_Rule, ?Prolog_Clause)` module: `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`.

11.4 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.

Finally, we note that the choice of DCG style is orthogonal to whether the DCG mode is list or datalog.

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.

Chapter 12

Exception Handling

We define the term *exceptions* as errors in program execution that are handled by a non-local change in execution state. Exception handling in XSB is ISO-compatible, and has been extended to handle tabled evaluations.

12.1 The Mechanics of Exception Handling

We address the case of non-tabled evaluations before discussing the extensions for tabling.

12.1.1 Exception Handling in Non-Tabled Evaluations

The preferred mechanism for dealing with exceptions in XSB is to use the predicates `catch/3` and `default_user_error_handler/1` together with one of XSB's error predicates (such as `misc_error/1`). These predicates are ISO-compatible, and their use can give a great deal of control to exception handling. At a high level, when an exception is encountered an error term T is *thrown*. In a non-tabled Prolog program, throwing an error term T causes XSB to examine its choice point stack until it finds a *catcher* that unifies with T. This catcher then calls a *handler*. If no explicit catcher for T exists, a default handler is invoked, which usually results in an abort, and returns execution to the top-level of the interpreter, or to the calling C function.[1]

A handler is set up when `catch(Goal,Catcher,Handler)` is called. At the time

[1]Starting in Version 3.5.1, XSB uses the ISO compliant `error/2` for error terms, rather than `error/3` as in previous versions.

of the call, a continuation is saved (i.e. a Prolog choice point), and `Goal` is called. If no exceptions are encountered, answers for `Goal` are obtained as usual. However, within the execution of `Goal`, an exception might be thrown by calling a Prolog predicate in the `error_handler` module, or by executing a C-level error function. [2] As mentioned above, when an error is thrown in an environment Env, XSB searches for an ancestor Env_{anc} of Env in which `catch/3` was called, and in which the catcher (second argument) unifies with `Error`. If such an ancestor is found, program execution reverts to the ancestor and all intervening choice points are removed. The catcher's `Handler` goal is called and the exception is thereby handled. On the other hand, if no ancestor in the user's program was called using `catch/3` the exception is handled via the handler associated with XSB's goal interpreter at the top-level command line or C API. This top-level handler checks whether a clause with head `default_user_error_handler(Term)` has been asserted, such that `Term` unifies with `Error`. If so, this handler is executed. If not, XSB's default system error handler in invoked an error message is output and execution returns to the top level of the interpreter.

The following, somewhat fanciful, example helps clarify these concepts [3]. Consider the predicate `userdiv/2` (Figure 12.1) which is designed to be called with the first argument instantiated to a number. A second number is then read from a console, and the first number is divided by the second, and unified with the second argument of `userdiv/2`. By using `catch/3` and `throw/1` together the various types of errors can be caught.

The behavior of this program on some representative inputs is shown below.

```
| ?- userdiv(p(1),F).
++Error[XSB/Runtime/P]: [Type (p(1) in place of number)]  in arg 1 of predicate userdiv1/2
Forward Continuation...
... machine:xsb_backtrace/1
... error_handler:type_error/4
... standard:call/1
... x_interp:_$call/1
... x_interp:call_query/1
... standard:call/1
... standard:catch/3
... x_interp:interpreter/0
... loader:ll_code_call/3
... standard:call/1
```

[2] A user-defined error type is desired, the Prolog predicate `throw/1` can also be called directly.

[3] Code for this example can be found in `$XSBDIR/examples/exceptions.P`.

```
:- import error_writeln/1 from standard.
:- import type_error/4 from error_handler.

userdiv(X,Ans):-
        catch(userdiv1(X,Ans),mydiv1(Y),handleUserdiv(Y,X)).

userdiv1(X,Ans):-
        (number(X) -> true; type_error(number,X,userdiv1/2,1)),
        write('Enter a number: '),read(Y),
        (number(Y) -> true ; throw(mydiv1(error1(Y)))),
        (Y < 0 -> throw(mydiv1(error2(Y))); true),
        (Y =:= 0 -> throw(error(zerodivision,userdiv/1,[])); true),
        Ans is X/Y.

handleUserdiv(error1(Y),_X):-
        error_writeln(['a non-numeric denominator was entered in userdiv/1: ',Y]),fail.
handleUserdiv(error2(Y),_X):-
        error_writeln(['a negative denominator was entered in userdiv/1: ',Y]),fail.
```

Figure 12.1: The `userdiv/1` program

```
... standard:catch/3

no
| ?- userdiv(3,F).
Enter a number: foo.
a non-numeric denominator was entered in userdiv/1: foo

no
|| ?- userdiv(3,F).
Enter a number: -1.
a negative denominator was entered in userdiv/1: -1

no
| ?- userdiv(3,Y).
Enter a number: 2.

Y = 1.5000

yes
```

Note, however the following behavior.

```
| ?- userdiv(3,F).
Enter a number: 0.
++Error[XSB/Runtime/P] uncaught exception: error(zerodivision,userdiv / 1)
Aborting...
```

By examining the program above, it can be seen that if `p(1)` is entered, the predicate `type_error/3` is called. `type_error/3` is an XSB mechanism to throw a type error from Prolog. Error terms thrown by system predicates such as `type_error/3` are in XSB's *standard error format*, which is ISO-compatable and which may encode useful information. For instance, the type error thrown in the above example is known to XSB's default system error handler which prints out a message along with a *backtrace* that indicates the calling context in which the error arose (this behavior can be controlled: see Section 12.5). Alternately, in the second case, when `-1` is entered, the (non-standard) error term `mydiv1(error2(-1))` is thrown, which is caught within `userdiv/2` and handled by `handleUserdiv/2`. Finally, when `0` is entered for the denominator, an error term of the form `error(zerodivision,userdiv/1)` is thrown, and this term does not unify with the second argument of the `catch/3` literal in the body of `userdiv/1`, or with any error in standard format. The error is instead caught

by XSB's default system error handler which prints an uncaught exception message and aborts to the top level of the interpreter.

XSB has two default system error handlers: one used when XSB is called as a stand-alone process, and another when XSB is embedded in a process. Each recognizes the same error formats (see Section 12.2), and handles the rest as uncaught exceptions. However, there may be times when an application requires special default handling: perhaps the application calls XSB from through a socket, so that aborts are not practical. As another example, perhaps XSB is being called from a graphical user interface via InterProlog [10] or some other interface, so that in addition to a special abort handling, one would like to display an error window. In these cases it is convenient to make use of the dynamic predicate `default_user_error_handler/1`. `default_user_error_handler/1` is called immediately before the default system error handler, and after it is ascertained that no catcher for an error term is available via a `catch/3` ancestor.

It is important to note that the system error handlers catch errors only in the main thread, and do not affect errors thrown by goals executed by `thread_create/[2,3]`. Error terms thrown by goals executed by non-detached threads are stored internally, and can be obtained by `thread_join/2`. Error terms thrown by detached threads are lost when the thread exits, so that any error handling for a detached thread should be performed within the thread itself. See Chapter 7 for further information.

Accordingly, suppose the following clause is asserted into `usermod`:

```
?- assert((default_user_error_handler(error(zerodivision,Pred)):-
        error_writeln(['Aborting: division by 0 in: ',Pred]))).
```

The behavior will now be

```
| ?- userdiv(4,F).
Enter a number: 0.
Aborting: division by 0 in: userdiv / 1
```

The actions of `catch/3` and `throw/1` resemble that of the Prolog cut in that they remove choice points that lie between a call to `throw/1` and the matching `catch/3` that serves as its ancestor.

The predicate `call_cleanup/2` (cf. Section 6.11) can be used with `catch/3`, since the goal `call_cleanup(Goal,Cleanup)` executes `Cleanup` whenever computation of `Goal` is completed, whether because `Goal` has thrown an exception, has failed, or has succeeded with its last answer. `call_cleanup/2` can thus be used to release resources created by `Goal` (such as streams, mutexes, database cursors, etc.). However, if `Goal`

throws an exception, `call_cleanup/2` will re-throw the exception after executing cleanup.

12.1.2 Exception Handling in Tabled Evaluation

The exception handling as previously described requires extensions in order to work well with tabled predicates. First, if an *unhandled* exception is thrown duing evaluation of a tabled subgoal S and S is not completed, the table for S is not meaningful and should be removed. (Tables that have been completed are not affected by exceptions.) Accordingly, the user will sometimes see the message:

```
Removing incomplete tables...
```

written to standard feedback. But what about exceptions that are *caught* during the computation of S?

The proper action to take in such a case is complicated by the scheduling mechanism of tabling which, as discussed in Chapter 5, is more complex than in Prolog. Rather than a simple depth-first search, as in Prolog, tabled evaluations effectively perform a series of fixed-point computations for various sets of mutually dependent subgoals, which are termed *SCCs* [4]. In fact, a tabled evaluation can be seen as a tree of SCCs (in batched evaluation) or a chain of SCCs (in local evaluation). In a tabled evalution XSB's throw mechanism searches for the nearest catcher C among its ancestors

- whose first argument unifies with the thrown error; and
- where C is between SCCs: that is where the set of subgoals that depend on C is disjoint from the set of subgoals upon which C depends. We term this the *SCC restriction* for exception handling.

This behavior can be best understood by an example. Consider the query `a(X)` to the program in Figure 12.2 which has the following output:

```
| ?- a(X).
a_calling_b
b_calling_a
```

[4] This term is used since sets of mutually dependent subgoals are formally modelled as (approximate) *Strongly Connected Components* within a dependency graph.

```
:- table a/1, b/1, c/1,d/1.
a(X):- writeln(a_calling_b),b(X).

b(X):- writeln(b_calling_a),a(X).
b(X):- writeln(b_calling_c),catch(c(X),_,(writeln(handled_1),fail)).

c(X):- writeln(c_calling_d),d(X).
c(X):- writeln(c_aborting),abort.
d(X):- writeln(d_calling_c),catch(c(X),_,(writeln(handled_2),fail)).
```

Figure 12.2: A program to illustrate exception handling in tabled evaluations

```
b_calling_c
c_calling_d
d_calling_c
c_aborting
Removing incomplete tables...
handled_1
```

Note that there are 2 SCCs, $\{a(X), b(X)\}$ and $\{c(X), d(X)\}$. When the `abort` is called in the body of `c(X)` the catch in the body of `d(X)` is its nearest ancestor; however this catch is skipped over, and the catch in the body of `b(X)` takes effect. This catch is between the SCCs – the first SCC depends on it, but the second doesn't. Due to the SCC restriction, the actual behavior of exception handling with tabling is thus somewhat less intuitive than in Prolog. If this restriction were lifted, there would be no guarantee that there existed a unique catch that was the closest ancestor of an exception.

While the above mechanism offers a great deal of flexibility, for many cases the best approach to exception handling is to keep it simple.

1. Use catches when there will be no tabled subgoal between an exception and its catcher. For instance, sometimes it may be annoying to have `atom_codes/2` throw an exception rather than failing, if given an integer in its first argument. This can be addressed by the predicate

   ```
   my_atom_codes(X,Y):-
      catch(atom_codes(1,B),error(type_error(A,B),C,D),writeln(E)).
   ```

which, for a type error, does not interact with tabling in any way.

2. Similarly, if only subgoals to *completed* tables occur between an exception and its catcher, exception handling behaves just as in case 1).

3. Otherwise, abort the entire tabled computation and handle it from there. (Unless you really know what you're doing!)

Obtaining Information about a Tabled Computation after an Exception is Thrown

XSB backtraces (Section 12.5) provide information about the context in which error is thrown, but in a tabled computation additional information is available. If the Prolog flag `exception_pre_action` is set to `print_incomplete_subgoals` (its default setting is `none`), then when an exception is thrown, incomplete tables and their SCC information at the time an exception is thrown are printed to a file. The file may be obtained through the predicate `get_scc_dumpfile/1` in the module `tables`. No file is generated unless the exception is thrown over at least one incomplete table.

12.2 XSB's Standard Format for Errors

All exceptions that occur during the execution of an XSB program can be caught. However, by structuring error terms in a consistent manner, different classes of errors can be handled much more easily by handlers, both system- and user-defined. This philosophy partly underlies the ISO Standard for defining classes of Prolog errors [37]. While the ISO standard defines various types of errors and how they should arise during execution of ISO Prolog predicates, it only partially defines the actual error terms a system should use. The ISO format can be represented as:

```
error(Tag,Context),
```

where `Tag` is specific to each class of error, while `Context` is implementation-dependent.[5]

[5] If a program catches errors itself, `error/2` may need to be imported from `error_handler`.

12.2.1 Error Tags

In XSB, the ISO-compliant values for `Tag` are given below.

`domain_error(Valid_type,Culprit)` is the tag for an ISO domain error, where `Valid_type` is the domain expected and `Culprit` is the term observed. Various ISO predicates may have specific domains for input values; and in addition unlike types, domains can be user-defined.

`evaluation_error(Flag)` is the tag for an ISO evaluation error (e.g. overflow or underflow), and `Flag` is the type of evaluation error encountered (e.g., `undefined`, if an arithmetic function is undefined for a given input).

`existence_error(Type,Culprit)` is the tag for an ISO existence error, where `Type` is the type of a resource (e.g., a predicate, stream, attribute handler, etc.) and `Culprit` is the term observed.

`instantiation_error` is the tag for an ISO instantiation error.

`permission_error(Op,Obj_type,Culprit)` is the tag for an ISO permission error, when an operation `Op` was applied to an object of type `Obj_type`, but `Culprit` was observed.

`representation_error(Flag)` is the tag for an ISO representation error (e.g., the maximum arity of a predicate has been exceeded), and `Flag` is the type of representation error encountered.

`resource_error(Flag)` is the tag for an ISO resource error (e.g. allowed memory has been used, or too many files have been opened), and `Flag` is the type of resource error encountered.

`syntax_error` and `syntax_error(Culprit)` are alternate tags for an ISO syntax error, where `Culprit` denotes a syntactically-incorrect sequence of tokens.

`system_error(Flag)` is the tag for an ISO system error, and `Flag` is the type of system error encountered.

`type_error(Valid_type,Culprit)` is the tag for an ISO type error, where `Valid_type` is the type expected and `Culprit` is the term observed. As opposed to domain errors, type errors should be used for checks of Prolog types only (i.e. integers, floats, atoms, etc.)

In addition, XSB also makes use of two other classes of errors.

`table_error` and `type_error(Subtype)` are the tags for an error arising when using XSB's tabling mechanism, when the condition giving rise to the error does npt easily fit under one of the above classes.

`error(thread_cancel,Id)` is the format of an error ball for a thread that has been cancelled by XSB thread `Id` (See Chapter 7 for details on thread cancellation.)

`misc_error` is the tag for an error that is not otherwise classified.

`misc_error(Level)` is the tag for `abort/0-1`, and is used to allow users to abort a suspended query. `Level` is the break level at which the error was thrown.

In Version 3.8 of XSB, errors for ISO predicates are usually, but always ISO-compliant (we still have a few non-compliant errors to catch). However, when XSB determines it is out of available system memory, recovering from such an error may be difficult at best. Accordingly the computation is aborted in the sequential engine, or XSB exits in the multi-threaded engine [6].

12.2.2 XSB-Specific Information in Error Terms

XSB also encodes other information in error terms, which may vary with the error thrown, the form in which XSB was compiled, and the version of XSB. In addition, the specifics of how the information is represented may vary, so that this information should always be represented through the access methods described in this section along with Section 12.3.2.

`Message` describes the error in human-readable format. Messages are present in all of XSB's system errors, and can be obtained through `xsb_error_get_message/2`.

`Goal` represents tabled goal that is closest to the environment of the thrown error. It is present in some, but not all error terms thrown by XSB and can be obtained as a term through `xsb_error_get_goal/2`, and as an atom through `xsb_error_get_goalatom/2`.

`Thread Id` is an atom `'th <tid>'` indicating the id of the thread that threw the error. Thread Id information is only present when using the multi-threaded version of XSB, and even in that version is not present in all error terms. The predicate `xsb_error_get_tid/2` can be used to obtain this information if present.

[6] This does not include overflowaing a memory limit specified by the flag `max_memory`.

`Backtrace` represents the stack of the forward continuations in the execution stack at the time the error was thrown. Backtraces are present by default in all XSB system error terms, and are described in Section 12.5. They may be obtained from an error term using the predicate `xsb_error_get_backtrace/2`.

12.3 Predicates to Throw and Handle Errors

12.3.1 Predicates to Throw Errors

XSB provides a variety of predicates that throw errors [7]. In general, we recommend the use of predicates such as `domain_error/4` over the direct use of `throw/1` when possible.

`throw(+ErrorTerm)` ISO

Throws the error `ErrorTerm`. Execution traverses up the choice point stack until a goal of the form `catch(Goal,Term,Handler)` is found such that `Term` unifies with `ErrorTerm`. In this case, `Handler` is called. If no catcher is found in the main thread, the system looks for a clause of `default_user_error_handler(Term)` such that `Term` unifies with `ErrorTerm` — if no such clause is found the default system error handler is called. In a non-main joinable thread, the error term is stored internally and the thread exits; in a detached thread, the thread exits with no action taken. `throw/1` is most useful in conjunction with specialized handlers for new types of errors not already supported in XSB.

`domain_error(+Valid_type,-Culprit,+Predicate,+Arg)` module: `error_handler`

Throws a domain error. Using the default system error handler (with the Prolog flag `backtrace_on_error` set to off) an example is

```
domain_error(posInt,-1,checkPosInt/3,3).
++Error[XSB/Runtime/P]: [Domain (-1 not in domain posInt)] in arg 3 of predicate
checkPosInt/3
```

`evaluation_error(+Flag,+Predicate,+Arg)` module: `error_handler`

Throws an evaluation error. Using the default system error handler (with the Prolog flag `backtrace_on_error` set to off) an example is

[7] C functions for throwing terms and ISO-style errors are described in Volume 2, Chapter 3 *Foreign Language Interface*.

```
evaluation_error(zero_divisor,unidiv/1,2).
++Error[XSB/Runtime/P]: [Evaluation (zero_divisor)] in arg 2 of predicate unidiv/2
```

existence_error(+Object_type,?Culprit,+Predicate,+Arg) module: error_handler

Throws an existence error. Using the default system error handler (with the Prolog flag backtrace_on_error set to off) an example is

```
existence_error(file,'myfile.P','load_intensional_rules/2',2).
++Error[XSB/Runtime/P]: [Existence (No file myfile.P exists)]  in arg 2 of predicate
load_intensional_rules/2
```

instantiation_error(+Predicate,+Arg,+State) module: error_handler

Throws an instantiation error. Using the default system error handler, an example (with the Prolog flag backtrace_on_error set to off) is

```
?- instantiation_error(foo/1,1,nonvar).
++Error[XSB/Runtime/P]: [Instantiation]  in arg 1 of predicate foo/1: must be nonvar
```

permission_error(+Op,+Obj_type,?Culprit,+Predicate) module: error_handler

Throws a permission error. Using the default system error handler, an example (with the Prolog flag backtrace_on_error set to off) is

```
| ?- permission_error(write,file,'myfile.P',foo/1).
++Error[XSB/Runtime/P]: [Permission (Operation) write on file: myfile.P]  in foo/1
```

representation_error(+Flag,+Predicate,+Arg) module: error_handler

Throws a representation error. Using the default system error handler, an example (with the Prolog flag backtrace_on_error set to off) is

```
representation_error(max_arity,assert/1,1).
++Error[XSB/Runtime/P]: [Representation (max_arity)] in arg 1 of predicate assert/1
```

resource_error(+Flag,+Predicate) module: error_handler

Throws a resource error. Using the default system error handler (with the Prolog flag backtrace_on_error set to off) and example is

```
resource_error(open_files,open/3)
++Error[XSB/Runtime/P]: [Resource (open_files)] in predicate open/3
```

`type_error(+Valid_type,-Culprit,+Predicate,+Arg)` module: `error_handler`

Throws a type error. Using the default system error handler, an example (with the Prolog flag `backtrace_on_error` set to off) is

```
| ?- type_error(atom,f(1),foo/1,1).
++Error[XSB/Runtime/P]: [Type (f(1) in place of atom)]  in arg 1 of predicate foo/1
```

`misc_error(+Message)` module: `error_handler`

Throws a miscellaneous error that will be caught by the default system handler at the current break level. Usually, miscellaneous errors should only be thrown when the cases above are not applicable, and the type of error is not of interest for structured error handling. Such situations occur can occur for instance in debugging, during program development. or for other reasons. Conceptually, `misc_error/1` differs from `abort/0-1` only when called from a break level.

`abort(+Message)`

`abort`

Throws a type of miscellaneous error that will be caught by the default system handler at the top level of the command-line intepreter (CLI), but not at the current break level (if any). This type of exception can be useful, for instance if a long-running query is interrupted by a ctrl-C, which suspends the query and starts a new break level for XSB. If a regular error is thrown, it will be caught by the CLI for the current break level, and so does not provide a way to abort the top-level query. On the other hand, the exception thrown by `abort/0-1` will only be caught by the top-level CLI.

12.3.2 Predicates used in Handling Errors

For best results, output for handling errors should be sent to XSB's standard error stream using the alias `user_error` or one of the predicates described below.

`catch(?Goal,?CatchTerm,+Handler)` ISO

Calls `Goal`, and sets up information so that future throws will be able to access `CatchTerm` under the mechanism mentioned above. `catch/3` does not attempt to clean up system level resources with the exception of incomplete tables, which are abolished as discussed in Section 12.1.2. However, it is left up to the handler to close any open files, reset current input and output, and so on [8].

[8] cf. the default system error handler, which performs these functions, if needed.

`default_user_error_handler(?CatchTerm)`
: Handles any error terms that unify with `CatchTerm` that are not caught by invocations of `catch/3`. This predicate closes open tables and release mutexes held by the calling thread, but does not attempt to clean up other system level resources, which is left to the handler.

`error_write(?Message)` module: `standard`
`error_writeln(?Message)` module: `standard`
: Utility routines for user-defined error catching. These predicates output `Message` to XSB's `STDERR` stream, rather than to XSB's `STDOUT` stream, as does `write/1` and `writeln/1`. In addition, if `Message` is a comma list, the elements in the comma list are output as if they were concatenated together. Each of these predicates must be implicitly from the module `standard`.

`xsb_error_get_message(Error,Message)` module: `error_handler`
: Obtains the message associated with an error in XSB's standard format. All errors in standard format have messages.

`xsb_error_get_goal(Error,?Goal` module: `error_handler`
: Obtains the goal (represented as a Prolog term), if any, from an error term that is in XSB's standard format. If the error term has no goal, the predicate fails.

`xsb_error_get_goalatom(Error,?GoalAtom` module: `error_handler`
: Obtains the goal (represented as a Prolog atom), if any, from an error term that is in XSB's standard format. If the error term has no goal, the predicate fails. This routine is slightly more efficient than `xsb_error_get_goal/2`.

`xsb_error_get_tid(Error,?Tid)` module: `error_handler`
: Obtains the atom `'th <tid>'` indicating the id of the thread that threw the error. Thread Id information is only present when using the multi-threaded version of XSB, and even in that version is not present in all error terms.

`xsb_error_get_backtrace(+Error,-Backtrace)` module: `error_handler`
: Obtains the backtrace — the stack of the forward continuations in the execution stack at the time the error was thrown. Backtraces are present by default in all XSB system error terms, and are described in Section 12.5.

12.4 Convenience Predicates

The following convenience predicates are provided to make a commonly used check and to throw an ISO error if the check is not satisfied; some are written directly in C

for speed. All these predicates must be imported from the module `error_handler`, which also contains provides a few other specialized checks.

`check_acyclic(?Term,+Predicate,+Arg)` module: `error_handler`
: Checks that `Term` is acyclic. If so, the predicate succeeds; if not it throws a miscellaneous error.

`check_atom(?Term,+Predicate,+Arg)` module: `error_handler`
: Checks that `Term` is an atom. If so, the predicate succeeds; if not it throws a type error.

`check_callable(?Term,+Predicate,+Arg)` module: `error_handler`
: Checks that `Term` is callable. If so, the predicate succeeds; if not it throws a type error.

`check_ground(?Term,+Predicate,+Arg)` module: `error_handler`
: Checks that `Term` is ground. If so, the predicate succeeds; if not it throws an instantiation error.

`check_integer(?Term,+Predicate,+Arg)` module: `error_handler`
: Checks that `Term` is an integer. If so, the predicate succeeds; if not it throws a type error.

`check_nonvar(?Term,+Predicate,+Arg)` module: `error_handler`
: Checks that `Term` is not a variable. If not, the predicate succeeds; if `Term` is a variable, it throws an instantiation error.

`check_nonvar_list(?Term,+Predicate,+Arg)` module: `error_handler`
: Checks that `Term` is a list, each of whose elements is ground. If so, the predicate succeeds; if not it throws an instantiation error.

`check_one_thread(+Operation,+Object_Type,+Predicate)` module: `error_handler`
: In the multi-threaded engine, `check_one_thread/3` checks that there is only one active thread: if not, a miscellaneous error is thrown indicating that `Operation` is not permitted on `ObjectType` as called by `Predicate`, when more than one thread is active. This check provides a convenient way to allow inclusion of certain operations that are difficult to make thread-safe by other means.

  In the single-threaded engine this predicate always succeeds.

`check_stream(?Stream,+Predicate,+Arg)` module: `error_handler`

Checks that `Stream` is a stream. If so, the predicate succeeds; if not it throws an instantiation error [9].

`check_var(?Term,+Predicate,+Arg)` module: `error_handler`

Checks that `Term` is a variable. If so, the predicate succeeds; if not it throws an instantiation error.

12.5 Backtraces

Displaying a backtrace of the calling context of an error in addition to an error message can greatly expedite debugging. For XSB's default error handler, backtraces are printed out by default, a behavior that can be overridden for a given thread by the command: `set_prolog_flag(backtrace_on_error,off)`. For users who write their own error handlers, the following predicates can be used to manipulate backtraces.

It is important to note that Prolog backtraces differ in a significant manner from backtraces obtained from other languages, such as C backtraces produced by GDB. This is because a Prolog backtrace obtains forward continuations from the local environment stack, and in the WAM, local stack frames are only created when a given clause requires permanent variables – otherwise these stack frames are optimized away. The precise conditions for optimizing away a local stack frame require an understanding of the WAM (and of a specific compiler). However in general, longer clauses with many variables require a local stack frame and their forward continuations will be displayed, while shorter clauses with fewer variables do not and their forward continuations will not be displayed.

`xsb_backtrace(-Backtrace)` module: `machine`

Upon success `Backtrace` is bound to a structure indicating the forward continuations for a point of execution. This structure should be treated as opaque, and manipulated by one of the predicates below.

`get_backtrace_list(+Backtrace,-PredicateList)` module: `error_handler`

Given a backtrace structure, this predicate produces a list of predicate identifiers or the form `Module:Predicate/Arity`. This list can be manipulated as desired by error handling routines.

[9] The representation of streams in XSB is subject to change.

`print_backtrace(+Backtrace)` module: `error_handler`

This predicate, which is used by XSB's default error handler, prints a backtrace structure to XSB's standard error stream.

When XSB generates a memory exception *at the OS level* (e.g., a segmentation violation or bus error) it prints out a backtrace and exits. This should be caused only by a bug in XSB or included C code. The first predicate in the backtrace that is printed in these circumstances may be incorrect or redundant. This is because the memory structures used to generate the backtrace are not always completely consistent, and so an interrupt at an unexpected point may result in the use of somewhat inconsistent information.

Chapter 13

Foreign Language Interface

When XSB is used to build real-world systems, a foreign-language interface may be necessary to:

- combine XSB with existing programs and libraries, thereby forming composite systems;
- interface XSB with the operating system, graphical user interfaces or other system level programs;
- speed up certain critical operations.

XSB has both a high-level and the low-level interface to C. The low-level interface is much more flexible, but it requires greater attention to details of how the data is passed between XSB and C. To connect XSB to a C program using the high-level interface requires very little work, but the program must be used "as is" and it must take the input and produce the output supported by this high-level interface. Before describing the interfaces themselves, we first describe aspects common to both the lower- and higher-level foreign language interfaces.

The foreign language interface can also support `C++` programs. Since XSB is written in `C`, the interface functions in the foreign `C++` module must have the declaration `extern "C"`, and a separate compiler option (e.g. specifying `g++` rather than `gcc`) may need to be given to ensure proper linkage, inclusion of `C++` libraries, etc. In addition, on certain platforms compilation may need to be done externally to XSB – see the `xasp` 1package for a example of using the foreign language interface with `C++` files. For the rest of this chapter, we restrict our attention to foreign predicates written in `C`.

13.1 Foreign Language Modules

Foreign predicates must always appear in modules, and these modules can contain only foreign predicates. A foreign module differs from a Prolog module in that the foreign module's source file must appear in a `*.c` file rather than a `*.P` file (or `.pl` file). This `*.c` file cannot contain a `main()` function. Furthermore, a `*.P` file with the same name *must not* be present or else the `*.c` file is ignored and the module is compiled as a regular Prolog module. The interface part of a foreign module, which has the same syntax as that of a normal module, is written in Prolog and must appear in a `*.H` file. If the lower-level interface is used, this `*.H` file contains explicit `export/1` declarations for the the foreign predicates that are to be used by other modules; if the higher-level interface is used, the declarations have the form `foreign_pred/1`.

The Prolog predicates attached to foreign functions are deterministic, in the sense that they succeed at most once for a given call and are not re-entered on backtracking. Note that this requirement imposes no serious limitation, since it is always possible to divide a foreign predicate into the part to be done on the first call and the part to be redone on backtracking. Backtracking can then take place at the Prolog level where it is more naturally expressed.

A foreign module can be compiled or consulted just like a normal Prolog module. Currently, predicates `consult/[1,2]` recompile both the `*.c` and the `*.H` files of a foreign module when at least one of them has been changed from the time the corresponding object files have been created (see the section *Compiling and Consulting* in Volume 1) [1]. The C compiler used to compile the `*.c` files can be set as a defaults to that used for the configuration of XSB (refer to the section *Getting Started with XSB* in Volume 1). This default behavior includes the C compilation options used to compile XSB when it was configured, along with a default set of include files so that header files in XSB directories can be obtained. Alternately, the user can add options to be passed to the C compiler. To give an example, the following command will compile file `file.c` using the default C Compiler with optimization and by including `/usr/local/X11/R6/include` to the directories that will be searched for header files.

```
:- consult(file, [cc_opts('-O2 -I/usr/local/X11/R6/include')]).
```

Note in particular, that if XSB were compiled with the `-g` debugging option, then the C file will be also [2]. Any Prolog compiler options are ignored when compiling a foreign module.

[1] In addition, if a C module compiled by the single-threaded XSB engine is loaded by the multi-threaded engine, it will be recompiled, and vice-versa.

[2] In a 64-bit platform, users may override the default compilation of XSB by the configuration

Prolog-specific directives such as `index`, `hilog`, `table`, `auto_table` or even `import` make no sense in the case of a foreign module and thus are ignored by the compiler. However, another directive, namely `ldoption`, is recognized in a foreign module and is used to instruct the dynamic loading and linking of the module. The syntax of the `ldoption` directive is simply:

```
:- ldoption(Option).
```

where `Option` should either be an atom or a list of atoms. Multiple `ldoption` directives may appear in the same `.H` file of a foreign module [3]. In Unix-derived systems, the foreign language interface of XSB uses `ld` command that combines object programs to create an executable file or another object program suitable for further `ld` processing. Version 3.8 of XSB assumes that the `ld` command resides in the file `/usr/bin/ld`.

13.2 Lower-Level Foreign Language Interface

Creating a foreign predicate using the lower-level foreign language interface is almost entirely a matter of writing C code. Consider the foreign module `$XSBDIR/examples/XSB_calling_c/simpl`
The .H file has the form:

```
:- export minus_one/2, my_sqrt/2, change_char/4.

:- ldoption('-lm').     % link together with the math library
```

When the lower level foreign language interface is used, C functions that implement foreign predicates must return values of type `int`. The return value is not used by a Prolog argument; rather if a non-zero is returned, the foreign predicate succeeds; a zero return value means failure.

---

options `-with-bits32` or `-with-bits64`. If either of these options is used, the default compilation options will pass along the appropriate memory options. If XSB is compiled with a memory option that is not the default of the platform, and if an externally compiled C file is to be loaded into XSB, it must be ensured that the C file has been compiled with the appropriate memory options: `-m32` or `-m64` if `gcc` is used.

[3] Mac OSX users using 10.3 or above should have the environment variable `MACOSX_DEPLOYMENT_TARGET` set to `10.3` so that the compiler generates code that can be dynamically linked by XSB. This should be done automatically by XSB on initialization, but it is useful to check if encountering problems.

At the C level, the function that implements the Prolog predicate must have the same name as the Prolog predicate (that is declared in the `*.H` file), and must have a special *context parameter* macro. The context parameter macro allows C functions to be used with both the single-threaded and multi-threaded engines, and are described in detail in Section 13.2.1. The Prolog level arguments are converted to C data structures through several predefined functions rather than through direct parameter passing [4]. The C file `simple_foreign.c` corresponding to the above `.H` file is as follows.

```
/*----------------------------------------------------------------------*/

#include <math.h>
#include <stdio.h>
#include <string.h>
#include <alloca.h>

/*----- Make sure your C compiler finds the following header file.    -----
  ----- One way to do this is to include the directory XSB/emu on the -----
  ----- compiler's command line with the -I (/I in Windows) option    -----*/

#include "cinterf.h"

/*---------------------------------------*/

int minus_one(CTXTdecl)
{
   int  i = ptoc_int(CTXTc 1);

   ctop_int(CTXTc 2, i-1);
   return TRUE;
}

/*---------------------------------------*/

int my_sqrt(CTXTdecl)
{
   int i = ptoc_int(CTXTc 1);
```

[4]The inclusion of context parameters changes the lower-level interface for Version 3.0. C files written for previous versions of XSB continue to work properly for the single-threaded engine in, but will not work properly for the multi-threaded engine.

```
   ctop_float(CTXTc 2, (float) pow((double)i, 0.5));
   return TRUE;
}

/*----------------------------------------*/

int change_char(CTXTdecl)
{
   char *str_in;
   int  pos;
   int c;
   char *str_out;

   str_in = (char *) ptoc_string(CTXTc 1);
   str_out = (char *) alloca(strlen(str_in)+1);
   strcpy(str_out, str_in);
   pos = ptoc_int(CTXTc (2);
   c = ptoc_int(CTXTc (3);
   if (c < 0 || c > 255) /* not a character */
     return FALSE; /* this predicate will fail on the Prolog side */

   str_out[pos-1] = c;

   extern_ctop_string(CTXTc 4, str_out);
   return TRUE;
}

/*----------------------------------------------------------------------*/
```

Before describing the C program used, here is a sample session illustrating the behavior of the predicates in `simple_foreign`.

```
XSB Version 2.0 (Gouden Carolus) of June 26, 1999
[i686-pc-linux-gnu; mode: optimal; engine: slg-wam; scheduling: batched]
| ?- [simple_foreign].
[Compiling C file ./simple_foreign.c using gcc]
[Compiling Foreign Module ./simple_foreign]
[simple_foreign compiled, cpu time used: 0.0099993 seconds]
[simple_foreign loaded]

yes
| ?- change_char('Kostis', 2, w, TempStr),
```

```
     change_char(TempStr, 5, h, GrkName).

TempStr = Kwstis
GrkName = Kwsths;

no
| ?- minus_one(43, X).

X = 42;

no
| ?- minus_one(43, 42).                    % No output unification is allowed
Wrong arg in ctop_int 2a2 (Reg = 2)

yes
| ?- my_sqrt(4,X).

X = 2

yes
| ?- my_sqrt(23,X).

X = 4.7958;

no
```

Consider the function `minus_one()` above. As discussed, it takes a context parameter (explained below), and returns an integer, and as can be seen the return values can be specified by the macros `TRUE` and `FALSE`. From the Prolog perspective the first argument to `minus_one/2` is an (integer) input argument, while the second is an (integer) output argument. Input arguments for basic C types are translated from their Prolog representation to a C representation by functions of the form `ptoc\_<type>()` – here `ctop_int()`. The single parameter of such a function is the number of the Prolog argument that is to be transformed and the function returns the C representation. Output arguments are converted from C to Prolog by corresponding functions of the form `ctop_<type>()` – here `ctop_int()`. For converting C back to Prolog, the first parameter of `ctop_int()` is the number of the Prolog argument to be transformed and the second is the C value to be transformed. In the session output above, if an improper argument is given to `minus_one/2` it will emit a warning, and succeed. Also note that the call `my_sqrt(23,X)` succeeds once, but fails on backtracking since it is deterministic, as are all other foreign language functions.

The above example illustrates the exchange of *basic* types through the lower-level interface – e.g. atoms, integers, and floating-point numbers. The lower-level interface also allows a user to pass lists and terms between XSB and C as will be discussed in Section 13.2.3.

13.2.1 Context Parameters

When using the lower-level interface, *context parameters* must be added to many C functions in order for the functions to be used with XSB's multi-threaded engine. In the multi-threaded engine, variables for Prolog's virtual machine, as well as for thread-private data structures are stored in a *context structure*. This context structure must be passed to any functions that need to access elements of a thread's virtual machine – including many of the functions that are used to exchange data between Prolog and C. We note in passing that when using the multi-threaded engine, a user must ensure that foreign-language functions are thread-safe, by using standard multi-threaded programming techniques, including XSB's mutex predicates (see the Section *Predicates for Thread Synchronization* in Volume 1 of this manual). On the other hand, in the single-threaded engine virtual machine elements are kept in static variables, so that context parameters are not required.

The lower-level C interface makes use of a set of macros to address the requirements of the different engines. The data exchange functions discussed in this chapter, `ptoc_xxx`, `ctop_xxx`, `c2p_xxx`, `p2c_xxx`, and `p2p_xxx` usually, but not always, require information about a threads virtual machine state. If a C function directly or indirectly calls a data interchange function that requires a context parameter, the function must have a context parameter in its declaration, calls, and prototypes in order to be used by the multi-threaded engine. These context parameters have the following forms:

- In function *declarations*, use the macro `CTXTdecl` in the code for a function that would otherwise be `void`, and `CTXTdeclc` as the first argument in the code for a function with parameters (`CTXTdeclc` and `CTXTdecl` are similar, except that macro expansion of `CTXTdeclc` for the multi-threaded engine includes a comma). The example for `minus_one(CTXTdecl)` shows use of this macro.
- In function *calls* use the macro `CTXT` in the code for a function that would otherwise be `void`, and `CTXTc` as the first argument in the code for a function with parameters. As an example, a call to `minus_one` would have the form `minus_one(CTXT)`.

- In function *prototypes* use the macro `CTXTdecltype` in the code for a function that would otherwise be `void`, and `CTXTdecltypec` as the first argument in the code for a function with parameters. As an example, a prototype for `minus_one` would have the form `minus_one(CTXTdecltype)`.

Fortunately, when compiling with the multi-threaded engine, it is easy to determine at compile time whether context parameters are correct. If compilation of a function `foo` gives an error along the lines of:

```
foofile.c: In function ‘foo’:
foofile.c:109: error: ‘th’ undeclared (first use in this function)
```

Then the declaration of `foo` omitted a context parameter. If compilation gives an error along the lines of

```
foofile.c: In function ‘foo_caller’:
:
foofile.c:149: error: too few arguments to function ‘foo’
```

Then the call to `foo` may have omitted a context parameter.

Note that context parameters are *only* necessary if the lower-level interface is used. The higher-level interface automatically generates any context parameters it needs.

13.2.2 Exchanging Basic Data Types

The basic interface assumes that correct modes (*i.e.*, input or output parameters) and types are being passed between the C and Prolog levels. As a result, output unification should be explicitly performed in the Prolog level. The prototypes for the conversion functions between Prolog and C should be declared before the corresponding functions are used. This is done by including the `"cinterf.h"` header file. Under Unix, the XSB foreign C interface automatically finds this file in the `XSB/emu` directory. Under Windows, the user must compile and create the DLL out of the C file manually, so the compiler option '`/I...\XSB\emu`' is necessary.[5]

The following C functions are used to convert basic types between Prolog and C.

`int ptoc_int(CTXTdeclc int N)`

`CTXTdeclc` is a context parameter; `N` is assumed to hold a Prolog integer corresponding to the `N`th argument of a Prolog predicate. This function returns the value of that argument in as a C `int`.

[5]The foreign interface does not Cygwin.

`double ptoc_float(CTXTdeclc int N)`
: `CTXTdeclc` is a context parameter; `N` is assumed to hold a Prolog integer corresponding to the `N`th argument of a Prolog predicate. This function returns the value of that argument as a C `double`. By default, XSB provides double precision, but if XSB was configured with `-enable-fast-floats` less than single precision can be provided [6].

`char *ptoc_string(CTXTdeclc int N)`
: `CTXTdeclc` is a context parameter; `N` is assumed to hold a Prolog integer corresponding to the `N`th argument of a Prolog predicate. This function returns the value the C string (of type `char *`) that corresponds to this interned Prolog atom. *WARNING: the string should be copied before being manipulated in any way: otherwise unexpected results may arise whenever the interned Prolog atom is unified.*

`void ctop_int(CTXTdeclc int N, int V)`
: `CTXTdeclc` is a context parameter; argument `N` is assumed to hold a Prolog free variable, and this function binds that variable to an integer of value `V`.

`void ctop_float(CTXTdeclc int N, float V)`
: `CTXTdeclc` is a context parameter; argument `N` is assumed to hold a Prolog free variable, and this function binds that variable to a floating point number of value `V`.

`void extern_ctop_string(CTXTdeclc int N, char * V)`
: `CTXTdeclc` is a context parameter; argument `N` is assumed to hold a Prolog free variable. If needed, this function interns the string to which `V` points as a Prolog atom and then binds the variable in argument `N` to that atom.

13.2.3 Exchanging Complex Data Types

If the lower-level interface is used, exchanging basic data types is sufficient for most applications. Exchanging complex data types is also possible, although doing so is slightly more involved than exchanging basic types. To exchange complex data types, the lower-level interface uses only one C data type: `prolog_term`, which can point to any XSB term. On the C side, the type of the term can be checked and then

[6] The fast float configuration option does represents floating point values as directly tagged single precision values rather than as indirectly tagged double precision values. Speed increases in arithmetic can be gained from this optimization, in exchange for significant precision loss on floating point numbers.

processed accordingly. For instance, if the term turns out to be a structure, then it can be decomposed and the functor can be extracted along with the arguments. If the term happens to be a list, then it can be processed in a loop and each list member can be further decomposed into its atomic components. The advanced interface also provides functions to check the types of these atomic components and for converting them into C types.

We begin by presenting the functions used to exchange complex data types, before presenting a detailed example below. As when exchanging basic C types, the file `emu/cinterf.h` must be included in the C program in order to make the prototypes of the relevant functions known to the C compiler.

The first set of functions is typically used to check the type of Prolog terms passed into the C program.

`xsbBool is_attv((prolog_term) T)` *C function*
`is_attv(T)` returns `TRUE` if `T` represents an XSB attributed variable, and `FALSE` otherwise.

`xsbBool is_float((prolog_term) T)` *C function*
`is_float(T)` returns `TRUE` if `T` represents an XSB float value, and `FALSE` otherwise.

`xsbBool is_functor((prolog_term) T)` *C function*
`is_functor(T)` returns `TRUE` if `T` represents an XSB structure value (not a list), and `FALSE` otherwise.

`xsbBool is_int((prolog_term) T)` *C function*
`is_int(T)` returns `TRUE` if `T` represents an XSB integer value, and `FALSE` otherwise.

`xsbBool is_list((prolog_term) T)` *C function*
`is_list(T)` returns `TRUE` if `T` represents an XSB list value (not nil), and `FALSE` otherwise.

`xsbBool is_nil((prolog_term) T)` *C function*
`is_nil(T)` returns `TRUE` if `T` represents an XSB `[]` (nil) value, and `FALSE` otherwise.

`xsbBool is_string((prolog_term) T)` *C function*
`is_string(T)` returns `TRUE` if `T` represents an XSB atom value, and `FALSE` otherwise.

`xsbBool is_var((prolog_term) T)` *C function*

`is_var(T)` returns `TRUE` if `T` represents an XSB variable, and `FALSE` otherwise.

After checking the types of the arguments passed in from the Prolog side, the next task usually is to convert Prolog data into the types understood by C. This is done with the following functions. The first three convert between the basic types. The last two extract the functor name and the arity. Extraction of the components of a list and the arguments of a structured term is explained later.

`int p2c_int((prolog_term) V)` *C function*

The `prolog_term` parameter must represent a Prolog integer, and `p2c_int` returns the C representation of that integer.

`double p2c_float((prolog_term) V)` *C function*

The `prolog_term` parameter must represent a Prolog floating point number, and `p2c_float` returns the C representation of that floating point number.

`char *p2c_string((prolog_term) V)` *C function*

The `prolog_term` parameter must represent a (Prolog) atom, and `p2c_string` returns that atom as a C string. The pointer returned points to the actual atom name in XSB 's atom table, and thus it must NOT be modified by the calling program.

`char *p2c_functor((prolog_term) V)` *C function*

The `prolog_term` parameter must represent a structured term (not a list). `p2c_functor` returns the name of the main functor symbol of that term as a string. The pointer returned points to the actual functor name in XSB 's space, and thus it must NOT be modified by the calling program.

`int p2c_arity((prolog_term) V)` *C function*

The `prolog_term` parameter must represent a structured term (not a list). `p2c_arity` returns the arity of the main functor symbol of that term as a C `int`.

The next batch of functions support conversion of data in the opposite direction: from basic C types to the type `prolog_term`. These `c2p_*` functions all return a boolean value `TRUE` if successful and `FALSE` if unsuccessful. The XSB term argument must always contain an XSB variable, which will be bound to the indicated value as a side effect of the function call.

`xsbBool c2p_int(CTXTdeclc (int) N, (prolog_term) V)` *C function*
`CTXTdeclc` is a context parameter; `c2p_int` binds the `prolog_term V` (which must be a variable) to the integer value N, creating a Prolog integer.

`xsbBool c2p_float(CTXTdeclc (double) F, (prolog_term) V)` *C function*
`CTXTdeclc` is a context parameter; `c2p_float` binds the `prolog_term V` (which must be a variable) to the (double) float value F, creating a double Prolog float.

`xsbBool c2p_string(CTXTdeclc (char *) S, (prolog_term) V)` *C function*
`CTXTdeclc` is a context parameter; `c2p_string` binds the `prolog_term V` (which must be a variable) to the Prolog atom corresponding to the `char *S`. During this process the Prolog atom is interned into XSB's atom table.

The following functions create Prolog data structures within a C program. This is usually done in order to pass these structures back to the Prolog side.

`xsbBool c2p_functor(CTXTdeclc (char *) S, (int) N, (prolog_term) V)` *C function*
`CTXTdeclc` is a context parameter; `c2p_functor` binds the `prolog_term V` (which must be a variable) to an open term whose main functor symbol is given by `S` (of type char *) and whose arity is `N`. An open term is one with all arguments as new distinct variables.

`xsbBool c2p_list(CTXTdeclc (prolog_term) V)` *C function*
`CTXTdeclc` is a context parameter; `c2p_list` binds the `prolog_term V` (which must be a variable) to an open list term, i.e., a list term with both car and cdr as new distinct variables. Note: to create an empty list use the function `c2p_nil` described below.

`xsbBool c2p_nil(CTXTdeclc (prolog_term) V)` *C function*
`CTXTdeclc` is a context parameter; `c2p_nil` binds the `prolog_term V` (which must be a variable) to the atom `[]` (nil).

`prolog_term p2p_new()` *C function*
Create a new Prolog variable. This is sometimes needed when you want to create a Prolog term on the C side and pass it to the Prolog side.

To use the above functions, one must be able to get access to the components of the structured Prolog terms. This is done with the help of the following functions:

`prolog_term p2p_arg((prolog_term) T, (int) A)` *C function*
Parameter `T` must be a `prolog_term` that is a structured term (but not a list).

A is a positive integer (no larger than the arity of the term) that specifies an argument position of the term `T`. `p2p_arg` returns the A^{th} subfield of the term `T`.

`prolog_term p2p_car((prolog_term) T)` *C function*

Parameter `T` must be a `prolog_term` that is a list (not nil). `p2p_car` returns the car (i.e., head of the list) of the term T.

`prolog_term p2p_cdr((prolog_term) T)` *C function*

Parameter `T` must be a `prolog_term` that is a list (not nil). `p2p_cdr` returns the cdr (i.e., tail of the list) of the term T.

It is important to realize that these functions return the actual Prolog term that is, say, the head of a list or the actual argument of a structured term. Thus, assigning a value to such a Prolog term also modifies the head of the corresponding list or the relevant argument of the structured term. It is precisely this feature that allows passing structured terms and lists from the C side to the Prolog side. For instance,

```
prolog_term plist,        /* a Prolog list           */
            structure;    /* something like f(a,b,c) */
prolog_term tail, arg;
..........
tail = p2p_cdr(plist);         /* get the list tail  */
arg  = p2p_arg(structure, 2);  /* get the second arg */

/* Assume that the list tail was supposed to be a prolog variable */
if (is_var(tail))
   c2p_nil(CTXTc tail);  /* terminate the list */
else {
   fprintf(stderr, "Something wrong with the list tail!");
   exit(1);
}
/* Assume that the argument was supposed to be a prolog variable */
c2p_string(CTXTc "abcdef", arg);
```

In the above program fragment, we assume that both the tail of the list and the second argument of the term were supposed to be bound to Prolog variables. In case of the tail, we check if this is, indeed, the case. In case of the argument, no checks are done; XSB will issue an error (which might be hard to track down) if the second argument is not currently bound to a variable.

The last batch of functions is useful for passing data in and out of the Prolog side of XSB. The first function is the only way to get a `prolog_term` out of the Prolog side; the second function is sometimes needed in order to pass complex structures from C into Prolog.

`prolog_term reg_term(CTXTdeclc (int) R)` *C function*

`CTXTdeclc` is a context parameter. Parameter R is an argument number of the Prolog predicate implemented by this C function (range 1 to 255). The function `reg_term` returns the `prolog_term` in that predicate argument.

`xsbBool p2p_unify(CTXTdeclc prolog_term T1, prolog_term T2)` *C function*

Unify the two Prolog terms. This is useful when an argument of the Prolog predicate (implemented in C) is a structured term or a list, which acts both as input and output parameter. `CTXTdeclc` is a context parameter.

For instance, consider the Prolog call `test(X, f(Z))`, which is implemented by a C function with the following fragment:

```
prolog_term newterm, newvar, z_var, arg2;
.....
/* process argument 1 */
c2p_functor(CTXTc "func",1,reg_term(CTXTc 1));
c2p_string(CTXTc "str",p2p_arg(reg_term(CTXTc 1),1));
/* process argument 2 */
arg2 = reg_term(CTXTc 2);
z_var = p2p_arg(arg2, 1);  /* get the var Z */
/* bind newterm to abc(V), where V is a new var */
c2p_functor(CTXTc "abc", 1, newterm);
newvar = p2p_arg(newterm, 1);
newvar = p2p_new();
....
/* return TRUE (success), if unify; FALSE (failure) otherwise */
return p2p_unify(CTXTc z_var, newterm);
```

On exit, the variable `X` will be bound to the term `func(str)`. Processing argument 2 is more interesting. Here, argument 2 is used both for input and output. If `test` is called as above, then on exit Z will be bound to `abc(_h123)`, where `_h123` is some new Prolog variable. But if the call is `test(X,f(1))` or `test(X,f(Z,V))` then this call will *fail* (fail as in Prolog, *i.e.*, it is not an error), because the term passed back, `abc(_h123)`, does not unify with `f(1)` or `f(Z,V)`. This effect is achieved by the use of `p2p_unify` above.

We conclude this section with two real examples of functions that pass complex data in and out of the Prolog side of XSB. These functions are part of the POSIX regular expression matching package of XSB. The first function uses argument 2 to accept a list of complex Prolog terms from the Prolog side and does the processing on the C side. The second function does the opposite: it constructs a list of complex Prolog terms on the C side and passes it over to the Prolog side in argument 5.

(We should note that this second function could cause a heap overflow in XSB were it to build a large list of values. Instead of building a large list of values on the XSB heap, one would better design the functions to return smaller values, in which case XSB will be able to automatically expand the heap as necessary.)

```
/* XSB string substitution entry point: replace substrings specified in Arg2
   with strings in Arg3.
   In:
       Arg1: string
       Arg2: substring specification, a list [s(B1,E1),s(B2,E2),...]
       Arg3: list of replacement string
   Out:
       Arg4: new (output) string
   Always succeeds, unless error.
*/
int do_regsubstitute__(CTXTdecl)
{
  /* Prolog args are first assigned to these, so we could examine the types
     of these objects to determine if we got strings or atoms. */
  prolog_term input_term, output_term;
  prolog_term subst_reg_term, subst_spec_list_term, subst_spec_list_term1;
  prolog_term subst_str_term=(prolog_term)0,
    subst_str_list_term, subst_str_list_term1;
  char *input_string=NULL;    /* string where matches are to be found */
  char *subst_string=NULL;
  prolog_term beg_term, end_term;
  int beg_offset=0, end_offset=0, input_len;
  int last_pos = 0; /* last scanned pos in input string */
  /* the output buffer is made large enough to include the input string and the
     substitution string. */
  char subst_buf[MAXBUFSIZE];
  char *output_ptr;
  int conversion_required=FALSE; /* from C string to Prolog char list */

  input_term = reg_term(CTXTc 1);  /* Arg1: string to find matches in */
```

```
if (is_string(input_term)) /* check it */
  input_string = string_val(input_term);
else if (is_list(input_term)) {
  input_string =
    p_charlist_to_c_string(input_term, input_buffer, sizeof(input_buffer),
                           "RE_SUBSTITUTE", "input string");
  conversion_required = TRUE;
} else
  xsb_abort("RE_SUBSTITUTE: Arg 1 (the input string) must be an atom or a character list");

input_len = strlen(input_string);

/* arg 2: substring specification */
subst_spec_list_term = reg_term(CTXTc 2);
if (!is_list(subst_spec_list_term) && !is_nil(subst_spec_list_term))
  xsb_abort("RE_SUBSTITUTE: Arg 2 must be a list [s(B1,E1),s(B2,E2),...]");

/* handle substitution string */
subst_str_list_term = reg_term(CTXTc 3);
if (! is_list(subst_str_list_term))
  xsb_abort("RE_SUBSTITUTE: Arg 3 must be a list of strings");

output_term = reg_term(CTXTc 4);
if (! is_var(output_term))
  xsb_abort("RE_SUBSTITUTE: Arg 4 (the output) must be an unbound variable");

subst_spec_list_term1 = subst_spec_list_term;
subst_str_list_term1 = subst_str_list_term;

if (is_nil(subst_spec_list_term1)) {
  strncpy(output_buffer, input_string, sizeof(output_buffer));
  goto EXIT;
}
if (is_nil(subst_str_list_term1))
  xsb_abort("RE_SUBSTITUTE: Arg 3 must not be an empty list");

/* initialize output buf */
output_ptr = output_buffer;

do {
  subst_reg_term = p2p_car(subst_spec_list_term1);
  subst_spec_list_term1 = p2p_cdr(subst_spec_list_term1);
```

```
if (!is_nil(subst_str_list_term1)) {
  subst_str_term = p2p_car(subst_str_list_term1);
  subst_str_list_term1 = p2p_cdr(subst_str_list_term1);

  if (is_string(subst_str_term)) {
    subst_string = string_val(subst_str_term);
  } else if (is_list(subst_str_term)) {
    subst_string =
      p_charlist_to_c_string(subst_str_term, subst_buf, sizeof(subst_buf),
                             "RE_SUBSTITUTE", "substitution string");
  } else
    xsb_abort("RE_SUBSTITUTE: Arg 3 must be a list of strings");
}

beg_term = p2p_arg(subst_reg_term,1);
end_term = p2p_arg(subst_reg_term,2);

if (!is_int(beg_term) || !is_int(end_term))
  xsb_abort("RE_SUBSTITUTE: Non-integer in Arg 2");
else{
  beg_offset = int_val(beg_term);
  end_offset = int_val(end_term);
}
/* -1 means end of string */
if (end_offset < 0)
  end_offset = input_len;
if ((end_offset < beg_offset) || (beg_offset < last_pos))
  xsb_abort("RE_SUBSTITUTE: Substitution regions in Arg 2 not sorted");

/* do the actual replacement */
strncpy(output_ptr, input_string + last_pos, beg_offset - last_pos);
output_ptr = output_ptr + beg_offset - last_pos;
if (sizeof(output_buffer)
    > (output_ptr - output_buffer + strlen(subst_string)))
  strcpy(output_ptr, subst_string);
else
  xsb_abort("RE_SUBSTITUTE: Substitution result size %d > maximum %d",
            beg_offset + strlen(subst_string),
            sizeof(output_buffer));

last_pos = end_offset;
```

```
    output_ptr = output_ptr + strlen(subst_string);

  } while (!is_nil(subst_spec_list_term1));

  if (sizeof(output_buffer) > (output_ptr-output_buffer+input_len-end_offset))
    strcat(output_ptr, input_string+end_offset);

 EXIT:
  /* get result out */
  if (conversion_required)
    c_string_to_p_charlist(output_buffer,output_term,"RE_SUBSTITUTE","Arg 4");
  else
    /* DO NOT intern. When atom table garbage collection is in place, then
       replace the instruction with this:
                  c2p_string(CTXTc output_buffer, output_term);
       The reason for not interning is that in Web page
       manipulation it is often necessary to process the same string many
       times. This can cause atom table overflow. Not interning allows us to
       circumvent the problem.  */
    extern_ctop_string(CTXTc 4, output_buffer);

  return(TRUE);
}


/* XSB regular expression matcher entry point
   In:
       Arg1: regexp
       Arg2: string
       Arg3: offset
       Arg4: ignorecase
   Out:
       Arg5: list of the form [match(bo0,eo0), match(bo1,eo1),...]
             where bo*,eo* specify the beginning and ending offsets of the
             matched substrings.
             All matched substrings are returned. Parenthesized expressions are
             ignored.
*/
int do_bulkmatch__(CTXTdecl)
{
  prolog_term listHead, listTail;
  /* Prolog args are first assigned to these, so we could examine the types
```

```
   of these objects to determine if we got strings or atoms. */
prolog_term regexp_term, input_term, offset_term;
prolog_term output_term = p2p_new();
char *regexp_ptr=NULL;      /* regular expression ptr               */
char *input_string=NULL;    /* string where matches are to be found */
int ignorecase=FALSE;
int return_code, paren_number, offset;
regmatch_t *match_array;
int last_pos=0, input_len;
char regexp_buffer[MAXBUFSIZE];

if (first_call)
  initialize_regexp_tbl();

regexp_term = reg_term(CTXTc 1);  /* Arg1: regexp */
if (is_string(regexp_term)) /* check it */
  regexp_ptr = string_val(regexp_term);
else if (is_list(regexp_term))
  regexp_ptr =
    p_charlist_to_c_string(regexp_term, regexp_buffer, sizeof(regexp_buffer),
                           "RE_MATCH", "regular expression");
else
  xsb_abort("RE_MATCH: Arg 1 (the regular expression) must be an atom or a character list");

input_term = reg_term(CTXTc 2);  /* Arg2: string to find matches in */
if (is_string(input_term)) /* check it */
  input_string = string_val(input_term);
else if (is_list(input_term)) {
  input_string =
    p_charlist_to_c_string(input_term, input_buffer, sizeof(input_buffer),
                           "RE_MATCH", "input string");
} else
  xsb_abort("RE_MATCH: Arg 2 (the input string) must be an atom or a character list");

input_len = strlen(input_string);

offset_term = reg_term(CTXTc 3); /* arg3: offset within the string */
if (! is_int(offset_term))
  xsb_abort("RE_MATCH: Arg 3 (the offset) must be an integer");
offset = int_val(offset_term);
if (offset < 0 || offset > input_len)
  xsb_abort("RE_MATCH: Arg 3 (=%d) must be between 0 and %d", input_len);
```

```
  /* If arg 4 is bound to anything, then consider this as ignore case flag */
  if (! is_var(reg_term(CTXTc 4)))
    ignorecase = TRUE;

  last_pos = offset;
  /* returned result */
  listTail = output_term;
  while (last_pos < input_len) {
    c2p_list(CTXTc listTail); /* make it into a list */
    listHead = p2p_car(listTail); /* get head of the list */

    return_code = xsb_re_match(regexp_ptr, input_string+last_pos, ignorecase,
                               &match_array, &paren_number);
    /* exit on no match */
    if (! return_code) break;

    /* bind i-th match to listHead as match(beg,end) */
    c2p_functor(CTXTc "match", 2, listHead);
    c2p_int(CTXTc match_array[0].rm_so+last_pos, p2p_arg(listHead,1));
    c2p_int(CTXTc match_array[0].rm_eo+last_pos, p2p_arg(listHead,2));

    listTail = p2p_cdr(listTail);
    last_pos = match_array[0].rm_eo+last_pos;
  }
  c2p_nil(CTXTc listTail); /* bind tail to nil */
  return p2p_unify(CTXTc output_term, reg_term(CTXTc 5));
}
```

13.3 Foreign Modules That Call XSB Predicates

A C function that has been called from XSB through the lower-level foreign language interface may want to call back into XSB to have XSB evaluate a predicate. This can be done by using the interface described in Chapter 3 (Volume 2) on calling XSB from another language. The interface described there allows a caller to initialize XSB and pass queries to it. However, since XSB has already called a foreign module, XSB does not need to be initialized. However it does need to manage the registers that are in use to support interaction with the foreign module currently executing. So there are some minor differences with the interface described in Chapter 3.

First, XSB should not be initialized. I.e., a foreign module should **not** call

`xsb_init` or `xsb_init_string`. Second, the foreign module must protect the XSB registers it is currently using when it calls XSB. To do this, after it has retrieved its arguments into local variables and before it calls any XSB predicate, it must call `xsb_query_save(NumRegs)`, which saves the current XSB registers and initializes them to be able to accept a new query. `NumRegs` is the number of registers used to interact with the currently executing foreign routine (i.e., the arity of the predicate that called this foreign code.) When the foreign routine has completed its work, it will set the appropriate registers with the appropriate return values and return to the caller. Before it does this, it must call `xsb_query_restore()` to restore the saved registers and prepare XSB for the return. Note that it must be called before any of the output registers are accessed to set return values. (It must also be called even if no values are returned.)

In summary the extra functions needed to call XSB from a foreign module are:

`int xsb_query_save(CTXTc (byte) NumRegs)` *C function*

This function is used in a foreign routine that is called from XSB. It is used to save the current contents of the XSB registers and to initialize them to be prepared to accept a query. It must be called after a foreign routine collects its input arguments from the XSB registers and before it invokes any XSB predicate.

`int xsb_query_restore(CTXT)` *C function*

This function is used in a foreign routine that is called from XSB and in turn calls an XSB predicate. It is used to restore the previously saved contents of the XSB registers. It must be called after all XSB predicates have been called and returned, and before the current foreign routine sets its output parameters and returns to XSB.

An example where a foreign module and XSB call each other recursively can be found in the directory `$XSB_DIR/examples/XSB_calling_c` and files `fibr.[cH]` and `fibp.P`.

13.4 Foreign Modules That Link Dynamically with Other Libraries

Sometimes a foreign module might have to link dynamically with other (non-XSB) libraries. Typically, this happens when the foreign module implements an interface

to a large external library of utilities. One example of this is the package `libwww` in the XSB distribution, which provides a high-level interface to the W3C's Libwww library for accessing the Web. The library is compiled into a set of shared objects and the `libwww` module has to link with them as well as with XSB.

The problem here is that the loader must know at run time where to look for the shared objects to link with. On Unix systems, this is specified using the environment variable `LD_LIBRARY_PATH`; on Windows, the variable name is `LIBPATH`. For instance, under Bourne shell or its derivatives, the following will do:

```
LD_LIBRARY_PATH=dir1:dir2:dir3
export LD_LIBRARY_PATH
```

One problem with this approach is that this variable must be set before starting XSB. The other problem is that such a global setting might interact with other foreign modules.

To alleviate the problem, XSB dynamically sets `LD_LIBRARY_PATH` (`LIBPATH` on Windows) before loading foreign modules by adding the directories specified in the `-L` option in `ldoption`. Unfortunately, this works on some systems (Linux), but not on others (Solaris). One route around this difficulty is to build a runtime library search path directly into the object code of the foreign module. This can be specified using a loader flag in `ldoption`. The problem here is that different systems use a different flag! To circumvent this, XSB provides a predicate that tries to guess the right flag for your system:

```
runtime_loader_flag(+Hint,-Flag)
```

Currently it knows about a handful of the most popular systems, but this will be expanded. The argument `Hint` is not currently used. It might be used in the future to provide `runtime_loader_flag` with additional information that can improve the accuracy of finding the right runtime flags for various systems.

The above predicate can be used as follows:

```
...,
runtime_loader_flag(_,Flag),
fmt_write_string(LDoptions, '%sdir1:dir2:dir2 %s', args(Flag,OldLDoption)),
fmt_write(File, ':- ldoption(%s).', LDoptions),
file_nl(File).
```

13.5 Higher-Level Foreign Language Interface

The high-level foreign predicate interface was designed to release the programmer from the burden of having to write low-level code to transfer data from XSB to C and vice-versa. Instead, all the user needs to do is to describe each C function and its corresponding Prolog predicates in the `.H` files. The interface then automatically generates *wrappers* that translate Prolog terms and structures to proper C types, and vice-versa. These wrappers also check for type-correctness of arguments to the C function; in addition, in Unix-derived systems the wrappers are automatically compiled and loaded along with the foreign predicates in the `.c` file [7].

As with the lower-level foreign interfaces, when predicates are defined in a foreign module `myfile.[cH]`, the predicates must be explicitly imported from the module to be used [8]. For an example of using the higher level interface, see `$XSBDIR/examples/XSB_calling_c/secon`

13.5.1 Declaration of high level foreign predicates

The basic formats of a foreign predicate declaration are:

```
:- foreign_pred predname([+-]parg1, [+-]parg2,...)
                from funcname(carg1:type1, carg2:type2,
                    ...):functype.
```

and

```
:- private_foreign_pred predname([+-]parg1, [+-]parg2,...)
                from funcname(carg1:type1, carg2:type2,
                    ...):functype.
```

where:

`foreign_pred`

`private_foreign_pred`

These declare new foreign predicates. For most cases, the declaration `foreign_pred` can be used in both the multi-threaded and the sequential engine. The declaration `private_foreign_pred` needs to be used only in the multi-threaded engine

[7] for Windows, please see special instructions in Section 13.6.

[8] In Version 3.8, a foreign module that uses the higher-level C interface must be explicitly consulted before it can be used.

when the external foreign function, `funcname` contains a context parameter as its first argument because `funcname` needs to access thread-private data or other information from the context of the XSB thread (see Section 13.2.1). This case is uncommen, and mostly occurs for users who are creating XSB packages (e.g. the XASP interface to Smodels).

`predname`
: is the name of the foreign Prolog predicate.

`parg1, parg2, ...`
: are the predicate arguments. Each argument is preceded by either '+' or '-', indicating its mode as input or output respectively. The names of the arguments must be the same as those used in the declaration of the corresponding C function. If a C argument is used both for input and output, then the corresponding Prolog argument can appear twice: once with "+" and once with "-". In addition, a special argument `retval` is used to denote the argument that corresponds to the return value of the C function; it must always have the mode '-'.

`funcname`
: is the name of the function in the `.c` file. At compile-time a C function with name `predname` will be generated which will translate arguments from Prolog to C, call `funcname`, and then translate arguments back from C to Prolog.

`carg1, carg2, ...`
: is the list of arguments of the C function. The names used for the arguments must match the names used in the Prolog declaration.

`type1, type2, ...`
: are the types associated to the arguments of the C function. This is not the set of C types, but rather a set of descriptive types, as defined in Table 13.5.1.

`functype`
: is the return type of the C function.

Using the higher-level interface, the same C code can be used for both the sequential and the multi-threaded engines, and no context parameters are required in a user's C code unless thread context information is explicitly needed. However, a foreign module compiled for the single-threaded engine will need to be recompiled for the multi-threaded engine and vice-versa.

Descriptive Type	Mode Usage	Associated C Type	Comments
int	+	int	integer numbers
float	+	double	floating point numbers
atom	+	unsigned long	atom represented as an unsigned long
chars	+	char *	the textual representation of an atom is passed to C as a string
chars(*size*)	+	char *	the textual representation of an atom is passed to C as a string in a buffer of size *size*
string	+	char *	a prolog list of characters is passed to C as a string
string(*size*)	+	char *	a prolog list of characters is passed to C as a string
term	+	`prolog_term`	the unique representation of a term
intptr	+	int *	the location of a given integer
floatptr	+	double *	the location of a given floating point number
atomptr	+	unsigned long *	the location of the unique representation of a given atom
charsptr	+	char **	the location of the textual representation of an atom
stringptr	+	char **	the location of the textual representation of a list of characters
termptr	+	`prolog_term` *	the location of the unique representation of a term
intptr	-	int *	the integer value returned is passed to Prolog
floatptr	-	double *	the floating point number is passed back to Prolog
charsptr	-	char **	the string returned is passed to Prolog as an atom
stringptr	-	char **	the string returned is passed back as a list of characters
atomptr	-	unsigned long *	the number returned is passed back to Prolog as the unique representation of an atom
termptr	-	`prolog_term` *	the number returned is passed to Prolog as the unique representation of a term
chars(*size*)	+-	char *	the atom is copied from Prolog to a buffer, passed to C and converted back to Prolog afterwards
string(*size*)	+-	char *	the list of characters is copied from Prolog to a buffer, passed to C and back to Prolog afterwards
intptr	+-	int *	an integer is passed from Prolog to C and from C back to Prolog
floatptr	+-	double *	a float number is passed from Prolog to C, and back to Prolog
atomptr	+-	unsigned long *	the unique representation of an atom is passed to C, and back to Prolog
charsptr	+-	char **	the atom is passed to C as a string, and a string is passed to Prolog as an atom
stringptr	+-	char **	the list of characters is passed to C, and a string passed to Prolog as a list of characters
termptr	+-	`prolog_term` *	the unique representation of a term is passed to C, and back to Prolog

Table 13.1: Allowed combinations of types and modes, and their meanings

Table 13.5.1 provides the correspondence between the types allowed on the C side of a foreign module declaration and the types allowed on the Prolog side of the declaration.

In all modes and types, checks are performed to ensure the types of the arguments. Also, all arguments of type '-' are checked to be free variables at call time.

13.6 Compiling Foreign Modules on Windows and under Cygwin

Due to the complexity of creating makefiles for the different compilers under Windows, XSB doesn't attempt to compile and build DLL's for the Windows foreign modules automatically. However, for almost all typical cases the user should be able to easily adapt the sample makefile for Microsoft VC++:

```
XSB/examples/XSB_calling_c/MakefileForCreatingDLLs
```

It is important that the C program will have the following lines near the top of the file:

```
#include "xsb_config.h"
#ifdef WIN_NT
#define XSB_DLL
#endif
#include "cinterf.h"
```

Note that these same DLLs will work under Cygwin — XSB's C interface under Cygwin is like that under Windows rather than Unix.

If the above makefile cannot be adapted, then the user has to create the DLL herself. The process is, roughly, as follows: first, compile the module from within XSB. This will create the XSB-specific object file, and (if using the higher-level C interface) the *wrappers*. The *wrappers* are created in a file named `xsb_wrap_`*modulename*`.c`.

Then, create a project, using the compiler of choice, for a dynamically-linked library that exports symbols. In this project, the user must include the source code of the module along with the *wrapper* created by XSB. This DLL should be linked against the library

```
XSB\config\x86-pc-windows\bin\xsb.lib
```

which is distributed with XSB. In VC++, this library should be added as part of the linkage specification. In addition, the following directories for included header files must be specified as part of the preprocessor setup:

```
XSB\config\x86-pc-windows
XSB\prolog_includes
XSB\emu
```

In VC++, make sure you check off the "No precompiled headers" box as part of the "Precompiled headers" specification. All these options are available through the `Project»Settings` menu item.

13.7 Functions for Use in Foreign Code

In addition to functions for passing data between Prolog an C, XSB contains other functions that may be useful in Foreign C code. We mention a few here that pertain to throwing exceptions from C code (cf. Volume 1 Chapter 8: *Exception Handling*). These functions can be used by code that uses either the lower- or higher-level interface.

`void xsb_domain_error(CTXTdeclc char *valid_domain,Cell culprit,char *pred,int arity,int`
C function

Used to throw an ISO-style domain error from foreign code, indicating that `culprit` is not in domain `valid_domain` in argument `arg` of `pred/arity`.

Example: The code fragment

```
Cell num;
:
xsb_domain_error(CTXTc "not_less_than_zero",num,"atom_length",2,2);
```

in `atom_length/2` gives rise to the behavior

```
| ?- atom_length(abcde,-1).
++Error[XSB/Runtime/P]: [Domain (-1 not in domain not_less_than_zero)]
             in arg 2 of predicate atom_length/2)
```

`void xsb_existence_error(CTXTdeclc char *objType,Cell culprit,char *pred,int arity,int a`
C function

Used to throw an ISO-style existence error from foreign code, indicating that an object `culprit` of type `objType` does not exist, in argument `arg` of `pred/arity`.

Example: The code fragment

```
Cell tid;
:
xsb_existence_error(CTXTc "thread",reg[2],"xsb_thread_join",1,1);
```

in `thread_join/1` gives rise to a the behavior

```
| ?- thread_join(7).
++Error[XSB/Runtime/P]: [Existence (No thread 1 exists)]
             in arg 1 of predicate thread_join/1)
```

if a thread with thread id `7` does not exist.

`void xsb_instantiation_error(CTXTdeclc char *pred,int arity,int arg,char *state)` *C function*

Used to throw an ISO-style instantiation error from foreign code. If `state` is a `NULL` pointer, the message indicates that there is an instantiation error for argument `arg` of of `pred/arity`. If `state` is non-`NULL`, the message additionally indicates that argument `arg` must be `state`.

Example: The code fragment

```
xsb_instantiation_error(CTXTc "atom_length",2,1,NULL);
```

in `atom_length/2` gives rise to a the behavior

```
| ?- atom_length(X,Y).
++Error[XSB/Runtime/P]: [Instantiation]  in arg 1 of predicate atom_length/2
```

`void xsb_misc_error(CTXTdeclc char *message,char *pred,int arity)` *C function*

Used to throw a non ISO-error from foreign code, printing `message` and indicating that the error arose in `pred/arity`.

`void xsb_permission_error(CTXTdeclc char *op,char *obj,Cell culprit,char *pred,int arity` *C function*

Used to throw an ISO-style permission error from foreign code, indicating that an operation of type `op` on type `obj` is not permitted on `culprit`, in argument `arg` of `pred/arity`.

Example: The code fragment

```
xsb_permission_error(CTXTc "unlock mutex","mutex not held by thread",
                     xsb_thread_id,"mutex_unlock",2);
```

in `mutex_unlock/1` gives rise to a the behavior

```
| ?- mutex_unlock(mymut).
++Error[XSB/Runtime/P]: [Permission (Operation) unlock mutex on mutex not held
 by thread: 0] in predicate mutex_unlock/1)
```

if thread `0` does not own mutex `mymut`.

`void xsb_resource_error(CTXTdeclc char *resource,char *pred,int arity)` *C function*

Used to indicate that there are not sufficient resources of type `resource` for `pred/arity` to succeed.

Example: The code fragment

```
xsb_resource_error(th,"system threads","thread_create",2);
```

in `thread_create/1` gives rise to a the behavior

```
| ?- thread_create(X).
++Error[XSB/Runtime/P]: [Resource (system threads))] in predicate thread_create/2)
```

If the number of system threads has been exceeded.

`void xsb_type_error(CTXTdeclc char *valid_type,Cell culprit,char *pred,int arity,int arg` *C function*

Used to throw an ISO-style type error from foreign code, indicating that `culprit` is not in ISO type `valid_type` in argument `arg` of `pred/arity`.

Example: The code fragment

```
Cell num;
:
if (!isinteger(num)) xsb_type_error(CTXTc "integer",num,"atom_length",2,2);
```

in `atom_length/2` gives rise to the behavior

```
| ?- atom_length(foo,a).
++Error[XSB/Runtime/P]: [Type (a in place of integer)] in arg 2
of predicate atom_length/2)
```

`void xsb_throw(CTXTdeclc prolog_term Ball)` *C function*

Used to throw a Prolog term from C code, when an ISO-style error is not required. The term can be caught and handled by the Prolog predicate `catch/3` just as any other thrown term; however if it is not caught, XSB's default error handler will treat it as an unhandled exception.

Chapter 14

Embedding XSB in a Process

There are many situations in which it is desirable to use XSB as a rule- or constraint-processing subcomponent of a larger system that is written in another language. Depending on the intended architecture, it may be appropriate for XSB to reside in its own process, separate from other components of an application, and communicating through sockets, a database, or some other mechanism. However it is often useful for XSB to reside in the same process as other components. To do this, one wants to be able to *call* XSB from the host language, providing queries for XSB to evaluate. An interface for calling XSB from C is provided for this purpose and is described in this chapter. Based on this C interface, XSB can also be called from Java either through a JNI or a socket-based interface, as described in the documentation for InterProlog, available through `xsb.sourceforge.net`. To call XSB from Visual Basic, a DLL is created as described in this chapter, and additional declarations must be made in visual basic as described in the web page "How to use XSB DLL from Visual Basic" `http://xsb.sourceforge.net/vbdll.html`. In addition, the interface described in this chapter has also been extended to allow XSB to be called from Delphi and Ruby. However, since all of these interfaces – Java, Ruby, Delphi and Visual Basic – depend on XSB's C API, we refer in this chapter to C programs or threads calling XSB, although each of the examples suitably modified can be extended to other calling languages.

New to Version 3.1 are extensions to the C API to allow multiple XSB threads to be called from multiple C threads [1]. In this Chapter, we provide an overview of XSB's C API, and then elaborate its use through a series of examples, beginning with a single XSB thread called by a single C thread, then showing how a C thread can

[1] XSB's threading model is based on POSIX threads, which can be called in Windows through a variety of POSIX APIs – see Volume 1 chapter 8 *Multi-threaded Programming in XSB*.

interact with multiple XSB threads, and finally discuss how multiple XSB threads can interact with multiple POSIX threads. Finally, Section 14.3 describes each C function in the API.

14.1 Calling XSB from C

XSB provides several C functions (declared in `$XSBDIR/emu/cinterf.h` and defined in `$XSBDIR/emu/cinterf.c`), which can be called from C to interact with XSB as a subroutine. These functions allow a C program to interact with XSB in a number of ways.

- XSB may be initialized, using most of the parameters available from the command-line.
- XSB may then execute a series of *commands* or *queries*. A command is a deterministic query which simply succeeds or fails without performing any unification on the query term. On the other hand, a non-deterministic query can be evaluated so that its answer substitutions are retrieved one at a time, as they are produced, just as if XSB were called on a command line. Alternately a non-deterministic query can be closed in the case where not every answer to the query is needed. Only one query per thread can be active at a time. I.e., an application must completely finish processing one query to a given thread T (either by retrieving all the answers for it, or by issuing a call to `xsb_close_query()`, before trying to evaluate another using T.
- Finally, XSB can be closed, so that no more queries can be made to any XSB threads.

In general, while any functions in the C API to XSB can be intermixed, the functions can be classified as belonging to three different levels.

- *A VarString level* which uses an XSB-specific C-type definition for variable-length strings (Section 14.4), to return answers.
- *A fixed-string level* provides routines that return answers in fixed-length strings.
- *A register-oriented level* that requires users to set up queries by setting registers for XSB which are made globally available to calling functions. The mechanisms for this resemble the lower-level C interface discussed in Chapter 13. This level of interface should only be used for the single-threaded applications, as it is

difficult to prevent race-conditions at this level of interface when multiple C threads are used to call XSB.

The appropriate level to use depends on the nature of the calling program, the speed desired, and the expertise of the programmer. By and large, functions in the `VarString` level are the the easiest and safest to use, but they depend on a C type definition that may not be available to all calling programs (e.g. it may be difficult to use if the calling program is not directly based on C, such as Visual Basic or Delphi). For such applications functions from the fixed-string level would need to be used instead. In general, most applications should use either functions from the `VarString` or the fixed-string level, rather than the register-oriented level. This latter level should only be used by programmers who are willing to work at a low interface level, when the utmost speed is needed by an application, and when multiple threads do not need to interact with XSB.

14.2 Examples of Calling XSB

We introduce a series of examples of how XSB would be called using the string-level interfaces. Simple examples of the register-level interface are given in the `XSB/examples/c_calling_XSB` subdirectory, in files `cmain.c`, `cmain2.c`, `ctest.P`, and `Makefile`, but are not discussed in this section.

We structure out discussion by first showing how to construct a C program to call the single-threaded engine alone in Section 14.2.1. This example is mostly pedagogic: with a small amount of extra coding a C program can be constructed to call both the single- and the multi-threaded engine, and these extensions are discussed in Section 14.2.2. Next, we show how to a C program can call and manage multiple XSB threads in Section 14.2.3. Finally, we show how multiple XSB threads can interact with multiple C threads in Section 14.2.3.

14.2.1 The XSB API for the Sequential Engine Only

We start with a simple program shown, in Figure 14.1, that will call the following XSB predicate

```
p(a,b,c).
p(1,2,3).
p([1,2],[3,4],[5,6]).
p(A,B,A).
```

```
r(c,b,a).
r(3,2,1).
r([5,6],[3,4],[1,2]).
r(_A,B,B).
```

and backtrack through unifying answers (cf. `$XSBDIR/examples/c_calling_xsb/edb.P`). . This example will only compile properly if the sequential engine is used, and its style is `not` recommended: it will be shown in Section 14.2.2 how to extend the style.

We discuss the program in Figure 14.1 in detail. This program, slightly modified so that it compiles with the multi-threaded engine is in `$XSBDIR/examples/c_calling_xsb/cvartest.c`. An executable for this program can be make most easily by calling `$XSBDIR/examples/c_calling_xsb/mak` which makes the executable `cvstest`.

The program begins by including some standard C headers: note that `string.h` is needed for string manipulation routines such as `strcpy`. In addition, the XSB library header `cinterf.h` is necessary for the XSB C API. Since the program in Figure 14.1 uses functions in the `VarString` interface, within `main()` the routine `XSB_StrDefine(return_string)` declares and initializes a structure of type `VarString`, named `return_string`.

The next order of business is to initialize XSB. In order to do this, `xsb_init_string()` needs to know the installation directory for XSB, which must be passed as part of the initialization string. In Figure 14.1 this is done by manipulating the path of the executable (`cvstest`) that calls XSB. In fact any other approach would also work as long as the XSB installation directory were passed. Within the initialization string, other command line arguments can be passed to XSB if desired with the following exceptions: the arguments `-B` (boot module), `-D` (command loop driver), `-i` (interpreter) and `-d` (disassembler) cannot be used when calling XSB from a foreign language [2]. As a final point on initialization, note that the function `xsb_init()` can also be used to initialize XSB based on an argument vector and count (see Section 14.3).

Note that the calling program checks for any errors returned by `xsb_init_string()` and other API commands. In general, `xsb_init_string()` may throw an error if the XSB's installation directory has become corrupted, or for similar reasons. This mechanism for error handling is different than that used if XSB is called in its usual stand-alone mode, in which case such an error would cause XSB to exit). An error

[2] In previous versions of XSB, initialization from the C level required a `-n` option to be passed. This is no longer required.

```
#include <stdio.h>
#include <string.h>

/* cinterf.h is necessary for the XSB API, as well as the path manipulation routines*/
#include "cinterf.h"

extern char *xsb_executable_full_path(char *);
extern char *strip_names_from_path(char*, int);

int main(int argc, char *argv[]) {

  char init_string[1024];
  int rc;
  XSB_StrDefine(return_string);

  /* xsb_init_string() relies on the calling program to pass the absolute or relative
     path name of the XSB installation directory. We assume that the current
     program is sitting in the directory ../examples/c_calling_xsb/
     To get the installation directory, we strip 3 file names from the path. */

  strcpy(init_string,strip_names_from_path(xsb_executable_full_path(argv[0]),3));

  if (xsb_init_string(init_string) == XSB_ERROR) {
    fprintf(stderr,"++initializing XSB: %s/%s\n",xsb_get_init_error_type(),
            xsb_get_init_error_message());
    exit(XSB_ERROR);
  }

  /* Create command to consult a file: edb.P, and send it. */
  if (xsb_command_string("consult('edb.P').") == XSB_ERROR)
    fprintf(stderr,"++Error consulting edb.P: %s/%s\n",xsb_get_error_type(),xsb_get_error_messa

  rc = xsb_query_string_string("p(X,Y,Z).",&return_string,"|");
  while (rc == XSB_SUCCESS) {
    printf("Return %s\n",(return_string.string));
    rc = xsb_next_string(&return_string,"|");
  }

 if (rc == XSB_ERROR)
    fprintf(stderr,"++Query Error: %s/%s\n"xsb_get_error_type(),xsb_get_error_message());

  xsb_close();
}
```

Figure 14.1: Calling the Sequential Engine Using the `VarString` Interface

returned by XSB's API are similar to an error ball described in Volume 1 *Exception Handling* in that it has both a *type* and a *message*. For normal Prolog exceptions, XSB's API will throw the same kinds of errors as XSB called in a stand-alone (or server) mode, i.e. instantiation errors, type errors, etc. However XSB's API adds two new error types:

- `init_error` is used as the type of an error discovered upon initialization of XSB, before query and command processing has begun. If an `init_error` is raised, XSB has not been properly initialized and will not run.
- `unrecoverable_error` is used to indicate that XSB has encountered an error, (such as a memory allocation error), during command or query processing from which it cannot recover. Such an error would cause XSB to immediately exit if it were called in a stand-alone mode. In general the calling program should handle unrecoverable errors as fatal since there is a good chance that the error conditions will affect the calling program as well as XSB.

Errors raised by `xsb_init_string()` usually have type `init_type`.

and a string pointer to the associated message can be found by the function `xsb_get_init_error_message()`.

As can be seen from the example, handling errors from commands is done in manner similar to that of initialization. For non-initialization errors, a string pointer to the type can be obtained by `xsb_get_error_type()`, while a string pointer to the message can be obtained by `xsb_get_error_message()`.

Next in Figure 14.1 the file `edb.P` is consulted (containing the `p/3` and `r/3` predicates shown above). Note, that the argument to `xsb_command_string` must be a syntactically valid Prolog term ending with a period, otherwise a syntax error will be thrown, which may be displayed through `xsb_get_error_type()` and `xsb_get_error_message()` [3].

Queries to XSB are a little more complicated than commands. Since a query may return multiple solutions, a query should usually be called from inside a loop. In Figure 14.1, the query is opened with `xsb_query_string()`. If the query has at least one answer, `xsb_query_string()` will return `XSB_SUCCESS`; if the query fails, it will return `XSB_FAILURE`, and if there is an exception it will return `XSB_ERROR` as usual. Any answer will be returned as a string in the `VarString return_string`, and each

[3]Most XSB errors are handled in this manner when XSB is called through its API. A few errors will print directly to `stderr` and some XSB warnings will print to `stdwarn` which upon startup is dup-ed to `stderr`.

argument of the query will be separated by the character `|`. Thus, in our example, the first answer will write the string

```
a|b|c
```

Once a query has been opened, subsequent answers can be obtained via `xsb_next_string()`. These answers are written to `return_string` in the same manner as `xsb_query_string_string()`.

```
1|2|3
[1,2]|[3,4]|[5,6]
_h102|_h116|_h102
```

A query is automatically closed when no more answers can be derived from it. Alternately, a query that may have answers remaining can be closed using the command `xsb_close_query()`. If the calling application will need to pass more queries or commands to XSB nothing need be done at this point: a new queries or commands can be invoked using one of the functions just discussed. However if the calling process is finished with XSB and will never need it again during the life of the process, it can call `xsb_close()`.

An Example using Fixed Strings

Figure 14.2 shows a fragment of code indicating how the previous example would be modified if the fixed-string interface were used. Note that `return_string` now becomes a pointer to explicitly malloc-ed memory. To open the query `p(X,Y,Z)` the function `xsb_query_string_string_b()` is called, with the `_b` indicating that a fixed buffer is being used rather than a `VarString`. The call is similar to `xsb_query_string_string()`, except that the length `anslen` of the buffer pointed to by `return_string` is now also required. If the answer to be returned (including separators) is longer than `anslen`, `xsb_query_string_string_b()` will return `XSB_OVERFLOW`. If this happens, a new answer buffer can be used (here the old one is realloc-ed) and the answer retrieved via `xsb_get_last_answer_string`. Similarly, further answers are obtained via `xsb_next_string_b()` whose length must be checked. Thus the only difference between the fixed-string level and the `VarString` level is that the length of each answer should be checked and `xsb_get_last_answer_string()` called if necessary.

14.2.2 The General XSB API

The previous section showed how to use the XSB API with both the `VarString` type and without, but did not consider the multi-threaded engine. In fact, there

```
int retsize = 15;
char *return_string;
int anslen;

return_string = malloc(retsize);

rc = xsb_query_string_string_b(CTXTc "p(X,Y,Z).",return_string,retsize,&anslen,"|");

while (rc == XSB_SUCCESS || rc == XSB_OVERFLOW) {

  if (rc == XSB_OVERFLOW) {
    return_string = (char *) realloc(return_string,anslen);
    return_size = anslen;
    rc = xsb_get_last_answer_string(CTXTc return_string,retsize,&anslen);
  }

  printf("Return %s %d\n",return_string,anslen);
  rc = xsb_next_string_b(CTXTc return_string,15,&anslen,"|");
}
```

Figure 14.2: Calling XSB using the Fixed String Interface

are different ways to use XSB's multi-threading that can have advantages for various situations. In the first mode, threads are managed from Prolog, with a single XSB thread called from the API; that XSB thread can then create another XSB thread that does work, and the first thread can return almost immediately to handle more requests from the API's caller. A second model allows the caller to manipulate a pool of several XSB threads, so that different XSB threads may be called from different threads over the API. In this model each C, Java, Ruby, or other thread could a number of different Prolog threads. In this section we sketch how to use the API to illustrate the first model, and sketch the second model in the next section.

Figure 14.3 shows how relevant portions of the previous `VarString` example can be adapted to use the multi-threaded engine. The main change is that a new variable is introduced on the C side that points to the context of the main thread. As pointed out in Chapter 13, each thread in the multi-threaded engine has a *context* in which is kept much of its thread-specific data (excluding tables and dynamic code). Of the threads running in the multi-threaded engine the thread created upon the call to `xsb_init()` is designated as the `main thread`, and is closed only upon calling `xsb_close()`.

Within the multi-threaded engine, a call to an API function such as `xsb_query_string_string()` is actually a call to a specific thread to do some work (using a thread context pointer). Accordingly, since any errors produced will be specific to a given thread, all calls to error reporting functions are also thread-specific. If no specific thread is needed, it may be best just to use the main thread, which is what is done in Figure 14.3. The thread context pointer `th` is initialized to the main thread using the API macro `xsb_get_main_thread()`. Afterwards, this pointer is passed into the various interface functions by making use of XSB macros defined in `context.h` In the multi-threaded engine, these macros are defined as

```
#define CTXT th
#define CTXTc th,
```

while in the single-threaded engine they are defined as empty strings, as is `xsb_get_main_thread()`. As a result the code in Figure 14.3 will compile and run properly both for the single-threaded and the multi-threaded engines.

At this stage, suppose one wanted a new thread to execute a specific command, say `do_foo`. In this case, a C call such as

```
xsb_query_string_string(CTXTc "thread_create(do_foo,Id).",&return_string,"|")
```

creates a thread to execute the command, and returns the thread id of the newly created thread in `return_string`. The behavior of this newly created thread is

```
.....

/* context.h is necessary for the type of a thread context. */
#include "context.h"

int main(int argc, char *argv[])
{

  char init_string[MAXPATHLEN];
  int rc;
  XSB_StrDefine(return_string);

  strcpy(init_string,strip_names_from_path(xsb_executable_full_path(argv[0]),3));
  if (xsb_init_string(init_string) == XSB_ERROR) {
    fprintf(stderr,"++initializing XSB: %s/%s\n",xsb_get_init_error_type(),
            xsb_get_init_error_message());
    exit(XSB_ERROR);
  }

#ifdef MULTI_THREAD
  th_context *th = xsb_get_main_thread();
#endif

  /* Create command to consult a file: edb.P, and send it. */
  if (xsb_command_string(CTXTc "consult('edb.P').") == XSB_ERROR)
    fprintf(stderr,"++Error consulting edb.P: %s/%s\n",xsb_get_error_type(CTXT),
            xsb_get_error_message(CTXT));

  rc = xsb_query_string_string(CTXTc "p(X,Y,Z).",&return_string,"|");
  while (rc == XSB_SUCCESS) {
    printf("Return %s\n",(return_string.string));
    rc = xsb_next_string(CTXTc &return_string,"|");
  }

 if (rc == XSB_ERROR)
    fprintf(stderr,"++Query Error: %s/%s\n",xsb_get_error_type(CTXT),xsb_get_error_message(CTX

  xsb_close();
}
```

Figure 14.3: Calling the Single- or Multi-Threaded Engine Using the `VarString` Interface

exactly the same as if it were created from the XSB command line: in particular the newly created thread will automatically exit upon completion of its command. As a somewhat technical point, there are two different ways of referring to XSB threads. The foreign language interfaces described in Chapter 13 and here use pointers to thread contexts so that the interfaces use much of the same code as the XSB engine. However Prolog refers to threads using thread identifiers. The two different forms can be converted into each other by the functions `xsb_thread_id_to_context()` and `xsb_thread_context_to_id()`.

14.2.3 Managing Multiple XSB Threads through the API

The ability to pass thread contexts into query and command functions allows a great deal of flexibility [4]. Once XSB is initialized, XSB threads can be created from C and can execute independently of each other, effectively giving the ability for different calling threads to query XSB in a mechanism reminiscent of database cursors.

Figure 14.4 illustrates a very simple example of this. XSB is initialized and the file `edb.P` consulted exactly as in Figure 14.4. However, the function `xsb_ccall_thread_create()` causes the XSB thread `p_th` to create a new thread, causes the new thread to call the same command loop as the main thread, and sets `r_th` to point to the context of the new thread. The new thread `r_th` can be used for commands or queries just as `p_th`. Figure 14.4 shows that queries to the two threads can be interleaved, and errors for both threads can be checked and reported independently.

It is important to note that since each thread created by `xsb_ccall_thread_create()` goes into a command-loop similar to the command loop, it will stay around until it is explicitly killed or until XSB is closed. The call

```
xsb_kill_thread(r_th);
```

is needed to make `r_th` to exit. Once a thread is exited, all of its data structures will be freed, including those that support `xsb_get_error_type()` and `xsb_get_error_message()` [5].

[4] For the sake of brevity, we sometimes abuse notation and do not always distinguish between thread-contexts and their pointers.

[5] Note that causing XSB's main thread to exit will cause the entire process to exit – not just XSB.

```
.....
/* context.h is necessary for the type of a thread context. */
#include "context.h"

int main(int argc, char *argv[])
{

  static th_context *p_th, *r_th;
  char init_string[MAXPATHLEN];
  int rcp, rcr;
  XSB_StrDefine(p_return_string);
  XSB_StrDefine(r_return_string);

  strcpy(init_string,strip_names_from_path(xsb_executable_full_path(argv[0]),3));

  if (xsb_init_string(init_string)) {
    fprintf(stderr,"%s initializing XSB: %s/%s\n",xsb_get_init_error_type(),
            xsb_get_init_error_message());cin
    exit(XSB_ERROR);
  }

  p_th = xsb_get_main_thread();

  /* Create command to consult a file: edb.P, and send it. */
  if (xsb_command_string(p_th, "consult('edb.P').") == XSB_ERROR)
    fprintf(stderr,"++Error consulting edb.P: %s/%s\n",xsb_get_error_type(p_th),
    xsb_get_error_message(p_th));

  xsb_ccall_thread_create(p_th,&r_th);

  rcp = xsb_query_string_string(p_th,"p(X,Y,Z).",&p_return_string,"|");
  rcr = xsb_query_string_string(r_th,"r(X,Y,Z).",&r_return_string,"|");

  while (rcp == XSB_SUCCESS && rcr == XSB_SUCCESS) {

    printf("Return p %s\n",(p_return_string.string));
    rcp = xsb_next_string(p_th, &p_return_string,"|");

    printf("Return r %s\n",(r_return_string.string));
    rcr = xsb_next_string(r_th, &r_return_string,"|");
  }

 if (rcp == XSB_ERROR)
   fprintf(stderr,"++Query Error p: %s/%s\n",xsb_get_error_type(p_th),xsb_get_error_message(p_t
 if (rcr == XSB_ERROR)
   fprintf(stderr,"++Query Error r: %s/%s\n",xsb_get_error_type(r_th),xsb_get_error_message(r_t

 xsb_close();
}
```

14.2.4 Calling Multiple XSB Threads using Multiple C Threads

Figure 14.4 shows how two XSB threads can be created, can receive different queries and can interleave their backtracking and answer return. Although Figure 14.4 demonstrated only backtracking through simple predicates, the mechanism employed works for complicated examples using tabling, dynamic code, and other features. All this provides a sophisticated interface, but it is not "fully" multi-threaded in the following sense. When a C thread T causes XSB to execute a command or query the thread must wait until the calling function returns before proceeding. In certain applications it may be useful, for example, for T to create a C thread T_{new} which runs asynchronously from T, executing the XSB command or query and then exiting. Alternately, an application may want to have a pool of C threads that can interact with a pool of XSB threads.

XSB's C API has been designed to support these features. Figure 14.5 shows fragments of Figure 14.4 rewritten so that the routines to print out the answers to the queries `p(X,Y,Z)` and `r(X,Y,Z)` can be called from C threads specially designed for this purpose. More specifically, the routine `query_ps()` calls `p_th` to query `p(X,Y,Z)` and backtrack through its answers – its use of a single `void *` argument and a `void *` return reflect the requirements of functions that are to be called using `pthread_create()`.

We note several points about this example. First the XSB API is a low-level API that can be used to build application specific interfaces, and some experience with pthread programming is useful if multiple XSB threads are called from multiple C threads. For instance, one issue is fairness. When called from the C API each XSB thread X_T makes use of mutexes to ensure that it answers only one query or command at a time. If multiple C threads are are waiting for X_T to respond to requests or queries, there is no guarantee that the requests will be processed in any sort of order, or even that a request will eventually be handled (In order to ensure this, the calling program would have to use a queue or some other scheduling mechanism to send requests to the XSB thread). In addition, it is important to note that, *the main XSB thread should only be called from the C thread that initialized XSB.*. This restriction is due to the current design of synchronizing an XSB thread with calling threads, and may be lifted in the future.

Protected and Non-Protected API Functions

Example 14.5 shows that, when the `Varstring` functions are used, if a single calling thread opens a query to an XSB thread X_T, X_T will be protected from queries

```
.....

void *query_ps(void * arg) {
  int rc;
  th_context *p_th;
  XSB_StrDefine(p_return_string);
  p_th = (th_context *)arg;

  rc = xsb_query_string_string(p_th,"p(X,Y,Z).",&p_return_string,"|");
  while (rc == XSB_SUCCESS) {
    printf("Return p %s\n",(p_return_string.string));
    rc = xsb_next_string(p_th, &p_return_string,"|");
  }

 if (rc == XSB_ERROR)
   fprintf(stderr,"++Query Error p: %s/%s\n",xsb_get_error_type(p_th),xsb_get_error_message(p_t
 return NULL;
}


int main(int argc, char *argv[]) {

  char init_string[MAXPATHLEN];
  static th_context *p_th, *r_th;
  int pstatus, rstatus;
  pthread_t pthread_id,rthread_id;
  XSB_StrDefine(p_return_string);
  XSB_StrDefine(r_return_string);

.....

  main_th = xsb_get_main_thread();

  /* Create command to consult a file: edb.P, and send it. */
  if (xsb_command_string(xsb_get_main_thread(), "consult('edb.P').") == XSB_ERROR)
    fprintf(stderr,"++Error consulting edb.P: %s/%s\n",xsb_get_error_type(main_th),
    xsb_get_error_message(main_th));

  xsb_ccall_thread_create(main_th,&r_th);
  xsb_ccall_thread_create(main_th,&p_th);

  pthread_create(&rthread_id,NULL,command_rs,r_th);
  pthread_create(&pthread_id,NULL,command_ps,p_th);
  pthread_create(&rthread_id,NULL,command_rs,r_th);
  pthread_create(&pthread_id,NULL,command_ps,p_th);

  rstatus = pthread_join(rthread_id,&rreturn);
  if (rstatus != 0) fprintf(stderr,"R join returns status %d\n",rstatus);
  pstatus = pthread_join(pthread_id,&preturn);
  if (pstatus != 0) fprintf(stderr,"P join returns status %d\n",pstatus);
```

and commands posed by other C threads until the query is closed, failed out of, or exits via an error. In fact, queries (and commands) are protected when the `Varstring` or fixed string interfaces are used. However, consider what may happen when the register level interface is used. In this case, a calling thread may call one or more API functions to set up the registers, execute a command or query, call several more API functions to obtain the output, and so on. For this reason, if an application uses API commands that depend on user manipulation of registers (`xsb_command()`,`xsb_query()`,`xsb_query_string()`, and `xsb_next()`) the user must ensure that only one calling thread interacts with an XSB thread when that thread in the course of executing a command or query. See `$XSB_DIR/examples/c_calling_xsb/cregs_th` for an example of how mutexes can be used to protect XSB threads.

When writing multi-threaded applications in XSB, be sure to be aware of how multiple threads share (and do not share) dynamic data and tables. By default dynamic predicates (and tables) are unique to a given thread. For data to be shared by multiple threads, a predicate must be declared to be `shared`. See section 7.2 for details.

14.3 A C API for XSB

14.3.1 Initializing and Closing XSB

`int xsb_init_string(char *options)` *C function*

This function is used to initialize XSB via an initialization string `*options`, and must be called before any other calls can be made. The initialization string must include the path to the XSB directory installation directory `$XSB_DIR`, which is expanded to an absolute path by XSB. Any other command line options may be included just as in a command line except `-D`, `-d`, `-B` and `-i`. For example, a call from an executable in a sibling directory of XSB might have the form

```
xsb_init_string("../XSB -e startup.");
```

which initializes XSB with the goal `?- startup.`

Return Codes

- `XSB_SUCCESS` indicates that initialization returned successfully.
- `XSB_ERROR`
  - `init_error` if any error occurred during initialization.

  - `permission_error` if `xsb_init_string()` is called after XSB has already been correctly initialized.

`int xsb_init(int argc, char *argv[])` *C function*

This function is a variant of `xsb_init_string()` which passes initialization arguments as an argument vector: `argc` is the count of the number of arguments in the `argv` vector. The `argv` vector is exactly as would be passed from the command line to XSB.

- `argv[0]` must be an absolute or relative path name of the XSB installation directory (*i.e.*, `$XSB_DIR`). Here is an example, which assumes that we invoke the C program from the XSB installation directory.

```
int main(int argc, char *argv[])
{
  int myargc = 1;
  char *myargv[1];

  /* XSB_init relies on the calling program to pass the addr of the XSB
     installation directory. From here, it will find all the libraries */
  myargv[0] = ".";

  /* Initialize xsb */
  xsb_init(myargc,myargv);
}
```

The return codes for `xsb_init()` are the same as those for `xsb_init_string()`.

`int xsb_close()` *C function*

This routine closes the entire connection to XSB . After this, no more calls can be made (not even calls to `xsb_init_string()` or `xsb_init()`). In Version 3.8, no guarantee is made that all space used by XSB will be restored to the process (even when the process has dynamically linked to XSB), but space for any XSB tables is freed.

Return Codes

- `XSB_SUCCESS` indicates that XSB was closed successfully.
- `XSB_ERROR`
  - `permission_error` if `xsb_closed()` when XSB has not been (correctly) initialized.

14.3.2 Passing Commands to XSB

`int xsb_command_string(th_context *th, char *cmd)` *C function*

This function passes a command to the XSB thread designated by `th` (the first argument is not used in the single-threaded engine). No query can be active in `th` when the command is called. The command is a string consisting of a Prolog (or HiLog) term terminated by a period (.).

When used in the multi-threaded engine, `xsb_command_string` protects the called thread from API calls from other pthreads until the command is finished.

Return Codes

- `XSB_SUCCESS` indicates that the command succeeded.
- `XSB_FAILURE` indicates that the command failed.
- `XSB_ERROR`
  - `permission_error` if `xsb_command_string()` is called while a query is open in `th`.
  - Otherwise, any queries thrown during execution of the command are accessable through `xsb_get_error_type(th)` and `xsb_get_error_message(th)`.

`int xsb_command(th_context *th)` *C function*

This function passes a command to the XSB thread designated by `th` (the first argument is not used in the single-threaded engine). Any previous query must have already been closed. Before calling `xsb_command()`, the calling program must construct the term representing the command in register 1 in the XSB thread's space. This can be done by using the `c2p_*` (and `p2p_*`) routines, which are described in Section 13.2.3 below. Register 2 may also be set before the call to `xsb_query()` (using `xsb_make_vars(int)` and `xsb_set_var_*()`) in which case any variables set to values in the `ret/n` term will be so bound in the call to the command goal. `xsb_command` invokes the command represented in register 1 and returns `XSB_SUCCESS` if the command succeeds, `XSB_FAILURE` if it fails, and `XSB_ERROR` if an error is thrown while executing the command.

When used in the multi-threaded engine, `xsb_command_string` *does not protect* the called thread from API calls from other pthreads until the command is finished. It is the user's responsibility to protect the XSB thread, using a mutex or other concurrency control, from the time the goal begins to be constructed in the register 1 until the command has completed.

Apart from the steps necessary to formulate the query and the lack of protection of the XSB thread, the behavior of `xsb_command()` is similar to that of `xsb_command_string()`, including its return codes.

14.3.3 Querying XSB

`int xsb_query_string_string(th_context *th, char *query, VarString *buff,char *sep)`

C function

This function opens a query to the XSB thread designated by `th` (the first argument is not used in the single-threaded engine); it returns the first answer (if there is one) as a `VarString`. Any previous query to `th` must have already been closed. Any query may return multiple data answers. The first is found and made available to the caller as a result of this call. To get any subsequent answers, `xsb_next_string()` must be called. An example call is:

```
rc = xsb_query_string_string(th, "append(X,Y,[a,b,c]).",buff,";");
```

The second argument is the period-terminated query string. The third argument is a pointer to a variable string buffer in which the subroutine returns the answer (if any.) The variable string data type `VarString` is explained in Section 14.4. (Use `xsb_query_string_string_b()` if you cannot declare a parameter of this type in your programming language.) The last argument is a string provided by the caller, which is used to separate arguments in the returned answer. For the example query, `buff` would be set to the string:

```
[];[a,b,c]
```

which is the first answer to the append query. There are two fields of this answer, corresponding to the two variables in the query, `X` and `Y`. The bindings of those variables make up the answer and the individual fields are separated by the `sep` string, here the semicolon (`;`). In the answer string, XSB atoms are printed without quotes. Complex terms are printed in a canonical form, with atoms quoted if necessary, and lists produced in the normal list notation.

When used in the multi-threaded engine, `xsb_query_string_string` protects the called thread from API calls from other pthreads until the entire query is finished.

Return Codes

- `XSB_SUCCESS` indicates that the query succeeded.
- `XSB_FAILURE` indicates that the query failed.
- `XSB_ERROR`
  - `permission_error` if `xsb_query_string_string()` is called while a query to `th` is open.

  - Otherwise, any errors thrown during execution of the query are accessable through `xsb_get_error_type()` and `xsb_get_error_message()`.

`int xsb_query_string_string_b(th_context *th,char *query,char *buff,int bufflen,int *anslen,cha`

This function provides a lower-level alternative to `xsb_query_string_string` (not using the `VarString` type), which makes it easier for non-C callers (such as Visual Basic or Delphi) to access XSB functionality. Any previous query to `th` must have already been closed. Any query may return possibly multiple data answers. The first is found and made available to the caller as a result of this call. To get any subsequent answers, `xsb_next_string_b()` or a similar function must be called. The first and last arguments are the same as in `xsb_query_string_string()`. The `buff`, `bufflen`, and `anslen` parameters are used to pass the answer (if any) back to the caller. `buff` is a character array provided by the caller in which the answer is returned. `bufflen` is the length of the buffer (`buff`) and is provided by the caller. `anslen` is returned by this routine and is the length of the computed answer. If that length is less than `bufflen`, then the answer is put in `buff` (and null-terminated). If the answer is longer than will fit in the buffer (including the null terminator), then the answer is not copied to the buffer and `XSB_OVERFLOW` is returned. In this case the caller can retrieve the answer by providing a bigger buffer (of size greater than the returned `anslen`) in a call to `xsb_get_last_answer_string()`.

When used in the multi-threaded engine, `xsb_query_string_string_b` protects the called thread from API calls from other pthreads until the entire query is finished.

Return Codes

- `XSB_SUCCESS` indicates that the query succeeded.
- `XSB_FAILURE` indicates that the query failed.
- `XSB_ERROR`
  - `permission_error` if `xsb_query_string_string_b()` is called while a query to `th` is open.
  - Otherwise, any queries thrown during execution of the command are accessable through `xsb_get_error_type()` and `xsb_get_error_message()`.
- `XSB_OVERFLOW` indicates that the query succeeded, but the answer was too long for the buffer.

`int xsb_query(th_context *th)` *C function*

This function passes a query to the XSB thread `th`. Any previous query to `th` must have already been closed. Any query may return possibly multiple

data answers. The first is found and made available to the caller as a result of this call. To get any subsequent answers, `xsb_next()` or a similar function must be called. Before calling `xsb_query()` the caller must construct the term representing the query in the XSB thread's register 1 (using routines described in Section 13.2.3 below.) If the query has no answers (i.e., just fails), register 1 is set back to a free variable and `xsb_query()` returns `XSB_FAILURE`. If the query has at least one answer, the variables in the query term in register 1 are bound to those answers and `xsb_query()` returns `XSB_SUCCESS`. In addition, register 2 is bound to a term whose main functor symbol is `ret/n`, where n is the number of variables in the query. The main subfields of this term are set to the variable values for the first answer. (These fields can be accessed by the functions `p2c_*`, or the functions `xsb_var_*`, described in Section 13.2.3 below.) Thus there are two places the answers are returned. Register 2 is used to make it easier to access them. Register 2 may also be set before the call to `xsb_query()` (using `xsb_make_vars(int)` and `xsb_set_var_*()`) in which case any variables set to values in the `ret/n` term will be so bound in the call to the goal.

When used in the multi-threaded engine, `xsb_query` *does not protect* the called thread from API calls from other pthreads until the query is finished, or even when the registers are being accessed. It is the user's responsibility to protect the XSB thread, using a mutex or other concurrency control, from the time the goal begins to be constructed in the register 1 until the query is closed, failed, or exited upon error.

`int xsb_get_last_answer_string(th_context *th, char *buff, int bufflen, int *anslen)` *C function*

This function is used only when a call `xsb_query_string_string_b()` or `xsb_next_string_b()` to `th` returns `XSB_OVERFLOW`, indicating that the buffer provided was not big enough to contain the computed answer. In that case the user may allocate a larger buffer and then call this routine to retrieve the answer (that had been saved.) Only one answer is saved per thread, so this routine must called immediately after the failing call in order to get the right answer. The parameters are the same as the 2nd through 4th parameters of `xsb_query_string_string_b()`.

Return Codes

- `XSB_OVERFLOW` indicates that the answer was `still` too long for the buffer.

`int xsb_query_string(th_context *th,char *query)` *C function*

This function passes a query to the XSB thread `th`. The query is a string consisting of a term that can be read by the XSB reader. The string must

be terminated with a period (.). Any previous query must have already been closed. In all other respects, `xsb_query_string()` is similar to `xsb_query()`, except the only way to retrieve answers is through Register 2. The ability to create the return structure and bind variables in it is particularly useful in this function.

When used in the multi-threaded engine, `xsb_query_string` *does not protect* the called thread from API calls from other pthreads until the query is finished, or even when the registers are being accessed. It is the user's responsibility to protect the XSB thread, using a mutex or other concurrency control, from the time the goal begins to be constructed in the register 1 until the query is closed, failed, or exited upon error.

Return Codes

- `XSB_SUCCESS` indicates that the query succeeded.
- `XSB_FAILURE` indicates that the query failed.
- `XSB_ERROR` indicates that an error occurred while executing the query.

`int xsb_next_string(th_context *th,VarString *buff,char *sep)` *C function*

This routine is called after `xsb_query_string()` to retrieve a subsequent answer in `buff`. If a query is not open in `th`, an error is returned. This function treats answers just as `xsb_query_string_string()`. For example after the example call

```
rc = xsb_query_string_string(th,"append(X,Y,[a,b,c]).",buff,";");
```

which returns with buff set to

```
[];[a,b,c]
```

Then a call:

```
rc = xsb_next_string(th,buff,";");
```

returns with buff set to

```
[a];[b,c]
```

the second answer to the indicated query.

In the multi-threaded engine, `xsb_next_string()` protects the XSB thread from concurrent access by other threads as long as the query was invoked by `xsb_query_string_string(_b)`.

Return Codes

- `XSB_SUCCESS` indicates that the query succeeded.
- `XSB_FAILURE` indicates that the query failed.
- `XSB_ERROR` indicates that an error occurred while executing the query.

`int xsb_next_string_b(th_context *th, char *buff, int bufflen, int *anslen, char *sep)` *C function*

This function is a variant of `xsb_next_string()` that does not use the `VarString` type. Its parameters are the same as the 3rd through 6th parameters of `xsb_query_string_string_b` The next answer to the current query is returned in `buff`, if there is enough space. If the buffer would overflow, this routine returns `XSB_OVERFLOW`, and the answer can be retrieved by providing a larger buffer in a call to `xsb_get_last_answer_string_b()`. In any case, the length of the answer is returned in `anslen`.

In the multi-threaded engine, `xsb_next_string()` protects the XSB thread from concurrent access by other threads as long as the query was invoked by `xsb_query_string_string(_b)`.

Return Codes

- `XSB_SUCCESS` indicates that backtracking into the query succeeded.
- `XSB_FAILURE` indicates that backtracking into the query failed.
- `XSB_ERROR` indicates that an error occurred while further executing the query.
- `XSB_OVERFLOW` indicates that backtracking into the query succeeded, but the new answer was too long for the buffer.

`int xsb_next(th_context *)` *C function*

This function is called after `xsb_query()` (which must have returned `XSB_SUCCESS`) to retrieve more answers. It rebinds the query variables in the term in register 1 and rebinds the argument fields of the `ret/n` answer term in register 2 to reflect the next answer to the query. Its return codes are as with `xsb_next_string()`.

When used in the multi-threaded engine, `xsb_next` *does not protect* the called thread from API calls from other pthreads until the query is finished, or even when the registers are being accessed. It is the user's responsibility to protect

the XSB thread, using a mutex or other concurrency control, through the time that registers are accessed by the calling program.

`int xsb_close_query(th_context *th)` *C function*

This function allows a user to close a query to `th` before all its answers have been retrieved. Since XSB is (usually) a tuple-at-a-time system, answers that are not retrieved are not computed so that closing a query may save time. If a given query Q is open, it is an error to open a new query without closing Q either by retrieving all its answers or explicitly calling `xsb_close_query()` to close Q. Calling `xsb_close_query()` when no query is open gives an error message, but otherwise has no effect.

Return Codes

- `XSB_SUCCESS` indicates that the current query was closed.
- `XSB_ERROR`
  - `permission_error` if `xsb_close_query()` is called while no query is open.

`int xsb_add_c_predicate(th_context *,char *,char *,int,int(*)())` *C function*

This function, called for example as:
`xsb_add_c_predicate(th, modname, predname, arity, cfunc)`
registers a C function (defined by the caller, here named `cfunc`) as a foreign function to be invoked by a Prolog predicate. The arguments are: the name of the module of the Prolog predicate being defined (NULL indicates usermod), the name of the Prolog predicate, the arity of the Prolog predicate, and the function pointer of the function defining the foreign routine. That function must get (and return) its arguments using the ctop (and ptoc) functions of the `cinterf` Foreign Language Interface (13). This `xsb_add_c_predicate` function always returns 0. After this function has been called, the predicate `modname:predname/arity` can be called in Prolog and will result in `cfunc` being invoked as a foreign language function.

The `th_context` argument is used only for the multi-threaded XSB engine.

14.3.4 Obtaining Information about Errors

`char * xsb_get_init_error_message()` *C function*

Used to find error messages if `xsb_init_string()` or `xsb_init()` returns `XSB_ERROR`.

Any errors returned by these functions have type `init_error`. Because initialization errors occur before XSB or any of its threads have been initialized, initialization errors do not require a thread context for input.

`char * xsb_get_error_type(th_context *th)` *C function*

If a function called for `th` returned `XSB_ERROR` this function provides a pointer to a string representing the type of the error. Types are as in Volume 1 *Exception Handling* with the addition of `init_error` for errors that occur during initialization of XSB, and `unrecoverable_error` for errors from which no recovery is possible for XSB (e.g. inability to allocate new memory).

`char *xsb_get_error_message(th_context *th)` *C function*

If a function called for `th` returned `XSB_ERROR` this function provides a pointer to a string representing a message associated with the error. For errors raised within the Prolog portion of execution, messages are as in Volume 1 *Exception Handling*.

14.3.5 Thread Management from Calling Programs

`int xsb_ccall_thread_create(th_context *callingThread, th_context **newThread)` *C function*

Causes `callingThread` to create a thread pointed to by `newThread`. `newThread` runs exactly the same interpreter loop as `callingThread` and all API functions will work on `newThread` just as on the main thread, or any other thread. `newThread` will be non-detached, and will inherit any private parameters from `callingThread`. To create a thread to do a specific task or a detached thread, rather than one that executes a command loop, simply call the query `thread_create/[2,3]` from one of the query functions.

`th_context *xsb_get_main_thread()` *C function*

Returns a pointer to the thread context of XSB's main thread. If XSB has not been initialized or has been closed this function returns 0.

`xsb_tid xsb_thread_id_to_context(th_context *th)` *C function*

`th_context *xsb_thread_context_to_id(xsb_tid id)` *C function*

14.4 The Variable-length String Data Type

XSB uses variable-length strings to communicate with certain C subroutines when the size of the output that needs to be passed from the Prolog side to the C side is not known. Variable-length strings adjust themselves depending on the size of the data they must hold and are ideal for this situation. For instance, as we have seem the two subroutines `xsb_query_string_string(query,buff,sep)` and `xsb_next_string(buff,sep)` use the variable string data type, `VarString`, for their second argument. To use this data type, make sure that

```
#include "cinterf.h"
```

appears at the top of the program file. Variables of the `VarString` type are declared using a macro that must appear in the declaration section of the program:

```
XSB_StrDefine(buf);
```

There is one important consideration concerning VarString with the *automatic* storage class: they must be *destroyed* on exit (see `XSB_StrDestroy`, below) from the procedure that defines them, or else there will be a memory leak. It is not necessary to destroy static `VarString`'s.

The public attributes of the type are `int length` and `char *string`. Thus, `buf.string` represents the actual contents of the buffer and `buf.length` is the length of that data. Although the length and the contents of a `VarString` string is readily accessible, the user **must not** modify these items directly. Instead, he should use the macros provided for that purpose:

- `XSB_StrSet(VarString *vstr, char *str)`: Assign the value of the regular null-terminated C string to the `VarString vstr`. The size of `vstr` is adjusted automatically.
- `XSB_StrSetV(VarString *vstr1, VarString *vstr2)`: Like `XSB_StrSet`, but the second argument is a variable-length string, not a regular C string.
- `XSB_StrAppend(VarString *vstr, char *str)`: Append the null-terminated string `str` to the `VarString vstr`. The size of `vstr` is adjusted.
- `XSB_StrPrepend(VarString *vstr, char *str)`: Like `XSB_StrAppend`, except that `str` is prepended.

- `XSB_StrAppendV(VarString *vstr1, VarString *vstr2)`: Like `XSB_StrAppend`, except that the second string is also a `VarString`.
- `XSB_StrPrependV(VarString *vstr1, VarString *vstr2)`: Like `XSB_StrAppendV`, except that the second string is prepended.
- `XSB_StrCompare(VarString *vstr1, VarString *vstr2)`: Compares two `VarString`. If the first one is lexicographically larger, then the result is positive; if the first string is smaller, than the result is negative; if the two strings have the same content (*i.e.*, `vstr1->string` equals `vstr2->string` then the result is zero.
- `XSB_StrCmp(VarString *vstr, char *str)`: Like `XSB_StrCompare` but the second argument is a regular, null-terminated string.
- `XSB_StrAppendBlk(VarString *vstr, char *blk, int size)`: This is like `XSB_StrAppend`, but the second argument is not assumed to be null-terminated. Instead, `size` characters pointed to by `blk` are appended to `vstr`. The size of `vstr` is adjusted, but the content is *not* null terminated.
- `XSB_StrPrependBlk(VarString *vstr, char *blk, int size)`: Like `XSB_StrPrepend`, but `blk` is not assumed to point to a null-terminated string. Instead, `size` characters from the region pointed to by `blk` are prepended to `vstr`.
- `XSB_StrNullTerminate(VarString *vstr)`: Null-terminates the `VarString` string `vstr`. This is used in conjunction with `XSB_StrAppendBlk`, because the latter does not null-terminate variable-length strings.
- `XSB_StrEnsureSize(VarString *vstr, int minsize)`: Ensure that the string has room for at least `minsize` bytes. This is a low-level routine, which is used to interface to procedures that do not use `VarString` internally. If the string is larger than `minsize`, the size might actually shrink to the nearest increment that is larger `minsize`.
- `XSB_StrShrink(VarString *vstr, int increment)`: Shrink the size of `vstr` to the minimum necessary to hold the data. `increment` becomes the new increment by which `vstr` is adjusted. Since `VarString` is automatically shrunk by `XSB_StrSet`, it is rarely necessary to shrink a `VarString` explicitly. However, one might want to change the adjustment increment using this macro (the default increment is 128).
- `XSB_StrDestroy(VarString *vstr)`: Destroys a `VarString`. Explicit destruction is necessary for `VarString`'s with the automatic storage class. Otherwise, memory leak is possible.

14.5 Passing Data into an XSB Module

The previous chapter described the low-level XSB/C interface that supports passing the data of arbitrary complexity between XSB and C. However, in cases when data needs to be passed into an executable XSB module by the main C program, the following higher-level interface should suffice. (This interface is actually implemented using macros that call the lower level functions.) These routines can be used to construct commands and queries into XSB 's register 1, which is necessary before calling `xsb_query()` or `xsb_command()`.

`void xsb_make_vars((int) N)` *C function*

`xsb_make_vars` creates a return structure of arity `N` in Register 2. So this routine may called before calling any of `xsb_query`, `xsb_query_string`, `xsb_command`, or `xsb_command_string` if parameters are to be set to be sent to the goal. It must be called before calling one of the `xsb_set_var_*` routines can be called. `N` must be the number of variables in the query that is to be evaluated.

`void xsb_set_var_int((int) Val, (int) N)` *C function*

`set_and_int` sets the N^{th} field in the return structure to the integer value `Val`. It is used to set the value of the N^{th} variable in a query before calling `xsb_query` or `xsb_query_string`. When called in XSB, the query will have the N^{th} variable set to this value.

`void xsb_set_var_string((char *) Val, (int) N)` *C function*

`set_and_string` sets the N^{th} field in the return structure to the atom with name `Val`. It is used to set the value of the N^{th} variable in a query before calling `xsb_query` or `xsb_query_string`. When called in XSB, the query will have the N^{th} variable set to this value.

`void xsb_set_var_float((float) Val, (int) N)` *C function*

`set_and_float` sets the N^{th} field in the return structure to the floating point number with value `Val`. It is used to set the value of the N^{th} variable in a query before calling `xsb_query` or `xsb_query_string`. When called in XSB, the query will have the N^{th} variable set to this value.

`prolog_int xsb_var_int((int) N)` *C function*

`xsb_var_int` is called after `xsb_query` or `xsb_query_string` returns an answer. It returns the value of the N^{th} variable in the query as set in the returned answer. This variable must have an integer value (which is cast to `long` in a 64-bit architecture).

`char* xsb_var_string((int) N)` *C function*

`xsb_var_string` is called after `xsb_query` or `xsb_query_string` returns an answer. It returns the value of the $\texttt{N}^{th}$ variable in the query as set in the returned answer. This variable must have an atom value.

`prolog_float xsb_var_float((int) N)` *C function*

`xsb_var_float` is called after `xsb_query` or `xsb_query_string` returns an answer. It returns the value of the $\texttt{N}^{th}$ variable in the query as set in the returned answer. This variable must have a floating point value (which is cast to `double` in a 64-bit architecture).

14.6 Creating an XSB Module that Can be Called from C

To create an executable that includes calls to the above C functions, these routines, and the XSB routines that they call, must be included in the link (`ld`) step.

Unix instructions: You must link your C program, which should include the main procedure, with the XSB object file located in

```
$XSBDIR/config/<your-system-architecture>/saved.o/xsb.o
```

Your program should include the file `cinterf.h` located in the `XSB/emu` subdirectory, which defines the routines described earlier, which you will need to use in order to talk to XSB. It is therefore recommended to compile your program with the option `-I$XSB_DIR/XSB/emu`.

The file `$XSB_DIR/config/your-system-architecture/modMakefile` is a makefile you can use to build your programs and link them with XSB. It is generated automatically and contains all the right settings for your architecture, but you will have to fill in the name of your program, etc.

It is also possible to compile and link your program with XSB using XSB itself as follows:

```
:- xsb_configuration(compiler_flags,CFLAGS),
        xsb_configuration(loader_flags,LDFLAGS),
        xsb_configuration(config_dir,CONFDIR),
        xsb_configuration(emudir,EMUDIR),
```

```
xsb_configuration(compiler,Compiler),
str_cat(CONFDIR, '/saved.o/', ObjDir),
write('Compiling myprog.c ... '),
shell([Compiler, ' -I', EMUDIR, ' -c ', CFLAGS, ' myprog.c ']),
shell([Compiler, CFLAGE, ' -o ', './myprog ',
       ObjDir, 'xsb.o ', ' myprog.o ', LDFLAGS]),
writeln(done).
```

This works for every architecture and is often more convenient than using the make files [6]. There are simple examples of C programs calling XSB in the `$XSB_DIR/examples/c_calling_XSB` directory, in files `cmain.c`, `ctest.P`, `cmain2.c`.

Windows instructions: To call XSB from C, you must build it as a DLL, which is done as follows:

```
cd $XSB_DIR\XSB\build
makexsb_wind DLL="yes"
```

The DLL, which you can call dynamically from your program is then found in

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

Since your program must include the file `cinterf.h`, it is recommended to compile it with the option `/I$XSB_DIR\XSB\emu`.

[6] The variable `CFLAGS` is needed in the linking stage in order to ensure that the appropriate memory option is passed if XSB is configured `-with-bits32` or `-with-bits64` to override the default on a 64-bit platform.

Chapter 15

Library Utilities

In this chapter we introduce libraries of some useful predicates that are supplied with XSB. Interfaces and more elaborate packages are documented in later chapters. These predicates are available only when imported them from (or explicitly consult) the corresponding modules.

15.1 List Processing

The XSB library contains various list utilities, some of which are listed below. These predicates should be explicitly imported from the module specified after the skeletal specification of each predicate. There are a lot more useful list processing predicates in various modules of the XSB system, and the interested user can find them by looking at the sources.

`append(?List1, ?List2, ?List3)` module: `basics`

Succeeds if list `List3` is the concatenation of lists `List1` and `List2`.

`member(?Element, ?List)` module: `basics`

Checks whether `Element` unifies with any element of list `List`, succeeding more than once if there are multiple such elements.

`memberchk(?Element, ?List)` module: `basics`

Similar to `member`/2, except that `memberchk`/2 is deterministic, i.e. does not succeed more than once for any call.

`ith(?Index, ?List, ?Element)` module: `basics`

Succeeds if the `Index`$^{\texttt{th}}$ element of the list `List` unifies with `Element`. Fails if

Index is not a positive integer or greater than the length of List. Either Index and List, or List and Element, should be instantiated (but not necessarily ground) at the time of the call.

delete_ith(+Index, +List, ?Element, ?RestList) module: listutil
Succeeds if the Indexth element of the list List unifies with Element, and RestList is List with Element removed. Fails if Index is not a positive integer or greater than the length of List.

log_ith(?Index, ?Tree, ?Element) module: basics
Succeeds if the Indexth element of the Tree Tree unifies with Element. Fails if Index is not a positive integer or greater than the number of elements that can be in Tree. Either Index and Tree, or Tree and Element, should be instantiated (but not necessarily ground) at the time of the call. Tree is a list of full binary trees, the first being of depth 0, and each one being of depth one greater than its predecessor. So log_ith/3 is very similar to ith/3 except it uses a tree instead of a list to obtain log-time access to its elements.

log_ith_bound(?Index, ?Tree, ?Element) module: basics
is like log_ith/3, but only if the Indexth element of Tree is non-variable and equal to Element. This predicate can be used in both directions, and is most useful with Index unbound, since it will then bind Index and Element for each non-variable element in Tree (in time proportional to $N * logN$, for N the number of non-variable entries in Tree.)

log_ith_new(?Index, ?Tree, ?Element) module: basics
binds Element to the "end" of the log_list Tree, and unifies Index with its corresponding index. The "end" of a log_list is the first element after the one with the largest index that has been added. This can be used to simulate adding an element to the end of an open-tailed list, but with better complexity.

log_ith_to_list(?Tree, ?List) module: basics
constructs the List that contains all bound values in the log_list Tree, in the order they appear in Tree.

length(?List, ?Length) module: basics
Succeeds if the length of the list List is Length. This predicate is deterministic if List is instantiated to a list of definite length, but is nondeterministic if List is a variable or has a variable tail. If List is uninstantiated, it is unified with a list of length Length that contains variables.

`same_length(?List1, ?List2)` module: `basics`

Succeeds if list `List1` and `List2` are both lists of the same number of elements. No relation between the types or values of their elements is implied. This predicate may be used to generate either list (containing variables as elements) given the other, or to generate two lists of the same length, in which case the arguments will be bound to lists of length $0, 1, 2, \ldots$.

`select(?Element, ?L1, ?L2)` module: `basics`

`List2` derives from `List1` by selecting (removing) an `Element` non-deterministically.

`reverse(+List, ?ReversedList)` module: `basics`

Succeeds if `ReversedList` is the reverse of list `List`. If `List` is not a proper list, `reverse/2` can succeed arbitrarily many times. It works only one way.

`perm(+List, ?Perm)` module: `basics`

Succeeds when `List` and `Perm` are permutations of each other. The main use of `perm/2` is to generate permutations of a given list. `List` must be a proper list. `Perm` may be partly instantiated.

`subseq(?Sequence, ?SubSequence, ?Complement)` module: `basics`

Succeeds when `SubSequence` and `Complement` are both subsequences of the list `Sequence` (the order of corresponding elements being preserved) and every element of `Sequence` which is not in `SubSequence` is in the `Complement` and vice versa. That is,

$$length(\texttt{Sequence}) = length(\texttt{SubSequence}) + length(\texttt{Complement})$$

for example, `subseq([1,2,3,4], [1,3], [2,4])`. The main use of `subseq/3` is to generate subsets and their complements together, but can also be used to interleave two lists in all possible ways.

`merge(+List1, +List2, ?List3)` module: `listutil`

Succeeds if `List3` is the list resulting from "merging" lists `List1` and `List2`, i.e. the elements of `List1` together with any element of `List2` not occurring in `List1`. If `List1` or `List2` contain duplicates, `List3` may also contain duplicates.

`absmerge(+List1, +List2, ?List3)` module: `listutil`

Predicate `absmerge/3` is similar to `merge/3`, except that it uses predicate `absmember/2` described below rather than `member/2`.

`absmember(+Element, +List)` module: `listutil`

Similar to `member`/2, except that it checks for identity (through the use of predicate `'=='/2`) rather than unifiability (through `'='/2`) of `Element` with elements of `List`.

`member2(?Element, ?List)` module: `listutil`

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

`closetail(?List)` module: `listutil`

Predicate `closetail/1` closes the tail of an open-ended list. It succeeds only once.

15.1.1 Processing Comma Lists

It is often useful to process comma lists when meta-interpreting or preprocessing. XSB libraries include the following simple utilities.

`comma_to_list(+CommaList,-List)` module: `basics`

Transforms `CommaList` to `List`.

`comma_append(?CL1,?CL2,?CL3)` module: `basics`
`comma_length(?CommaList,?Length)` module: `basics`
`comma_member(?Element,?CommaList)` module: `basics`
`comma_memberchk(?Element,?CommaList)` module: `basics`

Analogues for comma lists of `append/3`, `length/3`, `member/2` and `memberchk/2`, respectively.

15.2 Attributed Variables

Attributed variables are a special data type that associates variables with arbitrary attributes as well as supports extensible unification. Attributed variables have proven to be a flexible and powerful mechanism to extend a classic logic programming system with the ability of constraint solving. Our low-level API for constraints closely resembles that of hProlog [23] and SWI [95].

15.2.1 Low-level Interface

Attributes of variables are pairs of attribute module names and values. An attribute module name can be any atom. A value can be any XSB value (term, variable, atom, ...). Any variable has at most one attribute for a particular attribute module. Attribute modules are distinct from XSB modules: although it is most efficient to keep each handlers for each attribute module in their own XSB module. c Attributes can be manipulated with the following three predicates (`get_attr/3`, `put_attr/3` and `del_attr/2`) defined in the module `machine`.

`get_attr(-Var,+Mod, ?Val)` module: `machine`

Gets the value of the attribute of `Var` in attribute module `Mod`. Non-variable terms in `Var` cause a type error. `Val` will be unified with the value of the attribute, if it exists. Otherwise the predicate fails.

`put_attr(-Var,+Mod, ?Val)` module: `machine`

Sets the value of the attribute of `Var` in attribute module `Mod`. Non-variable terms in `Var` cause a type error. The previous value of the attribute is overwritten, if it exists.

`del_attr(-Var, +Mod)` module: `machine`

Removes the attribute of `Var` in attribute module `Mod`. Non-variable terms in `Var` cause a type error. The previous value of the attribute is removed, if it exists.

One has to extend the default unification algorithm for used attributes by installing a handler in the following way:

`:- install_verify_attribute_handler(`$+Mod, -AttrValue, -Target, +Handler, +WarningFlag$`)`
`:- install_verify_attribute_handler(`$+Mod, -AttrValue, -Target, +Handler$`)`

The predicates `install_verify_attribute_handler/5` and `install_verify_attribute_handler/4` are defined in module `machine`. Mod is the attribute `Module` and $Handler$ is a term with arguments $AttrValue$ and $Target$. The $Handler$ term has to correspond to a handler predicate that takes the value of the attribute ($AttrValue$) and the term that the attributed value is bound to ($Target$) as arguments. The argument $WarningFlag$ in the 5-argument version of the predicate can be used to suppress the warning issued when replacing the `verify_attribute_handler` for a module. If the argument is `warning_on` then the warning is issued if a handler for the module already exists. Otherwise, the warning is suppressed. The 4-argument version of the predicate does *not* suppress the warning.

To get good efficiency, it is usually best to keep the handlers for each attribute module in separate XSB modules. The handler is called after the unification of an attributed variable with a term or other attributed variable, if the attributed variable has an attribute in the corresponding module. The two arguments of the unification are already bound at the time the handler is called, i.e. the handler is a post-unify handler.

Here, by giving the implementation of a simple finite domain constraint solver (see the file `fd.P` below), we show how these low-level predicates for attributed variables can be used. In this example, an attribute in the module `fd` is used and the value of this attribute is a list of terms.

```
%% File: fd.P
%%
%% A simple finite domain constrait solver implemented using the low-level
%% attributes variables interface.

:- import put_attr/3, get_attr/3, del_attr/2,
   install_verify_attribute_handler/4 from machine.
:- import member/2 from basics.

:- install_verify_attribute_handler(fd,AttrValue,Target,fd_handler(AttrValue,Target)).

fd_handler(Da, Target) :-
        (var(Target),                       % Target is an attributed variable
         get_attr(Target, fd, Db) ->        % has a domain
           intersection(Da, Db, [E|Es]),    % intersection not empty
           (Es = [] ->                      % exactly one element
              Target = E                    % bind Var (and Value) to E
           ;  put_attr(Target, fd, [E|Es])  % update Var's (and Value's)
           )
        ;  member(Target, Da)               % is Target a member of Da?
        ).

intersection([], _, []).
intersection([H|T], L2, [H|L3]) :-
        member(H, L2), !,
        intersection(T, L2, L3).
intersection([_|T], L2, L3) :-
        intersection(T, L2, L3).

domain(X, Dom) :-
```

```
        var(Dom), !,
        get_attr(X, fd, Dom).
domain(X, List) :-
        List = [El|Els],                    % at least one element
        (Els = []                           % exactly one element
         -> X = El                          % implied binding
        ;  put_attr(Fresh, fd, List),       % create a new attributed variable
           X = Fresh                        % may call verify_attributes/2
        ).

show_domain(X) :-                           % print out the domain of X
        var(X),                             % X must be a variable
        get_attr(X, fd, D),
        write('Domain of '), write(X),
        write(' is '), writeln(D).
```

When writing or porting a constraint package, it is usually useful to adjust the way that correct answer substitutions are shown in the command line. This can be controlled using the following two predicates:

`install_attribute_portray_hook(Module,Attribute,Handler)` module: `machine`

This hook is called by the command-line interpreter when printing out the value of each variable in a top-level query. When a printing out an attributed variable, any appropriate handlers are called to portray the constraints represented by the attribute. As an example, the `bounds` package (cf. Volume II: Constraints Packages) uses a hook to print out the bounds of variables:

```
| ?- X in 1..10,Y in 1..10,X + 4 #< Y -3.

X = _h629 { bounds : 1 .. 2 }
Y = _h673 { bounds : 9 .. 10 }
```

Writing a handler can be as simple as possible or as elaborate as desired. In the case of `bounds` the handler is simple:

```
bounds_attr_portray_hook(bounds(L,U,_)) :- write(L..U).
```

The hook is installed when the constraint package is loaded by placing in the package loader directive such as:

```
:- install_attribute_portray_hook(bounds,Attr,bounds_attr_portray_hook(Attr)).
```

Note that the hook will be indexed on the module associated with the attribute (in this case `bounds`). XSB's command-line interpreter will unify the second argument of the portray hook with the attribute, and then call `Handler`.

`install_attribute_constraint_hook(Module,Vars,Names,Handler)` module: `machine`

For some constraint packages, it may not be particularly useful to associate constraints with variables: instead, the projection of global constraints onto the variables of the top-level query may be more useful. This is the case in the CLP(R) package (cf. Volume II Constraints Packages), where the command-line interaction may look as follows:

```
| ?- {X = 2*Y,Y >= 7},inf(X,F).
 { X >= 14.0000 }
 { Y = 0.5000 * X }

X = _h8841
Y = _h9506
F = 14.0000
```

In XSB, the (projection of the) global constraints in CLP(R) are displayed by the following routines:

```
clpr_portray_varlist(Vars,Names):-
filter_varlist(Vars,Names,V1,N1),
dump(V1,N1,Constraints),
member(C,Constraints),
console_write(' { '), console_write(C),console_writeln(' } '),
fail.
clpr_portray_varlist(_V,_N).

filter_varlist([],[],[],[]).
filter_varlist([V1|R1],[N1|R2],[V1|R3],[N1|R4]):-
var(V1),!,
filter_varlist(R1,R2,R3,R4).
filter_varlist([_V1|R1],[_N1|R2],R3,R4):-
filter_varlist(R1,R2,R3,R4).
```

This predicate sets up a call to the CLP(R) library predicate `dump/3`, whose constraints it then writes out to the console. Analogous to the portray hook, the console hook is installed using the directive:

```
:- install_constraint_portray_hook(clpr,Vars,Names,clpr_portray_varlist(Vars,Names)).
```

If the `clpr` module is loaded, the command line interpreter checks any constraint portray hooks upon the first success of a top-level goal. It then unifies the second argument `Vars` with the variables of the goal, and `Names` with the names of the variables of the goal which are then passed on to `Handler`

15.3 `constraintLib`: a library for CLP

XSB supports constraint logic programming through its engine-level support of attributed variables (Section 15.2), and its support for constraint handling rules (CHR) (cf. Volume II: Constraint Handling Rules). The `constraintLib` library includes routines for delaying and examining bindings that are commonly used to implement CHR and other constraint libraries.

When processing constraints, it is often useful to delay a goal based on the instantiation level of a term or set of terms. For instance a `3 > X + Y` should be delayed until both `X` and `Y` are instantiated. However the goal should be reinvoked as soon as possible after both are instantiated in order to prune search paths that may not be useful to pursue. The predicate `when/2` provides a useful mechanism to delay goals based on instantiation patterns [1].

`when(+Condition,Goal)` module: `constraintLib`

Delays the execution of `Goal` until `Condition` is satisfied, whereupon `Goal` will be executed. Condition can have the form

- `?=(Term1,Term2)`
- `nonvar(Term)`
- `ground(Term)` [2]
- `(Condition,Condition)`
- `(Condition ; Condition)`

Example: The following session illustrates the use of when/2 to delay a goal.

```
|?- when(nonvar(X),writeln(test(1-2,nonvar))),writeln(test(1,nonvar)),X = f(_Y).
```

[1] Despite the similar name, this method of delaying is conceptually different from SLG DELAYING discussed in Volume 1 of this manual, which is used for resolving cycles of dependencies in computing the well-founded semantics, and is not based on the state of instantiation of a term.

[2] To use `ground/1` in the condition, it must be imported into the file where it is used.

```
test(1,nonvar)
test(1 - 2,nonvar)

X = f(_h245)
```

`unifiable(X, Y, -Unifier)` module: `constraintLib`

If `X` and `Y` can unify, succeeds unifying `Unifier` with a list of terms of the form `Var = Value` representing a most general unifier of `X` and `Y`. `unifiable/3` can handle cyclic terms. Attributed variables are handled as normal variables. Associated hooks are not executed [3].

`setarg(+Index,+Term,+Value)` module: `constraintLib`

`set_arg(+Index,+Term,+Value)` module: `machine`

The predicate `setarg/3` provides an efficient but non-logical way to update argument `Index` of a Prolog term `Term` to `Value` via destructive assignment and without the necessity of copying `Term`. `setarg/3` should be used sparingly, to ensure both clarity and portability of code.

Example

```
|?- X = p(f(1),g(2),r([a])),
    writeln(zero(X)),
    (  setarg(2,X,g([b])),
       writeln(one(X)),
       fail
     ; writeln(two(X))).
zero(p(f(1),g(2),r([a])))
one(p(f(1),g([b]),r([a])))
two(p(f(1),g(2),r([a])))

X = p(f(1),g(2),r([a]))
```

Error Cases

- `Index` is a variable
  - `instantiation_error`
- `Index` neither a variable nor an integer
  - `type_error(integer,Index)`
- `Index` is less than 0

[3] In Version 3.8, `unifiable/3` is implemented as a Prolog predicate and so is slower than many of the predicates in this section.

  - `domain_error(not_less_than_zero,Index)`
- `Term` is a variable
  - `instantiation_error`
- `Term` neither a variable nor a compound term
  - `type_error(compound,Term)`

`term_variables(+Term,-Variables)` module: `constraintLib`

Given any Prolog term `Term` as input, returns a sorted list of variables in the term.

15.4 Formatted Output

`format(+String,+Control)` module: `format`
`format(+Stream,+String,+Control)` module: `format`

`format/2` and `format/3` act as a Prolog analog to the C `stdio` function `printf()`, allowing formatted output [4].

Output is formatted according to `String` which can contain either a format control sequence, or any other character which will appear verbatim in the output. Control sequences act as place-holders for the actual terms that will be output. Thus

```
?- format("Hello ~q!",world).
```

will print `Hello world!`.

If there is only one control sequence, the corresponding element may be supplied alone in `Control`. If there are more, `Control` must be a list of these elements. If there are none then `Control` must be an empty list. There have to be as many elements in `Control` as control sequences in `String`.

The character `~` introduces a control sequence. To print a `~` just repeat it:

```
?- format("Hello ~~world!", []).
```

[4] The `format` family of predicates is due to Quintus Prolog, by way of Ciao.

will output `Hello ~world!`.

The general format of a control sequence is `~NC`. The character `C` determines the type of the control sequence. `N` is an optional numeric argument. An alternative form of `N` is `*`. `*` implies that the next argument in `Arguments` should be used as a numeric argument in the control sequence. For example:

```
?- format("Hello~4cworld!", [0'x]).
```

and

```
?- format("Hello~*cworld!", [4,0'x]).
```

both produce

```
Helloxxxxworld!
```

The following control sequences are available in XSB.

- `~a` The argument is an atom. The atom is printed without quoting.
- `~Nc` (Print character.) The argument is a number that will be interpreted as an UTF-8 code. `N` defaults to one and is interpreted as the number of times to print the character.
- `~f` (Print float). The argument is a float. The float will be printed out by XSB.
- `~d` (Print integer). The argument is an integer, and will be printed out by XSB.
- `~Ns` (Print string.) The argument is a list of UTF-8 codes. Exactly `N` characters will be printed. `N` defaults to the length of the string. Example:

  ```
  ?- format("Hello ~4s ~4s!", ["new","world"]).
  ?- format("Hello ~s world!", ["new"]).
  ```

  will print as

  ```
  Hello new worl!
  Hello new world!
  ```

  respectively.

- `~i` (Ignore argument.) The argument may be of any type. The argument will be ignored. Example:

  ```
  ?- format("Hello ~i~s world!", ["old","new"]).
  ```

  will print as

  ```
  Hello new world!
  ```

- `~k` (Print canonical.) The argument may be of any type. The argument will be passed to `write_canonical/2`). Example:

  ```
  ?- format("Hello ~k world!", a+b+c).
  ```

  will print as

  ```
  Hello +(+(a,b),c) world!
  ```

- `~q` (Print quoted.) The argument may be of any type. The argument will be passed to `writeq/2`. Example:

  ```
  ?- format("Hello ~q world!", [['A','B']]).
  ```

  will print as

  ```
  Hello ['A','B'] world!
  ```

- `~w` (write.) The argument may be of any type. The argument will be passed to `write/2`. Example:

  ```
  ?- format("Hello ~w world!", [['A','B']]).
  ```

  will print as

  ```
  Hello [A,B] world!
  ```

- `~Nn` (Print newline.) Print `N` newlines. `N` defaults to 1. Example:

  ```
  ?- format("Hello ~n world!", []).
  ```

  will print as

  ```
  Hello
   world!
  ```

15.5 Low-level Atom Manipulation Predicates

XSB has a number of low-level predicates that, despite their names, examine properties of atoms. The functionality of these predicates is often a subset of the ISO predicate `sub_atom/5`, but these predicates are faster, as they are more specialized, and have been written in C.

These predicates are especially powerful when they are combined with pattern-matching facilities provided by the `pcre` package described in Volume 2 of this manual).

It is important to note, that not all string manipulation predicates have been made thread-safe in Version 3.8. In addition, as noted, the predicates may or may not properly handle (non-ASCII) UTF-8 characters.

`str_sub(+Sub, +Str, ?Pos)` module: `string`
`str_sub(+Sub, +Str)` module: `string`

Succeeds if `Sub` is a substring of `Str`. In that case, `Pos` unifies with the position where the match occurred. Positions start from 0. `str_sub/2` is also available, which is equivalent to having `_` in the third argument of `str_sub/3` [5].

`str_match(+Sub, +Str, +Direction, ?Beg, ?End)` module: `string`

This is an enhanced version of the previous predicate. `Direction` can be `forward` or `reverse` (or any abbreviation of these). If `forward`, the predicate finds the first match of `Sub` from the beginning of `Str`. If `reverse`, it finds the first match from the end of the string (*i.e.*, the last match of `Sub` from the beginning of `Str`). `Beg` and `End` must be integers or unbound variables. (It is possible that one is bound and another is not.) `Beg` unifies with the offset of the first character where `Sub` matched, and `End` unifies with the offset of the next character to the right of `Sub` (such a character might not exist, but the offset is still defined). Offsets start from 0.

Both `Beg` and `End` can be bound to negative integers. In this case, the value represents the offset from the *second* character past the end of `Str`. Thus `-1` represents the character next to the end of `Str` and can be used to check where the end of `Sub` matches in `Str`. In the following examples

[5] Currently, `str_sub/2` works properly for UTF-8 characters, but `str_sub/3` does not.

```
?- string_match(Sub,Str,forw,X,-1).
?- string_match(Sub,Str,rev,X,-1).
?- string_match(Sub,Str,forw,0,X).
```

the first checks if the *first* match of `Sub` from the beginning of `Str` is a suffix of `Str` (because `End` represents the character next to the last character in `Sub`, so `End=-1` means that the last characters of `Sub` and of `Str` occupy the same position). If so, `X` is bound to the offset (from the end of `Str`) of the first character of `Sub`. The second example checks if the *last* match of `Sub` in `Str` is a suffix of `Str` and binds `X` to the offset of the beginning of that match (counted from the beginning of `Str`). The last example checks if the first match of `Sub` is a prefix of `Str`. If so, `X` is bound to the offset (from the beginning of `Str`) of the last character of `Sub` [6].

`substring(+String, +BeginOffset, +EndOffset, -Result)` module: `string`

`String` can be an atom or a list of characters, and the offsets must be integers. If `EndOffset` is negative, endof(`String`)+`EndOffset`+1 is assumed. Thus, `-1` means end of string. If `BeginOffset` is less than 0, then 0 is assumed; if it is greater than the length of the string, then string end is assumed. If `EndOffset` is non-negative, but is less than `BeginOffset`, then empty string is returned.

Offsets start from 0.

The result returned in the fourth argument is a string, if `String` is an atom, or a list of characters, if so is `String`.

The `substring/4` predicate always succeeds (unless there is an error, such as wrong argument type).

Here are some examples:

```
| ?- substring('abcdefg', 3, 5, L).

L = de

| ?- substring("abcdefg", 4, -1, L).

L = [101,102]
```

(*i.e.*, L = `ef` represented using ASCII codes) [7].

[6] Currently, `string_match/5` does not work properly for UTF-8 characters.

[7] Currently, `substring/4` works properly for UTF-8 characters.

15.6 Script Writing Utilities

Prolog, (in particular XSB!) can be useful for writing scripts. Prolog's simple syntax and declarative semantics make it especially suitable for scripts that involve text processing. There are several ways to access script-writing commands from XSB. The first is to execute the command via the predicates `shell/1` or `shell/2`. These predicates can execute any command but they do not provide streamability across UNIX and Windows commands, and they do not return any output of commands to Prolog. Special predicates are provided to handle cross-platform compatibility and to bring output into XSB.

Effort has been made to make the these thread-safe; however in Version 3.8, calls to the XSB script writing utilities go through a single mutex, and may cause contention if many threads seek to concurrently use sockets.

`expand_filename(+FileName,-ExpandedName)` module: `machine`

Expands the file name passed as the first argument and binds the variable in the second argument to the expanded name. This includes (1) expanding Unix tildes, (2) prepending `FileName` to the current directory, and (3) "rectifying" the expanded file name. In rectification, the expanded file name is "rectified" so that multiple repeated slashes are replaced with a single slash, the intervening "./" are removed, and "../" are applied so that the preceding item in the path name is deleted. For instance, if the current directory is `/home`, then `abc//cde/..///ff/./b` will be converted into `/home/abc/ff/b`.

Under Windows, this predicates does rectification as described above, (using backslashes when appropriate), but it does not expand the tildes.

`expand_filename_no_prepend(+FileName,-ExpandedName)` module: `shell`

This predicate behaves as `expand_filename/2`, but only expands tildes and does rectification. It does not prepend the current working directory to relative file names.

`parse_filename(+FileName,-Dir,-Base,-Extension)` module: `machine`

This predicate parses file names by separating the directory part, the base name part, and file extension. If file extension is found, it is removed from the base name. Also, directory names are rectified and if a directory name starts with a tilde (in Unix), then it is expanded. Directory names always end with a slash or a backslash, as appropriate for the OS at hand.

For instance, ~`john///doe/dir1//../foo.bar` will be parsed into: `/home/john/doe/`, `foo`, and `bar` (where we assume that `/home/john` is what ~`john` expands into).

`sys_pid(-Pid)` module: `shell`

Get Id of the current process.

`sys_main_memory(-RamInBytes)` module: `shell`

Provides a platform-independent way to return the amount of RAM for the current machine, in bytes. [8]

15.6.1 Communication with Subprocesses

In the previous section, we have seen several predicates that allow XSB to create other processes. However, these predicates offer only a very limited way to communicate with these processes. The predicate `spawn_process/5` and friends come to the rescue. It allows a user to spawn any process (including multiple copies of XSB) and redirect its standard input and output to XSB streams. XSB can then write to the process and read from it. The section of socket I/O describes yet another mode of interprocess communication.

In addition, the predicate `pipe_open/2` described in this section lets one create any number of pipes (that do not need to be connected to the standard I/O stream) and talk to child processes through these pipes. All predicates in this section, except `pipe_open/2` and `fd2stream/2`, must be imported from module `shell`. The predicates `pipe_open/2` and `fd2stream/2` must be imported from `file_io`.

`spawn_process(+CmdSpec,-StreamToProc,-StreamFromProc,-ProcStderrStream,-ProcId)` module: `shell`

Spawn a new process specified by `CmdSpec`. `CmdSpec` must be either a single atom or a *list* of atoms. If it is an atom, then it must represent a shell command. If it is a list, the first member of the list must be the name of the program to run and the other elements must be arguments to the program. Program name must be specified in such a way as to make sure the OS can find it using the contents of the environment variable `PATH`. Also note that pipes, I/O redirection and such are not allowed in command specification. That is, `CmdSpec` must represent a single command. (But read about process plumbing below and about the related predicate `shell/5`.)

The next three parameters of `spawn_process` are XSB I/O stream identifiers for the process (leading to the subprocess standard input), from the process (from its standard output), and a stream capturing the subprocess standard error output. The last parameter is the system process id.

[8] Based on code by David Robert Nadeau under the Creative Commons license.

Here is a simple example of how it works.

```
| ?- import file_flush/2, file_read_line_atom/2 from file_io.
| ?- import file_nl/1 , file_write/2 from xsb_writ.

| ?- spawn_process([cat, '-'], To, From, Stderr, Pid),
     writeln(To,'Hello cat!'), flush_output(To,_), file_read_line_atom(From,Y).

To = 3
From = 4
Stderr = 5
Pid = 14328
Y = Hello cat!

yes
```

Here we created a new process, which runs the "`cat`" program with argument "–". This forces `cat` to read from standard input and write to standard output. The next line writes an atom and newline to the XSB stream `To`, which is bound to the standard input of the `cat` process (proc id `14328`). The `cat` process then copies the input to its standard output. Since standard output of the cat process is redirected to the XSB stream `From` in the parent process, the last line in our program is able to read it and return in the variable `Y`. Note that in the second line we used `flush_output/2`. Flushing the output is extremely important here, because XSB I/O pipe (file) streams are buffered. Thus, `cat` might not see its input until the buffer is filled up, so the above clause might hang. `flush_output/2` makes sure that the input is immediately available to the subprocess.

In addition to the above general schema, the user can tell `spawn_process/5` not to open one of the communication streams or to use one of the existing communication streams. This is useful when you do not expect to write or read to/from the subprocess or when one process wants to write to another (see the process plumbing example below). To tell that a certain stream is not needed, it suffices to bind that stream to an atom. For instance,

```
| ?- spawn_process([cat, '-'], To, none, none, _),
     nl(To), writeln(To,'Hello cat!'), flush_output(To).


To = 3,
Hello cat!
```

reads from XSB and copies the result to standard output. Likewise,

```
| ?- spawn_process('cat library.tex', none, From, none, _),
     file_read_line_atom(From, S).

From = 4
S = \chapter{Library Utilities} \label{library_utilities}
```

In each case, only one of the streams is open. (Note that the shell command is specified as an atom rather than a list.) Finally, if both streams are suppressed, then `spawn_process` reduces to the usual `shell/1` call (in fact, this is how `shell/1` is implemented):

```
| ?- spawn_process([pwd], none, none).

/usr/local/foo/bar
```

On the other hand, if any one of the three stream variables in `spawn_process` is bound to an already existing file stream, then the subprocess will use that stream (see the process plumbing example below).

One of the uses of XSB subprocesses is to create XSB servers that spawn subprocesses and control them. A spawned subprocess can be another XSB process. The following example shows one XSB process spawning another, sending it a goal to evaluate and obtaining the result:

```
| ?- spawn_process([xsb], To, From,Err,_),
     write(To,'assert(p(1)).'), flush_output(To,_),
     write(To,'p(X), writeln(X).'), flush_output(To,_),
     file_read_line_atom(From,XX).

XX = 1

yes
| ?-
```

Here the parent XSB process sends "`assert(p(1)).`" and then "`p(X), writeln(X).`" to the spawned XSB subprocess. The latter evaluates the goal and prints (via "`writeln(X)`") to its standard output. The main process reads it through the `From` stream and binds the variable `XX` to that output.

Finally, we should note that the stream variables in the `spawn_process` predicate can be used to do process plumbing, *i.e.*, redirect output of one subprocess into the input of another. Here is an example:

```
| ?- open(test,write,Stream),
     spawn_process([cat, 'data'], none, FromCat1, none, _),
     spawn_process([sort], FromCat1,Stream, none, _).
```

Here, we first open file `test`. Then `cat data` is spawned. This process has the input and standard error stream blocked (as indicated by the atom `none`), and its output goes into stream `FromCat1`. Then we spawn another process, `sort`, which picks the output from the first process (since it uses the stream `FromCat1` as its input) and sends its own output (the sorted version of `data`) to its output stream `Stream`. However, `Stream` has already been open for output into the file `test`. Thus, the overall result of the above clause is tantamount to the following shell command:

```
cat data | sort > test
```

***Important notes about spawned processes*:**

1. Asynchronous processes spawned by XSB do not disappear (at least on Unix) when they terminate, *unless* the XSB program executes a *wait* on them (see `process_control` below). Instead, such processes become defunct *zombies* (in Unix terminology); they do not do anything, but consume resources (such as file descriptors). So, when a subprocess is known to terminate, it must be waited on.

2. The XSB parent process must know how to terminate the asynchronous subprocesses it spawns. The drastic way is to kill it (see `process_control` below). Sometimes a subprocess might terminate by itself (*e.g.*, having finished reading a file). In other cases, the parent and the child programs must agree on a protocol by which the parent can tell the child to exit. The programs in the XSB subdirectory `examples/subprocess` illustrate this idea. If the child subprocess is another XSB process, then it can be terminated by sending the atom `end_of_file` or `halt` to the standard input of the child. (For this to work, the child XSB must waiting at the prompt).

3. It is very important to not forget to close the streams that the parent uses to communicate with the child. These are the streams that are provided in arguments 2,3,4 of `spawn_process`. The reason is that the child might terminate,

but these streams to the standard input of the child will remain open, since they belong to the parent process. As a result, the parent will own defunct I/O streams and might eventually run out of file descriptors or streams.

`process_status(+Pid,-Status)` module: `shell`

This predicate always succeeds. Given a process id, it binds the second argument (which must be an unbound variable) to one of the following atoms: `running`, `stopped`, `exited_normally`, `exited_abnormally`, `aborted`, `invalid`, and `unknown`. The `invalid` status is given to processes that never existed or that are not children of the parent XSB process. The `unknown` status is assigned when none of the other statuses can be assigned.

Note: process status (other than `running`) is system dependent. Windows does not seem to support `stopped` and `aborted`. Also, processes killed using the `process_control` predicate (described next) are often marked as `invalid` rather than `exited`, because Windows seems to lose all information about such processes. Process status might be inaccurate in some Unix systems as well, if the process has terminated and `wait()` has been executed on that process.

`process_control(+Pid,+Operation)` module: `shell`

Perform a process control `operation` on the process with the given `Pid`. Currently, the only supported operations are `kill` (an atom) and `wait(Code)` (a term). The former causes the process to exit unconditionally, and the latter waits for process completion. When the process exits, `Code` is bound to the process exit code. The code for normal termination is 0.

This predicate succeeds, if the operation was performed successfully. Otherwise, it fails. The `wait` operation fails if the process specified in `Pid` does not exist or is not a child of the parent XSB process.

The `kill` operation might fail, if the process to be killed does not exist or if the parent XSB process does not have the permission to terminate that process. Unix and Windows have different ideas as to what these permissions are. See *kill(2)* for Unix and *TerminateProcess* for Windows.

Note: under Windows, the programmer's manual warns of dire consequences if one kills a process that has DLLs attached to it.

`get_process_table(-ProcessList)` module: `shell`

This predicate is imported from module `shell`. It binds `ProcessList` to the list of terms, each describing one of the active XSB subprocesses (created via `spawn_process/5`). Each term has the form:

```
process(Pid,ToStream,FromStream,StderrStream,CommandLine).
```

The first argument in the term is the process id of the corresponding process, the next three arguments describe the three standard streams of the process, and the last is an atom that shows the command line used to invoke the process. This predicate always succeeds.

`shell(+CmdSpec,-StreamToProc, -StreamFromProc, -ProcStderr, -ErrorCode)`
module: `shell`

The arguments of this predicate are similar to those of `spawn_process`, except for the following: (1) The first argument is an atom or a list of atoms, like in `spawn_process`. However, if it is a list of atoms, then the resulting shell command is obtained by string concatenation. This is different from `spawn_process` where each member of the list must represent an argument to the program being invoked (and which must be the first member of that list). (2) The last argument is the error code returned by the shell command and not a process id. The code -1 and 127 mean that the shell command failed.

The `shell/5` predicate is similar to `spawn_process` in that it spawns another process and can capture that process' input and output streams. The important difference, however, is that XSB will wait until the process spawned by `shell/5` terminates. In contrast, the process spawned by `spawn_process` will run concurrently with XSB. In this latter case, XSB must explicitly synchronize with the spawned subprocess using the predicate `process_control/2` (using the `wait` operation), as described earlier.

The fact that XSB must wait until `shell/5` finishes has a very important implication: the amount of data the can be sent to and from the shell command is limited (1K is probably safe). This is because the shell command communicates with XSB via pipes, which have limited capacity. So, if the pipe is filled, XSB will hang waiting for `shell/5` to finish and `shell/5` will wait for XSB to consume data from the pipe. Thus, use `spawn_process/5` for any kind of significant data exchange between external processes and XSB.

Another difference between these two forms of spawning subprocesses is that `CmdSpec` in `shell/5` can represent *any* shell statement, including those that have pipes and I/O redirection. In contrast, `spawn_process` only allows command of the form "program args". For instance,

```
| ?- open(test,write,Stream),
     shell('cat | sort > data', Stream, none, none, ErrCode)
```

As seen from this example, the same rules for blocking I/O streams apply to `shell/5`. Finally, we should note that the already familiar standard predicates `shell/1` and `shell/2` (documented in Volume 1) are implemented using `shell/5`, and `shell/5` shares their error cases.

Notes:

1. With `shell/5`, you do not have to worry about terminating child processes: XSB waits until the child exits automatically. However, since communication pipes have limited capacity, this method can be used only for exchanging small amounts of information between parent and child.
2. The earlier remark about the need to close I/O streams to the child *does* apply.

`pipe_open(-ReadPipe, -WritePipe)` module: `shell`

Open a new pipe and return the read end and the write end of that pipe. If the operation fails, both `ReadPipe` and `WritePipe` are bound to negative numbers. The pipes returned by the `pipe_open/2` predicate are small integers that represent file descriptors used by the underlying OS. They are **not XSB I/O streams**, and they cannot be used for I/O directly. To use them, one must convert them to streams using `open/3` or `open/4`. [9]

The best way to illustrate how one can create a new pipe to a child (even if the child has been created earlier) is to show an example. Consider two programs, `parent.P` and `child.P`. The parent copy of XSB consults `parent.P`, which does the following: First, it creates a pipe and spawns a copy of XSB. Then it tells the child copy of XSB to assert the fact `pipe(RP)`, where `RP` is a number representing the read part of the pipe. Next, the parent XSB tells the child XSB to consult the program `child.P`. Finally, it sends the message `Hello!`.

The `child.P` program gets the pipe from predicate `pipe/1` (note that the parent tells the child XSB to first assert `pipe(RP)` and only then to consult the `child.P` file). After that, the child reads a message from the pipe and prints it to its standard output. Both programs are shown below:

```
%% parent.P
```

[9] XSB does not convert pipe file descriptors into I/O streams automatically. Because of the way XSB I/O streams are represented, they are not inherited by the child process and they do not make sense to the child process (especially if the child is not another XSB process). Therefore, we must pass the child processes an OS file descriptor instead. The child then converts these descriptor into XSB I/O streams.

```
:- import pipe_open/2 from file_io.
%% Create the pipe and pass it to the child process
?- pipe_open(RP,WP),
   %% WF is now the XSB I/O stream bound to the write part of the pipe
   open(pipe(WP),write,WF),
   %% ProcInput becomes the XSB stream leading directly to the child's stdin
   spawn_process(nxsb1, ProcInput, block, block, Process),
   %% Tell the child where the reading part of the pipe is
   fmt_write(ProcInput, "assert(pipe(%d)).\n", arg(RP)),
   fmt_write(ProcInput, "[child].\n", _),
   flush_output(ProcInput, _),
   %% Pass a message through the pipe
   fmt_write(WF, "Hello!\n", _),
   flush_output(WF, _),
   fmt_write(ProcInput, "end_of_file.\n",_), % send end_of_file atom to child
   flush_output(ProcInput, _),
   %% wait for child (so as to not leave zombies around;
   %% zombies quit when the parent finishes, but they consume resources)
   process_control(Process, wait),
   %% Close the ports used to commuicate with the process
   %% Otherwise, the parent might run out of file descriptors
   %% (if many processes were spawned)
   close(ProcInput), close(WF).


%% child.P
:- import file_read_line_atom/2 from file_io.
:- dynamic pipe/1.
?- pipe(P), open(pipe(P),read,F),
   %% Acknowledge receipt of the pipe
   fmt_write("\nPipe %d received\n", arg(P)),
   %% Get a message from the parent and print it to stdout
   file_read_line_atom(F, Line), write('Message was: '), writeln(Line).
```

This produces the following output:

```
| ?- [parent].                    <- parent XSB consults parent.P
[parent loaded]
yes
```

```
| ?- [xsb_configuration loaded]   <- parent.P spawns a child copy of XSB
[sysinitrc loaded]                   Here we see the startup messages of
[packaging loaded]                   the child copy
XSB Version 2.0 (Gouden Carolus) of June 27, 1999
[i686-pc-linux-gnu; mode: optimal; engine: slg-wam; scheduling: batched]
| ?-
yes
| ?- [Compiling ./child]          <- The child copy of received the pipe from
[child compiled, cpu time used: 0.1300 seconds]     the parent and then the
[child loaded]                                      request to consult child.P
Pipe 15 received                 <- child.P acknowledges receipt of the pipe
Message was: Hello!              <- child.P gets the message and prints it
yes
```

Observe that the parent process is very careful about making sure that the child terminates and also about closing the I/O streams after they are no longer needed.

Finally, we should note that this mechanism can be used to communicate through pipes with non-XSB processes as well. Indeed, an XSB process can create a pipe using `pipe_open` (*before* spawning a child process), pass one end of the pipe to a child process (which can be a C program), and use `open/3` to convert the other end of the pipe to an XSB stream. The C program, of course, does not need `open/3`, since it can use the pipe file handle directly. Likewise, a C program can spawn off an XSB process and pass it one end of a pipe. The XSB child-process can then convert this pipe fd to a file using `fd2iostream` and then talk to the parent C program.

`fd2iostream(+Pipe, -IOstream)` module: `shell`

Take a file descriptor and convert it to an XSB I/O stream. This predicate should be used only for user-defined I/O. Otherwise, use `open/{3,4}` when possible.

15.7 Socket I/O

The XSB socket library defines a number of predicates for communication over BSD-style sockets. Most are modeled after and are interfaces to the socket functions with the same name. For detailed information on sockets, the reader is referred to the

Unix man pages (another good source is *Unix Network Programming*, by W. Richard Stevens). Several examples of the use of the XSB sockets interface can be found in the `XSB/examples/` directory in the XSB distribution.

XSB supports two modes of communication via sockets: *stream-oriented* and *message-oriented*. In turn, stream-oriented communication can be *buffered* or *character-at-a-time*.

To use *buffered* stream-oriented communication, system socket handles must be converted to XSB I/O streams using `fd2iostream/2`. In these stream-oriented communication, messages have no boundaries, and communication appears to the processes as reading and writing to a file. At present, buffered stream-oriented communication works under Unix only.

Character-at-a-time stream communication is accomplished using the primitives `socket_put/3` and `socket_get0/3`. These correspond to the usual Prolog `put/1` and `get0/1` I/O primitives.

In message-oriented communication, processes exchange messages that have well-defined boundaries. The communicating processes use `socket_send/3` and `socket_recv/3` to talk to each other. XSB messages are represented as strings where the first four bytes (`sizeof(int)`) is an integer (represented in the binary network format — see the functions `htonl` and `ntohl` in socket documentation) and the rest is the body of the message. The integer in the header represents the length of the message body.

Effort has been made to make the socket interface thread-safe; however in Version 3.8, calls to the XSB socket interface go through a single mutex, and may cause contention if many threads seek to concurrently use sockets.

We now describe the XSB socket interface. All predicates below must be imported from the module `socket`. Note that almost all predicates have the last argument that unifies with the error code returned from the corresponding socket operation. This argument is explained separately.

General socket calls. These are used to open/close sockets, to establish connections, and set special socket options.

`socket(-Sockfd, ?ErrorCode)` module: `socket`

A socket `Sockfd` in the AF_INET domain is created. (The AF_UNIX domain is not yet implemented). `Sockfd` is bound to a small integer, called socket descriptor or socket handle.

`socket_set_option(+Sockfd,+OptionName,+Value)` module: `socket`

Set socket option. At present, only the `linger` option is supported. "Lingering" is a situation when a socket continues to live after it was shut down by the owner. This is used in order to let the client program that uses the socket to finish reading or writing from/to the socket. `Value` represents the number of seconds to linger. The value -1 means do not linger at all.

`socket_close(+Sockfd, ?ErrorCode)` module: `socket`
`Sockfd` is closed. Sockets used in `socket_connect/2` should not be closed by `socket_close/1` as they will be closed when the corresponding stream is closed.

`socket_bind(+Sockfd,+Port, ?ErrorCode)` module: `socket`
The socket `Sockfd` is bound to the specified local port number.

`socket_connect(+Sockfd,+Port,+Hostname,?ErrorCode)` module: `socket`
The socket Sockfd is connected to the address (`Hostname` and `Port`). If `socket_connect/4` terminates abnormally for any reason (connection refused, timeout, etc.), then XSb closes the socket `Sockfd` automatically, because such a socket cannot be used according to the BSD semantics. Therefore, it is always a good idea to check to the return code and reopen the socket, if the error code is not `SOCK_OK`.

`socket_listen(+Socket, +Length, ?ErrorCode)` module: `socket`
The socket `Sockfd` is defined to have a maximum backlog queue of `Length` pending connections.

`socket_accept(+Sockfd,-SockOut, ?ErrorCode)` module: `socket`
Block the caller until a connection attempt arrives. If the incoming queue is not empty, the first connection request is accepted, the call succeeds and returns a new socket, `SockOut`, which can be used for this new connection.

Buffered, message-based communication. These calls are similar to the `recv` and `send` calls in C, except that XSB wraps a higher-level message protocol around these low-level functions. More precisely, `socket_send/3` prepends a 4-byte field to each message, which indicates the length of the message body. When `socket_recv/3` reads a message, it first reads the 4-byte field to determine the length of the message and then reads the remainder of the message.

All this is transparent to the XSB user, but you should know these details if you want to use these details to communicate with external processes written in C and such. All this means that these external programs must implement the same protocol. The subtle point here is that different machines represent integers differently, so an integer must first be converted into the machine-independent network format using

the functions `htonl` and `ntohl` provided by the socket library. For instance, to send a message to XSB, one must do something like this:

```
char *message, *msg_body;
unsigned int msg_body_len, network_encoded_len;

  msg_body_len = strlen(msg_body);
  network_encoded_len = (unsigned int) htonl((unsigned long int) msg_body_len);
  memcpy((void *) message, (void *) &network_encoded_len, 4);
  strcpy(message+4, msg_body);
```

To read a message sent by XSB, one can do as follows:

```
int actual_len;
char lenbuf[4], msg_buff;
unsigned int msglen, net_encoded_len;

  actual_len = (long)recvfrom(sock_handle, lenbuf, 4, 0, NULL, 0);
  memcpy((void *) &net_encoded_len, (void *) lenbuf, 4);
  msglen = ntohl(net_encoded_len);

  msg_buff = calloc(msglen+1, sizeof(char))); // check if this suceeded!!!
  recvfrom(sock_handle, msg_buff, msglen, 0, NULL, 0);
```

If making the external processes follow the XSB protocol is not practical (because you did not write these programs), then you should use the character-at-a-time interface or, better, the buffered stream-based interface both of which are described in this section. At present, however, the buffered stream-based interface does not work on Windows.

`socket_recv(+Sockfd,-Message, ?ErrorCode)` module: `socket`

Receives a message from the connection identified by the socket descriptor `Sockfd`. Binds `Message` to the message. `socket_recv/3` provides a message-oriented interface. It understands message boundaries set by `socket_send/3`.

`socket_send(+Sockfd,+Message, ?ErrorCode)` module: `socket`

Takes a message (which must be an atom) and sends it through the connection specified by `Sockfd`. `socket_send/3` provides message-oriented communication. It prepends a 4-byte header to the message, which tells `socket_recv/3` the length of the message body.

Stream-oriented, character-at-a-time interface. Internally, this interface uses the same `sendto` and `recvfrom` socket calls, but they are executed for each character separately. This interface is appropriate when the message format is not known or when message boundaries are determined using special delimiters.

`socket_get0/3` creates the end-of-file condition when it receives the end-of-file character `CH_EOF_P` (a.k.a. 255) defined in `char_defs.h` (which must be included in the XSB program). C programs that need to send an end-of-file character should send `(char)-1`.

`socket_get0(+Sockfd, -Char, ?ErrorCode)` module: `socket`

The equivalent of `get0` for sockets.

`socket_put(+Sockfd, +Char, ?ErrorCode)` module: `socket`

Similar to put/1, but works on sockets.

Socket-probing. With the help of the predicate `socket_select/6` one can establish a group of asynchronous or synchronous socket connections. In the synchronous mode, this call is blocked until one of the sockets in the group becomes available for reading or writing, as described below. In the asynchronous mode, this call is used to probe the sockets periodically, to find out which sockets have data available for reading or which sockets have room in the buffer to write to.

The directory `XSB/examples/socket/select/` has a number of examples of the use of the socket-probing calls.

`socket_select(+SymConName,+Timeout,-ReadSockL,-WriteSockL,-ErrSockL,?ErrorCode)` module: `socket`

`SymConName` must be an atom that denotes an existing connection group, which must be previously created with `socket_set_select/4` (described below). `ReadSockL`, `WriteSockL`, `ErrSockL` are lists of socket handles (as returned by `socket/2`) that specify the available sockets that are available for reading, writing, or on which exception conditions occurred. `Timeout` must be an integer that specifies the timeout in seconds (0 means probe and exit immediately). If `Timeout` is a variable, then wait indefinitely until one of the sockets becomes available.

`socket_set_select(+SymConName,+ReadSockFdLst,+WriteSockFdLst,+ErrorSockFdLst)` module: `socket`

Creates a connection group with the symbolic name `SymConName` (an atom) for subsequent use by `socket_select/6`. `ReadSockFdLst`, `WriteSockFdLst`, and `ErrorSockFdLst` are lists of sockets for which `socket_select/6` will be used to monitor read, write, or exception conditions.

`socket_select_destroy(+SymConName)` module: `socket`

Destroys the specified connection group.

Error codes. The error code argument unifies with the error code returned by the corresponding socket commands. The error code -2 signifies *timeout* for timeout-enabled primitives (see below). The error code of zero signifies normal termination. Positive error codes denote specific failures, as defined in BSD sockets. When such a failure occurs, an error message is printed, but the predicate succeeds anyway. The specific error codes are part of the socket documentation. Unfortunately, the symbolic names and error numbers of these failures are different between Unix compilers and Visual C++. Thus, there is no portable, reliable way to refer to these error codes. The only reliably portable error codes that can be used in XSB programs defined through these symbolic constants:

```
#include "socket_defs_xsb.h"

#define SOCK_OK       0      /* indicates sucessful return from socket      */
#define SOCK_EOF     -1      /* end of file in socket_recv, socket_get0     */

#include "timer_defs_xsb.h"

#define TIMEOUT_ERR -2                /* Timeout error code */
```

Timeouts. XSB socket interface allows the programer to specify timeouts for certain operations. If the operations does not finish within the specified period of time, the operation is aborted and the corresponding predicate succeeds with the `TIMEOUT_ERR` error code. The following primitives are timeout-enabled: `socket_connect/4`, `socket_accept/3`, `socket_recv/3`, `socket_send/3`, `socket_get0/3`, and `socket_put/3`. To set a timeout value for any of the above primitives, the user should execute `set_timer/1` right before the subgoal to be timed. Note that timeouts are disabled after the corresponding timeout-enabled call completes or times out. Therefore, one must use `set_timer/1` before each call that needs to be controlled by a timeout mechanism.

The most common use of timeouts is to either abort or retry the operation that times out. For the latter, XSB provides the `sleep/1` primitive, which allows the program to wait for a few seconds before retrying.

The `set_timer/1` and `sleep/1` primitives are described below. They are standard predicates and do not need to be explicitly imported.

`set_timer(+Seconds)` S

et timeout value. If a timer-enabled goal executes after this value is set, the clock begins ticking. If the goal does not finish in time, it succeeds with the error code set to `TIMEOUT_ERR`. The timer is turned off after the goal executes (whether timed out or not and whether it succeeds or fails). This goal always succeeds.

Note that if the timer is not set, the timer-enabled goals execute "normally," without timeouts. In particular, they might block (say, on `socket_recv`, if data is not available).

`sleep(+Seconds)` P

ut XSB to sleep for the specified number of seconds. Execution resumes after the `Seconds` number of seconds. This goal always succeeds.

Here is an example of the use of the timer:

```
:- compiler_options([xpp_on]).
#include "timer_defs_xsb.h"

?- set_timer(3),  % wait for 3 secs
   socket_recv(Sockfd, Msg, ErrorCode),
   (ErrorCode == TIMEOUT_ERR
   -> writeln('Socket read timed out, retrying'),
      try_again(Sockfd)
   ;  write('Data received: '), writeln(Msg)
   ).
```

Apart from the above timer-enabled primitives, a timeout value can be given to `socket_select/6` directly, as an argument.

Buffered, stream-oriented communication. In Unix, socket descriptors can be "promoted" to file streams and the regular read/write commands can be used with such streams. In XSB, such promotion can be done using the following predicate:

`fd2ioport(+Pipe, -IOport)` module: `shell`

Take a socket descriptor and convert it to an XSB I/O port that can be used for regular file I/O.

Once `IOport` is obtained, all normal I/O primitives can be used by specifying the `IOport` as their first argument. This is, perhaps, the easiest and the most convenient way to use sockets in XSB. (This feature has not been implemented for Windows.)

Here is an example of the use of this feature:

```
:- compiler_options([xpp_on]).
#include "socket_defs_xsb.h"

?- (socket(Sockfd, SOCK_OK)
   ->   socket_connect(Sockfd1, 6020, localhost, Ecode),
        (Ecode == SOCK_OK
        -> fd2ioport(Sockfd, SockIOport),
           file_write(SockIOport, 'Hello Server!')
        ;  writeln('Can''t connect to server')
        ),
    ;   writeln('Can''t open socket'), fail
    ).
```

15.8 Arrays

The module `array1` provides a simple backtrackable array implementation that requires no copying. In Version 3.2, this package was changed to make use of the backtrackable destructive assignment made possible by `setarg/3`. We note that as of Version 3.2 this library provides simple syntactic sugar for `functor/3`, `arg/3` and `setarg/3` and relies on error messages for these predicates.

`array_new(-Array,+Size)` module: `array`

Creates a one dimensional empty array of size `Size`. All the elements of this array are variables.

`array_elt(+Array, +Index, ?Element)` module: `array`

Succeeds iff `Element` unifies with the `Index`-th element of array `Array`.

`array_update(+Array, +Index, +Elem)` module: `array`

Updates the array `Array` such that the `Index`-th element of the new array is `Elem` using destructive assignement. The implementation is quite efficient in that it avoids the copying of the entire array.

The following example shows the use of these predicates:

```
| ?- import array_new/2, array_elt/3, array_update/4 from array.
```

```
yes
| ?- array_new(A,3), array_update(A,1,1), array_update(A,2,2),
     ( array_update(A,3,3), writeln(first(A))
     ; array_update(A,3,6), writeln(second(A))
     ; array_update(A,3,7), writeln(third(A))),fail.

first(array(1,2,3))
second(array(1,2,6))
third(array(1,2,7))

no
```

15.9 The Profiling Library

XSB can provide Prolog-level profiling for Prolog programs, which allows the Prolog programmer to estimate what proportion of time is spent executing code for each predicate, and also what modes have been used to call a given predicate. It also helps to find unindexed accesses to dynamic predicates which may be the cause of poor performance. To enable profiling, XSB must be started with the command line parameter of `-p`. The module `xsb_profiling` contains the predicate `profile_call/1` that invokes profiling. The profiling library should only be used with the single-threaded engine in Version 3.8.

`profile_call(+Goal)` module: `xsb_profiling`

> Calls `Goal`, and when it first succeeds, prints to `userout` a table of predicate names indicating for each, the percentage of time spent executing that predicate's code. Within the table, the sum of the predicate times for each module is also given. `Goal` may backtrack, but profiling is done only for the time to the first success, so it is most appropriate to profile succeeding deterministic goals [10].

Profiling works by starting another thread that interrupts every 100th of a second and sets a flag so that the XSB emulator will determine the predicate of the currently executing code. The printout also includes the total number of interrupts and for each predicate, the raw number of times its code was determined to be executing. A predicate is printed only if its code was interrupted at least once. The numbers will

[10] This includes tabled subgoals under Local Evaluation, as such as goal will only succeed after deriving all of its answers.

be meaningful only for relatively long-running predicates, taking more than a couple of seconds.

When an interrupt occurs, the **next** *interrupt instruction* to be executed – a WAM call, execute, proceed or trust instruction – will charge its associated predicate by logging that predicate to a table. The system does not keep track of code addresses for tries (used to represent the results of completed tables, and trie-indexed asserted code), so for some interrupts the associated executing predicate cannot be determined. In these cases the interrupt is charged against an "unknown/?" pseudo-predicate, and this count is included in the output.

Profiling does not give the context from which the predicate is called, so you may want to make renamed copies of basic predicates to use in particular circumstances to determine their times.

Predicates compiled with the "optimize" option may provide misleading results under profiling. Note that all system predicates (including those in `basics`) are compiled with the "optimize" option, by default. That option causes tail-recursive predicates to use a "jump" instruction rather than an "execute" instruction to make the recursive call, and so an interrupt in such a loop will not be charged until the next interrupt instruction is executed. If much time is spent in the recursion, this might not be for a long time, and the interrupt might ultimately be charged to another predicate. (If an interrupt has not been charged by the time of the next interrupt, it is lost.)

Profiling is currently available under Windows, Mac OS X, and Linux. However, for the profiling algorithm to provide a good estimation, the thread that wakes and sets the interrupt flag must be of high priority and given the CPU when it wants it. Accordingly, the estimates may be better or worse depending on the scheduling strategy of a given platform [11].

The profiling module also provides support for determining when a dynamic predicate is invoked in a mode that isn't supported by any index. The XSB programmer can set a flag that will cause a message to be printed when a dynamic predicate is invoked, no index is applicable, and there are more than 20 potentially matching clauses. See `profile_unindexed_calls/1` below for details.

`profile_mode_call(+Goal)` module: `xsb_profiling`

[11] Windows and Mac OS X 10.6 provide good estimates. Some Linuxes however, do not charge about 20% of their interrupts due to thread scheduling issues. This loss of interrupts makes the profile estimate inefficient, but does not bias the estimate. We haven't figured out how to get priority scheduling for interrupts on all machines, so if you want profiling to work more efficiently, maybe you can help figure out how to get appropriate scheduling.

Calls the goal `Goal` and constructs a table of the modes in which the predicate is called and the number of times it is called in that mode. Modes are simply "b" for ground and "f" for variable. Counts are kept in a table with entries of the form `Pred(Md1,Md2,..,Mdn)` where Pred is the name of the called predicate and the Mdi are either 'f' or 'b', indicating free or bound for the corresponding argument. The table can be printed using `profile_mode_dump/0` and can be cleared using `profile_mode_init/0`.

`profile_mode_dump` module: `xsb_profiling`

Prints out the counts of calls in particular modes as accumulated using `profile_mode_call(+Goal)`.

`profile_mode_init` module: `xsb_profiling`

Clears the table that accumulates counts of calls in particular modes (done by `profile_mode_call(+Goal)`.

`profile_unindexed_calls(+Par)` module: `xsb_profiling`

Sets the kind of unindexed profiling to perform. If `Par` is `off`, no unindexed logging will be done. This is the default. If `Par` is `once` each call to a dynamic predicate that cannot use any index (and would backtrack through more than 20 clauses) will generate a log message to userout. Note that the predicate of the goal may have indexes, but the particular goal may not be able to take advantage of them. E.g., a totally open call to a predicate with many clauses will generate an unindexed message. By setting the `once` parameter, each unindexed call to a predicate will be logged only once; after logging is done, the log instruction is changed to a branch, so it will never produce another log message for that dynamic code. If `Par` is `on`, logging is done as for `once`, except every unindexed call to any dynamic predicate will be logged; i.e. the logging instruction is not changed after logging. If `Par` is a predicate specification (of the form Pred/Arity, Module:Pred/Arity, Term, or Module:Term), only unindexed calls to the indicated goal will be logged, and when each is logged a back-trace will be printed. This allows the programmer to find the location of an unindexed call.

15.10 Gensym

The Gensym library provides a convenient way to generate unique integers or constants.

`prepare(+Index)` module: `gensym`

Sets the initial integer to be used for generation to `Index`. Thus, the command `?- prepare(0)` would cause the first call to `gennum/1` to return `1`. `Index` must be a non-negative integer.

`gennum(-Var)` module: `gensym`

Unifies `Var` with a new integer.

`gensym(+Atom,-Var)` module: `z`

zzzgensym Generates a new integer, and concatenates this integer with `Atom`, unifying the result with `Var`. For instance a call `?- gensym(foo,Var)` might unify `Var` with `foo32`.

15.11 Random Number Generator

The following predicates are provided in module `random` to generate random numbers (both integers and floating numbers), based on the Wichmann-Hill Algorithm [94, 58]. The random number generator is entirely portable, and does not require any calls to the operating system. As noted below, it does require 3 seeds, each of which must be an integer in a given range. These seeds are thread-specific: thus different threads may generate independent sequences of random numbers.

`random(-Number)` module: `random`

Binds `Number` to a random float in the interval [0.0, 1.0). Note that 1.0 will never be generated.

`random(+Lower,+Upper,-Number)` module: `random`

Binds `Number` to a random integer in the interval [`Lower`,`Upper`) if `Lower` and `Upper` are integers. Otherwise `Number` is bound to a random float between `Lower` and `Upper`. `Upper` will never be generated.

`getrand(?State)` module: `random`

Tries to unify `State` with the term `rand(X,Y,Z)` where `X`,`Y`,and `Z` are integers describing the state of the random generator.

`setrand(rand(+X,+Y,+Z))` module: `random`

Sets the state of the random generator. `X`,`Y`, and `Z` must be integers in the ranges [1,30269), [1,30307), [1,30323), respectively.

`datime_setrand` module: `random`

This simple initialization utility sets the random seed triple based on a function of the current day, hour, minute and second.

`randseq(+K, +N, -RandomSeq)` module: `random`

Generates a sequence of `K` unique integers chosen randomly in the range from 1 to `N`. `RandomSeq` is not returned in any particular order.

`randset(+K, +N, -RandomSet)` module: `random`

Generates an ordered set of `K` unique integers chosen randomly in the range from 1 to `N`. The set is returned in reversed order, with the largest element first and the smallest last.

`gauss(-G1,-G2)` module: `random`

Generates two random numbers that are normally distributed with mean 0 and standard deviation 1. It uses the polar form of the Box-Muller transformation [8] of uniform random variables as generated by `random/1`.

`weibull(K,Lambda,X)` module: `random`

Generates a random number for the Weibull distribution:

$$f(x;k,\lambda) = \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1}e^{-(x/\lambda)^h}$$

based on the transformation

$$x = \lambda(-ln(U))^{1/k}$$

of a uniformly distributed random variable produced by `random/1`

`exponential(K,X)` module: `random`

Generates a random number for the exponential distribution:

$$f(x;k,\lambda) = \frac{e^{-(x/\lambda)^h}}{\lambda}$$

based on the transformation

$$x = \lambda(-ln(U))$$

of a uniformly distributed ranom variable produce by `random/1`. This is the same as the Weibull distribution with $k = 1$.

15.12 Loading Delimiter-Separated Files

A common file format uses comma separated values, the so-called csv files. The XSB module, `proc_files`, supports the loading of files in this, and similar, formats to define Prolog predicates.

`load_csv(+FileName,+PredSpec)` module: `proc_files`

`load_csv/2` takes a file name and a predicate specification, and reads a csv-formatted file into memory, defining the indicated dynamic predicate. The simplest form of `PredSpec` is `PredName/Arity`. In this case the arity must equal the number of fields in the csv file, and the predicate must be dynamic. Each line in the file will define one fact of the predicate `PredName/Arity`. Fields in the file enclosed in double quotes will be treated as single fields (and thus can contain commas and new-lines.) The dynamic predicate will be emptied before the facts from the file are added. Each field will be loaded as an atom (including fields that contain just integers.)

Alternatively, `PredSpec` may be of the form `predName(TypeSpec1,...,TypeSpecN)`, where `predName` is the name of the dynamic predicate to be defined by the file contents, and each `TypeSpecI` indicates the type of the corresponding field in the file. The permitted values of `TypeSpec` are:

- `atom` The corresponding field value will become an atom in the loaded fact.
- `integer` The corresponding field value will be converted to an integer in the loaded fact.
- `float` The corresponding field value will be converted to a float in the loaded fact.
- `term` The corresponding field must contain a Prolog term in canonical form, and it will be converted to that term in the loaded fact.
- `_` (A variable) Treated as `atom`.

`load_dsv(+FileName,+PredSpec,+Options)` module: `proc_files`

This predicate supports the loading of more general forms of files with value-separated fields. The `FileName` and `PredSpec` parameters are exactly as in `load_csv/2`, as described just above. `Options` is a list of options. (With an empty list, `load_dsv` acts as `load_csv/2`.) The options are:

`separator="Sep"` which indicates that the character(s) `Sep` will be used as the field separator. There may be one or more characters.

`delimiter="C"` which indicates that the single character `C` will be used as the field delimiter (the default being """", and I've yet to find a situation in which I want to change it.)

`titles` which indicates that the first line of the file should be ignored and not contribute a fact to the dynamic predicate.

15.13 Scanning in Prolog

Scanners, (sometimes called tokenizers) take an input string, usually in UTF-8 or similar format, and produce a scanned sequence of tokens. The requirements that various applications have for scanning differ in small but important ways – a character that is special to one application may be part of the token of another; or some applications may want lower case text converted to upper-case test. The `stdscan.P` library provides a simple scanner written in XSB that can be configured in several ways. While useful, this scanner is not intended to be as powerful as general-purpose scanners such as *lex* or *flex*.

`scan(+List,-Tokens)` module: `stdscan`

Given as input a `List` of character codes, `scan/2` scans this list producing a list of atoms constituting the lexical tokens. Its parameters are set via `set_scan_pars/1`.

Tokens produced are either a sequence of *letters* and/or *numbers* or consist of a single *special character* (e.g. `(` or `)`). Whitespaces may occur between tokens.

`scan(+List,+FieldSeparator,-Tokens)` module: `stdscan`

Given as input a `List` of character codes, along with a character code for a field separator, `scan/3` scans this list producing a list of list of atoms constituting the lexical tokens in each field. *scan/3* thus can be used to scan tabular information. Its parameters are set via `set_scan_pars/1`.

`set_scan_pars(+List)` module: `stdscan`

`set_scan_pars(+List)` is used to configure the tokenizer to a particular need. `List` is a list of parameters including the following:

- `whitespace`. The default action of the scanner is to return a list of tokens, with any whitespace removed. If `whitespace` is a parameter, then the scanner returns the token `" "` when it finds whitespace separating two tokens (unless the two tokens are letter sequences; since two letter sequences can be two tokens ONLY if they are separated by whitespace, such an indication of whitespace would be redundant.) Including the parameter `no_whitespace` undoes the effect of previously including `whitespace`.
- `upper_case` The default action of the parser is to treat lowercase letter differently from uppercase letters. This parameter should be set if conversion to uppercase should be done when producing a token that does *not* consist entirely of letters (e.g. one with mixed letters and digits). Including the parameter `no_case` undoes the effect of previously including `upper_case`.
- `upper_case_in_lit` The default action of the parser is to treat lowercase letter differently from uppercase letters. This parameter should be set if conversion to uppercase should be done when producing a token that consists entirely of letters. Including the parameter `no_case_in_lit` undoes the effect of previously including `upper_case`.
- `whitespace(Code)` adds `Code` as a whitespace code. By default, all ASCII codes less than or equal to `32` are regarded as whitespace.
- `letter(Code)` adds `Code` as a letter constituting a token. By default, ASCII codes for characters `a-z` and `A-Z` are regarded as letters.
- `special_char(Code)` adds `Code` as a special character. By default, ASCII codes for the following characters are regarded as special characters:

  ```
  | { } [ ] " % $ & ' ( ) * + , - . / : ; < = > ? @ \ ^ _ ~ `
  ```

`get_scan_pars(-List)` module: `stdscan`

`get_scan_pars/1` returns a list of the currently active parameters.

15.14 XSB Lint

The `xsb_lint_impexp.P` file contains a simple tool to analyze import/exports along with definitions and uses of predicates. It tries to find possible inconsistencies, producing warnings when it finds them and generating `import`, `document_import`, and `document_export` declarations that might (or might not) be useful. It can be used after a large multi-file, multi-module XSB program has been written to find possible inconsistencies in (or interesting aspects of) how predicates are defined and used.

We emphasize that the import and export statements generated by `checkImpExps/1/2` are suggestions only. The user is responsible for determining if they are indeed correct and should be added to the corresponding source file. There are situations in which adding such a generated import declaration may break existing code.

XSB source files that contain an `export` compiler directive are considered as modules. Predicates defined in modules, but not exported, are local to that module. When compiling a module, the XSB compiler generates useful warnings when predicates are used but not defined or defined but not used. All predicates that are defined in source files that do not contain an `export` directive are compiled to be defined in a global module, called `usermod`, and no undefined/unused warning messages are generated. The user may add `document_export` and `document_import` compiler directives (exactly analogous to the `export` and `import` directives) to non-module source files. These directives are ignored by the compiler in terms of code generation, but cause the define-use analysis to be performed, issuing warning messages as appropriate. This allows a user to get the benefit of the define-use analysis without using modules. (See Volume 1, Chapter 3 for more details.)

The `xsb_lint_impexp` utility processes both modules and regular XSB source files that may or may not contain `document_export` statements. `xsb_lint_impexp` is itself a module. To use it, one may explicitly call `xsb_lint_impexp:checkImpExps(...)`, or may consult `[xsb_lint_impexp]` and then call the `checkImpExps/{1,2}` predicate.

`checkImpExps(+Options,+FileNameList)`

checkImpExps/2xsb_lint `checkImpExps/2` reads all the XSB source files named in the list `FileNameList`, and all files they reference (recursively), and produces a listing that describes properties of how they reference predicates. All referenced files are found using the XSB `library_directory/1` predicate of directories. `checkImpExps/2` uses `add_lib_dir/1/2` directives in code files to update the directory paths. The user may also explicitly add paths by calling `add_lib_dir/1/2` before calling this predicate.

`Options` is a list of atoms (from the following list) indicating details of how `checkImpExps` should work and the messages it should produce.

1. `used_elsewhere`: Print a warning message in the case of a predicate defined in a file, not used there, but used elsewhere (in a file in `FileNameList` or a recursively referenced file). This can be useful to see whether it might be better to move the predicate definition to another file, but it produces many (irrelevant) warnings for predicates in multi-use libraries.

2. `unused`: Print a warning message in the case of a predicate that is exported but never used. This can be useful to see if a predicate might be deleted. Again this option produces many (irrelevant) warnings for predicates in multi-use libraries.
3. `all_files`: By default, only predicates in files that contain a `:- document_export` or `:- export` declaration are processed for warnings. This option causes predicates of *all* files (and modules) to be processed. This means that usermod code files without `document_export` declarations will have them generated.
4. `all_symbol_uses`: Treat *all* uses of symbols (even constants) as predicate uses for the purpose of generating imports. This means that symbols used as functor symbols but not a predicate symbols, will be treated as referring to the predicate symbol. This can be useful when a program defines its own meta-predicates and passes predicate terms to another module to be called. However, it can generate spurious messages when a common symbol is used as both a predicate and as an unrelated functor symbol. This differs from the default behavior of `checkImpExps/1/2` only in that the default does *not* consider 0-ary functor symbols as predicate uses, whereas this option does.
5. `no_symbol_uses`: Don't treat any purely functor uses of symbols as predicate uses for the purpose of generating imports. This means that a term that appears in an argument position (i.e., not as a called predicate) will *not* be considered as a use of the predicate symbol at the root of the term. Only symbols that are called (or appear in meat-argument positions of system-defined meta-predicates) will be considered as used.

The final two options allow the user to control *predicate usage analysis* – analysis of when a symbol s might be used as a predicate symbol. We term occurrences of s as a body literal of a rule or in a callable argument in a meta-predicate, as *strict predicate contexts*. The default behavior of this library is as follows.

- if s is a constant symbol, the predicate usage analysis of s is restricted to strict predicate contexts.
- if s is not a constant symbol, the predicate usage analysis of s is based on non-strict predicate contexts. I.e, *all* occurrences of s count as predicate contexts.

Predicate usage analysis can be restricted to strict predicate contexts for all symbols by the option `no_symbol_uses`. Alternatively analysis can be expanded so

that non-struct predicate contexts are used for all symbols (including constants) by the option `all_symbol_uses`.

`checkImpExps(+FileNameList)` module: `xsb_lint`

`checkImpExps/1` is currently equivalent to `checkImpExps([],FileNameList)`.

15.15 "Pure" Meta-programming in XSB with prolog_db.P

The `prolog_db` library provides predicates that support a form of "pure" meta-programming in XSB. A programmer can create a term data structure that represents a Prolog database (i.e., a set of rules, and herein called a *Prolog DB*), and then ask for a goal to be proved in such a Prolog DB.

A Prolog DB is kept as a trie, which is a ground Prolog term. Each level in the trie is implemented by a hash table, and hash tables are expanded and contracted as necessary. A set of clauses is canonically represented, i.e., no matter what sequence of `assert_in_db`'s and `retractall_in_db`'s one uses to construct a particular set of clauses, the resulting Prolog DBs (i.e. Prolog terms) are identical.

A Prolog DB represents an unordered set of clauses. The order in which clauses are returned from `clause_in_db` (and thus for `call_in_db`) is indeterminate, and may change from one call to the next (due to possible expansion or contraction of a hash table in the representation of a Prolog DB.)

A Prolog DB that is obtained from another Prolog DB by adding or deleting a single clause differs from it in only log subterms (unless a hash table has been resized). This means that it is efficient to intern these DB's, and to table them (as intern).

The predicates provided by the Prolog DB interface are as follows:

`empty_db(-EmptyPrologDB)` module: `prolog_db`

`empty_db/1` returns an empty Prolog DB. It is used to create an initial Prolog DB to pass to the other in_db predicates. [12]

`assert_in_db(+Clause,+DB0,-DB)` module: `prolog_db`

`assert_in_db/3` adds the clause, `Clause`, to the Prolog DB, `DB0`, and returns a new Prolog DB, `DB`. A Prolog DB is a *set* of clauses, so asserting a clause

[12] Since a Prolog DB is a term and must be passed as an argument, it differs from other implementations of tries in XSB, such as interned tries (cf. Chapter 8), or trie-indexed facts (cf. Section 6.14), which are persistent.

that is already in DB0 just returns that same database. No ordering of clauses is preserved, so cuts do not make sense and cannot be used in clauses. (The if-then else ('->'/3) should be used instead.)

`retractall_in_db(+ClauseHead,+DB0,-DB)` module: `prolog_db`
`retractall_in_db/3` removes all clauses whose heads unify with `ClauseHead` from `DB0` returning `DB`. If no clauses in `DB0` unify, then `DB0` is returned unchanged.

`clause_in_db(?ClauseHead,?ClauseBody,+DB)` module: `prolog_db`
`clause_in_db/3` returns all clauses in `DB` whose heads and bodies unify with `ClauseHead` and `ClauseBody`, respectively. (Note that, unlike `clause/2` in Prolog, `clause_in_db` can be called with `ClauseHead` as a variable.) Note also that the order of clauses is not preserved and is indeterminate.

`call_in_db(?Goal,+DB)` module: `prolog_db`
`call_in_db/2` calls `Goal` in `DB` and returns all instances of `Goal` provable by rules in `DB`. Clauses must not contain cuts (!). They can contain most Prolog constructs, including and, or, if-then-else, \+, calls to standard predicates, and calls explicitly modified by a module name. Such calls will be satisfied by calling the goal in the indicated module. So in this case one can think of a Prolog DB as being extended by the code in any module.

`load_in_db(+FileName,+DB0,-DB)` module: `prolog_db`
`load_in_db/3` reads the clauses from the file named `FileName` and asserts them into database `DB0` returning `DB`.

`load_in_db(+FileName,-DB)` module: `prolog_db`
`load_in_db/2` reads the clauses from the file named `FileName` and asserts them into an empty database returning `DB`.

`union_db(+DB1,+DB2,-DB3)` module: `prolog_db`
`union_db/3` returns in `DB3` the union of the sets of clauses in `DB1` and `DB2`.

15.16 Range Trees

This library contains predicates that provide support for range queries via *range trees*. Range trees store a set of keys and associated values. Arbitrary ranges of keys (along with their associated values) can be retrieved efficiently; such keys do not need to be numeric.. Such a library is needed because all engine-level indexes in XSB are hash-based and as such do not support efficient range queries.

Given a set of ground facts to be range-indexed on various arguments, several predicates support the easy construction of range trees from these facts. This basic interface includes predicates to create, add to, retrieve from, and delete from range trees.

Range trees use a balanced sort tree, similar to a B-Tree in that all leaf nodes are equidistant from the root, and nodes are at least half full. (Deletion of key-value pairs is supported, but trees are not rebalanced on delete, so significant use of delete can seriously degrade performance.) Range trees are stored in dynamic predicates and are identified by user-provided handles.

Example A user may have a predicate `data(Key,Val1,Val2,Val3)` and want to be able to do efficient range-valued queries for the second field, `Val1`, or for the fourth field, `Val3`. `Key` is a key for the `data/4` relation, i.e., no two different tuples have the same value for the `Key` fields. The user would make the following definitions: (See the specific predicate documentation below for these range predicates to understand their parameters in detail.)

```
:- import range_call/4, range_assert/3, range_retractall/4 from range_trees.

% to retrieve data by range from a range-indexed predicate
range_data(K,RV1,V2,RV3) :- range_call(data(K,_,V2,_),[1],[2,4],[RV1,RV3]).

% to add data to range-indexed predicate
assert_data(K,V1,V2,V3) :- range_assert(data(K,V1,V2,V3),[1],[2,4]).

% to delete data from range-indexed predicate
retractall_data(K,RV1,V2,RV3) :-
      range_retractall(data(K,_,V2,_),[1],[2,4],[RV1,RV3]).
```

With these definitions, the user will (init) and then add all data to `data/4` using `assert_data/4`, which will both assert a tuple to `data`/4 and add the tuples to the necessary range trees to support efficient range queries on the second or fourth argument.

A range query to `data/4` now is posed by calling the user-defined predicate `range_data/4`. For example to retrieve all tuples whose second field is between 4000 and 5000 (inclusive), the user would pose the query:

```
| ?- range_data(K,V1:[4000,5000],V2,V3).
```

This query will bind `K,V1,V2,V3` to quadruples in the `data/4` relation where `V1` is between 4000 and 5000 (inclusive.) Ranges are indicated by terms of the form `Var:[Low,High]`, where `Var` is the variable that will be bound on return, and `Low` and `High` are ground values indicating the lower and upper bounds of the desired range, respectively. The order of answers returned to a range query is indeterminate (i.e., not necessarily in increasing order on the range variable.)

One may also use a query such as:

```
| ?- range_data(K,4015,V2,V3).
```

which is treated as a range query with the same lower and upper bounds. For the declarations shown above, which indicate two range-indexed fields for `data/4`, one may also pose a query:

```
| ?- range_data(K,V1,V2,V3:[6015,7000]).
```

which would efficiently retrieve data/4 tuples whose fourth field is between 6015 and 7000 (inclusive.)

When multiple range-indexed arguments are given ranges (or constants) in a range query, only the first will be used for indexing.

`init_range_tree(+TreeId)` module: `range_trees`

This predicate initializes a named tree which will provide access to key-value pairs through range queries on the keys. `TreeId` is an arbitrary user-supplied ground term that identifies the particular tree.

`get_from_range_tree(+TreeId,+Lo,+Hi,?Key,-Val)` module: `range_trees`

Gets a range of Keys (and their associated values) between `Lo` and `Hi` (inclusive), using the ordering defined by `range_tree_compare/2`. `TreeId` is a user-provided tree identifier.

`add_to_range_tree(+TreeId,+Key,+Val)` module: `range_trees`

Adds a key-value pair to a range tree.

`delete_from_range_tree(+TreeId,+Key,?Val)` module: `range_trees`

Deletes all key-value pairs with the given `Key` and `Val` from the range tree. It does not re-balance the range tree, so after many deletes the tree may give bad performance.

`delete_from_range_tree(+TreeId,+Key)` module: `range_trees`
Deletes all key-value pairs with the given `Key` from the range tree. It does not re-balance the range tree, so after many deletes the tree may give bad performance.

`delete_range_tree(+TreeId)` module: `range_trees`
Deletes everything from the named tree (i.e., deletes the tree).

`delete_all_range_trees` module: `range_trees`
Deletes all range trees, reinitializing everything.

`range_call(Goal,KeyPosList,RangePosList,RangeFormList)` module: `range_trees`
Calls `Goal` using range indexing specifications in `RangeFormList`, which binds variables in Goal in positions `RangePosList`. For example, to call a predicate `data/4`, whose key is field 1 and which has range indexes on fields 2 and 4, one could call:

```
| ?- range_call(data(K,X,Y,Z),[1],[2,4],[_,_:[4000,4500]]).
```

This will efficiently return all triples of the stored predicate `data/4` whose fourth field is in the range 4000-4500. It is assumed that range indexes have been built for the second and fourth fields of `data/4` (normally by using `range_assert/3`.) This predicate is intended to be used by the user to define a predicate that can be used to get range-indexed access to another data predicate, as in the example above.

The `KeyPosList` is the list of positions in `Goal` that provide a key to the base predicate of `Goal`. That predicate should be indexed on this key. (If it is not a key or if it is not indexed on this key, there may be serious degradation of performance.) The `RangeFormList` is a list of range specifications, i.e., terms of form `Var:[Low,High]`, or constants or variables. The first in this list that has a value or range-specification (i.e., not a variable) will be used for range-indexing. The `RangePosList` is the (corresponding) list of argument positions in `Goal` that are range-indexed. These argument position lists must correspond to those used in `range_assert/3`.

`range_retractall(Goal,KeyPosList,RangePosList,RangeFormList)` module: `range_trees`
Removes all tuples in the database that would be retrieved by a call to `range_call/4`, with the same arguments. This also updates the range-indexes for this predicate.

Notice that this supports efficient retraction through use of range-restricted arguments in `Goal`. Note also, however, that since range trees are not rebalanced after deletion, heavy use of this predicate may cause performance degradation.

15.17 Miscellaneous Predicates

`term_hash(+Term,+HashSize,-HashVal)` module: `machine`

Given an arbitrary Prolog term, `Term`, that is to be hashed into a table of `HashSize` buckets, this predicate returns a hash value for `Term` that is between `0` and `HashSize -1`.

`crypto_hash(+Type,+Input,-Output)` module: `machine`

Given an atom `Input`, produces an encrypted string `Output` according to the algorithm specified in `Type`, which currently can be `sha1` or `md5`.

`pretty_print(+ClausePairs)` module: `pretty_print`
`pretty_print(+Stream,+ClausePairs)` module: `pretty_print`

The input to `pretty_print/1`, `ClausePairs`, can be either a list of clause pairs or a single clause pair. A clause pair is either a Prolog clause (or declaration) or a pair:

```
(Clause,Dict)
```

Where `Dict` is a list of the form `A = V` where `V` is a variable in `Clause` and `A` is the string to be used to denote the variable [13].

By default, `pretty_print/1` outputs atomic terms using `writeq/1`, but specialized output can be configured via asserting in `usermod` a term of the form

```
user_replacement_hook(Term,Call)
```

which will use Call to output an atomic literal A whenever A unifies with `Term`. For example, pretty printing weight constraints in XSB's `XASP` package is done via the hook

```
user_replacement_hook(weight_constr(Term),output_weight_constr(Term))
```

[13] Thus the list of variable names returned by `read_term/{2,3}` can be used directly in `Dict`.

which outputs a weight constraint in a (non-Prolog) syntax that is used by several ASP systems.

`module_of_term(+Term,?Module)` module: `machine`

Given a term `Term`, `module_of_term/2` returns the module of its main functor symbol in `Module`. If the module cannot be determined wither `unknown1` or `unknown2` is returrned, depending on the reason the module name cannot be determined.

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 3.10 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 in-between two non-special characters this adds all characters in-between (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 contortions.

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.

Bibliography

[1] H. Ait-Kaci. *The WAM: a (Real) Tutorial*. 1990. Available at http://wambook.sourceforge.net.

[2] 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 Symposium*, pages 424–439, 1994.

[3] J. Alferes, C. Damasio, and L. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14:93–147, 1995.

[4] F. Banchilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *ACM Principles of Database Systems*. ACM, 1986.

[5] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3):255–299, 1991.

[6] A. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[7] 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.

[8] G. Box and M. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[9] D. Butenhof. *Programming with POSIX Threads*. Prentice-Hall, 1997.

[10] M. Calejo. Interprolog: A declarative java-prolog interface. In *EPIA*. Springer-Verlag, 2001. See XSB's home page for downloading instructions.

[11] L. Castro and V. S. Costa. Understanding memory management in Prolog systems. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 11–26. Springer, 2001.

[12] L. Castro, T. Swift, and D. S. Warren. Suspending and resuming computations in engines for SLG evaluation. In *Practical Applications of Declarative Languages*, pages 332–346, 2002.

[13] L. Castro, T. Swift, and D. S. Warren. XASP: Answer Set Programming in XSB. Manual to Open-source software availible at xsb.sourceforge.net, 2002.

[14] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[15] W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[16] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[17] 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.

[18] B. Cui and T. Swift. Preference logic grammars: Fixed-point semantics and application to data standardization. *Artificial Intelligence*, 138:117–147, 2002.

[19] B. Cui, T. Swift, and D. S. Warren. From tabling to transformation: Implementing non-ground residual programs. In *International Workshop on Implementations of Declarative Languages*, 1999.

[20] B. Cui and D. S. Warren. A system for tabled constraint logic programming. In *Computational Logic*, pages 478–492, 2000.

[21] S. Dawson, C. R. Ramakrishnan, S. Skiena, and T. Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, 18(5):528–563, 1996.

[22] 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.

[23] B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, oct 2002. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.

[24] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *Priniclpes of Declarative Programming, 10th International Symposium*, pages 21–35. Springer-Verlag, 1998. LNCS 1490.

[25] 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. ACM Press, 1998.

[26] J. Desel and W. Reisig. Place/transition Petri nets. In *Lectures on Petri Nets I: Basic Models*, pages 122–174. Springer LNCS 1491, 1998.

[27] S. Dietrich. *Extension Tables for Recursive Query Evaluation*. PhD thesis, SUNY at Stony Brook, 1987.

[28] J. Freire, R. Hu, T. Swift, and D. S. Warren. Parallelizing tabled evaluation. In *7th International PLILP Symposium*, pages 115–132. Springer-Verlag, 1995.

[29] J. Freire, T. Swift, and D. S. Warren. Treating I/O seriously: Resolution reconsidered for disk. In *14th International Conferene on Logic Programming*, 1997. To Appear.

[30] J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*, 1998(3):243–268, 1998.

[31] J. Freire, T. Swift, and D. S. Warren. A formal framework for scheduling in SLG. In *International Workshop on Tabling in Parsing and Deduction*, 1998.

[32] T. Fruhwirth. Constraint handling rules. *Journal of Logic Programming*, 1998.

[33] J. Gartner, T. Swift, A. Tien, L. M. Pereira, and C. Damásio. Psychiatric diagnosis from the viewpoint of computational logic. In *International Conference on Computational Logic*, pages 1362–1376. Springer-Verlag, 2000. LNAI 1861.

[34] B. Grosof and T. Swift. Radial restraint: A semantically clean approach to bounded rationality for logic programs. In *Proceedings of the American Association for Artificial Intelligence*, 2013.

[35] H. Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2001.

[36] F. Harary. *Graph Theory.* Addison Wesley, 1969.

[37] ISO working group JTC1/SC22. Prolog international standard ISO-IEC 13211-1. Technical report, International Standards Organization, 1995.

[38] New built-in flags, predicates and functions proposal. Technical report, International Standards Organization, 2006. Edited by P. Moura, ISO/IEC DTR 13211-1:2006.

[39] Prolog multi-threaded support. Technical report, International Standards Organization, 2007. Edited by P. Moura, ISO/IEC DTR 13211-5:2007.

[40] E. Johnson, C. R. Ramakrishnan, I. V. Ramakrishnan, and P. Rao. A space efficient engine for subsumption-based tabled evaluation of logic programs. In A. Middeldorp and T. Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming*, number 1722 in Lecture Notes in Computer Science, pages 284–299. Springer-Verlag, Nov. 1999.

[41] T. Kanamori and T. Kawamura. Abstract interpretation based on OLDT resolution. *Journal of Logic Programming*, 15:1–30, 1993.

[42] 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.

[43] 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.

[44] M. Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12(4):335–368, 1992.

[45] R. Larson, D. S. Warren, J. Freire, and K. Sagonas. *Syntactica.* MIT Press, 1995.

[46] R. Larson, D. S. Warren, J. Freire, K. Sagonas, and P. Gomez. *Semantica.* MIT Press, 1996.

[47] J. Leite and L. M. Pereira. Iterated logic programming updates. In *International Conference on Logic Programming*, pages 265–278. MIT Press, 1998.

[48] B. Lewis and D. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, 1998.

[49] S. Liang and M. Kifer. A practical analysis of non-termination in large logic programs. *Theory and Practice of Logic Programming*, 13(4-5):705–719, 2013.

[50] S. Liang and M. Kifer. Terminyzer: An automatic non-termination analyzer for large logic programs. In *Practical Applications of Declarative Languages*, 2013.

[51] 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.

[52] 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.

[53] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[54] R. Marques. *Concurrent Tabling: Algorithms and Implementation*. PhD thesis, Universidade Nova de Lisboa, 2007.

[55] R. Marques and T. Swift. Concurrent and local evaluation of normal programs. In *International Conference on Logic Programming*, pages 206–222, 2008.

[56] R. Marques, T. Swift, and J. Cunha. Extending tabled logic programming with multi-threading: A systems perspective. In *CICLOPS*, pages 91–107, 2008.

[57] R. Marques, T. Swift, and J. Cunha. A simple and efficient implementation of concurrent local tabling. In *Practical Applications of Declarative Languages*, pages 264–278, 2010.

[58] A. McLeod. A remark on algorithm AS 183. *Applied Statistics*, 34:198–200, 1985.

[59] P. Moura. *Logtalk User Manual*. Available online from `http://logtalk.org`.

[60] I. Niemelä and P. Simons. SModels — An implementation of the stable model and well-founded semantics for normal LP. In *International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 420–429. Springer-Verlag, 1997.

[61] G. Pemmasani, H. Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online justification for tabled logic programs. In *Fuji International Symposium on Functional and Logic Programming*, pages 24–38, 2004.

[62] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM Principles of Database Systems*, pages 11–21, 1989.

[63] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings on the Conference on Automated Verification*, 1997.

[64] P. Rao, I. V. Ramakrishnan, K. Sagonas, T. Swift, and D. S. Warren. Efficient table access mechanisms for logic programs. *Journal of Logic Programming*, 38(1):31–54, Jan. 1999.

[65] F. Riguzzi and T. Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming*, 11(4-5):433–449, 2011.

[66] F. Riguzzi and T. Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming*, 13(2):279–302, 2013.

[67] F. Riguzzi and T. Swift. Terminating evaluation of logic programs with finite three-valued models. *ACM Transactions on Computational Logic*, 15(4), 2014.

[68] 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.

[69] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conference*. ACM, 1994.

[70] K. Sagonas, T. Swift, and D. S. Warren. An abstract machine for efficiently computing queries to well-founded models. *Journal of Logic Programming*, 45(1-3):1–41, 2000.

[71] K. Sagonas, T. Swift, and D. S. Warren. The limits of fixed-order computation. *Theoretical Computer Science*, 254(1-2):465–499, 2000.

[72] K. Sagonas and D. S. 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.

[73] D. Saha. *Incremental Evaluation of Tabled Logic Programs*. PhD thesis, SUNY Stony Brook, 2006.

[74] D. Saha and C. Ramakrishnan. Incemental and demand-driven points-to analysis using logic programming. In *ACM Principles and Practice of Declarative Programming*, 2005.

[75] H. Seki. On the power of Alexandrer templates. In *ACM Principles of Database Systems*, pages 150–159, 1989.

[76] K. Stenning and M. van Lambalgen. *Human Reasoning and Cognitive Science*. MIT Press, 2008.

[77] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[78] T. Swift. A new formulation of tabled resolution with delay. In *Recent Advances in Artifiial Intelligence*. Springer-Verlag, 1999.

[79] T. Swift. Tabling for non-monotonic programming. *Ann. Math. Artif. Intell.*, 25(3-4):201–240, 1999.

[80] T. Swift. Deduction in ontologies via answer set programming. In *International Conference on Logic Programming and Non-Monotonic Reasoning*, number 2923 in LNAI, pages 275–289, 2004.

[81] T. Swift. An engine for efficiently computing (sub-)models. In *International Conference on Logic Programming*, pages 514–518, 2009.

[82] T. Swift. Incremental tabling in support of knowledge representation and reasoning. *Theory and Practice of Logic Programming*, 14(4-5), 2014.

[83] T. Swift. Forest Logging: A trace-based analysis of large rule-based computations. *Semantic Web Journal*, 6, 2015.

[84] T. Swift and D. S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *JELIA*, 2010. Available at `http://www.cs.sunysb.edu/~tswift`.

[85] T. Swift and D. S. Warren. XSB: Extending the power of Prolog using tabling. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[86] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, 1986.

[87] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[88] A. van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

[89] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

[90] 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.

[91] H. Wan, B. Grosof, M. Kifer, P. Fodor, and S. Liang. Logic programming with defaults and argumentation theories. In *International Conference on Logic Programming*, pages 432–448, 2009.

[92] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

[93] D. S. Warren. Interning ground terms in XSB. In *Proceedings of CICLOPS 2013*, August 2013. In conjunction with ICLP'2013.

[94] B. A. Wichmann and I. D. Hill. Algorithm AS 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190, 1982.

[95] J. Wielemaker. *SWI Prolog version 5.6: Reference Manual*. University of Amsterdam, 2007.

[96] G. Yang, M. Kifer, and C. Zhao. *FLORA-2: User's Manual Version 0.94*, 2005. Avaliable via flora.sourceforge.net.

Index

\+/1, 193
\=/2, 200
\==/2, 200
; 236
^ /2, 192
!/0, 193
'∨'/2, 190
'∧'/2, 190
'/'/2, 189
'//'/2, 189
'«'/2, 190
'><'/2, 190
'»'/2, 190
**/2, 192
*/2, 189
+/2, 189
-/1, 190
-/2, 189
=../2, 216
=/2, 199
==/2, 199
?=/2, 200
@</2, 200
@= /2, 200
@=< /2, 200
@>/2, 200
@>= /2, 200
ISO, 207, 253, 272
[]/1 (consult), 47
$trace/0, 380
^ /2, 240
^=../2, 218
'C'/3, 431
abolish/1, 285
abolish_all_private_tables/0, 321
abolish_all_shared_tables/0, 321
abolish_all_tables/0, 320
abolish_module_tables/1, 322
abolish_nonincremental_tables/0, 320
abolish_table_pred/1, 317
abolish_table_pred/2, 318
abolish_table_subgoal/1, 319
abolish_table_subgoal/2, 320
abolish_table_subgoals/1, 318
abolish_table_subgoals/2, 319
abort/0, 446
abort/1, 446
absmember/2, 511
absmerge/3, 511
abstract_modes/2, 395
acos/1, 192
acyclic_term/1, 222
add_to_range_tree/3, 554
add_xsb_hook/1, 371
analyze_an_scc/2, 394
analyze_an_scc/3, 394
append/3, 509

arg/3, 214
arg0/3, 215
array_elt/3, 540
array_new/2, 540
array_update/3, 540
asin/1, 192
assert/1, 283
assert/3, 283
assert_in_db/3, 551
asserta/1, 282
assertz/1, 282
at_end_of_stream/0, 156
at_end_of_stream/1, 156
atan/1, 192
atan/2, 192
atan2/2, 192
atom/1, 206
atom_chars/2, 226
atom_codes/2, 223
atom_concat/3, 229
atom_length/2, 228
atomic/1, 207
bagof/3, 237
between/3, 197
bounded_call/3, 252
bounded_call/4, 46, 251
break/0, 271
c2p_float(), 462
c2p_functor(), 462
c2p_int(), 462
c2p_list(), 462
c2p_nil(), 462
c2p_string(), 462
call/1, 241
call/[2,10], 242
call_cleanup/2, 244
call_in_db/2, 552
call_tv/2, 242
callable/1, 209
catch/3, 446
cd/1, 185
ceiling/1, 191
char_code/2, 228
checkImpExps/1, 551
check_acyclic/3, 448
check_atom/3, 448
check_callable/3, 448
check_ground/3, 448
check_integern/3, 448
check_nonvar/3, 448
check_nonvar_list/3, 448
check_one_thread/3, 448
check_stream/3, 449
check_var/3, 449
check_variant/1, 202
check_variant/2, 202
clause/2, 286
clause_in_db/3, 552
close/1, 154
close/2, 153
closetail/1, 512
comma_append/3, 512
comma_length/2, 512
comma_member/2, 512
comma_memberchk/2, 512
comma_to_list/2, 512
compare/3, 201
compile/1, 47, 51
compile/2, 47, 50
compound/1, 207
concat_atom/2, 228
concat_atom/3, 229
console_write/1, 175
console_writeln/1, 175
consult, 36, 37
copy_term/2, 219
copy_term_nat/2, 219
cos/1, 192
crypto_hash/3, 556
current_atom/1, 265

current_functor/1, 264
current_index/2, 264
current_input/1, 253
current_module/1, 263
current_module/2, 264
current_op/3, 271
current_predicate/1, 263
current_prolog_flag/2, 254
current_timed_call/2, 250
cvt_canonical/2, 296
datime/1, 184
datime_setrand/0, 545
dcg/2, 432
debug/0, 377
debug_ctl/2, 377
debugging/0, 377
default_user_error_handler/1, 447
del_attr/2, 513
delete_all_range_trees/0, 555
delete_from_range_tree/2, 555
delete_from_range_tree/3, 554
delete_ith/4, 510
delete_range_tree/1, 555
delete_returns/2, 322
delete_trie/1, 369
dep_graph_scc_info/2, 411
div/2, 189
do_all/1, 197
do_all/2, 197
domain_error/4, 444
dynamic/1, 139, 289
e/0, 192
else/0, 64
elseif/1, 64
empty_db/1, 551
endif/0, 64
ensure_loaded/1, 48
ensure_loaded/2, 295
epoch_milliseconds/2, 184
epoch_seconds/1, 184
epsilon/0, 192
error_write/1, 175, 447
error_writeln/1, 175, 447
evaluation_error/3, 444
excess_vars/4, 240
existence_error/4, 445
expand_filename/2, 524
expand_filename_no_prepend/2, 524
expand_term/2, 429
explain_u_val/3, 130, 257, 407
explain_u_val/6, 257, 407
exponential/2, 545
fail/0, 193
fail_if/1, 193
false/0, 193
fd2ioport/2, 539
fd2iostream/2, 533
file_clone/3, 157
file_exists/1, 160
file_file_getbuf_atom/4, 181
file_getbuf_atom/3, 181
file_getbuf_list/3, 181
file_getbuf_list/4, 180
file_putbuf/4, 182
file_putbuf/5, 181
file_read_line_atom/1, 179
file_read_line_atom/2, 180
file_read_line_list/1, 179
file_read_line_list/2, 179
file_reopen/3, 157
file_truncate/3, 157
file_write_line/2, 180
file_write_line/3, 180
find_n/4, 241
findall/3, 238
findall/4, 238
float/1, 191
floor/1, 191
flush_all_output_streams/0, 158
flush_output/0, 155

flush_output/1, 155
fmt_read/3, 176
fmt_read/4, 176
fmt_write/2, 177
fmt_write/3, 177
fmt_write_string/3, 179
forall/2, 244
foreign_pred/0, 473
forest_log_overview/0, 394
format/2, 519
format/3, 519
functor/3, 210
gauss/2, 545
gc_atoms/0, 235
gc_dynamic/1, 286
gc_heap/0, 273
gc_tables/1, 322
gennum/1, 544
gensym/2, 544
get_attr/3, 513
get_backtrace_list/2, 449
get_byte/1, 163
get_byte/2, 162
get_call/3, 310
get_calls/3, 311
get_calls_for_table/2, 303
get_char/1, 161
get_char/2, 161
get_code/1, 162
get_code/2, 162
get_from_range_tree/6, 554
get_idg_info/1, 403
get_idg_info/2, 403
get_incr_sccs/1, 142
get_incr_sccs/2, 142
get_incr_sccs_with_deps/2, 142
get_incr_sccs_with_deps/3, 142
get_process_table/1, 529
get_rdg_info/1, 405
get_rdg_info/2, 405
get_residual/2, 305, 315, 405
get_residual_sccs/3, 130, 257, 406
get_residual_sccs/5, 130, 257, 406, 407
get_returns/2, 312
get_returns/3, 313
get_returns_and_tvs/3, 313
get_returns_for_call/2, 304
get_scan_pars/1, 548
get_scc_dumpfile/0, 258
get_scc_dumpfile/1, 308
get_scc_size/3, 394
get_sdg_info/1, 401
get_sdg_subgoal_info/1, 402
getenv/2, 185
getrand/1, 544
ground/1, 201
ground_and_acyclic/1, 201
ground_or_cyclic/1, 201
hilog_arg/3, 215
hilog_functor/3, 212
hilog_op/3, 271
hilog_symbol/1, 270
if/1, 64
include/1, 63
incr_assert/1, 139
incr_asserta/1, 139
incr_assertz/1, 139
incr_directly_depends/2, 141
incr_invalid_subgoals/1, 143
incr_invalidate_calls/1, 140
incr_is_invalid/1, 144
incr_retract/1, 139
incr_retractall/1, 140
incr_trans_depends/2, 142
incr_trie_intern/2, 141
incr_trie_uninternall/2, 141
index/2, 286
init_range_trees/1, 554

install_attribute_constraint_hook/4, 516
install_attribute_portray_hook/3, 515
install_verify_attribute_handler/4, 513
install_verify_attribute_handler/5, 513
instantiation_error/4, 445
integer/1, 207
intern_term/2, 220
invalidate_tables_for/2, 323
is/2, 188
is_acyclic/1, 222
is_attv/1, 209
is_attv(), 460
is_charlist/1, 208
is_charlist/2, 208
is_cyclic/1, 222
is_float(), 460
is_functor(), 460
is_incremental_subgoal/1, 141
is_int(), 460
is_list/1, 208
is_list(), 460
is_most_general_term/1, 209
is_nil(), 460
is_number_atom/1, 209
is_string(), 460
is_var(), 461
ith/3, 509
keysort/2, 204
length/2, 510
listing/0, 267
listing/1, 268
load_csv/2, 546
load_dsv/3, 546
load_dyn/1, 292
load_dyn/2, 293
load_dync/1, 294
load_dync/2, 294
load_forest_log/1, 387
load_in_db/2, 552
load_in_db/3, 552
local_datime/1, 184
log/1, 192
log10/1, 192
log_forest/2, 385
log_ith/3, 510
log_ith_bound/3, 510
log_ith_new/3, 510
log_ith_to_list/2, 510
max/2, 190
member/2, 509
member2/2, 512
memberchk/2, 509
merge/3, 511
message/1, 176
message_queue_create/2, 349
message_queue_destroy/1, 349
messageln/1, 176
min/2, 190
misc_error/1, 446
mod/2, 191
module_of_term/2, 557
module_property/2, 267
morph_dep_graph/3, 408
mutex_create/1, 352
mutex_destroy/1, 353
mutex_lock/1, 353
mutex_property/2, 355
mutex_trylock/1, 354
mutex_unlock/1, 354
mutex_unlock_all/0, 355
name/2, 225
new_trie/1, 367
nl/0, 166
nl/1, 166
nodebug/0, 377
nonvar/1, 206

nospy/1, 377
not/1, 193
not_exists/1, 194
notrace/0, 374
number/1, 207
number_chars/2, 227
number_codes/2, 224
number_digits/2, 228
numbervars/1, 175
numbervars/3, 174
numbervars/4, 174
once/1, 243
op/3, 84
open/3, 151
open/4, 152
otherwise/0, 193
p2c_arity(), 461
p2c_float(), 461
p2c_functor(), 461
p2c_int(), 461
p2c_string(), 461
p2p_arg(), 462
p2p_car(), 463
p2p_cdr(), 463
p2p_new(), 462
p2p_unify(), 464
parse_filename/4, 524
parsort/4, 205
path_sysop/2, 186, 187
path_sysop/3, 186–188
peek_byte/1, 164, 165
peek_char/1, 163
peek_char/2, 163
peek_code/1, 164
peek_code/2, 164
perm/2, 511
permission_error/4, 445
phrase/2, 429
phrase/3, 429
pi/0, 192
pid/1, 525
pipe_open/2, 531
predicate_property/2, 265
prepare/1, 544
pretty_print/1, 556
pretty_print/2, 556
print_backtrace/1, 449
print_dep_graph/1, 411
print_sdg_info/0, 411
print_sdg_subgoal_info/0, 411
print_sdg_subgoal_info, 441
private_foreign_pred/0, 473
process_control/2, 529
process_status/2, 529
profile_call/1, 541
profile_mode_call/1, 542
profile_mode_dump/0, 543
profile_mode_init/0, 543
profile_unindexed_calls/1, 543
prompt/2, 273
proper_hilog/1, 210
put_attr/3, 513
put_char/1, 165
put_char/2, 165
put_code/1, 166
put_code/2, 165
putenv/2, 185
random/1, 544
random/3, 544
randseq/3, 545
randset/3, 545
range_call/4, 555
range_retractall/4, 555
read/1, 167
read/2, 167
read_canonical/1, 167
read_canonical/2, 168
read_term/2, 168
read_term/3, 168
real/1, 207

reclaim_uninterned_rn/1, 369
reg_term(), 464
rem/2, 191
remove_xsb_hook/1, 372
repeat/0, 197
representation_error/3, 445
resource_error/3, 445
retract/1, 283
retractall/1, 284
retractall_in_db/3, 552
reverse/2, 511
round/1, 192
runtime_loader_flag/2, 472
same_length/2, 510
scan/2, 547
scan/3, 547
see/1, 158
seeing/1, 159
seen/0, 159
select/3, 511
set_arg/3, 518
set_dcg_style/1, 432
set_dcg_supported_table/1, 432
set_forest_logging_for_pred/2, 395
set_global_compiler_options/1, 52
set_input/1, 154
set_output/1, 154
set_prolog_flag/2, 262
set_scan_pars/1, 547
set_stream_position/2, 156
set_timer/1, 539
setarg/3, 518
setof/3, 236
setrand/1, 545
shell/1, 182
shell/2, 183
shell/5, 530
shell_to_list/3, 183
shell_to_list/4, 183
sign/1, 192
sin/1, 192
sleep/1, 185, 539
socket/2, 534
socket_accept/3, 535
socket_bind/3, 535
socket_close/2, 535
socket_connect/4, 535
socket_get0/3, 537
socket_listen/3, 535
socket_put/3, 537
socket_recv/3, 536
socket_select/6, 537
socket_select_destroy/1, 538
socket_send/3, 536
socket_set_option/3, 534
socket_set_select/4, 537
sort/2, 203
spawn_process/5, 525
spy/1, 377
sqrt/1, 192
statistics/0, 258, 273
statistics/1, 258, 277
statistics/2, 278
storage_commit/1, 298
storage_delete_fact/3, 297
storage_delete_fact_bt/2, 298
storage_delete_keypair/3, 297
storage_delete_keypair_bt/3, 298
storage_find_fact/2, 297
storage_find_keypair/3, 297
storage_insert_fact/3, 297
storage_insert_fact_bt/2, 298
storage_insert_keypair/4, 297
storage_insert_keypair_bt/4, 298
storage_reclaim_space/1, 298
str_match/5, 522
str_sub/3, 522
stream_property/2, 154
string_substitute/4, 232

structure/1, 208
sub_atom/5 , 230
subseq/3, 511
substring/4, 523
subsumes/2, 201
subsumes_chk/2, 201
subsumes_term/2, 202
sys_main_memory/1, 525
tab/1, 166
table/1, 88, 138, 139, 300
table_dump/2, 388, 396
table_dump/3, 388, 396
table_index/2, 324
table_once/1, 244
table_state/1, 307
table_state/4, 307
tan/1, 192
tell/1, 159
telling/1, 160
term_depth/2, 219
term_expansion/2, 425, 430
term_hash/3, 556
term_size/2, 220
term_to_atom/2, 235
term_to_atom/3, 233
term_to_codes/2, 235
term_to_codes/3 , 235
term_variables/2, 219, 519
tfindall/3, 238
thread_cancel/1, 345
thread_create/1, 343
thread_create/2, 343
thread_create/3, 341
thread_detach/1, 344
thread_disable_cancel/0, 347
thread_enable_cancel/0, 347
thread_exit/1, 344
thread_get_message/1, 351
thread_get_message/2, 350
thread_join/2, 343
thread_peek_message/1, 351
thread_peek_message/2, 351
thread_property/2, 347
thread_self/1, 344
thread_send_message/2, 350
thread_signal/2, 346
thread_sleep/1, 348
thread_yield/0, 347
three_valued_scc/1, 394
throw/1, 444
time/1, 281
timed_call/2, 247, 414
timed_call_cancel/0, 251
timed_call_modify/1, 250, 414
tmpfile_open/1, 158
tnot/1, 104, 194
told/0, 160
tphrase/1, 429
tphrase_set_string/1, 431
tphrase_set_string_auto_abolish/1, 431
tphrase_set_string_keeping_tables/1, 431
trace/0, 374
trace/2, 376
trie_bulk_delete/2, 365
trie_bulk_insert/2, 364
trie_bulk_unify/3, 366
trie_create/2, 360
trie_delete/2, 363
trie_drop/1, 364
trie_insert/2, 361
trie_intern/2, 367
trie_intern/5, 367
trie_interned/2, 368
trie_interned/4, 368
trie_property/2, 366
trie_truncate/1, 363
trie_unify/2, 362
trie_unintern2, 368

trie_unintern_nr/2, 368
trimcore/0, 273
true/0, 193
truncate/1, 192
truth_value/2, 243
type_error/4, 446
u_not/1, 195, 406, 407
unifiable/3, 200, 518
unify_with_occurs_check/2, 199
union_db/3, 552
unmark_uninterned_nr/2, 369
unnumbervars/3, 175
url_decode/2, 161
url_encode/2, 160
var/1, 205
variant/2, 202
variant_get_residual/2, 305, 405
warning/1, 175
weibull/3, 545
when/2, 517
with_mutex/2, 352
word/3, 428
write/1, 171
write/2, 171
write_canonical/1, 172
write_canonical/2, 173
write_prolog/1, 173
write_term/2, 169
write_term/3, 170
writeln/1, 173
writeln/2, 173
writeq/1, 172
writeq/2, 172
xor/2, 190
xsb_add_c_predicate(), 502
xsb_assert_hook/1, 372
xsb_backtrace/1, 449
xsb_ccall_thread_create(), 503
xsb_close_query(), 502
xsb_close(), 495
xsb_command_string(), 496
xsb_command(), 496
xsb_configuration/2, 262, 269
xsb_domain_error(), 477
xsb_error_get_backtrace/2, 447
xsb_error_get_goal/2, 447
xsb_error_get_goalatom/2, 447
xsb_error_get_message/2, 447
xsb_error_get_tid/2, 447
xsb_existence_error(), 477
xsb_exit_hook/1, 372
xsb_get_error_message(), 502, 503
xsb_get_error_type(), 503
xsb_get_last_answer_string_b(), 499
xsb_get_main_thread(), 503
xsb_init_string(), 494
xsb_init(), 495
xsb_instantiation_error(), 478
xsb_make_vars(), 506
xsb_misc_error(), 478
xsb_next_string(), 500, 501
xsb_next(), 501
xsb_permission_error(), 478
xsb_query_restore(), 471
xsb_query_save(), 471
xsb_query_string_string_b, 498
xsb_query_string_string(), 497
xsb_query_string(), 499
xsb_query(), 498
xsb_resource_error(), 479
xsb_retract_hook/1, 373
xsb_set_var_float(), 506
xsb_set_var_int(), 506
xsb_set_var_string(), 506
xsb_thread_context_to_id(), 503
xsb_thread_id_to_context(), 503
xsb_throw(), 479
xsb_type_error(), 479
xsb_var_float(), 507

`xsb_var_int()`, 506
`xsb_var_string()`, 507
`!/0`, 241, 423, 424, 431
`\+/1`, 104
`->/2`, 196
`do_all/1`, 197
64-bit architectures, 13, 452, 507

abort
  trace facility, 375
abstraction of terms
  answer, 122, 126
  size metric, 124
  subgoal, 122, 125
acc (compiler), 10
aggregate predicates
  prolog, 236
aliases
  message queues, 348
  mutexes, 352
  streams, 149
    user_error, 149
    user_input, 149
    user_message, 149
    user_output, 149
    user_warning, 149
  threads, 341
  tries, 360
answer abstraction, 116
answer substitution, 309
attributed variables, 4, 92, 144, 171, 173, 174, 208, 219, 220, 256, 385

backtrackable updates, 296–298
base file name, 22
bounded rationality, 122, 126
break level, 375, 446
byte code
  files
    compiler, 50

canonical format, 49, 172, 294
cc (compiler), 10
character code constancts, 74
character sets, 151, 257
Code authors
  Warren, David S., 551, 552
compilation
  conditional
    gpp, 54
    Prolog directives, 63
compiler, 50
  `cmplib`, 50
  directives, 62
  inlines, 71
  invoking, 50
  options, 52
  specialization, 60
compiler options
  `xpp_on`, 53
compiler options
  `mi_warn`, 59
  `modeinfer`, 59
  `optimize`, 53
  `spec_dump`, 59
  `spec_off`, 59
  `spec_repr`, 59
  `ti_dump`, 59
  `ti_long_names`, 59
  `unfold_off`, 59
configuration, 8
Constraint Handling Rules, 4
control, 193
CP1252, 151
cut, 193, 241, 423, 424, 431

debugger, 374
  ports, 374
declarations

`auto_table`, 58, 66, 91
`document_export/1`, 31
`document_import/1`, 31
`import/1`, 39
`index/2`, 68
`multifile/2`, 50
`suppl_table`, 58, 67, 91
`table as`, 94
definite clause grammars, 423
datalog mode, 428
list mode, 427
style, 433
directives
Compiler, 62
indexing, 68
modes, 64
tabling, 66
dynamic loading of files, 49

emulator
command line options, 40
errors with position, 257
exceptions, 434–450

Flora-2, 6, 221
floundering, 194, 195

garbage collection, 45, 255
atoms, 235
dynamic clauses, 286
heap, 273
tables, 322
gcc, 10
GPP, 50, 53
`gpp_include_dir`, 53
`gpp_options`, 54
`quit_on_error`, 58
`xpp_dump/N`, 55
`xpp_dump`, 55
`xpp_on/N`, 55
grammars
definite clause, 423

Incremental Dependency Graph (IDG), 132, 136, 141, 279, 319, 405, 412
displaying, 278
indexing, 281–283, 286, 387
composite, 288
directives, 68
dynamic predicates, 286
hash-based, 287
multiple-argument, 288
star, 4, 288
transformational, 69
trie-based, 288, 356
inlines
Compiler, 71
installation into shared directories, 8
Interface
InterProlog, 11
ODBC, 11
Oracle, 11
SModels, 11
interned terms, 220
InterProlog, 4
Interface, 11
interrupt instruction, 542
invoking the Compiler, 50
ISO
errors, 441
ISO Compatibility, 71

LATIN-1, 151
LD_LIBRARY_PATH, 472
LIBPATH, 472
load search path, 36
low-level tracing, 379

memory management, 45
message queues, 333
mode analysis

compiler options, 59
modes
directives, 64
multi-threading, 4, 326–355
mutexes
user defined, 351

negation
stable models, 112
stratified, 102
unstratified, 106
notational conventions, 7

occurs check, 198, 220, 257
ODBC Interface, 4, 11
options
command line arguments, 40
compiler, 52
Oracle Interface, 11

packages, 38
`bootstrap_userpackage/3`, 38
`package_configuration/2`, 39
`unload_package/1`, 39
permanent variables, 449
predicate indicator, 252, 253
preprocessing, 50
Prolog flags
`max_table_subgoal_size`, 126
Prolog flags, 253
`atom_garbage_collection`, 235, 255
`backtrace_on_error`, 255, 449
`bounded`, 254
`character_set`, 257
`clause_garbage_collection`, 255
`dcg_style`, 255
`debug`, 254
`dialect`, 254
`double_quotes`, 254
`errors_with_position`, 257
`exception_action`, 188, 257
`exception_pre_action`, 258, 308, 441
`goal`, 255
`heap_garbage_collection`, 255, 273
`heap_margin`, 255
`integer_rounding_function`, 254
`max_answer_list_action`, 220
`max_answer_list_depth`, 220
`max_answer_term_action`, 220
`max_answer_term_depth`, 220
`max_answers_for_subgoal_action`, 259
`max_answers_for_subgoal`, 259
`max_incomplete_subgoals_action`, 258
`max_incomplete_subgoals`, 258
`max_integer`, 254
`max_memory`, 46, 260
`max_queue_size`, 261
`max_scc_subgoals_action`, 258
`max_scc_subgoals`, 258
`max_tab_usage`, 258
`max_table_answer_action`, 129
`max_table_answer_size_action`, 259
`max_table_answer_size`, 129, 259
`max_table_subgoal_action`, 126, 220
`max_table_subgoal_depth`, 220
`max_table_subgoal_size_action`, 258
`max_table_subgoal_size`, 258
`max_threads`, 261, 328
`min_integer`, 254
`shared_predicates`, 261
`table_gc_action`, 255, 315, 317–319
`thread_complsize`, 261, 342

`thread_detached`, 261, 342
`thread_glsize`, 261, 342
`thread_pdlsize`, 261, 342
`thread_tcpsize`, 261, 342
`tracing`, 256
`unify_with_occurs_check`, 198, 220, 257
`unknown`, 29, 254
`version_data`, 254
`warning_action`, 175, 256
`write_attributes`, 171, 256
`write_depth`, 256
Prolog-commons, 253
Prologs
hProlog, 512
SWI, 281, 512
YAP, 281

Random Variables, 544
Exponential, 545
Normal, 545
Weibull, 545
residual dependency graph, 130, 141, 405, 412
residual program, 111, 305, 405, 412
restraint
answer count, 129
radial, 127, 407
ReturnHandle, 299

scheduling strategy, 11
sets, bags, 236
shared_predicates, 44, 262, 329
Silk, 221
skeleton, 299
SModels Interface, 11
source file designator, 22
specialization
Compiler, 60
compiler options, 59
stable models, 110
stacks
default sizes, 40
expanding, 40
standard predicates, 39, 53, 60
state of the system, 252
streams, 148
STDDBG, 150
STDERR, 150, 175, 255, 447
STDFDBK, 150, 175
STDIN, 150
STDMSG, 150, 176
STDOUT, 150, 175
STDWARN, 150, 175
system, 150, 158
strongly connected components (SCCs), 96, 308, 384, 439, 441
substitution factor, 309
syntax
atoms, 75
escaped characters, 75
floats, 75
integers, 73
binary, 73
hexidecimal, 73
octal, 73
system, state of, 252

table_index, 324
tabled subgoals
complete, 104
incomplete, 104, 279
TableEntryHandle, 299
tabling
abolishes, 317, 320–322
incremental, 319, 320
multi-threading, 317, 319–321
transitive vs. single, 317, 318
and exceptions, 258, 308, 439
answer completion, 109

answer subsumption, 91, 113–115, 117, 144, 265, 300
automatic, 58
call subsumption, 40, 44, 91, 92, 94, 144, 265, 273, 300, 302, 309, 310
interaction with meta-logical predicates, 100
call variance, 91, 92, 144, 265, 273, 300, 302, 309, 310
compiler options, 58
complete evaluation, 103, 389
conditional answers, 106, 315, 317
consumer, 88
cuts, 98
declarations, 138
directives, 66, 300
dynamic predicates, 300
early completion of subgoals, 104, 384, 389
incremental, 130, 138, 144, 265, 273, 300, 317, 361
invalid subgoals, 132
interned terms, 94
multi-threaded, 316, 317
negation, 102
opaque, 138, 300
private, 300
producer, generator, 88
scheduling strategies, 95
shared, 44, 300
similarity measures, 91
strategy selection, 300
supplemental, 58
table abolishing, 314
table inspection, 302, 309
Tck/Tk, 11
term depth
definition, 219
term indicator, 252, 253
term size, 258
definition, 220
termination, 67, 90, 91
answer count restraint, 122
answer subsumption, 114
radial restraint, 122, 258, 406, 407
subgoal abstraction, 122, 258
terms
comparison of, 198
cyclic, 198, 201, 219, 220, 257, 361, 365
unification of, 198
thread
thread status, 332
valid, 331
trace
logging, 376
options, 375
tracing, 374–387
low-level, 379
Prolog Program Controls, 379
Prolog Programs, 374
transaction logic, 296
tries
and incremental tabling, 135
asserted, 287
indexing, 288
depth limit, 221
interned, 356–370
tripwires, 258, 413
max_answers_for_subgoal, 129, 422
max_incomplete_subgoals, 421
max_memory, 414
max_scc_subgoals, 421
max_table_answer_size, 125, 128, 421
max_table_subgoal_size, 421
timed call, 246, 414

Unicode
    UTF-8, 75, 151, 161, 166, 223, 226, 235
unification factoring
    compiler options, 59
Unknown predicate handling, 254

VarString, 504
view consistency, 133, 315

well-founded semantics, 109

XASP, 6, 11
xsbdoc, 6, 31
xsbrc.P initialization file, 36

Index of Standard XSB Predicates

\+/1, 193
\=/2, 200
\==/2, 200
^ /2, 192
!/0, 193
'∨'/2, 190
'∧'/2, 190
'/'/2, 189
'//'/2, 189
'«'/2, 190
'><'/2, 190
'»'/2, 190
**/2, 192
*/2, 189
+/2, 189
-/1, 190
-/2, 189
=../2, 216
=/2, 199
==/2, 199
?=/2, 200
@</2, 200
@= /2, 200
@=< /2, 200
@>/2, 200
@>= /2, 200
ISO, 207, 253, 272
[]/1 (consult), 47
$trace/0, 380
^ /2, 240
^=../2, 218
'C'/3, 431
abolish/1, 285
abolish_all_private_tables/0, 321
abolish_all_shared_tables/0, 321
abolish_all_tables/0, 320
abolish_module_tables/1, 322
abolish_nonincremental_tables/0, 320
abolish_table_pred/1, 317
abolish_table_pred/2, 318
abolish_table_subgoal/1, 319
abolish_table_subgoal/2, 320
abolish_table_subgoals/1, 318
abolish_table_subgoals/2, 319
abort/0, 446
abort/1, 446
acos/1, 192
acyclic_term/1, 222
arg/3, 214
arg0/3, 215
asin/1, 192
assert/1, 283
assert/3, 283
asserta/1, 282
assertz/1, 282
at_end_of_stream/0, 156
at_end_of_stream/1, 156

atan/1, 192
atan/2, 192
atan2/2, 192
atom/1, 206
atom_chars/2, 226
atom_codes/2, 223
atom_concat/3, 229
atom_length/2, 228
atomic/1, 207
bagof/3, 237
break/0, 271
call/1, 241
call/[2,10], 242
call_cleanup/2, 244
call_tv/2, 242
callable/1, 209
catch/3, 446
cd/1, 185
ceiling/1, 191
char_code/2, 228
clause/2, 286
close/1, 154
close/2, 153
compare/3, 201
compile/1, 47, 51
compile/2, 47, 50
compound/1, 207
consult, 36, 37
copy_term/2, 219
cos/1, 192
current_atom/1, 265
current_functor/1, 264
current_index/2, 264
current_input/1, 253
current_module/1, 263
current_module/2, 264
current_op/3, 271
current_predicate/1, 263
current_prolog_flag/2, 254
current_timed_call/2, 250
debug/0, 377
debug_ctl/2, 377
debugging/0, 377
default_user_error_handler/1, 447
delete_returns/2, 322
div/2, 189
do_all/1, 197
do_all/2, 197
dynamic/1, 139, 289
e/0, 192
else/0, 64
elseif/1, 64
endif/0, 64
ensure_loaded/1, 48
ensure_loaded/2, 295
epsilon/0, 192
expand_term/2, 429
fail/0, 193
fail_if/1, 193
false/0, 193
file_clone/3, 157
file_exists/1, 160
file_read_line_atom/1, 179
file_read_line_atom/2, 180
file_read_line_list/1, 179
file_read_line_list/2, 179
file_reopen/3, 157
findall/3, 238
findall/4, 238
float/1, 191
floor/1, 191
flush_output/0, 155
flush_output/1, 155
fmt_read/3, 176
fmt_read/4, 176
fmt_write/2, 177
fmt_write/3, 177
fmt_write_string/3, 179
forall/2, 244
foreign_pred/0, 473

functor/3, 210
gc_atoms/0, 235
gc_dynamic/1, 286
gc_heap/0, 273
gc_tables/1, 322
get_byte/1, 163
get_byte/2, 162
get_call/3, 310
get_calls/3, 311
get_calls_for_table/2, 303
get_char/1, 161
get_char/2, 161
get_code/1, 162
get_code/2, 162
get_residual/2, 305
get_returns/2, 312
get_returns/3, 313
get_returns_and_tvs/3, 313
get_returns_for_call/2, 304
ground/1, 201
ground_and_acyclic/1, 201
ground_or_cyclic/1, 201
hilog_arg/3, 215
hilog_functor/3, 212
hilog_op/3, 271
hilog_symbol/1, 270
if/1, 64
include/1, 63
index/2, 286
integer/1, 207
invalidate_tables_for/2, 323
is/2, 188
is_acyclic/1, 222
is_attv/1, 209
is_charlist/1, 208
is_charlist/2, 208
is_cyclic/1, 222
is_list/1, 208
is_most_general_term/1, 209
is_number_atom/1, 209
keysort/2, 204
listing/0, 267
listing/1, 268
load_dyn/1, 292
load_dyn/2, 293
load_dync/1, 294
load_dync/2, 294
log/1, 192
log10/1, 192
max/2, 190
message_queue_create/2, 349
message_queue_destroy/1, 349
min/2, 190
mod/2, 191
module_property/2, 267
mutex_create/1, 352
mutex_destroy/1, 353
mutex_lock/1, 353
mutex_property/2, 355
mutex_trylock/1, 354
mutex_unlock/1, 354
mutex_unlock_all/0, 355
name/2, 225
nl/0, 166
nl/1, 166
nodebug/0, 377
nonvar/1, 206
nospy/1, 377
not/1, 193
not_exists/1, 194
notrace/0, 374
number/1, 207
number_chars/2, 227
number_codes/2, 224
number_digits/2, 228
once/1, 243
op/3, 84
open/3, 151
open/4, 152
otherwise/0, 193

path_sysop/2, 186, 187
path_sysop/3, 186–188
peek_byte/1, 164, 165
peek_char/1, 163
peek_char/2, 163
peek_code/1, 164
peek_code/2, 164
phrase/2, 429
phrase/3, 429
pi/0, 192
predicate_property/2, 265
private_foreign_pred/0, 473
prompt/2, 273
proper_hilog/1, 210
put_char/1, 165
put_char/2, 165
put_code/1, 166
put_code/2, 165
read/1, 167
read/2, 167
read_canonical/1, 167
read_canonical/2, 168
read_term/2, 168
read_term/3, 168
real/1, 207
rem/2, 191
repeat/0, 197
retract/1, 283
retractall/1, 284
round/1, 192
see/1, 158
seeing/1, 159
seen/0, 159
set_dcg_style/1, 432
set_global_compiler_options/1, 52
set_input/1, 154
set_output/1, 154
set_prolog_flag/2, 262
set_stream_position/2, 156
set_timer/1, 539
setof/3, 236
shell/1, 182
shell/2, 183
shell_to_list/3, 183
shell_to_list/4, 183
sign/1, 192
sin/1, 192
sleep/1, 539
sort/2, 203
spy/1, 377
sqrt/1, 192
statistics/0, 273
statistics/1, 277
statistics/2, 278
storage_commit/1, 298
storage_delete_fact/3, 297
storage_delete_fact_bt/2, 298
storage_delete_keypair/3, 297
storage_delete_keypair_bt/3, 298
storage_find_fact/2, 297
storage_find_keypair/3, 297
storage_insert_fact/3, 297
storage_insert_fact_bt/2, 298
storage_insert_keypair/4, 297
storage_insert_keypair_bt/4, 298
storage_reclaim_space/1, 298
stream_property/2, 154
structure/1, 208
sub_atom/5 , 230
subsumes_term/2, 202
tab/1, 166
table/1, 88, 138, 139, 300
table_index/2, 324
table_once/1, 244
table_state/1, 307
table_state/4, 307
tan/1, 192
tell/1, 159
telling/1, 160
term_depth/2, 219

term_expansion/2, 425, 430
term_size/2, 220
term_variables/2, 219
tfindall/3, 238
thread_cancel/1, 345
thread_create/1, 343
thread_create/2, 343
thread_create/3, 341
thread_detach/1, 344
thread_exit/1, 344
thread_get_message/1, 351
thread_get_message/2, 350
thread_join/2, 343
thread_peek_message/1, 351
thread_peek_message/2, 351
thread_property/2, 347
thread_self/1, 344
thread_send_message/2, 350
thread_signal/2, 346
thread_sleep/1, 348
thread_yield/0, 347
throw/1, 444
time/1, 281
timed_call/2, 247
timed_call_cancel/0, 251
timed_call_modify/1, 250
tmpfile_open/1, 158
tnot/1, 194
told/0, 160
tphrase/1, 429
tphrase_set_string/1, 431
trace/0, 374
trace/2, 376
true/0, 193
truncate/1, 192
unify_with_occurs_check/2, 199
url_decode/2, 161
url_encode/2, 160
var/1, 205
variant_get_residual/2, 305
with_mutex/2, 352
word/3, 428
write/1, 171
write/2, 171
write_canonical/1, 172
write_canonical/2, 173
write_prolog/1, 173
write_term/2, 169
write_term/3, 170
writeln/1, 173
writeln/2, 173
writeq/1, 172
writeq/2, 172
xor/2, 190
xsb_configuration/2, 269